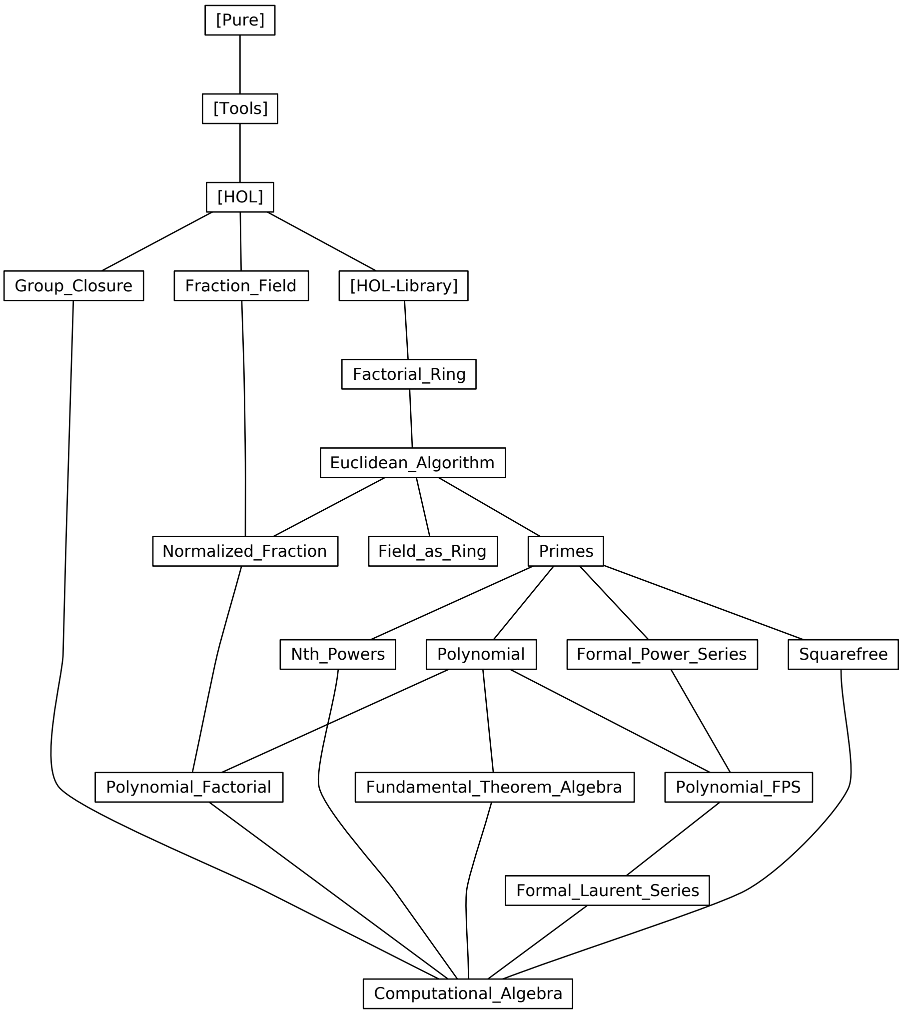
Computational Algebra

May 23, 2024

Contents


[Pure]
[Tools]
[HOL]
Group_Closure
Fraction_Field
[HOL-Library]
Factorial_Ring
Euclidean_Algorithm
Normalized_Fraction
Field_as_Ring
Primes
Nth_Powers
Polynomial
Formal_Power_Series
Squarefree
Polynomial_Factorial
Fundamental_Theorem_Algebra
Polynomial_FPS
Formal_Laurent_Series
Computational_Algebra

1 Factorial (semi)rings

theory *Factorial-Ring*
imports
Main
HOL−Library.Multiset
begin

unbundle *multiset.lifting*

1.1 Irreducible and prime elements

context *comm-semiring-1*
begin

definition *irreducible* :: $'a \Rightarrow bool$ **where**
$irreducible\ p \longleftrightarrow p \neq 0 \wedge \neg p\ dvd\ 1 \wedge (\forall a\ b.\ p = a * b \longrightarrow a\ dvd\ 1 \vee b\ dvd\ 1)$

lemma *not-irreducible-zero* [*simp*]: $\neg irreducible\ 0$
⟨*proof*⟩

lemma *irreducible-not-unit*: $irreducible\ p \Longrightarrow \neg p\ dvd\ 1$
⟨*proof*⟩

lemma *not-irreducible-one* [*simp*]: $\neg irreducible\ 1$
⟨*proof*⟩

lemma *irreducibleI*:
$p \neq 0 \Longrightarrow \neg p\ dvd\ 1 \Longrightarrow (\bigwedge a\ b.\ p = a * b \Longrightarrow a\ dvd\ 1 \vee b\ dvd\ 1) \Longrightarrow irreducible\ p$
⟨*proof*⟩

lemma *irreducibleD*: $irreducible\ p \Longrightarrow p = a * b \Longrightarrow a\ dvd\ 1 \vee b\ dvd\ 1$
⟨*proof*⟩

lemma *irreducible-mono*:
assumes *irr*: *irreducible b* **and** *a dvd b* $\neg a\ dvd\ 1$
shows *irreducible a*
⟨*proof*⟩

lemma *irreducible-multD*:
assumes *l*: *irreducible* $(a*b)$
shows $a\ dvd\ 1 \wedge irreducible\ b \vee b\ dvd\ 1 \wedge irreducible\ a$
⟨*proof*⟩

lemma *irreducible-power-iff* [*simp*]:
$irreducible\ (p\ \hat{}\ n) \longleftrightarrow irreducible\ p \wedge n = 1$
⟨*proof*⟩

definition *prime-elem* :: *'a* ⇒ *bool* **where**
prime-elem p ⟷ *p* ≠ *0* ∧ ¬*p dvd 1* ∧ (∀ *a b*. *p dvd* (*a* ∗ *b*) ⟶ *p dvd a* ∨ *p dvd b*)

lemma *not-prime-elem-zero* [*simp*]: ¬*prime-elem 0*
⟨*proof*⟩

lemma *prime-elem-not-unit*: *prime-elem p* ⟹ ¬*p dvd 1*
⟨*proof*⟩

lemma *prime-elemI*:
p ≠ *0* ⟹ ¬*p dvd 1* ⟹ (⋀*a b*. *p dvd* (*a* ∗ *b*) ⟹ *p dvd a* ∨ *p dvd b*) ⟹ *prime-elem p*
⟨*proof*⟩

lemma *prime-elem-dvd-multD*:
prime-elem p ⟹ *p dvd* (*a* ∗ *b*) ⟹ *p dvd a* ∨ *p dvd b*
⟨*proof*⟩

lemma *prime-elem-dvd-mult-iff*:
prime-elem p ⟹ *p dvd* (*a* ∗ *b*) ⟷ *p dvd a* ∨ *p dvd b*
⟨*proof*⟩

lemma *not-prime-elem-one* [*simp*]:
¬ *prime-elem 1*
⟨*proof*⟩

lemma *prime-elem-not-zeroI*:
assumes *prime-elem p*
shows *p* ≠ *0*
⟨*proof*⟩

lemma *prime-elem-dvd-power*:
prime-elem p ⟹ *p dvd x* ^ *n* ⟹ *p dvd x*
⟨*proof*⟩

lemma *prime-elem-dvd-power-iff*:
prime-elem p ⟹ *n* > *0* ⟹ *p dvd x* ^ *n* ⟷ *p dvd x*
⟨*proof*⟩

lemma *prime-elem-imp-nonzero* [*simp*]:
ASSUMPTION (*prime-elem x*) ⟹ *x* ≠ *0*
⟨*proof*⟩

lemma *prime-elem-imp-not-one* [*simp*]:
ASSUMPTION (*prime-elem x*) ⟹ *x* ≠ *1*
⟨*proof*⟩

end

lemma (**in** *normalization-semidom*) *irreducible-cong*:
assumes *normalize a = normalize b*
shows *irreducible a* ⟷ *irreducible b*
⟨*proof*⟩

lemma (**in** *normalization-semidom*) *associatedE1*:
assumes *normalize a = normalize b*
obtains *u* **where** *is-unit u a = u* ∗ *b*
⟨*proof*⟩

lemma (**in** *normalization-semidom*) *associatedE2*:
assumes *normalize a = normalize b*
obtains *u* **where** *is-unit u b = u* ∗ *a*
⟨*proof*⟩

lemma (**in** *normalization-semidom*) *normalize-power-normalize*:
normalize (*normalize x* ^ *n*) = *normalize* (*x* ^ *n*)
⟨*proof*⟩

context *algebraic-semidom*
begin

lemma *prime-elem-imp-irreducible*:
assumes *prime-elem p*
shows *irreducible p*
⟨*proof*⟩

lemma (**in** *algebraic-semidom*) *unit-imp-no-irreducible-divisors*:
assumes *is-unit x irreducible p*
shows ¬*p dvd x*
⟨*proof*⟩

lemma *unit-imp-no-prime-divisors*:
assumes *is-unit x prime-elem p*
shows ¬*p dvd x*
⟨*proof*⟩

lemma *prime-elem-mono*:
assumes *prime-elem p* ¬*q dvd 1 q dvd p*
shows *prime-elem q*
⟨*proof*⟩

lemma *irreducibleD'*:
assumes *irreducible a b dvd a*
shows *a dvd b* ∨ *is-unit b*

⟨*proof*⟩

lemma *irreducibleI'*:
assumes $a \neq 0$ ¬*is-unit a* $\bigwedge b.$ *b dvd a* $\Longrightarrow$ *a dvd b* ∨ *is-unit b*
shows *irreducible a*
⟨*proof*⟩

lemma *irreducible-altdef*:
irreducible x $\longleftrightarrow$ $x \neq 0$ ∧ ¬*is-unit x* ∧ (∀ *b*. *b dvd x* ⟶ *x dvd b* ∨ *is-unit b*)
⟨*proof*⟩

lemma *prime-elem-multD*:
assumes *prime-elem* (*a* ∗ *b*)
shows *is-unit a* ∨ *is-unit b*
⟨*proof*⟩

lemma *prime-elemD2*:
assumes *prime-elem p* **and** *a dvd p* **and** ¬ *is-unit a*
shows *p dvd a*
⟨*proof*⟩

lemma *prime-elem-dvd-prod-msetE*:
assumes *prime-elem p*
assumes *dvd*: *p dvd prod-mset A*
obtains *a* **where** *a* ∈# *A* **and** *p dvd a*
⟨*proof*⟩

context
begin

lemma *prime-elem-powerD*:
assumes *prime-elem* (*p* ^ *n*)
shows *prime-elem p* ∧ *n* = *1*
⟨*proof*⟩

lemma *prime-elem-power-iff*:
prime-elem (*p* ^ *n*) ⟷ *prime-elem p* ∧ *n* = *1*
⟨*proof*⟩

end

lemma *irreducible-mult-unit-left*:
is-unit a ⟹ *irreducible* (*a* ∗ *p*) ⟷ *irreducible p*
⟨*proof*⟩

lemma *prime-elem-mult-unit-left*:
is-unit a ⟹ *prime-elem* (*a* ∗ *p*) ⟷ *prime-elem p*
⟨*proof*⟩

lemma *prime-elem-dvd-cases*:
assumes *pk*: *p*∗*k dvd m*∗*n* **and** *p*: *prime-elem p*
shows (∃ *x*. *k dvd x*∗*n* ∧ *m* = *p*∗*x*) ∨ (∃ *y*. *k dvd m*∗*y* ∧ *n* = *p*∗*y*)
⟨*proof*⟩

lemma *prime-elem-power-dvd-prod*:
assumes *pc*: *p*^*c dvd m*∗*n* **and** *p*: *prime-elem p*
shows ∃ *a b*. *a*+*b* = *c* ∧ *p*^*a dvd m* ∧ *p*^*b dvd n*
⟨*proof*⟩

lemma *prime-elem-power-dvd-cases*:
assumes *p* ^ *c dvd m* ∗ *n* **and** *a* + *b* = *Suc c* **and** *prime-elem p*
shows *p* ^ *a dvd m* ∨ *p* ^ *b dvd n*
⟨*proof*⟩

lemma *prime-elem-not-unit′* [*simp*]:
ASSUMPTION (*prime-elem x*) ⟹ ¬*is-unit x*
⟨*proof*⟩

lemma *prime-elem-dvd-power-iff*:
assumes *prime-elem p*
shows *p dvd a* ^ *n* ⟷ *p dvd a* ∧ *n* > *0*
⟨*proof*⟩

lemma *prime-power-dvd-multD*:
assumes *prime-elem p*
assumes *p* ^ *n dvd a* ∗ *b* **and** *n* > *0* **and** ¬ *p dvd a*
shows *p* ^ *n dvd b*
⟨*proof*⟩

end

1.2 Generalized primes: normalized prime elements

context *normalization-semidom*
begin

lemma *irreducible-normalized-divisors*:
assumes *irreducible x y dvd x normalize y* = *y*
shows *y* = *1* ∨ *y* = *normalize x*
⟨*proof*⟩

lemma *irreducible-normalize-iff* [*simp*]: *irreducible* (*normalize x*) = *irreducible x*
⟨*proof*⟩

lemma *prime-elem-normalize-iff* [*simp*]: *prime-elem* (*normalize x*) = *prime-elem x*
⟨*proof*⟩

lemma *prime-elem-associated*:
  assumes *prime-elem p* **and** *prime-elem q* **and** *q dvd p*
  shows *normalize q = normalize p*
⟨*proof*⟩

definition *prime* :: *'a ⇒ bool* **where**
  prime p ⟷ prime-elem p ∧ normalize p = p

lemma *not-prime-0* [*simp*]: ¬*prime 0* ⟨*proof*⟩

lemma *not-prime-unit*: *is-unit x* ⟹ ¬*prime x*
  ⟨*proof*⟩

lemma *not-prime-1* [*simp*]: ¬*prime 1* ⟨*proof*⟩

lemma *primeI*: *prime-elem x* ⟹ *normalize x = x* ⟹ *prime x*
  ⟨*proof*⟩

lemma *prime-imp-prime-elem* [*dest*]: *prime p* ⟹ *prime-elem p*
  ⟨*proof*⟩

lemma *normalize-prime*: *prime p* ⟹ *normalize p = p*
  ⟨*proof*⟩

lemma *prime-normalize-iff* [*simp*]: *prime* (*normalize p*) ⟷ *prime-elem p*
  ⟨*proof*⟩

lemma *prime-power-iff*:
  prime (*p* ^ *n*) ⟷ *prime p* ∧ *n* = *1*
  ⟨*proof*⟩

lemma *prime-imp-nonzero* [*simp*]:
  ASSUMPTION (*prime x*) ⟹ *x* ≠ *0*
  ⟨*proof*⟩

lemma *prime-imp-not-one* [*simp*]:
  ASSUMPTION (*prime x*) ⟹ *x* ≠ *1*
  ⟨*proof*⟩

lemma *prime-not-unit'* [*simp*]:
  ASSUMPTION (*prime x*) ⟹ ¬*is-unit x*
  ⟨*proof*⟩

lemma *prime-normalize'* [*simp*]: *ASSUMPTION* (*prime x*) ⟹ *normalize x = x*
  ⟨*proof*⟩

lemma *unit-factor-prime*: *prime x* ⟹ *unit-factor x = 1*
  ⟨*proof*⟩

lemma *unit-factor-prime'* [*simp*]: *ASSUMPTION* (*prime x*) $\Longrightarrow$ *unit-factor x* = *1*
⟨*proof*⟩

lemma *prime-imp-prime-elem'* [*simp*]: *ASSUMPTION* (*prime x*) $\Longrightarrow$ *prime-elem x*
⟨*proof*⟩

lemma *prime-dvd-multD*: *prime p* $\Longrightarrow$ *p dvd a* $*$ *b* $\Longrightarrow$ *p dvd a* $\vee$ *p dvd b*
⟨*proof*⟩

lemma *prime-dvd-mult-iff*: *prime p* $\Longrightarrow$ *p dvd a* $*$ *b* $\longleftrightarrow$ *p dvd a* $\vee$ *p dvd b*
⟨*proof*⟩

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

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

lemma *prime-dvd-prod-mset-iff*: *prime p* $\Longrightarrow$ *p dvd prod-mset A* $\longleftrightarrow$ ($\exists$ *x*. *x* $\in$# *A* $\wedge$ *p dvd x*)
⟨*proof*⟩

lemma *prime-dvd-prod-iff*: *finite A* $\Longrightarrow$ *prime p* $\Longrightarrow$ *p dvd prod f A* $\longleftrightarrow$ ($\exists$ *x*$\in$*A*. *p dvd f x*)
⟨*proof*⟩

lemma *primes-dvd-imp-eq*:
assumes *prime p prime q p dvd q*
shows *p* = *q*
⟨*proof*⟩

lemma *prime-dvd-prod-mset-primes-iff*:
assumes *prime p* $\bigwedge$*q*. *q* $\in$# *A* $\Longrightarrow$ *prime q*
shows *p dvd prod-mset A* $\longleftrightarrow$ *p* $\in$# *A*
⟨*proof*⟩

lemma *prod-mset-primes-dvd-imp-subset*:
assumes *prod-mset A dvd prod-mset B* $\bigwedge$*p*. *p* $\in$# *A* $\Longrightarrow$ *prime p* $\bigwedge$*p*. *p* $\in$# *B* $\Longrightarrow$ *prime p*
shows *A* $\subseteq$# *B*
⟨*proof*⟩

lemma *prod-mset-dvd-prod-mset-primes-iff*:
assumes $\bigwedge$*x*. *x* $\in$# *A* $\Longrightarrow$ *prime x* $\bigwedge$*x*. *x* $\in$# *B* $\Longrightarrow$ *prime x*
shows *prod-mset A dvd prod-mset B* $\longleftrightarrow$ *A* $\subseteq$# *B*

⟨*proof*⟩

lemma *is-unit-prod-mset-primes-iff*:
assumes $\bigwedge x.\ x \in\# A \Longrightarrow$ *prime* x
shows *is-unit* (*prod-mset* A) $\longleftrightarrow A = \{\#\}$
⟨*proof*⟩

lemma *prod-mset-primes-irreducible-imp-prime*:
assumes *irred*: *irreducible* (*prod-mset* A)
assumes A: $\bigwedge x.\ x \in\# A \Longrightarrow$ *prime* x
assumes B: $\bigwedge x.\ x \in\# B \Longrightarrow$ *prime* x
assumes C: $\bigwedge x.\ x \in\# C \Longrightarrow$ *prime* x
assumes *dvd*: *prod-mset* A *dvd* *prod-mset* B $*$ *prod-mset* C
shows *prod-mset* A *dvd* *prod-mset* B $\vee$ *prod-mset* A *dvd* *prod-mset* C
⟨*proof*⟩

lemma *prod-mset-primes-finite-divisor-powers*:
assumes A: $\bigwedge x.\ x \in\# A \Longrightarrow$ *prime* x
assumes B: $\bigwedge x.\ x \in\# B \Longrightarrow$ *prime* x
assumes $A \neq \{\#\}$
shows *finite* $\{n.$ *prod-mset* A ^ n *dvd* *prod-mset* $B\}$
⟨*proof*⟩

end

1.3 In a semiring with GCD, each irreducible element is a prime element

context *semiring-gcd*
begin

lemma *irreducible-imp-prime-elem-gcd*:
assumes *irreducible* x
shows *prime-elem* x
⟨*proof*⟩

lemma *prime-elem-imp-coprime*:
assumes *prime-elem* p $\neg p$ *dvd* n
shows *coprime* p n
⟨*proof*⟩

lemma *prime-imp-coprime*:
assumes *prime* p $\neg p$ *dvd* n
shows *coprime* p n
⟨*proof*⟩

lemma *prime-elem-imp-power-coprime*:
prime-elem $p \Longrightarrow \neg$ p *dvd* $a \Longrightarrow$ *coprime* a (p ^ m)
⟨*proof*⟩

lemma *prime-imp-power-coprime*:
prime p $\Longrightarrow$ $\neg$ *p dvd a* $\Longrightarrow$ *coprime a* (*p* ^ *m*)
⟨*proof*⟩

lemma *prime-elem-divprod-pow*:
assumes *p*: *prime-elem p* **and** *ab*: *coprime a b* **and** *pab*: *p*^*n dvd a* $*$ *b*
shows *p*^*n dvd a* $\vee$ *p*^*n dvd b*
⟨*proof*⟩

lemma *primes-coprime*:
prime p $\Longrightarrow$ *prime q* $\Longrightarrow$ *p* $\neq$ *q* $\Longrightarrow$ *coprime p q*
⟨*proof*⟩

end

1.4 Factorial semirings: algebraic structures with unique prime factorizations

class *factorial-semiring* = *normalization-semidom* +
assumes *prime-factorization-exists*:
x $\neq$ *0* $\Longrightarrow$ $\exists$ *A*. ($\forall$ *x*. *x* $\in$# *A* $\longrightarrow$ *prime-elem x*) $\wedge$ *normalize* (*prod-mset A*) = *normalize x*

Alternative characterization

lemma (**in** *normalization-semidom*) *factorial-semiring-altI-aux*:
assumes *finite-divisors*: $\bigwedge$*x*. *x* $\neq$ *0* $\Longrightarrow$ *finite* {*y*. *y dvd x* $\wedge$ *normalize y* = *y*}
assumes *irreducible-imp-prime-elem*: $\bigwedge$*x*. *irreducible x* $\Longrightarrow$ *prime-elem x*
assumes *x* $\neq$ *0*
shows $\exists$ *A*. ($\forall$ *x*. *x* $\in$# *A* $\longrightarrow$ *prime-elem x*) $\wedge$ *normalize* (*prod-mset A*) = *normalize x*
⟨*proof*⟩

lemma *factorial-semiring-altI*:
assumes *finite-divisors*: $\bigwedge$*x*::'*a*. *x* $\neq$ *0* $\Longrightarrow$ *finite* {*y*. *y dvd x* $\wedge$ *normalize y* = *y*}
assumes *irreducible-imp-prime*: $\bigwedge$*x*::'*a*. *irreducible x* $\Longrightarrow$ *prime-elem x*
shows *OFCLASS*('*a* :: *normalization-semidom*, *factorial-semiring-class*)
⟨*proof*⟩

Properties

context *factorial-semiring*
begin

lemma *prime-factorization-exists'*:
assumes *x* $\neq$ *0*
obtains *A* **where** $\bigwedge$*x*. *x* $\in$# *A* $\Longrightarrow$ *prime x normalize* (*prod-mset A*) = *normalize x*
⟨*proof*⟩

lemma *irreducible-imp-prime-elem*:
assumes *irreducible x*
shows *prime-elem x*
⟨*proof*⟩

lemma *finite-divisor-powers*:
assumes $y \neq 0$ ¬*is-unit x*
shows *finite* {*n. x* ^ *n dvd y*}
⟨*proof*⟩

lemma *finite-prime-divisors*:
assumes $x \neq 0$
shows *finite* {*p. prime p* ∧ *p dvd x*}
⟨*proof*⟩

lemma *infinite-unit-divisor-powers*:
assumes $y \neq 0$
assumes *is-unit x*
shows *infinite* {*n. x*^*n dvd y*}
⟨*proof*⟩

corollary *is-unit-iff-infinite-divisor-powers*:
assumes $y \neq 0$
shows *is-unit x* ⟷ *infinite* {*n. x*^*n dvd y*}
⟨*proof*⟩

lemma *prime-elem-iff-irreducible*: *prime-elem x* ⟷ *irreducible x*
⟨*proof*⟩

lemma *prime-divisor-exists*:
assumes $a \neq 0$ ¬*is-unit a*
shows ∃ *b. b dvd a* ∧ *prime b*
⟨*proof*⟩

lemma *prime-divisors-induct* [*case-names zero unit factor*]:
assumes *P 0* ⋀*x. is-unit x* ⟹ *P x* ⋀*p x. prime p* ⟹ *P x* ⟹ *P* (*p* ∗ *x*)
shows *P x*
⟨*proof*⟩

lemma *no-prime-divisors-imp-unit*:
assumes $a \neq 0$ ⋀*b. b dvd a* ⟹ *normalize b* = *b* ⟹ ¬ *prime-elem b*
shows *is-unit a*
⟨*proof*⟩

lemma *prime-divisorE*:
assumes $a \neq 0$ **and** ¬ *is-unit a*
obtains *p* **where** *prime p* **and** *p dvd a*
⟨*proof*⟩

definition *multiplicity* :: $'a \Rightarrow 'a \Rightarrow nat$ **where**
multiplicity p x = (*if finite* {*n. p* ^ *n dvd x*} *then Max* {*n. p* ^ *n dvd x*} *else 0*)

lemma *multiplicity-dvd*: *p* ^ *multiplicity p x dvd x*
⟨*proof*⟩

lemma *multiplicity-dvd'*: *n* ≤ *multiplicity p x* ⟹ *p* ^ *n dvd x*
⟨*proof*⟩

context
fixes *x p* :: *'a*
assumes *xp*: *x* ≠ *0* ¬*is-unit p*
begin

lemma *multiplicity-eq-Max*: *multiplicity p x* = *Max* {*n. p* ^ *n dvd x*}
⟨*proof*⟩

lemma *multiplicity-geI*:
assumes *p* ^ *n dvd x*
shows *multiplicity p x* ≥ *n*
⟨*proof*⟩

lemma *multiplicity-lessI*:
assumes ¬*p* ^ *n dvd x*
shows *multiplicity p x* < *n*
⟨*proof*⟩

lemma *power-dvd-iff-le-multiplicity*:
p ^ *n dvd x* ⟷ *n* ≤ *multiplicity p x*
⟨*proof*⟩

lemma *multiplicity-eq-zero-iff*:
shows *multiplicity p x* = *0* ⟷ ¬*p dvd x*
⟨*proof*⟩

lemma *multiplicity-gt-zero-iff*:
shows *multiplicity p x* > *0* ⟷ *p dvd x*
⟨*proof*⟩

lemma *multiplicity-decompose*:
¬*p dvd* (*x div p* ^ *multiplicity p x*)
⟨*proof*⟩

lemma *multiplicity-decompose'*:
obtains *y* **where** *x* = *p* ^ *multiplicity p x* ∗ *y* ¬*p dvd y*
⟨*proof*⟩

end

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

lemma *prime-elem-multiplicity-eq-zero-iff*:
prime-elem p $\Longrightarrow$ *x* $\neq$ *0* $\Longrightarrow$ *multiplicity p x = 0* $\longleftrightarrow$ $\neg$*p dvd x*
⟨*proof*⟩

lemma *prime-multiplicity-other*:
assumes *prime p prime q p* $\neq$ *q*
shows *multiplicity p q = 0*
⟨*proof*⟩

lemma *prime-multiplicity-gt-zero-iff*:
prime-elem p $\Longrightarrow$ *x* $\neq$ *0* $\Longrightarrow$ *multiplicity p x > 0* $\longleftrightarrow$ *p dvd x*
⟨*proof*⟩

lemma *multiplicity-unit-left*: *is-unit p* $\Longrightarrow$ *multiplicity p x = 0*
⟨*proof*⟩

lemma *multiplicity-unit-right*:
assumes *is-unit x*
shows *multiplicity p x = 0*
⟨*proof*⟩

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

lemma *multiplicity-eqI*:
assumes *p ^ n dvd x* $\neg$*p ^ Suc n dvd x*
shows *multiplicity p x = n*
⟨*proof*⟩

context
fixes *x p* :: *'a*
assumes *xp*: *x* $\neq$ *0* $\neg$*is-unit p*
begin

lemma *multiplicity-times-same*:
assumes *p* $\neq$ *0*
shows *multiplicity p* (*p* $*$ *x*) = *Suc* (*multiplicity p x*)
⟨*proof*⟩

end

lemma *multiplicity-same-power'*: *multiplicity p* (*p ^ n*) = (*if p = 0* $\vee$ *is-unit p then 0 else n*)
⟨*proof*⟩

lemma *multiplicity-same-power*:
  $p \neq 0 \Longrightarrow \neg$*is-unit p* $\Longrightarrow$ *multiplicity p* $(p \mathbin{\char`\^} n) = n$
  ⟨*proof*⟩

lemma *multiplicity-prime-elem-times-other*:
  assumes *prime-elem p* $\neg p$ *dvd q*
  shows *multiplicity p* $(q * x) =$ *multiplicity p x*
⟨*proof*⟩

lemma *multiplicity-self*:
  assumes $p \neq 0$ $\neg$*is-unit p*
  shows *multiplicity p p* $= 1$
⟨*proof*⟩

lemma *multiplicity-times-unit-left*:
  assumes *is-unit c*
  shows *multiplicity* $(c * p)$ $x =$ *multiplicity p x*
⟨*proof*⟩

lemma *multiplicity-times-unit-right*:
  assumes *is-unit c*
  shows *multiplicity p* $(c * x) =$ *multiplicity p x*
⟨*proof*⟩

lemma *multiplicity-normalize-left* [*simp*]:
  multiplicity (*normalize p*) $x =$ *multiplicity p x*
⟨*proof*⟩

lemma *multiplicity-normalize-right* [*simp*]:
  multiplicity p (*normalize x*) $=$ *multiplicity p x*
⟨*proof*⟩

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

lemma *multiplicity-prime-power* [*simp*]: *prime-elem p* $\Longrightarrow$ *multiplicity p* $(p \mathbin{\char`\^} n)$ $= n$
  ⟨*proof*⟩

lift-definition *prime-factorization* :: *'a* ⇒ *'a multiset* **is**
  λx p. if prime p then multiplicity p x else 0
⟨*proof*⟩

abbreviation *prime-factors* :: *'a* ⇒ *'a set* **where**
  prime-factors a ≡ *set-mset* (*prime-factorization a*)

lemma *count-prime-factorization-nonprime*:
  ¬*prime p* $\Longrightarrow$ *count* (*prime-factorization x*) $p = 0$

⟨*proof*⟩

lemma *count-prime-factorization-prime*:
prime p $\Longrightarrow$ *count* (*prime-factorization x*) *p* = *multiplicity p x*
⟨*proof*⟩

lemma *count-prime-factorization*:
count (*prime-factorization x*) *p* = (*if prime p then multiplicity p x else 0*)
⟨*proof*⟩

lemma *dvd-imp-multiplicity-le*:
assumes *a dvd b b* $\neq$ *0*
shows *multiplicity p a* $\leq$ *multiplicity p b*
⟨*proof*⟩

lemma *prime-power-inj*:
assumes *prime a a* ^ *m* = *a* ^ *n*
shows *m* = *n*
⟨*proof*⟩

lemma *prime-power-inj'*:
assumes *prime p prime q*
assumes *p* ^ *m* = *q* ^ *n m* > *0 n* > *0*
shows *p* = *q m* = *n*
⟨*proof*⟩

lemma *prime-power-eq-one-iff* [*simp*]: *prime p* $\Longrightarrow$ *p* ^ *n* = *1* $\longleftrightarrow$ *n* = *0*
⟨*proof*⟩

lemma *one-eq-prime-power-iff* [*simp*]: *prime p* $\Longrightarrow$ *1* = *p* ^ *n* $\longleftrightarrow$ *n* = *0*
⟨*proof*⟩

lemma *prime-power-inj''*:
assumes *prime p prime q*
shows *p* ^ *m* = *q* ^ *n* $\longleftrightarrow$ (*m* = *0* $\wedge$ *n* = *0*) $\vee$ (*p* = *q* $\wedge$ *m* = *n*)
⟨*proof*⟩

lemma *prime-factorization-0* [*simp*]: *prime-factorization 0* = {#}
⟨*proof*⟩

lemma *prime-factorization-empty-iff*:
prime-factorization x = {#} $\longleftrightarrow$ *x* = *0* $\vee$ *is-unit x*
⟨*proof*⟩

lemma *prime-factorization-unit*:
assumes *is-unit x*
shows *prime-factorization x* = {#}
⟨*proof*⟩

lemma *prime-factorization-1* [*simp*]: *prime-factorization 1* = {#}
⟨*proof*⟩

lemma *prime-factorization-times-prime*:
assumes $x \neq 0$ *prime p*
shows *prime-factorization* $(p * x)$ = {#*p*#} + *prime-factorization x*
⟨*proof*⟩

lemma *prod-mset-prime-factorization-weak*:
assumes $x \neq 0$
shows *normalize* (*prod-mset* (*prime-factorization x*)) = *normalize x*
⟨*proof*⟩

lemma *in-prime-factors-iff*:
$p \in$ *prime-factors x* $\longleftrightarrow x \neq 0 \land p$ *dvd x* $\land$ *prime p*
⟨*proof*⟩

lemma *in-prime-factors-imp-prime* [*intro*]:
$p \in$ *prime-factors x* $\Longrightarrow$ *prime p*
⟨*proof*⟩

lemma *in-prime-factors-imp-dvd* [*dest*]:
$p \in$ *prime-factors x* $\Longrightarrow$ *p dvd x*
⟨*proof*⟩

lemma *prime-factorsI*:
$x \neq 0 \Longrightarrow$ *prime p* $\Longrightarrow$ *p dvd x* $\Longrightarrow p \in$ *prime-factors x*
⟨*proof*⟩

lemma *prime-factors-dvd*:
$x \neq 0 \Longrightarrow$ *prime-factors x* = {*p*. *prime p* $\land$ *p dvd x*}
⟨*proof*⟩

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

lemma *prime-factorization-prime*:
assumes *prime p*
shows *prime-factorization p* = {#*p*#}
⟨*proof*⟩

lemma *prime-factorization-prod-mset-primes*:
assumes $\bigwedge p.\ p \in\#\ A \Longrightarrow$ *prime p*
shows *prime-factorization* (*prod-mset A*) = *A*
⟨*proof*⟩

lemma *prime-factorization-cong*:
normalize x = *normalize y* $\Longrightarrow$ *prime-factorization x* = *prime-factorization y*

⟨*proof*⟩

lemma *prime-factorization-unique*:
assumes $x \neq 0$ $y \neq 0$
shows *prime-factorization* x = *prime-factorization* $y \longleftrightarrow$ *normalize* x = *normalize* y
⟨*proof*⟩

lemma *prime-factorization-normalize* [*simp*]:
prime-factorization (*normalize* x) = *prime-factorization* x
⟨*proof*⟩

lemma *prime-factorization-eqI-strong*:
assumes $\bigwedge p.\ p \in\# P \Longrightarrow$ *prime* p *prod-mset* $P = n$
shows *prime-factorization* $n = P$
⟨*proof*⟩

lemma *prime-factorization-eqI*:
assumes $\bigwedge p.\ p \in\# P \Longrightarrow$ *prime* p *normalize* (*prod-mset* P) = *normalize* n
shows *prime-factorization* $n = P$
⟨*proof*⟩

lemma *prime-factorization-mult*:
assumes $x \neq 0$ $y \neq 0$
shows *prime-factorization* $(x * y)$ = *prime-factorization* x + *prime-factorization* y
⟨*proof*⟩

lemma *prime-factorization-prod*:
assumes *finite* A $\bigwedge x.\ x \in A \Longrightarrow f\ x \neq 0$
shows *prime-factorization* (*prod* f A) = ($\sum n \in A$. *prime-factorization* (f n))
⟨*proof*⟩

lemma *prime-elem-multiplicity-mult-distrib*:
assumes *prime-elem* p $x \neq 0$ $y \neq 0$
shows *multiplicity* p $(x * y)$ = *multiplicity* p x + *multiplicity* p y
⟨*proof*⟩

lemma *prime-elem-multiplicity-prod-mset-distrib*:
assumes *prime-elem* p $0 \notin\# A$
shows *multiplicity* p (*prod-mset* A) = *sum-mset* (*image-mset* (*multiplicity* p) A)
⟨*proof*⟩

lemma *prime-elem-multiplicity-power-distrib*:
assumes *prime-elem* p $x \neq 0$
shows *multiplicity* p $(x \mathbin{\char`^} n) = n$ * *multiplicity* p x
⟨*proof*⟩

lemma *prime-elem-multiplicity-prod-distrib*:
assumes *prime-elem p* $0 \notin f$ ‘ *A finite A*
shows *multiplicity p* (*prod f A*) = ($\sum$ *x*∈*A*. *multiplicity p* (*f x*))
⟨*proof*⟩

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

lemma *prime-factorization-prime-power*:
prime p $\Longrightarrow$ *prime-factorization* (*p* ^ *n*) = *replicate-mset n p*
⟨*proof*⟩

lemma *prime-factorization-subset-iff-dvd*:
assumes [*simp*]: $x \neq 0$ $y \neq 0$
shows *prime-factorization x* ⊆# *prime-factorization y* ⟷ *x dvd y*
⟨*proof*⟩

lemma *prime-factorization-subset-imp-dvd*:
$x \neq 0$ $\Longrightarrow$ (*prime-factorization x* ⊆# *prime-factorization y*) $\Longrightarrow$ *x dvd y*
⟨*proof*⟩

lemma *prime-factorization-divide*:
assumes *b dvd a*
shows *prime-factorization* (*a div b*) = *prime-factorization a* − *prime-factorization b*
⟨*proof*⟩

lemma *zero-not-in-prime-factors* [*simp*]: $0 \notin$ *prime-factors x*
⟨*proof*⟩

lemma *prime-prime-factors*:
prime p $\Longrightarrow$ *prime-factors p* = {*p*}
⟨*proof*⟩

lemma *prime-factors-product*:
$x \neq 0$ $\Longrightarrow$ $y \neq 0$ $\Longrightarrow$ *prime-factors* (*x* ∗ *y*) = *prime-factors x* ∪ *prime-factors y*
⟨*proof*⟩

lemma *dvd-prime-factors* [*intro*]:
$y \neq 0$ $\Longrightarrow$ *x dvd y* $\Longrightarrow$ *prime-factors x* ⊆ *prime-factors y*
⟨*proof*⟩

lemma *multiplicity-le-imp-dvd*:
assumes $x \neq 0$ $\bigwedge p$. *prime p* $\Longrightarrow$ *multiplicity p x* ≤ *multiplicity p y*
shows *x dvd y*
⟨*proof*⟩

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

lemma *multiplicity-eq-imp-eq*:
assumes $x \neq 0$ $y \neq 0$
assumes $\bigwedge p.\ prime\ p \Longrightarrow multiplicity\ p\ x = multiplicity\ p\ y$
shows $normalize\ x = normalize\ y$
⟨*proof*⟩

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

lemma *prime-factorization-unique″*:
assumes $\forall p \in\#\ M.\ prime\ p$ $\forall p \in\#\ N.\ prime\ p$ $normalize\ (\prod i \in\#\ M.\ i) = normalize\ (\prod i \in\#\ N.\ i)$
shows $M = N$
⟨*proof*⟩

lemma *multiplicity-cong*:
$(\bigwedge r.\ p\ \hat{}\ r\ dvd\ a \longleftrightarrow p\ \hat{}\ r\ dvd\ b) \Longrightarrow multiplicity\ p\ a = multiplicity\ p\ b$
⟨*proof*⟩

lemma *not-dvd-imp-multiplicity-0*:
assumes $\neg p\ dvd\ x$
shows $multiplicity\ p\ x = 0$
⟨*proof*⟩

lemma *multiplicity-zero-left* [*simp*]: $multiplicity\ 0\ x = 0$
⟨*proof*⟩

lemma *inj-on-Prod-primes*:
assumes $\bigwedge P\ p.\ P \in A \Longrightarrow p \in P \Longrightarrow prime\ p$
assumes $\bigwedge P.\ P \in A \Longrightarrow finite\ P$
shows *inj-on Prod A*
⟨*proof*⟩

lemma *divides-primepow-weak*:
assumes *prime p* **and** $a\ dvd\ p\ \hat{}\ n$
obtains m **where** $m \leq n$ **and** $normalize\ a = normalize\ (p\ \hat{}\ m)$
⟨*proof*⟩

lemma *divide-out-primepow-ex*:
assumes $n \neq 0$ $\exists p \in prime\text{-}factors\ n.\ P\ p$
obtains $p\ k\ n'$ **where** $P\ p$ $prime\ p$ $p\ dvd\ n$ $\neg p\ dvd\ n'$ $k > 0$ $n = p\ \hat{}\ k * n'$
⟨*proof*⟩

lemma *divide-out-primepow*:
assumes $n \neq 0$ $\neg$*is-unit n*
obtains *p k n'* **where** *prime p p dvd n* $\neg$*p dvd n'* $k > 0$ $n = p$ ^ $k * n'$
⟨*proof*⟩

1.5 GCD and LCM computation with unique factorizations

definition *gcd-factorial a b* = (*if a* = *0 then normalize b*
else if b = *0 then normalize a*
else normalize (*prod-mset* (*prime-factorization a* ∩# *prime-factorization b*)))

definition *lcm-factorial a b* = (*if a* = *0* ∨ *b* = *0 then 0*
else normalize (*prod-mset* (*prime-factorization a* ∪# *prime-factorization b*)))

definition *Gcd-factorial A* =
(*if A* ⊆ {*0*} *then 0 else normalize* (*prod-mset* (*Inf* (*prime-factorization* ‘ (*A* − {*0*})))))

definition *Lcm-factorial A* =
(*if A* = {} *then 1*
else if 0 ∉ *A* ∧ *subset-mset.bdd-above* (*prime-factorization* ‘ (*A* − {*0*})) *then*
normalize (*prod-mset* (*Sup* (*prime-factorization* ‘ *A*)))
else
0)

lemma *prime-factorization-gcd-factorial*:
assumes [*simp*]: $a \neq 0$ $b \neq 0$
shows *prime-factorization* (*gcd-factorial a b*) = *prime-factorization a* ∩# *prime-factorization b*
⟨*proof*⟩

lemma *prime-factorization-lcm-factorial*:
assumes [*simp*]: $a \neq 0$ $b \neq 0$
shows *prime-factorization* (*lcm-factorial a b*) = *prime-factorization a* ∪# *prime-factorization b*
⟨*proof*⟩

lemma *prime-factorization-Gcd-factorial*:
assumes ¬*A* ⊆ {*0*}
shows *prime-factorization* (*Gcd-factorial A*) = *Inf* (*prime-factorization* ‘ (*A* − {*0*}))
⟨*proof*⟩

lemma *prime-factorization-Lcm-factorial*:
assumes *0* ∉ *A subset-mset.bdd-above* (*prime-factorization* ‘ *A*)
shows *prime-factorization* (*Lcm-factorial A*) = *Sup* (*prime-factorization* ‘ *A*)
⟨*proof*⟩

lemma *gcd-factorial-commute*: *gcd-factorial a b* = *gcd-factorial b a*
⟨*proof*⟩

lemma *gcd-factorial-dvd1*: *gcd-factorial a b dvd a*
⟨*proof*⟩

lemma *gcd-factorial-dvd2*: *gcd-factorial a b dvd b*
⟨*proof*⟩

lemma *normalize-gcd-factorial* [*simp*]: *normalize* (*gcd-factorial a b*) = *gcd-factorial a b*
⟨*proof*⟩

lemma *normalize-lcm-factorial* [*simp*]: *normalize* (*lcm-factorial a b*) = *lcm-factorial a b*
⟨*proof*⟩

lemma *gcd-factorial-greatest*: *c dvd gcd-factorial a b* **if** *c dvd a c dvd b* **for** *a b c*
⟨*proof*⟩

lemma *lcm-factorial-gcd-factorial*:
lcm-factorial a b = *normalize* (*a* ∗ *b div gcd-factorial a b*) **for** *a b*
⟨*proof*⟩

lemma *normalize-Gcd-factorial*:
normalize (*Gcd-factorial A*) = *Gcd-factorial A*
⟨*proof*⟩

lemma *Gcd-factorial-eq-0-iff*:
Gcd-factorial A = *0* ⟷ *A* ⊆ {*0*}
⟨*proof*⟩

lemma *Gcd-factorial-dvd*:
assumes *x* ∈ *A*
shows *Gcd-factorial A dvd x*
⟨*proof*⟩

lemma *Gcd-factorial-greatest*:
assumes ⋀*y*. *y* ∈ *A* ⟹ *x dvd y*
shows *x dvd Gcd-factorial A*
⟨*proof*⟩

lemma *normalize-Lcm-factorial*:
normalize (*Lcm-factorial A*) = *Lcm-factorial A*
⟨*proof*⟩

lemma *Lcm-factorial-eq-0-iff*:
Lcm-factorial A = *0* ⟷ *0* ∈ *A* ∨ ¬*subset-mset.bdd-above* (*prime-factorization* ‘ *A*)

⟨*proof*⟩

lemma *dvd-Lcm-factorial*:
assumes $x \in A$
shows *x dvd Lcm-factorial A*
⟨*proof*⟩

lemma *Lcm-factorial-least*:
assumes $\bigwedge y.\ y \in A \Longrightarrow y$ *dvd x*
shows *Lcm-factorial A dvd x*
⟨*proof*⟩

lemmas *gcd-lcm-factorial* =
gcd-factorial-dvd1 gcd-factorial-dvd2 gcd-factorial-greatest
normalize-gcd-factorial lcm-factorial-gcd-factorial
normalize-Gcd-factorial Gcd-factorial-dvd Gcd-factorial-greatest
normalize-Lcm-factorial dvd-Lcm-factorial Lcm-factorial-least

end

class *factorial-semiring-gcd* = *factorial-semiring* + *gcd* + *Gcd* +
assumes *gcd-eq-gcd-factorial*: *gcd a b* = *gcd-factorial a b*
and *lcm-eq-lcm-factorial*: *lcm a b* = *lcm-factorial a b*
and *Gcd-eq-Gcd-factorial*: *Gcd A* = *Gcd-factorial A*
and *Lcm-eq-Lcm-factorial*: *Lcm A* = *Lcm-factorial A*
begin

lemma *prime-factorization-gcd*:
assumes [*simp*]: $a \neq 0$ $b \neq 0$
shows *prime-factorization* (*gcd a b*) = *prime-factorization a* ∩# *prime-factorization b*
⟨*proof*⟩

lemma *prime-factorization-lcm*:
assumes [*simp*]: $a \neq 0$ $b \neq 0$
shows *prime-factorization* (*lcm a b*) = *prime-factorization a* ∪# *prime-factorization b*
⟨*proof*⟩

lemma *prime-factorization-Gcd*:
assumes *Gcd A* ≠ *0*
shows *prime-factorization* (*Gcd A*) = *Inf* (*prime-factorization* ‘ (*A* − {*0*}))
⟨*proof*⟩

lemma *prime-factorization-Lcm*:
assumes *Lcm A* ≠ *0*
shows *prime-factorization* (*Lcm A*) = *Sup* (*prime-factorization* ‘ *A*)
⟨*proof*⟩

lemma *prime-factors-gcd* [*simp*]:
$a \neq 0 \Longrightarrow b \neq 0 \Longrightarrow$ *prime-factors* (*gcd a b*) =
prime-factors a ∩ *prime-factors b*
⟨*proof*⟩

lemma *prime-factors-lcm* [*simp*]:
$a \neq 0 \Longrightarrow b \neq 0 \Longrightarrow$ *prime-factors* (*lcm a b*) =
prime-factors a ∪ *prime-factors b*
⟨*proof*⟩

subclass *semiring-gcd*
⟨*proof*⟩

subclass *semiring-Gcd*
⟨*proof*⟩

lemma
assumes $x \neq 0$ $y \neq 0$
shows *gcd-eq-factorial′*:
gcd x y = *normalize* (∏ *p* ∈ *prime-factors x* ∩ *prime-factors y*.
p ^ *min* (*multiplicity p x*) (*multiplicity p y*)) (**is** - = *?rhs1*)
and *lcm-eq-factorial′*:
lcm x y = *normalize* (∏ *p* ∈ *prime-factors x* ∪ *prime-factors y*.
p ^ *max* (*multiplicity p x*) (*multiplicity p y*)) (**is** - = *?rhs2*)
⟨*proof*⟩

lemma
assumes $x \neq 0$ $y \neq 0$ *prime p*
shows *multiplicity-gcd*: *multiplicity p* (*gcd x y*) = *min* (*multiplicity p x*) (*multiplicity p y*)
and *multiplicity-lcm*: *multiplicity p* (*lcm x y*) = *max* (*multiplicity p x*) (*multiplicity p y*)
⟨*proof*⟩

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

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

end

class *factorial-ring-gcd* = *factorial-semiring-gcd* + *idom*
begin

subclass *ring-gcd* ⟨*proof*⟩

subclass *idom-divide* ⟨*proof*⟩

end

class *factorial-semiring-multiplicative* =
factorial-semiring + *normalization-semidom-multiplicative*
begin

lemma *normalize-prod-mset-primes*:
$(\bigwedge p.\ p \in\# A \Longrightarrow prime\ p) \Longrightarrow normalize\ (prod\text{-}mset\ A) = prod\text{-}mset\ A$
⟨*proof*⟩

lemma *prod-mset-prime-factorization*:
assumes $x \neq 0$
shows $prod\text{-}mset\ (prime\text{-}factorization\ x) = normalize\ x$
⟨*proof*⟩

lemma *prime-decomposition*: $unit\text{-}factor\ x * prod\text{-}mset\ (prime\text{-}factorization\ x) = x$
⟨*proof*⟩

lemma *prod-prime-factors*:
assumes $x \neq 0$
shows $(\prod p \in prime\text{-}factors\ x.\ p \ \hat{}\ multiplicity\ p\ x) = normalize\ x$
⟨*proof*⟩

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

lemma *divides-primepow*:
assumes *prime p* **and** *a dvd p* ^ *n*
obtains *m* **where** $m \leq n$ **and** $normalize\ a = p \ \hat{}\ m$
⟨*proof*⟩

lemma *Ex-other-prime-factor*:
assumes $n \neq 0$ **and** $\neg(\exists k.\ normalize\ n = p \ \hat{}\ k)$ *prime p*
shows $\exists q \in prime\text{-}factors\ n.\ q \neq p$
⟨*proof*⟩

Now a string of results due to Maya Kdzioka

lemma *multiplicity-dvd-iff-dvd*:
assumes $x \neq 0$
shows $p\hat{}k\ dvd\ x \longleftrightarrow p\hat{}k\ dvd\ p\hat{}multiplicity\ p\ x$
⟨*proof*⟩

lemma *multiplicity-decomposeI*:
assumes $x = p\hat{}k * x'$ **and** $\neg$ *p dvd* x' **and** $p \neq 0$
shows *multiplicity p x* = *k*
⟨*proof*⟩

lemma *multiplicity-sum-lt*:
assumes *multiplicity p a* < *multiplicity p b a* ≠ *0 b* ≠ *0*
shows *multiplicity p* (*a* + *b*) = *multiplicity p a*
⟨*proof*⟩

corollary *multiplicity-sum-min*:
assumes *multiplicity p a* ≠ *multiplicity p b a* ≠ *0 b* ≠ *0*
shows *multiplicity p* (*a* + *b*) = *min* (*multiplicity p a*) (*multiplicity p b*)
⟨*proof*⟩

end

lifting-update *multiset.lifting*
lifting-forget *multiset.lifting*

end

2 Abstract euclidean algorithm in euclidean (semi)rings

theory *Euclidean-Algorithm*
imports *Factorial-Ring*
begin

2.1 Generic construction of the (simple) euclidean algorithm

class *normalization-euclidean-semiring* = *euclidean-semiring* + *normalization-semidom*
begin

lemma *euclidean-size-normalize* [*simp*]:
euclidean-size (*normalize a*) = *euclidean-size a*
⟨*proof*⟩

context
begin

qualified function *gcd* :: *'a* ⇒ *'a* ⇒ *'a*
where *gcd a b* = (*if b* = *0 then normalize a else gcd b* (*a mod b*))
⟨*proof*⟩
termination
⟨*proof*⟩

declare *gcd.simps* [*simp del*]

lemma *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*⟩ **lemma** *gcd-0*:
gcd a 0 = normalize a
⟨*proof*⟩ **lemma** *gcd-mod*:
$a \neq 0 \Longrightarrow gcd\ a\ (b\ mod\ a) = gcd\ b\ a$
⟨*proof*⟩ **definition** *lcm* :: *'a ⇒ 'a ⇒ 'a*
where *lcm a b = normalize (a ∗ b div gcd a b)*

qualified definition *Lcm* :: *'a set ⇒ 'a* — Somewhat complicated definition of Lcm that has the advantage of working for infinite sets as well
where
[*code del*]: *Lcm 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)

qualified definition *Gcd* :: *'a set ⇒ 'a*
where [*code del*]: *Gcd A = Lcm {d. ∀ a∈A. d dvd a}*

end

lemma *semiring-gcd*:
class.semiring-gcd one zero times gcd lcm
divide plus minus unit-factor normalize
⟨*proof*⟩

interpretation *semiring-gcd one zero times gcd lcm*
divide plus minus unit-factor normalize
⟨*proof*⟩

lemma *semiring-Gcd*:
class.semiring-Gcd one zero times gcd lcm Gcd Lcm
divide plus minus unit-factor normalize
⟨*proof*⟩

interpretation *semiring-Gcd one zero times gcd lcm Gcd Lcm*
divide plus minus unit-factor normalize
⟨*proof*⟩

subclass *factorial-semiring*
⟨*proof*⟩

lemma *Gcd-eucl-set* [*code*]:
Gcd (set xs) = fold gcd xs 0
⟨*proof*⟩

lemma *Lcm-eucl-set* [*code*]:
Lcm (*set xs*) = *fold lcm xs 1*
⟨*proof*⟩

end

hide-const (**open**) *gcd lcm Gcd Lcm*

lemma *prime-elem-int-abs-iff* [*simp*]:
fixes *p* :: *int*
shows *prime-elem* |*p*| ⟷ *prime-elem p*
⟨*proof*⟩

lemma *prime-elem-int-minus-iff* [*simp*]:
fixes *p* :: *int*
shows *prime-elem* (− *p*) ⟷ *prime-elem p*
⟨*proof*⟩

lemma *prime-int-iff*:
fixes *p* :: *int*
shows *prime p* ⟷ *p* > *0* ∧ *prime-elem p*
⟨*proof*⟩

2.2 The (simple) euclidean algorithm as gcd computation

class *euclidean-semiring-gcd* = *normalization-euclidean-semiring* + *gcd* + *Gcd* +
assumes *gcd-eucl*: *Euclidean-Algorithm.gcd* = *GCD.gcd*
and *lcm-eucl*: *Euclidean-Algorithm.lcm* = *GCD.lcm*
assumes *Gcd-eucl*: *Euclidean-Algorithm.Gcd* = *GCD.Gcd*
and *Lcm-eucl*: *Euclidean-Algorithm.Lcm* = *GCD.Lcm*
begin

subclass *semiring-gcd*
⟨*proof*⟩

subclass *semiring-Gcd*
⟨*proof*⟩

subclass *factorial-semiring-gcd*
⟨*proof*⟩

lemma *gcd-mod-right* [*simp*]:
a ≠ *0* ⟹ *gcd a* (*b mod a*) = *gcd a b*
⟨*proof*⟩

lemma *gcd-mod-left* [*simp*]:
b ≠ *0* ⟹ *gcd* (*a mod b*) *b* = *gcd a b*
⟨*proof*⟩

lemma *euclidean-size-gcd-le1* [*simp*]:
assumes $a \neq 0$
shows *euclidean-size* (*gcd a b*) ≤ *euclidean-size a*
⟨*proof*⟩

lemma *euclidean-size-gcd-le2* [*simp*]:
$b \neq 0 \Longrightarrow$ *euclidean-size* (*gcd a b*) ≤ *euclidean-size b*
⟨*proof*⟩

lemma *euclidean-size-gcd-less1*:
assumes $a \neq 0$ **and** ¬ *a dvd b*
shows *euclidean-size* (*gcd a b*) < *euclidean-size a*
⟨*proof*⟩

lemma *euclidean-size-gcd-less2*:
assumes $b \neq 0$ **and** ¬ *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** ¬ *b dvd a*
shows *euclidean-size a* < *euclidean-size* (*lcm a b*)
⟨*proof*⟩

lemma *euclidean-size-lcm-less2*:
assumes $a \neq 0$ **and** ¬ *a dvd b*
shows *euclidean-size b* < *euclidean-size* (*lcm a b*)
⟨*proof*⟩

end

lemma *factorial-euclidean-semiring-gcdI*:
OFCLASS(*'a*::{*factorial-semiring-gcd*, *normalization-euclidean-semiring*}, *euclidean-semiring-gcd-class*)
⟨*proof*⟩

