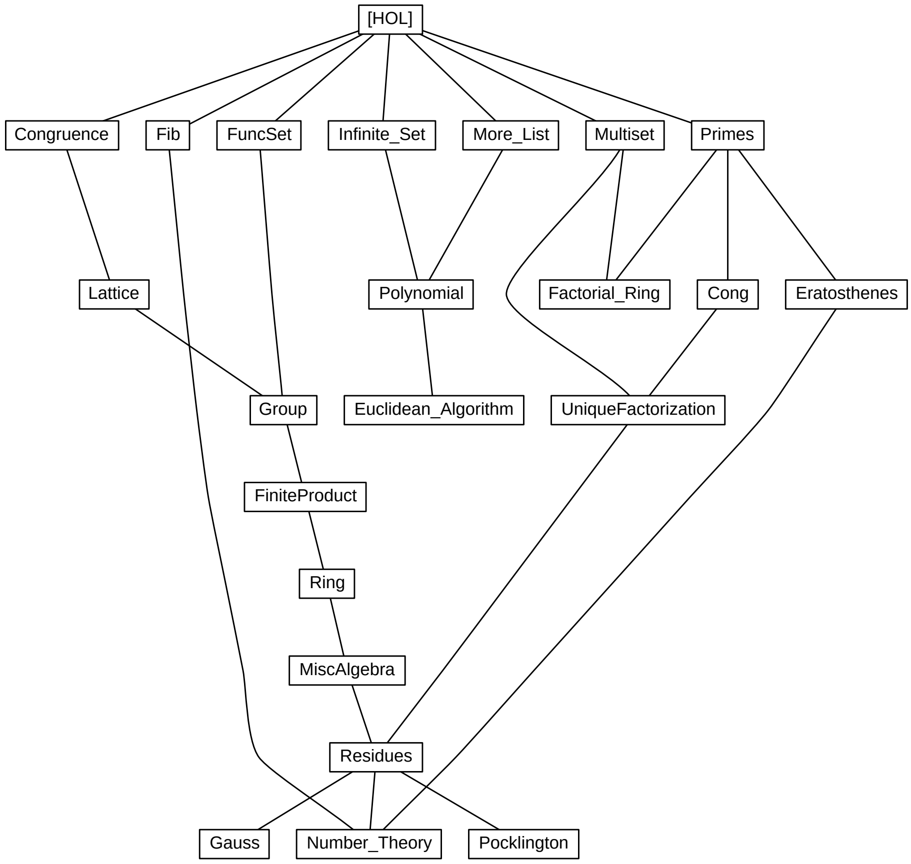
Various results of number theory

April 17, 2016

Contents


[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*
  ⟨*proof*⟩

lemma *prime-gt-0-nat*: *prime p* ⟹ *p* > *0*
  ⟨*proof*⟩

lemma *prime-ge-1-nat*: *prime p* ⟹ *p* >= *1*
  ⟨*proof*⟩

lemma *prime-gt-1-nat*: *prime p* ⟹ *p* > *1*
  ⟨*proof*⟩

lemma *prime-ge-Suc-0-nat*: *prime p* ⟹ *p* >= *Suc 0*
  ⟨*proof*⟩

lemma *prime-gt-Suc-0-nat*: *prime p* ⟹ *p* > *Suc 0*
  ⟨*proof*⟩

lemma *prime-ge-2-nat*: *prime p* ⟹ *p* >= *2*
  ⟨*proof*⟩

lemma *prime-imp-coprime-nat*: *prime p* ⟹ ¬ *p dvd n* ⟹ *coprime p n*
  ⟨*proof*⟩

lemma *prime-int-altdef*:
  prime p = (*1* < *p* ∧ (∀ *m*::*int*. *m* ≥ *0* ⟶ *m dvd p* ⟶
    m = *1* ∨ *m* = *p*))
  ⟨*proof*⟩

lemma *prime-imp-coprime-int*:
  fixes *n*::*int* **shows** *prime p* ⟹ ¬ *p dvd n* ⟹ *coprime p n*
  ⟨*proof*⟩

lemma *prime-dvd-mult-nat*: *prime p* ⟹ *p dvd m* ∗ *n* ⟹ *p dvd m* ∨ *p dvd n*
  ⟨*proof*⟩

lemma *prime-dvd-mult-int*:
fixes *n::int* **shows** *prime p* $\Longrightarrow$ *p dvd m* $*$ *n* $\Longrightarrow$ *p dvd m* $\vee$ *p dvd n*
⟨*proof*⟩

lemma *prime-dvd-mult-eq-nat* [*simp*]: *prime p* $\Longrightarrow$
p dvd m $*$ *n* = (*p dvd m* $\vee$ *p dvd n*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *prime-dvd-power-nat*: *prime p* $\Longrightarrow$ *p dvd x^n* $\Longrightarrow$ *p dvd x*
⟨*proof*⟩

lemma *prime-dvd-power-int*:
fixes *x::int* **shows** *prime p* $\Longrightarrow$ *p dvd x^n* $\Longrightarrow$ *p dvd x*
⟨*proof*⟩

lemma *prime-dvd-power-nat-iff*: *prime p* $\Longrightarrow$ *n* > *0* $\Longrightarrow$
p dvd x^n $\longleftrightarrow$ *p dvd x*
⟨*proof*⟩

lemma *prime-dvd-power-int-iff*:
fixes *x::int*
shows *prime p* $\Longrightarrow$ *n* > *0* $\Longrightarrow$ *p dvd x^n* $\longleftrightarrow$ *p dvd x*
⟨*proof*⟩

1.1.1 Make prime naively executable

lemma *zero-not-prime-nat* [*simp*]: ~*prime* (*0::nat*)
⟨*proof*⟩

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

lemma *Suc-0-not-prime-nat* [*simp*]: ~*prime* (*Suc 0*)
⟨*proof*⟩

lemma *prime-nat-code* [*code*]:
prime p $\longleftrightarrow$ *p* > *1* $\wedge$ ($\forall$ *n* $\in$ {*1*<..<*p*}. ~ *n dvd p*)
⟨*proof*⟩

lemma *prime-nat-simp*:
$\quad prime\ p \longleftrightarrow p > 1 \wedge (\forall n \in set\ [2..<p].\ \neg\ n\ dvd\ p)$
$\langle proof \rangle$

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

lemma *two-is-prime-nat* [*simp*]: $prime\ (2::nat)$
$\langle proof \rangle$

A bit of regression testing:

lemma $prime(97::nat)$ $\langle proof \rangle$
lemma $prime(997::nat)$ $\langle proof \rangle$

lemma *prime-imp-power-coprime-nat*: $prime\ p \Longrightarrow \sim p\ dvd\ a \Longrightarrow coprime\ a\ (p\hat{}m)$
$\langle proof \rangle$

lemma *prime-imp-power-coprime-int*:
fixes $a::int$ **shows** $prime\ p \Longrightarrow \sim p\ dvd\ a \Longrightarrow coprime\ a\ (p\hat{}m)$
$\langle proof \rangle$

lemma *primes-coprime-nat*: $prime\ p \Longrightarrow prime\ q \Longrightarrow p \neq q \Longrightarrow coprime\ p\ q$
$\langle proof \rangle$

lemma *primes-imp-powers-coprime-nat*:
$\quad prime\ p \Longrightarrow prime\ q \Longrightarrow p \sim= q \Longrightarrow coprime\ (p\hat{}m)\ (q\hat{}n)$
$\langle proof \rangle$

lemma *prime-factor-nat*:
$n \neq (1::nat) \Longrightarrow \exists p.\ prime\ p \wedge p\ dvd\ n$
$\langle proof \rangle$

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*: $p\hat{}n\ dvd\ a * b$
shows $p\hat{}n\ dvd\ a \vee p\hat{}n\ dvd\ b$
$\langle proof \rangle$

1.2 Infinitely many primes

lemma *next-prime-bound*: $\exists p.\ prime\ p \wedge n < p \wedge p <= fact\ n + 1$
$\langle proof \rangle$

lemma *bigger-prime*: $\exists p.\ prime\ p \wedge p > (n::nat)$
$\langle proof \rangle$

lemma *primes-infinite*: $\neg\ (finite\ \{(p::nat).\ prime\ p\})$
$\langle proof \rangle$

1.3 Powers of Primes

Versions for type nat only

lemma *prime-product*:
  fixes p::*nat*
  assumes *prime* $(p * q)$
  shows $p = 1 \vee q = 1$
⟨*proof*⟩

lemma *prime-exp*:
  fixes p::*nat*
  shows *prime* $(p\hat{\ }n) \longleftrightarrow$ *prime* $p \wedge n = 1$
⟨*proof*⟩

lemma *prime-power-mult*:
  fixes p::*nat*
  assumes p: *prime* p **and** xy: $x * y = p \hat{\ } k$
  shows $\exists\, i\, j.\ x = p \hat{\ }i \wedge y = p\hat{\ } j$
⟨*proof*⟩

lemma *prime-power-exp*:
  fixes p::*nat*
  assumes p: *prime* p **and** n: $n \neq 0$
    and xn: $x\hat{\ }n = p\hat{\ }k$ **shows** $\exists\, i.\ x = p\hat{\ }i$
  ⟨*proof*⟩

lemma *divides-primepow*:
  fixes p::*nat*
  assumes p: *prime* p
  shows d *dvd* $p\hat{\ }k \longleftrightarrow (\exists\ i.\ i \leq k \wedge d = p\ \hat{\ }i)$
⟨*proof*⟩

1.4 Chinese Remainder Theorem Variants

lemma *bezout-gcd-nat*:
  fixes a::*nat* **shows** $\exists\, x\, y.\ a * x - b * y = gcd\ a\ b \vee b * x - a * y = gcd\ a\ b$
  ⟨*proof*⟩

lemma *gcd-bezout-sum-nat*:
  fixes a::*nat*
  assumes $a * x + b * y = d$
  shows *gcd* a b *dvd* d
⟨*proof*⟩

A binary form of the Chinese Remainder Theorem.

lemma *chinese-remainder*:
  fixes a::*nat* **assumes** ab: *coprime* a b **and** a: $a \neq 0$ **and** b: $b \neq 0$
  shows $\exists\, x\ q1\ q2.\ x = u + q1 * a \wedge x = v + q2 * b$
⟨*proof*⟩

Primality

lemma *coprime-bezout-strong*:
  fixes *a*::*nat* **assumes** *coprime a b* $b \neq 1$
  shows $\exists x\ y.\ a * x = b * y + 1$
⟨*proof*⟩

lemma *bezout-prime*:
  assumes *p*: *prime p* **and** *pa*: ¬ *p dvd a*
  shows $\exists x\ y.\ a{*}x = Suc\ (p{*}y)$
⟨*proof*⟩

end

2 Congruence

theory *Cong*
imports *Primes*
begin

2.1 Turn off *One-nat-def*

lemma *power-eq-one-eq-nat* [*simp*]: $((x{::}nat)\hat{\ }m = 1) = (m = 0 \mid x = 1)$
  ⟨*proof*⟩

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

2.2 Main definitions

class *cong* =
  fixes *cong* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$ ((*1*[- = -] '(()*mod* -')))
begin

abbreviation *notcong* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$ ((*1*[- ≠ -] '(()*mod* -')))
  where *notcong x y m* ≡ ¬ *cong x y m*

end

instantiation *nat* :: *cong*
begin

definition *cong-nat* :: $nat \Rightarrow nat \Rightarrow nat \Rightarrow bool$
  where *cong-nat x y m* = ((*x mod m*) = (*y mod m*))

instance ⟨*proof*⟩

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 ⟨*proof*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *cong-0-int* [*simp*, *presburger*]: ([(*a*::*int*) = *b*] (*mod 0*)) = (*a* = *b*)
⟨*proof*⟩

lemma *cong-1-nat* [*simp*, *presburger*]: [(*a*::*nat*) = *b*] (*mod 1*)
⟨*proof*⟩

lemma *cong-Suc-0-nat* [*simp*, *presburger*]: [(*a*::*nat*) = *b*] (*mod Suc 0*)
⟨*proof*⟩

lemma *cong-1-int* [*simp*, *presburger*]: [(*a*::*int*) = *b*] (*mod 1*)
⟨*proof*⟩

lemma *cong-refl-nat* [*simp*]: [(*k*::*nat*) = *k*] (*mod m*)
⟨*proof*⟩

lemma *cong-refl-int* [*simp*]: $[(k::int) = k]$ $(mod\ m)$
⟨*proof*⟩

lemma *cong-sym-nat*: $[(a::nat) = b]$ $(mod\ m) \Longrightarrow [b = a]$ $(mod\ m)$
⟨*proof*⟩

lemma *cong-sym-int*: $[(a::int) = b]$ $(mod\ m) \Longrightarrow [b = a]$ $(mod\ m)$
⟨*proof*⟩

lemma *cong-sym-eq-nat*: $[(a::nat) = b]$ $(mod\ m) = [b = a]$ $(mod\ m)$
⟨*proof*⟩

lemma *cong-sym-eq-int*: $[(a::int) = b]$ $(mod\ m) = [b = a]$ $(mod\ m)$
⟨*proof*⟩

lemma *cong-trans-nat* [*trans*]:
$[(a::nat) = b]$ $(mod\ m) \Longrightarrow [b = c]$ $(mod\ m) \Longrightarrow [a = c]$ $(mod\ m)$
⟨*proof*⟩

lemma *cong-trans-int* [*trans*]:
$[(a::int) = b]$ $(mod\ m) \Longrightarrow [b = c]$ $(mod\ m) \Longrightarrow [a = c]$ $(mod\ m)$
⟨*proof*⟩

lemma *cong-add-nat*:
$[(a::nat) = b]$ $(mod\ m) \Longrightarrow [c = d]$ $(mod\ m) \Longrightarrow [a + c = b + d]$ $(mod\ m)$
⟨*proof*⟩

lemma *cong-add-int*:
$[(a::int) = b]$ $(mod\ m) \Longrightarrow [c = d]$ $(mod\ m) \Longrightarrow [a + c = b + d]$ $(mod\ m)$
⟨*proof*⟩

lemma *cong-diff-int*:
$[(a::int) = b]$ $(mod\ m) \Longrightarrow [c = d]$ $(mod\ m) \Longrightarrow [a - c = b - d]$ $(mod\ m)$
⟨*proof*⟩

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

lemma *cong-diff-nat*:
assumes$[a = b]$ $(mod\ m)$ $[c = d]$ $(mod\ m)$ $(a::nat) >= c$ $b >= d$
shows $[a - c = b - d]$ $(mod\ m)$
⟨*proof*⟩

lemma *cong-mult-nat*:
$[(a::nat) = b]$ $(mod\ m) \Longrightarrow [c = d]$ $(mod\ m) \Longrightarrow [a * c = b * d]$ $(mod\ m)$
⟨*proof*⟩

lemma *cong-mult-int*:
$[(a::int) = b]\ (mod\ m) \Longrightarrow [c = d]\ (mod\ m) \Longrightarrow [a * c = b * d]\ (mod\ m)$
⟨*proof*⟩

lemma *cong-exp-nat*: $[(x::nat) = y]\ (mod\ n) \Longrightarrow [x\hat{}k = y\hat{}k]\ (mod\ n)$
⟨*proof*⟩

lemma *cong-exp-int*: $[(x::int) = y]\ (mod\ n) \Longrightarrow [x\hat{}k = y\hat{}k]\ (mod\ n)$
⟨*proof*⟩

lemma *cong-setsum-nat* [*rule-format*]:
$(\forall x\in A.\ [((f\ x)::nat) = g\ x]\ (mod\ m)) \longrightarrow$
$[(\sum x\in A.\ f\ x) = (\sum x\in A.\ g\ x)]\ (mod\ m)$
⟨*proof*⟩

lemma *cong-setsum-int* [*rule-format*]:
$(\forall x\in A.\ [((f\ x)::int) = g\ x]\ (mod\ m)) \longrightarrow$
$[(\sum x\in A.\ f\ x) = (\sum x\in A.\ g\ x)]\ (mod\ m)$
⟨*proof*⟩

lemma *cong-setprod-nat* [*rule-format*]:
$(\forall x\in A.\ [((f\ x)::nat) = g\ x]\ (mod\ m)) \longrightarrow$
$[(\prod x\in A.\ f\ x) = (\prod x\in A.\ g\ x)]\ (mod\ m)$
⟨*proof*⟩

lemma *cong-setprod-int* [*rule-format*]:
$(\forall x\in A.\ [((f\ x)::int) = g\ x]\ (mod\ m)) \longrightarrow$
$[(\prod x\in A.\ f\ x) = (\prod x\in A.\ g\ x)]\ (mod\ m)$
⟨*proof*⟩

lemma *cong-scalar-nat*: $[(a::nat)= b]\ (mod\ m) \Longrightarrow [a * k = b * k]\ (mod\ m)$
⟨*proof*⟩

lemma *cong-scalar-int*: $[(a::int)= b]\ (mod\ m) \Longrightarrow [a * k = b * k]\ (mod\ m)$
⟨*proof*⟩

lemma *cong-scalar2-nat*: $[(a::nat)= b]\ (mod\ m) \Longrightarrow [k * a = k * b]\ (mod\ m)$
⟨*proof*⟩

lemma *cong-scalar2-int*: $[(a::int)= b]\ (mod\ m) \Longrightarrow [k * a = k * b]\ (mod\ m)$
⟨*proof*⟩

lemma *cong-mult-self-nat*: $[(a::nat) * m = 0]\ (mod\ m)$
⟨*proof*⟩

lemma *cong-mult-self-int*: $[(a::int) * m = 0]\ (mod\ m)$
⟨*proof*⟩

lemma *cong-eq-diff-cong-0-int*: $[(a::int) = b]$ $(mod\ m) = [a - b = 0]$ $(mod\ m)$
⟨*proof*⟩

lemma *cong-eq-diff-cong-0-aux-int*: $a >= b \Longrightarrow$
$[(a::int) = b]$ $(mod\ m) = [tsub\ a\ b = 0]$ $(mod\ m)$
⟨*proof*⟩

lemma *cong-eq-diff-cong-0-nat*:
assumes $(a::nat) >= b$
shows $[a = b]$ $(mod\ m) = [a - b = 0]$ $(mod\ m)$
⟨*proof*⟩

lemma *cong-diff-cong-0′-nat*:
$[(x::nat) = y]$ $(mod\ n) \longleftrightarrow$
$(if\ x <= y\ then\ [y - x = 0]\ (mod\ n)\ else\ [x - y = 0]\ (mod\ n))$
⟨*proof*⟩

lemma *cong-altdef-nat*: $(a::nat) >= b \Longrightarrow [a = b]$ $(mod\ m) = (m\ dvd\ (a - b))$
⟨*proof*⟩

lemma *cong-altdef-int*: $[(a::int) = b]$ $(mod\ m) = (m\ dvd\ (a - b))$
⟨*proof*⟩

lemma *cong-abs-int*: $[(x::int) = y]$ $(mod\ abs\ m) = [x = y]$ $(mod\ m)$
⟨*proof*⟩

lemma *cong-square-int*:
fixes $a::int$
shows $[\![\ prime\ p;\ 0 < a;\ [a * a = 1]\ (mod\ p)\ ]\!]$
$\Longrightarrow [a = 1]$ $(mod\ p) \lor [a = - 1]$ $(mod\ p)$
⟨*proof*⟩

lemma *cong-mult-rcancel-int*:
$coprime\ k\ (m::int) \Longrightarrow [a * k = b * k]$ $(mod\ m) = [a = b]$ $(mod\ m)$
⟨*proof*⟩

lemma *cong-mult-rcancel-nat*:
$coprime\ k\ (m::nat) \Longrightarrow [a * k = b * k]$ $(mod\ m) = [a = b]$ $(mod\ m)$
⟨*proof*⟩

lemma *cong-mult-lcancel-nat*:
$coprime\ k\ (m::nat) \Longrightarrow [k * a = k * b\ ]$ $(mod\ m) = [a = b]$ $(mod\ m)$
⟨*proof*⟩

lemma *cong-mult-lcancel-int*:
$coprime\ k\ (m::int) \Longrightarrow [k * a = k * b]$ $(mod\ m) = [a = b]$ $(mod\ m)$
⟨*proof*⟩

lemma *coprime-cong-mult-int*:
[(*a*::*int*) = *b*] (*mod m*) ⟹ [*a* = *b*] (*mod n*) ⟹ *coprime m n*
⟹ [*a* = *b*] (*mod m* ∗ *n*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *cong-less-imp-eq-nat*: *0* ≤ (*a*::*nat*) ⟹
a < *m* ⟹ *0* ≤ *b* ⟹ *b* < *m* ⟹ [*a* = *b*] (*mod m*) ⟹ *a* = *b*
⟨*proof*⟩

lemma *cong-less-imp-eq-int*: *0* ≤ (*a*::*int*) ⟹
a < *m* ⟹ *0* ≤ *b* ⟹ *b* < *m* ⟹ [*a* = *b*] (*mod m*) ⟹ *a* = *b*
⟨*proof*⟩

lemma *cong-less-unique-nat*:
0 < (*m*::*nat*) ⟹ (∃!*b*. *0* ≤ *b* ∧ *b* < *m* ∧ [*a* = *b*] (*mod m*))
⟨*proof*⟩

lemma *cong-less-unique-int*:
0 < (*m*::*int*) ⟹ (∃!*b*. *0* ≤ *b* ∧ *b* < *m* ∧ [*a* = *b*] (*mod m*))
⟨*proof*⟩

lemma *cong-iff-lin-int*: ([(*a*::*int*) = *b*] (*mod m*)) = (∃ *k*. *b* = *a* + *m* ∗ *k*)
⟨*proof*⟩