2.3 The extended euclidean algorithm

class *euclidean-ring-gcd* = *euclidean-semiring-gcd* + *idom*
begin

subclass *euclidean-ring* ⟨*proof*⟩
subclass *ring-gcd* ⟨*proof*⟩
subclass *factorial-ring-gcd* ⟨*proof*⟩

function *euclid-ext-aux* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow ('a \times 'a) \times 'a$
  where *euclid-ext-aux* s' s t' t r' r = (
    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 s $(s' - q * s)$ t $(t' - q * t)$ r $(r'$ *mod* $r))$
  ⟨*proof*⟩
termination
  ⟨*proof*⟩

abbreviation (*input*) *euclid-ext* :: $'a \Rightarrow 'a \Rightarrow ('a \times 'a) \times 'a$
  where *euclid-ext* ≡ *euclid-ext-aux 1 0 0 1*

lemma
  assumes *gcd* r' r = *gcd* a b
  assumes $s' * a + t' * b = r'$
  assumes $s * a + t * b = r$
  assumes *euclid-ext-aux* s' s t' t r' r = $((x, y), c)$
  shows *euclid-ext-aux-eq-gcd*: c = *gcd* a b
    and *euclid-ext-aux-bezout*: $x * a + y * b$ = *gcd* a b
⟨*proof*⟩

declare *euclid-ext-aux.simps* [*simp del*]

definition *bezout-coefficients* :: $'a \Rightarrow 'a \Rightarrow 'a \times 'a$
  where [*code*]: *bezout-coefficients* a b = *fst* (*euclid-ext* a b)

lemma *bezout-coefficients-0*:
  bezout-coefficients a 0 = (1 *div unit-factor* a, 0)
  ⟨*proof*⟩

lemma *bezout-coefficients-left-0*:
  bezout-coefficients 0 a = (0, 1 *div unit-factor* a)
  ⟨*proof*⟩

lemma *bezout-coefficients*:
  assumes *bezout-coefficients* a b = (x, y)
  shows $x * a + y * b$ = *gcd* a b
  ⟨*proof*⟩

lemma *bezout-coefficients-fst-snd*:
  fst (*bezout-coefficients* a b) $* a +$ *snd* (*bezout-coefficients* a b) $* b$ = *gcd* a b
  ⟨*proof*⟩

lemma *euclid-ext-eq* [*simp*]:

euclid-ext a b = (*bezout-coefficients a b*, *gcd a b*) (**is** *?p* = *?q*)
⟨*proof*⟩

declare *euclid-ext-eq* [*symmetric*, *code-unfold*]

end

class *normalization-euclidean-semiring-multiplicative* =
normalization-euclidean-semiring + *normalization-semidom-multiplicative*
begin

subclass *factorial-semiring-multiplicative* ⟨*proof*⟩

end

class *field-gcd* =
field + *unique-euclidean-ring* + *euclidean-ring-gcd* + *normalization-semidom-multiplicative*
begin

subclass *normalization-euclidean-semiring-multiplicative* ⟨*proof*⟩

subclass *normalization-euclidean-semiring* ⟨*proof*⟩

subclass *semiring-gcd-mult-normalize* ⟨*proof*⟩

end

2.4 Typical instances

instance *nat* :: *normalization-euclidean-semiring* ⟨*proof*⟩

instance *nat* :: *euclidean-semiring-gcd*
⟨*proof*⟩

instance *nat* :: *normalization-euclidean-semiring-multiplicative* ⟨*proof*⟩

lemma *prime-factorization-Suc-0* [*simp*]: *prime-factorization* (*Suc 0*) = {#}
⟨*proof*⟩

instance *int* :: *normalization-euclidean-semiring* ⟨*proof*⟩

instance *int* :: *euclidean-ring-gcd*
⟨*proof*⟩

instance *int* :: *normalization-euclidean-semiring-multiplicative* ⟨*proof*⟩

lemma (**in** *idom*) *prime-CHAR-semidom*:
assumes *CHAR('a)* > *0*
shows *prime CHAR('a)*

⟨*proof*⟩

end

3 Primes

theory *Primes*
imports *Euclidean-Algorithm*
begin

3.1 Primes on *nat* and *int*

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

lemma *prime-ge-2-nat*:
$p \geq 2$ **if** *prime p* **for** *p* :: *nat*
⟨*proof*⟩

lemma *prime-ge-2-int*:
$p \geq 2$ **if** *prime p* **for** *p* :: *int*
⟨*proof*⟩

lemma *prime-ge-0-int*: *prime p* ⟹ $p \geq (0::int)$
⟨*proof*⟩

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

lemma *prime-gt-0-int*: *prime p* ⟹ $p > (0::int)$
⟨*proof*⟩

lemma *prime-ge-1-nat*: *prime p* ⟹ $p \geq (1::nat)$
⟨*proof*⟩

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

lemma *prime-ge-1-int*: *prime p* ⟹ $p \geq (1::int)$
⟨*proof*⟩

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

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

lemma *prime-gt-1-int*: *prime p* $\Longrightarrow$ *p* > (*1*::*int*)
⟨*proof*⟩

lemma *prime-natI*:
prime p **if** *p* ≥ *2* **and** $\bigwedge$*m n*. *p dvd m* ∗ *n* $\Longrightarrow$ *p dvd m* ∨ *p dvd n* **for** *p* :: *nat*
⟨*proof*⟩

lemma *prime-intI*:
prime p **if** *p* ≥ *2* **and** $\bigwedge$*m n*. *p dvd m* ∗ *n* $\Longrightarrow$ *p dvd m* ∨ *p dvd n* **for** *p* :: *int*
⟨*proof*⟩

lemma *prime-elem-nat-iff* [*simp*]:
prime-elem n ⟷ *prime n* **for** *n* :: *nat*
⟨*proof*⟩

lemma *prime-elem-iff-prime-abs* [*simp*]:
prime-elem k ⟷ *prime* |*k*| **for** *k* :: *int*
⟨*proof*⟩

lemma *prime-nat-int-transfer* [*simp*]:
prime (*int n*) ⟷ *prime n* (**is** *?P* ⟷ *?Q*)
⟨*proof*⟩

lemma *prime-nat-iff-prime* [*simp*]:
prime (*nat k*) ⟷ *prime k*
⟨*proof*⟩

lemma *prime-int-nat-transfer*:
prime k ⟷ *k* ≥ *0* ∧ *prime* (*nat k*)
⟨*proof*⟩

lemma *prime-nat-naiveI*:
prime p **if** *p* ≥ *2* **and** *dvd*: $\bigwedge$*n*. *n dvd p* $\Longrightarrow$ *n* = *1* ∨ *n* = *p* **for** *p* :: *nat*
⟨*proof*⟩

lemma *prime-int-naiveI*:
prime p **if** *p* ≥ *2* **and** *dvd*: $\bigwedge$*k*. *k dvd p* $\Longrightarrow$ |*k*| = *1* ∨ |*k*| = *p* **for** *p* :: *int*
⟨*proof*⟩

lemma *prime-nat-iff*:
prime (*n* :: *nat*) ⟷ (*1* < *n* ∧ (∀ *m*. *m dvd n* ⟶ *m* = *1* ∨ *m* = *n*))
⟨*proof*⟩

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

lemma *prime-nat-not-dvd*:
assumes *prime p p* > *n n* ≠ (*1*::*nat*)

shows $\neg n$ *dvd* p
⟨*proof*⟩

lemma *prime-int-not-dvd*:
assumes *prime* p $p > n$ $n > (1::int)$
shows $\neg n$ *dvd* p
⟨*proof*⟩

lemma *prime-odd-nat*: *prime* $p \Longrightarrow p > (2::nat) \Longrightarrow$ *odd* p
⟨*proof*⟩

lemma *prime-odd-int*: *prime* $p \Longrightarrow p > (2::int) \Longrightarrow$ *odd* p
⟨*proof*⟩

lemma *prime-int-altdef*:
prime $p = (1 < p \wedge (\forall m::int.\ m \geq 0 \longrightarrow m$ *dvd* $p \longrightarrow$
$m = 1 \vee m = p))$
⟨*proof*⟩

lemma *not-prime-eq-prod-nat*:
assumes $m > 1$ $\neg$ *prime* $(m::nat)$
shows $\exists n\ k.\ n = m * k \wedge 1 < m \wedge m < n \wedge 1 < k \wedge k < n$
⟨*proof*⟩

3.2 Largest exponent of a prime factor

Possibly duplicates other material, but avoid the complexities of multisets.

lemma *prime-power-cancel-less*:
assumes *prime* p **and** *eq*: $m * (p \mathbin{\hat{}} k) = m' * (p \mathbin{\hat{}} k')$ **and** *less*: $k < k'$ **and** $\neg$ p *dvd* m
shows *False*
⟨*proof*⟩

lemma *prime-power-cancel*:
assumes *prime* p **and** *eq*: $m * (p \mathbin{\hat{}} k) = m' * (p \mathbin{\hat{}} k')$ **and** $\neg$ p *dvd* m $\neg$ p *dvd* m'
shows $k = k'$
⟨*proof*⟩

lemma *prime-power-cancel2*:
assumes *prime* p $m * (p \mathbin{\hat{}} k) = m' * (p \mathbin{\hat{}} k')$ $\neg$ p *dvd* m $\neg$ p *dvd* m'
obtains $m = m'$ $k = k'$
⟨*proof*⟩

lemma *prime-power-canonical*:
fixes $m :: nat$
assumes *prime* p $m > 0$
shows $\exists k\ n.\ \neg\ p$ *dvd* $n \wedge m = n * p \mathbin{\hat{}} k$
⟨*proof*⟩

3.2.1 Make prime naively executable

lemma *prime-nat-iff'*:
  $prime\ (p :: nat) \longleftrightarrow p > 1 \wedge (\forall n \in \{2..<p\}.\ \neg\ n\ dvd\ p)$
⟨*proof*⟩

lemma *prime-int-iff'*:
  $prime\ (p :: int) \longleftrightarrow p > 1 \wedge (\forall n \in \{2..<p\}.\ \neg\ n\ dvd\ p)$ (**is** $?P \longleftrightarrow ?Q$)
⟨*proof*⟩

lemma *prime-int-numeral-eq* [*simp*]:
  $prime\ (numeral\ m :: int) \longleftrightarrow prime\ (numeral\ m :: nat)$
  ⟨*proof*⟩

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

lemma *prime-nat-numeral-eq* [*simp*]:
  $prime\ (numeral\ m :: nat) \longleftrightarrow$
    $(1::nat) < numeral\ m \wedge$
    $(\forall n::nat \in set\ [2..<numeral\ m].\ \neg\ n\ dvd\ numeral\ m)$
  ⟨*proof*⟩

A bit of regression testing:

lemma $prime(97::nat)$ ⟨*proof*⟩
lemma $prime(97::int)$ ⟨*proof*⟩

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

lemma *prime-factor-int*:
  fixes $k :: int$
  assumes $|k| \neq 1$
  obtains p **where** $prime\ p\ p\ dvd\ k$
⟨*proof*⟩

3.3 Infinitely many primes

lemma *next-prime-bound*: $\exists p::nat.\ prime\ p \wedge n < p \wedge p \leq fact\ n + 1$
⟨*proof*⟩

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

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

3.4 Powers of Primes

Versions for type nat only

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

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

lemma *prime-power-exp-nat*:
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-nat*:
fixes $p :: nat$
assumes p: $prime\ p$
shows $d\ dvd\ p\ \hat{}\ k \longleftrightarrow (\exists\, i{\le}k.\ d = p\ \hat{}\ i)$
⟨*proof*⟩

3.5 Chinese Remainder Theorem Variants

lemma *bezout-gcd-nat*:
fixes $a::nat$ **shows** $\exists\, x\ y.\ a * x - b * y = gcd\ a\ b \lor b * x - a * y = gcd\ a\ b$
⟨*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 $\exists\, x\ q1\ q2.\ x = u + q1 * a \land x = v + q2 * b$
⟨*proof*⟩

Primality

lemma *coprime-bezout-strong*:
fixes $a::nat$ **assumes** $coprime\ a\ b\quad b \neq 1$

shows $\exists x\ y.\ a * x = b * y + 1$
⟨*proof*⟩

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

3.6 Multiplicity and primality for natural numbers and integers

lemma *prime-factors-gt-0-nat*:
$p \in prime\text{-}factors\ x \Longrightarrow p > (0{::}nat)$
⟨*proof*⟩

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

lemma *prime-factors-ge-0-int* [*elim*]:
fixes *n* :: *int*
shows $p \in prime\text{-}factors\ n \Longrightarrow p \geq 0$
⟨*proof*⟩

lemma *prod-mset-prime-factorization-int*:
fixes *n* :: *int*
assumes $n > 0$
shows *prod-mset* (*prime-factorization n*) = *n*
⟨*proof*⟩

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

lemma *prod-mset-prime-factorization-nat* [*simp*]:
$(n{::}nat) > 0 \Longrightarrow prod\text{-}mset\ (prime\text{-}factorization\ n) = n$
⟨*proof*⟩

lemma *prime-factorization-nat*:
$n > (0{::}nat) \Longrightarrow n = (\prod p \in prime\text{-}factors\ n.\ p\ \widehat{}\ multiplicity\ p\ n)$
⟨*proof*⟩

lemma *prime-factorization-int*:
$n > (0{::}int) \Longrightarrow n = (\prod p \in prime\text{-}factors\ n.\ p\ \widehat{}\ multiplicity\ p\ n)$
⟨*proof*⟩

lemma *prime-factorization-unique-nat*:
fixes *f* :: *nat* ⇒ -
assumes *S-eq*: $S = \{p.\ 0 < f\ p\}$

and *finite S*
and *S*: $\forall p{\in}S.\ prime\ p\ n = (\prod p{\in}S.\ p\ \hat{}\ f\ p)$
shows $S = prime\text{-}factors\ n \wedge (\forall p.\ prime\ p \longrightarrow f\ p = multiplicity\ p\ n)$
⟨*proof*⟩

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

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

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

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

lemma *abs-prod*: $abs\ (prod\ f\ A :: 'a :: linordered\text{-}idom) = prod\ (\lambda x.\ abs\ (f\ x))\ A$
⟨*proof*⟩

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

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

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

lemma *multiplicity-characterization-int*: $S = \{p.\ 0 < f\ (p::int)\} \Longrightarrow$

finite S $\Longrightarrow$ $\forall$ *p*$\in$*S*. *prime p* $\Longrightarrow$ *prime p* $\Longrightarrow$ *n* = ($\prod$*p*$\in$*S*. *p* ^ *f p*) $\Longrightarrow$ *multiplicity p n* = *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*) $\Longrightarrow$ *prime p* $\Longrightarrow$
multiplicity p ($\prod$ *p* | *p* $\geq$ *0* $\wedge$ *0* < *f p*. *p* ^ *f p*) = *f p*
⟨*proof*⟩

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

lemma *multiplicity-eq-nat*:
fixes *x* **and** *y*::*nat*
assumes *x* > *0* *y* > *0* $\bigwedge$*p*. *prime p* $\Longrightarrow$ *multiplicity p x* = *multiplicity p y*
shows *x* = *y*
⟨*proof*⟩

lemma *multiplicity-eq-int*:
fixes *x y* :: *int*
assumes *x* > *0* *y* > *0* $\bigwedge$*p*. *prime p* $\Longrightarrow$ *multiplicity p x* = *multiplicity p y*
shows *x* = *y*
⟨*proof*⟩

lemma *multiplicity-prod-prime-powers*:
assumes *finite S* $\bigwedge$*x*. *x* $\in$ *S* $\Longrightarrow$ *prime x prime p*
shows *multiplicity p* ($\prod$ *p* $\in$ *S*. *p* ^ *f p*) = (*if p* $\in$ *S then f p else 0*)
⟨*proof*⟩

lemma *prime-factorization-prod-mset*:
assumes *0* $\notin$# *A*
shows *prime-factorization* (*prod-mset A*) = $\sum_{\#}$(*image-mset prime-factorization A*)
⟨*proof*⟩

lemma *prime-factors-prod*:
assumes *finite A* **and** *0* $\notin$ *f* ‘ *A*
shows *prime-factors* (*prod f A*) = $\bigcup$((*prime-factors* $\circ$ *f*) ‘ *A*)
⟨*proof*⟩

lemma *prime-factors-fact*:
prime-factors (*fact n*) = {*p* $\in$ {*2*..*n*}. *prime p*} (**is** *?M* = *?N*)
⟨*proof*⟩

lemma *prime-dvd-fact-iff*:
assumes *prime p*
shows *p dvd fact n* $\longleftrightarrow$ *p* $\leq$ *n*
⟨*proof*⟩

lemma *dvd-choose-prime*:
  assumes *kn*: $k < n$ **and** *k*: $k \neq 0$ **and** *n*: $n \neq 0$ **and** *prime-n*: *prime n*
  shows *n dvd* (*n choose k*)
⟨*proof*⟩

lemma (**in** *ring-1*) *minus-power-prime-CHAR*:
  assumes $p = CHAR('a)$ *prime p*
  shows $(-x :: 'a) \mathbin{\char`\^} p = -(x \mathbin{\char`\^} p)$
⟨*proof*⟩

3.7 Rings and fields with prime characteristic

We introduce some type classes for rings and fields with prime characteristic.

class *semiring-prime-char* = *semiring-1* +
  assumes *prime-char-aux*: ∃ *n*. *prime n* ∧ *of-nat n* = (*0* :: *'a*)
begin

lemma *CHAR-pos* [*intro*, *simp*]: *CHAR*(*'a*) > *0*
  ⟨*proof*⟩

lemma *CHAR-nonzero* [*simp*]: *CHAR*(*'a*) ≠ *0*
  ⟨*proof*⟩

lemma *CHAR-prime* [*intro*, *simp*]: *prime CHAR*(*'a*)
  ⟨*proof*⟩

end

lemma *semiring-prime-charI* [*intro?*]:
  prime CHAR(*'a* :: *semiring-1*) ⟹ *OFCLASS*(*'a*, *semiring-prime-char-class*)
  ⟨*proof*⟩

lemma *idom-prime-charI* [*intro?*]:
  assumes *CHAR*(*'a* :: *idom*) > *0*
  shows *OFCLASS*(*'a*, *semiring-prime-char-class*)
⟨*proof*⟩

class *comm-semiring-prime-char* = *comm-semiring-1* + *semiring-prime-char*
class *comm-ring-prime-char* = *comm-ring-1* + *semiring-prime-char*
begin
subclass *comm-semiring-prime-char* ⟨*proof*⟩
end
class *idom-prime-char* = *idom* + *semiring-prime-char*
begin
subclass *comm-ring-prime-char* ⟨*proof*⟩
end

class *field-prime-char* = *field* +

assumes *pos-char-exists*: $\exists\, n>0.$ *of-nat* $n = (0 :: 'a)$
begin
subclass *idom-prime-char*
⟨*proof*⟩
end

lemma *field-prime-charI* [*intro?*]:
$n > 0 \Longrightarrow$ *of-nat* $n = (0 :: 'a :: field) \Longrightarrow OFCLASS('a,$ *field-prime-char-class*$)$
⟨*proof*⟩

lemma *field-prime-charI'* [*intro?*]:
$CHAR('a :: field) > 0 \Longrightarrow OFCLASS('a,$ *field-prime-char-class*$)$
⟨*proof*⟩

3.8 Finite fields

class *finite-field* = *field-prime-char* + *finite*

lemma *finite-fieldI* [*intro?*]:
assumes *finite* (*UNIV* :: *'a* :: *field set*)
shows $OFCLASS('a,$ *finite-field-class*$)$
⟨*proof*⟩

On a finite field with n elements, taking the n-th power of an element is the identity. This is an obvious consequence of the fact that the multiplicative group of the field is a finite group of order $n - 1$, so $x \hat{} n = 1$ for any non-zero x.

Note that this result is sharp in the sense that the multiplicative group of a finite field is cyclic, i.e. it contains an element of order $n - 1$. (We don't prove this here.)

lemma *finite-field-power-card-eq-same*:
fixes x :: *'a* :: *finite-field*
shows $x \hat{}$ *card* (*UNIV* :: *'a set*) $= x$
⟨*proof*⟩

lemma *finite-field-power-card-power-eq-same*:
fixes x :: *'a* :: *finite-field*
assumes m = *card* (*UNIV* :: *'a set*) $\hat{}\ n$
shows $x \hat{}\ m = x$
⟨*proof*⟩

class *enum-finite-field* = *finite-field* +
fixes *enum-finite-field* :: *nat* ⇒ *'a*
assumes *enum-finite-field*: *enum-finite-field* ‘ {..<*card* (*UNIV* :: *'a set*)} = *UNIV*
begin

lemma *inj-on-enum-finite-field*: *inj-on enum-finite-field* {..<*card* (*UNIV* :: *'a set*)}
⟨*proof*⟩

end

To get rid of the pending sort hypotheses, we prove that the field with 2 elements is indeed a finite field.

typedef *gf2* = {*0*, *1* :: *nat*}
⟨*proof*⟩

setup-lifting *type-definition-gf2*

instantiation *gf2* :: *field*
begin
lift-definition *zero-gf2* :: *gf2* **is** *0* ⟨*proof*⟩
lift-definition *one-gf2* :: *gf2* **is** *1* ⟨*proof*⟩
lift-definition *uminus-gf2* :: *gf2* ⇒ *gf2* **is** *λx. x* ⟨*proof*⟩
lift-definition *plus-gf2* :: *gf2* ⇒ *gf2* ⇒ *gf2* **is** *λx y. if x = y then 0 else 1* ⟨*proof*⟩
lift-definition *minus-gf2* :: *gf2* ⇒ *gf2* ⇒ *gf2* **is** *λx y. if x = y then 0 else 1* ⟨*proof*⟩
lift-definition *times-gf2* :: *gf2* ⇒ *gf2* ⇒ *gf2* **is** *λx y. x ∗ y* ⟨*proof*⟩
lift-definition *inverse-gf2* :: *gf2* ⇒ *gf2* **is** *λx. x* ⟨*proof*⟩
lift-definition *divide-gf2* :: *gf2* ⇒ *gf2* ⇒ *gf2* **is** *λx y. x ∗ y* ⟨*proof*⟩

instance
⟨*proof*⟩

end

instance *gf2* :: *finite-field*
⟨*proof*⟩

3.9 The Freshman's Dream in rings of prime characteristic

lemma (**in** *comm-semiring-1*) *freshmans-dream*:
fixes *x y* :: *'a* **and** *n* :: *nat*
assumes *prime CHAR('a)*
assumes *n-def*: *n = CHAR('a)*
shows (*x* + *y*) ^ *n* = *x* ^ *n* + *y* ^ *n*
⟨*proof*⟩

lemma (**in** *comm-semiring-1*) *freshmans-dream'*:
assumes [*simp*]: *prime CHAR('a)* **and** *m = CHAR('a) ^ n*
shows (*x* + *y* :: *'a*) ^ *m* = *x* ^ *m* + *y* ^ *m*
⟨*proof*⟩

lemma (**in** *comm-semiring-1*) *freshmans-dream-sum*:
fixes *f* :: *'b* ⇒ *'a*
assumes *prime CHAR('a)* **and** *n = CHAR('a)*
shows *sum f A* ^ *n* = *sum* (*λi. f i* ^ *n*) *A*
⟨*proof*⟩

lemma (**in** *comm-semiring-1*) *freshmans-dream-sum'*:
  fixes *f* :: *'b* ⇒ *'a*
  assumes *prime CHAR('a) m = CHAR('a) ^ n*
  shows *sum f A ^ m = sum (λi. f i ^ m) A*
  ⟨*proof*⟩

lemmas *prime-imp-coprime-nat = prime-imp-coprime*[**where** *?'a = nat*]
lemmas *prime-imp-coprime-int = prime-imp-coprime*[**where** *?'a = int*]
lemmas *prime-dvd-mult-nat = prime-dvd-mult-iff*[**where** *?'a = nat*]
lemmas *prime-dvd-mult-int = prime-dvd-mult-iff*[**where** *?'a = int*]
lemmas *prime-dvd-mult-eq-nat = prime-dvd-mult-iff*[**where** *?'a = nat*]
lemmas *prime-dvd-mult-eq-int = prime-dvd-mult-iff*[**where** *?'a = int*]
lemmas *prime-dvd-power-nat = prime-dvd-power*[**where** *?'a = nat*]
lemmas *prime-dvd-power-int = prime-dvd-power*[**where** *?'a = int*]
lemmas *prime-dvd-power-nat-iff = prime-dvd-power-iff*[**where** *?'a = nat*]
lemmas *prime-dvd-power-int-iff = prime-dvd-power-iff*[**where** *?'a = int*]
lemmas *prime-imp-power-coprime-nat = prime-imp-power-coprime*[**where** *?'a = nat*]
lemmas *prime-imp-power-coprime-int = prime-imp-power-coprime*[**where** *?'a = int*]
lemmas *primes-coprime-nat = primes-coprime*[**where** *?'a = nat*]
lemmas *primes-coprime-int = primes-coprime*[**where** *?'a = nat*]
lemmas *prime-divprod-pow-nat = prime-elem-divprod-pow*[**where** *?'a = nat*]
lemmas *prime-exp = prime-elem-power-iff*[**where** *?'a = nat*]

Code generation

context
begin

qualified definition *prime-nat* :: *nat* ⇒ *bool*
  where [*simp*, *code-abbrev*]: *prime-nat = prime*

lemma *prime-nat-naive* [*code*]:
  prime-nat p ⟷ p > 1 ∧ (∀ n ∈{1<..<p}. ¬ n dvd p)
  ⟨*proof*⟩ **definition** *prime-int* :: *int* ⇒ *bool*
  where [*simp*, *code-abbrev*]: *prime-int = prime*

lemma *prime-int-naive* [*code*]:
  prime-int p ⟷ p > 1 ∧ (∀ n ∈{1<..<p}. ¬ n dvd p)
  ⟨*proof*⟩

lemma *prime(997::nat)* ⟨*proof*⟩

lemma *prime(997::int)* ⟨*proof*⟩

end

end

4 Polynomials as type over a ring structure

theory *Polynomial*
imports
Complex-Main
HOL−Library.More-List
HOL−Library.Infinite-Set
Primes
begin

context *semidom-modulo*
begin

lemma *not-dvd-imp-mod-neq-0*:
‹*a mod b* ≠ *0*› **if** ‹¬ *b dvd a*›
⟨*proof*⟩

end

4.1 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* ≠ *0* ⟹ *x ## xs* = *x # xs*
⟨*proof*⟩

lemma *strip-while-not-0-Cons-eq* [*simp*]:
strip-while (λ*x*. *x* = *0*) (*x # xs*) = *x ## strip-while* (λ*x*. *x* = *0*) *xs*
⟨*proof*⟩

lemma *tl-cCons* [*simp*]: *tl* (*x ## xs*) = *xs*
⟨*proof*⟩

4.2 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*⟩

lemma *coeff-Abs-poly*:
  assumes $\bigwedge i.\ i > n \Longrightarrow f\ i = 0$
  shows $coeff\ (Abs\text{-}poly\ f) = f$
⟨*proof*⟩

4.3 Degree of a polynomial

definition $degree :: 'a::zero\ poly \Rightarrow nat$
  where $degree\ p = (LEAST\ n.\ \forall\, i>n.\ coeff\ p\ i = 0)$

lemma *degree-cong*:
  assumes $\bigwedge i.\ coeff\ p\ i = 0 \longleftrightarrow coeff\ q\ i = 0$
  shows $degree\ p = degree\ q$
⟨*proof*⟩

lemma *coeff-Abs-poly-If-le*:
  $coeff\ (Abs\text{-}poly\ (\lambda i.\ if\ i \le n\ then\ f\ i\ else\ 0)) = (\lambda i.\ if\ i \le n\ then\ f\ i\ else\ 0)$
⟨*proof*⟩

lemma *coeff-eq-0*:
  assumes $degree\ p < n$
  shows $coeff\ p\ n = 0$
⟨*proof*⟩

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

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

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

4.4 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*⟩

lemma *degree-lessI*:
 assumes *p ≠ 0 ∨ n > 0 ∀ k≥n. coeff p k = 0*
 shows *degree p < n*
⟨*proof*⟩

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

lemma *coeff-0-degree-minus-1*: *coeff rrr dr = 0 ⟹ degree rrr ≤ dr ⟹ degree rrr ≤ dr − 1*
 ⟨*proof*⟩

4.5 List-style constructor for polynomials

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

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

lemma *coeff-pCons'*: *poly.coeff* (*pCons c p*) *n =* (*if n = 0 then c else poly.coeff p*

$(n - 1))$
⟨*proof*⟩

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*⟩

4.6 Quickcheck generator for polynomials

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

4.7 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*)

4.8 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 *Poly-append-replicate-zero* [*simp*]: *Poly* (*as* @ *replicate n 0*) = *Poly as*
  ⟨*proof*⟩

lemma *Poly-snoc-zero* [*simp*]: *Poly* (*as* @ [*0*]) = *Poly as*
  ⟨*proof*⟩

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

lemma *Poly-on-rev-starting-with-0* [*simp*]: *hd as* = *0* ⟹ *Poly* (*rev* (*tl as*)) = *Poly* (*rev as*)
  ⟨*proof*⟩

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

lemma *coeff-Poly-eq* [*simp*]: *coeff* (*Poly xs*) = *nth-default 0 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*: $p \neq 0 \Longrightarrow coeffs\ p \neq []$
⟨*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*: $p \neq 0 \Longrightarrow length\ (coeffs\ p) = degree\ p + 1$
⟨*proof*⟩

lemma *coeffs-nth*: $p \neq 0 \Longrightarrow n \leq degree\ p \Longrightarrow coeffs\ p\ !\ n = coeff\ p\ n$
⟨*proof*⟩

lemma *coeff-in-coeffs*: $p \neq 0 \Longrightarrow n \leq degree\ p \Longrightarrow coeff\ p\ n \in set\ (coeffs\ p)$
⟨*proof*⟩

lemma *not-0-cCons-eq* [*simp*]: $p \neq 0 \Longrightarrow a\ \#\#\ coeffs\ p = a\ \#\ coeffs\ p$
⟨*proof*⟩

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

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

lemma *no-trailing-coeffs* [*simp*]:
$no\text{-}trailing\ (HOL.eq\ 0)\ (coeffs\ p)$
⟨*proof*⟩

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

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

lemma *nth-default-coeffs-eq*: $nth\text{-}default\ 0\ (coeffs\ p) = coeff\ p$
⟨*proof*⟩

lemma [*code*]: $coeff\ p = nth\text{-}default\ 0\ (coeffs\ p)$
⟨*proof*⟩

lemma *coeffs-eqI*:
assumes *coeff*: $\bigwedge n.\ coeff\ p\ n = nth\text{-}default\ 0\ xs\ n$
assumes *zero*: $no\text{-}trailing\ (HOL.eq\ 0)\ xs$
shows $coeffs\ p = xs$

⟨*proof*⟩

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

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

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

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

lemma *set-coeffs-subset-singleton-0-iff* [*simp*]:
set (*coeffs p*) ⊆ {*0*} ⟷ *p* = *0*
⟨*proof*⟩

lemma *set-coeffs-not-only-0* [*simp*]:
set (*coeffs p*) ≠ {*0*}
⟨*proof*⟩

lemma *forall-coeffs-conv*:
(∀ *n*. *P* (*coeff p n*)) ⟷ (∀ *c* ∈ *set* (*coeffs p*). *P c*) **if** *P 0*
⟨*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*⟩

Reconstructing the polynomial from the list

definition *poly-of-list* :: ′*a*::*comm-monoid-add list* ⇒ ′*a poly*
where [*simp*]: *poly-of-list* = *Poly*

lemma *poly-of-list-impl* [*code abstract*]: *coeffs* (*poly-of-list as*) = *strip-while* (*HOL.eq* *0*) *as*
⟨*proof*⟩

4.9 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 ≠ *0* ⟹ *fold-coeffs f* (*pCons a p*) = *f a* ∘ *fold-coeffs f p*
⟨*proof*⟩

4.10 Canonical morphism on polynomials – evaluation

definition *poly* :: ‹*'a::comm-semiring-0 poly* ⇒ *'a* ⇒ *'a*›
where ‹*poly p a* = *horner-sum id a* (*coeffs p*)›

lemma *poly-eq-fold-coeffs*:
‹*poly p* = *fold-coeffs* (λ*a f x*. *a* + *x* ∗ *f x*) *p* (λ*x*. *0*)›
⟨*proof*⟩

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

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

4.11 Monomials

lift-definition *monom* :: *'a ⇒ nat ⇒ 'a::zero poly*
is *λa 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* = [:*a*:]
⟨*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*: *poly* (*monom a n*) *x* = *a* ∗ *x* ^ *n*
for *a x* :: *'a::comm-semiring-1*
⟨*proof*⟩

lemma *monom-eq-iff'*: *monom c n* = *monom d m* ⟷ *c* = *d* ∧ (*c* = *0* ∨ *n* = *m*)
⟨*proof*⟩

lemma *monom-eq-const-iff*: *monom c n* = [:*d*:] ⟷ *c* = *d* ∧ (*c* = *0* ∨ *n* = *0*)
⟨*proof*⟩

4.12 Leading coefficient

abbreviation *lead-coeff* :: $'a$::*zero poly* ⇒ $'a$
  where *lead-coeff p* ≡ *coeff p* (*degree p*)

lemma *lead-coeff-pCons*[*simp*]:
  $p \neq 0 \Longrightarrow$ *lead-coeff* (*pCons a p*) = *lead-coeff p*
  $p = 0 \Longrightarrow$ *lead-coeff* (*pCons a p*) = *a*
  ⟨*proof*⟩

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

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

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

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

lemma *degree-diff-le-max*: *degree* (*p* − *q*) ≤ *max* (*degree p*) (*degree q*)

for *p q* :: *'a::ab-group-add poly*
⟨*proof*⟩

lemma *degree-diff-le*: *degree p* ≤ *n* ⟹ *degree q* ≤ *n* ⟹ *degree* (*p* − *q*) ≤ *n*
for *p q* :: *'a::ab-group-add poly*
⟨*proof*⟩

lemma *degree-diff-less*: *degree p* < *n* ⟹ *degree q* < *n* ⟹ *degree* (*p* − *q*) < *n*
for *p q* :: *'a::ab-group-add poly*
⟨*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-sum*: *coeff* (∑ *x*∈*A*. *p x*) *i* = (∑ *x*∈*A*. *coeff* (*p x*) *i*)
⟨*proof*⟩

lemma *monom-sum*: *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 uminus* (*coeffs p*)
⟨*proof*⟩

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

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

lemma *poly-minus* [*simp*]: *poly* (− *p*) *x* = − *poly p x*
for *x* :: *'a::comm-ring*

⟨*proof*⟩

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

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

lemma *poly-sum-list*: *poly* (∑ *p*←*ps*. *p*) *y* = (∑ *p*←*ps*. *poly p y*)
⟨*proof*⟩

lemma *poly-sum-mset*: *poly* (∑ *x*∈#*A*. *p x*) *y* = (∑ *x*∈#*A*. *poly* (*p x*) *y*)
⟨*proof*⟩

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

lemma *degree-sum-less*:
assumes ⋀*x*. *x* ∈ *A* ⟹ *degree* (*f x*) < *n n* > *0*
shows *degree* (*sum f A*) < *n*
⟨*proof*⟩

lemma *poly-as-sum-of-monoms'*:
assumes *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*⟩

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

lift-definition *smult* :: *'a::comm-semiring-0* ⇒ *'a poly* ⇒ *'a poly*
is *λa 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* (− *p*) = − *smult a p*
for *a* :: ′*a*::*comm-ring*
⟨*proof*⟩

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

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

lemma *smult-diff-left*: *smult* (*a* − *b*) *p* = *smult a p* − *smult b p*
for *a b* :: ′*a*::*comm-ring*
⟨*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 *smult-Poly*: *smult c* (*Poly xs*) = *Poly* (*map* ((∗) *c*) *xs*)
⟨*proof*⟩

lemma *degree-smult-eq* [*simp*]: *degree* (*smult a p*) = (*if a* = *0 then 0 else degree p*)
for *a* :: ′*a*::{*comm-semiring-0*,*semiring-no-zero-divisors*}
⟨*proof*⟩

lemma *smult-eq-0-iff* [*simp*]: *smult a p* = *0* ⟷ *a* = *0* ∨ *p* = *0*
 for *a* :: *′a*::{*comm-semiring-0*,*semiring-no-zero-divisors*}
 ⟨*proof*⟩

lemma *coeffs-smult* [*code abstract*]:
 coeffs (*smult a p*) = (*if a* = *0 then* [] *else map* (*Groups.times a*) (*coeffs p*))
 for *p* :: *′a*::{*comm-semiring-0*,*semiring-no-zero-divisors*} *poly*
⟨*proof*⟩

lemma *smult-eq-iff*:
 fixes *b* :: *′a* :: *field*
 assumes *b* ≠ *0*
 shows *smult a p* = *smult b q* ⟷ *smult* (*a* / *b*) *p* = *q*
  (**is** *?lhs* ⟷ *?rhs*)
⟨*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*: (*p* + *q*) ∗ *r* = *p* ∗ *r* + *q* ∗ *r*
 for *p q r* :: *′a poly*
 ⟨*proof*⟩

instance
⟨*proof*⟩

end

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

instance *poly* :: ({*comm-semiring-0*,*semiring-no-zero-divisors*}) *semiring-no-zero-divisors*
⟨*proof*⟩

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

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

lemma *coeff-mult-0*: $coeff\ (p * q)\ 0 = coeff\ p\ 0 * coeff\ q\ 0$
⟨*proof*⟩

lemma *degree-mult-le*: $degree\ (p * q) \leq degree\ p + degree\ q$
⟨*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

lift-definition *one-poly* :: *'a poly*
is $\lambda n.\ of\text{-}bool\ (n = 0)$
⟨*proof*⟩

lemma *coeff-1* [*simp*]:
$coeff\ 1\ n = of\text{-}bool\ (n = 0)$
⟨*proof*⟩

lemma *one-pCons*:
$1 = [:1:]$
⟨*proof*⟩

lemma *pCons-one*:
$[:1:] = 1$
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *poly-1* [*simp*]:
$poly\ 1\ x = 1$

⟨*proof*⟩

lemma *one-poly-eq-simps* [*simp*]:
1 = [:*1*:] ⟷ *True*
[:*1*:] = *1* ⟷ *True*
⟨*proof*⟩

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

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

lemma *smult-one* [*simp*]:
smult c 1 = [:*c*:]
⟨*proof*⟩

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

lemma *monom-eq-1-iff*:
monom c n = *1* ⟷ *c* = *1* ∧ *n* = *0*
⟨*proof*⟩

lemma *monom-altdef*:
monom c n = *smult c* ([:*0*, *1*:] ^ *n*)
⟨*proof*⟩

instance *poly* :: ({*comm-semiring-1*,*semiring-1-no-zero-divisors*}) *semiring-1-no-zero-divisors*
⟨*proof*⟩
instance *poly* :: (*comm-ring*) *comm-ring* ⟨*proof*⟩
instance *poly* :: (*comm-ring-1*) *comm-ring-1* ⟨*proof*⟩
instance *poly* :: (*comm-ring-1*) *comm-semiring-1-cancel* ⟨*proof*⟩

lemma *prod-smult*: (∏ *x*∈*A*. *smult* (*c x*) (*p x*)) = *smult* (*prod c A*) (*prod p A*)
⟨*proof*⟩

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

lemma *coeff-0-power*: *coeff* (*p* ^ *n*) *0* = *coeff p 0* ^ *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-power* [*simp*]: *poly* (*p* ^ *n*) *x* = *poly p x* ^ *n*
for *p* :: ′*a*::*comm-semiring-1 poly*
⟨*proof*⟩

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

lemma *poly-prod-list*: *poly* (∏ *p*←*ps*. *p*) *y* = (∏ *p*←*ps*. *poly p y*)
⟨*proof*⟩

lemma *poly-prod-mset*: *poly* (∏ *x*∈#*A*. *p x*) *y* = (∏ *x*∈#*A*. *poly* (*p x*) *y*)
⟨*proof*⟩

lemma *poly-const-pow*: [: *c* :] ^ *n* = [: *c* ^ *n* :]
⟨*proof*⟩

lemma *monom-power*: *monom c n* ^ *k* = *monom* (*c* ^ *k*) (*n* ∗ *k*)
⟨*proof*⟩

lemma *degree-prod-sum-le*: *finite S* ⟹ *degree* (*prod f S*) ≤ *sum* (*degree* ∘ *f*) *S*
⟨*proof*⟩

lemma *coeff-0-prod-list*: *coeff* (*prod-list xs*) *0* = *prod-list* (*map* (λ*p*. *coeff p 0*) *xs*)
⟨*proof*⟩

lemma *coeff-monom-mult*: *coeff* (*monom c n* ∗ *p*) *k* = (*if k* < *n then 0 else c* ∗ *coeff p* (*k* − *n*))
⟨*proof*⟩

lemma *monom-1-dvd-iff*′: *monom 1 n dvd p* ⟷ (∀ *k*<*n*. *coeff p k* = *0*)
⟨*proof*⟩

4.15 Mapping polynomials

definition *map-poly* :: (′*a* :: *zero* ⇒ ′*b* :: *zero*) ⇒ ′*a poly* ⇒ ′*b poly*
where *map-poly f p* = *Poly* (*map f* (*coeffs p*))

lemma *map-poly-0* [*simp*]: *map-poly f 0* = *0*
⟨*proof*⟩

lemma *map-poly-1*: *map-poly f 1* = [:*f 1*:]
⟨*proof*⟩

lemma *map-poly-1*′ [*simp*]: *f 1* = *1* ⟹ *map-poly f 1* = *1*
⟨*proof*⟩

lemma *coeff-map-poly*:
assumes $f\ 0 = 0$
shows $coeff\ (map\text{-}poly\ f\ p)\ n = f\ (coeff\ p\ n)$
⟨*proof*⟩

lemma *coeffs-map-poly* [*code abstract*]:
$coeffs\ (map\text{-}poly\ f\ p) = strip\text{-}while\ ((=)\ 0)\ (map\ f\ (coeffs\ p))$
⟨*proof*⟩

lemma *coeffs-map-poly′*:
assumes $\bigwedge x.\ x \neq 0 \Longrightarrow f\ x \neq 0$
shows $coeffs\ (map\text{-}poly\ f\ p) = map\ f\ (coeffs\ p)$
⟨*proof*⟩

lemma *set-coeffs-map-poly*:
$(\bigwedge x.\ f\ x = 0 \longleftrightarrow x = 0) \Longrightarrow set\ (coeffs\ (map\text{-}poly\ f\ p)) = f\ `\ set\ (coeffs\ p)$
⟨*proof*⟩

lemma *degree-map-poly*:
assumes $\bigwedge x.\ x \neq 0 \Longrightarrow f\ x \neq 0$
shows $degree\ (map\text{-}poly\ f\ p) = degree\ p$
⟨*proof*⟩

lemma *map-poly-eq-0-iff*:
assumes $f\ 0 = 0\ \bigwedge x.\ x \in set\ (coeffs\ p) \Longrightarrow x \neq 0 \Longrightarrow f\ x \neq 0$
shows $map\text{-}poly\ f\ p = 0 \longleftrightarrow p = 0$
⟨*proof*⟩

lemma *map-poly-smult*:
assumes $f\ 0 = 0 \bigwedge c\ x.\ f\ (c * x) = f\ c * f\ x$
shows $map\text{-}poly\ f\ (smult\ c\ p) = smult\ (f\ c)\ (map\text{-}poly\ f\ p)$
⟨*proof*⟩

lemma *map-poly-pCons*:
assumes $f\ 0 = 0$
shows $map\text{-}poly\ f\ (pCons\ c\ p) = pCons\ (f\ c)\ (map\text{-}poly\ f\ p)$
⟨*proof*⟩

lemma *map-poly-map-poly*:
assumes $f\ 0 = 0\ g\ 0 = 0$
shows $map\text{-}poly\ f\ (map\text{-}poly\ g\ p) = map\text{-}poly\ (f \circ g)\ p$
⟨*proof*⟩

lemma *map-poly-id* [*simp*]: $map\text{-}poly\ id\ p = p$
⟨*proof*⟩

lemma *map-poly-id′* [*simp*]: $map\text{-}poly\ (\lambda x.\ x)\ p = p$
⟨*proof*⟩

lemma *map-poly-cong*:
assumes $(\bigwedge x.\ x \in set\ (coeffs\ p) \Longrightarrow f\ x = g\ x)$
shows $map\text{-}poly\ f\ p = map\text{-}poly\ g\ p$
⟨*proof*⟩

lemma *map-poly-monom*: $f\ 0 = 0 \Longrightarrow map\text{-}poly\ f\ (monom\ c\ n) = monom\ (f\ c)\ n$
⟨*proof*⟩

lemma *map-poly-idI*:
assumes $\bigwedge x.\ x \in set\ (coeffs\ p) \Longrightarrow f\ x = x$
shows $map\text{-}poly\ f\ p = p$
⟨*proof*⟩

lemma *map-poly-idI'*:
assumes $\bigwedge x.\ x \in set\ (coeffs\ p) \Longrightarrow f\ x = x$
shows $p = map\text{-}poly\ f\ p$
⟨*proof*⟩

lemma *smult-conv-map-poly*: $smult\ c\ p = map\text{-}poly\ (\lambda x.\ c * x)\ p$
⟨*proof*⟩

lemma *poly-cnj*: $cnj\ (poly\ p\ z) = poly\ (map\text{-}poly\ cnj\ p)\ (cnj\ z)$
⟨*proof*⟩

lemma *poly-cnj-real*:
assumes $\bigwedge n.\ poly.coeff\ p\ n \in \mathbb{R}$
shows $cnj\ (poly\ p\ z) = poly\ p\ (cnj\ z)$
⟨*proof*⟩

lemma *real-poly-cnj-root-iff*:
assumes $\bigwedge n.\ poly.coeff\ p\ n \in \mathbb{R}$
shows $poly\ p\ (cnj\ z) = 0 \longleftrightarrow poly\ p\ z = 0$
⟨*proof*⟩

lemma *sum-to-poly*: $(\sum x{\in}A.\ [:f\ x:]) = [:\sum x{\in}A.\ f\ x:]$
⟨*proof*⟩

lemma *diff-to-poly*: $[:c:] - [:d:] = [:c - d:]$
⟨*proof*⟩

lemma *mult-to-poly*: $[:c:] * [:d:] = [:c * d:]$
⟨*proof*⟩

lemma *prod-to-poly*: $(\prod x{\in}A.\ [:f\ x:]) = [:\prod x{\in}A.\ f\ x:]$
⟨*proof*⟩

lemma *poly-map-poly-cnj* [*simp*]: $poly\ (map\text{-}poly\ cnj\ p)\ x = cnj\ (poly\ p\ (cnj\ x))$
⟨*proof*⟩

4.16 Conversions

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

lemma *of-nat-monom*:
of-nat n = *monom* (*of-nat n*) *0*
⟨*proof*⟩

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

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

lemma *of-int-poly*:
of-int k = [:*of-int k*:]
⟨*proof*⟩

lemma *of-int-monom*:
of-int k = *monom* (*of-int k*) *0*
⟨*proof*⟩

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

lemma *lead-coeff-of-int* [*simp*]:
lead-coeff (*of-int k*) = *of-int k*
⟨*proof*⟩

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

lemma *poly-of-int* [*simp*]: *poly* (*of-int n*) *x* = *of-int n*
⟨*proof*⟩

lemma *poly-numeral* [*simp*]: *poly* (*numeral n*) *x* = *numeral n*
⟨*proof*⟩

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

lemma *numeral-monom*:
numeral n = *monom* (*numeral n*) *0*
⟨*proof*⟩

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

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

lemma *coeff-linear-poly-power*:
fixes *c* :: *'a* :: *semiring-1*
assumes $i \leq n$
shows *coeff* ([:*a*, *b*:] ^ *n*) *i* = *of-nat* (*n choose i*) ∗ *b* ^ *i* ∗ *a* ^ (*n* − *i*)
⟨*proof*⟩

4.17 Lemmas about divisibility

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

lemma *dvd-smult-cancel*: *p dvd smult a q* ⟹ *a* ≠ *0* ⟹ *p dvd q*
for *a* :: *'a*::*field*
⟨*proof*⟩

lemma *dvd-smult-iff*: *a* ≠ *0* ⟹ *p dvd smult a q* ⟷ *p dvd q*
for *a* :: *'a*::*field*
⟨*proof*⟩

lemma *smult-dvd-cancel*:
assumes *smult a p dvd q*
shows *p dvd q*
⟨*proof*⟩

lemma *smult-dvd*: *p dvd q* ⟹ *a* ≠ *0* ⟹ *smult a p dvd q*
for *a* :: *'a*::*field*
⟨*proof*⟩

lemma *smult-dvd-iff*: *smult a p dvd q* ⟷ (*if a* = *0 then q* = *0 else p dvd q*)
for *a* :: *'a*::*field*
⟨*proof*⟩

lemma *is-unit-smult-iff*: *smult c p dvd 1* ⟷ *c dvd 1* ∧ *p dvd 1*
⟨*proof*⟩

4.18 Polynomials form an integral domain

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

instance *poly* :: ({*ring-char-0*, *comm-ring-1*}) *ring-char-0*

⟨*proof*⟩

lemma *semiring-char-poly* [*simp*]: $CHAR('a :: comm\text{-}semiring\text{-}1\ poly) = CHAR('a)$
⟨*proof*⟩

instance *poly* :: ({*semiring-prime-char*,*comm-semiring-1*}) *semiring-prime-char*
⟨*proof*⟩
instance *poly* :: ({*comm-semiring-prime-char*,*comm-semiring-1*}) *comm-semiring-prime-char*
⟨*proof*⟩
instance *poly* :: ({*comm-ring-prime-char*,*comm-semiring-1*}) *comm-ring-prime-char*
⟨*proof*⟩
instance *poly* :: ({*idom-prime-char*,*comm-semiring-1*}) *idom-prime-char*
⟨*proof*⟩

lemma *degree-mult-eq*: $p \neq 0 \Longrightarrow q \neq 0 \Longrightarrow degree\ (p * q) = degree\ p + degree\ q$
for *p q* :: *'a*::{*comm-semiring-0*,*semiring-no-zero-divisors*} *poly*
⟨*proof*⟩

lemma *degree-prod-sum-eq*:
$(\bigwedge x.\ x \in A \Longrightarrow f\ x \neq 0) \Longrightarrow$
$degree\ (prod\ f\ A :: 'a :: idom\ poly) = (\sum x{\in}A.\ degree\ (f\ x))$
⟨*proof*⟩

lemma *dvd-imp-degree*:
‹$degree\ x \leq degree\ y$› **if** ‹*x dvd y*› ‹$x \neq 0$› ‹$y \neq 0$›
for *x y* :: ‹*'a*::{*comm-semiring-1*,*semiring-no-zero-divisors*} *poly*›
⟨*proof*⟩

lemma *degree-prod-eq-sum-degree*:
fixes *A* :: *'a set*
and *f* :: $'a \Rightarrow 'b$::*idom poly*
assumes *f0*: $\forall i{\in}A.\ f\ i \neq 0$
shows $degree\ (\prod i{\in}A.\ (f\ i)) = (\sum i{\in}A.\ degree\ (f\ i))$
⟨*proof*⟩

lemma *degree-mult-eq-0*:
$degree\ (p * q) = 0 \longleftrightarrow p = 0 \lor q = 0 \lor (p \neq 0 \land q \neq 0 \land degree\ p = 0 \land degree\ q = 0)$
for *p q* :: *'a*::{*comm-semiring-0*,*semiring-no-zero-divisors*} *poly*
⟨*proof*⟩

lemma *degree-power-eq*: $p \neq 0 \Longrightarrow degree\ ((p :: 'a :: idom\ poly)\ \widehat{}\ n) = n * degree\ p$
⟨*proof*⟩

lemma *degree-mult-right-le*:
fixes *p q* :: *'a*::{*comm-semiring-0*,*semiring-no-zero-divisors*} *poly*
assumes $q \neq 0$
shows $degree\ p \leq degree\ (p * q)$

⟨*proof*⟩

lemma *coeff-degree-mult*: *coeff* (*p* ∗ *q*) (*degree* (*p* ∗ *q*)) = *coeff q* (*degree q*) ∗ *coeff p* (*degree p*)
for *p q* :: *'a*::{*comm-semiring-0*,*semiring-no-zero-divisors*} *poly*
⟨*proof*⟩

lemma *dvd-imp-degree-le*: *p dvd q* ⟹ *q* ≠ *0* ⟹ *degree p* ≤ *degree q*
for *p q* :: *'a*::{*comm-semiring-1*,*semiring-no-zero-divisors*} *poly*
⟨*proof*⟩

lemma *divides-degree*:
fixes *p q* :: *'a* ::{*comm-semiring-1*,*semiring-no-zero-divisors*} *poly*
assumes *p dvd q*
shows *degree p* ≤ *degree q* ∨ *q* = *0*
⟨*proof*⟩

lemma *const-poly-dvd-iff*:
fixes *c* :: *'a*::{*comm-semiring-1*,*semiring-no-zero-divisors*}
shows [:*c*:] *dvd p* ⟷ (∀ *n*. *c dvd coeff p n*)
⟨*proof*⟩

lemma *const-poly-dvd-const-poly-iff* [*simp*]: [:*a*:] *dvd* [:*b*:] ⟷ *a dvd b*
for *a b* :: *'a*::{*comm-semiring-1*,*semiring-no-zero-divisors*}
⟨*proof*⟩

lemma *lead-coeff-mult*: *lead-coeff* (*p* ∗ *q*) = *lead-coeff p* ∗ *lead-coeff q*
for *p q* :: *'a*::{*comm-semiring-0*, *semiring-no-zero-divisors*} *poly*
⟨*proof*⟩

lemma *lead-coeff-prod*: *lead-coeff* (*prod f A*) = (∏ *x*∈*A*. *lead-coeff* (*f x*))
for *f* :: *'a* ⇒ *'b*::{*comm-semiring-1*, *semiring-no-zero-divisors*} *poly*
⟨*proof*⟩

lemma *lead-coeff-smult*: *lead-coeff* (*smult c p*) = *c* ∗ *lead-coeff p*
for *p* :: *'a*::{*comm-semiring-0*,*semiring-no-zero-divisors*} *poly*
⟨*proof*⟩

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

lemma *lead-coeff-power*: *lead-coeff* (*p* ^ *n*) = *lead-coeff p* ^ *n*
for *p* :: *'a*::{*comm-semiring-1*,*semiring-no-zero-divisors*} *poly*
⟨*proof*⟩

4.19 Polynomials form an ordered integral domain

definition *pos-poly* :: *'a*::*linordered-semidom poly* ⇒ *bool*
where *pos-poly p* ⟷ *0* < *coeff p* (*degree p*)

lemma *pos-poly-pCons*: *pos-poly* (*pCons a p*) ⟷ *pos-poly p* ∨ (*p* = *0* ∧ *0* < *a*)
⟨*proof*⟩

lemma *not-pos-poly-0* [*simp*]: ¬ *pos-poly 0*
⟨*proof*⟩

lemma *pos-poly-add*: *pos-poly p* ⟹ *pos-poly q* ⟹ *pos-poly* (*p* + *q*)
⟨*proof*⟩

lemma *pos-poly-mult*: *pos-poly p* ⟹ *pos-poly q* ⟹ *pos-poly* (*p* ∗ *q*)
⟨*proof*⟩

lemma *pos-poly-total*: *p* = *0* ∨ *pos-poly p* ∨ *pos-poly* (− *p*)
for *p* :: *'a*::*linordered-idom poly*
⟨*proof*⟩

lemma *pos-poly-coeffs* [*code*]: *pos-poly p* ⟷ (*let as* = *coeffs p in as* ≠ [] ∧ *last as* > *0*)
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

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

definition *x* < *y* ⟷ *pos-poly* (*y* − *x*)

definition *x* ≤ *y* ⟷ *x* = *y* ∨ *pos-poly* (*y* − *x*)

definition |*x*::*'a poly*| = (*if x* < *0 then* − *x else x*)

definition *sgn* (*x*::*'a poly*) = (*if x* = *0 then 0 else if 0* < *x then 1 else* − *1*)

instance
⟨*proof*⟩

end

TODO: Simplification rules for comparisons

4.20 Synthetic division and polynomial roots

4.20.1 Synthetic division

Synthetic division is simply division by the linear polynomial $x - c$.

definition *synthetic-divmod* :: *'a*::*comm-semiring-0 poly* ⇒ *'a* ⇒ *'a poly* × *'a*
where *synthetic-divmod p c* = *fold-coeffs* (λ*a* (*q*, *r*). (*pCons r q*, *a* + *c* ∗ *r*)) *p* (*0*, *0*)

definition *synthetic-div* :: *'a::comm-semiring-0 poly* ⇒ *'a* ⇒ *'a poly*
  where *synthetic-div p c* = *fst* (*synthetic-divmod p c*)

lemma *synthetic-divmod-0* [*simp*]: *synthetic-divmod 0 c* = (*0*, *0*)
  ⟨*proof*⟩

lemma *synthetic-divmod-pCons* [*simp*]:
  synthetic-divmod (*pCons a p*) *c* = (λ(*q*, *r*). (*pCons r q*, *a* + *c* ∗ *r*)) (*synthetic-divmod p c*)
  ⟨*proof*⟩

lemma *synthetic-div-0* [*simp*]: *synthetic-div 0 c* = *0*
  ⟨*proof*⟩

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

lemma *snd-synthetic-divmod*: *snd* (*synthetic-divmod p c*) = *poly p c*
  ⟨*proof*⟩

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

lemma *synthetic-div-eq-0-iff*: *synthetic-div p c* = *0* ⟷ *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'*: [:−*c*, *1*:] ∗ *synthetic-div p c* + [:*poly p c*:] = *p*
  for *c* :: *'a::comm-ring-1*
  ⟨*proof*⟩

4.20.2 Polynomial roots

lemma *poly-eq-0-iff-dvd*: *poly p c* = *0* ⟷ [:− *c*, *1*:] *dvd p*
  (**is** *?lhs* ⟷ *?rhs*)
  for *c* :: *'a::comm-ring-1*
⟨*proof*⟩

lemma *dvd-iff-poly-eq-0*: $[:c,\ 1:]$ *dvd p* $\longleftrightarrow$ *poly p* $(-\ c) = 0$
for c :: *'a::comm-ring-1*
⟨*proof*⟩

lemma *poly-roots-finite*: $p \neq 0 \Longrightarrow$ *finite* $\{x.$ *poly p x* $= 0\}$
for p :: *'a::{comm-ring-1,ring-no-zero-divisors} poly*
⟨*proof*⟩

lemma *poly-eq-poly-eq-iff*: *poly p* = *poly q* $\longleftrightarrow p = q$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
for *p q* :: *'a::{comm-ring-1,ring-no-zero-divisors,ring-char-0} poly*
⟨*proof*⟩

lemma *poly-all-0-iff-0*: $(\forall x.$ *poly p x* $= 0) \longleftrightarrow p = 0$
for p :: *'a::{ring-char-0,comm-ring-1,ring-no-zero-divisors} poly*
⟨*proof*⟩

lemma *card-poly-roots-bound*:
fixes p :: *'a::{comm-ring-1,ring-no-zero-divisors} poly*
assumes $p \neq 0$
shows *card* $\{x.$ *poly p x* $= 0\} \leq$ *degree p*
⟨*proof*⟩

lemma *poly-eqI-degree*:
fixes *p q* :: *'a :: {comm-ring-1, ring-no-zero-divisors} poly*
assumes $\bigwedge x.\ x \in A \Longrightarrow$ *poly p x* = *poly q x*
assumes *card A* > *degree p* *card A* > *degree q*
shows $p = q$
⟨*proof*⟩

4.20.3 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*: *coeff* $([:a,\ 1:]$ ^ $n)$ $n = 1$
for a :: *'a::comm-semiring-1*
⟨*proof*⟩

lemma *degree-linear-power*: *degree* $([:a,\ 1:]$ ^ $n) = n$
for a :: *'a::comm-semiring-1*
⟨*proof*⟩

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

lemma *order-2*:
assumes $p \neq 0$
shows ¬ $[:-a,\ 1:]$ ^ *Suc* (*order a p*) *dvd p*

⟨*proof*⟩

lemma *order*: $p \neq 0 \Longrightarrow [:-a,\ 1:]\ \widehat{}\ order\ a\ p\ dvd\ p \wedge \neg\ [:-a,\ 1:]\ \widehat{}\ Suc\ (order\ a\ p)\ dvd\ p$
⟨*proof*⟩

lemma *order-degree*:
assumes *p*: $p \neq 0$
shows $order\ a\ p \leq degree\ p$
⟨*proof*⟩

lemma *order-root*: $poly\ p\ a = 0 \longleftrightarrow p = 0 \vee order\ a\ p \neq 0$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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

lemma *order-unique-lemma*:
fixes *p* :: *'a*::*idom poly*
assumes $[:-a,\ 1:]\ \widehat{}\ n\ dvd\ p\ \neg\ [:-a,\ 1:]\ \widehat{}\ Suc\ n\ dvd\ p$
shows $order\ a\ p = n$
⟨*proof*⟩

lemma *order-mult*:
assumes $p * q \neq 0$ **shows** $order\ a\ (p * q) = order\ a\ p + order\ a\ q$
⟨*proof*⟩

lemma *order-smult*:
assumes $c \neq 0$
shows $order\ x\ (smult\ c\ p) = order\ x\ p$
⟨*proof*⟩

lemma *order-gt-0-iff*: $p \neq 0 \Longrightarrow order\ x\ p > 0 \longleftrightarrow poly\ p\ x = 0$
⟨*proof*⟩

lemma *order-eq-0-iff*: $p \neq 0 \Longrightarrow order\ x\ p = 0 \longleftrightarrow poly\ p\ x \neq 0$
⟨*proof*⟩

Next three lemmas contributed by Wenda Li

lemma *order-1-eq-0* [*simp*]:$order\ x\ 1 = 0$
⟨*proof*⟩

lemma *order-uminus*[*simp*]: $order\ x\ (-p) = order\ x\ p$
⟨*proof*⟩

lemma *order-power-n-n*: $order\ a\ ([:-a,1:]\widehat{}n)=n$
⟨*proof*⟩

lemma *order-0-monom* [*simp*]: $c \neq 0 \Longrightarrow order\ 0\ (monom\ c\ n) = n$
⟨*proof*⟩

lemma *dvd-imp-order-le*: $q \neq 0 \Longrightarrow p\ dvd\ q \Longrightarrow Polynomial.order\ a\ p \leq Polynomial.order\ a\ q$
⟨*proof*⟩

Now justify the standard squarefree decomposition, i.e. $f\ /\ gcd\ f\ f'$.

lemma *order-divides*: $[:-a,\ 1:]\ \widehat{}\ n\ dvd\ p \longleftrightarrow p = 0 \vee n \leq order\ a\ p$
⟨*proof*⟩

lemma *order-decomp*:
assumes $p \neq 0$
shows $\exists\, q.\ p = [:-\ a,\ 1:]\ \widehat{}\ order\ a\ p * q \wedge \neg\ [:-\ a,\ 1:]\ dvd\ q$
⟨*proof*⟩

lemma *monom-1-dvd-iff*: $p \neq 0 \Longrightarrow monom\ 1\ n\ dvd\ p \longleftrightarrow n \leq order\ 0\ p$
⟨*proof*⟩

lemma *poly-root-order-induct* [*case-names 0 no-roots root*]:
fixes $p :: 'a :: idom\ poly$
assumes $P\ 0\ \bigwedge p.\ (\bigwedge x.\ poly\ p\ x \neq 0) \Longrightarrow P\ p$
$\bigwedge p\ x\ n.\ n > 0 \Longrightarrow poly\ p\ x \neq 0 \Longrightarrow P\ p \Longrightarrow P\ ([:-x,\ 1:]\ \widehat{}\ n * p)$
shows $P\ p$
⟨*proof*⟩

context
includes *multiset.lifting*
begin

lift-definition *proots* :: $('a :: idom)\ poly \Rightarrow 'a\ multiset$ **is**
$\lambda(p :: 'a\ poly)\ (x :: 'a).\ if\ p = 0\ then\ 0\ else\ order\ x\ p$
⟨*proof*⟩

lemma *proots-0* [*simp*]: $proots\ (0 :: 'a :: idom\ poly) = \{\#\}$
⟨*proof*⟩

lemma *proots-1* [*simp*]: $proots\ (1 :: 'a :: idom\ poly) = \{\#\}$
⟨*proof*⟩

lemma *proots-const* [*simp*]: $proots\ [:\ x\ :] = 0$
⟨*proof*⟩

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

lemma *count-proots* [*simp*]:
$p \neq 0 \Longrightarrow count\ (proots\ p)\ a = order\ a\ p$

⟨*proof*⟩

lemma *set-count-proots* [*simp*]:
$p \neq 0 \Longrightarrow$ *set-mset* (*proots p*) = {*x*. *poly p x* = *0*}
⟨*proof*⟩

lemma *proots-uminus* [*simp*]: *proots* (−*p*) = *proots p*
⟨*proof*⟩

lemma *proots-smult* [*simp*]: $c \neq 0 \Longrightarrow$ *proots* (*smult c p*) = *proots p*
⟨*proof*⟩

lemma *proots-mult*:
assumes $p \neq 0$ $q \neq 0$
shows *proots* ($p * q$) = *proots p* + *proots q*
⟨*proof*⟩

lemma *proots-prod*:
assumes $\bigwedge x.\ x \in A \Longrightarrow f\ x \neq 0$
shows *proots* ($\prod x{\in}A.\ f\ x$) = ($\sum x{\in}A.$ *proots* ($f\ x$))
⟨*proof*⟩

lemma *proots-prod-mset*:
assumes $0 \notin\# A$
shows *proots* ($\prod p{\in}\#A.\ p$) = ($\sum p{\in}\#A.$ *proots p*)
⟨*proof*⟩

lemma *proots-prod-list*:
assumes $0 \notin$ *set ps*
shows *proots* ($\prod p{\leftarrow}ps.\ p$) = ($\sum p{\leftarrow}ps.$ *proots p*)
⟨*proof*⟩

lemma *proots-power*: *proots* (p ^ n) = *repeat-mset n* (*proots p*)
⟨*proof*⟩

lemma *proots-linear-factor* [*simp*]: *proots* [:*x*, *1*:] = {#−*x*#}
⟨*proof*⟩

lemma *size-proots-le*: *size* (*proots p*) $\leq$ *degree p*
⟨*proof*⟩

end

4.21 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 $Q\ 0$
and $\bigwedge p.\ (\bigwedge a.\ P\ a \Longrightarrow poly\ p\ a \neq 0) \Longrightarrow Q\ p$
and $\bigwedge a\ p.\ P\ a \Longrightarrow Q\ p \Longrightarrow Q\ ([:a,\ -1:] * p)$
shows $Q\ p$
⟨*proof*⟩

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

4.22 Composition of polynomials

definition *pcompose* :: *'a::comm-semiring-0 poly* ⇒ *'a poly* ⇒ *'a poly*
where *pcompose p q* = *fold-coeffs* ($\lambda a\ c.$ [:*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-altdef*: *pcompose p q* = *poly* (*map-poly* ($\lambda x.$ [:*x*:]) *p*) *q*
⟨*proof*⟩

lemma *coeff-pcompose-0* [*simp*]:
coeff (*pcompose p q*) *0* = *poly p* (*coeff q 0*)
⟨*proof*⟩

lemma *pcompose-1*: *pcompose 1 p* = *1*
for *p* :: *'a::comm-semiring-1 poly*
⟨*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*: *pcompose* $(p + q)$ r = *pcompose* p r + *pcompose* q r
for p q r :: *'a*::{*comm-semiring-0*, *ab-semigroup-add*} *poly*
⟨*proof*⟩

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

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

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

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

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

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

lemma *pcompose-sum*: *pcompose* (*sum* f A) p = *sum* (λi. *pcompose* (f i) p) A
⟨*proof*⟩

lemma *pcompose-prod*: *pcompose* (*prod* f A) p = *prod* (λi. *pcompose* (f i) p) A
⟨*proof*⟩

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

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

lemma *degree-pcompose*: *degree* (*pcompose* p q) = *degree* p ∗ *degree* q
for p q :: *'a*::{*comm-semiring-0*,*semiring-no-zero-divisors*} *poly*
⟨*proof*⟩

lemma *pcompose-eq-0*:
fixes p q :: *'a*::{*comm-semiring-0*,*semiring-no-zero-divisors*} *poly*
assumes *pcompose* p q = *0* *degree* q > *0*
shows p = *0*

⟨*proof*⟩

lemma *pcompose-eq-0-iff*:
fixes $p\ q$ :: *'a*::{*comm-semiring-0*,*semiring-no-zero-divisors*} *poly*
assumes *degree* $q > 0$
shows *pcompose* $p\ q = 0 \longleftrightarrow p = 0$
⟨*proof*⟩

lemma *coeff-pcompose-linear*:
coeff (*pcompose* p [:*0*, a :: *'a* :: *comm-semiring-1*:]) $i = a$ ^ i ∗ *coeff* $p\ i$
⟨*proof*⟩

lemma *lead-coeff-comp*:
fixes $p\ q$ :: *'a*::{*comm-semiring-1*,*semiring-no-zero-divisors*} *poly*
assumes *degree* $q > 0$
shows *lead-coeff* (*pcompose* $p\ q$) = *lead-coeff* p ∗ *lead-coeff* q ^ (*degree* p)
⟨*proof*⟩

lemma *coeff-pcompose-monom-linear* [*simp*]:
fixes p :: *'a* :: *comm-ring-1 poly*
shows *coeff* (*pcompose* p (*monom* c (*Suc 0*))) $k = c$ ^ k ∗ *coeff* $p\ k$
⟨*proof*⟩

lemma *of-nat-mult-conv-smult*: *of-nat* $n * P$ = *smult* (*of-nat* n) P
⟨*proof*⟩

lemma *numeral-mult-conv-smult*: *numeral* $n * P$ = *smult* (*numeral* n) P
⟨*proof*⟩

lemma *sum-order-le-degree*:
assumes $p \neq 0$
shows ($\sum x$ | *poly* $p\ x = 0$. *order* $x\ p$) ≤ *degree* p
⟨*proof*⟩

4.23 Closure properties of coefficients

context
fixes R :: *'a* :: *comm-semiring-1 set*
assumes *R-0*: $0 \in R$
assumes *R-plus*: $\bigwedge x\ y.\ x \in R \Longrightarrow y \in R \Longrightarrow x + y \in R$
assumes *R-mult*: $\bigwedge x\ y.\ x \in R \Longrightarrow y \in R \Longrightarrow x * y \in R$
begin

lemma *coeff-mult-semiring-closed*:
assumes $\bigwedge i$. *coeff* $p\ i \in R$ $\bigwedge i$. *coeff* $q\ i \in R$
shows *coeff* $(p * q)\ i \in R$
⟨*proof*⟩

lemma *coeff-pcompose-semiring-closed*:

assumes $\bigwedge i.$ *coeff p i* $\in R$ $\bigwedge i.$ *coeff q i* $\in R$
shows *coeff (pcompose p q) i* $\in R$
⟨*proof*⟩

end

4.24 Shifting polynomials

definition *poly-shift* :: *nat* ⇒ *'a::zero poly* ⇒ *'a poly*
where *poly-shift n p* = *Abs-poly* (λ*i. coeff p* (*i* + *n*))

lemma *nth-default-drop*: *nth-default x* (*drop n xs*) *m* = *nth-default x xs* (*m* + *n*)
⟨*proof*⟩

lemma *nth-default-take*: *nth-default x* (*take n xs*) *m* = (*if m* < *n then nth-default x xs m else x*)
⟨*proof*⟩

lemma *coeff-poly-shift*: *coeff* (*poly-shift n p*) *i* = *coeff p* (*i* + *n*)
⟨*proof*⟩

lemma *poly-shift-id* [*simp*]: *poly-shift 0* = (λ*x. x*)
⟨*proof*⟩

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

lemma *poly-shift-1*: *poly-shift n 1* = (*if n* = *0 then 1 else 0*)
⟨*proof*⟩

lemma *poly-shift-monom*: *poly-shift n* (*monom c m*) = (*if m* ≥ *n then monom c* (*m* − *n*) *else 0*)
⟨*proof*⟩

lemma *coeffs-shift-poly* [*code abstract*]:
coeffs (*poly-shift n p*) = *drop n* (*coeffs p*)
⟨*proof*⟩

4.25 Truncating polynomials

definition *poly-cutoff*
where *poly-cutoff n p* = *Abs-poly* (λ*k. if k* < *n then coeff p k else 0*)

lemma *coeff-poly-cutoff*: *coeff* (*poly-cutoff n p*) *k* = (*if k* < *n then coeff p k else 0*)
⟨*proof*⟩

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

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

lemma *coeffs-poly-cutoff* [*code abstract*]:
coeffs (*poly-cutoff n p*) = *strip-while* ((=) *0*) (*take n* (*coeffs p*))
⟨*proof*⟩

4.26 Reflecting polynomials

definition *reflect-poly* :: *'a::zero poly* ⇒ *'a poly*
where *reflect-poly p* = *Poly* (*rev* (*coeffs p*))

lemma *coeffs-reflect-poly* [*code abstract*]:
coeffs (*reflect-poly p*) = *rev* (*dropWhile* ((=) *0*) (*coeffs p*))
⟨*proof*⟩

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

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

lemma *coeff-reflect-poly*:
coeff (*reflect-poly p*) *n* = (*if n* > *degree p then 0 else coeff p* (*degree p* − *n*))
⟨*proof*⟩

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

lemma *reflect-poly-at-0-eq-0-iff* [*simp*]: *poly* (*reflect-poly p*) *0* = *0* ⟷ *p* = *0*
⟨*proof*⟩

lemma *reflect-poly-pCons'*:
p ≠ *0* ⟹ *reflect-poly* (*pCons c p*) = *reflect-poly p* + *monom c* (*Suc* (*degree p*))
⟨*proof*⟩

lemma *reflect-poly-const* [*simp*]: *reflect-poly* [:*a*:] = [:*a*:]
⟨*proof*⟩

lemma *poly-reflect-poly-nz*:
x ≠ *0* ⟹ *poly* (*reflect-poly p*) *x* = *x* ^ *degree p* ∗ *poly p* (*inverse x*)
for *x* :: *'a::field*
⟨*proof*⟩

lemma *coeff-0-reflect-poly* [*simp*]: *coeff* (*reflect-poly p*) *0* = *lead-coeff p*
⟨*proof*⟩

lemma *poly-reflect-poly-0* [*simp*]: *poly* (*reflect-poly p*) *0* = *lead-coeff p*
⟨*proof*⟩

lemma *reflect-poly-reflect-poly* [*simp*]: *coeff p 0* ≠ *0* ⟹ *reflect-poly* (*reflect-poly p*) = *p*
⟨*proof*⟩

lemma *degree-reflect-poly-le*: *degree* (*reflect-poly p*) ≤ *degree p*
⟨*proof*⟩

lemma *reflect-poly-pCons*: *a* ≠ *0* ⟹ *reflect-poly* (*pCons a p*) = *Poly* (*rev* (*a* # *coeffs p*))
⟨*proof*⟩

lemma *degree-reflect-poly-eq* [*simp*]: *coeff p 0* ≠ *0* ⟹ *degree* (*reflect-poly p*) = *degree p*
⟨*proof*⟩

lemma *reflect-poly-eq-0-iff* [*simp*]: *reflect-poly p* = *0* ⟷ *p* = *0*
⟨*proof*⟩

lemma *reflect-poly-mult*: *reflect-poly* (*p* ∗ *q*) = *reflect-poly p* ∗ *reflect-poly q*
for *p q* :: ′*a*::{*comm-semiring-0*,*semiring-no-zero-divisors*} *poly*
⟨*proof*⟩

lemma *reflect-poly-smult*: *reflect-poly* (*smult c p*) = *smult c* (*reflect-poly p*)
for *p* :: ′*a*::{*comm-semiring-0*,*semiring-no-zero-divisors*} *poly*
⟨*proof*⟩

lemma *reflect-poly-power*: *reflect-poly* (*p* ^ *n*) = *reflect-poly p* ^ *n*
for *p* :: ′*a*::{*comm-semiring-1*,*semiring-no-zero-divisors*} *poly*
⟨*proof*⟩

lemma *reflect-poly-prod*: *reflect-poly* (*prod f A*) = *prod* (λ*x*. *reflect-poly* (*f x*)) *A*
for *f* :: - ⇒ -::{*comm-semiring-0*,*semiring-no-zero-divisors*} *poly*
⟨*proof*⟩

lemma *reflect-poly-prod-list*: *reflect-poly* (*prod-list xs*) = *prod-list* (*map reflect-poly xs*)
for *xs* :: -::{*comm-semiring-0*,*semiring-no-zero-divisors*} *poly list*
⟨*proof*⟩

lemma *reflect-poly-Poly-nz*:
no-trailing (*HOL.eq 0*) *xs* ⟹ *reflect-poly* (*Poly xs*) = *Poly* (*rev xs*)
⟨*proof*⟩

lemmas *reflect-poly-simps* =
reflect-poly-0 reflect-poly-1 reflect-poly-const reflect-poly-smult reflect-poly-mult reflect-poly-power reflect-poly-prod reflect-poly-prod-list

4.27 Derivatives

function *pderiv* :: (*'a* :: {*comm-semiring-1*,*semiring-no-zero-divisors*}) *poly* ⇒ *'a poly*
  where *pderiv* (*pCons a p*) = (*if p* = *0 then 0 else p* + *pCons 0* (*pderiv p*))
  ⟨*proof*⟩

termination *pderiv*
  ⟨*proof*⟩

declare *pderiv.simps*[*simp del*]

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*::{*comm-semiring-1*,*semiring-no-zero-divisors*} ⇒ *'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*::{*comm-semiring-1*,*semiring-no-zero-divisors*} *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 *coeffs-pderiv-code* [*code abstract*]: *coeffs* (*pderiv p*) = *pderiv-coeffs* (*coeffs p*)
  ⟨*proof*⟩

lemma *pderiv-eq-0-iff*: *pderiv p* = *0* ⟷ *degree p* = *0*
  for *p* :: *'a*::{*comm-semiring-1*,*semiring-no-zero-divisors*,*semiring-char-0*} *poly*
⟨*proof*⟩

lemma *degree-pderiv*: *degree* (*pderiv p*) = *degree p* − *1*
for *p* :: *'a*::{*comm-semiring-1*,*semiring-no-zero-divisors*,*semiring-char-0*} *poly*
⟨*proof*⟩

lemma *not-dvd-pderiv*:
fixes *p* :: *'a*::{*comm-semiring-1*,*semiring-no-zero-divisors*,*semiring-char-0*} *poly*
assumes *degree p* ≠ *0*
shows ¬ *p dvd pderiv p*
⟨*proof*⟩

lemma *dvd-pderiv-iff* [*simp*]: *p dvd pderiv p* ⟷ *degree p* = *0*
for *p* :: *'a*::{*comm-semiring-1*,*semiring-no-zero-divisors*,*semiring-char-0*} *poly*
⟨*proof*⟩

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* :: - :: *idom poly*) − *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* ^ *Suc n*) = *smult* (*of-nat* (*Suc n*)) (*p* ^ *n*) ∗ *pderiv p*
⟨*proof*⟩

lemma *pderiv-power*:
pderiv (*p* ^ *n*) = *smult* (*of-nat n*) (*p* ^ (*n* − *1*) ∗ *pderiv p*)
⟨*proof*⟩

lemma *pderiv-monom*:
pderiv (*monom c n*) = *monom* (*of-nat n* ∗ *c*) (*n* − *1*)
⟨*proof*⟩

lemma *pderiv-pcompose*: *pderiv* (*pcompose p q*) = *pcompose* (*pderiv p*) *q* ∗ *pderiv q*
⟨*proof*⟩

lemma *pderiv-prod*: *pderiv* (*prod f* (*as*)) = ($\sum$ *a*∈*as*. *prod f* (*as* − {*a*}) ∗ *pderiv* (*f a*))
⟨*proof*⟩

lemma *coeff-higher-pderiv*:
coeff ((*pderiv* ^^ *m*) *f*) *n* = *pochhammer* (*of-nat* (*Suc n*)) *m* ∗ *coeff f* (*n* + *m*)
⟨*proof*⟩

lemma *higher-pderiv-0* [*simp*]: (*pderiv* ^^ *n*) *0* = *0*
⟨*proof*⟩

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

lemma *higher-pderiv-smult*: (*pderiv* ^^ *n*) (*smult c p*) = *smult c* ((*pderiv* ^^ *n*) *p*)
⟨*proof*⟩

lemma *higher-pderiv-monom*:
m ≤ *n* + *1* ⟹ (*pderiv* ^^ *m*) (*monom c n*) = *monom* (*pochhammer* (*int n* − *int m* + *1*) *m* ∗ *c*) (*n* − *m*)
⟨*proof*⟩

lemma *higher-pderiv-monom-eq-zero*:
m > *n* + *1* ⟹ (*pderiv* ^^ *m*) (*monom c n*) = *0*
⟨*proof*⟩

lemma *higher-pderiv-sum*: (*pderiv* ^^ *n*) (*sum f A*) = ($\sum$ *x*∈*A*. (*pderiv* ^^ *n*) (*f x*))
⟨*proof*⟩

lemma *higher-pderiv-sum-mset*: (*pderiv* ^^ *n*) (*sum-mset A*) = ($\sum$ *p*∈#*A*. (*pderiv* ^^ *n*) *p*)
⟨*proof*⟩

lemma *higher-pderiv-sum-list*: (*pderiv* ^^ *n*) (*sum-list ps*) = ($\sum$ *p*←*ps*. (*pderiv* ^^ *n*) *p*)
⟨*proof*⟩

lemma *degree-higher-pderiv*: *Polynomial.degree* ((*pderiv* ^^ *n*) *p*) = *Polynomial.degree p* − *n*
for *p* :: *'a*::{*comm-semiring-1*,*semiring-no-zero-divisors*,*semiring-char-0*} *poly*
⟨*proof*⟩

lemma *DERIV-pow2*: *DERIV* (λ*x*. *x* ^ *Suc n*) *x* :> *real* (*Suc n*) ∗ (*x* ^ *n*)
⟨*proof*⟩
declare *DERIV-pow2* [*simp*] *DERIV-pow* [*simp*]

lemma *DERIV-add-const*: *DERIV f x* :> *D* $\Longrightarrow$ *DERIV* ($\lambda x.$ *a* + *f x* :: *'a::real-normed-field*) *x* :> *D*
⟨*proof*⟩

lemma *poly-DERIV* [*simp*]: *DERIV* ($\lambda x.$ *poly p x*) *x* :> *poly* (*pderiv p*) *x*
⟨*proof*⟩

lemma *poly-isCont*[*simp*]:
fixes *x::'a::real-normed-field*
shows *isCont* ($\lambda x.$ *poly p x*) *x*
⟨*proof*⟩

lemma *tendsto-poly* [*tendsto-intros*]: (*f* $\longrightarrow$ *a*) *F* $\Longrightarrow$ (($\lambda x.$ *poly p* (*f x*)) $\longrightarrow$ *poly p a*) *F*
for *f* :: - ⇒ *'a::real-normed-field*
⟨*proof*⟩

lemma *continuous-within-poly*: *continuous* (*at z within s*) (*poly p*)
for *z* :: *'a::{real-normed-field}*
⟨*proof*⟩

lemma *continuous-poly* [*continuous-intros*]: *continuous F f* $\Longrightarrow$ *continuous F* ($\lambda x.$ *poly p* (*f x*))
for *f* :: - ⇒ *'a::real-normed-field*
⟨*proof*⟩

lemma *continuous-on-poly* [*continuous-intros*]:
fixes *p* :: *'a* :: {*real-normed-field*} *poly*
assumes *continuous-on A f*
shows *continuous-on A* ($\lambda x.$ *poly p* (*f x*))
⟨*proof*⟩

Consequences of the derivative theorem above.

lemma *poly-differentiable*[*simp*]: ($\lambda x.$ *poly p x*) *differentiable* (*at x*)
for *x* :: *real*
⟨*proof*⟩

lemma *poly-IVT-pos*: *a* < *b* $\Longrightarrow$ *poly p a* < *0* $\Longrightarrow$ *0* < *poly p b* $\Longrightarrow$ $\exists$ *x*. *a* < *x* $\wedge$ *x* < *b* $\wedge$ *poly p x* = *0*
for *a b* :: *real*
⟨*proof*⟩

lemma *poly-IVT-neg*: *a* < *b* $\Longrightarrow$ *0* < *poly p a* $\Longrightarrow$ *poly p b* < *0* $\Longrightarrow$ $\exists$ *x*. *a* < *x* $\wedge$ *x* < *b* $\wedge$ *poly p x* = *0*
for *a b* :: *real*
⟨*proof*⟩

lemma *poly-IVT*: *a* < *b* $\Longrightarrow$ *poly p a* * *poly p b* < *0* $\Longrightarrow$ $\exists$ *x*>*a*. *x* < *b* $\wedge$ *poly p x* = *0*

for *p* :: *real poly*
⟨*proof*⟩

lemma *poly-MVT*: $a < b \Longrightarrow \exists x.\ a < x \wedge x < b \wedge poly\ p\ b - poly\ p\ a = (b - a) * poly\ (pderiv\ p)\ x$
for *a b* :: *real*
⟨*proof*⟩

lemma *poly-MVT′*:
fixes *a b* :: *real*
assumes $\{min\ a\ b..max\ a\ b\} \subseteq A$
shows $\exists x \in A.\ poly\ p\ b - poly\ p\ a = (b - a) * poly\ (pderiv\ p)\ x$
⟨*proof*⟩

lemma *poly-pinfty-gt-lc*:
fixes *p* :: *real poly*
assumes *lead-coeff p* > *0*
shows $\exists n.\ \forall\ x \geq n.\ poly\ p\ x \geq lead\text{-}coeff\ p$
⟨*proof*⟩

lemma *lemma-order-pderiv1*:
pderiv ([:− *a*, *1*:] ^ *Suc n* ∗ *q*) = [:− *a*, *1*:] ^ *Suc n* ∗ *pderiv q* +
smult (*of-nat* (*Suc n*)) (*q* ∗ [:− *a*, *1*:] ^ *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*:] ^ *n* ∗ *q*
and *nd*: ¬ [:− *a*, *1*:] *dvd q*
shows *n* = *Suc* (*order a* (*pderiv p*))
⟨*proof*⟩

lemma *order-pderiv*: *order a p* = *Suc* (*order a* (*pderiv p*))
if *pderiv p* ≠ *0 order a p* ≠ *0*
for *p* :: *′a::field-char-0 poly*
⟨*proof*⟩

lemma *poly-squarefree-decomp-order*:
fixes *p* :: *′a::field-char-0 poly*
assumes *pderiv p* ≠ *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* ⟹ *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*)
for *p* :: *'a::field-char-0 poly*
⟨*proof*⟩

lemma *order-pderiv2*:
pderiv p ≠ *0* ⟹ *order a p* ≠ *0* ⟹ *order a* (*pderiv p*) = *n* ⟷ *order a p* = *Suc n*
for *p* :: *'a::field-char-0 poly*
⟨*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*:]
for *p* :: *'a::{semidom,semiring-char-0} poly*
⟨*proof*⟩

lemma *rsquarefree-roots*: *rsquarefree p* ⟷ (∀ *a*. ¬ (*poly p a* = *0* ∧ *poly* (*pderiv p*) *a* = *0*))
for *p* :: *'a::field-char-0 poly*
⟨*proof*⟩

lemma *rsquarefree-root-order*:
assumes *rsquarefree p poly p z* = *0 p* ≠ *0*
shows *order z p* = *1*
⟨*proof*⟩

lemma *poly-squarefree-decomp*:
fixes *p* :: *'a::field-char-0 poly*
assumes *pderiv p* ≠ *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*⟩

lemma *has-field-derivative-poly* [*derivative-intros*]:
assumes (*f has-field-derivative f'*) (*at x within A*)
shows ((λ*x*. *poly p* (*f x*)) *has-field-derivative*
(*f'* ∗ *poly* (*pderiv p*) (*f x*))) (*at x within A*)
⟨*proof*⟩

4.28 Algebraic numbers

lemma *intpolyE*:
assumes ⋀*i*. *poly.coeff p i* ∈ ℤ
obtains *q* **where** *p* = *map-poly of-int q*
⟨*proof*⟩

lemma *ratpolyE*:
  assumes $\bigwedge i.\ poly.coeff\ p\ i \in \mathbb{Q}$
  obtains q **where** $p = map\text{-}poly\ of\text{-}rat\ q$
⟨*proof*⟩

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.

definition $algebraic :: 'a :: field\text{-}char\text{-}0 \Rightarrow bool$
  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*: $(\bigwedge i.\ coeff\ p\ i \in \mathbb{Z}) \Longrightarrow p \neq 0 \Longrightarrow poly\ p\ x = 0 \Longrightarrow algebraic\ x$
  ⟨*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 *algebraic-altdef*: $algebraic\ x \longleftrightarrow (\exists p.\ (\forall i.\ coeff\ p\ i \in \mathbb{Q}) \wedge p \neq 0 \wedge poly\ p\ x = 0)$
  for $p :: 'a::field\text{-}char\text{-}0\ poly$
⟨*proof*⟩

lemma *algebraicI'*: $(\bigwedge i.\ coeff\ p\ i \in \mathbb{Q}) \Longrightarrow p \neq 0 \Longrightarrow poly\ p\ x = 0 \Longrightarrow algebraic\ x$
  ⟨*proof*⟩

lemma *algebraicE'*:
  assumes $algebraic\ (x :: 'a :: field\text{-}char\text{-}0)$
  obtains p **where** $p \neq 0\ poly\ (map\text{-}poly\ of\text{-}int\ p)\ x = 0$
⟨*proof*⟩

lemma *algebraicE'-nonzero*:
  assumes $algebraic\ (x :: 'a :: field\text{-}char\text{-}0)\ x \neq 0$
  obtains p **where** $p \neq 0\ coeff\ p\ 0 \neq 0\ poly\ (map\text{-}poly\ of\text{-}int\ p)\ x = 0$
⟨*proof*⟩

lemma *rat-imp-algebraic*: $x \in \mathbb{Q} \Longrightarrow algebraic\ x$
⟨*proof*⟩

lemma *algebraic-0* [*simp*, *intro*]: *algebraic 0*

and *algebraic-1* [*simp*, *intro*]: *algebraic 1*
and *algebraic-numeral* [*simp*, *intro*]: *algebraic* (*numeral n*)
and *algebraic-of-nat* [*simp*, *intro*]: *algebraic* (*of-nat k*)
and *algebraic-of-int* [*simp*, *intro*]: *algebraic* (*of-int m*)
⟨*proof*⟩

lemma *algebraic-ii* [*simp*, *intro*]: *algebraic* i
⟨*proof*⟩

lemma *algebraic-minus* [*intro*]:
assumes *algebraic x*
shows *algebraic* (−*x*)
⟨*proof*⟩

lemma *algebraic-minus-iff* [*simp*]:
algebraic (−*x*) ⟷ *algebraic* (*x* :: ′*a* :: *field-char-0*)
⟨*proof*⟩

lemma *algebraic-inverse* [*intro*]:
assumes *algebraic x*
shows *algebraic* (*inverse x*)
⟨*proof*⟩

lemma *algebraic-root*:
assumes *algebraic y*
and *poly p x* = *y* **and** ∀ *i*. *coeff p i* ∈ ℤ **and** *lead-coeff p* = *1* **and** *degree p* > *0*
shows *algebraic x*
⟨*proof*⟩

lemma *algebraic-abs-real* [*simp*]:
algebraic |*x* :: *real*| ⟷ *algebraic x*
⟨*proof*⟩

lemma *algebraic-nth-root-real* [*intro*]:
assumes *algebraic x*
shows *algebraic* (*root n x*)
⟨*proof*⟩

lemma *algebraic-sqrt* [*intro*]: *algebraic x* ⟹ *algebraic* (*sqrt x*)
⟨*proof*⟩

lemma *algebraic-csqrt* [*intro*]: *algebraic x* ⟹ *algebraic* (*csqrt x*)
⟨*proof*⟩

lemma *algebraic-cnj* [*intro*]:
assumes *algebraic x*
shows *algebraic* (*cnj x*)
⟨*proof*⟩

lemma *algebraic-cnj-iff* [*simp*]: *algebraic* (*cnj x*) ⟷ *algebraic x*
⟨*proof*⟩

lemma *algebraic-of-real* [*intro*]:
assumes *algebraic x*
shows *algebraic* (*of-real x*)
⟨*proof*⟩

lemma *algebraic-of-real-iff* [*simp*]:
algebraic (*of-real x* :: *'a* :: {*real-algebra-1*,*field-char-0*}) ⟷ *algebraic x*
⟨*proof*⟩

4.29 Algebraic integers

inductive *algebraic-int* :: *'a* :: *field* ⇒ *bool* **where**
⟦*lead-coeff p* = *1*; ∀ *i*. *coeff p i* ∈ ℤ; *poly p x* = *0*⟧ ⟹ *algebraic-int x*

lemma *algebraic-int-altdef-ipoly*:
fixes *x* :: *'a* :: *field-char-0*
shows *algebraic-int x* ⟷ (∃ *p*. *poly* (*map-poly of-int p*) *x* = *0* ∧ *lead-coeff p* = *1*)
⟨*proof*⟩

theorem *rational-algebraic-int-is-int*:
assumes *algebraic-int x* **and** *x* ∈ ℚ
shows *x* ∈ ℤ
⟨*proof*⟩

lemma *algebraic-int-imp-algebraic* [*dest*]: *algebraic-int x* ⟹ *algebraic x*
⟨*proof*⟩

lemma *int-imp-algebraic-int*:
assumes *x* ∈ ℤ
shows *algebraic-int x*
⟨*proof*⟩

lemma *algebraic-int-0* [*simp*, *intro*]: *algebraic-int 0*
and *algebraic-int-1* [*simp*, *intro*]: *algebraic-int 1*
and *algebraic-int-numeral* [*simp*, *intro*]: *algebraic-int* (*numeral n*)
and *algebraic-int-of-nat* [*simp*, *intro*]: *algebraic-int* (*of-nat k*)
and *algebraic-int-of-int* [*simp*, *intro*]: *algebraic-int* (*of-int m*)
⟨*proof*⟩

lemma *algebraic-int-ii* [*simp*, *intro*]: *algebraic-int* i
⟨*proof*⟩

lemma *algebraic-int-minus* [*intro*]:
assumes *algebraic-int x*

shows *algebraic-int* $(-x)$
⟨*proof*⟩

lemma *algebraic-int-minus-iff* [*simp*]:
algebraic-int $(-x) \longleftrightarrow$ *algebraic-int* $(x :: {'a} :: \textit{field-char-0})$
⟨*proof*⟩

lemma *algebraic-int-inverse* [*intro*]:
assumes *poly p x* = *0* **and** ∀ *i*. *coeff p i* ∈ ℤ **and** *coeff p 0* = *1*
shows *algebraic-int* (*inverse x*)
⟨*proof*⟩

lemma *algebraic-int-root*:
assumes *algebraic-int y*
and *poly p x* = *y* **and** ∀ *i*. *coeff p i* ∈ ℤ **and** *lead-coeff p* = *1* **and** *degree p* > *0*
shows *algebraic-int x*
⟨*proof*⟩

lemma *algebraic-int-abs-real* [*simp*]:
algebraic-int |*x* :: *real*| ⟷ *algebraic-int x*
⟨*proof*⟩

lemma *algebraic-int-nth-root-real* [*intro*]:
assumes *algebraic-int x*
shows *algebraic-int* (*root n x*)
⟨*proof*⟩

lemma *algebraic-int-sqrt* [*intro*]: *algebraic-int x* ⟹ *algebraic-int* (*sqrt x*)
⟨*proof*⟩

lemma *algebraic-int-csqrt* [*intro*]: *algebraic-int x* ⟹ *algebraic-int* (*csqrt x*)
⟨*proof*⟩

lemma *algebraic-int-cnj* [*intro*]:
assumes *algebraic-int x*
shows *algebraic-int* (*cnj x*)
⟨*proof*⟩

lemma *algebraic-int-cnj-iff* [*simp*]: *algebraic-int* (*cnj x*) ⟷ *algebraic-int x*
⟨*proof*⟩

lemma *algebraic-int-of-real* [*intro*]:
assumes *algebraic-int x*
shows *algebraic-int* (*of-real x*)
⟨*proof*⟩

lemma *algebraic-int-of-real-iff* [*simp*]:
algebraic-int (*of-real x* :: *'a* :: {*field-char-0*, *real-algebra-1*}) ⟷ *algebraic-int x*

⟨*proof*⟩

4.30 Division of polynomials

4.30.1 Division in general

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

fun *divide-poly-main* :: *'a ⇒ 'a poly ⇒ 'a poly ⇒ 'a poly ⇒ nat ⇒ nat ⇒ 'a poly*
where
divide-poly-main lc q r d dr (*Suc n*) =
(*let cr = coeff r dr*; *a = cr div lc*; *mon = monom a n in*
if False ∨ *a* ∗ *lc* = *cr then* — *False* ∨ is only because of problem in function-package
divide-poly-main
lc
(*q* + *mon*)
(*r* − *mon* ∗ *d*)
d (*dr* − *1*) *n else 0*)
| *divide-poly-main lc q r d dr 0* = *q*

definition *divide-poly* :: *'a poly ⇒ 'a poly ⇒ 'a poly*
where *divide-poly f g* =
(*if g* = *0 then 0*
else
divide-poly-main (*coeff g* (*degree g*)) *0 f g* (*degree f*)
(*1* + *length* (*coeffs f*) − *length* (*coeffs g*)))

lemma *divide-poly-main*:
assumes *d*: *d* ≠ *0 lc* = *coeff d* (*degree d*)
and *degree* (*d* ∗ *r*) ≤ *dr divide-poly-main lc q* (*d* ∗ *r*) *d dr n* = *q'*
and *n* = *1* + *dr* − *degree d* ∨ *dr* = *0* ∧ *n* = *0* ∧ *d* ∗ *r* = *0*
shows *q'* = *q* + *r*
⟨*proof*⟩

lemma *divide-poly-main-0*: *divide-poly-main 0 0 r d dr n* = *0*
⟨*proof*⟩

lemma *divide-poly*:
assumes *g*: *g* ≠ *0*
shows (*f* ∗ *g*) *div g* = (*f* :: *'a poly*)
⟨*proof*⟩

lemma *divide-poly-0*: *f div 0* = *0*
for *f* :: *'a poly*
⟨*proof*⟩

instance
⟨*proof*⟩

end

instance *poly* :: (*idom-divide*) *algebraic-semidom* ⟨*proof*⟩

lemma *div-const-poly-conv-map-poly*:
  assumes [:*c*:] *dvd p*
  shows *p div* [:*c*:] = *map-poly* (λ*x*. *x div c*) *p*
⟨*proof*⟩

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

lemma *is-unit-triv*: *a* ≠ *0* ⟹ *is-unit* [:*a*:]
  for *a* :: *'a*::*field*
  ⟨*proof*⟩

lemma *is-unit-iff-degree*:
  fixes *p* :: *'a*::*field poly*
  assumes *p* ≠ *0*
  shows *is-unit p* ⟷ *degree p* = *0*
    (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *is-unit-pCons-iff*: *is-unit* (*pCons a p*) ⟷ *p* = *0* ∧ *a* ≠ *0*
  for *p* :: *'a*::*field poly*
  ⟨*proof*⟩

lemma *is-unit-monom-trivial*: *is-unit p* ⟹ *monom* (*coeff p* (*degree p*)) *0* = *p*
  for *p* :: *'a*::*field poly*
  ⟨*proof*⟩

lemma *is-unit-const-poly-iff*: [:*c*:] *dvd 1* ⟷ *c dvd 1*
  for *c* :: *'a*::{*comm-semiring-1*,*semiring-no-zero-divisors*}
  ⟨*proof*⟩

lemma *is-unit-polyE*:
  fixes *p* :: *'a* :: {*comm-semiring-1*,*semiring-no-zero-divisors*} *poly*
  assumes *p dvd 1*
  obtains *c* **where** *p* = [:*c*:] *c dvd 1*
⟨*proof*⟩

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

⟨*proof*⟩

lemma *is-unit-poly-iff*: *p dvd 1* ⟷ (∃ *c*. *p* = [:*c*:] ∧ *c dvd 1*)
for *p* :: *'a*::{*comm-semiring-1*,*semiring-no-zero-divisors*} *poly*
⟨*proof*⟩

lemma *root-imp-reducible-poly*:
fixes *x* :: *'a* :: *field*
assumes *poly p x* = *0* **and** *degree p* > *1*
shows ¬*irreducible p*
⟨*proof*⟩

lemma *reducible-polyI*:
fixes *p* :: *'a* :: *field poly*
assumes *p* = *q* ∗ *r degree q* > *0 degree r* > *0*
shows ¬*irreducible p*
⟨*proof*⟩

4.30.2 Pseudo-Division

This part is by René Thiemann and Akihisa Yamada.

fun *pseudo-divmod-main* ::
'a :: *comm-ring-1* ⇒ *'a poly* ⇒ *'a poly* ⇒ *'a poly* ⇒ *nat* ⇒ *nat* ⇒ *'a poly* × *'a poly*
where
pseudo-divmod-main lc q r d dr (*Suc n*) =
(*let*
rr = *smult lc r*;
qq = *coeff r dr*;
rrr = *rr* − *monom qq n* ∗ *d*;
qqq = *smult lc q* + *monom qq n*
in pseudo-divmod-main lc qqq rrr d (*dr* − *1*) *n*)
| *pseudo-divmod-main lc q r d dr 0* = (*q*,*r*)

definition *pseudo-divmod* :: *'a* :: *comm-ring-1 poly* ⇒ *'a poly* ⇒ *'a poly* × *'a poly*
where *pseudo-divmod p q* ≡
if q = *0 then* (*0*, *p*)
else
pseudo-divmod-main (*coeff q* (*degree q*)) *0 p q* (*degree p*)
(*1* + *length* (*coeffs p*) − *length* (*coeffs q*))

lemma *pseudo-divmod-main*:
assumes *d*: *d* ≠ *0 lc* = *coeff d* (*degree d*)
and *degree r* ≤ *dr pseudo-divmod-main lc q r d dr n* = (*q'*,*r'*)
and *n* = *1* + *dr* − *degree d* ∨ *dr* = *0* ∧ *n* = *0* ∧ *r* = *0*
shows (*r'* = *0* ∨ *degree r'* < *degree d*) ∧ *smult* (*lc*^*n*) (*d* ∗ *q* + *r*) = *d* ∗ *q'* + *r'*
⟨*proof*⟩

lemma *pseudo-divmod*:

assumes *g*: $g \neq 0$
and $*$: *pseudo-divmod f g* = (*q*,*r*)
shows *smult* (*coeff g* (*degree g*) ^(*Suc* (*degree f*) − *degree g*)) *f* = *g* ∗ *q* + *r* (**is** *?A*)
and *r* = *0* ∨ *degree r* < *degree g* (**is** *?B*)
⟨*proof*⟩

definition *pseudo-mod-main lc r d dr n* = *snd* (*pseudo-divmod-main lc 0 r d dr n*)

lemma *snd-pseudo-divmod-main*:
snd (*pseudo-divmod-main lc q r d dr n*) = *snd* (*pseudo-divmod-main lc q′ r d dr n*)
⟨*proof*⟩

definition *pseudo-mod* :: ′*a*::{*comm-ring-1*,*semiring-1-no-zero-divisors*} *poly* ⇒ ′*a poly* ⇒ ′*a poly*
where *pseudo-mod f g* = *snd* (*pseudo-divmod f g*)

lemma *pseudo-mod*:
fixes *f g* :: ′*a*::{*comm-ring-1*,*semiring-1-no-zero-divisors*} *poly*
defines *r* ≡ *pseudo-mod f g*
assumes *g*: $g \neq 0$
shows ∃ *a q*. *a* ≠ *0* ∧ *smult a f* = *g* ∗ *q* + *r r* = *0* ∨ *degree r* < *degree g*
⟨*proof*⟩

lemma *fst-pseudo-divmod-main-as-divide-poly-main*:
assumes *d*: $d \neq 0$
defines *lc*: *lc* ≡ *coeff d* (*degree d*)
shows *fst* (*pseudo-divmod-main lc q r d dr n*) =
divide-poly-main lc (*smult* (*lc*^*n*) *q*) (*smult* (*lc*^*n*) *r*) *d dr n*
⟨*proof*⟩

4.30.3 Division in polynomials over fields

lemma *pseudo-divmod-field*:
fixes *g* :: ′*a*::*field poly*
assumes *g*: $g \neq 0$
and $*$: *pseudo-divmod f g* = (*q*,*r*)
defines *c* ≡ *coeff g* (*degree g*) ^(*Suc* (*degree f*) − *degree g*)
shows *f* = *g* ∗ *smult* (*1*/*c*) *q* + *smult* (*1*/*c*) *r*
⟨*proof*⟩

lemma *divide-poly-main-field*:
fixes *d* :: ′*a*::*field poly*
assumes *d*: $d \neq 0$
defines *lc*: *lc* ≡ *coeff d* (*degree d*)
shows *divide-poly-main lc q r d dr n* =
fst (*pseudo-divmod-main lc* (*smult* ((*1* / *lc*)^*n*) *q*) (*smult* ((*1* / *lc*)^*n*) *r*) *d dr*

n)
  ⟨*proof*⟩

lemma *divide-poly-field*:
  fixes *f g* :: *'a::field poly*
  defines *f' ≡ smult* ((*1* / *coeff g* (*degree g*)) ^ (*Suc* (*degree f*) − *degree g*)) *f*
  shows *f div g* = *fst* (*pseudo-divmod f' g*)
⟨*proof*⟩

instantiation *poly* :: ({*semidom-divide-unit-factor*,*idom-divide*}) *normalization-semidom*
begin

definition *unit-factor-poly* :: *'a poly ⇒ 'a poly*
  where *unit-factor-poly p* = [:*unit-factor* (*lead-coeff p*):]

definition *normalize-poly* :: *'a poly ⇒ 'a poly*
  where *normalize p* = *p div* [:*unit-factor* (*lead-coeff p*):]

instance
⟨*proof*⟩

end

instance *poly* :: ({*semidom-divide-unit-factor*,*idom-divide*,*normalization-semidom-multiplicative*})
  normalization-semidom-multiplicative
  ⟨*proof*⟩

lemma *normalize-poly-eq-map-poly*: *normalize p* = *map-poly* (*λx. x div unit-factor* (*lead-coeff p*)) *p*
⟨*proof*⟩

lemma *coeff-normalize* [*simp*]:
  coeff (*normalize p*) *n* = *coeff p n div unit-factor* (*lead-coeff p*)
  ⟨*proof*⟩

class *field-unit-factor* = *field* + *unit-factor* +
  assumes *unit-factor-field* [*simp*]: *unit-factor* = *id*
begin

subclass *semidom-divide-unit-factor*
⟨*proof*⟩

end

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

lemma *normalize-monom* [*simp*]: *normalize* (*monom a n*) = *monom* (*normalize*

a) *n*
⟨*proof*⟩

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

lemma *normalize-const-poly*: *normalize* [:*c*:] = [:*normalize c*:]
⟨*proof*⟩

lemma *normalize-smult*:
fixes *c* :: *'a* :: {*normalization-semidom-multiplicative*, *idom-divide*}
shows *normalize* (*smult c p*) = *smult* (*normalize c*) (*normalize p*)
⟨*proof*⟩

instantiation *poly* :: (*field*) *idom-modulo*
begin

definition *modulo-poly* :: *'a poly* ⇒ *'a poly* ⇒ *'a poly*
where *mod-poly-def*: *f mod g* =
(*if g* = *0 then f else pseudo-mod* (*smult* ((*1* / *lead-coeff g*) ^ (*Suc* (*degree f*) − *degree g*)) *f*) *g*)

instance
⟨*proof*⟩

end

lemma *pseudo-divmod-eq-div-mod*:
‹*pseudo-divmod f g* = (*f div g*, *f mod g*)› **if** ‹*lead-coeff g* = *1*›
⟨*proof*⟩

lemma *degree-mod-less-degree*:
‹*degree* (*x mod y*) < *degree y*› **if** ‹*y* ≠ *0*› ‹¬ *y dvd x*›
⟨*proof*⟩

instantiation *poly* :: (*field*) *unique-euclidean-ring*
begin

definition *euclidean-size-poly* :: *'a poly* ⇒ *nat*
where *euclidean-size-poly p* = (*if p* = *0 then 0 else 2* ^ *degree p*)

definition *division-segment-poly* :: *'a poly* ⇒ *'a poly*
where [*simp*]: *division-segment-poly p* = *1*

instance ⟨*proof*⟩

end

lemma *euclidean-relation-polyI* [*case-names by0 divides euclidean-relation*]:

‹$(x\ div\ y,\ x\ mod\ y) = (q,\ r)$›
if *by0*: ‹$y = 0 \Longrightarrow q = 0 \wedge r = x$›
and *divides*: ‹$y \neq 0 \Longrightarrow y\ dvd\ x \Longrightarrow r = 0 \wedge x = q * y$›
and *euclidean-relation*: ‹$y \neq 0 \Longrightarrow \neg\ y\ dvd\ x \Longrightarrow degree\ r < degree\ y \wedge x = q * y + r$›
⟨*proof*⟩

lemma *div-poly-eq-0-iff*:
‹$x\ div\ y = 0 \longleftrightarrow x = 0 \vee y = 0 \vee degree\ x < degree\ y$› **for** *x y* :: ‹*'a::field poly*›
⟨*proof*⟩

lemma *div-poly-less*:
‹$x\ div\ y = 0$› **if** ‹$degree\ x < degree\ y$› **for** *x y* :: ‹*'a::field poly*›
⟨*proof*⟩

lemma *mod-poly-less*:
‹$x\ mod\ y = x$› **if** ‹$degree\ x < degree\ y$›
⟨*proof*⟩

lemma *degree-div-less*:
‹$degree\ (x\ div\ y) < degree\ x$›
if ‹$degree\ x > 0$› ‹$degree\ y > 0$›
for *x y* :: ‹*'a::field poly*›
⟨*proof*⟩

lemma *degree-mod-less'*: $b \neq 0 \Longrightarrow a\ mod\ b \neq 0 \Longrightarrow degree\ (a\ mod\ b) < degree\ b$
⟨*proof*⟩

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

lemma *div-smult-left*: ‹$smult\ a\ x\ div\ y = smult\ a\ (x\ div\ y)$› (**is** *?Q*)
and *mod-smult-left*: ‹$smult\ a\ x\ mod\ y = smult\ a\ (x\ mod\ y)$› (**is** *?R*)
for *x y* :: ‹*'a::field poly*›
⟨*proof*⟩

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

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

lemma *poly-div-add-left*: ‹$(x + y)\ div\ z = x\ div\ z + y\ div\ z$› (**is** *?Q*)
and *poly-mod-add-left*: ‹$(x + y)\ mod\ z = x\ mod\ z + y\ mod\ z$› (**is** *?R*)
for *x y z* :: ‹*'a::field poly*›
⟨*proof*⟩

lemma *poly-div-diff-left*: $(x - y)$ *div* $z = x$ *div* $z - y$ *div* z
for $x\ y\ z$ :: *'a::field poly*
⟨*proof*⟩

lemma *poly-mod-diff-left*: $(x - y)$ *mod* $z = x$ *mod* $z - y$ *mod* z
for $x\ y\ z$ :: *'a::field poly*
⟨*proof*⟩

lemma *div-smult-right*: ‹*x div smult a y* = *smult* (*inverse a*) (*x div y*)› (**is** *?Q*)
and *mod-smult-right*: ‹*x mod smult a y* = (*if a* = *0 then x else x mod y*)› (**is** *?R*)
⟨*proof*⟩

lemma *mod-mult-unit-eq*:
‹*x mod* (*z* ∗ *y*) = *x mod y*›
if ‹*is-unit z*›
for $x\ y\ z$ :: ‹*'a::field poly*›
⟨*proof*⟩

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

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

lemma *poly-div-mult-right*: ‹*x div* (*y* ∗ *z*) = (*x div y*) *div z*› (**is** *?Q*)
and *poly-mod-mult-right*: ‹*x mod* (*y* ∗ *z*) = *y* ∗ (*x div y mod z*) + *x mod y*› (**is** *?R*)
for $x\ y\ z$ :: ‹*'a::field poly*›
⟨*proof*⟩

lemma *dvd-pCons-imp-dvd-pCons-mod*:
‹*y dvd pCons a* (*x mod y*)› **if** ‹*y dvd pCons a x*›
⟨*proof*⟩

lemma *degree-less-if-less-eqI*:
‹*degree x* < *degree y*› **if** ‹*degree x* ≤ *degree y*› ‹*coeff x* (*degree y*) = *0*› ‹*x* ≠ *0*›
⟨*proof*⟩

lemma *div-pCons-eq*:
‹*pCons a p div q* = (*if q* = *0 then 0 else pCons* (*coeff* (*pCons a* (*p mod q*)) (*degree q*) / *lead-coeff q*) (*p div q*))› (**is** *?Q*)
and *mod-pCons-eq*:
‹*pCons a p mod q* = (*if q* = *0 then pCons a p else pCons a* (*p mod q*) − *smult* (*coeff* (*pCons a* (*p mod q*)) (*degree q*) / *lead-coeff q*) *q*)› (**is** *?R*)
for $x\ y$ :: ‹*'a::field poly*›
⟨*proof*⟩

```
lemma div-mod-fold-coeffs:
  (p div q, p mod q) =
    (if q = 0 then (0, p)
     else
      fold-coeffs
        (λa (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⟩

lemma mod-pCons:
  fixes a :: 'a::field
    and x y :: 'a::field poly
  assumes y: y ≠ 0
  defines 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⟩
```