lemma *cong-iff-lin-nat*:
([(*a*::*nat*) = *b*] (*mod m*)) ⟷ (∃ *k1 k2*. *b* + *k1* ∗ *m* = *a* + *k2* ∗ *m*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *cong-gcd-eq-int*: [(*a*::*int*) = *b*] (*mod m*) ⟹ *gcd a m* = *gcd b m*
⟨*proof*⟩

lemma *cong-gcd-eq-nat*:
[(*a*::*nat*) = *b*] (*mod m*) ⟹*gcd a m* = *gcd b m*
⟨*proof*⟩

lemma *cong-imp-coprime-nat*: [(*a*::*nat*) = *b*] (*mod m*) ⟹ *coprime a m* ⟹ *coprime b m*
⟨*proof*⟩

lemma *cong-imp-coprime-int*: [(*a*::*int*) = *b*] (*mod m*) ⟹ *coprime a m* ⟹ *coprime b m*
⟨*proof*⟩

lemma *cong-cong-mod-nat*: $[(a::nat) = b]$ $(mod\ m) = [a\ mod\ m = b\ mod\ m]$ $(mod\ m)$
⟨*proof*⟩

lemma *cong-cong-mod-int*: $[(a::int) = b]$ $(mod\ m) = [a\ mod\ m = b\ mod\ m]$ $(mod\ m)$
⟨*proof*⟩

lemma *cong-minus-int* [*iff*]: $[(a::int) = b]$ $(mod\ -m) = [a = b]$ $(mod\ m)$
⟨*proof*⟩

lemma *cong-add-lcancel-nat*:
$[(a::nat) + x = a + y]$ $(mod\ n) \longleftrightarrow [x = y]$ $(mod\ n)$
⟨*proof*⟩

lemma *cong-add-lcancel-int*:
$[(a::int) + x = a + y]$ $(mod\ n) \longleftrightarrow [x = y]$ $(mod\ n)$
⟨*proof*⟩

lemma *cong-add-rcancel-nat*: $[(x::nat) + a = y + a]$ $(mod\ n) \longleftrightarrow [x = y]$ $(mod\ n)$
⟨*proof*⟩

lemma *cong-add-rcancel-int*: $[(x::int) + a = y + a]$ $(mod\ n) \longleftrightarrow [x = y]$ $(mod\ n)$
⟨*proof*⟩

lemma *cong-add-lcancel-0-nat*: $[(a::nat) + x = a]$ $(mod\ n) \longleftrightarrow [x = 0]$ $(mod\ n)$
⟨*proof*⟩

lemma *cong-add-lcancel-0-int*: $[(a::int) + x = a]$ $(mod\ n) \longleftrightarrow [x = 0]$ $(mod\ n)$
⟨*proof*⟩

lemma *cong-add-rcancel-0-nat*: $[x + (a::nat) = a]$ $(mod\ n) \longleftrightarrow [x = 0]$ $(mod\ n)$
⟨*proof*⟩

lemma *cong-add-rcancel-0-int*: $[x + (a::int) = a]$ $(mod\ n) \longleftrightarrow [x = 0]$ $(mod\ n)$
⟨*proof*⟩

lemma *cong-dvd-modulus-nat*: $[(x::nat) = y]$ $(mod\ m) \Longrightarrow n\ dvd\ m \Longrightarrow$
$[x = y]$ $(mod\ n)$
⟨*proof*⟩

lemma *cong-dvd-modulus-int*: $[(x::int) = y]$ $(mod\ m) \Longrightarrow n\ dvd\ m \Longrightarrow [x = y]$ $(mod\ n)$
⟨*proof*⟩

lemma *cong-dvd-eq-nat*: $[(x::nat) = y]$ $(mod\ n) \Longrightarrow n\ dvd\ x \longleftrightarrow n\ dvd\ y$
⟨*proof*⟩

lemma *cong-dvd-eq-int*: $[(x::int) = y]$ $(mod\ n) \Longrightarrow n\ dvd\ x \longleftrightarrow n\ dvd\ y$
⟨*proof*⟩

lemma *cong-mod-nat*: $(n::nat)$ ~$= 0 \Longrightarrow [a\ mod\ n = a]$ $(mod\ n)$
⟨*proof*⟩

lemma *cong-mod-int*: $(n::int)$ ~$= 0 \Longrightarrow [a\ mod\ n = a]$ $(mod\ n)$
⟨*proof*⟩

lemma *mod-mult-cong-nat*: $(a::nat)$ ~$= 0 \Longrightarrow b$ ~$= 0$
$\Longrightarrow [x\ mod\ (a * b) = y]$ $(mod\ a) \longleftrightarrow [x = y]$ $(mod\ a)$
⟨*proof*⟩

lemma *neg-cong-int*: $([(a::int) = b]$ $(mod\ m)) = ([-a = -b]$ $(mod\ m))$
⟨*proof*⟩

lemma *cong-modulus-neg-int*: $([(a::int) = b]$ $(mod\ m)) = ([a = b]$ $(mod\ -m))$
⟨*proof*⟩

lemma *mod-mult-cong-int*: $(a::int)$ ~$= 0 \Longrightarrow b$ ~$= 0$
$\Longrightarrow [x\ mod\ (a * b) = y]$ $(mod\ a) \longleftrightarrow [x = y]$ $(mod\ a)$
⟨*proof*⟩

lemma *cong-to-1-nat*: $([(a::nat) = 1]$ $(mod\ n)) \Longrightarrow (n\ dvd\ (a - 1))$
⟨*proof*⟩

lemma *cong-0-1-nat'*: $[(0::nat) = Suc\ 0]$ $(mod\ n) = (n = Suc\ 0)$
⟨*proof*⟩

lemma *cong-0-1-nat*: $[(0::nat) = 1]$ $(mod\ n) = (n = 1)$
⟨*proof*⟩

lemma *cong-0-1-int*: $[(0::int) = 1]$ $(mod\ n) = ((n = 1) \mid (n = -1))$
⟨*proof*⟩

lemma *cong-to-1'-nat*: $[(a::nat) = 1]$ $(mod\ n) \longleftrightarrow$
$a = 0 \wedge n = 1 \vee (\exists m.\ a = 1 + m * n)$
⟨*proof*⟩

lemma *cong-le-nat*: $(y::nat) <= x \Longrightarrow [x = y]$ $(mod\ n) \longleftrightarrow (\exists q.\ x = q * n + y)$
⟨*proof*⟩

lemma *cong-solve-nat*: $(a::nat) \neq 0 \Longrightarrow EX\ x.\ [a * x = gcd\ a\ n]$ $(mod\ n)$
⟨*proof*⟩

lemma *cong-solve-int*: $(a::int) \neq 0 \Longrightarrow EX\ x.\ [a * x = gcd\ a\ n]$ $(mod\ n)$

⟨proof⟩

lemma *cong-solve-dvd-nat*:
assumes *a*: (*a*::*nat*) ≠ *0* **and** *b*: *gcd a n dvd d*
shows *EX x*. [*a* ∗ *x* = *d*] (*mod n*)
⟨proof⟩

lemma *cong-solve-dvd-int*:
assumes *a*: (*a*::*int*) ≠ *0* **and** *b*: *gcd a n dvd d*
shows *EX x*. [*a* ∗ *x* = *d*] (*mod n*)
⟨proof⟩

lemma *cong-solve-coprime-nat*: *coprime* (*a*::*nat*) *n* ⟹ *EX x*. [*a* ∗ *x* = *1*] (*mod n*)
⟨proof⟩

lemma *cong-solve-coprime-int*: *coprime* (*a*::*int*) *n* ⟹ *EX x*. [*a* ∗ *x* = *1*] (*mod n*)
⟨proof⟩

lemma *coprime-iff-invertible-nat*:
m > *0* ⟹ *coprime a m* = (*EX x*. [*a* ∗ *x* = *Suc 0*] (*mod m*))
⟨proof⟩

lemma *coprime-iff-invertible-int*: *m* > (*0*::*int*) ⟹ *coprime a m* = (*EX x*. [*a* ∗ *x* = *1*] (*mod m*))
⟨proof⟩

lemma *coprime-iff-invertible′-nat*: *m* > *0* ⟹ *coprime a m* =
(*EX x*. *0* ≤ *x* & *x* < *m* & [*a* ∗ *x* = *Suc 0*] (*mod m*))
⟨proof⟩

lemma *coprime-iff-invertible′-int*: *m* > (*0*::*int*) ⟹ *coprime a m* =
(*EX x*. *0* <= *x* & *x* < *m* & [*a* ∗ *x* = *1*] (*mod m*))
⟨proof⟩

lemma *cong-cong-lcm-nat*: [(*x*::*nat*) = *y*] (*mod a*) ⟹
[*x* = *y*] (*mod b*) ⟹ [*x* = *y*] (*mod lcm a b*)
⟨proof⟩

lemma *cong-cong-lcm-int*: [(*x*::*int*) = *y*] (*mod a*) ⟹
[*x* = *y*] (*mod b*) ⟹ [*x* = *y*] (*mod lcm a b*)
⟨proof⟩

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*))
⟨proof⟩

lemma *cong-cong-setprod-coprime-int* [*rule-format*]: *finite A* $\Longrightarrow$
$(\forall\, i{\in}A.\ (\forall\, j{\in}A.\ i \neq j \longrightarrow coprime\ (m\ i)\ (m\ j))) \longrightarrow$
$(\forall\, i{\in}A.\ [(x{::}int) = y]\ (mod\ m\ i)) \longrightarrow$
$[x = y]\ (mod\ (\prod i{\in}A.\ m\ i))$
⟨*proof*⟩

lemma *binary-chinese-remainder-aux-nat*:
assumes *a*: *coprime* (*m1*::*nat*) *m2*
shows $EX\ b1\ b2.\ [b1 = 1]\ (mod\ m1) \wedge [b1 = 0]\ (mod\ m2) \wedge$
$[b2 = 0]\ (mod\ m1) \wedge [b2 = 1]\ (mod\ m2)$
⟨*proof*⟩

lemma *binary-chinese-remainder-aux-int*:
assumes *a*: *coprime* (*m1*::*int*) *m2*
shows $EX\ b1\ b2.\ [b1 = 1]\ (mod\ m1) \wedge [b1 = 0]\ (mod\ m2) \wedge$
$[b2 = 0]\ (mod\ m1) \wedge [b2 = 1]\ (mod\ m2)$
⟨*proof*⟩

lemma *binary-chinese-remainder-nat*:
assumes *a*: *coprime* (*m1*::*nat*) *m2*
shows $EX\ x.\ [x = u1]\ (mod\ m1) \wedge [x = u2]\ (mod\ m2)$
⟨*proof*⟩

lemma *binary-chinese-remainder-int*:
assumes *a*: *coprime* (*m1*::*int*) *m2*
shows $EX\ x.\ [x = u1]\ (mod\ m1) \wedge [x = u2]\ (mod\ m2)$
⟨*proof*⟩

lemma *cong-modulus-mult-nat*: $[(x{::}nat) = y]\ (mod\ m * n) \Longrightarrow$
$[x = y]\ (mod\ m)$
⟨*proof*⟩

lemma *cong-modulus-mult-int*: $[(x{::}int) = y]\ (mod\ m * n) \Longrightarrow$
$[x = y]\ (mod\ m)$
⟨*proof*⟩

lemma *cong-less-modulus-unique-nat*:
$[(x{::}nat) = y]\ (mod\ m) \Longrightarrow x < m \Longrightarrow y < m \Longrightarrow x = y$
⟨*proof*⟩

lemma *binary-chinese-remainder-unique-nat*:
assumes *a*: *coprime* (*m1*::*nat*) *m2*
and *nz*: $m1 \neq 0\ m2 \neq 0$
shows $EX!\ x.\ x < m1 * m2 \wedge [x = u1]\ (mod\ m1) \wedge [x = u2]\ (mod\ m2)$
⟨*proof*⟩

lemma *chinese-remainder-aux-nat*:
fixes $A :: 'a\ set$
and $m :: 'a \Rightarrow nat$

assumes *fin*: *finite A*
and *cop*: $ALL\ i : A.\ (ALL\ j : A.\ i \neq j \longrightarrow coprime\ (m\ i)\ (m\ j))$
shows $EX\ b.\ (ALL\ i : A.\ [b\ i = 1]\ (mod\ m\ i) \wedge [b\ i = 0]\ (mod\ (\prod j \in A - \{i\}.\ m\ j)))$
⟨*proof*⟩

lemma *chinese-remainder-nat*:
fixes $A :: 'a\ set$
and $m :: 'a \Rightarrow nat$
and $u :: 'a \Rightarrow nat$
assumes *fin*: *finite A*
and *cop*: $ALL\ i{:}A.\ (ALL\ j : A.\ i \neq j \longrightarrow coprime\ (m\ i)\ (m\ j))$
shows $EX\ x.\ (ALL\ i{:}A.\ [x = u\ i]\ (mod\ m\ i))$
⟨*proof*⟩

lemma *coprime-cong-prod-nat* [*rule-format*]: $finite\ A \Longrightarrow$
$(\forall i{\in}A.\ (\forall j{\in}A.\ i \neq j \longrightarrow coprime\ (m\ i)\ (m\ j))) \longrightarrow$
$(\forall i{\in}A.\ [(x{::}nat) = y]\ (mod\ m\ i)) \longrightarrow$
$[x = y]\ (mod\ (\prod i{\in}A.\ m\ i))$
⟨*proof*⟩

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*⟩

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*:
$n > 0 \Longrightarrow (\exists M.\ (\forall p{::}nat \in set\text{-}mset\ M.\ prime\ p) \wedge n = (\prod i \in\#\ M.\ i))$
⟨*proof*⟩

lemma *multiset-prime-factorization-unique-aux*:
fixes $a :: nat$

assumes $\forall p \in$*set-mset* M. *prime* p
and $\forall p \in$*set-mset* N. *prime* p
and $(\prod i \in\# M. i)$ *dvd* $(\prod i \in\# N. i)$
shows *count* M $a \leq$ *count* N a
⟨*proof*⟩

lemma *multiset-prime-factorization-unique*:
assumes $\forall p$::*nat* $\in$ *set-mset* M. *prime* p
and $\forall p \in$ *set-mset* N. *prime* p
and $(\prod i \in\# M. i) = (\prod i \in\# N. i)$
shows $M = N$
⟨*proof*⟩

definition *multiset-prime-factorization* :: *nat* ⇒ *nat multiset*
where
multiset-prime-factorization n =
(*if* $n > 0$
then THE M. ($\forall p \in$ *set-mset* M. *prime* p) ∧ $n = (\prod i \in\# M. i)$
else {#})

lemma *multiset-prime-factorization*: $n > 0 \Longrightarrow$
($\forall p \in$ *set-mset* (*multiset-prime-factorization* n). *prime* p) ∧
$n = (\prod i \in\#$ (*multiset-prime-factorization* n). i)
⟨*proof*⟩

3.2 Prime factors and multiplicity for nat and int

class *unique-factorization* =
fixes *multiplicity* :: '*a* ⇒ '*a* ⇒ *nat*
and *prime-factors* :: '*a* ⇒ '*a set*

Definitions for the natural numbers.

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

definition *multiplicity-nat* :: *nat* ⇒ *nat* ⇒ *nat*
where *multiplicity-nat* p n = *count* (*multiset-prime-factorization* n) p

definition *prime-factors-nat* :: *nat* ⇒ *nat set*
where *prime-factors-nat* n = *set-mset* (*multiset-prime-factorization* n)

instance ⟨*proof*⟩

end

Definitions for the integers.

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

definition *multiplicity-int* :: *int* ⇒ *int* ⇒ *nat*
where *multiplicity-int p n* = *multiplicity* (*nat p*) (*nat n*)

definition *prime-factors-int* :: *int* ⇒ *int set*
where *prime-factors-int n* = *int* ‘ (*prime-factors* (*nat n*))

instance ⟨*proof*⟩

end

3.3 Set up transfer

lemma *transfer-nat-int-prime-factors*: *prime-factors* (*nat n*) = *nat* ‘ *prime-factors* *n*
⟨*proof*⟩

lemma *transfer-nat-int-prime-factors-closure*: $n \geq 0 \Longrightarrow$ *nat-set* (*prime-factors* *n*)
⟨*proof*⟩

lemma *transfer-nat-int-multiplicity*:
$p \geq 0 \Longrightarrow n \geq 0 \Longrightarrow$ *multiplicity* (*nat p*) (*nat n*) = *multiplicity p n*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *transfer-int-nat-prime-factors-closure*: *is-nat n* ⟹ *nat-set* (*prime-factors* *n*)
⟨*proof*⟩

lemma *transfer-int-nat-multiplicity*: *multiplicity* (*int p*) (*int n*) = *multiplicity p n*
⟨*proof*⟩

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* $\Longrightarrow p \geq 0$

⟨*proof*⟩

lemma *prime-factors-prime-nat* [*intro*]:
fixes n :: *nat*
shows $p \in$ *prime-factors* $n \Longrightarrow$ *prime* p
⟨*proof*⟩

lemma *prime-factors-prime-int* [*intro*]:
fixes n :: *int*
assumes $n \geq 0$ **and** $p \in$ *prime-factors* n
shows *prime* p
⟨*proof*⟩

lemma *prime-factors-gt-0-nat*:
fixes p :: *nat*
shows $p \in$ *prime-factors* $x \Longrightarrow p > 0$
⟨*proof*⟩

lemma *prime-factors-gt-0-int*:
shows $x \geq 0 \Longrightarrow p \in$ *prime-factors* $x \Longrightarrow$ *int* $p > (0::int)$
⟨*proof*⟩

lemma *prime-factors-finite-nat* [*iff*]:
fixes n :: *nat*
shows *finite* (*prime-factors* n)
⟨*proof*⟩

lemma *prime-factors-finite-int* [*iff*]:
fixes n :: *int*
shows *finite* (*prime-factors* n)
⟨*proof*⟩

lemma *prime-factors-altdef-nat*:
fixes n :: *nat*
shows *prime-factors* $n = \{p.$ *multiplicity* $p\ n > 0\}$
⟨*proof*⟩

lemma *prime-factors-altdef-int*:
fixes n :: *int*
shows *prime-factors* $n = \{p.\ p \geq 0 \wedge$ *multiplicity* $p\ n > 0\}$
⟨*proof*⟩

lemma *prime-factorization-nat*:
fixes n :: *nat*
shows $n > 0 \Longrightarrow n = (\prod p \in$ *prime-factors* $n.\ p$ ^ *multiplicity* $p\ n)$
⟨*proof*⟩

lemma *prime-factorization-int*:
fixes n :: *int*

assumes $n > 0$
shows $n = (\prod p \in \textit{prime-factors}\ n.\ p\ \hat{}\ \textit{multiplicity}\ p\ n)$
⟨*proof*⟩

lemma *prime-factorization-unique-nat*:
fixes $f :: \textit{nat} \Rightarrow$ -
assumes *S-eq*: $S = \{p.\ 0 < f\ p\}$
and *finite* S
and S: $\forall p \in S.\ \textit{prime}\ p\ n = (\prod p \in S.\ p\ \hat{}\ f\ p)$
shows $S = \textit{prime-factors}\ n \wedge (\forall p.\ f\ p = \textit{multiplicity}\ p\ n)$
⟨*proof*⟩

lemma *prime-factors-characterization-nat*:
$S = \{p.\ 0 < f\ (p::\textit{nat})\} \Longrightarrow$
finite $S \Longrightarrow \forall p \in S.\ \textit{prime}\ p \Longrightarrow n = (\prod p \in S.\ p\ \hat{}\ f\ p) \Longrightarrow \textit{prime-factors}\ n =$
S
⟨*proof*⟩

lemma *prime-factors-characterization′-nat*:
finite $\{p.\ 0 < f\ (p::\textit{nat})\} \Longrightarrow$
$(\forall p.\ 0 < f\ p \longrightarrow \textit{prime}\ p) \Longrightarrow$
prime-factors $(\prod p \mid 0 < f\ p.\ p\ \hat{}\ f\ p) = \{p.\ 0 < f\ p\}$
⟨*proof*⟩

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

lemma *primes-characterization′-int* [*rule-format*]:
finite $\{p.\ p \geq 0 \wedge 0 < f\ (p::\textit{int})\} \Longrightarrow \forall p.\ 0 < f\ p \longrightarrow \textit{prime}\ p \Longrightarrow$
prime-factors $(\prod p \mid p \geq 0 \wedge 0 < f\ p.\ p\ \hat{}\ f\ p) = \{p.\ p \geq 0 \wedge 0 < f\ p\}$
⟨*proof*⟩

lemma *prime-factors-characterization-int*:
$S = \{p.\ 0 < f\ (p::\textit{int})\} \Longrightarrow \textit{finite}\ S \Longrightarrow$
$\forall p \in S.\ \textit{prime}\ (\textit{nat}\ p) \Longrightarrow n = (\prod p \in S.\ p\ \hat{}\ f\ p) \Longrightarrow \textit{prime-factors}\ n = S$
⟨*proof*⟩

lemma *multiplicity-characterization-nat*:
$S = \{p.\ 0 < f\ (p::\textit{nat})\} \Longrightarrow \textit{finite}\ S \Longrightarrow \forall p \in S.\ \textit{prime}\ p \Longrightarrow$
$n = (\prod p \in S.\ p\ \hat{}\ f\ p) \Longrightarrow \textit{multiplicity}\ p\ n = f\ p$
⟨*proof*⟩

lemma *multiplicity-characterization′-nat*: *finite* $\{p.\ 0 < f\ (p::\textit{nat})\} \longrightarrow$
$(\forall p.\ 0 < f\ p \longrightarrow \textit{prime}\ p) \longrightarrow$
multiplicity $p\ (\prod p \mid 0 < f\ p.\ p\ \hat{}\ f\ p) = f\ p$