4.30.4 List-based versions for fast implementation

```
fun minus-poly-rev-list :: 'a :: group-add list ⇒ 'a list ⇒ 'a list
  where
    minus-poly-rev-list (x # xs) (y # ys) = (x − y) # (minus-poly-rev-list xs ys)
  | minus-poly-rev-list xs [] = xs
  | minus-poly-rev-list [] (y # ys) = []

fun pseudo-divmod-main-list ::
  'a::comm-ring-1 ⇒ 'a list ⇒ 'a list ⇒ 'a list ⇒ nat ⇒ 'a list × 'a list
  where
    pseudo-divmod-main-list lc q r d (Suc n) =
      (let
        rr = map ((∗) lc) r;
        a = hd r;
        qqq = cCons a (map ((∗) lc) q);
        rrr = tl (if a = 0 then rr else minus-poly-rev-list rr (map ((∗) a) d))
       in pseudo-divmod-main-list lc qqq rrr d n)
  | pseudo-divmod-main-list lc q r d 0 = (q, r)

fun pseudo-mod-main-list :: 'a::comm-ring-1 ⇒ 'a list ⇒ 'a list ⇒ nat ⇒ 'a list
  where
    pseudo-mod-main-list lc r d (Suc n) =
      (let
        rr = map ((∗) lc) r;
        a = hd r;
        rrr = tl (if a = 0 then rr else minus-poly-rev-list rr (map ((∗) a) d))
       in pseudo-mod-main-list lc rrr d n)
  | pseudo-mod-main-list lc r d 0 = r
```

fun *divmod-poly-one-main-list* ::
  'a::comm-ring-1 list ⇒ 'a list ⇒ 'a list ⇒ nat ⇒ 'a list × 'a list
 where
  divmod-poly-one-main-list q r d (Suc n) =
   (let
    a = hd r;
    qqq = cCons a q;
    rr = tl (if a = 0 then r else minus-poly-rev-list r (map ((∗) a) d))
   in divmod-poly-one-main-list qqq rr d n)
 | *divmod-poly-one-main-list q r d 0 = (q, r)*

fun *mod-poly-one-main-list* :: *'a::comm-ring-1 list ⇒ 'a list ⇒ nat ⇒ 'a list*
 where
  mod-poly-one-main-list r d (Suc n) =
   (let
    a = hd r;
    rr = tl (if a = 0 then r else minus-poly-rev-list r (map ((∗) a) d))
   in mod-poly-one-main-list rr d n)
 | *mod-poly-one-main-list r d 0 = r*

definition *pseudo-divmod-list* :: *'a::comm-ring-1 list ⇒ 'a list ⇒ 'a list × 'a list*
 where *pseudo-divmod-list p q =*
  (if q = [] then ([], p)
  else
   (let rq = rev q;
     (qu,re) = pseudo-divmod-main-list (hd rq) [] (rev p) rq (1 + length p − length q)
   in (qu, rev re)))

definition *pseudo-mod-list* :: *'a::comm-ring-1 list ⇒ 'a list ⇒ 'a list*
 where *pseudo-mod-list p q =*
  (if q = [] then p
  else
   (let
    rq = rev q;
    re = pseudo-mod-main-list (hd rq) (rev p) rq (1 + length p − length q)
   in rev re))

lemma *minus-zero-does-nothing*: *minus-poly-rev-list x (map ((∗) 0) y) = x*
 for *x* :: *'a::ring list*
 ⟨*proof*⟩

lemma *length-minus-poly-rev-list* [*simp*]: *length (minus-poly-rev-list xs ys) = length xs*
 ⟨*proof*⟩

lemma *if-0-minus-poly-rev-list*:
 (if a = 0 then x else minus-poly-rev-list x (map ((∗) a) y)) =
  minus-poly-rev-list x (map ((∗) a) y)

for *a* :: *'a::ring*
⟨*proof*⟩

lemma *Poly-append*: *Poly* (*a* @ *b*) = *Poly a* + *monom 1* (*length a*) ∗ *Poly b*
for *a* :: *'a::comm-semiring-1 list*
⟨*proof*⟩

lemma *minus-poly-rev-list*: *length p* ≥ *length q* ⟹
Poly (*rev* (*minus-poly-rev-list* (*rev p*) (*rev q*))) =
Poly p − *monom 1* (*length p* − *length q*) ∗ *Poly q*
for *p q* :: *'a* :: *comm-ring-1 list*
⟨*proof*⟩

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

lemma *head-minus-poly-rev-list*:
length d ≤ *length r* ⟹ *d* ≠ [] ⟹
hd (*minus-poly-rev-list* (*map* ((∗) (*last d*)) *r*) (*map* ((∗) (*hd r*)) (*rev d*))) = *0*
for *d r* :: *'a::comm-ring list*
⟨*proof*⟩

lemma *Poly-map*: *Poly* (*map* ((∗) *a*) *p*) = *smult a* (*Poly p*)
⟨*proof*⟩

lemma *last-coeff-is-hd*: *xs* ≠ [] ⟹ *coeff* (*Poly xs*) (*length xs* − *1*) = *hd* (*rev xs*)
⟨*proof*⟩

lemma *pseudo-divmod-main-list-invar*:
assumes *leading-nonzero*: *last d* ≠ *0*
and *lc*: *last d* = *lc*
and *d* ≠ []
and *pseudo-divmod-main-list lc q* (*rev r*) (*rev d*) *n* = (*q'*, *rev r'*)
and *n* = *1* + *length r* − *length d*
shows *pseudo-divmod-main lc* (*monom 1 n* ∗ *Poly q*) (*Poly r*) (*Poly d*) (*length r* − *1*) *n* =
(*Poly q'*, *Poly r'*)
⟨*proof*⟩

lemma *pseudo-divmod-impl* [*code*]:
pseudo-divmod f g = *map-prod poly-of-list poly-of-list* (*pseudo-divmod-list* (*coeffs f*) (*coeffs g*))
for *f g* :: *'a::comm-ring-1 poly*
⟨*proof*⟩

lemma *pseudo-mod-main-list*:
snd (*pseudo-divmod-main-list l q xs ys n*) = *pseudo-mod-main-list l xs ys n*
⟨*proof*⟩

lemma *pseudo-mod-impl*[*code*]: *pseudo-mod f g* = *poly-of-list* (*pseudo-mod-list* (*coeffs f*) (*coeffs g*))
⟨*proof*⟩

4.30.5 Improved Code-Equations for Polynomial (Pseudo) Division

lemma *pdivmod-via-pseudo-divmod*:
‹(*f div g*, *f mod g*) =
(*if g* = *0 then* (*0*, *f*)
else
let
ilc = *inverse* (*lead-coeff g*);
h = *smult ilc g*;
(*q*,*r*) = *pseudo-divmod f h*
in (*smult ilc q*, *r*))›
(**is** ‹*?l* = *?r*›)
⟨*proof*⟩

lemma *pdivmod-via-pseudo-divmod-list*:
(*f div g*, *f mod g*) =
(*let cg* = *coeffs g in*
if cg = [] *then* (*0*, *f*)
else
let
cf = *coeffs f*;
ilc = *inverse* (*last cg*);
ch = *map* ((∗) *ilc*) *cg*;
(*q*, *r*) = *pseudo-divmod-main-list 1* [] (*rev cf*) (*rev ch*) (*1* + *length cf* − *length cg*)
in (*poly-of-list* (*map* ((∗) *ilc*) *q*), *poly-of-list* (*rev r*)))
⟨*proof*⟩

lemma *pseudo-divmod-main-list-1*: *pseudo-divmod-main-list 1* = *divmod-poly-one-main-list*
⟨*proof*⟩

fun *divide-poly-main-list* :: ′*a*::*idom-divide* ⇒ ′*a list* ⇒ ′*a list* ⇒ ′*a list* ⇒ *nat* ⇒ ′*a list*
where
divide-poly-main-list lc q r d (*Suc n*) =
(*let*
cr = *hd r*
in if cr = *0 then divide-poly-main-list lc* (*cCons cr q*) (*tl r*) *d n else let*
a = *cr div lc*;
qq = *cCons a q*;
rr = *minus-poly-rev-list r* (*map* ((∗) *a*) *d*)
in if hd rr = *0 then divide-poly-main-list lc qq* (*tl rr*) *d n else* [])
| *divide-poly-main-list lc q r d 0* = *q*

lemma *divide-poly-main-list-simp* [*simp*]:
  divide-poly-main-list lc q r d (*Suc n*) =
    (*let*
      cr = *hd r*;
      a = *cr div lc*;
      qq = *cCons a q*;
      rr = *minus-poly-rev-list r* (*map* ((∗) *a*) *d*)
     in if hd rr = *0 then divide-poly-main-list lc qq* (*tl rr*) *d n else* [])
  ⟨*proof*⟩

declare *divide-poly-main-list.simps*(*1*)[*simp del*]

definition *divide-poly-list* :: *'a::idom-divide poly* ⇒ *'a poly* ⇒ *'a poly*
  where *divide-poly-list f g* =
    (*let cg* = *coeffs g in*
      if cg = [] *then g*
      else
        let
          cf = *coeffs f*;
          cgr = *rev cg*
         in poly-of-list (*divide-poly-main-list* (*hd cgr*) [] (*rev cf*) *cgr* (*1* + *length cf*
− *length cg*)))

lemmas *pdivmod-via-divmod-list* = *pdivmod-via-pseudo-divmod-list*[*unfolded pseudo-divmod-main-list-1*]

lemma *mod-poly-one-main-list*: *snd* (*divmod-poly-one-main-list q r d n*) = *mod-poly-one-main-list*
r d n
  ⟨*proof*⟩

lemma *mod-poly-code* [*code*]:
  f mod g =
    (*let cg* = *coeffs g in*
      if cg = [] *then f*
      else
        let
          cf = *coeffs f*;
          ilc = *inverse* (*last cg*);
          ch = *map* ((∗) *ilc*) *cg*;
          r = *mod-poly-one-main-list* (*rev cf*) (*rev ch*) (*1* + *length cf* − *length cg*)
        in poly-of-list (*rev r*))
  (**is** - = *?rhs*)
⟨*proof*⟩

definition *div-field-poly-impl* :: *'a* :: *field poly* ⇒ *'a poly* ⇒ *'a poly*
  where *div-field-poly-impl f g* =
    (*let cg* = *coeffs g in*
      if cg = [] *then 0*
      else
        let

cf = *coeffs f*;
ilc = *inverse* (*last cg*);
ch = *map* ((∗) *ilc*) *cg*;
q = *fst* (*divmod-poly-one-main-list* [] (*rev cf*) (*rev ch*) (*1* + *length cf* − *length cg*))
in poly-of-list ((*map* ((∗) *ilc*) *q*)))

We do not declare the following lemma as code equation, since then polynomial division on non-fields will no longer be executable. However, a code-unfold is possible, since *div-field-poly-impl* is a bit more efficient than the generic polynomial division.

lemma *div-field-poly-impl*[*code-unfold*]: (*div*) = *div-field-poly-impl*
⟨*proof*⟩

lemma *divide-poly-main-list*:
assumes *lc0*: *lc* ≠ *0*
and *lc*: *last d* = *lc*
and *d*: *d* ≠ []
and *n* = (*1* + *length r* − *length d*)
shows *Poly* (*divide-poly-main-list lc q* (*rev r*) (*rev d*) *n*) =
divide-poly-main lc (*monom 1 n* ∗ *Poly q*) (*Poly r*) (*Poly d*) (*length r* − *1*) *n*
⟨*proof*⟩

lemma *divide-poly-list*[*code*]: *f div g* = *divide-poly-list f g*
⟨*proof*⟩

4.31 Primality and irreducibility in polynomial rings

lemma *prod-mset-const-poly*: (∏ *x*∈#*A*. [:*f x*:]) = [:*prod-mset* (*image-mset f A*):]
⟨*proof*⟩

lemma *irreducible-const-poly-iff*:
fixes *c* :: ′*a* :: {*comm-semiring-1*,*semiring-no-zero-divisors*}
shows *irreducible* [:*c*:] ⟷ *irreducible c*
⟨*proof*⟩

lemma *lift-prime-elem-poly*:
assumes *prime-elem* (*c* :: ′*a* :: *semidom*)
shows *prime-elem* [:*c*:]
⟨*proof*⟩

lemma *prime-elem-const-poly-iff*:
fixes *c* :: ′*a* :: *semidom*
shows *prime-elem* [:*c*:] ⟷ *prime-elem c*
⟨*proof*⟩

4.32 Content and primitive part of a polynomial

definition *content* :: ′*a*::*semiring-gcd poly* ⇒ ′*a*

where *content p* = *gcd-list* (*coeffs p*)

lemma *content-eq-fold-coeffs* [*code*]: *content p* = *fold-coeffs gcd p 0*
⟨*proof*⟩

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

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

lemma *content-const* [*simp*]: *content* [:*c*:] = *normalize c*
⟨*proof*⟩

lemma *const-poly-dvd-iff-dvd-content*: [:*c*:] *dvd p* ⟷ *c dvd content p*
for *c* :: ′*a*::*semiring-gcd*
⟨*proof*⟩

lemma *content-dvd* [*simp*]: [:*content p*:] *dvd p*
⟨*proof*⟩

lemma *content-dvd-coeff* [*simp*]: *content p dvd coeff p n*
⟨*proof*⟩

lemma *content-dvd-coeffs*: *c* ∈ *set* (*coeffs p*) ⟹ *content p dvd c*
⟨*proof*⟩

lemma *normalize-content* [*simp*]: *normalize* (*content p*) = *content p*
⟨*proof*⟩

lemma *is-unit-content-iff* [*simp*]: *is-unit* (*content p*) ⟷ *content p* = *1*
⟨*proof*⟩

lemma *content-smult* [*simp*]:
fixes *c* :: ′*a* :: {*normalization-semidom-multiplicative*, *semiring-gcd*}
shows *content* (*smult c p*) = *normalize c* ∗ *content p*
⟨*proof*⟩

lemma *content-eq-zero-iff* [*simp*]: *content p* = *0* ⟷ *p* = *0*
⟨*proof*⟩

definition *primitive-part* :: ′*a* :: *semiring-gcd poly* ⇒ ′*a poly*
where *primitive-part p* = *map-poly* (λ*x*. *x div content p*) *p*

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

lemma *content-times-primitive-part* [*simp*]: *smult* (*content p*) (*primitive-part p*) = *p*

for *p* :: *'a* :: *semiring-gcd poly*
⟨*proof*⟩

lemma *primitive-part-eq-0-iff* [*simp*]: *primitive-part p* = *0* ⟷ *p* = *0*
⟨*proof*⟩

lemma *content-primitive-part* [*simp*]:
fixes *p* :: *'a* :: {*normalization-semidom-multiplicative*, *semiring-gcd*} *poly*
assumes *p* ≠ *0*
shows *content* (*primitive-part p*) = *1*
⟨*proof*⟩

lemma *content-decompose*:
obtains *p'* :: *'a* :: {*normalization-semidom-multiplicative*, *semiring-gcd*} *poly*
where *p* = *smult* (*content p*) *p' content p'* = *1*
⟨*proof*⟩

lemma *content-dvd-contentI* [*intro*]: *p dvd q* ⟹ *content p dvd content q*
⟨*proof*⟩

lemma *primitive-part-const-poly* [*simp*]: *primitive-part* [:*x*:] = [:*unit-factor x*:]
⟨*proof*⟩

lemma *primitive-part-prim*: *content p* = *1* ⟹ *primitive-part p* = *p*
⟨*proof*⟩

lemma *degree-primitive-part* [*simp*]: *degree* (*primitive-part p*) = *degree p*
⟨*proof*⟩

lemma *smult-content-normalize-primitive-part* [*simp*]:
fixes *p* :: *'a* :: {*normalization-semidom-multiplicative*, *semiring-gcd*, *idom-divide*} *poly*
shows *smult* (*content p*) (*normalize* (*primitive-part p*)) = *normalize p*
⟨*proof*⟩

context
begin

private

lemma *content-1-mult*:
fixes *f g* :: *'a* :: {*semiring-gcd*, *factorial-semiring*} *poly*
assumes *content f* = *1 content g* = *1*
shows *content* (*f* ∗ *g*) = *1*
⟨*proof*⟩

lemma *content-mult*:
fixes *p q* :: *'a* :: {*factorial-semiring*, *semiring-gcd*, *normalization-semidom-multiplicative*} *poly*

shows *content* $(p * q) = $ *content* $p *$ *content* q
⟨*proof*⟩

end

lemma *primitive-part-mult*:
fixes $p\ q$ :: $'a$ :: {*factorial-semiring*, *semiring-Gcd*, *ring-gcd*, *idom-divide*, *normalization-semidom-multiplicative*} *poly*
shows *primitive-part* $(p * q) = $ *primitive-part* $p *$ *primitive-part* q
⟨*proof*⟩

lemma *primitive-part-smult*:
fixes p :: $'a$ :: {*factorial-semiring*, *semiring-Gcd*, *ring-gcd*, *idom-divide*, *normalization-semidom-multiplicative*} *poly*
shows *primitive-part* (*smult a p*) = *smult* (*unit-factor a*) (*primitive-part p*)
⟨*proof*⟩

lemma *primitive-part-dvd-primitive-partI* [*intro*]:
fixes $p\ q$ :: $'a$ :: {*factorial-semiring*, *semiring-Gcd*, *ring-gcd*, *idom-divide*, *normalization-semidom-multiplicative*} *poly*
shows *p dvd q* $\Longrightarrow$ *primitive-part p dvd primitive-part q*
⟨*proof*⟩

lemma *content-prod-mset*:
fixes A :: $'a$ :: {*factorial-semiring*, *semiring-Gcd*, *normalization-semidom-multiplicative*} *poly multiset*
shows *content* (*prod-mset A*) = *prod-mset* (*image-mset content A*)
⟨*proof*⟩

lemma *content-prod-eq-1-iff*:
fixes $p\ q$ :: $'a$ :: {*factorial-semiring*, *semiring-Gcd*, *normalization-semidom-multiplicative*} *poly*
shows *content* $(p * q) = 1 \longleftrightarrow$ *content* $p = 1 \land$ *content* $q = 1$
⟨*proof*⟩

4.33 A typeclass for algebraically closed fields

Since the required sort constraints are not available inside the class, we have to resort to a somewhat awkward way of writing the definition of algebraically closed fields:

class *alg-closed-field* = *field* +
assumes *alg-closed*: $n > 0 \Longrightarrow f\ n \neq 0 \Longrightarrow \exists x.\ (\sum k \leq n.\ f\ k * x\ \hat{}\ k) = 0$

We can then however easily show the equivalence to the proper definition:

lemma *alg-closed-imp-poly-has-root*:
assumes *degree* (p :: $'a$ :: *alg-closed-field poly*) > 0
shows $\exists x.$ *poly p x* $= 0$
⟨*proof*⟩

lemma *alg-closedI* [*Pure.intro*]:
  assumes $\bigwedge p :: 'a\ poly.\ degree\ p > 0 \Longrightarrow lead\text{-}coeff\ p = 1 \Longrightarrow \exists x.\ poly\ p\ x = 0$
  shows $OFCLASS('a :: field,\ alg\text{-}closed\text{-}field\text{-}class)$
⟨*proof*⟩

lemma (**in** *alg-closed-field*) *nth-root-exists*:
  assumes $n > 0$
  shows $\exists y.\ y \mathbin{\char`\^} n = (x :: 'a)$
⟨*proof*⟩

We can now prove by induction that every polynomial of degree n splits into a product of n linear factors:

lemma *alg-closed-imp-factorization*:
  fixes $p :: 'a :: alg\text{-}closed\text{-}field\ poly$
  assumes $p \neq 0$
  shows $\exists A.\ size\ A = degree\ p \wedge p = smult\ (lead\text{-}coeff\ p)\ (\prod x \in\# A.\ [:-x,\ 1:])$
  ⟨*proof*⟩

As an alternative characterisation of algebraic closure, one can also say that any polynomial of degree at least 2 splits into non-constant factors:

lemma *alg-closed-imp-reducible*:
  assumes $degree\ (p :: 'a :: alg\text{-}closed\text{-}field\ poly) > 1$
  shows $\neg irreducible\ p$
⟨*proof*⟩

When proving algebraic closure through reducibility, we can assume w.l.o.g. that the polynomial is monic and has a non-zero constant coefficient:

lemma *alg-closedI-reducible*:
  assumes $\bigwedge p :: 'a\ poly.\ degree\ p > 1 \Longrightarrow lead\text{-}coeff\ p = 1 \Longrightarrow coeff\ p\ 0 \neq 0 \Longrightarrow$
    $\neg irreducible\ p$
  shows $OFCLASS('a :: field,\ alg\text{-}closed\text{-}field\text{-}class)$
⟨*proof*⟩

Using a clever Tschirnhausen transformation mentioned e.g. in the article by Nowak [1], we can also assume w.l.o.g. that the coefficient a_{n-1} is zero.

lemma *alg-closedI-reducible-coeff-deg-minus-one-eq-0*:
  assumes $\bigwedge p :: 'a\ poly.\ degree\ p > 1 \Longrightarrow lead\text{-}coeff\ p = 1 \Longrightarrow coeff\ p\ (degree\ p$
$-\ 1) = 0 \Longrightarrow$
    $coeff\ p\ 0 \neq 0 \Longrightarrow \neg irreducible\ p$
  shows $OFCLASS('a :: field\text{-}char\text{-}0,\ alg\text{-}closed\text{-}field\text{-}class)$
⟨*proof*⟩

As a consequence of the full factorisation lemma proven above, we can also show that any polynomial with at least two different roots splits into two non-constant coprime factors:

lemma *alg-closed-imp-poly-splits-coprime*:

assumes *degree* (*p* :: *'a* :: {*alg-closed-field*} *poly*) > *1*
assumes *poly p x* = *0 poly p y* = *0 x* ≠ *y*
obtains *r s* **where** *degree r* > *0 degree s* > *0 coprime r s p* = *r* ∗ *s*
⟨*proof*⟩

no-notation *cCons* (**infixr** ## *65*)

end

5 A formalization of formal power series

theory *Formal-Power-Series*
imports
Complex-Main
Euclidean-Algorithm
Primes
begin

5.1 The type of formal power series

typedef *'a fps* = {*f* :: *nat* ⇒ *'a*. *True*}
morphisms *fps-nth Abs-fps*
⟨*proof*⟩

notation *fps-nth* (**infixl** $ *75*)

lemma *expand-fps-eq*: *p* = *q* ⟷ (∀ *n*. *p* $ *n* = *q* $ *n*)
⟨*proof*⟩

lemmas *fps-eq-iff* = *expand-fps-eq*

lemma *fps-ext*: (⋀*n*. *p* $ *n* = *q* $ *n*) ⟹ *p* = *q*
⟨*proof*⟩

lemma *fps-nth-Abs-fps* [*simp*]: *Abs-fps f* $ *n* = *f n*
⟨*proof*⟩

Definition of the basic elements 0 and 1 and the basic operations of addition, negation and multiplication.

instantiation *fps* :: (*zero*) *zero*
begin
definition *fps-zero-def*: *0* = *Abs-fps* (λ*n*. *0*)
instance ⟨*proof*⟩
end

lemma *fps-zero-nth* [*simp*]: *0* $ *n* = *0*
⟨*proof*⟩

lemma *fps-nonzero-nth*: *f* ≠ *0* ⟷ (∃ *n*. *f* $ *n* ≠ *0*)

⟨*proof*⟩

lemma *fps-nonzero-nth-minimal*: $f \neq 0 \longleftrightarrow (\exists\, n.\ f \ \$ \ n \neq 0 \wedge (\forall\, m < n.\ f \ \$ \ m = 0))$
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *fps-nonzeroI*: $f\$n \neq 0 \Longrightarrow f \neq 0$
⟨*proof*⟩

instantiation *fps* :: ({*one*, *zero*}) *one*
begin
definition *fps-one-def*: $1 = \textit{Abs-fps}\ (\lambda n.\ \textit{if}\ n = 0\ \textit{then}\ 1\ \textit{else}\ 0)$
instance ⟨*proof*⟩
end

lemma *fps-one-nth* [*simp*]: $1 \ \$ \ n = (\textit{if}\ n = 0\ \textit{then}\ 1\ \textit{else}\ 0)$
⟨*proof*⟩

instantiation *fps* :: (*plus*) *plus*
begin
definition *fps-plus-def*: $(+) = (\lambda f\ g.\ \textit{Abs-fps}\ (\lambda n.\ f \ \$ \ n + g \ \$ \ n))$
instance ⟨*proof*⟩
end

lemma *fps-add-nth* [*simp*]: $(f + g) \ \$ \ n = f \ \$ \ n + g \ \$ \ n$
⟨*proof*⟩

instantiation *fps* :: (*minus*) *minus*
begin
definition *fps-minus-def*: $(-) = (\lambda f\ g.\ \textit{Abs-fps}\ (\lambda n.\ f \ \$ \ n - g \ \$ \ n))$
instance ⟨*proof*⟩
end

lemma *fps-sub-nth* [*simp*]: $(f - g) \ \$ \ n = f \ \$ \ n - g \ \$ \ n$
⟨*proof*⟩

instantiation *fps* :: (*uminus*) *uminus*
begin
definition *fps-uminus-def*: $\textit{uminus} = (\lambda f.\ \textit{Abs-fps}\ (\lambda n.\ - (f \ \$ \ n)))$
instance ⟨*proof*⟩
end

lemma *fps-neg-nth* [*simp*]: $(- f) \ \$ \ n = - (f \ \$ \ n)$
⟨*proof*⟩

lemma *fps-neg-0* [*simp*]: $-(0::'a::\textit{group-add fps}) = 0$
⟨*proof*⟩

instantiation *fps* :: ({*comm-monoid-add*, *times*}) *times*
begin
definition *fps-times-def*: $(*) = (\lambda f\ g.\ Abs\text{-}fps\ (\lambda n.\ \sum i{=}0..n.\ f\ \$\ i * g\ \$\ (n - i)))$
instance ⟨*proof*⟩
end

lemma *fps-mult-nth*: $(f * g)\ \$\ n = (\sum i{=}0..n.\ f\$i * g\$(n - i))$
⟨*proof*⟩

lemma *fps-mult-nth-0* [*simp*]: $(f * g)\ \$\ 0 = f\ \$\ 0 * g\ \$\ 0$
⟨*proof*⟩

lemma *fps-mult-nth-1*: $(f * g)\ \$\ 1 = f\$0 * g\$1 + f\$1 * g\$0$
⟨*proof*⟩

lemma *fps-mult-nth-1'* [*simp*]: $(f * g)\ \$\ Suc\ 0 = f\$0 * g\$Suc\ 0 + f\$Suc\ 0 * g\$0$
⟨*proof*⟩

lemmas *mult-nth-0* = *fps-mult-nth-0*
lemmas *mult-nth-1* = *fps-mult-nth-1*

instance *fps* :: ({*comm-monoid-add*, *mult-zero*}) *mult-zero*
⟨*proof*⟩

declare *atLeastAtMost-iff* [*presburger*]
declare *Bex-def* [*presburger*]
declare *Ball-def* [*presburger*]

lemma *mult-delta-left*:
fixes *x y* :: *'a::mult-zero*
shows (*if b then x else 0*) $*$ *y* = (*if b then x* $*$ *y else 0*)
⟨*proof*⟩

lemma *mult-delta-right*:
fixes *x y* :: *'a::mult-zero*
shows *x* $*$ (*if b then y else 0*) = (*if b then x* $*$ *y else 0*)
⟨*proof*⟩

lemma *fps-one-mult*:
fixes *f* :: *'a::{comm-monoid-add, mult-zero, monoid-mult} fps*
shows $1 * f = f$
and $f * 1 = f$
⟨*proof*⟩

5.2 Subdegrees

definition *subdegree* :: (*'a::zero*) *fps* ⇒ *nat* **where**
subdegree f = (*if f* = *0 then 0 else LEAST n. f$n* ≠ *0*)

lemma *subdegreeI*:
assumes $f \$ d \neq 0$ **and** $\bigwedge i.\ i < d \Longrightarrow f \$ i = 0$
shows *subdegree* $f = d$
⟨*proof*⟩

lemma *nth-subdegree-nonzero* [*simp,intro*]: $f \neq 0 \Longrightarrow f \$ \mathit{subdegree}\ f \neq 0$
⟨*proof*⟩

lemma *nth-less-subdegree-zero* [*dest*]: $n < \mathit{subdegree}\ f \Longrightarrow f \$ n = 0$
⟨*proof*⟩

lemma *subdegree-geI*:
assumes $f \neq 0$ $\bigwedge i.\ i < n \Longrightarrow f\$i = 0$
shows *subdegree* $f \geq n$
⟨*proof*⟩

lemma *subdegree-greaterI*:
assumes $f \neq 0$ $\bigwedge i.\ i \leq n \Longrightarrow f\$i = 0$
shows *subdegree* $f > n$
⟨*proof*⟩

lemma *subdegree-leI*:
$f \$ n \neq 0 \Longrightarrow \mathit{subdegree}\ f \leq n$
⟨*proof*⟩

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

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

lemma *subdegree-eq-0-iff*: *subdegree* $f = 0 \longleftrightarrow f = 0 \vee f \$ 0 \neq 0$
⟨*proof*⟩

lemma *subdegree-eq-0* [*simp*]: $f \$ 0 \neq 0 \Longrightarrow \mathit{subdegree}\ f = 0$
⟨*proof*⟩

lemma *nth-subdegree-zero-iff* [*simp*]: $f \$ \mathit{subdegree}\ f = 0 \longleftrightarrow f = 0$
⟨*proof*⟩

lemma *fps-nonzero-subdegree-nonzeroI*: *subdegree* $f > 0 \Longrightarrow f \neq 0$
⟨*proof*⟩

lemma *subdegree-uminus* [*simp*]:
subdegree $(-(f::('a::\mathit{group\text{-}add})\ \mathit{fps})) = \mathit{subdegree}\ f$
⟨*proof*⟩

lemma *subdegree-minus-commute* [*simp*]:

fixes *f* :: *'a::group-add fps*
shows *subdegree* (*f*−*g*) = *subdegree* (*g* − *f*)
⟨*proof*⟩

lemma *subdegree-add-ge'*:
fixes *f g* :: *'a::monoid-add fps*
assumes *f* + *g* ≠ *0*
shows *subdegree* (*f* + *g*) ≥ *min* (*subdegree f*) (*subdegree g*)
⟨*proof*⟩

lemma *subdegree-add-ge*:
assumes *f* ≠ −(*g* :: (*'a* :: *group-add*) *fps*)
shows *subdegree* (*f* + *g*) ≥ *min* (*subdegree f*) (*subdegree g*)
⟨*proof*⟩

lemma *subdegree-add-eq1*:
assumes *f* ≠ *0*
and *subdegree f* < *subdegree* (*g* :: *'a::monoid-add fps*)
shows *subdegree* (*f* + *g*) = *subdegree f*
⟨*proof*⟩

lemma *subdegree-add-eq2*:
assumes *g* ≠ *0*
and *subdegree g* < *subdegree* (*f* :: *'a* :: *monoid-add fps*)
shows *subdegree* (*f* + *g*) = *subdegree g*
⟨*proof*⟩

lemma *subdegree-diff-eq1*:
assumes *f* ≠ *0*
and *subdegree f* < *subdegree* (*g* :: *'a* :: *group-add fps*)
shows *subdegree* (*f* − *g*) = *subdegree f*
⟨*proof*⟩

lemma *subdegree-diff-eq1-cancel*:
assumes *f* ≠ *0*
and *subdegree f* < *subdegree* (*g* :: *'a* :: *cancel-comm-monoid-add fps*)
shows *subdegree* (*f* − *g*) = *subdegree f*
⟨*proof*⟩

lemma *subdegree-diff-eq2*:
assumes *g* ≠ *0*
and *subdegree g* < *subdegree* (*f* :: *'a* :: *group-add fps*)
shows *subdegree* (*f* − *g*) = *subdegree g*
⟨*proof*⟩

lemma *subdegree-diff-ge* [*simp*]:
assumes *f* ≠ (*g* :: *'a* :: *group-add fps*)
shows *subdegree* (*f* − *g*) ≥ *min* (*subdegree f*) (*subdegree g*)
⟨*proof*⟩

lemma *subdegree-diff-ge′*:
fixes *f g* :: *′a* :: *comm-monoid-diff fps*
assumes $f - g \neq 0$
shows *subdegree* $(f - g) \geq$ *subdegree f*
⟨*proof*⟩

lemma *nth-subdegree-mult-left* [*simp*]:
fixes *f g* :: (*′a* :: {*mult-zero*,*comm-monoid-add*}) *fps*
shows $(f * g)$ \$ (*subdegree f*) = *f* \$ *subdegree f* ∗ *g* \$ *0*
⟨*proof*⟩

lemma *nth-subdegree-mult-right* [*simp*]:
fixes *f g* :: (*′a* :: {*mult-zero*,*comm-monoid-add*}) *fps*
shows $(f * g)$ \$ (*subdegree g*) = *f* \$ *0* ∗ *g* \$ *subdegree g*
⟨*proof*⟩

lemma *nth-subdegree-mult* [*simp*]:
fixes *f g* :: (*′a* :: {*mult-zero*,*comm-monoid-add*}) *fps*
shows $(f * g)$ \$ (*subdegree f* + *subdegree g*) = *f* \$ *subdegree f* ∗ *g* \$ *subdegree g*
⟨*proof*⟩

lemma *fps-mult-nth-eq0*:
fixes *f g* :: *′a*::{*comm-monoid-add*,*mult-zero*} *fps*
assumes *n* < *subdegree f* + *subdegree g*
shows $(f*g)$ \$ *n* = *0*
⟨*proof*⟩

lemma *fps-mult-subdegree-ge*:
fixes *f g* :: *′a*::{*comm-monoid-add*,*mult-zero*} *fps*
assumes $f*g \neq 0$
shows *subdegree* $(f*g) \geq$ *subdegree f* + *subdegree g*
⟨*proof*⟩

lemma *subdegree-mult′*:
fixes *f g* :: *′a*::{*comm-monoid-add*,*mult-zero*} *fps*
assumes *f* \$ *subdegree f* ∗ *g* \$ *subdegree g* ≠ *0*
shows *subdegree* $(f*g)$ = *subdegree f* + *subdegree g*
⟨*proof*⟩

lemma *subdegree-mult* [*simp*]:
fixes *f g* :: *′a* :: {*semiring-no-zero-divisors*} *fps*
assumes $f \neq 0$ $g \neq 0$
shows *subdegree* $(f * g)$ = *subdegree f* + *subdegree g*
⟨*proof*⟩

lemma *fps-mult-nth-conv-upto-subdegree-left*:
fixes *f g* :: (*′a* :: {*mult-zero*,*comm-monoid-add*}) *fps*
shows $(f * g)$ \$ *n* = (∑ *i*=*subdegree f*..*n*. *f* \$ *i* ∗ *g* \$ (*n* − *i*))

⟨*proof*⟩

lemma *fps-mult-nth-conv-upto-subdegree-right*:
fixes *f g* :: (*'a* :: {*mult-zero*,*comm-monoid-add*}) *fps*
shows $(f * g)$ \$ $n = (\sum i{=}0..n - subdegree\ g.\ f$ \$ $i * g$ \$ $(n - i))$
⟨*proof*⟩

lemma *fps-mult-nth-conv-inside-subdegrees*:
fixes *f g* :: (*'a* :: {*mult-zero*,*comm-monoid-add*}) *fps*
shows $(f * g)$ \$ $n = (\sum i{=}subdegree\ f..n - subdegree\ g.\ f$ \$ $i * g$ \$ $(n - i))$
⟨*proof*⟩

lemma *fps-mult-nth-outside-subdegrees*:
fixes *f g* :: (*'a* :: {*mult-zero*,*comm-monoid-add*}) *fps*
shows $n < subdegree\ f \Longrightarrow (f * g)$ \$ $n = 0$
and $n < subdegree\ g \Longrightarrow (f * g)$ \$ $n = 0$
⟨*proof*⟩

5.3 Ring structure

instance *fps* :: (*semigroup-add*) *semigroup-add*
⟨*proof*⟩

instance *fps* :: (*ab-semigroup-add*) *ab-semigroup-add*
⟨*proof*⟩

instance *fps* :: (*monoid-add*) *monoid-add*
⟨*proof*⟩

instance *fps* :: (*comm-monoid-add*) *comm-monoid-add*
⟨*proof*⟩

instance *fps* :: (*cancel-semigroup-add*) *cancel-semigroup-add*
⟨*proof*⟩

instance *fps* :: (*cancel-ab-semigroup-add*) *cancel-ab-semigroup-add*
⟨*proof*⟩

instance *fps* :: (*cancel-comm-monoid-add*) *cancel-comm-monoid-add* ⟨*proof*⟩

instance *fps* :: (*group-add*) *group-add*
⟨*proof*⟩

instance *fps* :: (*ab-group-add*) *ab-group-add*
⟨*proof*⟩

instance *fps* :: (*zero-neq-one*) *zero-neq-one*
⟨*proof*⟩

lemma *fps-mult-assoc-lemma*:
  fixes k :: *nat*
    and f :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *'a::comm-monoid-add*
  shows $(\sum j=0..k.\ \sum i=0..j.\ f\ i\ (j - i)\ (n - j)) =$
      $(\sum j=0..k.\ \sum i=0..k - j.\ f\ j\ i\ (n - j - i))$
  ⟨*proof*⟩

instance *fps* :: (*semiring-0*) *semiring-0*
⟨*proof*⟩

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

lemma *fps-mult-commute-lemma*:
  fixes n :: *nat*
    and f :: *nat* ⇒ *nat* ⇒ *'a::comm-monoid-add*
  shows $(\sum i=0..n.\ f\ i\ (n - i)) = (\sum i=0..n.\ f\ (n - i)\ i)$
  ⟨*proof*⟩

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

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

instance *fps* :: (*semiring-1*) *semiring-1*
⟨*proof*⟩

instance *fps* :: (*comm-semiring-1*) *comm-semiring-1*
  ⟨*proof*⟩

instance *fps* :: (*semiring-1-cancel*) *semiring-1-cancel* ⟨*proof*⟩

lemma *fps-square-nth*: $(f\hat{}2)\ \$\ n = (\sum k\le n.\ f\ \$\ k * f\ \$\ (n - k))$
  ⟨*proof*⟩

lemma *fps-sum-nth*: *sum f S* $ n = *sum* (λk. (*f k*) $ *n*) *S*
⟨*proof*⟩

definition *fps-const c* = *Abs-fps* (λn. *if n* = *0 then c else 0*)

lemma *fps-nth-fps-const* [*simp*]: *fps-const c* $ *n* = (*if n* = *0 then c else 0*)
  ⟨*proof*⟩

lemma *fps-const-0-eq-0* [*simp*]: *fps-const 0* = *0*
  ⟨*proof*⟩

lemma *fps-const-nonzero-eq-nonzero*: $c \neq 0 \Longrightarrow$ *fps-const c* ≠ *0*
  ⟨*proof*⟩

lemma *fps-const-eq-0-iff* [*simp*]: *fps-const c* = *0* ⟷ *c* = *0*
⟨*proof*⟩

lemma *fps-const-1-eq-1* [*simp*]: *fps-const 1* = *1*
⟨*proof*⟩

lemma *fps-const-eq-1-iff* [*simp*]: *fps-const c* = *1* ⟷ *c* = *1*
⟨*proof*⟩

lemma *subdegree-fps-const* [*simp*]: *subdegree* (*fps-const c*) = *0*
⟨*proof*⟩

lemma *fps-const-neg* [*simp*]: − (*fps-const* (*c*::′*a*::*group-add*)) = *fps-const* (− *c*)
⟨*proof*⟩

lemma *fps-const-add* [*simp*]: *fps-const* (*c*::′*a*::*monoid-add*) + *fps-const d* = *fps-const* (*c* + *d*)
⟨*proof*⟩

lemma *fps-const-add-left*: *fps-const* (*c*::′*a*::*monoid-add*) + *f* =
Abs-fps (λ*n*. *if n* = *0 then c* + *f*\$*0 else f*\$*n*)
⟨*proof*⟩

lemma *fps-const-add-right*: *f* + *fps-const* (*c*::′*a*::*monoid-add*) =
Abs-fps (λ*n*. *if n* = *0 then f*\$*0* + *c else f*\$*n*)
⟨*proof*⟩

lemma *fps-const-sub* [*simp*]: *fps-const* (*c*::′*a*::*group-add*) − *fps-const d* = *fps-const* (*c* − *d*)
⟨*proof*⟩

lemmas *fps-const-minus* = *fps-const-sub*

lemma *fps-const-mult*[*simp*]:
fixes *c d* :: ′*a*::{*comm-monoid-add*,*mult-zero*}
shows *fps-const c* ∗ *fps-const d* = *fps-const* (*c* ∗ *d*)
⟨*proof*⟩

lemma *fps-const-mult-left*:
fps-const (*c*::′*a*::{*comm-monoid-add*,*mult-zero*}) ∗ *f* = *Abs-fps* (λ*n*. *c* ∗ *f*\$*n*)
⟨*proof*⟩

lemma *fps-const-mult-right*:
f ∗ *fps-const* (*c*::′*a*::{*comm-monoid-add*,*mult-zero*}) = *Abs-fps* (λ*n*. *f*\$*n* ∗ *c*)
⟨*proof*⟩

lemma *fps-mult-left-const-nth* [*simp*]:
(*fps-const* (*c*::′*a*::{*comm-monoid-add*,*mult-zero*}) ∗ *f*)\$*n* = *c*∗ *f*\$*n*
⟨*proof*⟩

lemma *fps-mult-right-const-nth* [*simp*]:
($f * $ *fps-const* (c::$'a$::{*comm-monoid-add*,*mult-zero*}))\$$n$ = f\$$n * c$
⟨*proof*⟩

lemma *fps-const-power* [*simp*]: *fps-const* c ^ n = *fps-const* (c^n)
⟨*proof*⟩

instance *fps* :: (*ring*) *ring* ⟨*proof*⟩

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

instance *fps* :: (*ring-1*) *ring-1* ⟨*proof*⟩

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

instance *fps* :: (*semiring-no-zero-divisors*) *semiring-no-zero-divisors*
⟨*proof*⟩

instance *fps* :: (*semiring-1-no-zero-divisors*) *semiring-1-no-zero-divisors* ⟨*proof*⟩

instance *fps* :: ({*cancel-semigroup-add*,*semiring-no-zero-divisors-cancel*})
semiring-no-zero-divisors-cancel
⟨*proof*⟩

instance *fps* :: (*ring-no-zero-divisors*) *ring-no-zero-divisors* ⟨*proof*⟩

instance *fps* :: (*ring-1-no-zero-divisors*) *ring-1-no-zero-divisors* ⟨*proof*⟩

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

lemma *fps-of-nat*: *fps-const* (*of-nat* c) = *of-nat* c
⟨*proof*⟩

lemma *fps-of-int*: *fps-const* (*of-int* c) = *of-int* c
⟨*proof*⟩

lemma *semiring-char-fps* [*simp*]: *CHAR*($'a$ :: *comm-semiring-1 fps*) = *CHAR*($'a$)
⟨*proof*⟩

instance *fps* :: ({*semiring-prime-char*,*comm-semiring-1*}) *semiring-prime-char*
⟨*proof*⟩
instance *fps* :: ({*comm-semiring-prime-char*,*comm-semiring-1*}) *comm-semiring-prime-char*
⟨*proof*⟩
instance *fps* :: ({*comm-ring-prime-char*,*comm-semiring-1*}) *comm-ring-prime-char*
⟨*proof*⟩
instance *fps* :: ({*idom-prime-char*,*comm-semiring-1*}) *idom-prime-char*
⟨*proof*⟩

lemma *fps-numeral-fps-const*: *numeral k* = *fps-const* (*numeral k*)
⟨*proof*⟩

lemmas *numeral-fps-const* = *fps-numeral-fps-const*

lemma *neg-numeral-fps-const*:
(− *numeral k* :: ′*a* :: *ring-1 fps*) = *fps-const* (− *numeral k*)
⟨*proof*⟩

lemma *fps-numeral-nth*: *numeral n* $ *i* = (*if i* = *0 then numeral n else 0*)
⟨*proof*⟩

lemma *fps-numeral-nth-0* [*simp*]: *numeral n* $ *0* = *numeral n*
⟨*proof*⟩

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

lemma *fps-nth-of-nat* [*simp*]:
(*of-nat c*) $ *n* = (*if n=0 then of-nat c else 0*)
⟨*proof*⟩

lemma *fps-nth-of-int* [*simp*]:
(*of-int c*) $ *n* = (*if n=0 then of-int c else 0*)
⟨*proof*⟩

lemma *fps-mult-of-nat-nth* [*simp*]:
shows (*of-nat k* ∗ *f*) $ *n* = *of-nat k* ∗ *f*$*n*
and (*f* ∗ *of-nat k*) $ *n* = *f*$*n* ∗ *of-nat k*
⟨*proof*⟩

lemma *fps-mult-of-int-nth* [*simp*]:
shows (*of-int k* ∗ *f*) $ *n* = *of-int k* ∗ *f*$*n*
and (*f* ∗ *of-int k*) $ *n* = *f*$*n* ∗ *of-int k*
⟨*proof*⟩

lemma *numeral-neq-fps-zero* [*simp*]: (*numeral f* :: ′*a* :: *field-char-0 fps*) ≠ *0*
⟨*proof*⟩

lemma *subdegree-power-ge*:
f^*n* ≠ *0* ⟹ *subdegree* (*f*^*n*) ≥ *n* ∗ *subdegree f*
⟨*proof*⟩

lemma *fps-pow-nth-below-subdegree*:
k < *n* ∗ *subdegree f* ⟹ (*f*^*n*) $ *k* = *0*
⟨*proof*⟩

lemma *fps-pow-base* [*simp*]:

$(f \,\hat{}\, n)$ \$ $(n * subdegree\ f) = (f$ \$ $subdegree\ f) \,\hat{}\, n$
⟨proof⟩

lemma *subdegree-power-eqI*:
fixes f :: $'a$::*semiring-1 fps*
shows $(f$ \$ $subdegree\ f) \,\hat{}\, n \neq 0 \Longrightarrow subdegree\ (f \,\hat{}\, n) = n * subdegree\ f$
⟨proof⟩

lemma *subdegree-power* [*simp*]:
subdegree $((f$ :: $('a$ :: *semiring-1-no-zero-divisors*) *fps*$) \,\hat{}\, n) = n * subdegree\ f$
⟨proof⟩

lemma *minus-one-power-iff*: $(-\ (1::'a::ring\text{-}1)) \,\hat{}\, n$ = (*if even n then 1 else* $-$ *1*)
⟨proof⟩

definition *fps-X* = *Abs-fps* (λn. *if* $n = 1$ *then 1 else 0*)

lemma *subdegree-fps-X* [*simp*]: *subdegree* (*fps-X* :: ($'a$ :: *zero-neq-one*) *fps*) = *1*
⟨proof⟩

lemma *fps-X-mult-nth* [*simp*]:
fixes f :: $'a$::{*comm-monoid-add*,*mult-zero*,*monoid-mult*} *fps*
shows (*fps-X* $*$ f) \$ n = (*if* $n = 0$ *then 0 else* f \$ $(n - 1)$)
⟨proof⟩

lemma *fps-X-mult-right-nth* [*simp*]:
fixes a :: $'a$::{*comm-monoid-add*,*mult-zero*,*monoid-mult*} *fps*
shows (a $*$ *fps-X*) \$ n = (*if* $n = 0$ *then 0 else* a \$ $(n - 1)$)
⟨proof⟩

lemma *fps-mult-fps-X-commute*:
fixes a :: $'a$::{*comm-monoid-add*,*mult-zero*,*monoid-mult*} *fps*
shows *fps-X* $*$ a = a $*$ *fps-X*
⟨proof⟩

lemma *fps-mult-fps-X-power-commute*: *fps-X* $\hat{}\, k * a = a *$ *fps-X* $\hat{}\, k$
⟨proof⟩

lemma *fps-subdegree-mult-fps-X*:
fixes f :: $'a$::{*comm-monoid-add*,*mult-zero*,*monoid-mult*} *fps*
assumes $f \neq 0$
shows *subdegree* (*fps-X* $*$ f) = *subdegree* $f + 1$
and *subdegree* (f $*$ *fps-X*) = *subdegree* $f + 1$
⟨proof⟩

lemma *fps-mult-fps-X-nonzero*:
fixes f :: $'a$::{*comm-monoid-add*,*mult-zero*,*monoid-mult*} *fps*
assumes $f \neq 0$

shows *fps-X* ∗ *f* ≠ *0*
and *f* ∗ *fps-X* ≠ *0*
⟨*proof*⟩

lemma *fps-mult-fps-X-power-nonzero*:
assumes *f* ≠ *0*
shows *fps-X* ^ *n* ∗ *f* ≠ *0*
and *f* ∗ *fps-X* ^ *n* ≠ *0*
⟨*proof*⟩

lemma *fps-X-power-iff*: *fps-X* ^ *n* = *Abs-fps* (*λm. if m* = *n then 1 else 0*)
⟨*proof*⟩

lemma *fps-X-nth*[*simp*]: *fps-X*$*n* = (*if n* = *1 then 1 else 0*)
⟨*proof*⟩

lemma *fps-X-power-nth*[*simp*]: (*fps-X*^*k*) $*n* = (*if n* = *k then 1 else 0*)
⟨*proof*⟩

lemma *fps-X-power-subdegree*: *subdegree* (*fps-X*^*n*) = *n*
⟨*proof*⟩

lemma *fps-X-power-mult-nth*:
(*fps-X*^*k* ∗ *f*) $ *n* = (*if n* < *k then 0 else f* $ (*n* − *k*))
⟨*proof*⟩

lemma *fps-X-power-mult-right-nth*:
(*f* ∗ *fps-X*^*k*) $ *n* = (*if n* < *k then 0 else f* $ (*n* − *k*))
⟨*proof*⟩

lemma *fps-subdegree-mult-fps-X-power*:
assumes *f* ≠ *0*
shows *subdegree* (*fps-X* ^ *n* ∗ *f*) = *subdegree f* + *n*
and *subdegree* (*f* ∗ *fps-X* ^ *n*) = *subdegree f* + *n*
⟨*proof*⟩

lemma *fps-mult-fps-X-plus-1-nth*:
((*1*+*fps-X*)∗*a*) $*n* = (*if n* = *0 then* (*a*$*n* :: *'a*::*semiring-1*) *else a*$*n* + *a*$(*n* − *1*))
⟨*proof*⟩

lemma *fps-mult-right-fps-X-plus-1-nth*:
fixes *a* :: *'a* :: *semiring-1 fps*
shows (*a*∗(*1*+*fps-X*)) $ *n* = (*if n* = *0 then a*$*n else a*$*n* + *a*$(*n* − *1*))
⟨*proof*⟩

lemma *fps-X-neq-fps-const* [*simp*]: (*fps-X* :: *'a* :: *zero-neq-one fps*) ≠ *fps-const c*
⟨*proof*⟩

lemma *fps-X-neq-zero* [*simp*]: (*fps-X* :: *'a* :: *zero-neq-one fps*) ≠ *0*
⟨*proof*⟩

lemma *fps-X-neq-one* [*simp*]: (*fps-X* :: *'a* :: *zero-neq-one fps*) ≠ *1*
⟨*proof*⟩

lemma *fps-X-neq-numeral* [*simp*]: *fps-X* ≠ *numeral c*
⟨*proof*⟩

lemma *fps-X-pow-eq-fps-X-pow-iff* [*simp*]: *fps-X* ^ *m* = *fps-X* ^ *n* ⟷ *m* = *n*
⟨*proof*⟩

5.4 Shifting and slicing

definition *fps-shift* :: *nat* ⇒ *'a fps* ⇒ *'a fps* **where**
fps-shift n f = *Abs-fps* (λ*i*. *f* $ (*i* + *n*))

lemma *fps-shift-nth* [*simp*]: *fps-shift n f* $ *i* = *f* $ (*i* + *n*)
⟨*proof*⟩

lemma *fps-shift-0* [*simp*]: *fps-shift 0 f* = *f*
⟨*proof*⟩

lemma *fps-shift-zero* [*simp*]: *fps-shift n 0* = *0*
⟨*proof*⟩

lemma *fps-shift-one*: *fps-shift n 1* = (*if n* = *0 then 1 else 0*)
⟨*proof*⟩

lemma *fps-shift-fps-const*: *fps-shift n* (*fps-const c*) = (*if n* = *0 then fps-const c else 0*)
⟨*proof*⟩

lemma *fps-shift-numeral*: *fps-shift n* (*numeral c*) = (*if n* = *0 then numeral c else 0*)
⟨*proof*⟩

lemma *fps-shift-fps-X* [*simp*]:
n ≥ *1* ⟹ *fps-shift n fps-X* = (*if n* = *1 then 1 else 0*)
⟨*proof*⟩

lemma *fps-shift-fps-X-power* [*simp*]:
n ≤ *m* ⟹ *fps-shift n* (*fps-X* ^ *m*) = *fps-X* ^ (*m* − *n*)
⟨*proof*⟩

lemma *fps-shift-subdegree* [*simp*]:
n ≤ *subdegree f* ⟹ *subdegree* (*fps-shift n f*) = *subdegree f* − *n*
⟨*proof*⟩

lemma *fps-shift-fps-shift*:
fps-shift $(m + n)$ f = *fps-shift* m (*fps-shift* n f)
⟨*proof*⟩

lemma *fps-shift-fps-shift-reorder*:
fps-shift m (*fps-shift* n f) = *fps-shift* n (*fps-shift* m f)
⟨*proof*⟩

lemma *fps-shift-rev-shift*:
$m \leq n \Longrightarrow$ *fps-shift* n (*Abs-fps* ($\lambda k.$ *if* $k<m$ *then* 0 *else* f \$ $(k-m)$)) = *fps-shift* $(n-m)$ f
$m > n \Longrightarrow$ *fps-shift* n (*Abs-fps* ($\lambda k.$ *if* $k<m$ *then* 0 *else* f \$ $(k-m)$)) =
Abs-fps ($\lambda k.$ *if* $k<m-n$ *then* 0 *else* f \$ $(k-(m-n))$)
⟨*proof*⟩

lemma *fps-shift-add*:
fps-shift n $(f + g)$ = *fps-shift* n f + *fps-shift* n g
⟨*proof*⟩

lemma *fps-shift-diff*:
fps-shift n $(f - g)$ = *fps-shift* n f − *fps-shift* n g
⟨*proof*⟩

lemma *fps-shift-uminus*:
fps-shift n $(-f)$ = − *fps-shift* n f
⟨*proof*⟩

lemma *fps-shift-mult*:
assumes $n \leq$ *subdegree* (g :: $'b$ :: {*comm-monoid-add*, *mult-zero*} *fps*)
shows *fps-shift* n $(h*g)$ = h ∗ *fps-shift* n g
⟨*proof*⟩

lemma *fps-shift-mult-right-noncomm*:
assumes $n \leq$ *subdegree* (g :: $'b$ :: {*comm-monoid-add*, *mult-zero*} *fps*)
shows *fps-shift* n $(g*h)$ = *fps-shift* n g ∗ h
⟨*proof*⟩

lemma *fps-shift-mult-right*:
assumes $n \leq$ *subdegree* (g :: $'b$ :: *comm-semiring-0 fps*)
shows *fps-shift* n $(g*h)$ = h ∗ *fps-shift* n g
⟨*proof*⟩

lemma *fps-shift-mult-both*:
fixes f g :: $'a$::{*comm-monoid-add*, *mult-zero*} *fps*
assumes $m \leq$ *subdegree* f $n \leq$ *subdegree* g
shows *fps-shift* m f ∗ *fps-shift* n g = *fps-shift* $(m+n)$ $(f*g)$
⟨*proof*⟩

lemma *fps-shift-subdegree-zero-iff* [*simp*]:

fps-shift (*subdegree f*) *f* = *0* ⟷ *f* = *0*
⟨*proof*⟩

lemma *fps-shift-times-fps-X*:
fixes *f g* :: *'a*::{*comm-monoid-add*,*mult-zero*,*monoid-mult*} *fps*
shows *1* ≤ *subdegree f* ⟹ *fps-shift 1 f* ∗ *fps-X* = *f*
⟨*proof*⟩

lemma *fps-shift-times-fps-X'* [*simp*]:
fixes *f* :: *'a*::{*comm-monoid-add*,*mult-zero*,*monoid-mult*} *fps*
shows *fps-shift 1* (*f* ∗ *fps-X*) = *f*
⟨*proof*⟩

lemma *fps-shift-times-fps-X''*:
fixes *f* :: *'a*::{*comm-monoid-add*,*mult-zero*,*monoid-mult*} *fps*
shows *1* ≤ *n* ⟹ *fps-shift n* (*f* ∗ *fps-X*) = *fps-shift* (*n* − *1*) *f*
⟨*proof*⟩

lemma *fps-shift-times-fps-X-power*:
n ≤ *subdegree f* ⟹ *fps-shift n f* ∗ *fps-X* ^ *n* = *f*
⟨*proof*⟩

lemma *fps-shift-times-fps-X-power'* [*simp*]:
fps-shift n (*f* ∗ *fps-X*^*n*) = *f*
⟨*proof*⟩

lemma *fps-shift-times-fps-X-power''*:
m ≤ *n* ⟹ *fps-shift n* (*f* ∗ *fps-X*^*m*) = *fps-shift* (*n* − *m*) *f*
⟨*proof*⟩

lemma *fps-shift-times-fps-X-power'''*:
m > *n* ⟹ *fps-shift n* (*f* ∗ *fps-X*^*m*) = *f* ∗ *fps-X*^(*m* − *n*)
⟨*proof*⟩

lemma *subdegree-decompose*:
f = *fps-shift* (*subdegree f*) *f* ∗ *fps-X* ^ *subdegree f*
⟨*proof*⟩

lemma *subdegree-decompose'*:
n ≤ *subdegree f* ⟹ *f* = *fps-shift n f* ∗ *fps-X*^*n*
⟨*proof*⟩

instantiation *fps* :: (*zero*) *unit-factor*
begin
definition *fps-unit-factor-def* [*simp*]:
unit-factor f = *fps-shift* (*subdegree f*) *f*
instance ⟨*proof*⟩
end

lemma *fps-unit-factor-zero-iff*: *unit-factor* $(f::'a::zero\ fps) = 0 \longleftrightarrow f = 0$
⟨*proof*⟩

lemma *fps-unit-factor-nth-0*: $f \neq 0 \Longrightarrow$ *unit-factor* $f\ \$\ 0 \neq 0$
⟨*proof*⟩

lemma *fps-X-unit-factor*: *unit-factor* (*fps-X* :: *'a* :: *zero-neq-one fps*) = *1*
⟨*proof*⟩

lemma *fps-X-power-unit-factor*: *unit-factor* (*fps-X* ^ *n*) = *1*
⟨*proof*⟩

lemma *fps-unit-factor-decompose*:
f = *unit-factor f* ∗ *fps-X* ^ *subdegree f*
⟨*proof*⟩

lemma *fps-unit-factor-decompose'*:
f = *fps-X* ^ *subdegree f* ∗ *unit-factor f*
⟨*proof*⟩

lemma *fps-unit-factor-uminus*:
unit-factor (−*f*) = − *unit-factor* (*f*::*'a*::*group-add fps*)
⟨*proof*⟩

lemma *fps-unit-factor-shift*:
assumes $n \leq$ *subdegree f*
shows *unit-factor* (*fps-shift n f*) = *unit-factor f*
⟨*proof*⟩

lemma *fps-unit-factor-mult-fps-X*:
fixes *f* :: *'a*::{*comm-monoid-add*,*monoid-mult*,*mult-zero*} *fps*
shows *unit-factor* (*fps-X* ∗ *f*) = *unit-factor f*
and *unit-factor* (*f* ∗ *fps-X*) = *unit-factor f*
⟨*proof*⟩

lemma *fps-unit-factor-mult-fps-X-power*:
shows *unit-factor* (*fps-X* ^ *n* ∗ *f*) = *unit-factor f*
and *unit-factor* (*f* ∗ *fps-X* ^ *n*) = *unit-factor f*
⟨*proof*⟩

lemma *fps-unit-factor-mult-unit-factor*:
fixes *f g* :: *'a*::{*comm-monoid-add*,*mult-zero*} *fps*
shows *unit-factor* (*f* ∗ *unit-factor g*) = *unit-factor* (*f* ∗ *g*)
and *unit-factor* (*unit-factor f* ∗ *g*) = *unit-factor* (*f* ∗ *g*)
⟨*proof*⟩

lemma *fps-unit-factor-mult-both-unit-factor*:
fixes *f g* :: *'a*::{*comm-monoid-add*,*mult-zero*} *fps*
shows *unit-factor* (*unit-factor f* ∗ *unit-factor g*) = *unit-factor* (*f* ∗ *g*)

⟨*proof*⟩

lemma *fps-unit-factor-mult′*:
fixes *f g* :: *′a*::{*comm-monoid-add*,*mult-zero*} *fps*
assumes *f* \$ *subdegree f* ∗ *g* \$ *subdegree g* ≠ *0*
shows *unit-factor* (*f* ∗ *g*) = *unit-factor f* ∗ *unit-factor g*
⟨*proof*⟩

lemma *fps-unit-factor-mult*:
fixes *f g* :: *′a*::*semiring-no-zero-divisors fps*
shows *unit-factor* (*f* ∗ *g*) = *unit-factor f* ∗ *unit-factor g*
⟨*proof*⟩

definition *fps-cutoff n f* = *Abs-fps* (*λi. if i* < *n then f*\$*i else 0*)

lemma *fps-cutoff-nth* [*simp*]: *fps-cutoff n f* \$ *i* = (*if i* < *n then f*\$*i else 0*)
⟨*proof*⟩

lemma *fps-cutoff-zero-iff*: *fps-cutoff n f* = *0* ⟷ (*f* = *0* ∨ *n* ≤ *subdegree f*)
⟨*proof*⟩

lemma *fps-cutoff-0* [*simp*]: *fps-cutoff 0 f* = *0*
⟨*proof*⟩

lemma *fps-cutoff-zero* [*simp*]: *fps-cutoff n 0* = *0*
⟨*proof*⟩

lemma *fps-cutoff-one*: *fps-cutoff n 1* = (*if n* = *0 then 0 else 1*)
⟨*proof*⟩

lemma *fps-cutoff-fps-const*: *fps-cutoff n* (*fps-const c*) = (*if n* = *0 then 0 else fps-const c*)
⟨*proof*⟩

lemma *fps-cutoff-numeral*: *fps-cutoff n* (*numeral c*) = (*if n* = *0 then 0 else numeral c*)
⟨*proof*⟩

lemma *fps-shift-cutoff*:
fps-shift n f ∗ *fps-X*^*n* + *fps-cutoff n f* = *f*
⟨*proof*⟩

lemma *fps-shift-cutoff′*:
fps-X^*n* ∗ *fps-shift n f* + *fps-cutoff n f* = *f*
⟨*proof*⟩

lemma *fps-cutoff-left-mult-nth*:
k < *n* ⟹ (*fps-cutoff n f* ∗ *g*) \$ *k* = (*f* ∗ *g*) \$ *k*
⟨*proof*⟩

lemma *fps-cutoff-right-mult-nth*:
assumes $k < n$
shows $(f * \textit{fps-cutoff } n\ g)\ \$\ k = (f * g)\ \$\ k$
⟨*proof*⟩

5.5 Metrizability

instantiation *fps* :: ({*minus*,*zero*}) *dist*
begin

definition
dist-fps-def: *dist* (*a* :: *'a fps*) *b* = (*if a* = *b then 0 else inverse* (*2* ^ *subdegree* (*a* − *b*)))

lemma *dist-fps-ge0*: *dist* (*a* :: *'a fps*) *b* ≥ *0*
⟨*proof*⟩

instance ⟨*proof*⟩

end

instantiation *fps* :: (*group-add*) *metric-space*
begin

definition *uniformity-fps-def* [*code del*]:
(*uniformity* :: (*'a fps* × *'a fps*) *filter*) = (*INF e*∈{*0* <..}. *principal* {(*x*, *y*). *dist x y* < *e*})

definition *open-fps-def'* [*code del*]:
open (*U* :: *'a fps set*) ⟷ (∀ *x*∈*U*. *eventually* (λ(*x'*, *y*). *x'* = *x* ⟶ *y* ∈ *U*) *uniformity*)

lemma *dist-fps-sym*: *dist* (*a* :: *'a fps*) *b* = *dist b a*
⟨*proof*⟩

instance
⟨*proof*⟩

end

declare *uniformity-Abort*[**where** *'a*=*'a* :: *group-add fps*, *code*]

lemma *open-fps-def*: *open* (*S* :: *'a*::*group-add fps set*) = (∀ *a* ∈ *S*. ∃ *r*. *r* >*0* ∧ {*y*. *dist y a* < *r*} ⊆ *S*)
⟨*proof*⟩

The infinite sums and justification of the notation in textbooks.

lemma *reals-power-lt-ex*:

fixes $x\ y$ :: *real*
assumes *xp*: $x > 0$
and *y1*: $y > 1$
shows $\exists k{>}0.\ (1/y)\hat{}k < x$
⟨*proof*⟩

lemma *fps-sum-rep-nth*: (*sum* (λ*i*. *fps-const*(*a*$*i*)∗*fps-X*^*i*) {*0*..*m*})$*n* = (*if* *n* ≤ *m* *then* *a*$*n* *else* *0*)
⟨*proof*⟩

lemma *fps-notation*: (λ*n*. *sum* (λ*i*. *fps-const*(*a*$*i*) ∗ *fps-X*^*i*) {*0*..*n*}) ⟶ *a*
(**is** *?s* ⟶ *a*)
⟨*proof*⟩

5.6 Division

declare *sum.cong*[*fundef-cong*]

fun *fps-left-inverse-constructor* ::
′*a*::{*comm-monoid-add*,*times*,*uminus*} *fps* ⇒ ′*a* ⇒ *nat* ⇒ ′*a*
where
fps-left-inverse-constructor f a 0 = *a*
| *fps-left-inverse-constructor f a* (*Suc n*) =
− *sum* (λ*i*. *fps-left-inverse-constructor f a i* ∗ *f*$(*Suc n* − *i*)) {*0*..*n*} ∗ *a*

— This will construct a left inverse for f in case that *x* ∗ *f* $ *0* = (*1*::′*b*)
abbreviation *fps-left-inverse* ≡ (λ*f x*. *Abs-fps* (*fps-left-inverse-constructor f x*))

fun *fps-right-inverse-constructor* ::
′*a*::{*comm-monoid-add*,*times*,*uminus*} *fps* ⇒ ′*a* ⇒ *nat* ⇒ ′*a*
where
fps-right-inverse-constructor f a 0 = *a*
| *fps-right-inverse-constructor f a n* =
− *a* ∗ *sum* (λ*i*. *f*$*i* ∗ *fps-right-inverse-constructor f a* (*n* − *i*)) {*1*..*n*}

— This will construct a right inverse for f in case that *f* $ *0* ∗ *y* = (*1*::′*b*)
abbreviation *fps-right-inverse* ≡ (λ*f y*. *Abs-fps* (*fps-right-inverse-constructor f y*))

instantiation *fps* :: ({*comm-monoid-add*,*inverse*,*times*,*uminus*}) *inverse*
begin

— For backwards compatibility.
abbreviation *natfun-inverse*:: ′*a fps* ⇒ *nat* ⇒ ′*a*
where *natfun-inverse f* ≡ *fps-right-inverse-constructor f* (*inverse* (*f*$*0*))

definition *fps-inverse-def*: *inverse f* = *Abs-fps* (*natfun-inverse f*)
— With scalars from a (possibly non-commutative) ring, this defines a right inverse. Furthermore, if scalars are of class *mult-zero* and satisfy condition *inverse* (*0*::′*b*) = (*0*::′*b*), then this will evaluate to zero when the zeroth term is zero.

definition *fps-divide-def*: *f div g = fps-shift* (*subdegree g*) (*f* ∗ *inverse* (*unit-factor g*))
— If scalars are of class *mult-zero* and satisfy condition *inverse* (*0*::′*b*) = (*0*::′*b*), then div by zero will equal zero.

instance ⟨*proof*⟩

end

lemma *fps-lr-inverse-0-iff*:
(*fps-left-inverse f x*) $ *0* = *0* ⟷ *x* = *0*
(*fps-right-inverse f x*) $ *0* = *0* ⟷ *x* = *0*
⟨*proof*⟩

lemma *fps-inverse-0-iff′*: (*inverse f*) $ *0* = *0* ⟷ *inverse* (*f* $ *0*) = *0*
⟨*proof*⟩

lemma *fps-inverse-0-iff* [*simp*]: (*inverse f*) $ *0* = (*0*::′*a*::*division-ring*) ⟷ *f* $ *0* = *0*
⟨*proof*⟩

lemma *fps-lr-inverse-nth-0*:
(*fps-left-inverse f x*) $ *0* = *x* (*fps-right-inverse f x*) $ *0* = *x*
⟨*proof*⟩

lemma *fps-inverse-nth-0* [*simp*]: (*inverse f*) $ *0* = *inverse* (*f* $ *0*)
⟨*proof*⟩

lemma *fps-lr-inverse-starting0*:
fixes *f* :: ′*a*::{*comm-monoid-add*,*mult-zero*,*uminus*} *fps*
and *g* :: ′*b*::{*ab-group-add*,*mult-zero*} *fps*
shows *fps-left-inverse f 0* = *0*
and *fps-right-inverse g 0* = *0*
⟨*proof*⟩

lemma *fps-lr-inverse-eq0-imp-starting0*:
fps-left-inverse f x = *0* ⟹ *x* = *0*
fps-right-inverse f x = *0* ⟹ *x* = *0*
⟨*proof*⟩

lemma *fps-lr-inverse-eq-0-iff*:
fixes *x* :: ′*a*::{*comm-monoid-add*,*mult-zero*,*uminus*}
and *y* :: ′*b*::{*ab-group-add*,*mult-zero*}
shows *fps-left-inverse f x* = *0* ⟷ *x* = *0*
and *fps-right-inverse g y* = *0* ⟷ *y* = *0*
⟨*proof*⟩

lemma *fps-inverse-eq-0-iff′*:

fixes *f* :: *'a*::{*ab-group-add*,*inverse*,*mult-zero*} *fps*
shows *inverse f* = *0* ⟷ *inverse* (*f* $ *0*) = *0*
⟨*proof*⟩

lemma *fps-inverse-eq-0-iff*[*simp*]: *inverse f* = (*0*:: (*'a*::*division-ring*) *fps*) ⟷ *f* $ *0* = *0*
⟨*proof*⟩

lemmas *fps-inverse-eq-0'* = *iffD2*[*OF fps-inverse-eq-0-iff'*]
lemmas *fps-inverse-eq-0* = *iffD2*[*OF fps-inverse-eq-0-iff*]

lemma *fps-const-lr-inverse*:
fixes *a* :: *'a*::{*ab-group-add*,*mult-zero*}
and *b* :: *'b*::{*comm-monoid-add*,*mult-zero*,*uminus*}
shows *fps-left-inverse* (*fps-const a*) *x* = *fps-const x*
and *fps-right-inverse* (*fps-const b*) *y* = *fps-const y*
⟨*proof*⟩

lemma *fps-const-inverse*:
fixes *a* :: *'a*::{*comm-monoid-add*,*inverse*,*mult-zero*,*uminus*}
shows *inverse* (*fps-const a*) = *fps-const* (*inverse a*)
⟨*proof*⟩

lemma *fps-lr-inverse-zero*:
fixes *x* :: *'a*::{*ab-group-add*,*mult-zero*}
and *y* :: *'b*::{*comm-monoid-add*,*mult-zero*,*uminus*}
shows *fps-left-inverse 0 x* = *fps-const x*
and *fps-right-inverse 0 y* = *fps-const y*
⟨*proof*⟩

lemma *fps-inverse-zero-conv-fps-const*:
inverse (*0*::*'a*::{*comm-monoid-add*,*mult-zero*,*uminus*,*inverse*} *fps*) = *fps-const* (*inverse 0*)
⟨*proof*⟩

lemma *fps-inverse-zero'*:
assumes *inverse* (*0*::*'a*::{*comm-monoid-add*,*inverse*,*mult-zero*,*uminus*}) = *0*
shows *inverse* (*0*::*'a fps*) = *0*
⟨*proof*⟩

lemma *fps-inverse-zero* [*simp*]:
inverse (*0*::*'a*::*division-ring fps*) = *0*
⟨*proof*⟩

lemma *fps-lr-inverse-one*:
fixes *x* :: *'a*::{*ab-group-add*,*mult-zero*,*one*}
and *y* :: *'b*::{*comm-monoid-add*,*mult-zero*,*uminus*,*one*}
shows *fps-left-inverse 1 x* = *fps-const x*
and *fps-right-inverse 1 y* = *fps-const y*

⟨*proof*⟩

lemma *fps-lr-inverse-one-one*:
fps-left-inverse 1 1 = (*1*::′*a*::{*ab-group-add*,*mult-zero*,*one*} *fps*)
fps-right-inverse 1 1 = (*1*::′*b*::{*comm-monoid-add*,*mult-zero*,*uminus*,*one*} *fps*)
⟨*proof*⟩

lemma *fps-inverse-one′*:
assumes *inverse* (*1*::′*a*::{*comm-monoid-add*,*inverse*,*mult-zero*,*uminus*,*one*}) = *1*
shows *inverse* (*1* :: ′*a fps*) = *1*
⟨*proof*⟩

lemma *fps-inverse-one* [*simp*]: *inverse* (*1* :: ′*a* :: *division-ring fps*) = *1*
⟨*proof*⟩

lemma *fps-lr-inverse-minus*:
fixes *f* :: ′*a*::*ring-1 fps*
shows *fps-left-inverse* (−*f*) (−*x*) = − *fps-left-inverse f x*
and *fps-right-inverse* (−*f*) (−*x*) = − *fps-right-inverse f x*
⟨*proof*⟩

lemma *fps-inverse-minus* [*simp*]: *inverse* (−*f*) = −*inverse* (*f* :: ′*a* :: *division-ring fps*)
⟨*proof*⟩

lemma *fps-left-inverse*:
fixes *f* :: ′*a*::*ring-1 fps*
assumes *f0*: *x* ∗ *f*\$*0* = *1*
shows *fps-left-inverse f x* ∗ *f* = *1*
⟨*proof*⟩

lemma *fps-right-inverse*:
fixes *f* :: ′*a*::*ring-1 fps*
assumes *f0*: *f*\$*0* ∗ *y* = *1*
shows *f* ∗ *fps-right-inverse f y* = *1*
⟨*proof*⟩

It is possible in a ring for an element to have a left inverse but not a right inverse, or vice versa. But when an element has both, they must be the same.

lemma *fps-left-inverse-eq-fps-right-inverse*:
fixes *f* :: ′*a*::*ring-1 fps*
assumes *f0*: *x* ∗ *f*\$*0* = *1 f* \$ *0* ∗ *y* = *1*
— These assumptions imply that x equals y, but no need to assume that.
shows *fps-left-inverse f x* = *fps-right-inverse f y*
⟨*proof*⟩

lemma *fps-left-inverse-eq-fps-right-inverse-comm*:
fixes *f* :: ′*a*::*comm-ring-1 fps*

assumes *f0*: $x * f\$0 = 1$
shows *fps-left-inverse f x = fps-right-inverse f x*
⟨*proof*⟩

lemma *fps-left-inverse'*:
fixes *f* :: *'a::ring-1 fps*
assumes $x * f\$0 = 1$ $f\$0 * y = 1$
— These assumptions imply x equals y, but no need to assume that.
shows *fps-right-inverse f y* $* f = 1$
⟨*proof*⟩

lemma *fps-right-inverse'*:
fixes *f* :: *'a::ring-1 fps*
assumes $x * f\$0 = 1$ $f\$0 * y = 1$
— These assumptions imply x equals y, but no need to assume that.
shows $f *$ *fps-left-inverse f x* $= 1$
⟨*proof*⟩

lemma *inverse-mult-eq-1* [*intro*]:
assumes $f\$0 \neq$ (*0::'a::division-ring*)
shows *inverse f* $* f = 1$
⟨*proof*⟩

lemma *inverse-mult-eq-1'*:
assumes $f\$0 \neq$ (*0::'a::division-ring*)
shows $f *$ *inverse f* $= 1$
⟨*proof*⟩

lemma *fps-mult-left-inverse-unit-factor*:
fixes *f* :: *'a::ring-1 fps*
assumes $x * f$ \$ *subdegree f* $= 1$
shows *fps-left-inverse* (*unit-factor f*) $x * f$ = *fps-X* ^ *subdegree f*
⟨*proof*⟩

lemma *fps-mult-right-inverse-unit-factor*:
fixes *f* :: *'a::ring-1 fps*
assumes *f* \$ *subdegree f* $* y = 1$
shows $f *$ *fps-right-inverse* (*unit-factor f*) y = *fps-X* ^ *subdegree f*
⟨*proof*⟩

lemma *fps-mult-right-inverse-unit-factor-divring*:
(*f* :: *'a::division-ring fps*) $\neq 0 \Longrightarrow f *$ *inverse* (*unit-factor f*) = *fps-X* ^ *subdegree f*
⟨*proof*⟩

lemma *fps-left-inverse-idempotent-ring1*:
fixes *f* :: *'a::ring-1 fps*
assumes $x * f\$0 = 1$ $y * x = 1$
— These assumptions imply y equals $f\$0$, but no need to assume that.

shows *fps-left-inverse* (*fps-left-inverse f x*) *y* = *f*
⟨*proof*⟩

lemma *fps-left-inverse-idempotent-comm-ring1*:
fixes *f* :: *'a::comm-ring-1 fps*
assumes $x * f\$0 = 1$
shows *fps-left-inverse* (*fps-left-inverse f x*) ($f\$0$) = *f*
⟨*proof*⟩

lemma *fps-right-inverse-idempotent-ring1*:
fixes *f* :: *'a::ring-1 fps*
assumes $f\$0 * x = 1$ $x * y = 1$
— These assumptions imply *y* equals *f$0*, but no need to assume that.
shows *fps-right-inverse* (*fps-right-inverse f x*) *y* = *f*
⟨*proof*⟩

lemma *fps-right-inverse-idempotent-comm-ring1*:
fixes *f* :: *'a::comm-ring-1 fps*
assumes $f\$0 * x = 1$
shows *fps-right-inverse* (*fps-right-inverse f x*) ($f\$0$) = *f*
⟨*proof*⟩

lemma *fps-inverse-idempotent*[*intro*, *simp*]:
$f\$0 \neq (0::'a::division\text{-}ring) \Longrightarrow$ *inverse* (*inverse f*) = *f*
⟨*proof*⟩

lemma *fps-lr-inverse-unique-ring1*:
fixes *f g* :: *'a* :: *ring-1 fps*
assumes *fg*: $f * g = 1$ $g\$0 * f\$0 = 1$
shows *fps-left-inverse g* ($f\$0$) = *f*
and *fps-right-inverse f* ($g\$0$) = *g*
⟨*proof*⟩

lemma *fps-lr-inverse-unique-divring*:
fixes *f g* :: *'a* ::*division-ring fps*
assumes *fg*: $f * g = 1$
shows *fps-left-inverse g* ($f\$0$) = *f*
and *fps-right-inverse f* ($g\$0$) = *g*
⟨*proof*⟩

lemma *fps-inverse-unique*:
fixes *f g* :: *'a* :: *division-ring fps*
assumes *fg*: $f * g = 1$
shows *inverse f* = *g*
⟨*proof*⟩

lemma *inverse-fps-numeral*:
inverse (*numeral n* :: (*'a* :: *field-char-0*) *fps*) = *fps-const* (*inverse* (*numeral n*))
⟨*proof*⟩

lemma *inverse-fps-of-nat*:
  inverse (*of-nat n* :: *'a* :: {*semiring-1*,*times*,*uminus*,*inverse*} *fps*) =
    fps-const (*inverse* (*of-nat n*))
  ⟨*proof*⟩

lemma *fps-lr-inverse-mult-ring1*:
  fixes *f g* :: *'a*::*ring-1 fps*
  assumes *x*: *x* ∗ *f*\$*0* = *1 f*\$*0* ∗ *x* = *1*
  and *y*: *y* ∗ *g*\$*0* = *1 g*\$*0* ∗ *y* = *1*
  shows *fps-left-inverse* (*f* ∗ *g*) (*y*∗*x*) = *fps-left-inverse g y* ∗ *fps-left-inverse f x*
  and *fps-right-inverse* (*f* ∗ *g*) (*y*∗*x*) = *fps-right-inverse g y* ∗ *fps-right-inverse f x*
⟨*proof*⟩

lemma *fps-lr-inverse-mult-divring*:
  fixes *f g* :: *'a*::*division-ring fps*
  shows *fps-left-inverse* (*f* ∗ *g*) (*inverse* ((*f*∗*g*)\$*0*)) =
    fps-left-inverse g (*inverse* (*g*\$*0*)) ∗ *fps-left-inverse f* (*inverse* (*f*\$*0*))
  and *fps-right-inverse* (*f* ∗ *g*) (*inverse* ((*f*∗*g*)\$*0*)) =
    fps-right-inverse g (*inverse* (*g*\$*0*)) ∗ *fps-right-inverse f* (*inverse* (*f*\$*0*))
⟨*proof*⟩

lemma *fps-inverse-mult-divring*:
  inverse (*f* ∗ *g*) = *inverse g* ∗ *inverse* (*f* :: *'a*::*division-ring fps*)
  ⟨*proof*⟩

lemma *fps-inverse-mult*: *inverse* (*f* ∗ *g* :: *'a*::*field fps*) = *inverse f* ∗ *inverse g*
  ⟨*proof*⟩

lemma *fps-lr-inverse-gp-ring1*:
  fixes *ones ones-inv* :: *'a* :: *ring-1 fps*
  defines *ones* ≡ *Abs-fps* (λ*n*. *1*)
  and *ones-inv* ≡ *Abs-fps* (λ*n*. *if n*=*0 then 1 else if n*=*1 then* − *1 else 0*)
  shows *fps-left-inverse ones 1* = *ones-inv*
  and *fps-right-inverse ones 1* = *ones-inv*
⟨*proof*⟩

lemma *fps-lr-inverse-gp-ring1'*:
  fixes *ones* :: *'a* :: *ring-1 fps*
  defines *ones* ≡ *Abs-fps* (λ*n*. *1*)
  shows *fps-left-inverse ones 1* = *1* − *fps-X*
  and *fps-right-inverse ones 1* = *1* − *fps-X*
⟨*proof*⟩

lemma *fps-inverse-gp*:
  inverse (*Abs-fps*(λ*n*. (*1*::*'a*::*division-ring*))) =
    Abs-fps (λ*n*. *if n*= *0 then 1 else if n*=*1 then* − *1 else 0*)
  ⟨*proof*⟩

lemma *fps-inverse-gp'*: *inverse* (*Abs-fps* (*λn. 1::'a::division-ring*)) = *1* − *fps-X*
⟨*proof*⟩

lemma *fps-lr-inverse-one-minus-fps-X*:
fixes *ones* :: *'a* :: *ring-1 fps*
defines *ones* ≡ *Abs-fps* (*λn. 1*)
shows *fps-left-inverse* (*1* − *fps-X*) *1* = *ones*
and *fps-right-inverse* (*1* − *fps-X*) *1* = *ones*
⟨*proof*⟩

lemma *fps-inverse-one-minus-fps-X*:
fixes *ones* :: *'a* :: *division-ring fps*
defines *ones* ≡ *Abs-fps* (*λn. 1*)
shows *inverse* (*1* − *fps-X*) = *ones*
⟨*proof*⟩

lemma *fps-lr-one-over-one-minus-fps-X-squared*:
shows *fps-left-inverse* ((*1* − *fps-X*)^*2*) (*1::'a::ring-1*) = *Abs-fps* (*λn. of-nat* (*n+1*))
fps-right-inverse ((*1* − *fps-X*)^*2*) (*1::'a*) = *Abs-fps* (*λn. of-nat* (*n+1*))
⟨*proof*⟩

lemma *fps-one-over-one-minus-fps-X-squared'*:
assumes *inverse* (*1::'a::*{*ring-1,inverse*}) = *1*
shows *inverse* ((*1* − *fps-X*)^*2* :: *'a fps*) = *Abs-fps* (*λn. of-nat* (*n+1*))
⟨*proof*⟩

lemma *fps-one-over-one-minus-fps-X-squared*:
inverse ((*1* − *fps-X*)^*2* :: *'a* :: *division-ring fps*) = *Abs-fps* (*λn. of-nat* (*n+1*))
⟨*proof*⟩

lemma *fps-lr-inverse-fps-X-plus1*:
fps-left-inverse (*1* + *fps-X*) (*1::'a::ring-1*) = *Abs-fps* (*λn.* (−*1*)^*n*)
fps-right-inverse (*1* + *fps-X*) (*1::'a*) = *Abs-fps* (*λn.* (−*1*)^*n*)
⟨*proof*⟩

lemma *fps-inverse-fps-X-plus1'*:
assumes *inverse* (*1::'a::*{*ring-1,inverse*}) = *1*
shows *inverse* (*1* + *fps-X*) = *Abs-fps* (*λn.* (− (*1::'a*)) ^ *n*)
⟨*proof*⟩

lemma *fps-inverse-fps-X-plus1*:
inverse (*1* + *fps-X*) = *Abs-fps* (*λn.* (− (*1::'a::division-ring*)) ^ *n*)
⟨*proof*⟩

lemma *subdegree-lr-inverse*:
fixes *x* :: *'a::*{*comm-monoid-add,mult-zero,uminus*}
and *y* :: *'b::*{*ab-group-add,mult-zero*}

shows *subdegree* (*fps-left-inverse f x*) = *0*
and *subdegree* (*fps-right-inverse g y*) = *0*
⟨*proof*⟩

lemma *subdegree-inverse* [*simp*]:
fixes *f* :: *'a*::{*ab-group-add*,*inverse*,*mult-zero*} *fps*
shows *subdegree* (*inverse f*) = *0*
⟨*proof*⟩

lemma *fps-div-zero* [*simp*]:
0 div (*g* :: *'a* :: {*comm-monoid-add*,*inverse*,*mult-zero*,*uminus*} *fps*) = *0*
⟨*proof*⟩

lemma *fps-div-by-zero'*:
fixes *g* :: *'a*::{*comm-monoid-add*,*inverse*,*mult-zero*,*uminus*} *fps*
assumes *inverse* (*0*::*'a*) = *0*
shows *g div 0* = *0*
⟨*proof*⟩

lemma *fps-div-by-zero* [*simp*]: (*g*::*'a*::*division-ring fps*) *div 0* = *0*
⟨*proof*⟩

lemma *fps-divide-unit'*: *subdegree g* = *0* ⟹ *f div g* = *f* ∗ *inverse g*
⟨*proof*⟩

lemma *fps-divide-unit*: *g*$*0* ≠ *0* ⟹ *f div g* = *f* ∗ *inverse g*
⟨*proof*⟩

lemma *fps-divide-nth-0'*:
subdegree (*g*::*'a*::*division-ring fps*) = *0* ⟹ (*f div g*) $ *0* = *f* $ *0* / (*g* $ *0*)
⟨*proof*⟩

lemma *fps-divide-nth-0* [*simp*]:
g $ *0* ≠ *0* ⟹ (*f div g*) $ *0* = *f* $ *0* / (*g* $ *0* :: - :: *division-ring*)
⟨*proof*⟩

lemma *fps-divide-nth-below*:
fixes *f g* :: *'a*::{*comm-monoid-add*,*uminus*,*mult-zero*,*inverse*} *fps*
shows *n* < *subdegree f* − *subdegree g* ⟹ (*f div g*) $ *n* = *0*
⟨*proof*⟩

lemma *fps-divide-nth-base*:
fixes *f g* :: *'a*::*division-ring fps*
assumes *subdegree g* ≤ *subdegree f*
shows (*f div g*) $ (*subdegree f* − *subdegree g*) = *f* $ *subdegree f* ∗ *inverse* (*g* $ *subdegree g*)
⟨*proof*⟩

lemma *fps-divide-subdegree-ge*:

fixes *f g* :: *'a*::{*comm-monoid-add*,*uminus*,*mult-zero*,*inverse*} *fps*
assumes *f* / *g* ≠ *0*
shows *subdegree* (*f* / *g*) ≥ *subdegree f* − *subdegree g*
⟨*proof*⟩

lemma *fps-divide-subdegree*:
fixes *f g* :: *'a*::*division-ring fps*
assumes *f* ≠ *0 g* ≠ *0 subdegree g* ≤ *subdegree f*
shows *subdegree* (*f* / *g*) = *subdegree f* − *subdegree g*
⟨*proof*⟩

lemma *fps-divide-shift-numer*:
fixes *f g* :: *'a*::{*inverse*,*comm-monoid-add*,*uminus*,*mult-zero*} *fps*
assumes *n* ≤ *subdegree f*
shows *fps-shift n f* / *g* = *fps-shift n* (*f*/*g*)
⟨*proof*⟩

lemma *fps-divide-shift-denom*:
fixes *f g* :: *'a*::{*inverse*,*comm-monoid-add*,*uminus*,*mult-zero*} *fps*
assumes *n* ≤ *subdegree g subdegree g* ≤ *subdegree f*
shows *f* / *fps-shift n g* = *Abs-fps* (λ*k*. *if k*<*n then 0 else* (*f*/*g*) $ (*k*−*n*))
⟨*proof*⟩

lemma *fps-divide-unit-factor-numer*:
fixes *f g* :: *'a*::{*inverse*,*comm-monoid-add*,*uminus*,*mult-zero*} *fps*
shows *unit-factor f* / *g* = *fps-shift* (*subdegree f*) (*f*/*g*)
⟨*proof*⟩

lemma *fps-divide-unit-factor-denom*:
fixes *f g* :: *'a*::{*inverse*,*comm-monoid-add*,*uminus*,*mult-zero*} *fps*
assumes *subdegree g* ≤ *subdegree f*
shows
f / *unit-factor g* = *Abs-fps* (λ*k*. *if k*<*subdegree g then 0 else* (*f*/*g*) $ (*k*−*subdegree g*))
⟨*proof*⟩

lemma *fps-divide-unit-factor-both'*:
fixes *f g* :: *'a*::{*inverse*,*comm-monoid-add*,*uminus*,*mult-zero*} *fps*
assumes *subdegree g* ≤ *subdegree f*
shows *unit-factor f* / *unit-factor g* = *fps-shift* (*subdegree f* − *subdegree g*) (*f* / *g*)
⟨*proof*⟩

lemma *fps-divide-unit-factor-both*:
fixes *f g* :: *'a*::*division-ring fps*
assumes *subdegree g* ≤ *subdegree f*
shows *unit-factor f* / *unit-factor g* = *unit-factor* (*f* / *g*)
⟨*proof*⟩

lemma *fps-divide-self*:
$(f::'a::division\text{-}ring\ fps) \neq 0 \Longrightarrow f\ /\ f = 1$
⟨*proof*⟩

lemma *fps-divide-add*:
fixes $f\ g\ h :: 'a::\{semiring\text{-}0,inverse,uminus\}\ fps$
shows $(f + g)\ /\ h = f\ /\ h + g\ /\ h$
⟨*proof*⟩

lemma *fps-divide-diff*:
fixes $f\ g\ h :: 'a::\{ring,inverse\}\ fps$
shows $(f - g)\ /\ h = f\ /\ h - g\ /\ h$
⟨*proof*⟩

lemma *fps-divide-uminus*:
fixes $f\ g\ h :: 'a::\{ring,inverse\}\ fps$
shows $(- f)\ /\ g = - (f\ /\ g)$
⟨*proof*⟩

lemma *fps-divide-uminus'*:
fixes $f\ g\ h :: 'a::division\text{-}ring\ fps$
shows $f\ /\ (- g) = - (f\ /\ g)$
⟨*proof*⟩

lemma *fps-divide-times*:
fixes $f\ g\ h :: 'a::\{semiring\text{-}0,inverse,uminus\}\ fps$
assumes $subdegree\ h \leq subdegree\ g$
shows $(f * g)\ /\ h = f * (g\ /\ h)$
⟨*proof*⟩

lemma *fps-divide-times2*:
fixes $f\ g\ h :: 'a::\{comm\text{-}semiring\text{-}0,inverse,uminus\}\ fps$
assumes $subdegree\ h \leq subdegree\ f$
shows $(f * g)\ /\ h = (f\ /\ h) * g$
⟨*proof*⟩

lemma *fps-times-divide-eq*:
fixes $f\ g :: 'a::field\ fps$
assumes $g \neq 0$ **and** $subdegree\ f \geq subdegree\ g$
shows $f\ div\ g * g = f$
⟨*proof*⟩

lemma *fps-divide-times-eq*:
$(g :: 'a::division\text{-}ring\ fps) \neq 0 \Longrightarrow (f * g)\ div\ g = f$
⟨*proof*⟩

lemma *fps-divide-by-mult'*:
fixes $f\ g\ h :: 'a :: division\text{-}ring\ fps$
assumes $subdegree\ h \leq subdegree\ f$

shows f / $(g * h) = f$ / h / g
⟨*proof*⟩

lemma *fps-divide-by-mult*:
fixes *f g h* :: *'a* :: *field fps*
assumes *subdegree g* ≤ *subdegree f*
shows f / $(g * h) = f$ / g / h
⟨*proof*⟩

lemma *fps-divide-cancel*:
fixes *f g h* :: *'a* :: *division-ring fps*
shows $h \neq 0 \Longrightarrow (f * h)$ *div* $(g * h) = f$ *div* g
⟨*proof*⟩

lemma *fps-divide-1'*:
fixes *a* :: *'a*::{*comm-monoid-add*,*inverse*,*mult-zero*,*uminus*,*zero-neq-one*,*monoid-mult*} *fps*
assumes *inverse* (*1*::*'a*) = *1*
shows *a* / *1* = *a*
⟨*proof*⟩

lemma *fps-divide-1* [*simp*]: (*a* :: *'a*::*division-ring fps*) / *1* = *a*
⟨*proof*⟩

lemma *fps-divide-X'*:
fixes *f* :: *'a*::{*comm-monoid-add*,*inverse*,*mult-zero*,*uminus*,*zero-neq-one*,*monoid-mult*} *fps*
assumes *inverse* (*1*::*'a*) = *1*
shows *f* / *fps-X* = *fps-shift 1 f*
⟨*proof*⟩

lemma *fps-divide-X* [*simp*]: *a* / *fps-X* = *fps-shift 1* (*a*::*'a*::*division-ring fps*)
⟨*proof*⟩

lemma *fps-divide-X-power'*:
fixes *f* :: *'a*::{*semiring-1*,*inverse*,*uminus*} *fps*
assumes *inverse* (*1*::*'a*) = *1*
shows *f* / (*fps-X* ^ *n*) = *fps-shift n f*
⟨*proof*⟩

lemma *fps-divide-X-power* [*simp*]: *a* / (*fps-X* ^ *n*) = *fps-shift n* (*a*::*'a*::*division-ring fps*)
⟨*proof*⟩

lemma *fps-divide-shift-denom-conv-times-fps-X-power*:
fixes *f g* :: *'a*::{*semiring-1*,*inverse*,*uminus*} *fps*
assumes *n* ≤ *subdegree g subdegree g* ≤ *subdegree f*
shows *f* / *fps-shift n g* = *f* / *g* * *fps-X* ^ *n*
⟨*proof*⟩

lemma *fps-divide-unit-factor-denom-conv-times-fps-X-power*:
fixes *f g* :: *'a*::{*semiring-1*,*inverse*,*uminus*} *fps*
assumes *subdegree g* ≤ *subdegree f*
shows *f* / *unit-factor g* = *f* / *g* ∗ *fps-X* ^ *subdegree g*
⟨*proof*⟩

lemma *fps-shift-altdef'*:
fixes *f* :: *'a*::{*semiring-1*,*inverse*,*uminus*} *fps*
assumes *inverse* (*1*::*'a*) = *1*
shows *fps-shift n f* = *f div fps-X*^*n*
⟨*proof*⟩

lemma *fps-shift-altdef*:
fps-shift n f = (*f* :: *'a* :: *division-ring fps*) *div fps-X*^*n*
⟨*proof*⟩

lemma *fps-div-fps-X-power-nth'*:
fixes *f* :: *'a*::{*semiring-1*,*inverse*,*uminus*} *fps*
assumes *inverse* (*1*::*'a*) = *1*
shows (*f div fps-X*^*n*) $ *k* = *f* $ (*k* + *n*)
⟨*proof*⟩

lemma *fps-div-fps-X-power-nth*: ((*f* :: *'a* :: *division-ring fps*) *div fps-X*^*n*) $ *k* = *f* $ (*k* + *n*)
⟨*proof*⟩

lemma *fps-div-fps-X-nth'*:
fixes *f* :: *'a*::{*semiring-1*,*inverse*,*uminus*} *fps*
assumes *inverse* (*1*::*'a*) = *1*
shows (*f div fps-X*) $ *k* = *f* $ *Suc k*
⟨*proof*⟩

lemma *fps-div-fps-X-nth*: ((*f* :: *'a* :: *division-ring fps*) *div fps-X*) $ *k* = *f* $ *Suc k*
⟨*proof*⟩

lemma *divide-fps-const'*:
fixes *c* :: *'a* :: {*inverse*,*comm-monoid-add*,*uminus*,*mult-zero*}
shows *f* / *fps-const c* = *f* ∗ *fps-const* (*inverse c*)
⟨*proof*⟩

lemma *divide-fps-const* [*simp*]:
fixes *c* :: *'a* :: {*comm-semiring-0*,*inverse*,*uminus*}
shows *f* / *fps-const c* = *fps-const* (*inverse c*) ∗ *f*
⟨*proof*⟩

lemma *fps-const-divide*: *fps-const* (*x* :: - :: *division-ring*) / *fps-const y* = *fps-const* (*x* / *y*)
⟨*proof*⟩

lemma *fps-numeral-divide-divide*:
x / *numeral* b / *numeral* c = (x / *numeral* ($b * c$) :: *'a* :: *field fps*)
⟨*proof*⟩

lemma *fps-numeral-mult-divide*:
numeral $b * x$ / *numeral* c = (*numeral* b / *numeral* $c * x$ :: *'a* :: *field fps*)
⟨*proof*⟩

lemmas *fps-numeral-simps* =
fps-numeral-divide-divide fps-numeral-mult-divide inverse-fps-numeral neg-numeral-fps-const

lemma *fps-is-left-unit-iff-zeroth-is-left-unit*:
fixes f :: *'a* :: *ring-1 fps*
shows ($\exists g.\ 1 = f * g$) $\longleftrightarrow$ ($\exists k.\ 1 = f\$0 * k$)
⟨*proof*⟩

lemma *fps-is-right-unit-iff-zeroth-is-right-unit*:
fixes f :: *'a* :: *ring-1 fps*
shows ($\exists g.\ 1 = g * f$) $\longleftrightarrow$ ($\exists k.\ 1 = k * f\$0$)
⟨*proof*⟩

lemma *fps-is-unit-iff* [*simp*]: (f :: *'a* :: *field fps*) *dvd* $1 \longleftrightarrow f\ \$\ 0 \neq 0$
⟨*proof*⟩

lemma *subdegree-eq-0-left*:
fixes f :: *'a*::{*comm-monoid-add*,*zero-neq-one*,*mult-zero*} *fps*
assumes $\exists g.\ 1 = f * g$
shows *subdegree* $f = 0$
⟨*proof*⟩

lemma *subdegree-eq-0-right*:
fixes f :: *'a*::{*comm-monoid-add*,*zero-neq-one*,*mult-zero*} *fps*
assumes $\exists g.\ 1 = g * f$
shows *subdegree* $f = 0$
⟨*proof*⟩

lemma *subdegree-eq-0'* [*simp*]: (f :: *'a* :: *field fps*) *dvd* $1 \Longrightarrow$ *subdegree* $f = 0$
⟨*proof*⟩

lemma *fps-dvd1-left-trivial-unit-factor*:
fixes f :: *'a*::{*comm-monoid-add*, *zero-neq-one*, *mult-zero*} *fps*
assumes $\exists g.\ 1 = f * g$
shows *unit-factor* $f = f$
⟨*proof*⟩

lemma *fps-dvd1-right-trivial-unit-factor*:
fixes f :: *'a*::{*comm-monoid-add*, *zero-neq-one*, *mult-zero*} *fps*
assumes $\exists g.\ 1 = g * f$

shows *unit-factor f = f*
⟨*proof*⟩

lemma *fps-dvd1-trivial-unit-factor*:
(*f* :: *'a*::*comm-semiring-1 fps*) *dvd 1* ⟹ *unit-factor f = f*
⟨*proof*⟩

lemma *fps-unit-dvd-left*:
fixes *f* :: *'a* :: *division-ring fps*
assumes *f* $ *0* ≠ *0*
shows ∃ *g*. *1* = *f* ∗ *g*
⟨*proof*⟩

lemma *fps-unit-dvd-right*:
fixes *f* :: *'a* :: *division-ring fps*
assumes *f* $ *0* ≠ *0*
shows ∃ *g*. *1* = *g* ∗ *f*
⟨*proof*⟩

lemma *fps-unit-dvd* [*simp*]: (*f* $ *0* :: *'a* :: *field*) ≠ *0* ⟹ *f dvd g*
⟨*proof*⟩

lemma *dvd-left-imp-subdegree-le*:
fixes *f g* :: *'a*::{*comm-monoid-add*,*mult-zero*} *fps*
assumes ∃ *k*. *g* = *f* ∗ *k* *g* ≠ *0*
shows *subdegree f* ≤ *subdegree g*
⟨*proof*⟩

lemma *dvd-right-imp-subdegree-le*:
fixes *f g* :: *'a*::{*comm-monoid-add*,*mult-zero*} *fps*
assumes ∃ *k*. *g* = *k* ∗ *f* *g* ≠ *0*
shows *subdegree f* ≤ *subdegree g*
⟨*proof*⟩

lemma *dvd-imp-subdegree-le*:
f dvd g ⟹ *g* ≠ *0* ⟹ *subdegree f* ≤ *subdegree g*
⟨*proof*⟩

lemma *subdegree-le-imp-dvd-left-ring1*:
fixes *f g* :: *'a* :: *ring-1 fps*
assumes ∃ *y*. *f* $ *subdegree f* ∗ *y* = *1* *subdegree f* ≤ *subdegree g*
shows ∃ *k*. *g* = *f* ∗ *k*
⟨*proof*⟩

lemma *subdegree-le-imp-dvd-left-divring*:
fixes *f g* :: *'a* :: *division-ring fps*
assumes *f* ≠ *0* *subdegree f* ≤ *subdegree g*
shows ∃ *k*. *g* = *f* ∗ *k*
⟨*proof*⟩

lemma *subdegree-le-imp-dvd-right-ring1*:
fixes $f\ g$ :: $'a$ :: *ring-1 fps*
assumes $\exists x.\ x * f\ \$\ subdegree\ f = 1$ $subdegree\ f \leq subdegree\ g$
shows $\exists k.\ g = k * f$
⟨*proof*⟩

lemma *subdegree-le-imp-dvd-right-divring*:
fixes $f\ g$ :: $'a$ :: *division-ring fps*
assumes $f \neq 0$ $subdegree\ f \leq subdegree\ g$
shows $\exists k.\ g = k * f$
⟨*proof*⟩

lemma *fps-dvd-iff*:
assumes (f :: $'a$ :: *field fps*) $\neq 0$ $g \neq 0$
shows $f\ dvd\ g \longleftrightarrow subdegree\ f \leq subdegree\ g$
⟨*proof*⟩

lemma *subdegree-div'*:
fixes $p\ q$ :: $'a$::*division-ring fps*
assumes $\exists k.\ p = k * q$
shows $subdegree\ (p\ div\ q) = subdegree\ p - subdegree\ q$
⟨*proof*⟩

lemma *subdegree-div*:
fixes $p\ q$ :: $'a$ :: *field fps*
assumes $q\ dvd\ p$
shows $subdegree\ (p\ div\ q) = subdegree\ p - subdegree\ q$
⟨*proof*⟩

lemma *subdegree-div-unit'*:
fixes $p\ q$ :: $'a$ :: {*ab-group-add*,*mult-zero*,*inverse*} *fps*
assumes $q\ \$\ 0 \neq 0$ $p\ \$\ subdegree\ p * inverse\ (q\ \$\ 0) \neq 0$
shows $subdegree\ (p\ div\ q) = subdegree\ p$
⟨*proof*⟩

lemma *subdegree-div-unit''*:
fixes $p\ q$ :: $'a$ :: {*ring-no-zero-divisors*,*inverse*} *fps*
assumes $q\ \$\ 0 \neq 0$ $inverse\ (q\ \$\ 0) \neq 0$
shows $subdegree\ (p\ div\ q) = subdegree\ p$
⟨*proof*⟩

lemma *subdegree-div-unit*:
fixes $p\ q$ :: $'a$ :: *division-ring fps*
assumes $q\ \$\ 0 \neq 0$
shows $subdegree\ (p\ div\ q) = subdegree\ p$
⟨*proof*⟩

instantiation *fps* :: ({*comm-semiring-1*,*inverse*,*uminus*}) *modulo*

begin

definition *fps-mod-def*:
  f mod g = (if g = 0 then f else
    let h = unit-factor g in fps-cutoff (subdegree g) (f ∗ inverse h) ∗ h)

instance ⟨*proof*⟩

end

lemma *fps-mod-zero* [*simp*]:
  (*f*::*'a*::{*comm-semiring-1*,*inverse*,*uminus*} *fps*) *mod 0 = f*
  ⟨*proof*⟩

lemma *fps-mod-eq-zero*:
  assumes *g ≠ 0* **and** *subdegree f ≥ subdegree g*
  shows *f mod g = 0*
⟨*proof*⟩

lemma *fps-mod-unit* [*simp*]: *g$0 ≠ 0 ⟹ f mod g = 0*
  ⟨*proof*⟩

lemma *subdegree-mod*:
  assumes *subdegree* (*f*::*'a*::*field fps*) < *subdegree g*
  shows *subdegree* (*f mod g*) = *subdegree f*
⟨*proof*⟩

instance *fps* :: (*field*) *idom-modulo*
⟨*proof*⟩

instantiation *fps* :: (*field*) *normalization-semidom-multiplicative*
begin

definition *fps-normalize-def* [*simp*]:
  normalize f = (if f = 0 then 0 else fps-X ^ subdegree f)

instance ⟨*proof*⟩

end

5.7 Euclidean division

instantiation *fps* :: (*field*) *euclidean-ring-cancel*
begin

definition *fps-euclidean-size-def*:
  euclidean-size f = (if f = 0 then 0 else 2 ^ subdegree f)

instance ⟨*proof*⟩

end

instance *fps* :: (*field*) *normalization-euclidean-semiring* ⟨*proof*⟩

instantiation *fps* :: (*field*) *euclidean-ring-gcd*
begin
definition *fps-gcd-def*: (*gcd* :: *'a fps* ⇒ *-*) = *Euclidean-Algorithm.gcd*
definition *fps-lcm-def*: (*lcm* :: *'a fps* ⇒ *-*) = *Euclidean-Algorithm.lcm*
definition *fps-Gcd-def*: (*Gcd* :: *'a fps set* ⇒ *-*) = *Euclidean-Algorithm.Gcd*
definition *fps-Lcm-def*: (*Lcm* :: *'a fps set* ⇒ *-*) = *Euclidean-Algorithm.Lcm*
instance ⟨*proof*⟩
end

lemma *fps-gcd*:
  assumes [*simp*]: *f* ≠ *0* *g* ≠ *0*
  shows *gcd f g* = *fps-X* ^ *min* (*subdegree f*) (*subdegree g*)
⟨*proof*⟩

lemma *fps-gcd-altdef*: *gcd f g* =
  (*if f* = *0* ∧ *g* = *0 then 0 else*
   if f = *0 then fps-X* ^ *subdegree g else*
   if g = *0 then fps-X* ^ *subdegree f else*
    fps-X ^ *min* (*subdegree f*) (*subdegree g*))
  ⟨*proof*⟩

lemma *fps-lcm*:
  assumes [*simp*]: *f* ≠ *0* *g* ≠ *0*
  shows *lcm f g* = *fps-X* ^ *max* (*subdegree f*) (*subdegree g*)
⟨*proof*⟩

lemma *fps-lcm-altdef*: *lcm f g* =
  (*if f* = *0* ∨ *g* = *0 then 0 else fps-X* ^ *max* (*subdegree f*) (*subdegree g*))
  ⟨*proof*⟩

lemma *fps-Gcd*:
  assumes *A* − {*0*} ≠ {}
  shows *Gcd A* = *fps-X* ^ (*INF f*∈*A*−{*0*}. *subdegree f*)
⟨*proof*⟩

lemma *fps-Gcd-altdef*: *Gcd A* =
  (*if A* ⊆ {*0*} *then 0 else fps-X* ^ (*INF f*∈*A*−{*0*}. *subdegree f*))
  ⟨*proof*⟩

lemma *fps-Lcm*:
  assumes *A* ≠ {} *0* ∉ *A* *bdd-above* (*subdegree*‘*A*)
  shows *Lcm A* = *fps-X* ^ (*SUP f*∈*A*. *subdegree f*)
⟨*proof*⟩

lemma *fps-Lcm-altdef*:
Lcm A =
(*if 0* ∈ *A* ∨ ¬*bdd-above* (*subdegree*'*A*) *then 0 else*
if A = {} *then 1 else fps-X* ^ (*SUP f*∈*A*. *subdegree f*))
⟨*proof*⟩

5.8 Formal Derivatives

definition *fps-deriv f* = *Abs-fps* (λ*n*. *of-nat* (*n* + *1*) ∗ *f* $ (*n* + *1*))

lemma *fps-deriv-nth*[*simp*]: *fps-deriv f* $ *n* = *of-nat* (*n* + *1*) ∗ *f* $ (*n* + *1*)
⟨*proof*⟩

lemma *fps-0th-higher-deriv*:
(*fps-deriv* ^^ *n*) *f* $ *0* = *fact n* ∗ *f* $ *n*
⟨*proof*⟩

lemma *fps-deriv-mult*[*simp*]:
fps-deriv (*f* ∗ *g*) = *f* ∗ *fps-deriv g* + *fps-deriv f* ∗ *g*
⟨*proof*⟩

lemma *fps-deriv-fps-X*[*simp*]: *fps-deriv fps-X* = *1*
⟨*proof*⟩

lemma *fps-deriv-neg*[*simp*]:
fps-deriv (− (*f*:: ′*a*::*ring-1 fps*)) = − (*fps-deriv f*)
⟨*proof*⟩

lemma *fps-deriv-add*[*simp*]: *fps-deriv* (*f* + *g*) = *fps-deriv f* + *fps-deriv g*
⟨*proof*⟩

lemma *fps-deriv-sub*[*simp*]:
fps-deriv ((*f*:: ′*a*::*ring-1 fps*) − *g*) = *fps-deriv f* − *fps-deriv g*
⟨*proof*⟩

lemma *fps-deriv-const*[*simp*]: *fps-deriv* (*fps-const c*) = *0*
⟨*proof*⟩

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

lemma *fps-deriv-of-int* [*simp*]: *fps-deriv* (*of-int n*) = *0*
⟨*proof*⟩

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

lemma *fps-deriv-mult-const-left*[*simp*]:
fps-deriv (*fps-const c* ∗ *f*) = *fps-const c* ∗ *fps-deriv f*

⟨*proof*⟩

lemma *fps-deriv-linear*[*simp*]:
fps-deriv (*fps-const a* ∗ *f* + *fps-const b* ∗ *g*) =
fps-const a ∗ *fps-deriv f* + *fps-const b* ∗ *fps-deriv g*
⟨*proof*⟩