⟨*proof*⟩

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

lemma *multiplicity-characterization-int*: $S = \{p.\ 0 < f\ (p::int)\} \Longrightarrow$
$finite\ S \Longrightarrow \forall p{\in}S.\ prime\ (nat\ p) \Longrightarrow n = (\prod p{\in}S.\ p\ \hat{}\ f\ p) \Longrightarrow$
$p \geq 0 \Longrightarrow multiplicity\ p\ n = f\ p$
⟨*proof*⟩

lemma *multiplicity-zero-nat* [*simp*]: $multiplicity\ (p::nat)\ 0 = 0$
⟨*proof*⟩

lemma *multiplicity-zero-int* [*simp*]: $multiplicity\ (p::int)\ 0 = 0$
⟨*proof*⟩

lemma *multiplicity-one-nat′*: $multiplicity\ p\ (1::nat) = 0$
⟨*proof*⟩

lemma *multiplicity-one-nat* [*simp*]: $multiplicity\ p\ (Suc\ 0) = 0$
⟨*proof*⟩

lemma *multiplicity-one-int* [*simp*]: $multiplicity\ p\ (1::int) = 0$
⟨*proof*⟩

lemma *multiplicity-prime-nat* [*simp*]: $prime\ p \Longrightarrow multiplicity\ p\ p = 1$
⟨*proof*⟩

lemma *multiplicity-prime-power-nat* [*simp*]: $prime\ p \Longrightarrow multiplicity\ p\ (p\ \hat{}\ n) =$
n
⟨*proof*⟩

lemma *multiplicity-prime-power-int* [*simp*]: $prime\ p \Longrightarrow multiplicity\ p\ (int\ p\ \hat{}\ n)$
$= n$
⟨*proof*⟩

lemma *multiplicity-nonprime-nat* [*simp*]:
fixes $p\ n :: nat$
shows $\neg\ prime\ p \Longrightarrow multiplicity\ p\ n = 0$
⟨*proof*⟩

lemma *multiplicity-not-factor-nat* [*simp*]:
fixes $p\ n :: nat$
shows $p \notin prime\text{-}factors\ n \Longrightarrow multiplicity\ p\ n = 0$
⟨*proof*⟩

lemma *multiplicity-not-factor-int* [*simp*]:
  fixes n :: *int*
  shows $p \geq 0 \Longrightarrow p \notin$ *prime-factors* $n \Longrightarrow$ *multiplicity* p n = *0*
  ⟨*proof*⟩

lemma *multiplicity-product-aux-nat*: (k::*nat*) > *0* $\Longrightarrow l >$ *0* $\Longrightarrow$
    (*prime-factors* k) ∪ (*prime-factors* l) = *prime-factors* ($k * l$) ∧
    (∀ p. *multiplicity* p k + *multiplicity* p l = *multiplicity* p ($k * l$))
  ⟨*proof*⟩

lemma *multiplicity-product-aux-int*:
  assumes (k::*int*) > *0* **and** $l >$ *0*
  shows *prime-factors* k ∪ *prime-factors* l = *prime-factors* ($k * l$) ∧
    (∀ $p \geq$ *0*. *multiplicity* p k + *multiplicity* p l = *multiplicity* p ($k * l$))
  ⟨*proof*⟩

lemma *prime-factors-product-nat*: (k::*nat*) > *0* $\Longrightarrow l >$ *0* $\Longrightarrow$ *prime-factors* (k *
l) =
    prime-factors k ∪ *prime-factors* l
  ⟨*proof*⟩

lemma *prime-factors-product-int*: (k::*int*) > *0* $\Longrightarrow l >$ *0* $\Longrightarrow$ *prime-factors* (k *
l) =
    prime-factors k ∪ *prime-factors* l
  ⟨*proof*⟩

lemma *multiplicity-product-nat*: (k::*nat*) > *0* $\Longrightarrow l >$ *0* $\Longrightarrow$ *multiplicity* p ($k * l$)
=
    multiplicity p k + *multiplicity* p l
  ⟨*proof*⟩

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
  ⟨*proof*⟩

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 ($f\ x$))
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*))
⟨*proof*⟩

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 ($\prod p \in S.\ p$ ^ *f p*) = (*if* $p \in S$ *then f p else 0*)
⟨*proof*⟩

lemma *multiplicity-prod-prime-powers-int*:
(*p*::*int*) $\geq 0 \Longrightarrow$ *finite S* $\Longrightarrow \forall p \in S.$ *prime* (*nat p*) $\Longrightarrow$
multiplicity p ($\prod p \in S.\ p$ ^ *f p*) = (*if* $p \in S$ *then f p else 0*)
⟨*proof*⟩

lemma *multiplicity-distinct-prime-power-nat*:
prime p $\Longrightarrow$ *prime q* $\Longrightarrow p \neq q \Longrightarrow$ *multiplicity p* (*q* ^ *n*) = *0*
⟨*proof*⟩

lemma *multiplicity-distinct-prime-power-int*:
prime p $\Longrightarrow$ *prime q* $\Longrightarrow p \neq q \Longrightarrow$ *multiplicity p* (*int q* ^ *n*) = *0*
⟨*proof*⟩

lemma *dvd-multiplicity-nat*:
fixes *x y* :: *nat*
shows $0 < y \Longrightarrow$ *x dvd y* $\Longrightarrow$ *multiplicity p x* $\leq$ *multiplicity p y*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *dvd-prime-factors-nat* [*intro*]:
fixes *x y* :: *nat*
shows $0 < y \Longrightarrow$ *x dvd y* $\Longrightarrow$ *prime-factors x* $\leq$ *prime-factors y*
⟨*proof*⟩

lemma *dvd-prime-factors-int* [*intro*]:
fixes *x y* :: *int*
shows $0 < y \Longrightarrow 0 \leq x \Longrightarrow$ *x dvd y* $\Longrightarrow$ *prime-factors x* $\leq$ *prime-factors y*

⟨*proof*⟩

lemma *multiplicity-dvd-nat*:
fixes *x y* :: *nat*
shows $0 < x \Longrightarrow 0 < y \Longrightarrow \forall p.\ multiplicity\ p\ x \leq multiplicity\ p\ y \Longrightarrow x\ dvd\ y$
⟨*proof*⟩

lemma *multiplicity-dvd-int*:
fixes *x y* :: *int*
shows $0 < x \Longrightarrow 0 < y \Longrightarrow \forall p{\geq}0.\ multiplicity\ p\ x \leq multiplicity\ p\ y \Longrightarrow x\ dvd\ y$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *multiplicity-dvd′-int*:
fixes *x y* :: *int*
shows $0 < x \Longrightarrow 0 \leq y \Longrightarrow$
$\forall p.\ prime\ p \longrightarrow multiplicity\ p\ x \leq multiplicity\ p\ y \Longrightarrow x\ dvd\ y$
⟨*proof*⟩

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

lemma *dvd-multiplicity-eq-int*: $0 < (x{::}int) \Longrightarrow 0 < y \Longrightarrow$
$(x\ dvd\ y) = (\forall p{\geq}0.\ multiplicity\ p\ x \leq multiplicity\ p\ y)$
⟨*proof*⟩

lemma *prime-factors-altdef2-nat*:
fixes *n* :: *nat*
shows $n > 0 \Longrightarrow p \in prime\text{-}factors\ n \longleftrightarrow prime\ p \land p\ dvd\ n$
⟨*proof*⟩

lemma *prime-factors-altdef2-int*:
fixes *n* :: *int*
assumes $n > 0$
shows $p \in prime\text{-}factors\ n \longleftrightarrow prime\ p \land p\ dvd\ n$
⟨*proof*⟩

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 $x = y$
⟨*proof*⟩

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 $x = y$
⟨*proof*⟩

3.5 An application

lemma *gcd-eq-nat*:
fixes *x y* :: *nat*
assumes *pos* [*arith*]: $x > 0$ $y > 0$
shows *gcd x y* =
($\prod p \in$ *prime-factors x* $\cup$ *prime-factors y*. *p* ^ *min* (*multiplicity p x*) (*multiplicity p y*))
(**is** - = *?z*)
⟨*proof*⟩

lemma *lcm-eq-nat*:
assumes *pos* [*arith*]: $x > 0$ $y > 0$
shows *lcm* (*x*::*nat*) *y* =
($\prod p \in$ *prime-factors x* $\cup$ *prime-factors y*. *p* ^ *max* (*multiplicity p x*) (*multiplicity p y*))
(**is** - = *?z*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *gcd-lcm-distrib-nat*:
fixes *x y z* :: *nat*
shows *gcd x* (*lcm y z*) = *lcm* (*gcd x y*) (*gcd x z*)
⟨*proof*⟩

lemma *gcd-lcm-distrib-int*:

fixes *x y z* :: *int*
shows *gcd x* (*lcm y z*) = *lcm* (*gcd x y*) (*gcd x z*)
⟨*proof*⟩

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*}
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma (**in** *comm-monoid*) *units-comm-group*: *comm-group*(*units-of G*)
⟨*proof*⟩

lemma *units-of-carrier*: *carrier* (*units-of G*) = *Units G*
⟨*proof*⟩

lemma *units-of-mult*: *mult*(*units-of G*) = *mult G*
⟨*proof*⟩

lemma *units-of-one*: *one*(*units-of G*) = *one G*
⟨*proof*⟩

lemma (**in** *monoid*) *units-of-inv*: *x* : *Units G* ==> *m-inv* (*units-of G*) *x* = *m-inv G x*
⟨*proof*⟩

lemma (**in** *group*) *inj-on-const-mult*: *a*: (*carrier G*) ==> *inj-on* (%*x*. *a* ⊗ *x*) (*carrier G*)
⟨*proof*⟩

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

lemma (**in** *group*) *l-cancel-one* [*simp*]:
x : *carrier G* ⟹ *a* : *carrier G* ⟹ (*x* ⊗ *a* = *x*) = (*a* = *one G*)
⟨*proof*⟩

lemma (**in** *group*) *r-cancel-one* [*simp*]: *x* : *carrier G* ⟹ *a* : *carrier G* ⟹
(*a* ⊗ *x* = *x*) = (*a* = *one G*)
⟨*proof*⟩

lemma (**in** *group*) *l-cancel-one'* [*simp*]: *x* : *carrier G* ⟹ *a* : *carrier G* ⟹
(*x* = *x* ⊗ *a*) = (*a* = *one G*)
⟨*proof*⟩

lemma (**in** *group*) *r-cancel-one'* [*simp*]: *x* : *carrier G* ⟹ *a* : *carrier G* ⟹
(*x* = *a* ⊗ *x*) = (*a* = *one G*)
⟨*proof*⟩

lemma (**in** *comm-group*) *power-order-eq-one*:
assumes *fin* [*simp*]: *finite* (*carrier G*)
and *a* [*simp*]: *a* : *carrier G*
shows *a* (^) *card*(*carrier G*) = *one G*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma (**in** *monoid*) *inv-char*: *x* : *carrier G* ⟹ *y* : *carrier G* ⟹
x ⊗ *y* = **1** ⟹ *y* ⊗ *x* = **1** ⟹ *inv x* = *y*
⟨*proof*⟩

lemma (**in** *comm-monoid*) *comm-inv-char*: *x* : *carrier G* ⟹ *y* : *carrier G* ⟹
x ⊗ *y* = **1** ⟹ *inv x* = *y*

⟨*proof*⟩

lemma (**in** *ring*) *inv-neg-one* [*simp*]: *inv* (⊖ **1**) = ⊖ **1**
⟨*proof*⟩

lemma (**in** *monoid*) *inv-eq-imp-eq*: *x* : *Units G* ⟹ *y* : *Units G* ⟹
inv x = *inv y* ⟹ *x* = *y*
⟨*proof*⟩

lemma (**in** *ring*) *Units-minus-one-closed* [*intro*]: ⊖ **1** : *Units R*
⟨*proof*⟩

lemma (**in** *monoid*) *inv-one* [*simp*]: *inv* **1** = **1**
⟨*proof*⟩

lemma (**in** *ring*) *inv-eq-neg-one-eq*: *x* : *Units R* ⟹ (*inv x* = ⊖ **1**) = (*x* = ⊖ **1**)
⟨*proof*⟩

lemma (**in** *monoid*) *inv-eq-one-eq*: *x* : *Units G* ⟹ (*inv x* = **1**) = (*x* = **1**)
⟨*proof*⟩

4.2.3 This goes in FiniteProduct

lemma (**in** *comm-monoid*) *finprod-UN-disjoint*:
finite I ⟹ (*ALL i*:*I*. *finite* (*A i*)) ⟶ (*ALL i*:*I*. *ALL j*:*I*. *i* ~= *j* ⟶
(*A i*) *Int* (*A j*) = {}) ⟶
(*ALL i*:*I*. *ALL x*: (*A i*). *g x* : *carrier G*) ⟶
finprod G g (*UNION I A*) = *finprod G* (%*i*. *finprod G g* (*A i*)) *I*
⟨*proof*⟩

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* (⋃ *C*) = *finprod G* (*finprod G f*) *C*
⟨*proof*⟩

lemma (**in** *comm-monoid*) *finprod-one*:
finite A ⟹ (⋀*x*. *x*:*A* ⟹ *f x* = **1**) ⟹ *finprod G f A* = **1**
⟨*proof*⟩

lemma (**in** *cring*) *sum-zero-eq-neg*: *x* : *carrier R* ⟹ *y* : *carrier R* ⟹ *x* ⊕ *y* =
0 ⟹ *x* = ⊖ *y*
⟨*proof*⟩

lemma (**in** *domain*) *square-eq-one*:

fixes *x*
assumes [*simp*]: *x* : *carrier R*
and $x \otimes x = \mathbf{1}$
shows $x = \mathbf{1} \mid x = \ominus\mathbf{1}$
⟨*proof*⟩

lemma (**in** *Ring.domain*) *inv-eq-self*: *x* : *Units R* $\Longrightarrow$ *x* = *inv x* $\Longrightarrow$ $x = \mathbf{1} \vee x = \ominus\mathbf{1}$
⟨*proof*⟩

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

lemma (**in** *ring*) *finite-ring-finite-units* [*intro*]: *finite* (*carrier R*) $\Longrightarrow$ *finite* (*Units R*)
⟨*proof*⟩

lemma (**in** *monoid*) *units-of-pow*:
fixes *n* :: *nat*
shows $x \in$ *Units G* $\Longrightarrow$ $x\ (\hat{})_{units\text{-}of\ G}\ n = x\ (\hat{})_G\ n$
⟨*proof*⟩

lemma (**in** *cring*) *units-power-order-eq-one*: *finite* (*Units R*) $\Longrightarrow$ *a* : *Units R*
$\Longrightarrow$ *a* (ˆ) *card*(*Units R*) = $\mathbf{1}$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *comm-monoid*: *comm-monoid R*
⟨*proof*⟩

lemma *cring*: *cring R*
⟨*proof*⟩

end

sublocale *residues* < *cring*
⟨*proof*⟩

context *residues*
begin

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

lemma *res-carrier-eq*: *carrier R* = {*0..m* − *1*}
⟨*proof*⟩

lemma *res-add-eq*: *x* ⊕ *y* = (*x* + *y*) *mod m*
⟨*proof*⟩

lemma *res-mult-eq*: *x* ⊗ *y* = (*x* ∗ *y*) *mod m*
⟨*proof*⟩

lemma *res-zero-eq*: **0** = *0*
⟨*proof*⟩

lemma *res-one-eq*: **1** = *1*
⟨*proof*⟩

lemma *res-units-eq*: *Units R* = {*x*. *0* < *x* ∧ *x* < *m* ∧ *coprime x m*}
⟨*proof*⟩

lemma *res-neg-eq*: ⊖ *x* = (− *x*) *mod m*
⟨*proof*⟩

lemma *finite* [*iff*]: *finite* (*carrier R*)
⟨*proof*⟩

lemma *finite-Units* [*iff*]: *finite* (*Units R*)
⟨*proof*⟩

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

lemma *mod-in-carrier* [*iff*]: *a mod m* ∈ *carrier R*
⟨*proof*⟩

lemma *add-cong*: (*x mod m*) ⊕ (*y mod m*) = (*x* + *y*) *mod m*
⟨*proof*⟩

lemma *mult-cong*: (*x mod m*) ⊗ (*y mod m*) = (*x* ∗ *y*) *mod m*
⟨*proof*⟩

lemma *zero-cong*: **0** = *0*
⟨*proof*⟩

lemma *one-cong*: **1** = *1 mod m*
⟨*proof*⟩

lemma *pow-cong*: (*x mod m*) (ˆ) *n* = *x*ˆ*n mod m*
⟨*proof*⟩

lemma *neg-cong*: ⊖ (*x mod m*) = (− *x*) *mod m*
⟨*proof*⟩

lemma (**in** *residues*) *prod-cong*: *finite A* ⟹ (⨂ *i*∈*A*. (*f i*) *mod m*) = (∏ *i*∈*A*. *f i*) *mod m*
⟨*proof*⟩

lemma (**in** *residues*) *sum-cong*: *finite A* ⟹ (⨁ *i*∈*A*. (*f i*) *mod m*) = (∑ *i*∈*A*. *f i*) *mod m*
⟨*proof*⟩

lemma *mod-in-res-units* [*simp*]:
assumes *1* < *m* **and** *coprime a m*
shows *a mod m* ∈ *Units R*
⟨*proof*⟩

lemma *res-eq-to-cong*: (*a mod m*) = (*b mod m*) ⟷ [*a* = *b*] (*mod m*)
⟨*proof*⟩

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

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

Other useful facts about the residue ring.

lemma *one-eq-neg-one*: **1** = ⊖ **1** ⟹ *m* = *2*
⟨*proof*⟩

end

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*
  ⟨*proof*⟩

context *residues-prime*
begin

lemma *is-field*: *field R*
  ⟨*proof*⟩

lemma *res-prime-units-eq*: *Units R* = {*1..p* − *1*}
  ⟨*proof*⟩

end

sublocale *residues-prime* < *field*
  ⟨*proof*⟩

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*}
  ⟨*proof*⟩

lemma *prime-phi*:
  assumes *2* ≤ *p phi p* = *p* − *1*
  shows *prime p*
⟨*proof*⟩

lemma *phi-zero* [*simp*]: *phi 0* = *0*
  ⟨*proof*⟩

lemma *phi-one* [*simp*]: *phi 1* = *0*
  ⟨*proof*⟩

lemma (**in** *residues*) *phi-eq*: *phi m* = *card* (*Units R*)

⟨*proof*⟩

lemma (**in** *residues*) *euler-theorem1*:
assumes *a*: *gcd a m* = *1*
shows [*a*ˆ*phi m* = *1*] (*mod m*)
⟨*proof*⟩

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

lemma *euler-theorem*:
assumes *m* ≥ *0*
and *gcd a m* = *1*
shows [*a*ˆ*phi m* = *1*] (*mod m*)
⟨*proof*⟩

lemma (**in** *residues-prime*) *phi-prime*: *phi p* = *nat p* − *1*
⟨*proof*⟩

lemma *phi-prime*: *prime p* ⟹ *phi p* = *nat p* − *1*
⟨*proof*⟩

lemma *fermat-theorem*:
fixes *a* :: *int*
assumes *prime p*
and ¬ *p dvd a*
shows [*a*ˆ(*p* − *1*) = *1*] (*mod p*)
⟨*proof*⟩

lemma *fermat-theorem-nat*:
assumes *prime p* **and** ¬ *p dvd a*
shows [*a* ˆ (*p* − *1*) = *1*] (*mod p*)
⟨*proof*⟩

6.2 Wilson's theorem

lemma (**in** *field*) *inv-pair-lemma*: *x* ∈ *Units R* ⟹ *y* ∈ *Units R* ⟹
{*x*, *inv x*} ≠ {*y*, *inv y*} ⟹ {*x*, *inv x*} ∩ {*y*, *inv y*} = {}
⟨*proof*⟩

lemma (**in** *residues-prime*) *wilson-theorem1*:
assumes *a*: *p* > *2*
shows [*fact* (*p* − *1*) = (−*1*::*int*)] (*mod p*)
⟨*proof*⟩

lemma *wilson-theorem*:
assumes *prime p*
shows [*fact* (*p* − *1*) = − *1*] (*mod p*)
⟨*proof*⟩

end

7 Pocklington's Theorem for Primes

theory *Pocklington*
imports *Residues*
begin

7.1 Lemmas about previously defined terms

lemma *prime*:
  prime p $\longleftrightarrow$ *p* $\neq$ *0* $\wedge$ *p* $\neq$ *1* $\wedge$ ($\forall$ *m*. *0* < *m* $\wedge$ *m* < *p* $\longrightarrow$ *coprime p m*)
  (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *finite-number-segment*: *card* { *m*. *0* < *m* $\wedge$ *m* < *n* } = *n* − *1*
⟨*proof*⟩

7.2 Some basic theorems about solving congruences

lemma *cong-solve*:
  fixes *n*::*nat* **assumes** *an*: *coprime a n* **shows** $\exists$ *x*. [*a* ∗ *x* = *b*] (*mod n*)
⟨*proof*⟩

lemma *cong-solve-unique*:
  fixes *n*::*nat* **assumes** *an*: *coprime a n* **and** *nz*: *n* $\neq$ *0*
  shows $\exists$!*x*. *x* < *n* $\wedge$ [*a* ∗ *x* = *b*] (*mod n*)
⟨*proof*⟩