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

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

lemma *fps-deriv-mult-const-right*[*simp*]:
fps-deriv (*f* ∗ *fps-const c*) = *fps-deriv f* ∗ *fps-const c*
⟨*proof*⟩

lemma *fps-deriv-sum*:
fps-deriv (*sum f S*) = *sum* (λ*i*. *fps-deriv* (*f i*)) *S*
⟨*proof*⟩

lemma *fps-deriv-eq-0-iff* [*simp*]:
fps-deriv f = *0* ⟷ *f* = *fps-const* (*f*\$*0* :: ′*a*::{*semiring-no-zero-divisors*,*semiring-char-0*})
⟨*proof*⟩

lemma *fps-deriv-eq-iff*:
fixes *f g* :: ′*a*::{*ring-1-no-zero-divisors*,*semiring-char-0*} *fps*
shows *fps-deriv f* = *fps-deriv g* ⟷ (*f* = *fps-const*(*f*\$*0* − *g*\$*0*) + *g*)
⟨*proof*⟩

lemma *fps-deriv-eq-iff-ex*:
fixes *f g* :: ′*a*::{*ring-1-no-zero-divisors*,*semiring-char-0*} *fps*
shows (*fps-deriv f* = *fps-deriv g*) ⟷ (∃ *c*. *f* = *fps-const c* + *g*)
⟨*proof*⟩

fun *fps-nth-deriv* :: *nat* ⇒ ′*a*::*semiring-1 fps* ⇒ ′*a fps*
where
fps-nth-deriv 0 f = *f*
| *fps-nth-deriv* (*Suc n*) *f* = *fps-nth-deriv n* (*fps-deriv f*)

lemma *fps-nth-deriv-commute*: *fps-nth-deriv* (*Suc n*) *f* = *fps-deriv* (*fps-nth-deriv n f*)
⟨*proof*⟩

lemma *fps-nth-deriv-linear*[*simp*]:
fps-nth-deriv n (*fps-const a* ∗ *f* + *fps-const b* ∗ *g*) =
fps-const a ∗ *fps-nth-deriv n f* + *fps-const b* ∗ *fps-nth-deriv n g*
⟨*proof*⟩

lemma *fps-nth-deriv-neg*[*simp*]:
fps-nth-deriv n (− (*f* :: ′*a*::*ring-1 fps*)) = − (*fps-nth-deriv n f*)
⟨*proof*⟩

lemma *fps-nth-deriv-add*[*simp*]:
fps-nth-deriv n ((*f* :: ′*a*::*ring-1 fps*) + *g*) = *fps-nth-deriv n f* + *fps-nth-deriv n g*
⟨*proof*⟩

lemma *fps-nth-deriv-sub*[*simp*]:
fps-nth-deriv n ((*f* :: ′*a*::*ring-1 fps*) − *g*) = *fps-nth-deriv n f* − *fps-nth-deriv n g*
⟨*proof*⟩

lemma *fps-nth-deriv-0*[*simp*]: *fps-nth-deriv n 0* = *0*
⟨*proof*⟩

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

lemma *fps-nth-deriv-const*[*simp*]:
fps-nth-deriv n (*fps-const c*) = (*if n* = *0 then fps-const c else 0*)
⟨*proof*⟩

lemma *fps-nth-deriv-mult-const-left*[*simp*]:
fps-nth-deriv n (*fps-const c* ∗ *f*) = *fps-const c* ∗ *fps-nth-deriv n f*
⟨*proof*⟩

lemma *fps-nth-deriv-mult-const-right*[*simp*]:
fps-nth-deriv n (*f* ∗ *fps-const c*) = *fps-nth-deriv n f* ∗ *fps-const c*
⟨*proof*⟩

lemma *fps-nth-deriv-sum*:
fps-nth-deriv n (*sum f S*) = *sum* (λ*i*. *fps-nth-deriv n* (*f i* :: ′*a*::*ring-1 fps*)) *S*
⟨*proof*⟩

lemma *fps-deriv-maclauren-0*:
(*fps-nth-deriv k* (*f* :: ′*a*::*comm-semiring-1 fps*)) $ *0* = *of-nat* (*fact k*) ∗ *f* $ *k*
⟨*proof*⟩

lemma *fps-deriv-lr-inverse*:
fixes *x y* :: ′*a*::*ring-1*
assumes *x* ∗ *f*$*0* = *1* *f*$*0* ∗ *y* = *1*
— These assumptions imply *x* equals *y*, but no need to assume that.
shows *fps-deriv* (*fps-left-inverse f x*) =
− *fps-left-inverse f x* ∗ *fps-deriv f* ∗ *fps-left-inverse f x*
and *fps-deriv* (*fps-right-inverse f y*) =
− *fps-right-inverse f y* ∗ *fps-deriv f* ∗ *fps-right-inverse f y*
⟨*proof*⟩

lemma *fps-deriv-lr-inverse-comm*:
fixes $x :: 'a::comm\text{-}ring\text{-}1$
assumes $x * f\$0 = 1$
shows *fps-deriv* (*fps-left-inverse f x*) = − *fps-deriv f* ∗ (*fps-left-inverse f x*)^2
and *fps-deriv* (*fps-right-inverse f x*) = − *fps-deriv f* ∗ (*fps-right-inverse f x*)^2
⟨*proof*⟩

lemma *fps-inverse-deriv-divring*:
fixes $a :: 'a::division\text{-}ring\ fps$
assumes $a\$0 \neq 0$
shows *fps-deriv* (*inverse a*) = − *inverse a* ∗ *fps-deriv a* ∗ *inverse a*
⟨*proof*⟩

lemma *fps-inverse-deriv*:
fixes $a :: 'a::field\ fps$
assumes $a\$0 \neq 0$
shows *fps-deriv* (*inverse a*) = − *fps-deriv a* ∗ (*inverse a*)^2
⟨*proof*⟩

lemma *fps-inverse-deriv′*:
fixes $a :: 'a::field\ fps$
assumes *a0*: $a\ \$\ 0 \neq 0$
shows *fps-deriv* (*inverse a*) = − *fps-deriv a* / a^2
⟨*proof*⟩

lemma *fps-divide-deriv*:
assumes *b dvd* ($a :: 'a :: field\ fps$)
shows *fps-deriv* (*a* / *b*) = (*fps-deriv a* ∗ *b* − *a* ∗ *fps-deriv b*) / *b*^*2*
⟨*proof*⟩

lemma *fps-nth-deriv-fps-X*[*simp*]: *fps-nth-deriv n fps-X* = (*if n* = *0 then fps-X else if n*=*1 then 1 else 0*)
⟨*proof*⟩

5.9 Powers

lemma *fps-power-zeroth*: (*a*^*n*) $ *0* = (*a*$*0*)^*n*
⟨*proof*⟩

lemma *fps-power-zeroth-eq-one*: *a*$*0* = *1* ⟹ *a*^*n* $ *0* = *1*
⟨*proof*⟩

lemma *fps-power-first*:
fixes $a :: 'a::comm\text{-}semiring\text{-}1\ fps$
shows (*a*^*n*) $ *1* = *of-nat n* ∗ (*a*$*0*)^(*n*−*1*) ∗ *a*$*1*
⟨*proof*⟩

lemma *fps-power-first-eq*: *a* $ *0* = *1* ⟹ *a*^*n* $ *1* = *of-nat n* ∗ *a*$*1*

⟨*proof*⟩

lemma *fps-power-first-eq′*:
  assumes $a \$ 1 = 1$
  shows $a^n \$ 1 = of\text{-}nat\ n * (a\$0)^{(n-1)}$
⟨*proof*⟩

lemmas *startsby-one-power* = *fps-power-zeroth-eq-one*

lemma *startsby-zero-power*: $a \$ 0 = 0 \Longrightarrow n > 0 \Longrightarrow a^n \$0 = 0$
  ⟨*proof*⟩

lemma *startsby-power*: $a \$0 = v \Longrightarrow a^n \$0 = v^n$
  ⟨*proof*⟩

lemma *startsby-nonzero-power*:
  fixes a :: *′a::semiring-1-no-zero-divisors fps*
  shows $a \$ 0 \neq 0 \Longrightarrow a^n \$ 0 \neq 0$
  ⟨*proof*⟩

lemma *startsby-zero-power-iff* [*simp*]:
  $a^n \$0 = (0::'a::semiring\text{-}1\text{-}no\text{-}zero\text{-}divisors) \longleftrightarrow n \neq 0 \wedge a\$0 = 0$
⟨*proof*⟩

lemma *startsby-zero-power-prefix*:
  assumes *a0*: $a \$ 0 = 0$
  shows $\forall n < k.\ a ^ k \$ n = 0$
⟨*proof*⟩

lemma *startsby-zero-sum-depends*:
  assumes *a0*: $a \$0 = 0$
    and *kn*: $n \geq k$
  shows $sum\ (\lambda i.\ (a ^ i)\$k)\ \{0\ ..\ n\} = sum\ (\lambda i.\ (a ^ i)\$k)\ \{0\ ..\ k\}$
⟨*proof*⟩

lemma *startsby-zero-power-nth-same*:
  assumes *a0*: $a\$0 = 0$
  shows $a^n \$ n = (a\$1) ^ n$
⟨*proof*⟩

lemma *fps-lr-inverse-power*:
  fixes a :: *′a::ring-1 fps*
  assumes $x * a\$0 = 1$ $a\$0 * x = 1$
  shows $fps\text{-}left\text{-}inverse\ (a^n)\ (x^n) = fps\text{-}left\text{-}inverse\ a\ x ^ n$
  and $fps\text{-}right\text{-}inverse\ (a^n)\ (x^n) = fps\text{-}right\text{-}inverse\ a\ x ^ n$
⟨*proof*⟩

lemma *fps-inverse-power*:
  fixes a :: *′a::division-ring fps*

shows *inverse* (*a*^*n*) = *inverse a* ^ *n*
⟨*proof*⟩

lemma *fps-deriv-power′*:
fixes *a* :: ′*a*::*comm-semiring-1 fps*
shows *fps-deriv* (*a* ^ *n*) = (*of-nat n*) ∗ *fps-deriv a* ∗ *a* ^(*n* − *1*)
⟨*proof*⟩

lemma *fps-deriv-power*:
fixes *a* :: ′*a*::*comm-semiring-1 fps*
shows *fps-deriv* (*a* ^ *n*) = *fps-const* (*of-nat n*) ∗ *fps-deriv a* ∗ *a* ^(*n* − *1*)
⟨*proof*⟩

5.10 Integration

definition *fps-integral* :: ′*a*::{*semiring-1*,*inverse*} *fps* ⇒ ′*a* ⇒ ′*a fps*
where *fps-integral a a0* =
Abs-fps (λ*n*. *if n*=*0 then a0 else inverse* (*of-nat n*) ∗ *a*$(*n* − *1*))

abbreviation *fps-integral0 a* ≡ *fps-integral a 0*

lemma *fps-integral-nth-0-Suc* [*simp*]:
fixes *a* :: ′*a*::{*semiring-1*,*inverse*} *fps*
shows *fps-integral a a0* $ *0* = *a0*
and *fps-integral a a0* $ *Suc n* = *inverse* (*of-nat* (*Suc n*)) ∗ *a* $ *n*
⟨*proof*⟩

lemma *fps-integral-conv-plus-const*:
fps-integral a a0 = *fps-integral a 0* + *fps-const a0*
⟨*proof*⟩

lemma *fps-deriv-fps-integral*:
fixes *a* :: ′*a*::{*division-ring*,*ring-char-0*} *fps*
shows *fps-deriv* (*fps-integral a a0*) = *a*
⟨*proof*⟩

lemma *fps-integral0-deriv*:
fixes *a* :: ′*a*::{*division-ring*,*ring-char-0*} *fps*
shows *fps-integral0* (*fps-deriv a*) = *a* − *fps-const* (*a*$*0*)
⟨*proof*⟩

lemma *fps-integral-deriv*:
fixes *a* :: ′*a*::{*division-ring*,*ring-char-0*} *fps*
shows *fps-integral* (*fps-deriv a*) (*a*$*0*) = *a*
⟨*proof*⟩

lemma *fps-integral0-zero*:
fps-integral0 (*0*::′*a*::{*semiring-1*,*inverse*} *fps*) = *0*
⟨*proof*⟩

lemma *fps-integral0-fps-const'*:
fixes *c* :: *'a*::{*semiring-1*,*inverse*}
assumes *inverse* (*1*::*'a*) = *1*
shows *fps-integral0* (*fps-const c*) = *fps-const c* ∗ *fps-X*
⟨*proof*⟩

lemma *fps-integral0-fps-const*:
fixes *c* :: *'a*::*division-ring*
shows *fps-integral0* (*fps-const c*) = *fps-const c* ∗ *fps-X*
⟨*proof*⟩

lemma *fps-integral0-one'*:
assumes *inverse* (*1*::*'a*::{*semiring-1*,*inverse*}) = *1*
shows *fps-integral0* (*1*::*'a fps*) = *fps-X*
⟨*proof*⟩

lemma *fps-integral0-one*:
fps-integral0 (*1*::*'a*::*division-ring fps*) = *fps-X*
⟨*proof*⟩

lemma *fps-integral0-fps-const-mult-left*:
fixes *a* :: *'a*::*division-ring fps*
shows *fps-integral0* (*fps-const c* ∗ *a*) = *fps-const c* ∗ *fps-integral0 a*
⟨*proof*⟩

lemma *fps-integral0-fps-const-mult-right*:
fixes *a* :: *'a*::{*semiring-1*,*inverse*} *fps*
shows *fps-integral0* (*a* ∗ *fps-const c*) = *fps-integral0 a* ∗ *fps-const c*
⟨*proof*⟩

lemma *fps-integral0-neg*:
fixes *a* :: *'a*::{*ring-1*,*inverse*} *fps*
shows *fps-integral0* (−*a*) = − *fps-integral0 a*
⟨*proof*⟩

lemma *fps-integral0-add*:
fps-integral0 (*a*+*b*) = *fps-integral0 a* + *fps-integral0 b*
⟨*proof*⟩

lemma *fps-integral0-linear*:
fixes *a b* :: *'a*::*division-ring*
shows *fps-integral0* (*fps-const a* ∗ *f* + *fps-const b* ∗ *g*) =
fps-const a ∗ *fps-integral0 f* + *fps-const b* ∗ *fps-integral0 g*
⟨*proof*⟩

lemma *fps-integral0-linear2*:
fps-integral0 (*f* ∗ *fps-const a* + *g* ∗ *fps-const b*) =
fps-integral0 f ∗ *fps-const a* + *fps-integral0 g* ∗ *fps-const b*

⟨*proof*⟩

lemma *fps-integral-linear*:
fixes *a b a0 b0* :: *'a::division-ring*
shows
fps-integral (*fps-const a* ∗ *f* + *fps-const b* ∗ *g*) (*a*∗*a0* + *b*∗*b0*) =
fps-const a ∗ *fps-integral f a0* + *fps-const b* ∗ *fps-integral g b0*
⟨*proof*⟩

lemma *fps-integral0-sub*:
fixes *a b* :: *'a*::{*ring-1*,*inverse*} *fps*
shows *fps-integral0* (*a*−*b*) = *fps-integral0 a* − *fps-integral0 b*
⟨*proof*⟩

lemma *fps-integral0-of-nat*:
fps-integral0 (*of-nat n* :: *'a::division-ring fps*) = *of-nat n* ∗ *fps-X*
⟨*proof*⟩

lemma *fps-integral0-sum*:
fps-integral0 (*sum f S*) = *sum* (*λi. fps-integral0* (*f i*)) *S*
⟨*proof*⟩

lemma *fps-integral0-by-parts*:
fixes *a b* :: *'a*::{*division-ring*,*ring-char-0*} *fps*
shows
fps-integral0 (*a* ∗ *b*) =
a ∗ *fps-integral0 b* − *fps-integral0* (*fps-deriv a* ∗ *fps-integral0 b*)
⟨*proof*⟩

lemma *fps-integral0-fps-X*:
fps-integral0 (*fps-X*::*'a*::{*semiring-1*,*inverse*} *fps*) =
fps-const (*inverse* (*of-nat 2*)) ∗ $fps\text{-}X^2$
⟨*proof*⟩

lemma *fps-integral0-fps-X-power*:
fps-integral0 ((*fps-X*::*'a*::{*semiring-1*,*inverse*} *fps*) ^ *n*) =
fps-const (*inverse* (*of-nat* (*Suc n*))) ∗ *fps-X* ^ *Suc n*
⟨*proof*⟩

5.11 Composition

definition *fps-compose* :: *'a::semiring-1 fps* ⇒ *'a fps* ⇒ *'a fps* (**infixl** *oo 55*)
where *a oo b* = *Abs-fps* (*λn. sum* (*λi. a$i* ∗ (*b^i$n*)) {*0..n*})

lemma *fps-compose-nth*: (*a oo b*)$*n* = *sum* (*λi. a$i* ∗ (*b^i$n*)) {*0..n*}
⟨*proof*⟩

lemma *fps-compose-nth-0* [*simp*]: (*f oo g*) $ *0* = *f* $ *0*
⟨*proof*⟩

lemma *fps-compose-fps-X*[*simp*]: *a oo fps-X* = (*a* :: *'a::comm-ring-1 fps*)
⟨*proof*⟩

lemma *fps-const-compose*[*simp*]: *fps-const* (*a::'a::comm-ring-1*) *oo b* = *fps-const a*
⟨*proof*⟩

lemma *numeral-compose*[*simp*]: (*numeral k* :: *'a::comm-ring-1 fps*) *oo b* = *numeral k*
⟨*proof*⟩

lemma *neg-numeral-compose*[*simp*]: (− *numeral k* :: *'a::comm-ring-1 fps*) *oo b* = − *numeral k*
⟨*proof*⟩

lemma *fps-X-fps-compose-startby0*[*simp*]: *a$0* = *0* ⟹ *fps-X oo a* = (*a* :: *'a::comm-ring-1 fps*)
⟨*proof*⟩

5.12 Rules from Herbert Wilf's Generatingfunctionology

5.12.1 Rule 1

lemma *fps-power-mult-eq-shift*:
fps-X^Suc k ∗ *Abs-fps* (λ*n. a* (*n* + *Suc k*)) =
Abs-fps a − *sum* (λ*i. fps-const* (*a i* :: *'a::comm-ring-1*) ∗ *fps-X^i*) {*0* .. *k*}
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

5.12.2 Rule 2

definition *fps-XD* = (∗) *fps-X* ∘ *fps-deriv*

lemma *fps-XD-add*[*simp*]:*fps-XD* (*a* + *b*) = *fps-XD a* + *fps-XD* (*b* :: *'a::comm-ring-1 fps*)
⟨*proof*⟩

lemma *fps-XD-mult-const*[*simp*]:*fps-XD* (*fps-const* (*c::'a::comm-ring-1*) ∗ *a*) = *fps-const c* ∗ *fps-XD a*
⟨*proof*⟩

lemma *fps-XD-linear*[*simp*]: *fps-XD* (*fps-const c* ∗ *a* + *fps-const d* ∗ *b*) =
fps-const c ∗ *fps-XD a* + *fps-const d* ∗ *fps-XD* (*b* :: *'a::comm-ring-1 fps*)
⟨*proof*⟩

lemma *fps-XDN-linear*:
(*fps-XD* ^^ *n*) (*fps-const c* ∗ *a* + *fps-const d* ∗ *b*) =
fps-const c ∗ (*fps-XD* ^^ *n*) *a* + *fps-const d* ∗ (*fps-XD* ^^ *n*) (*b* :: *'a::comm-ring-1 fps*)
⟨*proof*⟩

lemma *fps-mult-fps-X-deriv-shift*: *fps-X*∗ *fps-deriv a* = *Abs-fps* (λ*n*. *of-nat n*∗ *a*$*n*)
⟨*proof*⟩

lemma *fps-mult-fps-XD-shift*:
(*fps-XD* ^^ *k*) (*a* :: ′*a*::*comm-ring-1 fps*) = *Abs-fps* (λ*n*. (*of-nat n* ^ *k*) ∗ *a*$*n*)
⟨*proof*⟩

5.12.3 Rule 3

Rule 3 is trivial and is given by `fps_times_def`.

5.12.4 Rule 5 — summation and "division" by $1 - X$

lemma *fps-divide-fps-X-minus1-sum-lemma*:
a = ((*1*::′*a*::*ring-1 fps*) − *fps-X*) ∗ *Abs-fps* (λ*n*. *sum* (λ*i*. *a* $ *i*) {*0*..*n*})
⟨*proof*⟩

lemma *fps-divide-fps-X-minus1-sum-ring1*:
assumes *inverse 1* = (*1*::′*a*::{*ring-1*,*inverse*})
shows *a* /((*1*::′*a fps*) − *fps-X*) = *Abs-fps* (λ*n*. *sum* (λ*i*. *a* $ *i*) {*0*..*n*})
⟨*proof*⟩

lemma *fps-divide-fps-X-minus1-sum*:
a /((*1*::′*a*::*division-ring fps*) − *fps-X*) = *Abs-fps* (λ*n*. *sum* (λ*i*. *a* $ *i*) {*0*..*n*})
⟨*proof*⟩

5.12.5 Rule 4 in its more general form

This generalizes Rule 3 for an arbitrary finite product of FPS, also the relevant instance of powers of a FPS.

definition *natpermute n k* = {*l* :: *nat list*. *length l* = *k* ∧ *sum-list l* = *n*}

lemma *natlist-trivial-1*: *natpermute n 1* = {[*n*]}
⟨*proof*⟩

lemma *natlist-trivial-Suc0* [*simp*]: *natpermute n* (*Suc 0*) = {[*n*]}
⟨*proof*⟩

lemma *append-natpermute-less-eq*:
assumes *xs* @ *ys* ∈ *natpermute n k*
shows *sum-list xs* ≤ *n*
and *sum-list ys* ≤ *n*
⟨*proof*⟩

lemma *natpermute-split*:
assumes *h* ≤ *k*
shows *natpermute n k* =

($\bigcup m \in\{0..n\}$. {*l1* @ *l2* |*l1 l2*. *l1* ∈ *natpermute m h* ∧ *l2* ∈ *natpermute* (*n* − *m*) (*k* − *h*)})
(**is** *?L* = *?R* **is** - = ($\bigcup m \in\{0..n\}$. *?S m*))
⟨*proof*⟩

lemma *natpermute-0*: *natpermute n 0* = (*if n* = *0 then* {[]} *else* {})
⟨*proof*⟩

lemma *natpermute-0'*[*simp*]: *natpermute 0 k* = (*if k* = *0 then* {[]} *else* {*replicate k 0*})
⟨*proof*⟩

lemma *natpermute-finite*: *finite* (*natpermute n k*)
⟨*proof*⟩

lemma *natpermute-contain-maximal*:
{*xs* ∈ *natpermute n* (*k* + *1*). *n* ∈ *set xs*} = ($\bigcup i\in\{0 .. k\}$. {(*replicate* (*k* + *1*) *0*) [*i*:=*n*]})
(**is** *?A* = *?B*)
⟨*proof*⟩

The general form.

lemma *fps-prod-nth*:
fixes *m* :: *nat*
and *a* :: *nat* ⇒ *'a*::*comm-ring-1 fps*
shows (*prod a* {*0* .. *m*}) $ *n* =
sum (λ*v*. *prod* (λ*j*. (*a j*) $ (*v*!*j*)) {*0*..*m*}) (*natpermute n* (*m*+*1*))
(**is** *?P m n*)
⟨*proof*⟩

The special form for powers.

lemma *fps-power-nth-Suc*:
fixes *m* :: *nat*
and *a* :: *'a*::*comm-ring-1 fps*
shows (*a* ^ *Suc m*)$*n* = *sum* (λ*v*. *prod* (λ*j*. *a* $ (*v*!*j*)) {*0*..*m*}) (*natpermute n* (*m*+*1*))
⟨*proof*⟩

lemma *fps-power-nth*:
fixes *m* :: *nat*
and *a* :: *'a*::*comm-ring-1 fps*
shows (*a* ^*m*)$*n* =
(*if m*=*0 then 1*$*n else sum* (λ*v*. *prod* (λ*j*. *a* $ (*v*!*j*)) {*0*..*m* − *1*}) (*natpermute n m*))
⟨*proof*⟩

lemmas *fps-nth-power-0* = *fps-power-zeroth*

lemma *natpermute-max-card*:

assumes *n0*: $n \neq 0$
shows *card* $\{xs \in natpermute\ n\ (k + 1).\ n \in set\ xs\} = k + 1$
⟨*proof*⟩

lemma *fps-power-Suc-nth*:
fixes *f* :: *'a* :: *comm-ring-1 fps*
assumes *k*: $k > 0$
shows $(f\ \widehat{}\ Suc\ m)\ \$\ k =$
$of\text{-}nat\ (Suc\ m) * (f\ \$\ k * (f\ \$\ 0)\ \widehat{}\ m) +$
$(\sum v\in\{v\in natpermute\ k\ (m+1).\ k \notin set\ v\}.\ \prod j = 0..m.\ f\ \$\ v\ !\ j)$
⟨*proof*⟩

lemma *fps-power-Suc-eqD*:
fixes *f g* :: *'a* :: {*idom*,*semiring-char-0*} *fps*
assumes $f\ \widehat{}\ Suc\ m = g\ \widehat{}\ Suc\ m\ f\ \$\ 0 = g\ \$\ 0\ f\ \$\ 0 \neq 0$
shows $f = g$
⟨*proof*⟩

lemma *fps-power-Suc-eqD'*:
fixes *f g* :: *'a* :: {*idom*,*semiring-char-0*} *fps*
assumes $f\ \widehat{}\ Suc\ m = g\ \widehat{}\ Suc\ m\ f\ \$\ subdegree\ f = g\ \$\ subdegree\ g$
shows $f = g$
⟨*proof*⟩

lemma *fps-power-eqD'*:
fixes *f g* :: *'a* :: {*idom*,*semiring-char-0*} *fps*
assumes $f\ \widehat{}\ m = g\ \widehat{}\ m\ f\ \$\ subdegree\ f = g\ \$\ subdegree\ g\ m > 0$
shows $f = g$
⟨*proof*⟩

lemma *fps-power-eqD*:
fixes *f g* :: *'a* :: {*idom*,*semiring-char-0*} *fps*
assumes $f\ \widehat{}\ m = g\ \widehat{}\ m\ f\ \$\ 0 = g\ \$\ 0\ f\ \$\ 0 \neq 0\ m > 0$
shows $f = g$
⟨*proof*⟩

lemma *fps-compose-inj-right*:
assumes *a0*: $a\$0 = (0::'a::idom)$
and *a1*: $a\$1 \neq 0$
shows $(b\ oo\ a = c\ oo\ a) \longleftrightarrow b = c$
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

5.13 Radicals

declare *prod.cong* [*fundef-cong*]

function *radical* :: $(nat \Rightarrow 'a \Rightarrow 'a) \Rightarrow nat \Rightarrow 'a::field\ fps \Rightarrow nat \Rightarrow 'a$
where

radical r 0 a 0 = 1
| *radical r 0 a (Suc n) = 0*
| *radical r (Suc k) a 0 = r (Suc k) (a$0)*
| *radical r (Suc k) a (Suc n) =*
  (a$ Suc n − sum (λxs. prod (λj. radical r (Suc k) a (xs ! j)) {0..k})
   {xs. xs ∈ natpermute (Suc n) (Suc k) ∧ Suc n ∉ set xs}) /
  (of-nat (Suc k) ∗ (radical r (Suc k) a 0)^k)
⟨*proof*⟩

termination *radical*
⟨*proof*⟩

definition *fps-radical r n a = Abs-fps (radical r n a)*

lemma *radical-0* [*simp*]: ⋀*n. 0 < n* ⟹ *radical r 0 a n = 0*
⟨*proof*⟩

lemma *fps-radical0*[*simp*]: *fps-radical r 0 a = 1*
⟨*proof*⟩

lemma *fps-radical-nth-0*[*simp*]: *fps-radical r n a $ 0 = (if n = 0 then 1 else r n (a$0))*
⟨*proof*⟩

lemma *fps-radical-power-nth*[*simp*]:
 assumes *r*: *(r k (a$0)) ^ k = a$0*
 shows *fps-radical r k a ^ k $ 0 = (if k = 0 then 1 else a$0)*
⟨*proof*⟩

lemma *power-radical*:
 fixes *a*:: *'a::field-char-0 fps*
 assumes *a0*: *a$0 ≠ 0*
 shows *(r (Suc k) (a$0)) ^ Suc k = a$0* ⟷ *(fps-radical r (Suc k) a) ^ (Suc k) = a*
  (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *radical-unique*:
 assumes *r0*: *(r (Suc k) (b$0)) ^ Suc k = b$0*
  and *a0*: *r (Suc k) (b$0 ::'a::field-char-0) = a$0*
  and *b0*: *b$0 ≠ 0*
 shows *a^(Suc k) = b* ⟷ *a = fps-radical r (Suc k) b*
  (**is** *?lhs* ⟷ *?rhs* **is** - ⟷ *a = ?r*)
⟨*proof*⟩

lemma *radical-power*:
 assumes *r0*: *r (Suc k) ((a$0) ^ Suc k) = a$0*
  and *a0*: *(a$0 :: 'a::field-char-0) ≠ 0*

shows (*fps-radical r* (*Suc k*) (*a* ^ *Suc k*)) = *a*
⟨*proof*⟩

lemma *fps-deriv-radical'*:
fixes *a* :: *'a::field-char-0 fps*
assumes *r0*: (*r* (*Suc k*) (*a$0*)) ^ *Suc k* = *a$0*
and *a0*: *a$0* ≠ *0*
shows *fps-deriv* (*fps-radical r* (*Suc k*) *a*) =
fps-deriv a / ((*of-nat* (*Suc k*)) ∗ (*fps-radical r* (*Suc k*) *a*) ^ *k*)
⟨*proof*⟩

lemma *fps-deriv-radical*:
fixes *a* :: *'a::field-char-0 fps*
assumes *r0*: (*r* (*Suc k*) (*a$0*)) ^ *Suc k* = *a$0*
and *a0*: *a$0* ≠ *0*
shows *fps-deriv* (*fps-radical r* (*Suc k*) *a*) =
fps-deriv a / (*fps-const* (*of-nat* (*Suc k*)) ∗ (*fps-radical r* (*Suc k*) *a*) ^ *k*)
⟨*proof*⟩

lemma *radical-mult-distrib*:
fixes *a* :: *'a::field-char-0 fps*
assumes *k*: *k* > *0*
and *ra0*: *r k* (*a* $ *0*) ^ *k* = *a* $ *0*
and *rb0*: *r k* (*b* $ *0*) ^ *k* = *b* $ *0*
and *a0*: *a* $ *0* ≠ *0*
and *b0*: *b* $ *0* ≠ *0*
shows *r k* ((*a* ∗ *b*) $ *0*) = *r k* (*a* $ *0*) ∗ *r k* (*b* $ *0*) ⟷
fps-radical r k (*a* ∗ *b*) = *fps-radical r k a* ∗ *fps-radical r k b*
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *radical-divide*:
fixes *a* :: *'a::field-char-0 fps*
assumes *kp*: *k* > *0*
and *ra0*: (*r k* (*a* $ *0*)) ^ *k* = *a* $ *0*
and *rb0*: (*r k* (*b* $ *0*)) ^ *k* = *b* $ *0*
and *a0*: *a$0* ≠ *0*
and *b0*: *b$0* ≠ *0*
shows *r k* ((*a* $ *0*) / (*b$0*)) = *r k* (*a$0*) / *r k* (*b* $ *0*) ⟷
fps-radical r k (*a*/*b*) = *fps-radical r k a* / *fps-radical r k b*
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *radical-inverse*:
fixes *a* :: *'a::field-char-0 fps*
assumes *k*: *k* > *0*
and *ra0*: *r k* (*a* $ *0*) ^ *k* = *a* $ *0*

and *r1*: $(r\ k\ 1)\hat{}k = 1$
and *a0*: $a\$0 \neq 0$
shows $r\ k\ (inverse\ (a\ \$\ 0)) = r\ k\ 1\ /\ (r\ k\ (a\ \$\ 0)) \longleftrightarrow$
fps-radical r k (inverse a) = fps-radical r k 1 / fps-radical r k a
⟨*proof*⟩

5.14 Chain rule

lemma *fps-compose-deriv*:
fixes *a* :: *'a::idom fps*
assumes *b0*: $b\$0 = 0$
shows *fps-deriv (a oo b) = ((fps-deriv a) oo b) * fps-deriv b*
⟨*proof*⟩

lemma *fps-poly-sum-fps-X*:
assumes $\forall\, i > n.\ a\$i = 0$
shows *a = sum (λi. fps-const (a$i) * fps-X^i) {0..n}* (**is** *a = ?r*)
⟨*proof*⟩

5.15 Compositional inverses

fun *compinv* :: *'a fps ⇒ nat ⇒ 'a::field*
where
compinv a 0 = fps-X$0
| *compinv a (Suc n) =*
*(fps-X$ Suc n − sum (λi. (compinv a i) * (a^i)$Suc n) {0 .. n}) / (a$1) ^ Suc n*

definition *fps-inv a = Abs-fps (compinv a)*

lemma *fps-inv*:
assumes *a0*: $a\$0 = 0$
and *a1*: $a\$1 \neq 0$
shows *fps-inv a oo a = fps-X*
⟨*proof*⟩

fun *gcompinv* :: *'a fps ⇒ 'a fps ⇒ nat ⇒ 'a::field*
where
gcompinv b a 0 = b$0
| *gcompinv b a (Suc n) =*
*(b$ Suc n − sum (λi. (gcompinv b a i) * (a^i)$Suc n) {0 .. n}) / (a$1) ^ Suc n*

definition *fps-ginv b a = Abs-fps (gcompinv b a)*

lemma *fps-ginv*:
assumes *a0*: $a\$0 = 0$
and *a1*: $a\$1 \neq 0$
shows *fps-ginv b a oo a = b*

⟨*proof*⟩

lemma *fps-inv-ginv*: *fps-inv* = *fps-ginv fps-X*
⟨*proof*⟩

lemma *fps-compose-1*[*simp*]: *1 oo a* = *1*
⟨*proof*⟩

lemma *fps-compose-0*[*simp*]: *0 oo a* = *0*
⟨*proof*⟩

lemma *fps-compose-0-right*[*simp*]: *a oo 0* = *fps-const* (*a* $ *0*)
⟨*proof*⟩

lemma *fps-compose-add-distrib*: (*a* + *b*) *oo c* = (*a oo c*) + (*b oo c*)
⟨*proof*⟩

lemma *fps-compose-sum-distrib*: (*sum f S*) *oo a* = *sum* (λ*i*. *f i oo a*) *S*
⟨*proof*⟩

lemma *convolution-eq*:
sum (λ*i*. *a* (*i* :: *nat*) ∗ *b* (*n* − *i*)) {*0* .. *n*} =
sum (λ(*i*,*j*). *a i* ∗ *b j*) {(*i*,*j*). *i* ≤ *n* ∧ *j* ≤ *n* ∧ *i* + *j* = *n*}
⟨*proof*⟩

lemma *product-composition-lemma*:
assumes *c0*: *c*$*0* = (*0*::′*a*::*idom*)
and *d0*: *d*$*0* = *0*
shows ((*a oo c*) ∗ (*b oo d*))$*n* =
sum (λ(*k*,*m*). *a*$*k* ∗ *b*$*m* ∗ (*c*^*k* ∗ *d*^*m*) $ *n*) {(*k*,*m*). *k* + *m* ≤ *n*} (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *sum-pair-less-iff*:
sum (λ((*k*::*nat*),*m*). *a k* ∗ *b m* ∗ *c* (*k* + *m*)) {(*k*,*m*). *k* + *m* ≤ *n*} =
sum (λ*s*. *sum* (λ*i*. *a i* ∗ *b* (*s* − *i*) ∗ *c s*) {*0*..*s*}) {*0*..*n*}
(**is** *?l* = *?r*)
⟨*proof*⟩

lemma *fps-compose-mult-distrib-lemma*:
assumes *c0*: *c*$*0* = (*0*::′*a*::*idom*)
shows ((*a oo c*) ∗ (*b oo c*))$*n* = *sum* (λ*s*. *sum* (λ*i*. *a*$*i* ∗ *b*$(*s* − *i*) ∗ (*c*^*s*) $ *n*)
{*0*..*s*}) {*0*..*n*}
⟨*proof*⟩

lemma *fps-compose-mult-distrib*:
assumes *c0*: *c* $ *0* = (*0*::′*a*::*idom*)
shows (*a* ∗ *b*) *oo c* = (*a oo c*) ∗ (*b oo c*)
⟨*proof*⟩

lemma *fps-compose-prod-distrib*:
  assumes *c0*: *c$0 = (0::'a::idom)*
  shows *prod a S oo c = prod (λk. a k oo c) S*
⟨*proof*⟩

lemma *fps-compose-divide*:
  assumes [*simp*]: *g dvd f h $ 0 = 0*
  shows *fps-compose f h = fps-compose (f / g :: 'a :: field fps) h ∗ fps-compose g h*
⟨*proof*⟩

lemma *fps-compose-divide-distrib*:
  assumes *g dvd f h $ 0 = 0 fps-compose g h ≠ 0*
  shows *fps-compose (f / g :: 'a :: field fps) h = fps-compose f h / fps-compose g h*
  ⟨*proof*⟩

lemma *fps-compose-power*:
  assumes *c0*: *c$0 = (0::'a::idom)*
  shows *(a oo c)^n = a^n oo c*
⟨*proof*⟩

lemma *fps-compose-uminus*: *− (a::'a::ring-1 fps) oo c = − (a oo c)*
  ⟨*proof*⟩

lemma *fps-compose-sub-distrib*: *(a − b) oo (c::'a::ring-1 fps) = (a oo c) − (b oo c)*
  ⟨*proof*⟩

lemma *fps-X-fps-compose*: *fps-X oo a = Abs-fps (λn. if n = 0 then (0::'a::comm-ring-1) else a$n)*
  ⟨*proof*⟩

lemma *fps-inverse-compose*:
  assumes *b0*: *(b$0 :: 'a::field) = 0*
    and *a0*: *a$0 ≠ 0*
  shows *inverse a oo b = inverse (a oo b)*
⟨*proof*⟩

lemma *fps-divide-compose*:
  assumes *c0*: *(c$0 :: 'a::field) = 0*
    and *b0*: *b$0 ≠ 0*
  shows *(a/b) oo c = (a oo c) / (b oo c)*
    ⟨*proof*⟩

lemma *gp*:
  assumes *a0*: *a$0 = (0::'a::field)*
  shows *(Abs-fps (λn. 1)) oo a = 1/(1 − a)*

(**is** *?one oo a* = -)
⟨*proof*⟩

lemma *fps-compose-radical*:
assumes *b0*: *b$0* = (*0*::*'a*::*field-char-0*)
and *ra0*: *r* (*Suc k*) (*a$0*) ^ *Suc k* = *a$0*
and *a0*: *a$0* ≠ *0*
shows *fps-radical r* (*Suc k*) *a oo b* = *fps-radical r* (*Suc k*) (*a oo b*)
⟨*proof*⟩

lemma *fps-const-mult-apply-left*: *fps-const c* ∗ (*a oo b*) = (*fps-const c* ∗ *a*) *oo b*
⟨*proof*⟩

lemma *fps-const-mult-apply-right*:
(*a oo b*) ∗ *fps-const* (*c*::*'a*::*comm-semiring-1*) = (*fps-const c* ∗ *a*) *oo b*
⟨*proof*⟩

lemma *fps-compose-assoc*:
assumes *c0*: *c$0* = (*0*::*'a*::*idom*)
and *b0*: *b$0* = *0*
shows *a oo* (*b oo c*) = *a oo b oo c* (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *fps-X-power-compose*:
assumes *a0*: *a$0*=*0*
shows *fps-X^k oo a* = (*a*::*'a*::*idom fps*)^*k*
(**is** *?l* = *?r*)
⟨*proof*⟩

lemma *fps-inv-right*:
assumes *a0*: *a$0* = *0*
and *a1*: *a$1* ≠ *0*
shows *a oo fps-inv a* = *fps-X*
⟨*proof*⟩

lemma *fps-inv-deriv*:
assumes *a0*: *a$0* = (*0*::*'a*::*field*)
and *a1*: *a$1* ≠ *0*
shows *fps-deriv* (*fps-inv a*) = *inverse* (*fps-deriv a oo fps-inv a*)
⟨*proof*⟩

lemma *fps-inv-idempotent*:
assumes *a0*: *a$0* = *0*
and *a1*: *a$1* ≠ *0*
shows *fps-inv* (*fps-inv a*) = *a*
⟨*proof*⟩

lemma *fps-ginv-ginv*:

assumes *a0*: *a$0 = 0*
and *a1*: *a$1 ≠ 0*
and *c0*: *c$0 = 0*
and *c1*: *c$1 ≠ 0*
shows *fps-ginv b (fps-ginv c a) = b oo a oo fps-inv c*
⟨*proof*⟩

lemma *fps-ginv-deriv*:
assumes *a0:a$0 = (0::'a::field)*
and *a1*: *a$1 ≠ 0*
shows *fps-deriv (fps-ginv b a) = (fps-deriv b / fps-deriv a) oo fps-ginv fps-X a*
⟨*proof*⟩

lemma *fps-compose-linear*:
fps-compose (f :: 'a :: comm-ring-1 fps) (fps-const c ∗ fps-X) = Abs-fps (λn. c^n ∗ f $ n)
⟨*proof*⟩

lemma *fps-compose-uminus'*:
fps-compose f (−fps-X :: 'a :: comm-ring-1 fps) = Abs-fps (λn. (−1)^n ∗ f $ n)
⟨*proof*⟩

5.16 Elementary series

5.16.1 Exponential series

definition *fps-exp x = Abs-fps (λn. x^n / of-nat (fact n))*

lemma *fps-exp-deriv*[*simp*]: *fps-deriv (fps-exp a) = fps-const (a::'a::field-char-0) ∗ fps-exp a*
(**is** *?l = ?r*)
⟨*proof*⟩

lemma *fps-exp-unique-ODE*:
fps-deriv a = fps-const c ∗ a ⟷ a = fps-const (a$0) ∗ fps-exp (c::'a::field-char-0)
(**is** *?lhs ⟷ ?rhs*)
⟨*proof*⟩

lemma *fps-exp-add-mult*: *fps-exp (a + b) = fps-exp (a::'a::field-char-0) ∗ fps-exp b* (**is** *?l = ?r*)
⟨*proof*⟩

lemma *fps-exp-nth*[*simp*]: *fps-exp a $ n = a^n / of-nat (fact n)*
⟨*proof*⟩

lemma *fps-exp-0*[*simp*]: *fps-exp (0::'a::field) = 1*
⟨*proof*⟩

lemma *fps-exp-neg*: *fps-exp (− a) = inverse (fps-exp (a::'a::field-char-0))*
⟨*proof*⟩

lemma *fps-exp-nth-deriv*[*simp*]:
fps-nth-deriv n (*fps-exp* (*a*::′*a*::*field-char-0*)) = (*fps-const a*)^*n* ∗ (*fps-exp a*)
⟨*proof*⟩

lemma *fps-X-compose-fps-exp*[*simp*]: *fps-X oo fps-exp* (*a*::′*a*::*field*) = *fps-exp a* − *1*
⟨*proof*⟩

lemma *fps-inv-fps-exp-compose*:
assumes *a*: *a* ≠ *0*
shows *fps-inv* (*fps-exp a* − *1*) *oo* (*fps-exp a* − *1*) = *fps-X*
and (*fps-exp a* − *1*) *oo fps-inv* (*fps-exp a* − *1*) = *fps-X*
⟨*proof*⟩

lemma *fps-exp-power-mult*: (*fps-exp* (*c*::′*a*::*field-char-0*))^*n* = *fps-exp* (*of-nat n* ∗ *c*)
⟨*proof*⟩

lemma *radical-fps-exp*:
assumes *r*: *r* (*Suc k*) *1* = *1*
shows *fps-radical r* (*Suc k*) (*fps-exp* (*c*::′*a*::*field-char-0*)) = *fps-exp* (*c* / *of-nat* (*Suc k*))
⟨*proof*⟩

lemma *fps-exp-compose-linear* [*simp*]:
fps-exp (*d*::′*a*::*field-char-0*) *oo* (*fps-const c* ∗ *fps-X*) = *fps-exp* (*c* ∗ *d*)
⟨*proof*⟩

lemma *fps-fps-exp-compose-minus* [*simp*]:
fps-compose (*fps-exp c*) (−*fps-X*) = *fps-exp* (−*c* :: ′*a* :: *field-char-0*)
⟨*proof*⟩

lemma *fps-exp-eq-iff* [*simp*]: *fps-exp c* = *fps-exp d* ⟷ *c* = (*d* :: ′*a* :: *field-char-0*)
⟨*proof*⟩

lemma *fps-exp-eq-fps-const-iff* [*simp*]:
fps-exp (*c* :: ′*a* :: *field-char-0*) = *fps-const c*′ ⟷ *c* = *0* ∧ *c*′ = *1*
⟨*proof*⟩

lemma *fps-exp-neq-0* [*simp*]: ¬*fps-exp* (*c* :: ′*a* :: *field-char-0*) = *0*
⟨*proof*⟩

lemma *fps-exp-eq-1-iff* [*simp*]: *fps-exp* (*c* :: ′*a* :: *field-char-0*) = *1* ⟷ *c* = *0*
⟨*proof*⟩

lemma *fps-exp-neq-numeral-iff* [*simp*]:
fps-exp (*c* :: ′*a* :: *field-char-0*) = *numeral n* ⟷ *c* = *0* ∧ *n* = *Num.One*
⟨*proof*⟩

5.16.2 Logarithmic series

lemma *Abs-fps-if-0*:
Abs-fps (λn. *if* $n = 0$ *then* (v::'a::*ring-1*) *else* f n) =
fps-const v + *fps-X* $*$ *Abs-fps* (λn. f (*Suc* n))
⟨*proof*⟩

definition *fps-ln* :: 'a::*field-char-0* ⇒ 'a *fps*
where *fps-ln* c = *fps-const* ($1/c$) $*$ *Abs-fps* (λn. *if* $n = 0$ *then* 0 *else* ($-$ 1) ^(n $-$ 1) / *of-nat* n)

lemma *fps-ln-deriv*: *fps-deriv* (*fps-ln* c) = *fps-const* ($1/c$) $*$ *inverse* (1 + *fps-X*)
⟨*proof*⟩

lemma *fps-ln-nth*: *fps-ln* c \$ n = (*if* $n = 0$ *then* 0 *else* $1/c$ $*$ (($-$ 1) ^(n $-$ 1) / *of-nat* n))
⟨*proof*⟩

lemma *fps-ln-0* [*simp*]: *fps-ln* c \$ 0 = 0 ⟨*proof*⟩

lemma *fps-ln-fps-exp-inv*:
fixes a :: 'a::*field-char-0*
assumes a: $a \neq 0$
shows *fps-ln* a = *fps-inv* (*fps-exp* a $-$ 1) (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *fps-ln-mult-add*:
assumes *c0*: $c \neq 0$
and *d0*: $d \neq 0$
shows *fps-ln* c + *fps-ln* d = *fps-const* ($c+d$) $*$ *fps-ln* ($c*d$)
(**is** *?r* = *?l*)
⟨*proof*⟩

lemma *fps-X-dvd-fps-ln* [*simp*]: *fps-X dvd fps-ln* c
⟨*proof*⟩

5.16.3 Binomial series

definition *fps-binomial* a = *Abs-fps* (λn. a *gchoose* n)

lemma *fps-binomial-nth*[*simp*]: *fps-binomial* a \$ n = a *gchoose* n
⟨*proof*⟩

lemma *fps-binomial-ODE-unique*:
fixes c :: 'a::*field-char-0*
shows *fps-deriv* a = (*fps-const* c $*$ a) / (1 + *fps-X*) ⟷ a = *fps-const* (a\$$0$) $*$ *fps-binomial* c
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *fps-binomial-ODE-unique′*:
(*fps-deriv a* = *fps-const c* ∗ *a* / (*1* + *fps-X*) ∧ *a* $ *0* = *1*) ⟷ (*a* = *fps-binomial c*)
⟨*proof*⟩

lemma *fps-binomial-deriv*: *fps-deriv* (*fps-binomial c*) = *fps-const c* ∗ *fps-binomial c* / (*1* + *fps-X*)
⟨*proof*⟩

lemma *fps-binomial-add-mult*: *fps-binomial* (*c*+*d*) = *fps-binomial c* ∗ *fps-binomial d* (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *fps-binomial-minus-one*: *fps-binomial* (− *1*) = *inverse* (*1* + *fps-X*)
(**is** *?l* = *inverse ?r*)
⟨*proof*⟩

lemma *fps-binomial-of-nat*: *fps-binomial* (*of-nat n*) = (*1* + *fps-X* :: ′*a* :: *field-char-0 fps*) ^ *n*
⟨*proof*⟩

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

lemma *fps-binomial-power*: *fps-binomial a* ^ *n* = *fps-binomial* (*of-nat n* ∗ *a*)
⟨*proof*⟩

lemma *fps-binomial-1*: *fps-binomial 1* = *1* + *fps-X*
⟨*proof*⟩

lemma *fps-binomial-minus-of-nat*:
fps-binomial (− *of-nat n*) = *inverse* ((*1* + *fps-X* :: ′*a* :: *field-char-0 fps*) ^ *n*)
⟨*proof*⟩

lemma *one-minus-const-fps-X-power*:
c ≠ *0* ⟹ (*1* − *fps-const c* ∗ *fps-X*) ^ *n* =
fps-compose (*fps-binomial* (*of-nat n*)) (−*fps-const c* ∗ *fps-X*)
⟨*proof*⟩

lemma *one-minus-fps-X-const-neg-power*:
inverse ((*1* − *fps-const c* ∗ *fps-X*) ^ *n*) =
fps-compose (*fps-binomial* (−*of-nat n*)) (−*fps-const c* ∗ *fps-X*)
⟨*proof*⟩

lemma *fps-X-plus-const-power*:
c ≠ *0* ⟹ (*fps-X* + *fps-const c*) ^ *n* =
fps-const (*c*^*n*) ∗ *fps-compose* (*fps-binomial* (*of-nat n*)) (*fps-const* (*inverse c*) ∗ *fps-X*)
⟨*proof*⟩

lemma *fps-X-plus-const-neg-power*:
c ≠ *0* ⟹ *inverse* ((*fps-X* + *fps-const c*) ^ *n*) =
fps-const (*inverse c*^*n*) ∗ *fps-compose* (*fps-binomial* (−*of-nat n*)) (*fps-const* (*inverse c*) ∗ *fps-X*)
⟨*proof*⟩

lemma *one-minus-const-fps-X-neg-power′*:
fixes *c* :: ′*a* :: *field-char-0*
assumes *n* > *0*
shows *inverse* ((*1* − *fps-const c* ∗ *fps-X*) ^ *n*) = *Abs-fps* (λ*k*. *of-nat* ((*n* + *k* − *1*) *choose k*) ∗ *c*^*k*)
⟨*proof*⟩

Vandermonde's Identity as a consequence.

lemma *gbinomial-Vandermonde*:
sum (λ*k*. (*a gchoose k*) ∗ (*b gchoose* (*n* − *k*))) {*0*..*n*} = (*a* + *b*) *gchoose n*
⟨*proof*⟩

lemma *binomial-Vandermonde*:
sum (λ*k*. (*a choose k*) ∗ (*b choose* (*n* − *k*))) {*0*..*n*} = (*a* + *b*) *choose n*
⟨*proof*⟩

lemma *binomial-Vandermonde-same*: *sum* (λ*k*. (*n choose k*)^2) {*0*..*n*} = (*2* ∗ *n*) *choose n*
⟨*proof*⟩

lemma *Vandermonde-pochhammer-lemma*:
fixes *a* :: ′*a*::*field-char-0*
assumes *b*: ⋀*j*. *j*<*n* ⟹ *b* ≠ *of-nat j*
shows *sum* (λ*k*. (*pochhammer* (− *a*) *k* ∗ *pochhammer* (− (*of-nat n*)) *k*) /
(*of-nat* (*fact k*) ∗ *pochhammer* (*b* − *of-nat n* + *1*) *k*)) {*0*..*n*} =
pochhammer (− (*a* + *b*)) *n* / *pochhammer* (− *b*) *n*
(**is** *?l* = *?r*)
⟨*proof*⟩

lemma *Vandermonde-pochhammer*:
fixes *a* :: ′*a*::*field-char-0*
assumes *c*: ∀ *i* ∈ {*0*..< *n*}. *c* ≠ − *of-nat i*
shows *sum* (λ*k*. (*pochhammer a k* ∗ *pochhammer* (− (*of-nat n*)) *k*) /
(*of-nat* (*fact k*) ∗ *pochhammer c k*)) {*0*..*n*} = *pochhammer* (*c* − *a*) *n* / *pochhammer c n*
⟨*proof*⟩

5.16.4 Trigonometric functions

definition *fps-sin* (*c*::′*a*::*field-char-0*) =
Abs-fps (λ*n*. *if even n then 0 else* (− *1*) ^((*n* − *1*) *div 2*) ∗ *c*^*n* /(*of-nat* (*fact*

n)))

definition *fps-cos* (*c*::'*a*::*field-char-0*) =
Abs-fps (λ*n*. *if even n then* (− *1*) ^ (*n div 2*) ∗ *c*^*n* / (*of-nat* (*fact n*)) *else 0*)

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

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

lemma *fps-sin-deriv*:
fps-deriv (*fps-sin c*) = *fps-const c* ∗ *fps-cos c*
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *fps-cos-deriv*: *fps-deriv* (*fps-cos c*) = *fps-const* (− *c*)∗ (*fps-sin c*)
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *fps-sin-cos-sum-of-squares*: (*fps-cos c*)^2 + (*fps-sin c*)^2 = *1*
(**is** *?lhs* = -)
⟨*proof*⟩

lemma *fps-sin-nth-0* [*simp*]: *fps-sin c* $ *0* = *0*
⟨*proof*⟩

lemma *fps-sin-nth-1* [*simp*]: *fps-sin c* $ *Suc 0* = *c*
⟨*proof*⟩

lemma *fps-sin-nth-add-2*:
fps-sin c $ (*n* + *2*) = − (*c* ∗ *c* ∗ *fps-sin c* $ *n* / (*of-nat* (*n* + *1*) ∗ *of-nat* (*n* + *2*)))
⟨*proof*⟩

lemma *fps-cos-nth-0* [*simp*]: *fps-cos c* $ *0* = *1*
⟨*proof*⟩

lemma *fps-cos-nth-1* [*simp*]: *fps-cos c* $ *Suc 0* = *0*
⟨*proof*⟩

lemma *fps-cos-nth-add-2*:
fps-cos c $ (*n* + *2*) = − (*c* ∗ *c* ∗ *fps-cos c* $ *n* / (*of-nat* (*n* + *1*) ∗ *of-nat* (*n* + *2*)))
⟨*proof*⟩

lemma *nat-add-1-add-1*: (*n*::*nat*) + *1* + *1* = *n* + *2*
⟨*proof*⟩

lemma *eq-fps-sin*:
  assumes *a0*: *a* $ *0* = *0*
    and *a1*: *a* $ *1* = *c*
    and *a2*: *fps-deriv* (*fps-deriv* *a*) = − (*fps-const* *c* ∗ *fps-const* *c* ∗ *a*)
  shows *fps-sin* *c* = *a*
⟨*proof*⟩

lemma *eq-fps-cos*:
  assumes *a0*: *a* $ *0* = *1*
    and *a1*: *a* $ *1* = *0*
    and *a2*: *fps-deriv* (*fps-deriv* *a*) = − (*fps-const* *c* ∗ *fps-const* *c* ∗ *a*)
  shows *fps-cos* *c* = *a*
⟨*proof*⟩

lemma *fps-sin-add*: *fps-sin* (*a* + *b*) = *fps-sin* *a* ∗ *fps-cos* *b* + *fps-cos* *a* ∗ *fps-sin* *b*
⟨*proof*⟩

lemma *fps-cos-add*: *fps-cos* (*a* + *b*) = *fps-cos* *a* ∗ *fps-cos* *b* − *fps-sin* *a* ∗ *fps-sin* *b*
⟨*proof*⟩

lemma *fps-sin-even*: *fps-sin* (− *c*) = − *fps-sin* *c*
  ⟨*proof*⟩

lemma *fps-cos-odd*: *fps-cos* (− *c*) = *fps-cos* *c*
  ⟨*proof*⟩

definition *fps-tan* *c* = *fps-sin* *c* / *fps-cos* *c*

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

lemma *fps-tan-deriv*: *fps-deriv* (*fps-tan* *c*) = *fps-const* *c* / (*fps-cos* *c*)^2
⟨*proof*⟩

Connection to *fps-exp* over the complex numbers — Euler and de Moivre.

lemma *fps-exp-ii-sin-cos*: *fps-exp* (i ∗ *c*) = *fps-cos* *c* + *fps-const* i ∗ *fps-sin* *c*
  (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *fps-exp-minus-ii-sin-cos*: *fps-exp* (− (i ∗ *c*)) = *fps-cos* *c* − *fps-const* i ∗ *fps-sin* *c*
  ⟨*proof*⟩

lemma *fps-cos-fps-exp-ii*: *fps-cos* *c* = (*fps-exp* (i ∗ *c*) + *fps-exp* (− i ∗ *c*)) / *fps-const* *2*
⟨*proof*⟩

lemma *fps-sin-fps-exp-ii*: *fps-sin c* = (*fps-exp* (i ∗ *c*) − *fps-exp* (− i ∗ *c*)) / *fps-const* (*2*∗i)
⟨*proof*⟩

lemma *fps-tan-fps-exp-ii*:
fps-tan c = (*fps-exp* (i ∗ *c*) − *fps-exp* (− i ∗ *c*)) /
(*fps-const* i ∗ (*fps-exp* (i ∗ *c*) + *fps-exp* (− i ∗ *c*)))
⟨*proof*⟩

lemma *fps-demoivre*:
(*fps-cos a* + *fps-const* i ∗ *fps-sin a*)^*n* =
fps-cos (*of-nat n* ∗ *a*) + *fps-const* i ∗ *fps-sin* (*of-nat n* ∗ *a*)
⟨*proof*⟩

5.17 Hypergeometric series

definition *fps-hypergeo as bs* (*c*::′*a*::*field-char-0*) =
Abs-fps (λ*n*. (*foldl* (λ*r a*. *r*∗ *pochhammer a n*) *1 as* ∗ *c*^*n*) /
(*foldl* (λ*r b*. *r* ∗ *pochhammer b n*) *1 bs* ∗ *of-nat* (*fact n*)))

lemma *fps-hypergeo-nth*[*simp*]: *fps-hypergeo as bs c* $ *n* =
(*foldl* (λ*r a*. *r*∗ *pochhammer a n*) *1 as* ∗ *c*^*n*) /
(*foldl* (λ*r b*. *r* ∗ *pochhammer b n*) *1 bs* ∗ *of-nat* (*fact n*))
⟨*proof*⟩

lemma *foldl-mult-start*:
fixes *v* :: ′*a*::*comm-ring-1*
shows *foldl* (λ*r x*. *r* ∗ *f x*) *v as* ∗ *x* = *foldl* (λ*r x*. *r* ∗ *f x*) (*v* ∗ *x*) *as*
⟨*proof*⟩

lemma *foldr-mult-foldl*:
fixes *v* :: ′*a*::*comm-ring-1*
shows *foldr* (λ*x r*. *r* ∗ *f x*) *as v* = *foldl* (λ*r x*. *r* ∗ *f x*) *v as*
⟨*proof*⟩

lemma *fps-hypergeo-nth-alt*:
fps-hypergeo as bs c $ *n* = *foldr* (λ*a r*. *r* ∗ *pochhammer a n*) *as* (*c* ^ *n*) /
foldr (λ*b r*. *r* ∗ *pochhammer b n*) *bs* (*of-nat* (*fact n*))
⟨*proof*⟩

lemma *fps-hypergeo-fps-exp*[*simp*]: *fps-hypergeo* [] [] *c* = *fps-exp c*
⟨*proof*⟩

lemma *fps-hypergeo-1-0*[*simp*]: *fps-hypergeo* [*1*] [] *c* = *1*/(*1* − *fps-const c* ∗ *fps-X*)
⟨*proof*⟩

lemma *fps-hypergeo-B*[*simp*]: *fps-hypergeo* [−*a*] [] (− *1*) = *fps-binomial a*
⟨*proof*⟩

lemma *fps-hypergeo-0*[*simp*]: *fps-hypergeo as bs c* $ *0* = *1*
⟨*proof*⟩

lemma *foldl-prod-prod*:
foldl (λ(*r*::'*b*::*comm-ring-1*) (*x*::'*a*::*comm-ring-1*). *r* ∗ *f x*) *v as* ∗ *foldl* (λ*r x*. *r* ∗ *g x*) *w as* =
foldl (λ*r x*. *r* ∗ *f x* ∗ *g x*) (*v* ∗ *w*) *as*
⟨*proof*⟩

lemma *fps-hypergeo-rec*:
fps-hypergeo as bs c $ *Suc n* = ((*foldl* (λ*r a*. *r*∗ (*a* + *of-nat n*)) *c as*) /
(*foldl* (λ*r b*. *r* ∗ (*b* + *of-nat n*)) (*of-nat* (*Suc n*)) *bs*)) ∗ *fps-hypergeo as bs c* $ *n*
⟨*proof*⟩

lemma *fps-XD-nth*[*simp*]: *fps-XD a* $ *n* = *of-nat n* ∗ *a*$*n*
⟨*proof*⟩

lemma *fps-XD-0th*[*simp*]: *fps-XD a* $ *0* = *0*
⟨*proof*⟩
lemma *fps-XD-Suc*[*simp*]: *fps-XD a* $ *Suc n* = *of-nat* (*Suc n*) ∗ *a* $ *Suc n*
⟨*proof*⟩

definition *fps-XDp c a* = *fps-XD a* + *fps-const c* ∗ *a*

lemma *fps-XDp-nth*[*simp*]: *fps-XDp c a* $ *n* = (*c* + *of-nat n*) ∗ *a*$*n*
⟨*proof*⟩

lemma *fps-XDp-commute*: *fps-XDp b* ∘ *fps-XDp* (*c*::'*a*::*comm-ring-1*) = *fps-XDp c* ∘ *fps-XDp b*
⟨*proof*⟩

lemma *fps-XDp0* [*simp*]: *fps-XDp 0* = *fps-XD*
⟨*proof*⟩

lemma *fps-XDp-fps-integral* [*simp*]:
fixes *a* :: '*a*::{*division-ring*,*ring-char-0*} *fps*
shows *fps-XDp 0* (*fps-integral a c*) = *fps-X* ∗ *a*
⟨*proof*⟩

lemma *fps-hypergeo-minus-nat*:
fps-hypergeo [− *of-nat n*] [− *of-nat* (*n* + *m*)] (*c*::'*a*::*field-char-0*) $ *k* =
(*if k* ≤ *n then*
pochhammer (− *of-nat n*) *k* ∗ *c* ^ *k* / (*pochhammer* (− *of-nat* (*n* + *m*)) *k* ∗ *of-nat* (*fact k*))
else 0)
fps-hypergeo [− *of-nat m*] [− *of-nat* (*m* + *n*)] (*c*::'*a*::*field-char-0*) $ *k* =
(*if k* ≤ *m then*

pochhammer (− *of-nat m*) *k* ∗ *c* ^ *k* / (*pochhammer* (− *of-nat* (*m* + *n*)) *k* ∗ *of-nat* (*fact k*))
else 0)
⟨*proof*⟩

lemma *pochhammer-rec-if*: *pochhammer a n* = (*if n* = *0 then 1 else a* ∗ *pochhammer* (*a* + *1*) (*n* − *1*))
⟨*proof*⟩

lemma *fps-XDp-foldr-nth* [*simp*]: *foldr* (*λc r*. *fps-XDp c* ∘ *r*) *cs* (*λc*. *fps-XDp c a*) *c0* $ *n* =
foldr (*λc r*. (*c* + *of-nat n*) ∗ *r*) *cs* (*c0* + *of-nat n*) ∗ *a*$*n*
⟨*proof*⟩

lemma *genric-fps-XDp-foldr-nth*:
assumes *f*: ∀ *n c a*. *f c a* $ *n* = (*of-nat n* + *k c*) ∗ *a*$*n*
shows *foldr* (*λc r*. *f c* ∘ *r*) *cs* (*λc*. *g c a*) *c0* $ *n* =
foldr (*λc r*. (*k c* + *of-nat n*) ∗ *r*) *cs* (*g c0 a* $ *n*)
⟨*proof*⟩

lemma *dist-less-imp-nth-equal*:
assumes *dist f g* < *inverse* (*2* ^ *i*)
and*j* ≤ *i*
shows *f* $ *j* = *g* $ *j*
⟨*proof*⟩

lemma *nth-equal-imp-dist-less*:
assumes ⋀*j*. *j* ≤ *i* ⟹ *f* $ *j* = *g* $ *j*
shows *dist f g* < *inverse* (*2* ^ *i*)
⟨*proof*⟩

lemma *dist-less-eq-nth-equal*: *dist f g* < *inverse* (*2* ^ *i*) ⟷ (∀ *j* ≤ *i*. *f* $ *j* = *g* $ *j*)
⟨*proof*⟩

instance *fps* :: (*comm-ring-1*) *complete-space*
⟨*proof*⟩

no-notation *fps-nth* (**infixl** $ *75*)

bundle *fps-notation*
begin
notation *fps-nth* (**infixl** $ *75*)
end

end

6 Converting polynomials to formal power series

theory *Polynomial-FPS*
  imports *Polynomial Formal-Power-Series*
begin

context
  includes *fps-notation*
begin

definition *fps-of-poly* **where**
  fps-of-poly p = *Abs-fps* (*coeff p*)

lemma *fps-of-poly-eq-iff*: *fps-of-poly p* = *fps-of-poly q* ⟷ *p* = *q*
  ⟨*proof*⟩

lemma *fps-of-poly-nth* [*simp*]: *fps-of-poly p* $ *n* = *coeff p n*
  ⟨*proof*⟩

lemma *fps-of-poly-const*: *fps-of-poly* [:*c*:] = *fps-const c*
⟨*proof*⟩

lemma *fps-of-poly-0* [*simp*]: *fps-of-poly 0* = *0*
  ⟨*proof*⟩

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

lemma *fps-of-poly-1′* [*simp*]: *fps-of-poly* [:*1*:] = *1*
  ⟨*proof*⟩

lemma *fps-of-poly-numeral* [*simp*]: *fps-of-poly* (*numeral n*) = *numeral n*
  ⟨*proof*⟩

lemma *fps-of-poly-numeral′* [*simp*]: *fps-of-poly* [:*numeral n*:] = *numeral n*
  ⟨*proof*⟩

lemma *fps-of-poly-fps-X* [*simp*]: *fps-of-poly* [:*0*, *1*:] = *fps-X*
  ⟨*proof*⟩

lemma *fps-of-poly-add*: *fps-of-poly* (*p* + *q*) = *fps-of-poly p* + *fps-of-poly q*
  ⟨*proof*⟩

lemma *fps-of-poly-diff*: *fps-of-poly* (*p* − *q*) = *fps-of-poly p* − *fps-of-poly q*
  ⟨*proof*⟩

lemma *fps-of-poly-uminus*: *fps-of-poly* (−*p*) = −*fps-of-poly p*
  ⟨*proof*⟩

lemma *fps-of-poly-mult*: *fps-of-poly* $(p * q)$ = *fps-of-poly* p * *fps-of-poly* q
⟨*proof*⟩

lemma *fps-of-poly-smult*:
fps-of-poly (*smult* c p) = *fps-const* c * *fps-of-poly* p
⟨*proof*⟩

lemma *fps-of-poly-sum*: *fps-of-poly* (*sum* f A) = *sum* (λx. *fps-of-poly* (f x)) A
⟨*proof*⟩

lemma *fps-of-poly-sum-list*: *fps-of-poly* (*sum-list* *xs*) = *sum-list* (*map* *fps-of-poly* *xs*)
⟨*proof*⟩

lemma *fps-of-poly-prod*: *fps-of-poly* (*prod* f A) = *prod* (λx. *fps-of-poly* (f x)) A
⟨*proof*⟩

lemma *fps-of-poly-prod-list*: *fps-of-poly* (*prod-list* *xs*) = *prod-list* (*map* *fps-of-poly* *xs*)
⟨*proof*⟩

lemma *fps-of-poly-pCons*:
fps-of-poly (*pCons* (c :: $'a$ :: *semiring-1*) p) = *fps-const* c + *fps-of-poly* p * *fps-X*
⟨*proof*⟩

lemma *fps-of-poly-pderiv*: *fps-of-poly* (*pderiv* p) = *fps-deriv* (*fps-of-poly* p)
⟨*proof*⟩

lemma *fps-of-poly-power*: *fps-of-poly* (p ^ n) = *fps-of-poly* p ^ n
⟨*proof*⟩

lemma *fps-of-poly-monom*: *fps-of-poly* (*monom* (c :: $'a$ :: *comm-ring-1*) n) = *fps-const* c * *fps-X* ^ n
⟨*proof*⟩

lemma *fps-of-poly-monom'*: *fps-of-poly* (*monom* (1 :: $'a$ :: *comm-ring-1*) n) = *fps-X* ^ n
⟨*proof*⟩

lemma *fps-of-poly-div*:
assumes (q :: $'a$ :: *field poly*) *dvd* p
shows *fps-of-poly* (p *div* q) = *fps-of-poly* p / *fps-of-poly* q
⟨*proof*⟩

lemma *fps-of-poly-divide-numeral*:
fps-of-poly (*smult* (*inverse* (*numeral* c :: $'a$ :: *field*)) p) = *fps-of-poly* p / *numeral* c
⟨*proof*⟩

lemma *subdegree-fps-of-poly*:
  assumes $p \neq 0$
  defines $n \equiv$ *Polynomial.order 0 p*
  shows *subdegree (fps-of-poly p) = n*
⟨*proof*⟩

lemma *fps-of-poly-dvd*:
  assumes *p dvd q*
  shows *fps-of-poly (p :: 'a :: field poly) dvd fps-of-poly q*
⟨*proof*⟩

lemmas *fps-of-poly-simps* =
  fps-of-poly-0 fps-of-poly-1 fps-of-poly-numeral fps-of-poly-const fps-of-poly-fps-X fps-of-poly-add fps-of-poly-diff fps-of-poly-uminus fps-of-poly-mult fps-of-poly-smult fps-of-poly-sum fps-of-poly-sum-list fps-of-poly-prod fps-of-poly-prod-list fps-of-poly-pCons fps-of-poly-pderiv fps-of-poly-power fps-of-poly-monom fps-of-poly-divide-numeral

lemma *fps-of-poly-pcompose*:
  assumes *coeff q 0 = (0 :: 'a :: idom)*
  shows *fps-of-poly (pcompose p q) = fps-compose (fps-of-poly p) (fps-of-poly q)*
  ⟨*proof*⟩

lemmas *reify-fps-atom* =
  fps-of-poly-0 fps-of-poly-1' fps-of-poly-numeral' fps-of-poly-const fps-of-poly-fps-X

The following simproc can reduce the equality of two polynomial FPSs two equality of the respective polynomials. A polynomial FPS is one that only has finitely many non-zero coefficients and can therefore be written as *fps-of-poly p* for some polynomial *p*.

This may sound trivial, but it covers a number of annoying side conditions like $1 + fps\text{-}X \neq 0$ that would otherwise not be solved automatically.

⟨*ML*⟩

lemma *fps-of-poly-linear*: *fps-of-poly [:a,1 :: 'a :: field:] = fps-X + fps-const a*
  ⟨*proof*⟩

lemma *fps-of-poly-linear'*: *fps-of-poly [:1,a :: 'a :: field:] = 1 + fps-const a ∗ fps-X*
  ⟨*proof*⟩

lemma *fps-of-poly-cutoff* [*simp*]:
  fps-of-poly (poly-cutoff n p) = fps-cutoff n (fps-of-poly p)
  ⟨*proof*⟩

lemma *fps-of-poly-shift* [*simp*]: *fps-of-poly (poly-shift n p) = fps-shift n (fps-of-poly p)*

⟨*proof*⟩

definition *poly-subdegree* :: *'a::zero poly* ⇒ *nat* **where**
poly-subdegree p = *subdegree* (*fps-of-poly p*)

lemma *coeff-less-poly-subdegree*:
k < *poly-subdegree p* ⟹ *coeff p k* = *0*
⟨*proof*⟩

definition *prefix-length* :: (*'a* ⇒ *bool*) ⇒ *'a list* ⇒ *nat* **where**
prefix-length P xs = *length* (*takeWhile P xs*)

primrec *prefix-length-aux* :: (*'a* ⇒ *bool*) ⇒ *nat* ⇒ *'a list* ⇒ *nat* **where**
prefix-length-aux P acc [] = *acc*
| *prefix-length-aux P acc* (*x#xs*) = (*if P x then prefix-length-aux P* (*Suc acc*) *xs else acc*)

lemma *prefix-length-aux-correct*: *prefix-length-aux P acc xs* = *prefix-length P xs* + *acc*
⟨*proof*⟩

lemma *prefix-length-code* [*code*]: *prefix-length P xs* = *prefix-length-aux P 0 xs*
⟨*proof*⟩

lemma *prefix-length-le-length*: *prefix-length P xs* ≤ *length xs*
⟨*proof*⟩

lemma *prefix-length-less-length*: (∃ *x*∈*set xs*. ¬*P x*) ⟹ *prefix-length P xs* < *length xs*
⟨*proof*⟩

lemma *nth-prefix-length*:
(∃ *x*∈*set xs*. ¬*P x*) ⟹ ¬*P* (*xs* ! *prefix-length P xs*)
⟨*proof*⟩

lemma *nth-less-prefix-length*:
n < *prefix-length P xs* ⟹ *P* (*xs* ! *n*)
⟨*proof*⟩

lemma *poly-subdegree-code* [*code*]: *poly-subdegree p* = *prefix-length* ((=) *0*) (*coeffs p*)
⟨*proof*⟩

end

end

7 A formalization of formal Laurent series

theory *Formal-Laurent-Series*
imports
Polynomial-FPS
begin

7.1 The type of formal Laurent series

7.1.1 Type definition

typedef (**overloaded**) *'a fls* = {*f*::*int* ⇒ *'a*::*zero*. ∀$_\infty$ *n*::*nat*. *f* (− *int n*) = *0*}
morphisms *fls-nth Abs-fls*
⟨*proof*⟩

setup-lifting *type-definition-fls*

unbundle *fps-notation*
notation *fls-nth* (**infixl** $$ *75*)

lemmas *fls-eqI* = *iffD1*[*OF fls-nth-inject*, *OF iffD2*, *OF fun-eq-iff*, *OF allI*]

lemma *fls-eq-iff*: *f* = *g* ⟷ (∀ *n*. *f* $$ *n* = *g* $$ *n*)
⟨*proof*⟩

lemma *nth-Abs-fls* [*simp*]: ∀$_\infty$*n*. *f* (− *int n*) = *0* ⟹ *Abs-fls f* $$ *n* = *f n*
⟨*proof*⟩

lemmas *nth-Abs-fls-finite-nonzero-neg-nth* = *nth-Abs-fls*[*OF iffD2*, *OF eventually-cofinite*]
lemmas *nth-Abs-fls-ex-nat-lower-bound* = *nth-Abs-fls*[*OF iffD2*, *OF MOST-nat*]
lemmas *nth-Abs-fls-nat-lower-bound* = *nth-Abs-fls-ex-nat-lower-bound*[*OF exI*]

lemma *nth-Abs-fls-ex-lower-bound*:
assumes ∃ *N*. ∀ *n*<*N*. *f n* = *0*
shows *Abs-fls f* $$ *n* = *f n*
⟨*proof*⟩

lemmas *nth-Abs-fls-lower-bound* = *nth-Abs-fls-ex-lower-bound*[*OF exI*]

lemmas *MOST-fls-neg-nth-eq-0* [*simp*] = *CollectD*[*OF fls-nth*]
lemmas *fls-finite-nonzero-neg-nth* = *iffD1*[*OF eventually-cofinite MOST-fls-neg-nth-eq-0*]

lemma *fls-nth-vanishes-below-natE*:
fixes *f* :: *'a*::*zero fls*
obtains *N* :: *nat*
where ∀ *n*>*N*. *f*$$(−*int n*) = *0*
⟨*proof*⟩

lemma *fls-nth-vanishes-belowE*:

fixes f :: $'a$::*zero fls*
obtains N :: *int*
where $\forall n{<}N.\ f\$\$n = 0$
⟨*proof*⟩

7.1.2 Definition of basic Laurent series

instantiation *fls* :: (*zero*) *zero*
begin
lift-definition *zero-fls* :: $'a$ *fls* **is** λ-. *0* ⟨*proof*⟩
instance ⟨*proof*⟩
end

lemma *fls-zero-nth* [*simp*]: $0\ \$\$\ n = 0$
⟨*proof*⟩

lemma *fls-zero-eqI*: $(\bigwedge n.\ f\$\$n = 0) \Longrightarrow f = 0$
⟨*proof*⟩

lemma *fls-nonzeroI*: $f\$\$n \neq 0 \Longrightarrow f \neq 0$
⟨*proof*⟩

lemma *fls-nonzero-nth*: $f \neq 0 \longleftrightarrow (\exists\ n.\ f\ \$\$\ n \neq 0)$
⟨*proof*⟩

lemma *fls-trivial-delta-eq-zero* [*simp*]: $b = 0 \Longrightarrow$ *Abs-fls* ($\lambda n.$ *if* $n{=}a$ *then* b *else* $0) = 0$
⟨*proof*⟩

lemma *fls-delta-nth* [*simp*]:
Abs-fls ($\lambda n.$ *if* $n{=}a$ *then* b *else* 0) $\$\$\ n$ = (*if* $n{=}a$ *then* b *else* 0)
⟨*proof*⟩

instantiation *fls* :: ({*zero*,*one*}) *one*
begin
lift-definition *one-fls* :: $'a$ *fls* **is** $\lambda k.$ *if* $k = 0$ *then* 1 *else* 0
⟨*proof*⟩
instance ⟨*proof*⟩
end

lemma *fls-one-nth* [*simp*]:
$1\ \$\$\ n$ = (*if* $n = 0$ *then* 1 *else* 0)
⟨*proof*⟩

instance *fls* :: (*zero-neq-one*) *zero-neq-one*
⟨*proof*⟩

definition *fls-const* :: $'a$::*zero* $\Rightarrow$ $'a$ *fls*
where *fls-const* $c \equiv$ *Abs-fls* ($\lambda n.$ *if* $n = 0$ *then* c *else* 0)

lemma *fls-const-nth* [*simp*]: *fls-const c* $$ *n* = (*if n* = *0 then c else 0*)
⟨*proof*⟩

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

lemma *fls-const-nonzero*: *c* ≠ *0* ⟹ *fls-const c* ≠ *0*
⟨*proof*⟩

lemma *fls-const-eq-0-iff* [*simp*]: *fls-const c* = *0* ⟷ *c* = *0*
⟨*proof*⟩

lemma *fls-const-1* [*simp*]: *fls-const 1* = *1*
⟨*proof*⟩

lemma *fls-const-eq-1-iff* [*simp*]: *fls-const c* = *1* ⟷ *c* = *1*
⟨*proof*⟩

lift-definition *fls-X* :: ′*a*::{*zero*,*one*} *fls*
is λ*n*. *if n* = *1 then 1 else 0*
⟨*proof*⟩

lemma *fls-X-nth* [*simp*]:
fls-X $$ *n* = (*if n* = *1 then 1 else 0*)
⟨*proof*⟩

lemma *fls-X-nonzero* [*simp*]: (*fls-X* :: ′*a* :: *zero-neq-one fls*) ≠ *0*
⟨*proof*⟩

lift-definition *fls-X-inv* :: ′*a*::{*zero*,*one*} *fls*
is λ*n*. *if n* = −*1 then 1 else 0*
⟨*proof*⟩

lemma *fls-X-inv-nth* [*simp*]:
fls-X-inv $$ *n* = (*if n* = −*1 then 1 else 0*)
⟨*proof*⟩

lemma *fls-X-inv-nonzero* [*simp*]: (*fls-X-inv* :: ′*a* :: *zero-neq-one fls*) ≠ *0*
⟨*proof*⟩

7.2 Subdegrees

lemma *unique-fls-subdegree*:
assumes *f* ≠ *0*
shows ∃!*n*. *f*$$*n* ≠ *0* ∧ (∀ *m*. *f*$$*m* ≠ *0* ⟶ *n* ≤ *m*)
⟨*proof*⟩

definition *fls-subdegree* :: (′*a*::*zero*) *fls* ⇒ *int*

where *fls-subdegree f* ≡ (*if f* = *0 then 0 else LEAST n*::*int*. *f*$$*n* ≠ *0*)

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

lemma *nth-fls-subdegree-nonzero* [*simp*]: *f* ≠ *0* ⟹ *f* $$ *fls-subdegree f* ≠ *0*
⟨*proof*⟩

lemma *nth-fls-subdegree-zero-iff*: (*f* $$ *fls-subdegree f* = *0*) ⟷ (*f* = *0*)
⟨*proof*⟩

lemma *fls-subdegree-leI*: *f* $$ *n* ≠ *0* ⟹ *fls-subdegree f* ≤ *n*
⟨*proof*⟩

lemma *fls-subdegree-leI'*: *f* $$ *n* ≠ *0* ⟹ *n* ≤ *m* ⟹ *fls-subdegree f* ≤ *m*
⟨*proof*⟩

lemma *fls-eq0-below-subdegree* [*simp*]: *n* < *fls-subdegree f* ⟹ *f* $$ *n* = *0*
⟨*proof*⟩

lemma *fls-subdegree-geI*: *f* ≠ *0* ⟹ (⋀*k*. *k* < *n* ⟹ *f* $$ *k* = *0*) ⟹ *n* ≤ *fls-subdegree f*
⟨*proof*⟩

lemma *fls-subdegree-ge0I*: (⋀*k*. *k* < *0* ⟹ *f* $$ *k* = *0*) ⟹ *0* ≤ *fls-subdegree f*
⟨*proof*⟩

lemma *fls-subdegree-greaterI*:
assumes *f* ≠ *0* ⋀*k*. *k* ≤ *n* ⟹ *f* $$ *k* = *0*
shows *n* < *fls-subdegree f*
⟨*proof*⟩

lemma *fls-subdegree-eqI*: *f* $$ *n* ≠ *0* ⟹ (⋀*k*. *k* < *n* ⟹ *f* $$ *k* = *0*) ⟹ *fls-subdegree f* = *n*
⟨*proof*⟩

lemma *fls-delta-subdegree* [*simp*]:
b ≠ *0* ⟹ *fls-subdegree* (*Abs-fls* (λ*n*. *if n*=*a then b else 0*)) = *a*
⟨*proof*⟩

lemma *fls-delta0-subdegree*: *fls-subdegree* (*Abs-fls* (λ*n*. *if n*=*0 then a else 0*)) = *0*
⟨*proof*⟩

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

lemma *fls-const-subdegree* [*simp*]: *fls-subdegree* (*fls-const c*) = *0*
⟨*proof*⟩

lemma *fls-X-subdegree* [*simp*]: *fls-subdegree* (*fls-X*::'*a*::{*zero-neq-one*} *fls*) = *1*
⟨*proof*⟩

lemma *fls-X-inv-subdegree* [*simp*]: *fls-subdegree* (*fls-X-inv*::'*a*::{*zero-neq-one*} *fls*) = −*1*
⟨*proof*⟩

lemma *fls-eq-above-subdegreeI*:
assumes $N \leq$ *fls-subdegree f* $N \leq$ *fls-subdegree g* $\forall k \geq N.$ *f* $$ *k* = *g* $$ *k*
shows *f* = *g*
⟨*proof*⟩

7.3 Shifting

7.3.1 Shift definition

definition *fls-shift* :: *int* ⇒ ('*a*::*zero*) *fls* ⇒ '*a fls*
where *fls-shift n f* ≡ *Abs-fls* (λ*k*. *f* $$ (*k*+*n*))
— Since the index set is unbounded in both directions, we can shift in either direction.

lemma *fls-shift-nth* [*simp*]: *fls-shift m f* $$ *n* = *f* $$ (*n*+*m*)
⟨*proof*⟩

lemma *fls-shift-eq-iff*: (*fls-shift m f* = *fls-shift m g*) ⟷ (*f* = *g*)
⟨*proof*⟩

lemma *fls-shift-0* [*simp*]: *fls-shift 0 f* = *f*
⟨*proof*⟩

lemma *fls-shift-subdegree* [*simp*]:
f ≠ *0* ⟹ *fls-subdegree* (*fls-shift n f*) = *fls-subdegree f* − *n*
⟨*proof*⟩

lemma *fls-shift-fls-shift* [*simp*]: *fls-shift m* (*fls-shift k f*) = *fls-shift* (*k*+*m*) *f*
⟨*proof*⟩

lemma *fls-shift-fls-shift-reorder*:
fls-shift m (*fls-shift k f*) = *fls-shift k* (*fls-shift m f*)
⟨*proof*⟩

lemma *fls-shift-zero* [*simp*]: *fls-shift m 0* = *0*
⟨*proof*⟩

lemma *fls-shift-eq0-iff*: *fls-shift m f* = *0* ⟷ *f* = *0*
⟨*proof*⟩

lemma *fls-shift-eq-1-iff*: *fls-shift n f* = *1* ⟷ *f* = *fls-shift* (−*n*) *1*
⟨*proof*⟩

lemma *fls-shift-nonneg-subdegree*: $m \leq$ *fls-subdegree f* $\Longrightarrow$ *fls-subdegree* (*fls-shift m f*) ≥ 0
⟨*proof*⟩

lemma *fls-shift-delta*:
fls-shift m (*Abs-fls* ($\lambda n.$ *if* $n=a$ *then b else 0*)) = *Abs-fls* ($\lambda n.$ *if* $n=a-m$ *then b else 0*)
⟨*proof*⟩

lemma *fls-shift-const*:
fls-shift m (*fls-const c*) = *Abs-fls* ($\lambda n.$ *if* $n=-m$ *then c else 0*)
⟨*proof*⟩

lemma *fls-shift-const-nth*:
fls-shift m (*fls-const c*) \$\$ *n* = (*if* $n=-m$ *then c else 0*)
⟨*proof*⟩

lemma *fls-X-conv-shift-1*: *fls-X* = *fls-shift* (-1) *1*
⟨*proof*⟩

lemma *fls-X-shift-to-one* [*simp*]: *fls-shift 1 fls-X* = *1*
⟨*proof*⟩

lemma *fls-X-inv-conv-shift-1*: *fls-X-inv* = *fls-shift 1 1*
⟨*proof*⟩

lemma *fls-X-inv-shift-to-one* [*simp*]: *fls-shift* (-1) *fls-X-inv* = *1*
⟨*proof*⟩

lemma *fls-X-fls-X-inv-conv*:
fls-X = *fls-shift* (-2) *fls-X-inv fls-X-inv* = *fls-shift 2 fls-X*
⟨*proof*⟩

7.3.2 Base factor

Similarly to the *unit-factor* for formal power series, we can decompose a formal Laurent series as a power of the implied variable times a series of subdegree 0. (See lemma *fls-base-factor-X-power-decompose*.) But we will call this something other *unit-factor* because it will not satisfy assumption *is-unit-unit-factor* of *semidom-divide-unit-factor*.

definition *fls-base-factor* :: (*'a*::*zero*) *fls* ⇒ *'a fls*
where *fls-base-factor-def*[*simp*]: *fls-base-factor f* = *fls-shift* (*fls-subdegree f*) *f*

lemma *fls-base-factor-nth*: *fls-base-factor f* \$\$ *n* = *f* \$\$ (*n* + *fls-subdegree f*)
⟨*proof*⟩

lemma *fls-base-factor-nonzero* [*simp*]: $f \neq 0 \Longrightarrow$ *fls-base-factor f* $\neq 0$
⟨*proof*⟩

lemma *fls-base-factor-subdegree* [*simp*]: *fls-subdegree* (*fls-base-factor f*) = *0*
⟨*proof*⟩

lemma *fls-base-factor-base* [*simp*]:
fls-base-factor f $$ *fls-subdegree* (*fls-base-factor f*) = *f* $$ *fls-subdegree f*
⟨*proof*⟩

lemma *fls-conv-base-factor-shift-subdegree*:
f = *fls-shift* (−*fls-subdegree f*) (*fls-base-factor f*)
⟨*proof*⟩

lemma *fls-base-factor-idem*:
fls-base-factor (*fls-base-factor* (*f*::′*a*::*zero fls*)) = *fls-base-factor f*
⟨*proof*⟩

lemma *fls-base-factor-zero*: *fls-base-factor* (*0*::′*a*::*zero fls*) = *0*
⟨*proof*⟩

lemma *fls-base-factor-zero-iff*: *fls-base-factor* (*f*::′*a*::*zero fls*) = *0* ⟷ *f* = *0*
⟨*proof*⟩

lemma *fls-base-factor-nth-0*: *f* ≠ *0* ⟹ *fls-base-factor f* $$ *0* ≠ *0*
⟨*proof*⟩

lemma *fls-base-factor-one*: *fls-base-factor* (*1*::′*a*::{*zero*,*one*} *fls*) = *1*
⟨*proof*⟩

lemma *fls-base-factor-const*: *fls-base-factor* (*fls-const c*) = *fls-const c*
⟨*proof*⟩

lemma *fls-base-factor-delta*:
fls-base-factor (*Abs-fls* (λ*n. if n*=*a then c else 0*)) = *fls-const c*
⟨*proof*⟩

lemma *fls-base-factor-X*: *fls-base-factor* (*fls-X*::′*a*::{*zero-neq-one*} *fls*) = *1*
⟨*proof*⟩

lemma *fls-base-factor-X-inv*: *fls-base-factor* (*fls-X-inv*::′*a*::{*zero-neq-one*} *fls*) = *1*
⟨*proof*⟩

lemma *fls-base-factor-shift* [*simp*]: *fls-base-factor* (*fls-shift n f*) = *fls-base-factor f*
⟨*proof*⟩

7.4 Conversion between formal power and Laurent series

7.4.1 Converting Laurent to power series

We can truncate a Laurent series at index 0 to create a power series, called the regular part.

lift-definition *fls-regpart* :: ($'a$::*zero*) *fls* ⇒ $'a$ *fps*
is λf. *Abs-fps* (λn. *f* (*int n*))
⟨*proof*⟩

lemma *fls-regpart-nth* [*simp*]: *fls-regpart f* \$ *n* = *f* \$\$ (*int n*)
⟨*proof*⟩

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

lemma *fls-regpart-one* [*simp*]: *fls-regpart 1* = *1*
⟨*proof*⟩

lemma *fls-regpart-Abs-fls*:
$\forall_\infty n$. *F* (− *int n*) = *0* ⟹ *fls-regpart* (*Abs-fls F*) = *Abs-fps* (λn. *F* (*int n*))
⟨*proof*⟩

lemma *fls-regpart-delta*:
fls-regpart (*Abs-fls* (λn. *if n=a then b else 0*)) =
(*if a* < *0 then 0 else Abs-fps* (λn. *if n=nat a then b else 0*))
⟨*proof*⟩

lemma *fls-regpart-const* [*simp*]: *fls-regpart* (*fls-const c*) = *fps-const c*
⟨*proof*⟩

lemma *fls-regpart-fls-X* [*simp*]: *fls-regpart fls-X* = *fps-X*
⟨*proof*⟩

lemma *fls-regpart-fls-X-inv* [*simp*]: *fls-regpart fls-X-inv* = *0*
⟨*proof*⟩

lemma *fls-regpart-eq0-imp-nonpos-subdegree*:
assumes *fls-regpart f* = *0*
shows *fls-subdegree f* ≤ *0*
⟨*proof*⟩

lemma *fls-subdegree-lt-fls-regpart-subdegree*:
fls-subdegree f ≤ *int* (*subdegree* (*fls-regpart f*))
⟨*proof*⟩

lemma *fls-regpart-subdegree-conv*:
assumes *fls-subdegree f* ≥ *0*
shows *subdegree* (*fls-regpart f*) = *nat* (*fls-subdegree f*)

— This is the best we can do since if the subdegree is negative, we might still have the bad luck that the term at index 0 is equal to 0.
⟨*proof*⟩

lemma *fls-eq-conv-fps-eqI*:
assumes *0* ≤ *fls-subdegree f 0* ≤ *fls-subdegree g fls-regpart f* = *fls-regpart g*
shows *f* = *g*
⟨*proof*⟩

lemma *fls-regpart-shift-conv-fps-shift*:
m ≥ *0* ⟹ *fls-regpart* (*fls-shift m f*) = *fps-shift* (*nat m*) (*fls-regpart f*)
⟨*proof*⟩

lemma *fps-shift-fls-regpart-conv-fls-shift*:
fps-shift m (*fls-regpart f*) = *fls-regpart* (*fls-shift m f*)
⟨*proof*⟩

lemma *fps-unit-factor-fls-regpart*:
fls-subdegree f ≥ *0* ⟹ *unit-factor* (*fls-regpart f*) = *fls-regpart* (*fls-base-factor f*)
⟨*proof*⟩

The terms below the zeroth form a polynomial in the inverse of the implied variable, called the principle part.

lift-definition *fls-prpart* :: (*'a::zero*) *fls* ⇒ *'a poly*
is λ*f*. *Abs-poly* (λ*n*. *if n* = *0 then 0 else f* (− *int n*))
⟨*proof*⟩

lemma *fls-prpart-coeff* [*simp*]: *coeff* (*fls-prpart f*) *n* = (*if n* = *0 then 0 else f* $$ (− *int n*))
⟨*proof*⟩

lemma *fls-prpart-eq0-iff*: (*fls-prpart f* = *0*) ⟷ (*fls-subdegree f* ≥ *0*)
⟨*proof*⟩

lemma *fls-prpart0* [*simp*]: *fls-prpart 0* = *0*
⟨*proof*⟩

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

lemma *fls-prpart-delta*:
fls-prpart (*Abs-fls* (λ*n*. *if n*=*a then b else 0*)) =
(*if a*<*0 then Poly* (*replicate* (*nat* (−*a*)) *0* @ [*b*]) *else 0*)
⟨*proof*⟩

lemma *fls-prpart-const* [*simp*]: *fls-prpart* (*fls-const c*) = *0*
⟨*proof*⟩

lemma *fls-prpart-X* [*simp*]: *fls-prpart fls-X* = *0*

⟨*proof*⟩

lemma *fls-prpart-X-inv*: *fls-prpart fls-X-inv* = [:*0*,*1*:]
⟨*proof*⟩

lemma *degree-fls-prpart* [*simp*]:
degree (*fls-prpart f*) = *nat* (−*fls-subdegree f*)
⟨*proof*⟩

lemma *fls-prpart-shift*:
assumes *m* ≤ *0*
shows *fls-prpart* (*fls-shift m f*) = *pCons 0* (*poly-shift* (*Suc* (*nat* (−*m*))) (*fls-prpart f*))
⟨*proof*⟩

lemma *fls-prpart-base-factor*: *fls-prpart* (*fls-base-factor f*) = *0*
⟨*proof*⟩

The essential data of a formal Laurant series resides from the subdegree up.

abbreviation *fls-base-factor-to-fps* :: (*'a*::*zero*) *fls* ⇒ *'a fps*
where *fls-base-factor-to-fps f* ≡ *fls-regpart* (*fls-base-factor f*)

lemma *fls-base-factor-to-fps-conv-fps-shift*:
assumes *fls-subdegree f* ≥ *0*
shows *fls-base-factor-to-fps f* = *fps-shift* (*nat* (*fls-subdegree f*)) (*fls-regpart f*)
⟨*proof*⟩

lemma *fls-base-factor-to-fps-nth*:
fls-base-factor-to-fps f $ *n* = *f* $$ (*fls-subdegree f* + *int n*)
⟨*proof*⟩

lemma *fls-base-factor-to-fps-base*: *f* ≠ *0* ⟹ *fls-base-factor-to-fps f* $ *0* ≠ *0*
⟨*proof*⟩

lemma *fls-base-factor-to-fps-nonzero*: *f* ≠ *0* ⟹ *fls-base-factor-to-fps f* ≠ *0*
⟨*proof*⟩

lemma *fls-base-factor-to-fps-subdegree* [*simp*]: *subdegree* (*fls-base-factor-to-fps f*) = *0*
⟨*proof*⟩

lemma *fls-base-factor-to-fps-trivial*:
fls-subdegree f = *0* ⟹ *fls-base-factor-to-fps f* = *fls-regpart f*
⟨*proof*⟩

lemma *fls-base-factor-to-fps-zero*: *fls-base-factor-to-fps 0* = *0*
⟨*proof*⟩

lemma *fls-base-factor-to-fps-one*: *fls-base-factor-to-fps 1* = *1*

⟨*proof*⟩

lemma *fls-base-factor-to-fps-delta*:
fls-base-factor-to-fps (*Abs-fls* (*λn. if n=a then c else 0*)) = *fps-const c*
⟨*proof*⟩

lemma *fls-base-factor-to-fps-const*:
fls-base-factor-to-fps (*fls-const c*) = *fps-const c*
⟨*proof*⟩

lemma *fls-base-factor-to-fps-X*:
fls-base-factor-to-fps (*fls-X*::′*a*::{*zero-neq-one*} *fls*) = *1*
⟨*proof*⟩

lemma *fls-base-factor-to-fps-X-inv*:
fls-base-factor-to-fps (*fls-X-inv*::′*a*::{*zero-neq-one*} *fls*) = *1*
⟨*proof*⟩

lemma *fls-base-factor-to-fps-shift*:
fls-base-factor-to-fps (*fls-shift m f*) = *fls-base-factor-to-fps f*
⟨*proof*⟩

lemma *fls-base-factor-to-fps-base-factor*:
fls-base-factor-to-fps (*fls-base-factor f*) = *fls-base-factor-to-fps f*
⟨*proof*⟩

lemma *fps-unit-factor-fls-base-factor*:
unit-factor (*fls-base-factor-to-fps f*) = *fls-base-factor-to-fps f*
⟨*proof*⟩

7.4.2 Converting power to Laurent series

We can extend a power series by 0s below to create a Laurent series.

definition *fps-to-fls* :: (′*a*::*zero*) *fps* ⇒ ′*a fls*
where *fps-to-fls f* ≡ *Abs-fls* (*λk*::*int. if k<0 then 0 else f* $ (*nat k*))

lemma *fps-to-fls-nth* [*simp*]:
(*fps-to-fls f*) $$ *n* = (*if n* < *0 then 0 else f*$(*nat n*))
⟨*proof*⟩

lemma *fps-to-fls-eq-imp-fps-eq*:
assumes *fps-to-fls f* = *fps-to-fls g*
shows *f* = *g*
⟨*proof*⟩

lemma *fps-to-fls-eq-iff* [*simp*]: *fps-to-fls f* = *fps-to-fls g* ⟷ *f* = *g*
⟨*proof*⟩

lemma *fps-zero-to-fls* [*simp*]: *fps-to-fls 0* = *0*

⟨*proof*⟩

lemma *fps-to-fls-nonzeroI*: *f* ≠ *0* ⟹ *fps-to-fls f* ≠ *0*
⟨*proof*⟩

lemma *fps-one-to-fls* [*simp*]: *fps-to-fls 1* = *1*
⟨*proof*⟩

lemma *fps-to-fls-Abs-fps*:
fps-to-fls (*Abs-fps F*) = *Abs-fls* (λ*n. if n*<*0 then 0 else F* (*nat n*))
⟨*proof*⟩

lemma *fps-delta-to-fls*:
fps-to-fls (*Abs-fps* (λ*n. if n*=*a then b else 0*)) = *Abs-fls* (λ*n. if n*=*int a then b else 0*)
⟨*proof*⟩

lemma *fps-const-to-fls* [*simp*]: *fps-to-fls* (*fps-const c*) = *fls-const c*
⟨*proof*⟩

lemma *fps-X-to-fls* [*simp*]: *fps-to-fls fps-X* = *fls-X*
⟨*proof*⟩

lemma *fps-to-fls-eq-0-iff* [*simp*]: (*fps-to-fls f* = *0*) ⟷ (*f*=*0*)
⟨*proof*⟩

lemma *fps-to-fls-eq-1-iff* [*simp*]: *fps-to-fls f* = *1* ⟷ *f* = *1*
⟨*proof*⟩

lemma *fls-subdegree-fls-to-fps-gt0*: *fls-subdegree* (*fps-to-fls f*) ≥ *0*
⟨*proof*⟩

lemma *fls-subdegree-fls-to-fps*: *fls-subdegree* (*fps-to-fls f*) = *int* (*subdegree f*)
⟨*proof*⟩

lemma *fps-shift-to-fls* [*simp*]:
n ≤ *subdegree f* ⟹ *fps-to-fls* (*fps-shift n f*) = *fls-shift* (*int n*) (*fps-to-fls f*)
⟨*proof*⟩

lemma *fls-base-factor-fps-to-fls*: *fls-base-factor* (*fps-to-fls f*) = *fps-to-fls* (*unit-factor f*)
⟨*proof*⟩

lemma *fls-regpart-to-fls-trivial* [*simp*]:
fls-subdegree f ≥ *0* ⟹ *fps-to-fls* (*fls-regpart f*) = *f*
⟨*proof*⟩

lemma *fls-regpart-fps-trivial* [*simp*]: *fls-regpart* (*fps-to-fls f*) = *f*
⟨*proof*⟩

lemma *fps-to-fls-base-factor-to-fps*:
fps-to-fls (*fls-base-factor-to-fps f*) = *fls-base-factor f*
⟨*proof*⟩

lemma *fls-conv-base-factor-to-fps-shift-subdegree*:
f = *fls-shift* (−*fls-subdegree f*) (*fps-to-fls* (*fls-base-factor-to-fps f*))
⟨*proof*⟩

lemma *fls-base-factor-to-fps-to-fls*:
fls-base-factor-to-fps (*fps-to-fls f*) = *unit-factor f*
⟨*proof*⟩

lemma *fls-as-fps*:
fixes *f* :: *'a* :: *zero fls* **and** *n* :: *int*
assumes *n*: *n* ≥ −*fls-subdegree f*
obtains *f'* **where** *f* = *fls-shift n* (*fps-to-fls f'*)
⟨*proof*⟩

lemma *fls-as-fps'*:
fixes *f* :: *'a* :: *zero fls* **and** *n* :: *int*
assumes *n*: *n* ≥ −*fls-subdegree f*
shows ∃*f'*. *f* = *fls-shift n* (*fps-to-fls f'*)
⟨*proof*⟩

abbreviation
fls-regpart-as-fls f ≡ *fps-to-fls* (*fls-regpart f*)
abbreviation
fls-prpart-as-fls f ≡
fls-shift (−*fls-subdegree f*) (*fps-to-fls* (*fps-of-poly* (*reflect-poly* (*fls-prpart f*))))

lemma *fls-regpart-as-fls-nth*:
fls-regpart-as-fls f $$ *n* = (*if n* < *0 then 0 else f* $$ *n*)
⟨*proof*⟩

lemma *fls-regpart-idem*:
fls-regpart (*fls-regpart-as-fls f*) = *fls-regpart f*
⟨*proof*⟩

lemma *fls-prpart-as-fls-nth*:
fls-prpart-as-fls f $$ *n* = (*if n* < *0 then f* $$ *n else 0*)
⟨*proof*⟩

lemma *fls-prpart-idem* [*simp*]: *fls-prpart* (*fls-prpart-as-fls f*) = *fls-prpart f*
⟨*proof*⟩

lemma *fls-regpart-prpart*: *fls-regpart* (*fls-prpart-as-fls f*) = *0*
⟨*proof*⟩

lemma *fls-prpart-regpart*: *fls-prpart* (*fls-regpart-as-fls f*) = *0*
⟨*proof*⟩

7.5 Algebraic structures

7.5.1 Addition

instantiation *fls* :: (*monoid-add*) *plus*
begin
lift-definition *plus-fls* :: *'a fls* ⇒ *'a fls* ⇒ *'a fls* **is** λ*f g n*. *f n* + *g n*
⟨*proof*⟩
instance ⟨*proof*⟩
end

lemma *fls-plus-nth* [*simp*]: (*f* + *g*) $$ *n* = *f* $$ *n* + *g* $$ *n*
⟨*proof*⟩

lemma *fls-plus-const*: *fls-const x* + *fls-const y* = *fls-const* (*x*+*y*)
⟨*proof*⟩

lemma *fls-plus-subdegree*:
f + *g* ≠ *0* ⟹ *fls-subdegree* (*f* + *g*) ≥ *min* (*fls-subdegree f*) (*fls-subdegree g*)
⟨*proof*⟩

lemma *fls-shift-plus* [*simp*]:
fls-shift m (*f* + *g*) = (*fls-shift m f*) + (*fls-shift m g*)
⟨*proof*⟩

lemma *fls-regpart-plus* [*simp*]: *fls-regpart* (*f* + *g*) = *fls-regpart f* + *fls-regpart g*
⟨*proof*⟩

lemma *fls-prpart-plus* [*simp*] : *fls-prpart* (*f* + *g*) = *fls-prpart f* + *fls-prpart g*
⟨*proof*⟩

lemma *fls-decompose-reg-pr-parts*:
fixes *f* :: *'a* :: *monoid-add fls*
defines *R* ≡ *fls-regpart-as-fls f*
and *P* ≡ *fls-prpart-as-fls f*
shows *f* = *P* + *R*
and *f* = *R* + *P*
⟨*proof*⟩

lemma *fps-to-fls-plus* [*simp*]: *fps-to-fls* (*f* + *g*) = *fps-to-fls f* + *fps-to-fls g*
⟨*proof*⟩

instance *fls* :: (*monoid-add*) *monoid-add*
⟨*proof*⟩

instance *fls* :: (*comm-monoid-add*) *comm-monoid-add*
⟨*proof*⟩

7.5.2 Subtraction and negatives

instantiation *fls* :: (*group-add*) *minus*
begin
lift-definition *minus-fls* :: *'a fls* ⇒ *'a fls* ⇒ *'a fls* **is** $\lambda f\ g\ n.\ f\ n - g\ n$
⟨*proof*⟩
instance ⟨*proof*⟩
end

lemma *fls-minus-nth* [*simp*]: $(f - g)$ $$ $n = f$ $$ $n - g$ $$ n
⟨*proof*⟩

lemma *fls-minus-const*: *fls-const* x − *fls-const* y = *fls-const* $(x-y)$
⟨*proof*⟩

lemma *fls-subdegree-minus*:
$f - g \neq 0 \Longrightarrow$ *fls-subdegree* $(f - g) \geq$ *min* (*fls-subdegree* f) (*fls-subdegree* g)
⟨*proof*⟩

lemma *fls-shift-minus* [*simp*]: *fls-shift* m $(f - g)$ = (*fls-shift* m f) − (*fls-shift* m g)
⟨*proof*⟩

lemma *fls-regpart-minus* [*simp*]: *fls-regpart* $(f - g)$ = *fls-regpart* f − *fls-regpart* g
⟨*proof*⟩

lemma *fls-prpart-minus* [*simp*] : *fls-prpart* $(f - g)$ = *fls-prpart* f − *fls-prpart* g
⟨*proof*⟩

lemma *fps-to-fls-minus* [*simp*]: *fps-to-fls* $(f - g)$ = *fps-to-fls* f − *fps-to-fls* g
⟨*proof*⟩

instantiation *fls* :: (*group-add*) *uminus*
begin
lift-definition *uminus-fls* :: *'a fls* ⇒ *'a fls* **is** $\lambda f\ n.\ - f\ n$
⟨*proof*⟩
instance ⟨*proof*⟩
end

lemma *fls-uminus-nth* [*simp*]: $(-f)$ $$ $n = - (f$ $$ $n)$
⟨*proof*⟩

lemma *fls-const-uminus*[*simp*]: *fls-const* $(-x)$ = −*fls-const* x
⟨*proof*⟩

lemma *fls-shift-uminus* [*simp*]: *fls-shift* m $(- f)$ = − (*fls-shift* m f)
⟨*proof*⟩

lemma *fls-regpart-uminus* [*simp*]: *fls-regpart* $(- f)$ = − *fls-regpart* f
⟨*proof*⟩

lemma *fls-prpart-uminus* [*simp*] : *fls-prpart* (− *f*) = − *fls-prpart f*
⟨*proof*⟩

lemma *fps-to-fls-uminus* [*simp*]: *fps-to-fls* (− *f*) = − *fps-to-fls f*
⟨*proof*⟩

instance *fls* :: (*group-add*) *group-add*
⟨*proof*⟩

instance *fls* :: (*ab-group-add*) *ab-group-add*
⟨*proof*⟩

lemma *fls-uminus-subdegree* [*simp*]: *fls-subdegree* (−*f*) = *fls-subdegree f*
⟨*proof*⟩

lemma *fls-subdegree-minus-sym*: *fls-subdegree* (*g* − *f*) = *fls-subdegree* (*f* − *g*)
⟨*proof*⟩

lemma *fls-regpart-sub-prpart*: *fls-regpart* (*f* − *fls-prpart-as-fls f*) = *fls-regpart f*
⟨*proof*⟩

lemma *fls-prpart-sub-regpart*: *fls-prpart* (*f* − *fls-regpart-as-fls f*) = *fls-prpart f*
⟨*proof*⟩

7.5.3 Multiplication

instantiation *fls* :: ({*comm-monoid-add*, *times*}) *times*
begin
definition *fls-times-def*:
(∗) = (λ*f g*.
fls-shift
(− (*fls-subdegree f* + *fls-subdegree g*))
(*fps-to-fls* (*fls-base-factor-to-fps f* ∗ *fls-base-factor-to-fps g*))
)
instance ⟨*proof*⟩
end

lemma *fls-times-nth-eq0*: *n* < *fls-subdegree f* + *fls-subdegree g* ⟹ (*f* ∗ *g*) $$ *n* = *0*
⟨*proof*⟩

lemma *fls-times-nth*:
fixes *f df g dg*
defines *df* ≡ *fls-subdegree f* **and** *dg* ≡ *fls-subdegree g*
shows (*f* ∗ *g*) $$ *n* = (∑ *i*=*df* + *dg*..*n*. *f* $$ (*i* − *dg*) ∗ *g* $$ (*dg* + *n* − *i*))
and (*f* ∗ *g*) $$ *n* = (∑ *i*=*df*..*n* − *dg*. *f* $$ *i* ∗ *g* $$ (*n* − *i*))
and (*f* ∗ *g*) $$ *n* = (∑ *i*=*dg*..*n* − *df*. *f* $$ (*df* + *i* − *dg*) ∗ *g* $$ (*dg* + *n* − *df* − *i*))

and $(f * g)$ $$ $n = (\sum i{=}0..n - (df + dg).\ f$ $$ $(df + i) * g$ $$ $(n - df - i))$
⟨*proof*⟩

lemma *fls-times-base* [*simp*]:
$(f * g)$ $$ (*fls-subdegree* f + *fls-subdegree* g) =
(f $$ *fls-subdegree* f) $*$ (g $$ *fls-subdegree* g)
⟨*proof*⟩

instance *fls* :: ({*comm-monoid-add*, *mult-zero*}) *mult-zero*
⟨*proof*⟩

lemma *fls-mult-one*:
fixes f :: *'a*::{*comm-monoid-add*, *mult-zero*, *monoid-mult*} *fls*
shows $1 * f = f$
and $f * 1 = f$
⟨*proof*⟩

lemma *fls-mult-const-nth* [*simp*]:
fixes f :: *'a*::{*comm-monoid-add*, *mult-zero*} *fls*
shows (*fls-const* $x * f$) $$ $n = x * f$$$n$
and (f $*$ *fls-const* x) $$ $n = f$$$n * x$
⟨*proof*⟩

lemma *fls-const-mult-const*[*simp*]:
fixes $x\ y$ :: *'a*::{*comm-monoid-add*, *mult-zero*}
shows *fls-const* x $*$ *fls-const* y = *fls-const* $(x{*}y)$
⟨*proof*⟩

lemma *fls-mult-subdegree-ge*:
fixes $f\ g$ :: *'a*::{*comm-monoid-add*,*mult-zero*} *fls*
assumes $f{*}g \neq 0$
shows *fls-subdegree* $(f{*}g) \geq$ *fls-subdegree* f + *fls-subdegree* g
⟨*proof*⟩

lemma *fls-mult-subdegree-ge-0*:
fixes $f\ g$ :: *'a*::{*comm-monoid-add*,*mult-zero*} *fls*
assumes *fls-subdegree* $f \geq 0$ *fls-subdegree* $g \geq 0$
shows *fls-subdegree* $(f{*}g) \geq 0$
⟨*proof*⟩

lemma *fls-mult-nonzero-base-subdegree-eq*:
fixes $f\ g$ :: *'a*::{*comm-monoid-add*,*mult-zero*} *fls*
assumes f $$ (*fls-subdegree* f) $*$ g $$ (*fls-subdegree* g) $\neq 0$
shows *fls-subdegree* $(f{*}g)$ = *fls-subdegree* f + *fls-subdegree* g
⟨*proof*⟩

lemma *fls-subdegree-mult* [*simp*]:
fixes $f\ g$ :: *'a*::*semiring-no-zero-divisors fls*

assumes $f \neq 0$ $g \neq 0$
shows *fls-subdegree* $(f * g)$ = *fls-subdegree* f + *fls-subdegree* g
⟨*proof*⟩

lemma *fls-shifted-times-simps*:
fixes f g :: $'a$::{*comm-monoid-add*, *mult-zero*} *fls*
shows $f *$ (*fls-shift* n g) = *fls-shift* n $(f{*}g)$ (*fls-shift* n f) $* g$ = *fls-shift* n $(f{*}g)$
⟨*proof*⟩

lemma *fls-shifted-times-transfer*:
fixes f g :: $'a$::{*comm-monoid-add*, *mult-zero*} *fls*
shows *fls-shift* n $f * g = f *$ *fls-shift* n g
⟨*proof*⟩

lemma *fls-times-both-shifted-simp*:
fixes f g :: $'a$::{*comm-monoid-add*, *mult-zero*} *fls*
shows (*fls-shift* m f) $*$ (*fls-shift* n g) = *fls-shift* $(m{+}n)$ $(f{*}g)$
⟨*proof*⟩

lemma *fls-base-factor-mult-base-factor*:
fixes f g :: $'a$::{*comm-monoid-add*, *mult-zero*} *fls*
shows *fls-base-factor* ($f *$ *fls-base-factor* g) = *fls-base-factor* $(f * g)$
and *fls-base-factor* (*fls-base-factor* $f * g$) = *fls-base-factor* $(f * g)$
⟨*proof*⟩

lemma *fls-base-factor-mult-both-base-factor*:
fixes f g :: $'a$::{*comm-monoid-add*,*mult-zero*} *fls*
shows *fls-base-factor* (*fls-base-factor* $f *$ *fls-base-factor* g) = *fls-base-factor* $(f * g)$
⟨*proof*⟩

lemma *fls-base-factor-mult*:
fixes f g :: $'a$::*semiring-no-zero-divisors* *fls*
shows *fls-base-factor* $(f * g)$ = *fls-base-factor* $f *$ *fls-base-factor* g
⟨*proof*⟩

lemma *fls-times-conv-base-factor-times*:
fixes f g :: $'a$::{*comm-monoid-add*, *mult-zero*} *fls*
shows
$f * g$ =
fls-shift ($-$(*fls-subdegree* f + *fls-subdegree* g)) (*fls-base-factor* $f *$ *fls-base-factor* g)
⟨*proof*⟩

lemma *fls-times-base-factor-conv-shifted-times*:
— Convenience form of lemma *fls-times-both-shifted-simp*.
fixes f g :: $'a$::{*comm-monoid-add*, *mult-zero*} *fls*
shows
fls-base-factor $f *$ *fls-base-factor* g = *fls-shift* (*fls-subdegree* f + *fls-subdegree* g)