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*⟩

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*)
  ⟨*proof*⟩

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*⟩

7.3 Lucas's theorem

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

lemma *phi-lowerbound-1*: **assumes** *n*: $n \geq 2$
shows *phi n* ≥ 1
⟨*proof*⟩

lemma *phi-lowerbound-1-nat*: **assumes** *n*: $n \geq 2$
shows *phi*(*int n*) ≥ 1
⟨*proof*⟩

lemma *euler-theorem-nat*:
fixes *m*::*nat*
assumes *coprime a m*
shows [*a* ^ *phi m* = *1*] (*mod m*)
⟨*proof*⟩

lemma *lucas-coprime-lemma*:
fixes *n*::*nat*
assumes *m*: *m*≠*0* **and** *am*: [*a*^*m* = *1*] (*mod n*)
shows *coprime a n*
⟨*proof*⟩

lemma *lucas-weak*:
fixes *n*::*nat*
assumes *n*: $n \geq 2$ **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*⟩

lemma *nat-exists-least-iff*: (∃ (*n*::*nat*). *P n*) ⟷ (∃ *n*. *P n* ∧ (∀ *m* < *n*. ¬ *P m*))
⟨*proof*⟩

lemma *nat-exists-least-iff'*: (∃ (*n*::*nat*). *P n*) ⟷ (*P* (*Least P*) ∧ (∀ *m* < (*Least P*). ¬ *P m*))
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

theorem *lucas*:
assumes *n2*: $n \geq 2$ **and** *an1*: [*a*^(*n* − *1*) = *1*] (*mod n*)
and *pn*: ∀ *p*. *prime p* ∧ *p dvd n* − *1* ⟶ [*a*^((*n* − *1*) *div p*) ≠ *1*] (*mod n*)
shows *prime n*
⟨*proof*⟩

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* (λ*d*. *d* > *0* ∧ [*a* ^*d* = *1*] (*mod n*)) *else 0*)

lemma *coprime-ord*:
fixes *n::nat*
assumes *coprime n a*
shows *ord n a > 0* ∧ [*a* ˆ(*ord n a*) = *1*] (*mod n*) ∧ (∀ *m*. *0* < *m* ∧ *m* < *ord n a* ⟶ [*a*ˆ *m* ≠ *1*] (*mod n*))
⟨*proof*⟩

lemma *ord-works*:
fixes *n::nat*
shows [*a* ˆ (*ord n a*) = *1*] (*mod n*) ∧ (∀ *m*. *0* < *m* ∧ *m* < *ord n a* ⟶ ~[*a*ˆ *m* = *1*] (*mod n*))
⟨*proof*⟩

lemma *ord*:
fixes *n::nat*
shows [*a*ˆ(*ord n a*) = *1*] (*mod n*) ⟨*proof*⟩

lemma *ord-minimal*:
fixes *n::nat*
shows *0* < *m* ⟹ *m* < *ord n a* ⟹ ~[*a*ˆ*m* = *1*] (*mod n*)
⟨*proof*⟩

lemma *ord-eq-0*:
fixes *n::nat*
shows *ord n a* = *0* ⟷ ~*coprime n a*
⟨*proof*⟩

lemma *divides-rexp*:
x dvd y ⟹ (*x::nat*) *dvd* (*y*ˆ(*Suc n*))
⟨*proof*⟩

lemma *ord-divides*:
fixes *n::nat*
shows [*a* ˆ *d* = *1*] (*mod n*) ⟷ *ord n a dvd d* (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *order-divides-phi*:
fixes *n::nat* **shows** *coprime n a* ⟹ *ord n a dvd phi n*
⟨*proof*⟩

lemma *order-divides-expdiff*:
fixes *n::nat* **and** *a::nat* **assumes** *na*: *coprime n a*
shows [*a*ˆ*d* = *a*ˆ*e*] (*mod n*) ⟷ [*d* = *e*] (*mod* (*ord n a*))
⟨*proof*⟩

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*⟩

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*⟩

lemma *prime-prime-factor-sqrt*:
prime n $\longleftrightarrow$ *n* $\neq$ *0* $\wedge$ *n* $\neq$ *1* $\wedge$ $\neg$ ($\exists$ *p. prime p* $\wedge$ *p dvd n* $\wedge$ $p^2 \leq n$)
(**is** *?lhs* $\longleftrightarrow$*?rhs*)
⟨*proof*⟩

7.6 Pocklington theorem

lemma *pocklington-lemma*:
assumes *n*: *n* $\geq$ *2* **and** *nqr*: *n* − *1* = *q*∗*r* **and** *an*: [*a*ˆ (*n* − *1*) = *1*] (*mod n*)
and *aq*:$\forall$ *p. prime p* $\wedge$ *p dvd q* $\longrightarrow$ *coprime* (*a*ˆ ((*n* − *1*) *div p*) − *1*) *n*
and *pp*: *prime p* **and** *pn*: *p dvd n*
shows [*p* = *1*] (*mod q*)
⟨*proof*⟩

theorem *pocklington*:
assumes *n*: *n* $\geq$ *2* **and** *nqr*: *n* − *1* = *q*∗*r* **and** *sqr*: $n \leq q^2$
and *an*: [*a*ˆ (*n* − *1*) = *1*] (*mod n*)
and *aq*: $\forall$ *p. prime p* $\wedge$ *p dvd q* $\longrightarrow$ *coprime* (*a*ˆ ((*n* − *1*) *div p*) − *1*) *n*
shows *prime n*
⟨*proof*⟩

lemma *pocklington-alt*:
assumes *n*: *n* $\geq$ *2* **and** *nqr*: *n* − *1* = *q*∗*r* **and** *sqr*: $n \leq q^2$
and *an*: [*a*ˆ (*n* − *1*) = *1*] (*mod n*)
and *aq*:$\forall$ *p. prime p* $\wedge$ *p dvd q* $\longrightarrow$ ($\exists$ *b.* [*a*ˆ((*n* − *1*) *div p*) = *b*] (*mod n*) $\wedge$ *coprime* (*b* − *1*) *n*)
shows *prime n*
⟨*proof*⟩

7.7 Prime factorizations

definition *primefact ps n* = (*foldr op* ∗ *ps* *1* = *n* $\wedge$ ($\forall$ *p*∈ *set ps. prime p*))

lemma *primefact*: **assumes** *n*: *n* $\neq$ *0*
shows $\exists$ *ps. primefact ps n*
⟨*proof*⟩

lemma *primefact-contains*:
assumes *pf*: *primefact ps n* **and** *p*: *prime p* **and** *pn*: *p dvd n*
shows *p* ∈ *set ps*
⟨*proof*⟩

lemma *primefact-variant*: *primefact ps n* ⟷ *foldr op* ∗ *ps 1* = *n* ∧ *list-all prime ps*
⟨*proof*⟩

lemma *lucas-primefact*:
assumes *n*: $n \geq 2$ **and** *an*: [*a*ˆ(*n* − *1*) = *1*] (*mod n*)
and *psn*: *foldr op* ∗ *ps 1* = *n* − *1*
and *psp*: *list-all* (λ*p*. *prime p* ∧ ¬ [*a*ˆ((*n* − *1*) *div p*) = *1*] (*mod n*)) *ps*
shows *prime n*
⟨*proof*⟩

lemma *pocklington-primefact*:
assumes *n*: $n \geq 2$ **and** *qrn*: *q*∗*r* = *n* − *1* **and** *nq2*: $n \leq q^2$
and *arnb*: (*a*ˆ*r*) *mod n* = *b* **and** *psq*: *foldr op* ∗ *ps 1* = *q*
and *bqn*: (*b*ˆ*q*) *mod n* = *1*
and *psp*: *list-all* (λ*p*. *prime p* ∧ *coprime* ((*b*ˆ(*q div p*)) *mod n* − *1*) *n*) *ps*
shows *prime n*
⟨*proof*⟩

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*⟧ ⟹ [*a* = *0*] (*mod p*) | [*b* = *0*] (*mod p*)
⟨*proof*⟩

lemma *cong-prime-prod-zero-int*:
fixes *a*::*int*
shows ⟦[*a* ∗ *b* = *0*] (*mod p*); *prime p*⟧ ⟹ [*a* = *0*] (*mod p*) | [*b* = *0*] (*mod p*)
⟨*proof*⟩

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 $B = (\lambda x.\ x * a)$ ' A
definition $C = (\lambda x.\ x\ mod\ p)$ ' B
definition $D = C \cap \{..\ (int\ p - 1)\ div\ 2\}$
definition $E = C \cap \{(int\ p - 1)\ div\ 2 <..\}$
definition $F = (\lambda x.\ (int\ p - x))$ ' E

8.1 Basic properties of p

lemma *odd-p*: *odd p*
⟨*proof*⟩

lemma *p-minus-one-l*: $(int\ p - 1)\ div\ 2 < p$
⟨*proof*⟩

lemma *p-eq2*: $int\ p = (2 * ((int\ p - 1)\ div\ 2)) + 1$
⟨*proof*⟩

lemma *p-odd-int*: **obtains** *z::int* **where** $int\ p = 2*z+1$ $0<z$
⟨*proof*⟩

8.2 Basic Properties of the Gauss Sets

lemma *finite-A*: *finite* (A)
⟨*proof*⟩

lemma *finite-B*: *finite* (B)
⟨*proof*⟩

lemma *finite-C*: *finite* (C)
⟨*proof*⟩

lemma *finite-D*: *finite* (D)
⟨*proof*⟩

lemma *finite-E*: *finite* (E)
⟨*proof*⟩

lemma *finite-F*: *finite* (F)
⟨*proof*⟩

lemma *C-eq*: $C = D \cup E$
⟨*proof*⟩

lemma *A-card-eq*: $card\ A = nat\ ((int\ p - 1)\ div\ 2)$
⟨*proof*⟩

lemma *inj-on-xa-A*: $inj\text{-}on\ (\lambda x.\ x * a)\ A$
⟨*proof*⟩

definition *ResSet* :: *int => int set => bool*
where $ResSet\ m\ X = (\forall\, y1\ y2.\ (y1 \in X\ \&\ y2 \in X\ \&\ [y1 = y2]\ (mod\ m)\ -->\ y1 = y2))$

lemma *ResSet-image*:
$[\![\ 0 < m;\ ResSet\ m\ A;\ \forall\, x \in A.\ \forall\, y \in A.\ ([f\,x = f\,y](mod\ m)\ -->\ x = y)\ ]\!] \Longrightarrow ResSet\ m\ (f\ `\ A)$
⟨*proof*⟩

lemma *A-res*: *ResSet p A*
⟨*proof*⟩

lemma *B-res*: *ResSet p B*
⟨*proof*⟩

lemma *SR-B-inj*: $inj\text{-}on\ (\lambda x.\ x\ mod\ p)\ B$
⟨*proof*⟩

lemma *inj-on-pminusx-E*: $inj\text{-}on\ (\lambda x.\ p - x)\ E$
⟨*proof*⟩

lemma *nonzero-mod-p*:
fixes *x*::*int* **shows** $[\![0 < x;\ x < int\ p]\!] \Longrightarrow [x \neq 0](mod\ p)$
⟨*proof*⟩

lemma *A-ncong-p*: $x \in A \Longrightarrow [x \neq 0](mod\ p)$
⟨*proof*⟩

lemma *A-greater-zero*: $x \in A \Longrightarrow 0 < x$
⟨*proof*⟩

lemma *B-ncong-p*: $x \in B \Longrightarrow [x \neq 0](mod\ p)$
⟨*proof*⟩

lemma *B-greater-zero*: $x \in B \Longrightarrow 0 < x$
⟨*proof*⟩

lemma *C-greater-zero*: $y \in C \Longrightarrow 0 < y$
⟨*proof*⟩

lemma *F-subset*: $F \subseteq \{x.\ 0 < x\ \&\ x \leq ((int\ p - 1)\ div\ 2)\}$
⟨*proof*⟩

lemma *D-subset*: $D \subseteq \{x.\ 0 < x\ \&\ x \leq ((p - 1)\ div\ 2)\}$
⟨*proof*⟩

lemma *F-eq*: $F = \{x. \exists\, y \in A.\ (\ x = p - ((y*a)\ mod\ p)\ \&\ (int\ p - 1)\ div\ 2 < (y*a)\ mod\ p)\}$
⟨*proof*⟩

lemma *D-eq*: $D = \{x. \exists\, y \in A.\ (\ x = (y*a)\ mod\ p\ \&\ (y*a)\ mod\ p \leq (int\ p - 1)\ div\ 2)\}$
⟨*proof*⟩

lemma *all-A-relprime*: **assumes** $x \in A$ **shows** $gcd\ x\ p = 1$
⟨*proof*⟩

lemma *A-prod-relprime*: $gcd\ (setprod\ id\ A)\ p = 1$
⟨*proof*⟩

8.3 Relationships Between Gauss Sets

lemma *StandardRes-inj-on-ResSet*: $ResSet\ m\ X \Longrightarrow (inj\text{-}on\ (\lambda b.\ b\ mod\ m)\ X)$
⟨*proof*⟩

lemma *B-card-eq-A*: $card\ B = card\ A$
⟨*proof*⟩

lemma *B-card-eq*: $card\ B = nat\ ((int\ p - 1)\ div\ 2)$
⟨*proof*⟩

lemma *F-card-eq-E*: $card\ F = card\ E$
⟨*proof*⟩

lemma *C-card-eq-B*: $card\ C = card\ B$
⟨*proof*⟩

lemma *D-E-disj*: $D \cap E = \{\}$
⟨*proof*⟩

lemma *C-card-eq-D-plus-E*: $card\ C = card\ D + card\ E$
⟨*proof*⟩

lemma *C-prod-eq-D-times-E*: $setprod\ id\ E * setprod\ id\ D = setprod\ id\ C$
⟨*proof*⟩

lemma *C-B-zcong-prod*: $[setprod\ id\ C = setprod\ id\ B]\ (mod\ p)$
⟨*proof*⟩

lemma *F-Un-D-subset*: $(F \cup D) \subseteq A$
⟨*proof*⟩

lemma *F-D-disj*: $(F \cap D) = \{\}$
⟨*proof*⟩

lemma *F-Un-D-card*: *card* $(F \cup D) = $ *nat* $((p - 1)$ *div* $2)$
⟨*proof*⟩

lemma *F-Un-D-eq-A*: $F \cup D = A$
⟨*proof*⟩

lemma *prod-D-F-eq-prod-A*: (*setprod id D*) $*$ (*setprod id F*) = *setprod id A*
⟨*proof*⟩

lemma *prod-F-zcong*: [*setprod id F* = $((-1)$ ^ (*card E*)) $*$ (*setprod id E*)] (*mod p*)
⟨*proof*⟩

8.4 Gauss' Lemma

lemma *aux*: *setprod id A* $*$ $(- 1)$ ^ *card E* $*$ *a* ^ *card A* $*$ $(- 1)$ ^ *card E* = *setprod id A* $*$ *a* ^ *card A*
⟨*proof*⟩

theorem *pre-gauss-lemma*:
[*a* ^ *nat*((*int p* $- 1)$ *div* $2) = (-1)$ ^ (*card E*)] (*mod p*)
⟨*proof*⟩

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
⟨*proof*⟩

lemma *fib-2* [*simp*]: $fib\ (2::nat) = 1$
⟨*proof*⟩

lemma *fib-plus-2*: $fib\ (n + 2) = fib\ (n + 1) + fib\ n$
⟨*proof*⟩

lemma *fib-add*: $fib\ (Suc\ (n+k)) = fib\ (Suc\ k) * fib\ (Suc\ n) + fib\ k * fib\ n$
⟨*proof*⟩

lemma *fib-neq-0-nat*: $n > 0 \Longrightarrow fib\ n > 0$
⟨*proof*⟩

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*: $int\ (fib\ (Suc\ (Suc\ n)) * fib\ n) - int((fib\ (Suc\ n))^2) = -((-1)\ \hat{}n)$
⟨*proof*⟩

lemma *fib-Cassini-nat*:
$fib\ (Suc\ (Suc\ n)) * fib\ n =$
$(if\ even\ n\ then\ (fib\ (Suc\ n))^2 - 1\ else\ (fib\ (Suc\ n))^2 + 1)$
⟨*proof*⟩

9.4 Law 6.111 of Concrete Mathematics

lemma *coprime-fib-Suc-nat*: $coprime\ (fib\ (n::nat))\ (fib\ (Suc\ n))$
⟨*proof*⟩

lemma *gcd-fib-add*: $gcd\ (fib\ m)\ (fib\ (n + m)) = gcd\ (fib\ m)\ (fib\ n)$
⟨*proof*⟩

lemma *gcd-fib-diff*: $m \le n \Longrightarrow gcd\ (fib\ m)\ (fib\ (n - m)) = gcd\ (fib\ m)\ (fib\ n)$
⟨*proof*⟩

lemma *gcd-fib-mod*: $0 < m \Longrightarrow gcd\ (fib\ m)\ (fib\ (n\ mod\ m)) = gcd\ (fib\ m)\ (fib\ n)$
⟨*proof*⟩

lemma *fib-gcd*: $fib\ (gcd\ m\ n) = gcd\ (fib\ m)\ (fib\ n)$
— Law 6.111
⟨*proof*⟩

theorem *fib-mult-eq-setsum-nat*: $fib\ (Suc\ n) * fib\ n = (\sum k \in \{..n\}.\ fib\ k * fib\ k)$
⟨*proof*⟩

9.5 Fibonacci and Binomial Coefficients

lemma *setsum-drop-zero*: $(\sum k = 0..Suc\ n.\ if\ 0<k\ then\ (f\ (k - 1))\ else\ 0) = (\sum j = 0..n.\ f\ j)$
⟨*proof*⟩

lemma *setsum-choose-drop-zero*:
$(\sum k = 0..Suc\ n.\ if\ k=0\ then\ 0\ else\ (Suc\ n - k)\ choose\ (k - 1)) = (\sum j = 0..n.\ (n-j)\ choose\ j)$
⟨*proof*⟩

lemma *ne-diagonal-fib*: $(\sum k = 0..n.\ (n-k)\ choose\ k) = fib\ (Suc\ n)$
⟨*proof*⟩

end

10 The sieve of Eratosthenes

theory *Eratosthenes*
imports *Main Primes*
begin

10.1 Preliminary: strict divisibility

context *dvd*
begin

abbreviation *dvd-strict* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infixl** *dvd′-strict 50*)
where
b dvd-strict a $\equiv$ *b dvd a* $\wedge \neg$ *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* :: $nat \Rightarrow marks \Rightarrow nat\ set$
where
numbers-of-marks n bs = *fst* ‘ $\{x \in set\ (enumerate\ n\ bs).\ snd\ x\}$

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*
⟨*proof*⟩

lemma *numbers-of-marks-Suc*:

numbers-of-marks (*Suc n*) *bs* = *Suc* ‘ *numbers-of-marks n bs*
⟨*proof*⟩

lemma *numbers-of-marks-replicate-False* [*simp*]:
numbers-of-marks n (*replicate m False*) = {}
⟨*proof*⟩

lemma *numbers-of-marks-replicate-True* [*simp*]:
numbers-of-marks n (*replicate m True*) = {*n*..<*n*+*m*}
⟨*proof*⟩

lemma *in-numbers-of-marks-eq*:
m ∈ *numbers-of-marks n bs* ⟷ *m* ∈ {*n*..<*n* + *length bs*} ∧ *bs* ! (*m* − *n*)
⟨*proof*⟩

lemma *sorted-list-of-set-numbers-of-marks*:
sorted-list-of-set (*numbers-of-marks n bs*) = *map fst* (*filter snd* (*enumerate n bs*))
⟨*proof*⟩

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* [] = []
⟨*proof*⟩

lemma *length-mark-out* [*simp*]: *length* (*mark-out n bs*) = *length bs*
⟨*proof*⟩

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*}
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*}
⟨*proof*⟩

lemma *sorted-primes-upto* [*iff*]: *sorted* (*primes-upto n*)
⟨*proof*⟩

lemma *distinct-primes-upto* [*iff*]: *distinct* (*primes-upto n*)
⟨*proof*⟩

lemma *set-primes-upto-sieve*:
set (*primes-upto n*) = *numbers-of-marks 2* (*sieve 1* (*replicate* (*n* − *1*) *True*))
⟨*proof*⟩

lemma *primes-upto-sieve* [*code*]:
primes-upto n = *map fst* (*filter snd* (*enumerate 2* (*sieve 1* (*replicate* (*n* − *1*) *True*))))
⟨*proof*⟩

lemma *prime-in-primes-upto*: *prime n* ⟷ *n* ∈ *set* (*primes-upto n*)
⟨*proof*⟩

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*⟩

lemma *smallest-prime-beyond-smallest*: *prime p* ⟹ *p* ≥ *n* ⟹ *smallest-prime-beyond n* ≤ *p*
  ⟨*proof*⟩

lemma *smallest-prime-beyond-eq*:
  prime p ⟹ *p* ≥ *n* ⟹ (⋀*q. prime q* ⟹ *q* ≥ *n* ⟹ *q* ≥ *p*) ⟹ *smallest-prime-beyond n* = *p*
  ⟨*proof*⟩

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*)
  ⟨*proof*⟩

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

lemma [*code*]: *smallest-prime-between m n* = *List.find* (λ*p. p* ≥ *m*) (*primes-upto n*)
⟨*proof*⟩

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*)
  ⟨*proof*⟩

lemma [*code*]: *smallest-prime-beyond n* = *smallest-prime-beyond-aux 2 n*
⟨*proof*⟩

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* ⇒ *bool*) ⇒ *'a list* ⇒ *'a list*
where
strip-while P = *rev* ∘ *dropWhile P* ∘ *rev*

lemma *strip-while-rev* [*simp*]:
strip-while P (*rev xs*) = *rev* (*dropWhile P xs*)
⟨*proof*⟩

lemma *strip-while-Nil* [*simp*]:
strip-while P [] = []
⟨*proof*⟩

lemma *strip-while-append* [*simp*]:
¬ *P x* ⟹ *strip-while P* (*xs* @ [*x*]) = *xs* @ [*x*]
⟨*proof*⟩

lemma *strip-while-append-rec* [*simp*]:
P x ⟹ *strip-while P* (*xs* @ [*x*]) = *strip-while P xs*
⟨*proof*⟩

lemma *strip-while-Cons* [*simp*]:
¬ *P x* ⟹ *strip-while P* (*x* # *xs*) = *x* # *strip-while P xs*
⟨*proof*⟩

lemma *strip-while-eq-Nil* [*simp*]:
strip-while P xs = [] ⟷ (∀ *x*∈*set xs*. *P x*)
⟨*proof*⟩

lemma *strip-while-eq-Cons-rec*:

$strip\text{-}while\ P\ (x\ \#\ xs) = x\ \#\ strip\text{-}while\ P\ xs \longleftrightarrow \neg\ (P\ x \wedge (\forall x{\in}set\ xs.\ P\ x))$
⟨proof⟩

lemma *strip-while-not-last* [*simp*]:
$\neg\ P\ (last\ xs) \Longrightarrow strip\text{-}while\ P\ xs = xs$
⟨proof⟩

lemma *split-strip-while-append*:
fixes $xs :: 'a\ list$
obtains $ys\ zs :: 'a\ list$
where $strip\text{-}while\ P\ xs = ys$ **and** $\forall x{\in}set\ zs.\ P\ x$ **and** $xs = ys\ @\ zs$
⟨proof⟩

lemma *strip-while-snoc* [*simp*]:
$strip\text{-}while\ P\ (xs\ @\ [x]) = (if\ P\ x\ then\ strip\text{-}while\ P\ xs\ else\ xs\ @\ [x])$
⟨proof⟩

lemma *strip-while-map*:
$strip\text{-}while\ P\ (map\ f\ xs) = map\ f\ (strip\text{-}while\ (P \circ f)\ xs)$
⟨proof⟩

definition $no\text{-}leading :: ('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow bool$
where
$no\text{-}leading\ P\ xs \longleftrightarrow (xs \neq [] \longrightarrow \neg\ P\ (hd\ xs))$

lemma *no-leading-Nil* [*simp*, *intro*!]:
$no\text{-}leading\ P\ []$
⟨proof⟩

lemma *no-leading-Cons* [*simp*, *intro*!]:
$no\text{-}leading\ P\ (x\ \#\ xs) \longleftrightarrow \neg\ P\ x$
⟨proof⟩

lemma *no-leading-append* [*simp*]:
$no\text{-}leading\ P\ (xs\ @\ ys) \longleftrightarrow no\text{-}leading\ P\ xs \wedge (xs = [] \longrightarrow no\text{-}leading\ P\ ys)$
⟨proof⟩

lemma *no-leading-dropWhile* [*simp*]:
$no\text{-}leading\ P\ (dropWhile\ P\ xs)$
⟨proof⟩

lemma *dropWhile-eq-obtain-leading*:
assumes $dropWhile\ P\ xs = ys$
obtains zs **where** $xs = zs\ @\ ys$ **and** $\bigwedge z.\ z \in set\ zs \Longrightarrow P\ z$ **and** $no\text{-}leading\ P$
ys
⟨proof⟩

lemma *dropWhile-idem-iff*:

dropWhile P xs = xs ⟷ no-leading P xs
⟨*proof*⟩

abbreviation *no-trailing* :: *('a ⇒ bool) ⇒ 'a list ⇒ bool*
where
no-trailing P xs ≡ no-leading P (rev xs)

lemma *no-trailing-unfold*:
no-trailing P xs ⟷ (xs ≠ [] ⟶ ¬ P (last xs))
⟨*proof*⟩

lemma *no-trailing-Nil* [*simp*, *intro*!]:
no-trailing P []
⟨*proof*⟩

lemma *no-trailing-Cons* [*simp*]:
no-trailing P (x # xs) ⟷ no-trailing P xs ∧ (xs = [] ⟶ ¬ P x)
⟨*proof*⟩

lemma *no-trailing-append-Cons* [*simp*]:
no-trailing P (xs @ y # ys) ⟷ no-trailing P (y # ys)
⟨*proof*⟩

lemma *no-trailing-strip-while* [*simp*]:
no-trailing P (strip-while P xs)
⟨*proof*⟩

lemma *strip-while-eq-obtain-trailing*:
assumes *strip-while P xs = ys*
obtains *zs* **where** *xs = ys @ zs* **and** *⋀z. z ∈ set zs ⟹ P z* **and** *no-trailing P ys*
⟨*proof*⟩

lemma *strip-while-idem-iff*:
strip-while P xs = xs ⟷ no-trailing P xs
⟨*proof*⟩

lemma *no-trailing-map*:
no-trailing P (map f xs) = no-trailing (P ∘ f) xs
⟨*proof*⟩

lemma *no-trailing-upt* [*simp*]:
no-trailing P [n..<m] ⟷ (n < m ⟶ ¬ P (m − 1))
⟨*proof*⟩

definition *nth-default* :: *'a ⇒ 'a list ⇒ nat ⇒ 'a*
where

nth-default dflt xs n = (*if n* < *length xs then xs* ! *n else dflt*)

lemma *nth-default-nth*:
n < *length xs* ⟹ *nth-default dflt xs n* = *xs* ! *n*
⟨*proof*⟩

lemma *nth-default-beyond*:
length xs ≤ *n* ⟹ *nth-default dflt xs n* = *dflt*
⟨*proof*⟩

lemma *nth-default-Nil* [*simp*]:
nth-default dflt [] *n* = *dflt*
⟨*proof*⟩

lemma *nth-default-Cons*:
nth-default dflt (*x* # *xs*) *n* = (*case n of 0* ⇒ *x* | *Suc n'* ⇒ *nth-default dflt xs n'*)
⟨*proof*⟩

lemma *nth-default-Cons-0* [*simp*]:
nth-default dflt (*x* # *xs*) *0* = *x*
⟨*proof*⟩

lemma *nth-default-Cons-Suc* [*simp*]:
nth-default dflt (*x* # *xs*) (*Suc n*) = *nth-default dflt xs n*
⟨*proof*⟩

lemma *nth-default-replicate-dflt* [*simp*]:
nth-default dflt (*replicate n dflt*) *m* = *dflt*
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *nth-default-append-trailing* [*simp*]:
nth-default dflt (*xs* @ *replicate n dflt*) = *nth-default dflt xs*
⟨*proof*⟩

lemma *nth-default-snoc-default* [*simp*]:
nth-default dflt (*xs* @ [*dflt*]) = *nth-default dflt xs*
⟨*proof*⟩

lemma *nth-default-eq-dflt-iff*:
nth-default dflt xs k = *dflt* ⟷ (*k* < *length xs* ⟶ *xs* ! *k* = *dflt*)
⟨*proof*⟩

lemma *in-enumerate-iff-nth-default-eq*:
x ≠ *dflt* ⟹ (*n*, *x*) ∈ *set* (*enumerate 0 xs*) ⟷ *nth-default dflt xs n* = *x*

⟨*proof*⟩

lemma *last-conv-nth-default*:
assumes *xs* ≠ []
shows *last xs* = *nth-default dflt xs* (*length xs* − *1*)
⟨*proof*⟩

lemma *nth-default-map-eq*:
f dflt′ = *dflt* ⟹ *nth-default dflt* (*map f xs*) *n* = *f* (*nth-default dflt′ xs n*)
⟨*proof*⟩

lemma *finite-nth-default-neq-default* [*simp*]:
finite {*k*. *nth-default dflt xs k* ≠ *dflt*}
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *map-nth-default*:
map (*nth-default x xs*) [*0*..<*length xs*] = *xs*
⟨*proof*⟩

lemma *range-nth-default* [*simp*]:
range (*nth-default dflt xs*) = *insert dflt* (*set xs*)
⟨*proof*⟩

lemma *nth-strip-while*:
assumes *n* < *length* (*strip-while P xs*)
shows *strip-while P xs* ! *n* = *xs* ! *n*
⟨*proof*⟩

lemma *length-strip-while-le*:
length (*strip-while P xs*) ≤ *length xs*
⟨*proof*⟩

lemma *nth-default-strip-while-dflt* [*simp*]:
nth-default dflt (*strip-while* (*op* = *dflt*) *xs*) = *nth-default dflt xs*
⟨*proof*⟩

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*⟩

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*: $infinite\ (S::nat\ set) \longleftrightarrow (\forall m.\ \exists n \geq m.\ n \in S)$
  ⟨*proof*⟩

lemma *infinite-nat-iff-unbounded*: $infinite\ (S::nat\ set) \longleftrightarrow (\forall m.\ \exists n > m.\ n \in S)$
  ⟨*proof*⟩

lemma *finite-nat-iff-bounded*: $finite\ (S::nat\ set) \longleftrightarrow (\exists k.\ S \subseteq \{..<k\})$
  ⟨*proof*⟩

lemma *finite-nat-iff-bounded-le*: $finite\ (S::nat\ set) \longleftrightarrow (\exists k.\ S \subseteq \{..\ k\})$
  ⟨*proof*⟩

lemma *finite-nat-bounded*: $finite\ (S::nat\ set) \Longrightarrow \exists k.\ S \subseteq \{..<k\}$
  ⟨*proof*⟩

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*: $\forall m > k.\ \exists n > m.\ n \in S \Longrightarrow infinite\ (S::nat\ set)$
⟨*proof*⟩

lemma *nat-not-finite*: $finite\ (UNIV::nat\ set) \Longrightarrow R$
  ⟨*proof*⟩

lemma *range-inj-infinite*:
  $inj\ (f::nat \Rightarrow 'a) \Longrightarrow infinite\ (range\ f)$
⟨*proof*⟩

The set of integers is also infinite.

lemma *infinite-int-iff-infinite-nat-abs*: $infinite\ (S::int\ set) \longleftrightarrow infinite\ ((nat\ o\ abs)\ `\ S)$
  ⟨*proof*⟩

proposition *infinite-int-iff-unbounded-le*: $infinite\ (S::int\ set) \longleftrightarrow (\forall m.\ \exists n.\ |n| \geq m \wedge n \in S)$
  ⟨*proof*⟩

proposition *infinite-int-iff-unbounded*: $infinite\ (S::int\ set) \longleftrightarrow (\forall m.\ \exists n.\ |n| > m \wedge n \in S)$
  ⟨*proof*⟩

proposition *finite-int-iff-bounded*: *finite* (S::*int set*) $\longleftrightarrow$ ($\exists$ k. *abs* ‘ $S \subseteq \{..<k\}$)
⟨*proof*⟩

proposition *finite-int-iff-bounded-le*: *finite* (S::*int set*) $\longleftrightarrow$ ($\exists$ k. *abs* ‘ $S \subseteq \{..\ k\}$)
⟨*proof*⟩

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*]: $\neg$ (*INFM* x. P x) $\longleftrightarrow$ (*MOST* x. $\neg$ P x) ⟨*proof*⟩
lemma *not-MOST* [*simp*]: $\neg$ (*MOST* x. P x) $\longleftrightarrow$ (*INFM* x. $\neg$ P x) ⟨*proof*⟩

lemma *INFM-const* [*simp*]: (*INFM* x::$'a$. P) $\longleftrightarrow$ $P \wedge$ *infinite* (*UNIV*::$'a$ *set*)
⟨*proof*⟩

lemma *MOST-const* [*simp*]: (*MOST* x::$'a$. P) $\longleftrightarrow$ $P \vee$ *finite* (*UNIV*::$'a$ *set*)
⟨*proof*⟩

lemma *INFM-imp-distrib*: (*INFM* x. P x $\longrightarrow$ Q x) $\longleftrightarrow$ ((*MOST* x. P x) $\longrightarrow$ (*INFM* x. Q x))
⟨*proof*⟩

lemma *MOST-imp-iff*: *MOST* x. P x $\Longrightarrow$ (*MOST* x. P x $\longrightarrow$ Q x) $\longleftrightarrow$ (*MOST* x. Q x)
⟨*proof*⟩

lemma *INFM-conjI*: *INFM* x. P x $\Longrightarrow$ *MOST* x. Q x $\Longrightarrow$ *INFM* x. P x $\wedge$ Q x
⟨*proof*⟩

Properties of quantifiers with injective functions.

lemma *INFM-inj*: *INFM* x. P (f x) $\Longrightarrow$ *inj* f $\Longrightarrow$ *INFM* x. P x
⟨*proof*⟩

lemma *MOST-inj*: *MOST* x. P x $\Longrightarrow$ *inj* f $\Longrightarrow$ *MOST* x. P (f x)
⟨*proof*⟩

Properties of quantifiers with singletons.

lemma *not-INFM-eq* [*simp*]:
$\neg$ (*INFM* x. $x = a$)
$\neg$ (*INFM* x. $a = x$)
⟨*proof*⟩

lemma *MOST-neq* [*simp*]:
MOST x. $x \neq a$
MOST x. $a \neq x$
⟨*proof*⟩

lemma *INFM-neq* [*simp*]:
$(INFM\ x::'a.\ x \neq a) \longleftrightarrow infinite\ (UNIV::'a\ set)$
$(INFM\ x::'a.\ a \neq x) \longleftrightarrow infinite\ (UNIV::'a\ set)$
⟨*proof*⟩

lemma *MOST-eq* [*simp*]:
$(MOST\ x::'a.\ x = a) \longleftrightarrow finite\ (UNIV::'a\ set)$
$(MOST\ x::'a.\ a = x) \longleftrightarrow finite\ (UNIV::'a\ set)$
⟨*proof*⟩

lemma *MOST-eq-imp*:
$MOST\ x.\ x = a \longrightarrow P\ x$
$MOST\ x.\ a = x \longrightarrow P\ x$
⟨*proof*⟩

Properties of quantifiers over the naturals.

lemma *MOST-nat*: $(\forall_\infty n.\ P\ (n::nat)) \longleftrightarrow (\exists m.\ \forall n>m.\ P\ n)$
⟨*proof*⟩

lemma *MOST-nat-le*: $(\forall_\infty n.\ P\ (n::nat)) \longleftrightarrow (\exists m.\ \forall n\geq m.\ P\ n)$
⟨*proof*⟩

lemma *INFM-nat*: $(\exists_\infty n.\ P\ (n::nat)) \longleftrightarrow (\forall m.\ \exists n>m.\ P\ n)$
⟨*proof*⟩

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

lemma *MOST-INFM*: $infinite\ (UNIV::'a\ set) \Longrightarrow MOST\ x::'a.\ P\ x \Longrightarrow INFM\ x::'a.\ P\ x$
⟨*proof*⟩

lemma *MOST-Suc-iff*: $(MOST\ n.\ P\ (Suc\ n)) \longleftrightarrow (MOST\ n.\ P\ n)$
⟨*proof*⟩

lemma
shows *MOST-SucI*: $MOST\ n.\ P\ n \Longrightarrow MOST\ n.\ P\ (Suc\ n)$
and *MOST-SucD*: $MOST\ n.\ P\ (Suc\ n) \Longrightarrow MOST\ n.\ P\ n$
⟨*proof*⟩

lemma *MOST-ge-nat*: $MOST\ n::nat.\ m \leq n$
⟨*proof*⟩

lemma *Inf-many-def*: $Inf\text{-}many\ P \longleftrightarrow infinite\ \{x.\ P\ x\}$ ⟨*proof*⟩
lemma *Alm-all-def*: $Alm\text{-}all\ P \longleftrightarrow \neg\ (INFM\ x.\ \neg\ P\ x)$ ⟨*proof*⟩
lemma *INFM-iff-infinite*: $(INFM\ x.\ P\ x) \longleftrightarrow infinite\ \{x.\ P\ x\}$ ⟨*proof*⟩
lemma *MOST-iff-cofinite*: $(MOST\ x.\ P\ x) \longleftrightarrow finite\ \{x.\ \neg\ P\ x\}$ ⟨*proof*⟩

lemma *INFM-EX*: $(\exists_\infty x.\ P\ x) \Longrightarrow (\exists x.\ P\ x)$ ⟨*proof*⟩
lemma *ALL-MOST*: $\forall x.\ P\ x \Longrightarrow \forall_\infty x.\ P\ x$ ⟨*proof*⟩
lemma *INFM-mono*: $\exists_\infty x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow \exists_\infty x.\ Q\ x$ ⟨*proof*⟩
lemma *MOST-mono*: $\forall_\infty x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow \forall_\infty x.\ Q\ x$ ⟨*proof*⟩
lemma *INFM-disj-distrib*: $(\exists_\infty x.\ P\ x \vee Q\ x) \longleftrightarrow (\exists_\infty x.\ P\ x) \vee (\exists_\infty x.\ Q\ x)$ ⟨*proof*⟩
lemma *MOST-rev-mp*: $\forall_\infty x.\ P\ x \Longrightarrow \forall_\infty x.\ P\ x \longrightarrow Q\ x \Longrightarrow \forall_\infty x.\ Q\ x$ ⟨*proof*⟩
lemma *MOST-conj-distrib*: $(\forall_\infty x.\ P\ x \wedge Q\ x) \longleftrightarrow (\forall_\infty x.\ P\ x) \wedge (\forall_\infty x.\ Q\ x)$ ⟨*proof*⟩
lemma *MOST-conjI*: *MOST* $x.\ P\ x \Longrightarrow$ *MOST* $x.\ Q\ x \Longrightarrow$ *MOST* $x.\ P\ x \wedge Q\ x$ ⟨*proof*⟩
lemma *INFM-finite-Bex-distrib*: *finite* $A \Longrightarrow$ (*INFM* $y.\ \exists x{\in}A.\ P\ x\ y) \longleftrightarrow (\exists x{\in}A.$ *INFM* $y.\ P\ x\ y)$ ⟨*proof*⟩
lemma *MOST-finite-Ball-distrib*: *finite* $A \Longrightarrow$ (*MOST* $y.\ \forall x{\in}A.\ P\ x\ y) \longleftrightarrow (\forall x{\in}A.$ *MOST* $y.\ P\ x\ y)$ ⟨*proof*⟩
lemma *INFM-E*: *INFM* $x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow$ *thesis*$) \Longrightarrow$ *thesis* ⟨*proof*⟩
lemma *MOST-I*: $(\bigwedge x.\ P\ x) \Longrightarrow$ *MOST* $x.\ P\ x$ ⟨*proof*⟩
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* $t.\ t \in s \wedge$ *finite* $\{s \in S.\ s < t\} \wedge$ *card* $\{s \in S.\ s < t\} = n$).

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

lemma *enumerate-Suc'*: *enumerate* S (*Suc* n) = *enumerate* $(S - \{$*enumerate* $S\ 0\})\ n$
  ⟨*proof*⟩

lemma *enumerate-in-set*: *infinite* $S \Longrightarrow$ *enumerate* $S\ n \in S$
  ⟨*proof*⟩

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

lemma *enumerate-step*: *infinite* $S \Longrightarrow$ *enumerate* $S\ n <$ *enumerate* S (*Suc* n)
  ⟨*proof*⟩

lemma *enumerate-mono*: $m < n \Longrightarrow$ *infinite* $S \Longrightarrow$ *enumerate* $S\ m <$ *enumerate* $S\ n$
  ⟨*proof*⟩

lemma *le-enumerate*:
  assumes *S*: *infinite* S

shows $n \le$ *enumerate S n*
⟨*proof*⟩

lemma *enumerate-Suc''*:
fixes *S* :: *'a::wellorder set*
assumes *infinite S*
shows *enumerate S* (*Suc n*) = (*LEAST s. s* ∈ *S* ∧ *enumerate S n* < *s*)
⟨*proof*⟩

lemma *enumerate-Ex*:
assumes *S*: *infinite* (*S::nat set*)
shows *s* ∈ *S* ⟹ ∃ *n. enumerate S n* = *s*
⟨*proof*⟩

lemma *bij-enumerate*:
fixes *S* :: *nat set*
assumes *S*: *infinite S*
shows *bij-betw* (*enumerate S*) *UNIV S*
⟨*proof*⟩

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 ## [] = []
⟨*proof*⟩

lemma *cCons-Cons-eq* [*simp*]:
x ## y # ys = *x # y # ys*
⟨*proof*⟩

lemma *cCons-append-Cons-eq* [*simp*]:
x ## xs @ y # ys = *x # xs @ y # ys*
⟨*proof*⟩

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

$x \neq 0 \Longrightarrow x \mathbin{\#\#} xs = x \mathbin{\#} xs$
⟨proof⟩

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⟩

lemma *tl-cCons* [*simp*]:
$tl\ (x \mathbin{\#\#} xs) = xs$
⟨proof⟩

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* ⟨proof⟩

setup-lifting *type-definition-poly*

lemma *poly-eq-iff*: $p = q \longleftrightarrow (\forall n.\ coeff\ p\ n = coeff\ q\ n)$
⟨proof⟩

lemma *poly-eqI*: $(\bigwedge n.\ coeff\ p\ n = coeff\ q\ n) \Longrightarrow p = q$
⟨proof⟩

lemma *MOST-coeff-eq-0*: $\forall_\infty\ n.\ coeff\ p\ n = 0$
⟨proof⟩

14.3 Degree of a polynomial

definition *degree* :: $'a{::}zero$ *poly* $\Rightarrow$ *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⟩

lemma *le-degree*: *coeff* $p\ n \neq 0 \Longrightarrow n \leq$ *degree* p
⟨proof⟩

lemma *degree-le*: $\forall i{>}n.\ coeff\ p\ i = 0 \Longrightarrow$ *degree* $p \leq n$
⟨proof⟩

lemma *less-degree-imp*: $n <$ *degree* $p \Longrightarrow \exists i{>}n.\ coeff\ p\ i \neq 0$
⟨proof⟩

14.4 The zero polynomial

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

begin

lift-definition *zero-poly* :: *'a poly*
  is λ-. *0* ⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *coeff-0* [*simp*]:
  coeff 0 n = *0*
  ⟨*proof*⟩

lemma *degree-0* [*simp*]:
  degree 0 = *0*
  ⟨*proof*⟩

lemma *leading-coeff-neq-0*:
  assumes *p* ≠ *0*
  shows *coeff p* (*degree p*) ≠ *0*
⟨*proof*⟩

lemma *leading-coeff-0-iff* [*simp*]:
  coeff p (*degree p*) = *0* ⟷ *p* = *0*
  ⟨*proof*⟩

14.5 List-style constructor for polynomials

lift-definition *pCons* :: *'a::zero* ⇒ *'a poly* ⇒ *'a poly*
  is λ*a p*. *case-nat a* (*coeff p*)
  ⟨*proof*⟩

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

lemma *coeff-pCons-0* [*simp*]:
  coeff (*pCons a p*) *0* = *a*
  ⟨*proof*⟩

lemma *coeff-pCons-Suc* [*simp*]:
  coeff (*pCons a p*) (*Suc n*) = *coeff p n*
  ⟨*proof*⟩

lemma *degree-pCons-le*:
  degree (*pCons a p*) ≤ *Suc* (*degree p*)
  ⟨*proof*⟩

lemma *degree-pCons-eq*:
  p ≠ *0* ⟹ *degree* (*pCons a p*) = *Suc* (*degree p*)
  ⟨*proof*⟩

lemma *degree-pCons-0*:
  degree (*pCons a 0*) = *0*
  ⟨*proof*⟩

lemma *degree-pCons-eq-if* [*simp*]:
  degree (*pCons a p*) = (*if p* = *0 then 0 else Suc* (*degree p*))
  ⟨*proof*⟩

lemma *pCons-0-0* [*simp*]:
  pCons 0 0 = *0*
  ⟨*proof*⟩

lemma *pCons-eq-iff* [*simp*]:
  pCons a p = *pCons b q* ⟷ *a* = *b* ∧ *p* = *q*
⟨*proof*⟩

lemma *pCons-eq-0-iff* [*simp*]:
  pCons a p = *0* ⟷ *a* = *0* ∧ *p* = *0*
  ⟨*proof*⟩

lemma *pCons-cases* [*cases type*: *poly*]:
  obtains (*pCons*) *a q* **where** *p* = *pCons a q*
⟨*proof*⟩

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*⟩

lemma *degree-eq-zeroE*:
  fixes *p* :: *'a::zero poly*
  assumes *degree p* = *0*
  obtains *a* **where** *p* = *pCons a 0*
⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *Poly-eq-0*:
  Poly as = *0* ⟷ (∃ *n*. *as* = *replicate n 0*)
  ⟨*proof*⟩

lemma *degree-Poly*: *degree* (*Poly xs*) ≤ *length xs*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *not-0-coeffs-not-Nil*:
  p ≠ *0* ⟹ *coeffs p* ≠ []
  ⟨*proof*⟩

lemma *coeffs-0-eq-Nil* [*simp*]:
  coeffs 0 = []
  ⟨*proof*⟩

lemma *coeffs-pCons-eq-cCons* [*simp*]:
  coeffs (*pCons a p*) = *a* ## *coeffs p*
⟨*proof*⟩

lemma *length-coeffs*: *p* ≠ *0* ⟹ *length* (*coeffs p*) = *degree p* + *1*
  ⟨*proof*⟩

lemma *coeffs-nth*:
  assumes *p* ≠ *0 n* ≤ *degree p*
  shows *coeffs p* ! *n* = *coeff p n*
  ⟨*proof*⟩

lemma *not-0-cCons-eq* [*simp*]:
  p ≠ *0* ⟹ *a* ## *coeffs p* = *a* # *coeffs p*
  ⟨*proof*⟩

lemma *Poly-coeffs* [*simp*, *code abstype*]:
Poly (*coeffs p*) = *p*
⟨*proof*⟩

lemma *coeffs-Poly* [*simp*]:
coeffs (*Poly as*) = *strip-while* (*HOL.eq 0*) *as*
⟨*proof*⟩

lemma *last-coeffs-not-0*:
$p \neq 0 \Longrightarrow$ *last* (*coeffs p*) $\neq 0$
⟨*proof*⟩

lemma *strip-while-coeffs* [*simp*]:
strip-while (*HOL.eq 0*) (*coeffs p*) = *coeffs p*
⟨*proof*⟩

lemma *coeffs-eq-iff*:
$p = q \longleftrightarrow$ *coeffs p* = *coeffs q* (**is** $?P \longleftrightarrow ?Q$)
⟨*proof*⟩

lemma *coeff-Poly-eq*:
coeff (*Poly xs*) *n* = *nth-default 0 xs n*
⟨*proof*⟩

lemma *nth-default-coeffs-eq*:
nth-default 0 (*coeffs p*) = *coeff p*
⟨*proof*⟩

lemma [*code*]:
coeff p = *nth-default 0* (*coeffs p*)
⟨*proof*⟩

lemma *coeffs-eqI*:
assumes *coeff*: $\bigwedge n.$ *coeff p n* = *nth-default 0 xs n*
assumes *zero*: $xs \neq [] \Longrightarrow$ *last xs* $\neq 0$
shows *coeffs p* = *xs*
⟨*proof*⟩

lemma *degree-eq-length-coeffs* [*code*]:
degree p = *length* (*coeffs p*) − *1*
⟨*proof*⟩

lemma *length-coeffs-degree*:
$p \neq 0 \Longrightarrow$ *length* (*coeffs p*) = *Suc* (*degree p*)
⟨*proof*⟩

lemma [*code abstract*]:
coeffs 0 = []
⟨*proof*⟩

lemma [*code abstract*]:
coeffs (*pCons a p*) = *a* ## *coeffs p*
⟨*proof*⟩

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

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

instance
⟨*proof*⟩

end

lemma [*code nbe*]: *HOL.equal* (*p* :: - *poly*) *p* ⟷ *True*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *fold-coeffs-pCons-eq* [*simp*]:
f 0 = *id* ⟹ *fold-coeffs f* (*pCons a p*) = *f a* ∘ *fold-coeffs f p*
⟨*proof*⟩

lemma *fold-coeffs-pCons-0-0-eq* [*simp*]:
fold-coeffs f (*pCons 0 0*) = *id*
⟨*proof*⟩

lemma *fold-coeffs-pCons-coeff-not-0-eq* [*simp*]:
a ≠ *0* ⟹ *fold-coeffs f* (*pCons a p*) = *f a* ∘ *fold-coeffs f p*
⟨*proof*⟩

lemma *fold-coeffs-pCons-not-0-0-eq* [*simp*]:
$p \neq 0 \Longrightarrow$ *fold-coeffs f* (*pCons a p*) = *f a* ∘ *fold-coeffs f p*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *poly-pCons* [*simp*]:
poly (*pCons a p*) *x* = *a* + *x* ∗ *poly p x*
⟨*proof*⟩

lemma *poly-altdef*:
poly p (*x* :: *'a* :: {*comm-semiring-0*, *semiring-1*}) = (∑ *i*≤*degree p*. *coeff p i* ∗ *x* ^ *i*)
⟨*proof*⟩

lemma *poly-0-coeff-0*: *poly p 0* = *coeff p 0*
⟨*proof*⟩

14.11 Monomials

lift-definition *monom* :: *'a* ⇒ *nat* ⇒ *'a::zero poly*
is λ*a m n*. *if m* = *n then a else 0*
⟨*proof*⟩

lemma *coeff-monom* [*simp*]:
coeff (*monom a m*) *n* = (*if m* = *n then a else 0*)
⟨*proof*⟩

lemma *monom-0*:
monom a 0 = *pCons a 0*
⟨*proof*⟩

lemma *monom-Suc*:
monom a (*Suc n*) = *pCons 0* (*monom a n*)
⟨*proof*⟩

lemma *monom-eq-0* [*simp*]: *monom 0 n* = *0*
⟨*proof*⟩

lemma *monom-eq-0-iff* [*simp*]: *monom a n* = *0* ⟷ *a* = *0*
⟨*proof*⟩

lemma *monom-eq-iff* [*simp*]: *monom a n* = *monom b n* ⟷ *a* = *b*
⟨*proof*⟩

lemma *degree-monom-le*: *degree* (*monom a n*) ≤ *n*
⟨*proof*⟩

lemma *degree-monom-eq*: *a* ≠ *0* ⟹ *degree* (*monom a n*) = *n*
⟨*proof*⟩

lemma *coeffs-monom* [*code abstract*]:
coeffs (*monom a n*) = (*if a* = *0 then* [] *else replicate n 0* @ [*a*])
⟨*proof*⟩

lemma *fold-coeffs-monom* [*simp*]:
a ≠ *0* ⟹ *fold-coeffs f* (*monom a n*) = *f 0* ^^ *n* ∘ *f a*
⟨*proof*⟩

lemma *poly-monom*:
fixes *a x* :: ′*a*::{*comm-semiring-1*}
shows *poly* (*monom a n*) *x* = *a* ∗ *x* ^ *n*
⟨*proof*⟩

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 λ*p q n. coeff p n* + *coeff q n*
⟨*proof*⟩

lemma *coeff-add* [*simp*]: *coeff* (*p* + *q*) *n* = *coeff p n* + *coeff q n*
⟨*proof*⟩

instance
⟨*proof*⟩

end

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

lift-definition *minus-poly* :: ′*a poly* ⇒ ′*a poly* ⇒ ′*a poly*
is λ*p q n. coeff p n* − *coeff q n*
⟨*proof*⟩

lemma *coeff-diff* [*simp*]: *coeff* (*p* − *q*) *n* = *coeff p n* − *coeff q n*
⟨*proof*⟩

instance
⟨*proof*⟩

end

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

lift-definition *uminus-poly* :: *'a poly* ⇒ *'a poly*
  is $\lambda p\ n.\ -\ coeff\ p\ n$
⟨*proof*⟩

lemma *coeff-minus* [*simp*]: *coeff* (− *p*) *n* = − *coeff p n*
  ⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *add-pCons* [*simp*]:
  pCons a p + *pCons b q* = *pCons* (*a* + *b*) (*p* + *q*)
  ⟨*proof*⟩

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

lemma *diff-pCons* [*simp*]:
  pCons a p − *pCons b q* = *pCons* (*a* − *b*) (*p* − *q*)
  ⟨*proof*⟩

lemma *degree-add-le-max*: *degree* (*p* + *q*) ≤ *max* (*degree p*) (*degree q*)
  ⟨*proof*⟩

lemma *degree-add-le*:
  ⟦*degree p* ≤ *n*; *degree q* ≤ *n*⟧ ⟹ *degree* (*p* + *q*) ≤ *n*
  ⟨*proof*⟩

lemma *degree-add-less*:
  ⟦*degree p* < *n*; *degree q* < *n*⟧ ⟹ *degree* (*p* + *q*) < *n*
  ⟨*proof*⟩

lemma *degree-add-eq-right*:
  degree p < *degree q* ⟹ *degree* (*p* + *q*) = *degree q*
  ⟨*proof*⟩

lemma *degree-add-eq-left*:
  degree q < *degree p* ⟹ *degree* (*p* + *q*) = *degree p*

⟨*proof*⟩

lemma *degree-minus* [*simp*]:
degree (− *p*) = *degree p*
⟨*proof*⟩

lemma *degree-diff-le-max*:
fixes *p q* :: ′*a* :: *ab-group-add poly*
shows *degree* (*p* − *q*) ≤ *max* (*degree p*) (*degree q*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *add-monom*: *monom a n* + *monom b n* = *monom* (*a* + *b*) *n*
⟨*proof*⟩

lemma *diff-monom*: *monom a n* − *monom b n* = *monom* (*a* − *b*) *n*
⟨*proof*⟩

lemma *minus-monom*: − *monom a n* = *monom* (−*a*) *n*
⟨*proof*⟩

lemma *coeff-setsum*: *coeff* (∑ *x*∈*A*. *p x*) *i* = (∑ *x*∈*A*. *coeff* (*p x*) *i*)
⟨*proof*⟩

lemma *monom-setsum*: *monom* (∑ *x*∈*A*. *a x*) *n* = (∑ *x*∈*A*. *monom* (*a x*) *n*)
⟨*proof*⟩

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*⟩

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

coeffs (− *p*) = *map* (λ*a*. − *a*) (*coeffs p*)
⟨*proof*⟩

lemma [*code*]:
fixes *p q* :: ′*a*::*ab-group-add poly*
shows *p* − *q* = *p* + − *q*
⟨*proof*⟩

lemma *poly-add* [*simp*]: *poly* (*p* + *q*) *x* = *poly p x* + *poly q x*
⟨*proof*⟩

lemma *poly-minus* [*simp*]:
fixes *x* :: ′*a*::*comm-ring*
shows *poly* (− *p*) *x* = − *poly p x*
⟨*proof*⟩

lemma *poly-diff* [*simp*]:
fixes *x* :: ′*a*::*comm-ring*
shows *poly* (*p* − *q*) *x* = *poly p x* − *poly q x*
⟨*proof*⟩

lemma *poly-setsum*: *poly* (∑ *k*∈*A*. *p k*) *x* = (∑ *k*∈*A*. *poly* (*p k*) *x*)
⟨*proof*⟩

lemma *degree-setsum-le*: *finite S* ⟹ (⋀ *p* . *p* ∈ *S* ⟹ *degree* (*f p*) ≤ *n*)
⟹ *degree* (*setsum f S*) ≤ *n*
⟨*proof*⟩

lemma *poly-as-sum-of-monoms*′:
assumes *n*: *degree p* ≤ *n*
shows (∑ *i*≤*n*. *monom* (*coeff p i*) *i*) = *p*
⟨*proof*⟩

lemma *poly-as-sum-of-monoms*: (∑ *i*≤*degree p*. *monom* (*coeff p i*) *i*) = *p*
⟨*proof*⟩

lemma *Poly-snoc*: *Poly* (*xs* @ [*x*]) = *Poly xs* + *monom x* (*length xs*)
⟨*proof*⟩

14.13 Multiplication by a constant, polynomial multiplication and the unit polynomial

lift-definition *smult* :: ′*a*::*comm-semiring-0* ⇒ ′*a poly* ⇒ ′*a poly*
is λ*a p n*. *a* ∗ *coeff p n*
⟨*proof*⟩

lemma *coeff-smult* [*simp*]:
coeff (*smult a p*) *n* = *a* ∗ *coeff p n*
⟨*proof*⟩

lemma *degree-smult-le*: *degree* (*smult a p*) ≤ *degree p*
⟨*proof*⟩

lemma *smult-smult* [*simp*]: *smult a* (*smult b p*) = *smult* (*a* ∗ *b*) *p*
⟨*proof*⟩

lemma *smult-0-right* [*simp*]: *smult a 0* = *0*
⟨*proof*⟩

lemma *smult-0-left* [*simp*]: *smult 0 p* = *0*
⟨*proof*⟩

lemma *smult-1-left* [*simp*]: *smult* (*1*::′*a*::*comm-semiring-1*) *p* = *p*
⟨*proof*⟩

lemma *smult-add-right*:
smult a (*p* + *q*) = *smult a p* + *smult a q*
⟨*proof*⟩

lemma *smult-add-left*:
smult (*a* + *b*) *p* = *smult a p* + *smult b p*
⟨*proof*⟩

lemma *smult-minus-right* [*simp*]:
smult (*a*::′*a*::*comm-ring*) (− *p*) = − *smult a p*
⟨*proof*⟩

lemma *smult-minus-left* [*simp*]:
smult (− *a*::′*a*::*comm-ring*) *p* = − *smult a p*
⟨*proof*⟩

lemma *smult-diff-right*:
smult (*a*::′*a*::*comm-ring*) (*p* − *q*) = *smult a p* − *smult a q*
⟨*proof*⟩

lemma *smult-diff-left*:
smult (*a* − *b*::′*a*::*comm-ring*) *p* = *smult a p* − *smult b p*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *smult-monom*: *smult a* (*monom b n*) = *monom* (*a* ∗ *b*) *n*

⟨*proof*⟩

lemma *degree-smult-eq* [*simp*]:
fixes *a* :: *'a::idom*
shows *degree* (*smult a p*) = (*if a* = *0 then 0 else degree p*)
⟨*proof*⟩

lemma *smult-eq-0-iff* [*simp*]:
fixes *a* :: *'a::idom*
shows *smult a p* = *0* ⟷ *a* = *0* ∨ *p* = *0*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *mult-pCons-left* [*simp*]:
pCons a p ∗ *q* = *smult a q* + *pCons 0* (*p* ∗ *q*)
⟨*proof*⟩

lemma *mult-poly-0-right*: *p* ∗ (*0::'a poly*) = *0*
⟨*proof*⟩

lemma *mult-pCons-right* [*simp*]:
p ∗ *pCons a q* = *smult a p* + *pCons 0* (*p* ∗ *q*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *mult-smult-right* [*simp*]:
p ∗ *smult a q* = *smult a* (*p* ∗ *q*)
⟨*proof*⟩

lemma *mult-poly-add-left*:

fixes *p q r* :: *'a poly*
shows (*p* + *q*) ∗ *r* = *p* ∗ *r* + *q* ∗ *r*
⟨*proof*⟩

instance
⟨*proof*⟩

end

instance *poly* :: (*comm-semiring-0-cancel*) *comm-semiring-0-cancel* ⟨*proof*⟩

lemma *coeff-mult*:
coeff (*p* ∗ *q*) *n* = ($\sum$ *i*≤*n*. *coeff p i* ∗ *coeff q* (*n*−*i*))
⟨*proof*⟩

lemma *degree-mult-le*: *degree* (*p* ∗ *q*) ≤ *degree p* + *degree q*
⟨*proof*⟩

lemma *mult-monom*: *monom a m* ∗ *monom b n* = *monom* (*a* ∗ *b*) (*m* + *n*)
⟨*proof*⟩

instantiation *poly* :: (*comm-semiring-1*) *comm-semiring-1*
begin

definition *one-poly-def*: *1* = *pCons 1 0*

instance
⟨*proof*⟩

end

instance *poly* :: (*comm-ring*) *comm-ring* ⟨*proof*⟩

instance *poly* :: (*comm-ring-1*) *comm-ring-1* ⟨*proof*⟩

lemma *coeff-1* [*simp*]: *coeff 1 n* = (*if n* = *0 then 1 else 0*)
⟨*proof*⟩

lemma *monom-eq-1* [*simp*]:
monom 1 0 = *1*
⟨*proof*⟩

lemma *degree-1* [*simp*]: *degree 1* = *0*
⟨*proof*⟩

lemma *coeffs-1-eq* [*simp*, *code abstract*]:
coeffs 1 = [*1*]
⟨*proof*⟩

lemma *degree-power-le*:
degree (*p* ^ *n*) ≤ *degree* *p* ∗ *n*
⟨*proof*⟩

lemma *poly-smult* [*simp*]:
poly (*smult a p*) *x* = *a* ∗ *poly p x*
⟨*proof*⟩

lemma *poly-mult* [*simp*]:
poly (*p* ∗ *q*) *x* = *poly p x* ∗ *poly q x*
⟨*proof*⟩

lemma *poly-1* [*simp*]:
poly 1 x = *1*
⟨*proof*⟩

lemma *poly-power* [*simp*]:
fixes *p* :: *'a*::{*comm-semiring-1*} *poly*
shows *poly* (*p* ^ *n*) *x* = *poly p x* ^ *n*
⟨*proof*⟩

lemma *poly-setprod*: *poly* (∏ *k*∈*A*. *p k*) *x* = (∏ *k*∈*A*. *poly* (*p k*) *x*)
⟨*proof*⟩

lemma *degree-setprod-setsum-le*: *finite S* ⟹ *degree* (*setprod f S*) ≤ *setsum* (*degree o f*) *S*
⟨*proof*⟩

14.14 Conversions from natural numbers

lemma *of-nat-poly*: *of-nat n* = [:*of-nat n* :: *'a* :: *comm-semiring-1*:]
⟨*proof*⟩

lemma *degree-of-nat* [*simp*]: *degree* (*of-nat n*) = *0*
⟨*proof*⟩

lemma *degree-numeral* [*simp*]: *degree* (*numeral n*) = *0*
⟨*proof*⟩

lemma *numeral-poly*: *numeral n* = [:*numeral n*:]
⟨*proof*⟩

14.15 Lemmas about divisibility

lemma *dvd-smult*: *p dvd q* ⟹ *p dvd smult a q*
⟨*proof*⟩

lemma *dvd-smult-cancel*:
fixes *a* :: *'a* :: *field*
shows *p dvd smult a q* ⟹ *a* ≠ *0* ⟹ *p dvd q*

⟨*proof*⟩

lemma *dvd-smult-iff*:
fixes *a* :: *'a::field*
shows $a \neq 0 \Longrightarrow p\ dvd\ smult\ a\ q \longleftrightarrow p\ dvd\ q$
⟨*proof*⟩

lemma *smult-dvd-cancel*:
$smult\ a\ p\ dvd\ q \Longrightarrow p\ dvd\ q$
⟨*proof*⟩

lemma *smult-dvd*:
fixes *a* :: *'a::field*
shows $p\ dvd\ q \Longrightarrow a \neq 0 \Longrightarrow smult\ a\ p\ dvd\ q$
⟨*proof*⟩

lemma *smult-dvd-iff*:
fixes *a* :: *'a::field*
shows $smult\ a\ p\ dvd\ q \longleftrightarrow (if\ a = 0\ then\ q = 0\ else\ p\ dvd\ q)$
⟨*proof*⟩

14.16 Polynomials form an integral domain

lemma *coeff-mult-degree-sum*:
$coeff\ (p * q)\ (degree\ p + degree\ q) =$
$coeff\ p\ (degree\ p) * coeff\ q\ (degree\ q)$
⟨*proof*⟩

instance *poly* :: (*idom*) *idom*
⟨*proof*⟩

lemma *degree-mult-eq*:
fixes *p q* :: *'a::semidom poly*
shows $[\![p \neq 0;\ q \neq 0]\!] \Longrightarrow degree\ (p * q) = degree\ p + degree\ q$
⟨*proof*⟩

lemma *degree-mult-right-le*:
fixes *p q* :: *'a::semidom poly*
assumes $q \neq 0$
shows $degree\ p \leq degree\ (p * q)$
⟨*proof*⟩

lemma *coeff-degree-mult*:
fixes *p q* :: *'a::semidom poly*
shows $coeff\ (p * q)\ (degree\ (p * q)) =$
$coeff\ q\ (degree\ q) * coeff\ p\ (degree\ p)$
⟨*proof*⟩

lemma *dvd-imp-degree-le*:

fixes $p\ q :: 'a::semidom\ poly$
shows $[\![p\ dvd\ q;\ q \neq 0]\!] \Longrightarrow degree\ p \leq degree\ q$
⟨*proof*⟩

lemma *divides-degree*:
assumes *pq*: $p\ dvd\ (q :: 'a :: semidom\ poly)$
shows $degree\ p \leq degree\ q \lor q = 0$
⟨*proof*⟩

14.17 Polynomials form an ordered integral domain

definition $pos\text{-}poly :: 'a::linordered\text{-}idom\ poly \Rightarrow bool$
where
$pos\text{-}poly\ p \longleftrightarrow 0 < coeff\ p\ (degree\ p)$

lemma *pos-poly-pCons*:
$pos\text{-}poly\ (pCons\ a\ p) \longleftrightarrow pos\text{-}poly\ p \lor (p = 0 \land 0 < a)$
⟨*proof*⟩

lemma *not-pos-poly-0* [*simp*]: $\neg\ pos\text{-}poly\ 0$
⟨*proof*⟩

lemma *pos-poly-add*: $[\![pos\text{-}poly\ p;\ pos\text{-}poly\ q]\!] \Longrightarrow pos\text{-}poly\ (p + q)$
⟨*proof*⟩

lemma *pos-poly-mult*: $[\![pos\text{-}poly\ p;\ pos\text{-}poly\ q]\!] \Longrightarrow pos\text{-}poly\ (p * q)$
⟨*proof*⟩

lemma *pos-poly-total*: $p = 0 \lor pos\text{-}poly\ p \lor pos\text{-}poly\ (-\ p)$
⟨*proof*⟩

lemma *last-coeffs-eq-coeff-degree*:
$p \neq 0 \Longrightarrow last\ (coeffs\ p) = coeff\ p\ (degree\ p)$
⟨*proof*⟩

lemma *pos-poly-coeffs* [*code*]:
$pos\text{-}poly\ p \longleftrightarrow (let\ as = coeffs\ p\ in\ as \neq [] \land last\ as > 0)$ (**is** $?P \longleftrightarrow ?Q$)
⟨*proof*⟩

instantiation *poly* :: (*linordered-idom*) *linordered-idom*
begin

definition
$x < y \longleftrightarrow pos\text{-}poly\ (y - x)$

definition
$x \leq y \longleftrightarrow x = y \lor pos\text{-}poly\ (y - x)$

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*⟩

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* :: $'a::comm\text{-}semiring\text{-}0\ poly \Rightarrow 'a \Rightarrow 'a\ poly \times 'a$
where
$synthetic\text{-}divmod\ p\ c = fold\text{-}coeffs\ (\lambda a\ (q,\ r).\ (pCons\ r\ q,\ a + c * r))\ p\ (0,\ 0)$

definition *synthetic-div* :: $'a::comm\text{-}semiring\text{-}0\ poly \Rightarrow 'a \Rightarrow 'a\ poly$
where
$synthetic\text{-}div\ p\ c = fst\ (synthetic\text{-}divmod\ p\ c)$

lemma *synthetic-divmod-0* [*simp*]:
$synthetic\text{-}divmod\ 0\ c = (0,\ 0)$
⟨*proof*⟩

lemma *synthetic-divmod-pCons* [*simp*]:
$synthetic\text{-}divmod\ (pCons\ a\ p)\ c = (\lambda(q,\ r).\ (pCons\ r\ q,\ a + c * r))\ (synthetic\text{-}divmod\ p\ c)$
⟨*proof*⟩

lemma *synthetic-div-0* [*simp*]:
$synthetic\text{-}div\ 0\ c = 0$
⟨*proof*⟩

lemma *synthetic-div-unique-lemma*: $smult\ c\ p = pCons\ a\ p \Longrightarrow p = 0$
⟨*proof*⟩

lemma *snd-synthetic-divmod*:
$snd\ (synthetic\text{-}divmod\ p\ c) = poly\ p\ c$
⟨*proof*⟩

lemma *synthetic-div-pCons* [*simp*]:
$synthetic\text{-}div\ (pCons\ a\ p)\ c = pCons\ (poly\ p\ c)\ (synthetic\text{-}div\ p\ c)$
⟨*proof*⟩

lemma *synthetic-div-eq-0-iff*:
$synthetic\text{-}div\ p\ c = 0 \longleftrightarrow degree\ p = 0$

⟨proof⟩

lemma *degree-synthetic-div*:
degree (*synthetic-div p c*) = *degree p* − *1*
⟨proof⟩

lemma *synthetic-div-correct*:
p + *smult c* (*synthetic-div p c*) = *pCons* (*poly p c*) (*synthetic-div p c*)
⟨proof⟩

lemma *synthetic-div-unique*:
p + *smult c q* = *pCons r q* ⟹ *r* = *poly p c* ∧ *q* = *synthetic-div p c*
⟨proof⟩

lemma *synthetic-div-correct′*:
fixes *c* :: ′*a*::*comm-ring-1*
shows [:−*c*, *1*:] ∗ *synthetic-div p c* + [:*poly p c*:] = *p*
⟨proof⟩

lemma *poly-eq-0-iff-dvd*:
fixes *c* :: ′*a*::*idom*
shows *poly p c* = *0* ⟷ [:−*c*, *1*:] *dvd p*
⟨proof⟩

lemma *dvd-iff-poly-eq-0*:
fixes *c* :: ′*a*::*idom*
shows [:*c*, *1*:] *dvd p* ⟷ *poly p* (−*c*) = *0*
⟨proof⟩

lemma *poly-roots-finite*:
fixes *p* :: ′*a*::*idom poly*
shows *p* ≠ *0* ⟹ *finite* {*x*. *poly p x* = *0*}
⟨proof⟩

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⟩

lemma *poly-all-0-iff-0*:
fixes *p* :: ′*a*::{*ring-char-0*, *idom*} *poly*
shows (∀ *x*. *poly p x* = *0*) ⟷ *p* = *0*
⟨proof⟩

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 ⟷

$x = q * y + r \wedge (\textit{if } y = 0 \textit{ then } q = 0 \textit{ else } r = 0 \vee \textit{degree } r < \textit{degree } y)$

lemma *pdivmod-rel-0*:
pdivmod-rel 0 y 0 0
⟨*proof*⟩

lemma *pdivmod-rel-by-0*:
pdivmod-rel x 0 0 x
⟨*proof*⟩

lemma *eq-zero-or-degree-less*:
assumes *degree p* ≤ *n* **and** *coeff p n* = *0*
shows *p* = *0* ∨ *degree p* < *n*
⟨*proof*⟩

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*⟩

lemma *pdivmod-rel-exists*: ∃ *q r*. *pdivmod-rel x y q r*
⟨*proof*⟩

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*⟩

lemma *pdivmod-rel-0-iff*: *pdivmod-rel 0 y q r* ⟷ *q* = *0* ∧ *r* = *0*
⟨*proof*⟩

lemma *pdivmod-rel-by-0-iff*: *pdivmod-rel x 0 q r* ⟷ *q* = *0* ∧ *r* = *x*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *mod-poly-eq*:
pdivmod-rel x y q r ⟹ *x mod y* = *r*
⟨*proof*⟩

lemma *pdivmod-rel*:
pdivmod-rel x y (*x div y*) (*x mod y*)
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *is-unit-monom-0*:
fixes *a* :: *'a::field*
assumes *a* ≠ *0*
shows *is-unit* (*monom a 0*)
⟨*proof*⟩

lemma *is-unit-triv*:
fixes *a* :: *'a::field*
assumes *a* ≠ *0*
shows *is-unit* [:*a*:]
⟨*proof*⟩

lemma *is-unit-iff-degree*:
assumes *p* ≠ *0*
shows *is-unit p* ⟷ *degree p* = *0* (**is** *?P* ⟷ *?Q*)
⟨*proof*⟩

lemma *is-unit-pCons-iff*:
is-unit (*pCons a p*) ⟷ *p* = *0* ∧ *a* ≠ *0* (**is** *?P* ⟷ *?Q*)
⟨*proof*⟩

lemma *is-unit-monom-trival*:
fixes *p* :: *'a::field poly*
assumes *is-unit p*
shows *monom* (*coeff p* (*degree p*)) *0* = *p*
⟨*proof*⟩

lemma *is-unit-polyE*:
assumes *is-unit p*
obtains *a* **where** *p* = *monom a 0* **and** *a* ≠ *0*

⟨*proof*⟩

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*⟩

end

lemma *unit-factor-monom* [*simp*]:
  unit-factor (*monom a n*) =
    (*if a* = *0 then 0 else monom a 0*)
  ⟨*proof*⟩

lemma *unit-factor-pCons* [*simp*]:
  unit-factor (*pCons a p*) =
    (*if p* = *0 then monom a 0 else unit-factor p*)
  ⟨*proof*⟩

lemma *normalize-monom* [*simp*]:
  normalize (*monom a n*) =
    (*if a* = *0 then 0 else monom 1 n*)
  ⟨*proof*⟩

lemma *degree-mod-less*:
  $y \neq 0 \Longrightarrow x \; mod \; y = 0 \lor degree \; (x \; mod \; y) < degree \; y$
  ⟨*proof*⟩

lemma *div-poly-less*: $degree \; x < degree \; y \Longrightarrow x \; div \; y = 0$
⟨*proof*⟩

lemma *mod-poly-less*: $degree \; x < degree \; y \Longrightarrow x \; mod \; y = x$
⟨*proof*⟩

lemma *pdivmod-rel-smult-left*:
  pdivmod-rel x y q r
    ⟹ *pdivmod-rel* (*smult a x*) *y* (*smult a q*) (*smult a r*)
  ⟨*proof*⟩

lemma *div-smult-left*: (*smult a x*) *div y* = *smult a* (*x div y*)
  ⟨*proof*⟩

lemma *mod-smult-left*: (*smult a x*) *mod y* = *smult a* (*x mod y*)
⟨*proof*⟩

lemma *poly-div-minus-left* [*simp*]:
fixes *x y* :: *'a::field poly*
shows (− *x*) *div y* = − (*x div y*)
⟨*proof*⟩

lemma *poly-mod-minus-left* [*simp*]:
fixes *x y* :: *'a::field poly*
shows (− *x*) *mod y* = − (*x mod y*)
⟨*proof*⟩

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'*)
⟨*proof*⟩

lemma *poly-div-add-left*:
fixes *x y z* :: *'a::field poly*
shows (*x* + *y*) *div z* = *x div z* + *y div z*
⟨*proof*⟩

lemma *poly-mod-add-left*:
fixes *x y z* :: *'a::field poly*
shows (*x* + *y*) *mod z* = *x mod z* + *y mod z*
⟨*proof*⟩

lemma *poly-div-diff-left*:
fixes *x y z* :: *'a::field poly*
shows (*x* − *y*) *div z* = *x div z* − *y div z*
⟨*proof*⟩

lemma *poly-mod-diff-left*:
fixes *x y z* :: *'a::field poly*
shows (*x* − *y*) *mod z* = *x mod z* − *y mod z*
⟨*proof*⟩

lemma *pdivmod-rel-smult-right*:
⟦*a* ≠ *0*; *pdivmod-rel x y q r*⟧
⟹ *pdivmod-rel x* (*smult a y*) (*smult* (*inverse a*) *q*) *r*
⟨*proof*⟩

lemma *div-smult-right*:
a ≠ *0* ⟹ *x div* (*smult a y*) = *smult* (*inverse a*) (*x div y*)
⟨*proof*⟩

lemma *mod-smult-right*: *a* ≠ *0* ⟹ *x mod* (*smult a y*) = *x mod y*

⟨*proof*⟩

lemma *poly-div-minus-right* [*simp*]:
fixes *x y* :: *'a::field poly*
shows $x\ div\ (-\ y) = -\ (x\ div\ y)$
⟨*proof*⟩

lemma *poly-mod-minus-right* [*simp*]:
fixes *x y* :: *'a::field poly*
shows $x\ mod\ (-\ y) = x\ mod\ y$
⟨*proof*⟩

lemma *pdivmod-rel-mult*:
⟦*pdivmod-rel x y q r*; *pdivmod-rel q z q' r'*⟧
⟹ *pdivmod-rel x* $(y * z)$ *q'* $(y * r' + r)$
⟨*proof*⟩

lemma *poly-div-mult-right*:
fixes *x y z* :: *'a::field poly*
shows $x\ div\ (y * z) = (x\ div\ y)\ div\ z$
⟨*proof*⟩

lemma *poly-mod-mult-right*:
fixes *x y z* :: *'a::field poly*
shows $x\ mod\ (y * z) = y * (x\ div\ y\ mod\ z) + x\ mod\ y$
⟨*proof*⟩

lemma *mod-pCons*:
fixes *a* **and** *x*
assumes *y*: $y \neq 0$
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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *mod-poly-code* [*code*]:
p mod q = *snd* (*pdivmod p q*)
⟨*proof*⟩

lemma *pdivmod-0*:
pdivmod 0 q = (*0*, *0*)
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *degree-linear-power*:
fixes *a* :: ′*a*::*comm-semiring-1*
shows *degree* ([:*a*, *1*:] ^ *n*) = *n*
⟨*proof*⟩

lemma *order-1*: [:−*a*, *1*:] ^ *order a p dvd p*
⟨*proof*⟩

lemma *order-2*: *p* ≠ *0* ⟹ ¬ [:−*a*, *1*:] ^ *Suc* (*order a p*) *dvd p*
⟨*proof*⟩

lemma *order*:
p ≠ *0* ⟹ [:−*a*, *1*:] ^ *order a p dvd p* ∧ ¬ [:−*a*, *1*:] ^ *Suc* (*order a p*) *dvd p*
⟨*proof*⟩

lemma *order-degree*:
assumes *p*: *p* ≠ *0*
shows *order a p* ≤ *degree p*
⟨*proof*⟩

lemma *order-root*: *poly p a* = *0* ⟷ *p* = *0* ∨ *order a p* ≠ *0*

⟨proof⟩

lemma *order-0I*: $poly\ p\ a \neq 0 \Longrightarrow order\ a\ p = 0$
⟨proof⟩

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⟩

lemma *dropWhile-replicate-append*:
dropWhile (op= a) (replicate n a @ ys) = dropWhile (op= a) ys
⟨proof⟩

lemma *Poly-append-replicate-0*: *Poly (xs @ replicate n 0) = Poly xs*
⟨proof⟩

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⟩

14.22 Composition of polynomials

definition *pcompose* :: $'a::comm\text{-}semiring\text{-}0\ poly \Rightarrow 'a\ poly \Rightarrow 'a\ poly$
where
*pcompose p q = fold-coeffs (λa c. [:a:] + q * c) p 0*

notation *pcompose* (**infixl** $\circ_p$ *71*)

lemma *pcompose-0* [*simp*]:
pcompose 0 q = 0
⟨proof⟩

lemma *pcompose-pCons*:
*pcompose (pCons a p) q = [:a:] + q * pcompose p q*
⟨proof⟩

lemma *pcompose-1*:

fixes *p* :: *'a* :: *comm-semiring-1 poly*
shows *pcompose 1 p* = *1*
⟨*proof*⟩

lemma *poly-pcompose*:
poly (*pcompose p q*) *x* = *poly p* (*poly q x*)
⟨*proof*⟩

lemma *degree-pcompose-le*:
degree (*pcompose p q*) ≤ *degree p* ∗ *degree q*
⟨*proof*⟩

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*⟩

lemma *pcompose-uminus*:
fixes *p r* :: *'a* :: *comm-ring poly*
shows *pcompose* (−*p*) *r* = −*pcompose p r*
⟨*proof*⟩

lemma *pcompose-diff*:
fixes *p q r* :: *'a* :: *comm-ring poly*
shows *pcompose* (*p* − *q*) *r* = *pcompose p r* − *pcompose q r*
⟨*proof*⟩

lemma *pcompose-smult*:
fixes *p r* :: *'a* :: *comm-semiring-0 poly*
shows *pcompose* (*smult a p*) *r* = *smult a* (*pcompose p r*)
⟨*proof*⟩

lemma *pcompose-mult*:
fixes *p q r* :: *'a* :: *comm-semiring-0 poly*
shows *pcompose* (*p* ∗ *q*) *r* = *pcompose p r* ∗ *pcompose q r*
⟨*proof*⟩

lemma *pcompose-assoc*:
pcompose p (*pcompose q r* :: *'a* :: *comm-semiring-0 poly*) =
pcompose (*pcompose p q*) *r*
⟨*proof*⟩

lemma *pcompose-idR*[*simp*]:
fixes *p* :: *'a* :: *comm-semiring-1 poly*
shows *pcompose p* [: *0*, *1* :] = *p*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *pcompose-const*[*simp*]:*pcompose* [:*a*:] *q* = [:*a*:] ⟨*proof*⟩

lemma *pcompose-0'*: *pcompose p 0* = [:*coeff p 0*:]
⟨*proof*⟩

lemma *degree-pcompose*:
fixes *p q*:: *'a*::*semidom poly*
shows *degree* (*pcompose p q*) = *degree p* ∗ *degree q*
⟨*proof*⟩

lemma *pcompose-eq-0*:
fixes *p q*:: *'a* :: *semidom poly*
assumes *pcompose p q* = *0 degree q* > *0*
shows *p* = *0*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *lead-coeff-0*[*simp*]:*lead-coeff 0* =*0*
⟨*proof*⟩

lemma *lead-coeff-mult*:
fixes *p q*::*'a* ::*idom poly*
shows *lead-coeff* (*p* ∗ *q*) = *lead-coeff p* ∗ *lead-coeff q*
⟨*proof*⟩

lemma *lead-coeff-add-le*:
assumes *degree p* < *degree q*
shows *lead-coeff* (*p*+*q*) = *lead-coeff q*
⟨*proof*⟩

lemma *lead-coeff-minus*:
lead-coeff (−*p*) = − *lead-coeff p*
⟨*proof*⟩

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* ^ (*degree p*)
⟨*proof*⟩

lemma *lead-coeff-smult*:
lead-coeff (*smult c p* :: *'a* :: *idom poly*) = *c* ∗ *lead-coeff p*
⟨*proof*⟩

lemma *lead-coeff-1* [*simp*]: *lead-coeff 1* = *1*
⟨*proof*⟩

lemma *lead-coeff-of-nat* [*simp*]:
lead-coeff (*of-nat n*) = (*of-nat n* :: *'a* :: {*comm-semiring-1*,*semiring-char-0*})
⟨*proof*⟩

lemma *lead-coeff-numeral* [*simp*]:
lead-coeff (*numeral n*) = *numeral n*
⟨*proof*⟩

lemma *lead-coeff-power*:
lead-coeff (*p* ^ *n* :: *'a* :: *idom poly*) = *lead-coeff p* ^ *n*
⟨*proof*⟩

lemma *lead-coeff-nonzero*: *p* ≠ *0* ⟹ *lead-coeff p* ≠ *0*
⟨*proof*⟩

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*))
⟨*proof*⟩

termination *pderiv*
⟨*proof*⟩

lemma *pderiv-0* [*simp*]:
pderiv 0 = *0*
⟨*proof*⟩

lemma *pderiv-pCons*:
pderiv (*pCons a p*) = *p* + *pCons 0* (*pderiv p*)
⟨*proof*⟩

lemma *pderiv-1* [*simp*]: *pderiv 1* = *0*

⟨proof⟩

lemma *pderiv-of-nat* [*simp*]: *pderiv* (*of-nat n*) = *0*
and *pderiv-numeral* [*simp*]: *pderiv* (*numeral m*) = *0*
⟨proof⟩

lemma *coeff-pderiv*: *coeff* (*pderiv p*) *n* = *of-nat* (*Suc n*) ∗ *coeff p* (*Suc n*)
⟨proof⟩

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⟩

lemma *map-upt-Suc*: *map f* [*0* ..< *Suc n*] = *f 0* # *map* (λ *i*. *f* (*Suc i*)) [*0* ..< *n*]
⟨proof⟩

lemma *coeffs-pderiv-code* [*code abstract*]:
coeffs (*pderiv p*) = *pderiv-coeffs* (*coeffs p*) ⟨proof⟩

context
assumes *SORT-CONSTRAINT*('*a*::{*semidom*, *semiring-char-0*})
begin

lemma *pderiv-eq-0-iff*:
pderiv (*p* :: '*a poly*) = *0* ⟷ *degree p* = *0*
⟨proof⟩

lemma *degree-pderiv*: *degree* (*pderiv* (*p* :: '*a poly*)) = *degree p* − *1*
⟨proof⟩

lemma *not-dvd-pderiv*:
assumes *degree* (*p* :: '*a poly*) ≠ *0*
shows ¬ *p dvd pderiv p*
⟨proof⟩

lemma *dvd-pderiv-iff* [*simp*]: (*p* :: '*a poly*) *dvd pderiv p* ⟷ *degree p* = *0*
⟨proof⟩

end

lemma *pderiv-singleton* [*simp*]: *pderiv* [:*a*:] = *0*

⟨proof⟩

lemma *pderiv-add*: *pderiv* (*p* + *q*) = *pderiv p* + *pderiv q*
⟨proof⟩

lemma *pderiv-minus*: *pderiv* (− *p* :: ′*a* :: *idom poly*) = − *pderiv p*
⟨proof⟩

lemma *pderiv-diff*: *pderiv* (*p* − *q*) = *pderiv p* − *pderiv q*
⟨proof⟩

lemma *pderiv-smult*: *pderiv* (*smult a p*) = *smult a* (*pderiv p*)
⟨proof⟩

lemma *pderiv-mult*: *pderiv* (*p* ∗ *q*) = *p* ∗ *pderiv q* + *q* ∗ *pderiv p*
⟨proof⟩

lemma *pderiv-power-Suc*:
pderiv (*p* ˆ *Suc n*) = *smult* (*of-nat* (*Suc n*)) (*p* ˆ *n*) ∗ *pderiv p*
⟨proof⟩

lemma *pderiv-setprod*: *pderiv* (*setprod f* (*as*)) =
($\sum$ *a* ∈ *as*. *setprod f* (*as* − {*a*}) ∗ *pderiv* (*f a*))
⟨proof⟩

lemma *DERIV-pow2*: *DERIV* (%*x*. *x* ˆ *Suc n*) *x* :> *real* (*Suc n*) ∗ (*x* ˆ *n*)
⟨proof⟩
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*
⟨proof⟩

lemma *poly-DERIV* [*simp*]: *DERIV* (%*x*. *poly p x*) *x* :> *poly* (*pderiv p*) *x*
⟨proof⟩

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⟩

Consequences of the derivative theorem above

lemma *poly-differentiable*[*simp*]: (%*x*. *poly p x*) *differentiable* (*at x*::*real filter*)
⟨proof⟩

lemma *poly-isCont*[*simp*]: *isCont* (%*x*. *poly p x*) (*x*::*real*)
⟨proof⟩

lemma *poly-IVT-pos*: [| $a < b$; *poly p* $(a::real) < 0$; $0 <$ *poly p b* |]
    ==> $\exists x.\ a < x$ & $x < b$ & (*poly p x* $= 0$)
⟨*proof*⟩

lemma *poly-IVT-neg*: [| $(a::real) < b$; $0 <$ *poly p a*; *poly p b* < 0 |]
    ==> $\exists x.\ a < x$ & $x < b$ & (*poly p x* $= 0$)
⟨*proof*⟩

lemma *poly-IVT*:
  fixes *p*::*real poly*
  assumes $a<b$ **and** *poly p a* $*$ *poly p b* < 0
  shows $\exists x>a.\ x < b \wedge$ *poly p x* $= 0$
⟨*proof*⟩

lemma *poly-MVT*: $(a::real) < b$ ==>
   $\exists x.\ a < x$ & $x < b$ & (*poly p b* $-$ *poly p a* $= (b - a) *$ *poly* (*pderiv p*) x)
⟨*proof*⟩

lemma *poly-MVT'*:
  assumes $\{min\ a\ b..max\ a\ b\} \subseteq A$
  shows $\exists x \in A.$ *poly p b* $-$ *poly p a* $= (b - a) *$ *poly* (*pderiv p*) $(x::real)$
⟨*proof*⟩

lemma *poly-pinfty-gt-lc*:
  fixes *p*:: *real poly*
  assumes *lead-coeff p* > 0
  shows $\exists\ n.\ \forall\ x \geq n.$ *poly p x* $\geq$ *lead-coeff p* ⟨*proof*⟩

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* $\Rightarrow$ *bool* **where**
  algebraic x $\longleftrightarrow$ ($\exists p.$ ($\forall i.$ *coeff p i* $\in \mathbb{Z}$) $\wedge\ p \neq 0 \wedge$ *poly p x* $= 0$)

lemma *algebraicI*:
  assumes $\bigwedge i.$ *coeff p i* $\in \mathbb{Z}$ $p \neq 0$ *poly p x* $= 0$
  shows *algebraic x*
  ⟨*proof*⟩

lemma *algebraicE*:
  assumes *algebraic x*
  obtains *p* **where** $\bigwedge i.$ *coeff p i* $\in \mathbb{Z}$ *p* $\neq$ *0 poly p x* = *0*
  ⟨*proof*⟩

lemma *quotient-of-denom-pos'*: *snd* (*quotient-of x*) > *0*
  ⟨*proof*⟩

lemma *of-int-div-in-Ints*:
  b dvd a ⟹ *of-int a div of-int b* ∈ (ℤ :: *'a* :: *ring-div set*)
⟨*proof*⟩

lemma *of-int-divide-in-Ints*:
  b dvd a ⟹ *of-int a* / *of-int b* ∈ (ℤ :: *'a* :: *field set*)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *order-decomp*:
  assumes *p* ≠ *0*
  shows ∃ *q*. *p* = [:− *a*, *1*:] ˆ *order a p* ∗ *q* ∧ ¬ [:− *a*, *1*:] *dvd q*
⟨*proof*⟩

lemma *order-pderiv*:
  ⟦*pderiv p* ≠ *0*; *order a* (*p* :: *'a* :: *field-char-0 poly*) ≠ *0*⟧ ⟹

(*order a p* = *Suc* (*order a* (*pderiv p*)))
⟨*proof*⟩

lemma *order-mult*: $p * q \neq 0 \Longrightarrow order\ a\ (p * q) = order\ a\ p + order\ a\ q$
⟨*proof*⟩

lemma *order-smult*:
assumes $c \neq 0$
shows *order x* (*smult c p*) = *order x p*
⟨*proof*⟩

lemma *order-1-eq-0* [*simp*]:*order x 1* = *0*
⟨*proof*⟩

lemma *order-power-n-n*: *order a* ([:−*a*,*1*:] ˆ*n*)=*n*
⟨*proof*⟩

Now justify the standard squarefree decomposition, i.e. f / gcd(f,f').

lemma *order-divides*: [:−*a*, *1*:] ˆ *n dvd p* ⟷ *p* = *0* ∨ *n* ≤ *order a p*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*}:]
⟨*proof*⟩

lemma *rsquarefree-roots*:
  fixes *p* :: *'a* :: *field-char-0 poly*
  shows *rsquarefree p* = (∀ *a*. ¬(*poly p a* = *0* ∧ *poly* (*pderiv p*) *a* = *0*))
⟨*proof*⟩

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*⟩

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⟩

lemma *dvd-euclidean-size-eq-imp-dvd*:
assumes $a \neq 0$ **and** *b-dvd-a*: *b dvd a* **and** *size-eq*: *euclidean-size a* = *euclidean-size b*
shows *a dvd b*
⟨proof⟩

function *gcd-eucl* :: $'a \Rightarrow 'a \Rightarrow 'a$
where
gcd-eucl a b = (*if b* = *0 then normalize a else gcd-eucl b* (*a mod b*))
⟨proof⟩
termination
⟨proof⟩

declare *gcd-eucl.simps* [*simp del*]

lemma *gcd-eucl-induct* [*case-names zero mod*]:
assumes *H1*: $\bigwedge b.\ P\ b\ 0$
and *H2*: $\bigwedge a\ b.\ b \neq 0 \Longrightarrow P\ b\ (a\ mod\ b) \Longrightarrow P\ a\ b$
shows *P a b*
⟨proof⟩

definition *lcm-eucl* :: $'a \Rightarrow 'a \Rightarrow 'a$
where
lcm-eucl a b = *normalize* (*a* ∗ *b*) *div gcd-eucl a b*

definition *Lcm-eucl* :: $'a\ set \Rightarrow 'a$ — 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* :: $'a\ set \Rightarrow 'a$
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*
⟨proof⟩

lemma *gcd-eucl-0-left*:
gcd-eucl 0 a = *normalize a*
⟨proof⟩

lemma *gcd-eucl-non-0*:
  $b \neq 0 \Longrightarrow$ *gcd-eucl a b* = *gcd-eucl b* (*a mod b*)
  ⟨*proof*⟩

lemma *gcd-eucl-dvd1* [*iff*]: *gcd-eucl a b dvd a*
  and *gcd-eucl-dvd2* [*iff*]: *gcd-eucl a b dvd b*
  ⟨*proof*⟩

lemma *normalize-gcd-eucl* [*simp*]:
  normalize (*gcd-eucl a b*) = *gcd-eucl a b*
  ⟨*proof*⟩

lemma *gcd-eucl-greatest*:
  fixes *k a b* :: *'a*
  shows *k dvd a* $\Longrightarrow$ *k dvd b* $\Longrightarrow$ *k dvd gcd-eucl a b*
⟨*proof*⟩

lemma *eq-gcd-euclI*:
  fixes *gcd* :: *'a* $\Rightarrow$ *'a* $\Rightarrow$ *'a*
  assumes $\bigwedge$*a b*. *gcd a b dvd a* $\bigwedge$*a b*. *gcd a b dvd b* $\bigwedge$*a b*. *normalize* (*gcd a b*) = *gcd a b*
    $\bigwedge$*a b k*. *k dvd a* $\Longrightarrow$ *k dvd b* $\Longrightarrow$ *k dvd gcd a b*
  shows *gcd* = *gcd-eucl*
  ⟨*proof*⟩

lemma *gcd-eucl-zero* [*simp*]:
  gcd-eucl a b = *0* $\longleftrightarrow$ *a* = *0* $\wedge$ *b* = *0*
  ⟨*proof*⟩

lemma *dvd-Lcm-eucl* [*simp*]: *a* $\in$ *A* $\Longrightarrow$ *a dvd Lcm-eucl A*
  and *Lcm-eucl-least*: ($\bigwedge$*a*. *a* $\in$ *A* $\Longrightarrow$ *a dvd b*) $\Longrightarrow$ *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*⟩

lemma *normalize-Lcm-eucl* [*simp*]:
  normalize (*Lcm-eucl A*) = *Lcm-eucl A*
⟨*proof*⟩

lemma *eq-Lcm-euclI*:
  fixes *lcm* :: *'a set* $\Rightarrow$ *'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*
  ⟨*proof*⟩

end

class *euclidean-ring* = *euclidean-semiring* + *idom*
begin

subclass *ring-div* ⟨*proof*⟩

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*))
⟨*proof*⟩
termination ⟨*proof*⟩

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))
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *euclid-ext-left-0*:
euclid-ext 0 a = (*0*, *1 div unit-factor a*, *normalize a*)
⟨*proof*⟩

lemma *euclid-ext-correct'*:
case euclid-ext x y of (*a*,*b*,*c*) ⇒ *a* ∗ *x* + *b* ∗ *y* = *c* ∧ *c* = *gcd-eucl x y*
⟨*proof*⟩

lemma *euclid-ext-gcd-eucl*:
(*case euclid-ext x y of* (*a*,*b*,*c*) ⇒ *c*) = *gcd-eucl x y*
⟨*proof*⟩

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*

⟨proof⟩

lemma *euclid-ext′-0*: *euclid-ext′ a 0 = (1 div unit-factor a, 0)*
⟨proof⟩

lemma *euclid-ext′-left-0*: *euclid-ext′ 0 a = (0, 1 div unit-factor a)*
⟨proof⟩

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*
⟨proof⟩

subclass *semiring-Gcd*
⟨proof⟩

lemma *gcd-non-0*:
$b \neq 0 \Longrightarrow gcd\ a\ b = gcd\ b\ (a\ mod\ b)$
⟨proof⟩

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
⟨proof⟩

lemma *gcd-mod2* [*simp*]:
gcd a (b mod a) = gcd a b
⟨proof⟩

lemma *euclidean-size-gcd-le1* [*simp*]:
assumes $a \neq 0$
shows *euclidean-size (gcd a b)* $\leq$ *euclidean-size a*
⟨proof⟩

lemma *euclidean-size-gcd-le2* [*simp*]:
$b \neq 0 \Longrightarrow$ *euclidean-size (gcd a b)* $\leq$ *euclidean-size b*
⟨proof⟩

lemma *euclidean-size-gcd-less1*:
assumes $a \neq 0$ **and** $\neg a$ *dvd b*
shows *euclidean-size (gcd a b)* $<$ *euclidean-size a*

⟨*proof*⟩

lemma *euclidean-size-gcd-less2*:
assumes $b \neq 0$ **and** $\neg b$ *dvd* a
shows *euclidean-size* (*gcd a b*) < *euclidean-size b*
⟨*proof*⟩

lemma *euclidean-size-lcm-le1*:
assumes $a \neq 0$ **and** $b \neq 0$
shows *euclidean-size a* ≤ *euclidean-size* (*lcm a b*)
⟨*proof*⟩

lemma *euclidean-size-lcm-le2*:
$a \neq 0 \Longrightarrow b \neq 0 \Longrightarrow$ *euclidean-size b* ≤ *euclidean-size* (*lcm a b*)
⟨*proof*⟩

lemma *euclidean-size-lcm-less1*:
assumes $b \neq 0$ **and** $\neg b$ *dvd* a
shows *euclidean-size a* < *euclidean-size* (*lcm a b*)
⟨*proof*⟩

lemma *euclidean-size-lcm-less2*:
assumes $a \neq 0$ **and** $\neg a$ *dvd* b
shows *euclidean-size b* < *euclidean-size* (*lcm a b*)
⟨*proof*⟩

lemma *Lcm-eucl-set* [*code*]:
Lcm-eucl (*set xs*) = *foldl lcm-eucl 1 xs*
⟨*proof*⟩

lemma *Gcd-eucl-set* [*code*]:
Gcd-eucl (*set xs*) = *foldl gcd-eucl 0 xs*
⟨*proof*⟩

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* ⟨*proof*⟩
subclass *ring-gcd* ⟨*proof*⟩

lemma *euclid-ext-gcd* [*simp*]:
(*case euclid-ext a b of* (-, - , *t*) ⇒ *t*) = *gcd a b*
⟨*proof*⟩

lemma *euclid-ext-gcd'* [*simp*]:
  euclid-ext a b = (*r*, *s*, *t*) ⟹ *t* = *gcd a b*
  ⟨*proof*⟩

lemma *euclid-ext-correct*:
  case euclid-ext x y of (*a*,*b*,*c*) ⇒ *a* ∗ *x* + *b* ∗ *y* = *c* ∧ *c* = *gcd x y*
  ⟨*proof*⟩

lemma *euclid-ext'-correct*:
  fst (*euclid-ext' a b*) ∗ *a* + *snd* (*euclid-ext' a b*) ∗ *b* = *gcd a b*
  ⟨*proof*⟩

lemma *bezout*: ∃ *s t*. *s* ∗ *a* + *t* ∗ *b* = *gcd a b*
  ⟨*proof*⟩

end

15.1 Typical instances

instantiation *nat* :: *euclidean-semiring*
begin

definition [*simp*]:
  euclidean-size-nat = (*id* :: *nat* ⇒ *nat*)

instance ⟨*proof*⟩

end

instantiation *int* :: *euclidean-ring*
begin

definition [*simp*]:
  euclidean-size-int = (*nat* ∘ *abs* :: *int* ⇒ *nat*)

instance
⟨*proof*⟩

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* ^ *degree p*)

lemma *euclidean-size-poly-0* [*simp*]:

euclidean-size (*0*::*'a poly*) = *0*
⟨*proof*⟩

lemma *euclidean-size-poly-not-0* [*simp*]:
p ≠ *0* ⟹ *euclidean-size p* = *2* ^ *degree p*
⟨*proof*⟩

instance
⟨*proof*⟩

end

instance *nat* :: *euclidean-semiring-gcd*
⟨*proof*⟩

instance *int* :: *euclidean-ring-gcd*
⟨*proof*⟩

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 ⟨*proof*⟩
end

lemma *poly-gcd-monic*:
lead-coeff (*gcd x y*) = (*if x* = *0* ∧ *y* = *0 then 0 else 1*)
⟨*proof*⟩

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*⟩

lemma *poly-gcd-unique*:

fixes *d x y* :: *- poly*
assumes *dvd1*: *d dvd x* **and** *dvd2*: *d dvd y*
and *greatest*: $\bigwedge k.\ k\ dvd\ x \Longrightarrow k\ dvd\ y \Longrightarrow k\ dvd\ d$
and *monic*: *coeff d* (*degree d*) = (*if x* = *0* ∧ *y* = *0 then 0 else 1*)
shows *d* = *gcd x y*
⟨*proof*⟩

lemma *poly-gcd-code* [*code*]:
gcd x y = (*if y* = *0 then normalize x else gcd y* (*x mod* (*y* :: *- poly*)))
⟨*proof*⟩

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*⟩

end

class *factorial-semiring* = *normalization-semidom* +
assumes *finite-divisors*: *a* ≠ *0* ⟹ *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* ≠ *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*
⟨*proof*⟩

lemma *is-prime-not-zeroI*:
assumes *is-prime p*
shows *p* ≠ *0*
⟨*proof*⟩

lemma *is-prime-multD*:
  assumes *is-prime* $(a * b)$
  shows *is-unit* $a \vee$ *is-unit* b
⟨*proof*⟩

lemma *is-primeD2*:
  assumes *is-prime* p **and** a *dvd* p **and** $\neg$ *is-unit* a
  shows p *dvd* a
⟨*proof*⟩

lemma *is-prime-mult-unit-left*:
  assumes *is-prime* p
    and *is-unit* a
  shows *is-prime* $(a * p)$
⟨*proof*⟩

lemma *is-primeI2*:
  assumes $p \neq 0$
  assumes $\neg$ *is-unit* p
  assumes P: $\bigwedge a\ b.\ p$ *dvd* $a * b \Longrightarrow p$ *dvd* $a \vee p$ *dvd* b
  shows *is-prime* p
⟨*proof*⟩

lemma *not-is-prime-divisorE*:
  assumes $a \neq 0$ **and** $\neg$ *is-unit* a **and** $\neg$ *is-prime* a
  obtains b **where** b *dvd* a **and** $\neg$ *is-unit* b **and** $\neg$ a *dvd* b
⟨*proof*⟩

lemma *is-prime-normalize-iff* [*simp*]:
  is-prime (*normalize* p) $\longleftrightarrow$ *is-prime* p (**is** $?P \longleftrightarrow ?Q$)
⟨*proof*⟩

lemma *no-prime-divisorsI*:
  assumes $\bigwedge b.\ b$ *dvd* $a \Longrightarrow$ *normalize* $b = b \Longrightarrow \neg$ *is-prime* b
  shows *is-unit* a
⟨*proof*⟩

lemma *prime-divisorE*:
  assumes $a \neq 0$ **and** $\neg$ *is-unit* a
  obtains p **where** *is-prime* p **and** p *dvd* a
  ⟨*proof*⟩

lemma *is-prime-associated*:
  assumes *is-prime* p **and** *is-prime* q **and** q *dvd* p
  shows *normalize* q = *normalize* p
⟨*proof*⟩

lemma *prime-dvd-mult-iff*:
  assumes *is-prime* p

shows *p dvd a* ∗ *b* ⟷ *p dvd a* ∨ *p dvd b*
⟨*proof*⟩

lemma *prime-dvd-msetprod*:
assumes *is-prime p*
assumes *dvd*: *p dvd msetprod A*
obtains *a* **where** *a* ∈# *A* **and** *p dvd a*
⟨*proof*⟩

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*⟩

lemma *prime-dvd-power-iff*:
assumes *is-prime p*
shows *p dvd a* ˆ *n* ⟷ *p dvd a* ∧ *n* > *0*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *is-prime-inj-power*:
assumes *is-prime p*
shows *inj* (*op* ˆ *p*)
⟨*proof*⟩

definition *factorization* :: *'a* ⇒ *'a multiset option*
where *factorization a* = (*if a* = *0 then None*
else Some (*setsum* (λ*p*. *replicate-mset* (*Max* {*n*. *p* ˆ *n dvd a*}) *p*)
{*p*. *p dvd a* ∧ *is-prime p* ∧ *normalize p* = *p*}))

lemma *factorization-normalize* [*simp*]:
factorization (*normalize a*) = *factorization a*
⟨*proof*⟩

lemma *factorization-0* [*simp*]:
factorization 0 = *None*
⟨*proof*⟩

lemma *factorization-eq-None-iff* [*simp*]:
factorization a = *None* ⟷ *a* = *0*
⟨*proof*⟩

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*⟩

lemma *factorization-cases* [*case-names 0 factorization*]:
assumes *0*: *a = 0* ⟹ *P*
assumes *factorization*: ⋀*A*. *a* ≠ *0* ⟹ *factorization a = Some A* ⟹ *msetprod A = normalize a*
⟹ *0* ∉# *A* ⟹ (⋀*p*. *p* ∈# *A* ⟹ *normalize p = p*) ⟹ (⋀*p*. *p* ∈# *A* ⟹ *is-prime p*) ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *factorizationE*:
assumes *a* ≠ *0*
obtains *A u* **where** *factorization a = Some A normalize a = msetprod A*
0 ∉# *A* ⋀*p*. *p* ∈# *A* ⟹ *is-prime p* ⋀*p*. *p* ∈# *A* ⟹ *normalize p = p*
⟨*proof*⟩

lemma *prime-dvd-mset-prod-iff*:
assumes *is-prime p normalize p = p* ⋀*p*. *p* ∈# *A* ⟹ *is-prime p* ⋀*p*. *p* ∈# *A* ⟹ *normalize p = p*
shows *p dvd msetprod A* ⟷ *p* ∈# *A*
⟨*proof*⟩

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*⟩

end

instantiation *nat* :: *factorial-semiring*
begin

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

lemma *is-prime-eq-prime*:
is-prime = *prime*
⟨*proof*⟩

instance ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *is-prime-nat-abs-iff* [*simp*]:
is-prime (*nat* |*k*|) ⟷ *is-prime k*
⟨*proof*⟩

instance ⟨*proof*⟩

end

end