$(f * g)$
⟨*proof*⟩

lemma *fls-times-conv-regpart*:
fixes *f g* :: *'a*::{*comm-monoid-add*,*mult-zero*} *fls*
assumes *fls-subdegree f* $\geq$ *0 fls-subdegree g* $\geq$ *0*
shows *fls-regpart* $(f * g) =$ *fls-regpart f* $*$ *fls-regpart g*
⟨*proof*⟩

lemma *fls-base-factor-to-fps-mult-conv-unit-factor*:
fixes *f g* :: *'a*::{*comm-monoid-add*,*mult-zero*} *fls*
shows
fls-base-factor-to-fps $(f * g) =$
unit-factor (*fls-base-factor-to-fps f* $*$ *fls-base-factor-to-fps g*)
⟨*proof*⟩

lemma *fls-base-factor-to-fps-mult'*:
fixes *f g* :: *'a*::{*comm-monoid-add*,*mult-zero*} *fls*
assumes (*f* $$ *fls-subdegree f*) $*$ (*g* $$ *fls-subdegree g*) $\neq$ *0*
shows *fls-base-factor-to-fps* $(f * g) =$ *fls-base-factor-to-fps f* $*$ *fls-base-factor-to-fps g*
⟨*proof*⟩

lemma *fls-base-factor-to-fps-mult*:
fixes *f g* :: *'a*::*semiring-no-zero-divisors fls*
shows *fls-base-factor-to-fps* $(f * g) =$ *fls-base-factor-to-fps f* $*$ *fls-base-factor-to-fps g*
⟨*proof*⟩

lemma *fls-times-conv-fps-times*:
fixes *f g* :: *'a*::{*comm-monoid-add*,*mult-zero*} *fls*
assumes *fls-subdegree f* $\geq$ *0 fls-subdegree g* $\geq$ *0*
shows $f * g =$ *fps-to-fls* (*fls-regpart f* $*$ *fls-regpart g*)
⟨*proof*⟩

lemma *fps-times-conv-fls-times*:
fixes *f g* :: *'a*::{*comm-monoid-add*,*mult-zero*} *fps*
shows $f * g =$ *fls-regpart* (*fps-to-fls f* $*$ *fps-to-fls g*)
⟨*proof*⟩

lemma *fls-times-fps-to-fls*:
fixes *f g* :: *'a*::{*comm-monoid-add*,*mult-zero*} *fps*
shows *fps-to-fls* $(f * g) =$ *fps-to-fls f* $*$ *fps-to-fls g*
⟨*proof*⟩

lemma *fls-X-times-conv-shift*:
fixes *f* :: *'a*::{*comm-monoid-add*,*mult-zero*,*monoid-mult*} *fls*
shows *fls-X* $* f =$ *fls-shift* (-1) *f f* $*$ *fls-X* $=$ *fls-shift* (-1) *f*
⟨*proof*⟩

lemmas *fls-X-times-comm* = *trans-sym*[*OF fls-X-times-conv-shift*]

lemma *fls-subdegree-mult-fls-X*:
  fixes *f* :: *'a*::{*comm-monoid-add*,*mult-zero*,*monoid-mult*} *fls*
  assumes *f* ≠ *0*
  shows *fls-subdegree* (*fls-X* ∗ *f*) = *fls-subdegree f* + *1*
  and *fls-subdegree* (*f* ∗ *fls-X*) = *fls-subdegree f* + *1*
  ⟨*proof*⟩

lemma *fls-mult-fls-X-nonzero*:
  fixes *f* :: *'a*::{*comm-monoid-add*,*mult-zero*,*monoid-mult*} *fls*
  assumes *f* ≠ *0*
  shows *fls-X* ∗ *f* ≠ *0*
  and *f* ∗ *fls-X* ≠ *0*
  ⟨*proof*⟩

lemma *fls-base-factor-mult-fls-X*:
  fixes *f* :: *'a*::{*comm-monoid-add*,*monoid-mult*,*mult-zero*} *fls*
  shows *fls-base-factor* (*fls-X* ∗ *f*) = *fls-base-factor f*
  and *fls-base-factor* (*f* ∗ *fls-X*) = *fls-base-factor f*
  ⟨*proof*⟩

lemma *fls-X-inv-times-conv-shift*:
  fixes *f* :: *'a*::{*comm-monoid-add*,*mult-zero*,*monoid-mult*} *fls*
  shows *fls-X-inv* ∗ *f* = *fls-shift 1 f f* ∗ *fls-X-inv* = *fls-shift 1 f*
  ⟨*proof*⟩

lemmas *fls-X-inv-times-comm* = *trans-sym*[*OF fls-X-inv-times-conv-shift*]

lemma *fls-subdegree-mult-fls-X-inv*:
  fixes *f* :: *'a*::{*comm-monoid-add*,*mult-zero*,*monoid-mult*} *fls*
  assumes *f* ≠ *0*
  shows *fls-subdegree* (*fls-X-inv* ∗ *f*) = *fls-subdegree f* − *1*
  and *fls-subdegree* (*f* ∗ *fls-X-inv*) = *fls-subdegree f* − *1*
  ⟨*proof*⟩

lemma *fls-mult-fls-X-inv-nonzero*:
  fixes *f* :: *'a*::{*comm-monoid-add*,*mult-zero*,*monoid-mult*} *fls*
  assumes *f* ≠ *0*
  shows *fls-X-inv* ∗ *f* ≠ *0*
  and *f* ∗ *fls-X-inv* ≠ *0*
  ⟨*proof*⟩

lemma *fls-base-factor-mult-fls-X-inv*:
  fixes *f* :: *'a*::{*comm-monoid-add*,*monoid-mult*,*mult-zero*} *fls*
  shows *fls-base-factor* (*fls-X-inv* ∗ *f*) = *fls-base-factor f*
  and *fls-base-factor* (*f* ∗ *fls-X-inv*) = *fls-base-factor f*
  ⟨*proof*⟩

lemma *fls-mult-assoc-subdegree-ge-0*:
fixes *f g h* :: *'a::semiring-0 fls*
assumes *fls-subdegree f* ≥ *0 fls-subdegree g* ≥ *0 fls-subdegree h* ≥ *0*
shows *f* ∗ *g* ∗ *h* = *f* ∗ (*g* ∗ *h*)
⟨*proof*⟩

lemma *fls-mult-assoc-base-factor*:
fixes *a b c* :: *'a::semiring-0 fls*
shows
fls-base-factor a ∗ *fls-base-factor b* ∗ *fls-base-factor c* =
fls-base-factor a ∗ (*fls-base-factor b* ∗ *fls-base-factor c*)
⟨*proof*⟩

lemma *fls-mult-distrib-subdegree-ge-0*:
fixes *f g h* :: *'a::semiring-0 fls*
assumes *fls-subdegree f* ≥ *0 fls-subdegree g* ≥ *0 fls-subdegree h* ≥ *0*
shows (*f* + *g*) ∗ *h* = *f* ∗ *h* + *g* ∗ *h*
and *h* ∗ (*f* + *g*) = *h* ∗ *f* + *h* ∗ *g*
⟨*proof*⟩

lemma *fls-mult-distrib-base-factor*:
fixes *a b c* :: *'a::semiring-0 fls*
shows
fls-base-factor a ∗ (*fls-base-factor b* + *fls-base-factor c*) =
fls-base-factor a ∗ *fls-base-factor b* + *fls-base-factor a* ∗ *fls-base-factor c*
⟨*proof*⟩

instance *fls* :: (*semiring-0*) *semiring-0*
⟨*proof*⟩

lemma *fls-mult-commute-subdegree-ge-0*:
fixes *f g* :: *'a::comm-semiring-0 fls*
assumes *fls-subdegree f* ≥ *0 fls-subdegree g* ≥ *0*
shows *f* ∗ *g* = *g* ∗ *f*
⟨*proof*⟩

lemma *fls-mult-commute-base-factor*:
fixes *a b c* :: *'a::comm-semiring-0 fls*
shows *fls-base-factor a* ∗ *fls-base-factor b* = *fls-base-factor b* ∗ *fls-base-factor a*
⟨*proof*⟩

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

instance *fls* :: (*semiring-1*) *semiring-1*
⟨*proof*⟩

lemma *fls-of-nat*: (*of-nat n* :: *'a::semiring-1 fls*) = *fls-const* (*of-nat n*)

⟨*proof*⟩

lemma *fls-of-nat-nth*: *of-nat n* $$ *k* = (*if k=0 then of-nat n else 0*)
⟨*proof*⟩

lemma *fls-mult-of-nat-nth* [*simp*]:
shows (*of-nat k* ∗ *f*) $$ *n* = *of-nat k* ∗ *f*$$*n*
and (*f* ∗ *of-nat k*) $$ *n* = *f*$$*n* ∗ *of-nat k*
⟨*proof*⟩

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

lemma *fls-shift-of-nat-nth*:
fls-shift k (*of-nat a*) $$ *n* = (*if n*=−*k then of-nat a else 0*)
⟨*proof*⟩

lemma *fls-base-factor-of-nat* [*simp*]:
fls-base-factor (*of-nat n* :: ′*a*::*semiring-1 fls*) = (*of-nat n* :: ′*a fls*)
⟨*proof*⟩

lemma *fls-regpart-of-nat* [*simp*]: *fls-regpart* (*of-nat n*) = (*of-nat n* :: ′*a*::*semiring-1 fps*)
⟨*proof*⟩

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

lemma *fls-base-factor-to-fps-of-nat*:
fls-base-factor-to-fps (*of-nat n*) = (*of-nat n* :: ′*a*::*semiring-1 fps*)
⟨*proof*⟩

lemma *fps-to-fls-of-nat*:
fps-to-fls (*of-nat n*) = (*of-nat n* :: ′*a*::*semiring-1 fls*)
⟨*proof*⟩

lemma *fps-to-fls-numeral* [*simp*]: *fps-to-fls* (*numeral n*) = *numeral n*
⟨*proof*⟩

lemma *fls-const-power*: *fls-const* (*a* ^ *b*) = *fls-const a* ^ *b*
⟨*proof*⟩

lemma *fls-const-numeral* [*simp*]: *fls-const* (*numeral n*) = *numeral n*
⟨*proof*⟩

lemma *fls-mult-of-numeral-nth* [*simp*]:
shows (*numeral k* ∗ *f*) $$ *n* = *numeral k* ∗ *f* $$ *n*
and (*f* ∗ *numeral k*) $$ *n* = *f* $$ *n* ∗ *numeral k*
⟨*proof*⟩

lemma *fls-nth-numeral′* [*simp*]:
numeral n $$ *0* = *numeral n k* ≠ *0* ⟹ *numeral n* $$ *k* = *0*
⟨*proof*⟩

instance *fls* :: (*comm-semiring-1*) *comm-semiring-1*
⟨*proof*⟩

instance *fls* :: (*ring*) *ring* ⟨*proof*⟩

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

instance *fls* :: (*ring-1*) *ring-1* ⟨*proof*⟩

lemma *fls-of-int-nonneg*: (*of-int* (*int n*) :: ′*a*::*ring-1 fls*) = *fls-const* (*of-int* (*int n*))
⟨*proof*⟩

lemma *fls-of-int*: (*of-int i* :: ′*a*::*ring-1 fls*) = *fls-const* (*of-int i*)
⟨*proof*⟩

lemma *fls-of-int-nth*: *of-int n* $$ *k* = (*if k*=*0 then of-int n else 0*)
⟨*proof*⟩

lemma *fls-mult-of-int-nth* [*simp*]:
shows (*of-int k* ∗ *f*) $$ *n* = *of-int k* ∗ *f*$$*n*
and (*f* ∗ *of-int k*) $$ *n* = *f*$$*n* ∗ *of-int k*
⟨*proof*⟩

lemma *fls-subdegree-of-int* [*simp*]: *fls-subdegree* (*of-int i*) = *0*
⟨*proof*⟩

lemma *fls-shift-of-int-nth*:
fls-shift k (*of-int i*) $$ *n* = (*if n*=−*k then of-int i else 0*)
⟨*proof*⟩

lemma *fls-base-factor-of-int* [*simp*]:
fls-base-factor (*of-int i* :: ′*a*::*ring-1 fls*) = (*of-int i* :: ′*a fls*)
⟨*proof*⟩

lemma *fls-regpart-of-int* [*simp*]:
fls-regpart (*of-int i*) = (*of-int i* :: ′*a*::*ring-1 fps*)
⟨*proof*⟩

lemma *fls-prpart-of-int* [*simp*]: *fls-prpart* (*of-int n*) = *0*
⟨*proof*⟩

lemma *fls-base-factor-to-fps-of-int*:
fls-base-factor-to-fps (*of-int i*) = (*of-int i* :: ′*a*::*ring-1 fps*)

⟨*proof*⟩

lemma *fps-to-fls-of-int*:
fps-to-fls (*of-int i*) = (*of-int i* :: *'a::ring-1 fls*)
⟨*proof*⟩

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

instance *fls* :: (*semiring-no-zero-divisors*) *semiring-no-zero-divisors*
⟨*proof*⟩

instance *fls* :: (*semiring-1-no-zero-divisors*) *semiring-1-no-zero-divisors* ⟨*proof*⟩

instance *fls* :: (*ring-no-zero-divisors*) *ring-no-zero-divisors* ⟨*proof*⟩

instance *fls* :: (*ring-1-no-zero-divisors*) *ring-1-no-zero-divisors* ⟨*proof*⟩

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

lemma *semiring-char-fls* [*simp*]: *CHAR*(*'a* :: *comm-semiring-1 fls*) = *CHAR*(*'a*)
⟨*proof*⟩

instance *fls* :: ({*semiring-prime-char*,*comm-semiring-1*}) *semiring-prime-char*
⟨*proof*⟩
instance *fls* :: ({*comm-semiring-prime-char*,*comm-semiring-1*}) *comm-semiring-prime-char*
⟨*proof*⟩
instance *fls* :: ({*comm-ring-prime-char*,*comm-semiring-1*}) *comm-ring-prime-char*
⟨*proof*⟩
instance *fls* :: ({*idom-prime-char*,*comm-semiring-1*}) *idom-prime-char*
⟨*proof*⟩

7.5.4 Powers

lemma *fls-subdegree-prod*:
fixes F :: *'a* ⇒ *'b* :: *field-char-0 fls*
assumes $\bigwedge x.\ x \in I \Longrightarrow F\ x \neq 0$
shows *fls-subdegree* ($\prod x{\in}I.\ F\ x$) = ($\sum x{\in}I.$ *fls-subdegree* ($F\ x$))
⟨*proof*⟩

lemma *fls-subdegree-prod'*:
fixes F :: *'a* ⇒ *'b* :: *field-char-0 fls*
assumes $\bigwedge x.\ x \in I \Longrightarrow$ *fls-subdegree* ($F\ x$) ≠ *0*
shows *fls-subdegree* ($\prod x{\in}I.\ F\ x$) = ($\sum x{\in}I.$ *fls-subdegree* ($F\ x$))
⟨*proof*⟩

lemma *fls-pow-subdegree-ge*:
f^n ≠ *0* ⟹ *fls-subdegree* (*f^n*) ≥ *n* ∗ *fls-subdegree f*
⟨*proof*⟩

lemma *fls-pow-nth-below-subdegree*:
$k < n * \textit{fls-subdegree } f \Longrightarrow (f \hat{} n) \$\$ \ k = 0$
⟨*proof*⟩

lemma *fls-pow-base* [*simp*]:
$(f \hat{} n) \$\$ \ (n * \textit{fls-subdegree } f) = (f \$\$ \ \textit{fls-subdegree } f) \hat{} n$
⟨*proof*⟩

lemma *fls-pow-subdegree-eqI*:
$(f \$\$ \ \textit{fls-subdegree } f) \hat{} n \neq 0 \Longrightarrow \textit{fls-subdegree } (f \hat{} n) = n * \textit{fls-subdegree } f$
⟨*proof*⟩

lemma *fls-unit-base-subdegree-power*:
$x * f \$\$ \ \textit{fls-subdegree } f = 1 \Longrightarrow \textit{fls-subdegree } (f \hat{} n) = n * \textit{fls-subdegree } f$
$f \$\$ \ \textit{fls-subdegree } f * y = 1 \Longrightarrow \textit{fls-subdegree } (f \hat{} n) = n * \textit{fls-subdegree } f$
⟨*proof*⟩

lemma *fls-base-dvd1-subdegree-power*:
$f \$\$ \ \textit{fls-subdegree } f \ \textit{dvd } 1 \Longrightarrow \textit{fls-subdegree } (f \hat{} n) = n * \textit{fls-subdegree } f$
⟨*proof*⟩

lemma *fls-pow-subdegree-ge0*:
assumes $\textit{fls-subdegree } f \geq 0$
shows $\textit{fls-subdegree } (f \hat{} n) \geq 0$
⟨*proof*⟩

lemma *fls-subdegree-pow*:
fixes $f :: 'a::\textit{semiring-1-no-zero-divisors fls}$
shows $\textit{fls-subdegree } (f \hat{} n) = n * \textit{fls-subdegree } f$
⟨*proof*⟩

lemma *fls-shifted-pow*:
$(\textit{fls-shift } m \ f) \hat{} n = \textit{fls-shift } (n*m) \ (f \hat{} n)$
⟨*proof*⟩

lemma *fls-pow-conv-fps-pow*:
assumes $\textit{fls-subdegree } f \geq 0$
shows $f \hat{} n = \textit{fps-to-fls } (\ (\textit{fls-regpart } f) \hat{} n \ )$
⟨*proof*⟩

lemma *fps-to-fls-power*: $\textit{fps-to-fls } (f \hat{} n) = \textit{fps-to-fls } f \hat{} n$
⟨*proof*⟩

lemma *fls-pow-conv-regpart*:
$\textit{fls-subdegree } f \geq 0 \Longrightarrow \textit{fls-regpart } (f \hat{} n) = (\textit{fls-regpart } f) \hat{} n$
⟨*proof*⟩

These two lemmas show that shifting 1 is equivalent to powers of the implied variable.

lemma *fls-X-power-conv-shift-1*: *fls-X* ^ *n* = *fls-shift* (−*n*) *1*
⟨*proof*⟩

lemma *fls-X-inv-power-conv-shift-1*: *fls-X-inv* ^ *n* = *fls-shift n 1*
⟨*proof*⟩

abbreviation *fls-X-intpow* ≡ (λ*i*. *fls-shift* (−*i*) *1*)
— Unifies *fls-X* and *fls-X-inv* so that *fls-X-intpow* returns the equivalent of the implied variable raised to the supplied integer argument of *fls-X-intpow*, whether positive or negative.

lemma *fls-X-intpow-nonzero*[*simp*]: (*fls-X-intpow i* :: ′*a*::*zero-neq-one fls*) ≠ *0*
⟨*proof*⟩

lemma *fls-X-intpow-power*: (*fls-X-intpow i*) ^ *n* = *fls-X-intpow* (*n* ∗ *i*)
⟨*proof*⟩

lemma *fls-X-power-nth* [*simp*]: *fls-X* ^ *n* $$ *k* = (*if k*=*n then 1 else 0*)
⟨*proof*⟩

lemma *fls-X-inv-power-nth* [*simp*]: *fls-X-inv* ^ *n* $$ *k* = (*if k*=−*n then 1 else 0*)
⟨*proof*⟩

lemma *fls-X-pow-nonzero*[*simp*]: (*fls-X* ^ *n* :: ′*a* :: *semiring-1 fls*) ≠ *0*
⟨*proof*⟩

lemma *fls-X-inv-pow-nonzero*[*simp*]: (*fls-X-inv* ^ *n* :: ′*a* :: *semiring-1 fls*) ≠ *0*
⟨*proof*⟩

lemma *fls-subdegree-fls-X-pow* [*simp*]: *fls-subdegree* (*fls-X* ^ *n*) = *n*
⟨*proof*⟩

lemma *fls-subdegree-fls-X-inv-pow* [*simp*]: *fls-subdegree* (*fls-X-inv* ^ *n*) = −*n*
⟨*proof*⟩

lemma *fls-subdegree-fls-X-intpow* [*simp*]:
fls-subdegree ((*fls-X-intpow i*) :: ′*a*::*zero-neq-one fls*) = *i*
⟨*proof*⟩

lemma *fls-X-pow-conv-fps-X-pow*: *fls-regpart* (*fls-X* ^ *n*) = *fps-X* ^ *n*
⟨*proof*⟩

lemma *fls-X-inv-pow-regpart*: *n* > *0* ⟹ *fls-regpart* (*fls-X-inv* ^ *n*) = *0*
⟨*proof*⟩

lemma *fls-X-intpow-regpart*:
fls-regpart (*fls-X-intpow i*) = (*if i*≥*0 then fps-X* ^ *nat i else 0*)
⟨*proof*⟩

lemma *fls-X-power-times-conv-shift*:
fls-X ^ *n* ∗ *f* = *fls-shift* (−*int n*) *f* *f* ∗ *fls-X* ^ *n* = *fls-shift* (−*int n*) *f*
⟨*proof*⟩

lemma *fls-X-inv-power-times-conv-shift*:
fls-X-inv ^ *n* ∗ *f* = *fls-shift* (*int n*) *f* *f* ∗ *fls-X-inv* ^ *n* = *fls-shift* (*int n*) *f*
⟨*proof*⟩

lemma *fls-X-intpow-times-conv-shift*:
fixes *f* :: '*a*::*semiring-1 fls*
shows *fls-X-intpow i* ∗ *f* = *fls-shift* (−*i*) *f* *f* ∗ *fls-X-intpow i* = *fls-shift* (−*i*) *f*
⟨*proof*⟩

lemmas *fls-X-power-times-comm* = *trans-sym*[*OF fls-X-power-times-conv-shift*]
lemmas *fls-X-inv-power-times-comm* = *trans-sym*[*OF fls-X-inv-power-times-conv-shift*]

lemma *fls-X-intpow-times-comm*:
fixes *f* :: '*a*::*semiring-1 fls*
shows *fls-X-intpow i* ∗ *f* = *f* ∗ *fls-X-intpow i*
⟨*proof*⟩

lemma *fls-X-intpow-times-fls-X-intpow*:
(*fls-X-intpow i* :: '*a*::*semiring-1 fls*) ∗ *fls-X-intpow j* = *fls-X-intpow* (*i*+*j*)
⟨*proof*⟩

lemma *fls-X-intpow-diff-conv-times*:
fls-X-intpow (*i*−*j*) = (*fls-X-intpow i* :: '*a*::*semiring-1 fls*) ∗ *fls-X-intpow* (−*j*)
⟨*proof*⟩

lemma *fls-mult-fls-X-power-nonzero*:
assumes *f* ≠ *0*
shows *fls-X* ^ *n* ∗ *f* ≠ *0* *f* ∗ *fls-X* ^ *n* ≠ *0*
⟨*proof*⟩

lemma *fls-mult-fls-X-inv-power-nonzero*:
assumes *f* ≠ *0*
shows *fls-X-inv* ^ *n* ∗ *f* ≠ *0* *f* ∗ *fls-X-inv* ^ *n* ≠ *0*
⟨*proof*⟩

lemma *fls-mult-fls-X-intpow-nonzero*:
fixes *f* :: '*a*::*semiring-1 fls*
assumes *f* ≠ *0*
shows *fls-X-intpow i* ∗ *f* ≠ *0* *f* ∗ *fls-X-intpow i* ≠ *0*
⟨*proof*⟩

lemma *fls-subdegree-mult-fls-X-power*:
assumes *f* ≠ *0*
shows *fls-subdegree* (*fls-X* ^ *n* ∗ *f*) = *fls-subdegree f* + *n*
and *fls-subdegree* (*f* ∗ *fls-X* ^ *n*) = *fls-subdegree f* + *n*

⟨proof⟩

lemma *fls-subdegree-mult-fls-X-inv-power*:
assumes $f \neq 0$
shows *fls-subdegree* (*fls-X-inv* ^ $n * f$) = *fls-subdegree* $f - n$
and *fls-subdegree* ($f *$ *fls-X-inv* ^ n) = *fls-subdegree* $f - n$
⟨proof⟩

lemma *fls-subdegree-mult-fls-X-intpow*:
fixes f :: ′*a*::*semiring-1 fls*
assumes $f \neq 0$
shows *fls-subdegree* (*fls-X-intpow* $i * f$) = *fls-subdegree* $f + i$
and *fls-subdegree* ($f *$ *fls-X-intpow* i) = *fls-subdegree* $f + i$
⟨proof⟩

lemma *fls-X-shift*:
fls-shift ($-$*int n*) *fls-X* = *fls-X* ^ *Suc n*
fls-shift (*int* (*Suc n*)) *fls-X* = *fls-X-inv* ^ n
⟨proof⟩

lemma *fls-X-inv-shift*:
fls-shift (*int n*) *fls-X-inv* = *fls-X-inv* ^ *Suc n*
fls-shift ($-$ *int* (*Suc n*)) *fls-X-inv* = *fls-X* ^ n
⟨proof⟩

lemma *fls-X-power-base-factor*: *fls-base-factor* (*fls-X* ^ n) = *1*
⟨proof⟩

lemma *fls-X-inv-power-base-factor*: *fls-base-factor* (*fls-X-inv* ^ n) = *1*
⟨proof⟩

lemma *fls-X-intpow-base-factor*: *fls-base-factor* (*fls-X-intpow* i) = *1*
⟨proof⟩

lemma *fls-base-factor-mult-fls-X-power*:
shows *fls-base-factor* (*fls-X* ^ $n * f$) = *fls-base-factor* f
and *fls-base-factor* ($f *$ *fls-X* ^ n) = *fls-base-factor* f
⟨proof⟩

lemma *fls-base-factor-mult-fls-X-inv-power*:
shows *fls-base-factor* (*fls-X-inv* ^ $n * f$) = *fls-base-factor* f
and *fls-base-factor* ($f *$ *fls-X-inv* ^ n) = *fls-base-factor* f
⟨proof⟩

lemma *fls-base-factor-mult-fls-X-intpow*:
fixes f :: ′*a*::*semiring-1 fls*
shows *fls-base-factor* (*fls-X-intpow* $i * f$) = *fls-base-factor* f
and *fls-base-factor* ($f *$ *fls-X-intpow* i) = *fls-base-factor* f
⟨proof⟩

lemma *fls-X-power-base-factor-to-fps*: *fls-base-factor-to-fps* (*fls-X* ^ *n*) = *1*
⟨*proof*⟩

lemma *fls-X-inv-power-base-factor-to-fps*: *fls-base-factor-to-fps* (*fls-X-inv* ^ *n*) = *1*
⟨*proof*⟩

lemma *fls-X-intpow-base-factor-to-fps*: *fls-base-factor-to-fps* (*fls-X-intpow* *i*) = *1*
⟨*proof*⟩

lemma *fls-base-factor-X-power-decompose*:
  fixes *f* :: ′*a*::*semiring-1 fls*
  shows *f* = *fls-base-factor f* ∗ *fls-X-intpow* (*fls-subdegree f*)
  and *f* = *fls-X-intpow* (*fls-subdegree f*) ∗ *fls-base-factor f*
  ⟨*proof*⟩

lemma *fls-normalized-product-of-inverses*:
  assumes *f* ∗ *g* = *1*
  shows *fls-base-factor f* ∗ *fls-base-factor g* =
    fls-X ^ (*nat* (−(*fls-subdegree f*+*fls-subdegree g*)))
  and *fls-base-factor f* ∗ *fls-base-factor g* =
    fls-X-intpow (−(*fls-subdegree f*+*fls-subdegree g*))
  ⟨*proof*⟩

lemma *fls-fps-normalized-product-of-inverses*:
  assumes *f* ∗ *g* = *1*
  shows *fls-base-factor-to-fps f* ∗ *fls-base-factor-to-fps g* =
    fps-X ^ (*nat* (−(*fls-subdegree f*+*fls-subdegree g*)))
  ⟨*proof*⟩

7.5.5 Inverses

abbreviation *fls-left-inverse* ::
  ′*a*::{*comm-monoid-add*,*uminus*,*times*} *fls* ⇒ ′*a* ⇒ ′*a fls*
  where
  fls-left-inverse f x ≡
    fls-shift (*fls-subdegree f*) (*fps-to-fls* (*fps-left-inverse* (*fls-base-factor-to-fps f*) *x*))

abbreviation *fls-right-inverse* ::
  ′*a*::{*comm-monoid-add*,*uminus*,*times*} *fls* ⇒ ′*a* ⇒ ′*a fls*
  where
  fls-right-inverse f y ≡
    fls-shift (*fls-subdegree f*) (*fps-to-fls* (*fps-right-inverse* (*fls-base-factor-to-fps f*) *y*))

instantiation *fls* :: ({*comm-monoid-add*,*uminus*,*times*,*inverse*}) *inverse*
begin
  definition *fls-divide-def*:

f div g =
fls-shift (*fls-subdegree g* − *fls-subdegree f*) (
fps-to-fls ((*fls-base-factor-to-fps f*) *div* (*fls-base-factor-to-fps g*))
)

definition *fls-inverse-def*:
inverse f = *fls-shift* (*fls-subdegree f*) (*fps-to-fls* (*inverse* (*fls-base-factor-to-fps f*)))
instance ⟨*proof*⟩
end

lemma *fls-inverse-def'*:
inverse f = *fls-right-inverse f* (*inverse* (*f* $$ *fls-subdegree f*))
⟨*proof*⟩

lemma *fls-lr-inverse-base*:
fls-left-inverse f x $$ (−*fls-subdegree f*) = *x*
fls-right-inverse f y $$ (−*fls-subdegree f*) = *y*
⟨*proof*⟩

lemma *fls-inverse-base*:
f ≠ *0* ⟹ *inverse f* $$ (−*fls-subdegree f*) = *inverse* (*f* $$ *fls-subdegree f*)
⟨*proof*⟩

lemma *fls-lr-inverse-starting0*:
fixes *f* :: *'a*::{*comm-monoid-add*,*mult-zero*,*uminus*} *fls*
and *g* :: *'b*::{*ab-group-add*,*mult-zero*} *fls*
shows *fls-left-inverse f 0* = *0*
and *fls-right-inverse g 0* = *0*
⟨*proof*⟩

lemma *fls-lr-inverse-eq0-imp-starting0*:
fls-left-inverse f x = *0* ⟹ *x* = *0*
fls-right-inverse f x = *0* ⟹ *x* = *0*
⟨*proof*⟩

lemma *fls-lr-inverse-eq-0-iff*:
fixes *x* :: *'a*::{*comm-monoid-add*,*mult-zero*,*uminus*}
and *y* :: *'b*::{*ab-group-add*,*mult-zero*}
shows *fls-left-inverse f x* = *0* ⟷ *x* = *0*
and *fls-right-inverse g y* = *0* ⟷ *y* = *0*
⟨*proof*⟩

lemma *fls-inverse-eq-0-iff'*:
fixes *f* :: *'a*::{*ab-group-add*,*inverse*,*mult-zero*} *fls*
shows *inverse f* = *0* ⟷ (*inverse* (*f* $$ *fls-subdegree f*) = *0*)
⟨*proof*⟩

lemma *fls-inverse-eq-0-iff*[*simp*]:

inverse f = (*0*:: ('*a*::*division-ring*) *fls*) ⟷ *f* $$ *fls-subdegree f* = *0*
⟨*proof*⟩

lemmas *fls-inverse-eq-0'* = *iffD2*[*OF fls-inverse-eq-0-iff'*]
lemmas *fls-inverse-eq-0* = *iffD2*[*OF fls-inverse-eq-0-iff*]

lemma *fls-lr-inverse-const*:
fixes *a* :: '*a*::{*ab-group-add*,*mult-zero*}
and *b* :: '*b*::{*comm-monoid-add*,*mult-zero*,*uminus*}
shows *fls-left-inverse* (*fls-const a*) *x* = *fls-const x*
and *fls-right-inverse* (*fls-const b*) *y* = *fls-const y*
⟨*proof*⟩

lemma *fls-inverse-const*:
fixes *a* :: '*a*::{*comm-monoid-add*,*inverse*,*mult-zero*,*uminus*}
shows *inverse* (*fls-const a*) = *fls-const* (*inverse a*)
⟨*proof*⟩

lemma *fls-lr-inverse-of-nat*:
fixes *x* :: '*a*::{*ring-1*,*mult-zero*}
and *y* :: '*b*::{*semiring-1*,*uminus*}
shows *fls-left-inverse* (*of-nat n*) *x* = *fls-const x*
and *fls-right-inverse* (*of-nat n*) *y* = *fls-const y*
⟨*proof*⟩

lemma *fls-inverse-of-nat*:
inverse (*of-nat n* :: '*a* :: {*semiring-1*,*inverse*,*uminus*} *fls*) = *fls-const* (*inverse* (*of-nat n*))
⟨*proof*⟩

lemma *fls-lr-inverse-of-int*:
fixes *x* :: '*a*::{*ring-1*,*mult-zero*}
shows *fls-left-inverse* (*of-int n*) *x* = *fls-const x*
and *fls-right-inverse* (*of-int n*) *x* = *fls-const x*
⟨*proof*⟩

lemma *fls-inverse-of-int*:
inverse (*of-int n* :: '*a* :: {*ring-1*,*inverse*,*uminus*} *fls*) = *fls-const* (*inverse* (*of-int n*))
⟨*proof*⟩

lemma *fls-lr-inverse-zero*:
fixes *x* :: '*a*::{*ab-group-add*,*mult-zero*}
and *y* :: '*b*::{*comm-monoid-add*,*mult-zero*,*uminus*}
shows *fls-left-inverse 0 x* = *fls-const x*
and *fls-right-inverse 0 y* = *fls-const y*
⟨*proof*⟩

lemma *fls-inverse-zero-conv-fls-const*:

inverse (*0*::′*a*::{*comm-monoid-add*,*mult-zero*,*uminus*,*inverse*} *fls*) = *fls-const* (*inverse* *0*)
⟨*proof*⟩

lemma *fls-inverse-zero*′:
assumes *inverse* (*0*::′*a*::{*comm-monoid-add*,*inverse*,*mult-zero*,*uminus*}) = *0*
shows *inverse* (*0*::′*a fls*) = *0*
⟨*proof*⟩

lemma *fls-inverse-zero* [*simp*]: *inverse* (*0*::′*a*::*division-ring fls*) = *0*
⟨*proof*⟩

lemma *fls-inverse-base2*:
fixes *f* :: ′*a*::{*comm-monoid-add*,*mult-zero*,*uminus*,*inverse*} *fls*
shows *inverse f* $$ (−*fls-subdegree f*) = *inverse* (*f* $$ *fls-subdegree f*)
⟨*proof*⟩

lemma *fls-lr-inverse-one*:
fixes *x* :: ′*a*::{*ab-group-add*,*mult-zero*,*one*}
and *y* :: ′*b*::{*comm-monoid-add*,*mult-zero*,*uminus*,*one*}
shows *fls-left-inverse 1 x* = *fls-const x*
and *fls-right-inverse 1 y* = *fls-const y*
⟨*proof*⟩

lemma *fls-lr-inverse-one-one*:
fls-left-inverse 1 1 =
(*1*::′*a*::{*ab-group-add*,*mult-zero*,*one*} *fls*)
fls-right-inverse 1 1 =
(*1*::′*b*::{*comm-monoid-add*,*mult-zero*,*uminus*,*one*} *fls*)
⟨*proof*⟩

lemma *fls-inverse-one*:
assumes *inverse* (*1*::′*a*::{*comm-monoid-add*,*inverse*,*mult-zero*,*uminus*,*one*}) = *1*
shows *inverse* (*1*::′*a fls*) = *1*
⟨*proof*⟩

lemma *fls-left-inverse-delta*:
fixes *b* :: ′*a*::{*ab-group-add*,*mult-zero*}
assumes *b* ≠ *0*
shows *fls-left-inverse* (*Abs-fls* (λ*n. if n*=*a then b else 0*)) *x* =
Abs-fls (λ*n. if n*=−*a then x else 0*)
⟨*proof*⟩

lemma *fls-right-inverse-delta*:
fixes *b* :: ′*a*::{*comm-monoid-add*,*mult-zero*,*uminus*}
assumes *b* ≠ *0*
shows *fls-right-inverse* (*Abs-fls* (λ*n. if n*=*a then b else 0*)) *x* =
Abs-fls (λ*n. if n*=−*a then x else 0*)
⟨*proof*⟩

lemma *fls-inverse-delta-nonzero*:
fixes b :: $'a$::{*comm-monoid-add*,*inverse*,*mult-zero*,*uminus*}
assumes $b \neq 0$
shows *inverse* (*Abs-fls* (λn. *if* $n=a$ *then* b *else* 0)) =
Abs-fls (λn. *if* $n=-a$ *then* *inverse* b *else* 0)
⟨*proof*⟩

lemma *fls-inverse-delta*:
fixes b :: $'a$::*division-ring*
shows *inverse* (*Abs-fls* (λn. *if* $n=a$ *then* b *else* 0)) =
Abs-fls (λn. *if* $n=-a$ *then* *inverse* b *else* 0)
⟨*proof*⟩

lemma *fls-lr-inverse-X*:
fixes x :: $'a$::{*ab-group-add*,*mult-zero*,*zero-neq-one*}
and y :: $'b$::{*comm-monoid-add*,*uminus*,*mult-zero*,*zero-neq-one*}
shows *fls-left-inverse fls-X* x = *fls-shift* 1 (*fls-const* x)
and *fls-right-inverse fls-X* y = *fls-shift* 1 (*fls-const* y)
⟨*proof*⟩

lemma *fls-lr-inverse-X'*:
fixes x :: $'a$::{*ab-group-add*,*mult-zero*,*zero-neq-one*,*monoid-mult*}
and y :: $'b$::{*comm-monoid-add*,*uminus*,*mult-zero*,*zero-neq-one*,*monoid-mult*}
shows *fls-left-inverse fls-X* x = *fls-const* x * *fls-X-inv*
and *fls-right-inverse fls-X* y = *fls-const* y * *fls-X-inv*
⟨*proof*⟩

lemma *fls-inverse-X'*:
assumes *inverse* 1 = (1::$'a$::{*comm-monoid-add*,*inverse*,*mult-zero*,*uminus*,*zero-neq-one*})
shows *inverse* (*fls-X*::$'a$ *fls*) = *fls-X-inv*
⟨*proof*⟩

lemma *fls-inverse-X*: *inverse* (*fls-X*::$'a$::*division-ring fls*) = *fls-X-inv*
⟨*proof*⟩

lemma *fls-lr-inverse-X-inv*:
fixes x :: $'a$::{*ab-group-add*,*mult-zero*,*zero-neq-one*}
and y :: $'b$::{*comm-monoid-add*,*uminus*,*mult-zero*,*zero-neq-one*}
shows *fls-left-inverse fls-X-inv* x = *fls-shift* (-1) (*fls-const* x)
and *fls-right-inverse fls-X-inv* y = *fls-shift* (-1) (*fls-const* y)
⟨*proof*⟩

lemma *fls-lr-inverse-X-inv'*:
fixes x :: $'a$::{*ab-group-add*,*mult-zero*,*zero-neq-one*,*monoid-mult*}
and y :: $'b$::{*comm-monoid-add*,*uminus*,*mult-zero*,*zero-neq-one*,*monoid-mult*}
shows *fls-left-inverse fls-X-inv* x = *fls-const* x * *fls-X*
and *fls-right-inverse fls-X-inv* y = *fls-const* y * *fls-X*
⟨*proof*⟩

lemma *fls-inverse-X-inv′*:
assumes *inverse 1* = (*1*::′*a*::{*comm-monoid-add*,*inverse*,*mult-zero*,*uminus*,*zero-neq-one*})
shows *inverse* (*fls-X-inv*::′*a fls*) = *fls-X*
⟨*proof*⟩

lemma *fls-inverse-X-inv*: *inverse* (*fls-X-inv*::′*a*::*division-ring fls*) = *fls-X*
⟨*proof*⟩

lemma *fls-lr-inverse-subdegree*:
assumes $x \neq 0$
shows *fls-subdegree* (*fls-left-inverse f x*) = − *fls-subdegree f*
and *fls-subdegree* (*fls-right-inverse f x*) = − *fls-subdegree f*
⟨*proof*⟩

lemma *fls-inverse-subdegree′*:
inverse (*f* $$ *fls-subdegree f*) ≠ *0* ⟹ *fls-subdegree* (*inverse f*) = − *fls-subdegree*
f
⟨*proof*⟩

lemma *fls-inverse-subdegree* [*simp*]:
fixes *f* :: ′*a*::*division-ring fls*
shows *fls-subdegree* (*inverse f*) = − *fls-subdegree f*
⟨*proof*⟩

lemma *fls-inverse-subdegree-base-nonzero*:
assumes *f* ≠ *0 inverse* (*f* $$ *fls-subdegree f*) ≠ *0*
shows *inverse f* $$ (*fls-subdegree* (*inverse f*)) = *inverse* (*f* $$ *fls-subdegree f*)
⟨*proof*⟩

lemma *fls-inverse-subdegree-base*:
fixes *f* :: ′*a*::{*ab-group-add*,*inverse*,*mult-zero*} *fls*
shows *inverse f* $$ (*fls-subdegree* (*inverse f*)) = *inverse* (*f* $$ *fls-subdegree f*)
⟨*proof*⟩

lemma *fls-lr-inverse-subdegree-0*:
assumes *fls-subdegree f* = *0*
shows *fls-subdegree* (*fls-left-inverse f x*) ≥ *0*
and *fls-subdegree* (*fls-right-inverse f x*) ≥ *0*
⟨*proof*⟩

lemma *fls-inverse-subdegree-0*:
fls-subdegree f = *0* ⟹ *fls-subdegree* (*inverse f*) ≥ *0*
⟨*proof*⟩

lemma *fls-lr-inverse-shift-nonzero*:
fixes *f* :: ′*a*::{*comm-monoid-add*,*mult-zero*,*uminus*} *fls*
assumes *f* ≠ *0*
shows *fls-left-inverse* (*fls-shift m f*) *x* = *fls-shift* (−*m*) (*fls-left-inverse f x*)

and *fls-right-inverse* (*fls-shift m f*) *x* = *fls-shift* (−*m*) (*fls-right-inverse f x*)
⟨*proof*⟩

lemma *fls-inverse-shift-nonzero*:
fixes *f* :: ′*a*::{*comm-monoid-add*,*inverse*,*mult-zero*,*uminus*} *fls*
assumes *f* ≠ *0*
shows *inverse* (*fls-shift m f*) = *fls-shift* (−*m*) (*inverse f*)
⟨*proof*⟩

lemma *fls-inverse-shift*:
fixes *f* :: ′*a*::*division-ring fls*
shows *inverse* (*fls-shift m f*) = *fls-shift* (−*m*) (*inverse f*)
⟨*proof*⟩

lemma *fls-left-inverse-base-factor*:
fixes *x* :: ′*a*::{*ab-group-add*,*mult-zero*}
assumes *x* ≠ *0*
shows *fls-left-inverse* (*fls-base-factor f*) *x* = *fls-base-factor* (*fls-left-inverse f x*)
⟨*proof*⟩

lemma *fls-right-inverse-base-factor*:
fixes *y* :: ′*a*::{*comm-monoid-add*,*mult-zero*,*uminus*}
assumes *y* ≠ *0*
shows *fls-right-inverse* (*fls-base-factor f*) *y* = *fls-base-factor* (*fls-right-inverse f y*)
⟨*proof*⟩

lemma *fls-inverse-base-factor*′:
fixes *f* :: ′*a*::{*comm-monoid-add*,*inverse*,*mult-zero*,*uminus*} *fls*
assumes *inverse* (*f* $$ *fls-subdegree f*) ≠ *0*
shows *inverse* (*fls-base-factor f*) = *fls-base-factor* (*inverse f*)
⟨*proof*⟩

lemma *fls-inverse-base-factor*:
fixes *f* :: ′*a*::{*ab-group-add*,*inverse*,*mult-zero*} *fls*
shows *inverse* (*fls-base-factor f*) = *fls-base-factor* (*inverse f*)
⟨*proof*⟩

lemma *fls-lr-inverse-regpart*:
assumes *fls-subdegree f* = *0*
shows *fls-regpart* (*fls-left-inverse f x*) = *fps-left-inverse* (*fls-regpart f*) *x*
and *fls-regpart* (*fls-right-inverse f y*) = *fps-right-inverse* (*fls-regpart f*) *y*
⟨*proof*⟩

lemma *fls-inverse-regpart*:
assumes *fls-subdegree f* = *0*
shows *fls-regpart* (*inverse f*) = *inverse* (*fls-regpart f*)
⟨*proof*⟩

lemma *fls-base-factor-to-fps-left-inverse*:
  fixes x :: *'a*::{*ab-group-add*,*mult-zero*}
  shows *fls-base-factor-to-fps* (*fls-left-inverse f x*) =
    fps-left-inverse (*fls-base-factor-to-fps f*) x
  ⟨*proof*⟩

lemma *fls-base-factor-to-fps-right-inverse-nonzero*:
  fixes y :: *'a*::{*comm-monoid-add*,*mult-zero*,*uminus*}
  assumes $y \neq 0$
  shows *fls-base-factor-to-fps* (*fls-right-inverse f y*) =
    fps-right-inverse (*fls-base-factor-to-fps f*) y
  ⟨*proof*⟩

lemma *fls-base-factor-to-fps-right-inverse*:
  fixes y :: *'a*::{*ab-group-add*,*mult-zero*}
  shows *fls-base-factor-to-fps* (*fls-right-inverse f y*) =
    fps-right-inverse (*fls-base-factor-to-fps f*) y
  ⟨*proof*⟩

lemma *fls-base-factor-to-fps-inverse-nonzero*:
  fixes f :: *'a*::{*comm-monoid-add*,*inverse*,*mult-zero*,*uminus*} *fls*
  assumes *inverse* (*f* $$ *fls-subdegree f*) ≠ *0*
  shows *fls-base-factor-to-fps* (*inverse f*) = *inverse* (*fls-base-factor-to-fps f*)
  ⟨*proof*⟩

lemma *fls-base-factor-to-fps-inverse*:
  fixes f :: *'a*::{*ab-group-add*,*inverse*,*mult-zero*} *fls*
  shows *fls-base-factor-to-fps* (*inverse f*) = *inverse* (*fls-base-factor-to-fps f*)
  ⟨*proof*⟩

lemma *fls-lr-inverse-fps-to-fls*:
  assumes *subdegree f* = *0*
  shows *fls-left-inverse* (*fps-to-fls f*) x = *fps-to-fls* (*fps-left-inverse f x*)
  and *fls-right-inverse* (*fps-to-fls f*) x = *fps-to-fls* (*fps-right-inverse f x*)
  ⟨*proof*⟩

lemma *fls-inverse-fps-to-fls*:
  subdegree f = *0* ⟹ *inverse* (*fps-to-fls f*) = *fps-to-fls* (*inverse f*)
  ⟨*proof*⟩

lemma *fls-lr-inverse-X-power*:
  fixes x :: *'a*::*ring-1*
  and y :: *'b*::{*semiring-1*,*uminus*}
  shows *fls-left-inverse* (*fls-X* ^ *n*) x = *fls-shift n* (*fls-const x*)
  and *fls-right-inverse* (*fls-X* ^ *n*) y = *fls-shift n* (*fls-const y*)
  ⟨*proof*⟩

lemma *fls-lr-inverse-X-power'*:
  fixes x :: *'a*::*ring-1*

and *y* :: *'b*::{*semiring-1*,*uminus*}
shows *fls-left-inverse* (*fls-X* ^ *n*) *x* = *fls-const x* ∗ *fls-X-inv* ^ *n*
and *fls-right-inverse* (*fls-X* ^ *n*) *y* = *fls-const y* ∗ *fls-X-inv* ^ *n*
⟨*proof*⟩

lemma *fls-inverse-X-power'*:
assumes *inverse 1* = (*1*::*'a*::{*semiring-1*,*uminus*,*inverse*})
shows *inverse* ((*fls-X* ^ *n*)::*'a fls*) = *fls-X-inv* ^ *n*
⟨*proof*⟩

lemma *fls-inverse-X-power*:
inverse ((*fls-X*::*'a*::*division-ring fls*) ^ *n*) = *fls-X-inv* ^ *n*
⟨*proof*⟩

lemma *fls-lr-inverse-X-inv-power*:
fixes *x* :: *'a*::*ring-1*
and *y* :: *'b*::{*semiring-1*,*uminus*}
shows *fls-left-inverse* (*fls-X-inv* ^ *n*) *x* = *fls-shift* (−*n*) (*fls-const x*)
and *fls-right-inverse* (*fls-X-inv* ^ *n*) *y* = *fls-shift* (−*n*) (*fls-const y*)
⟨*proof*⟩

lemma *fls-lr-inverse-X-inv-power'*:
fixes *x* :: *'a*::*ring-1*
and *y* :: *'b*::{*semiring-1*,*uminus*}
shows *fls-left-inverse* (*fls-X-inv* ^ *n*) *x* = *fls-const x* ∗ *fls-X* ^ *n*
and *fls-right-inverse* (*fls-X-inv* ^ *n*) *y* = *fls-const y* ∗ *fls-X* ^ *n*
⟨*proof*⟩

lemma *fls-inverse-X-inv-power'*:
assumes *inverse 1* = (*1*::*'a*::{*semiring-1*,*uminus*,*inverse*})
shows *inverse* ((*fls-X-inv* ^ *n*)::*'a fls*) = *fls-X* ^ *n*
⟨*proof*⟩

lemma *fls-inverse-X-inv-power*:
inverse ((*fls-X-inv*::*'a*::*division-ring fls*) ^ *n*) = *fls-X* ^ *n*
⟨*proof*⟩

lemma *fls-lr-inverse-X-intpow*:
fixes *x* :: *'a*::*ring-1*
and *y* :: *'b*::{*semiring-1*,*uminus*}
shows *fls-left-inverse* (*fls-X-intpow i*) *x* = *fls-shift i* (*fls-const x*)
and *fls-right-inverse* (*fls-X-intpow i*) *y* = *fls-shift i* (*fls-const y*)
⟨*proof*⟩

lemma *fls-lr-inverse-X-intpow'*:
fixes *x* :: *'a*::*ring-1*
and *y* :: *'b*::{*semiring-1*,*uminus*}
shows *fls-left-inverse* (*fls-X-intpow i*) *x* = *fls-const x* ∗ *fls-X-intpow* (−*i*)
and *fls-right-inverse* (*fls-X-intpow i*) *y* = *fls-const y* ∗ *fls-X-intpow* (−*i*)

⟨*proof*⟩

lemma *fls-inverse-X-intpow'*:
assumes *inverse 1* = (*1*::'*a*::{*semiring-1*,*uminus*,*inverse*})
shows *inverse* (*fls-X-intpow i* :: '*a fls*) = *fls-X-intpow* (−*i*)
⟨*proof*⟩

lemma *fls-inverse-X-intpow*:
inverse (*fls-X-intpow i* :: '*a*::*division-ring fls*) = *fls-X-intpow* (−*i*)
⟨*proof*⟩

lemma *fls-left-inverse*:
fixes *f* :: '*a*::*ring-1 fls*
assumes *x* ∗ *f* $$ *fls-subdegree f* = *1*
shows *fls-left-inverse f x* ∗ *f* = *1*
⟨*proof*⟩

lemma *fls-right-inverse*:
fixes *f* :: '*a*::*ring-1 fls*
assumes *f* $$ *fls-subdegree f* ∗ *y* = *1*
shows *f* ∗ *fls-right-inverse f y* = *1*
⟨*proof*⟩
lemma *fls-left-inverse-eq-fls-right-inverse*:
fixes *f* :: '*a*::*ring-1 fls*
assumes *x* ∗ *f* $$ *fls-subdegree f* = *1 f* $$ *fls-subdegree f* ∗ *y* = *1*
— These assumptions imply x equals y, but no need to assume that.
shows *fls-left-inverse f x* = *fls-right-inverse f y*
⟨*proof*⟩

lemma *fls-left-inverse-eq-inverse*:
fixes *f* :: '*a*::*division-ring fls*
shows *fls-left-inverse f* (*inverse* (*f* $$ *fls-subdegree f*)) = *inverse f*
⟨*proof*⟩

lemma *fls-right-inverse-eq-inverse*:
fixes *f* :: '*a*::*division-ring fls*
shows *fls-right-inverse f* (*inverse* (*f* $$ *fls-subdegree f*)) = *inverse f*
⟨*proof*⟩

lemma *fls-left-inverse-eq-fls-right-inverse-comm*:
fixes *f* :: '*a*::*comm-ring-1 fls*
assumes *x* ∗ *f* $$ *fls-subdegree f* = *1*
shows *fls-left-inverse f x* = *fls-right-inverse f x*
⟨*proof*⟩

lemma *fls-left-inverse'*:
fixes *f* :: '*a*::*ring-1 fls*
assumes *x* ∗ *f* $$ *fls-subdegree f* = *1 f* $$ *fls-subdegree f* ∗ *y* = *1*
— These assumptions imply x equals y, but no need to assume that.

shows *fls-right-inverse f y* ∗ *f* = *1*
⟨*proof*⟩

lemma *fls-right-inverse'*:
fixes *f* :: *'a::ring-1 fls*
assumes *x* ∗ *f* $$ *fls-subdegree f* = *1 f* $$ *fls-subdegree f* ∗ *y* = *1*
— These assumptions imply x equals y, but no need to assume that.
shows *f* ∗ *fls-left-inverse f x* = *1*
⟨*proof*⟩

lemma *fls-mult-left-inverse-base-factor*:
fixes *f* :: *'a::ring-1 fls*
assumes *x* ∗ (*f* $$ *fls-subdegree f*) = *1*
shows *fls-left-inverse* (*fls-base-factor f*) *x* ∗ *f* = *fls-X-intpow* (*fls-subdegree f*)
⟨*proof*⟩

lemma *fls-mult-right-inverse-base-factor*:
fixes *f* :: *'a::ring-1 fls*
assumes (*f* $$ *fls-subdegree f*) ∗ *y* = *1*
shows *f* ∗ *fls-right-inverse* (*fls-base-factor f*) *y* = *fls-X-intpow* (*fls-subdegree f*)
⟨*proof*⟩

lemma *fls-mult-inverse-base-factor*:
fixes *f* :: *'a::division-ring fls*
assumes *f* ≠ *0*
shows *f* ∗ *inverse* (*fls-base-factor f*) = *fls-X-intpow* (*fls-subdegree f*)
⟨*proof*⟩

lemma *fls-left-inverse-idempotent-ring1*:
fixes *f* :: *'a::ring-1 fls*
assumes *x* ∗ *f* $$ *fls-subdegree f* = *1 y* ∗ *x* = *1*
— These assumptions imply y equals *f* $$ *fls-subdegree f*, but no need to assume that.
shows *fls-left-inverse* (*fls-left-inverse f x*) *y* = *f*
⟨*proof*⟩

lemma *fls-left-inverse-idempotent-comm-ring1*:
fixes *f* :: *'a::comm-ring-1 fls*
assumes *x* ∗ *f* $$ *fls-subdegree f* = *1*
shows *fls-left-inverse* (*fls-left-inverse f x*) (*f* $$ *fls-subdegree f*) = *f*
⟨*proof*⟩

lemma *fls-right-inverse-idempotent-ring1*:
fixes *f* :: *'a::ring-1 fls*
assumes *f* $$ *fls-subdegree f* ∗ *x* = *1 x* ∗ *y* = *1*
— These assumptions imply y equals *f* $$ *fls-subdegree f*, but no need to assume that.
shows *fls-right-inverse* (*fls-right-inverse f x*) *y* = *f*
⟨*proof*⟩

lemma *fls-right-inverse-idempotent-comm-ring1*:
fixes f :: $'a$::*comm-ring-1 fls*
assumes f $$ *fls-subdegree* $f * x = 1$
shows *fls-right-inverse* (*fls-right-inverse* f x) (f $$ *fls-subdegree* f) = f
⟨*proof*⟩

lemma *fls-lr-inverse-unique-ring1*:
fixes f g :: $'a$ :: *ring-1 fls*
assumes *fg*: $f * g = 1$ g $$ *fls-subdegree* $g * f$ $$ *fls-subdegree* $f = 1$
shows *fls-left-inverse* g (f $$ *fls-subdegree* f) = f
and *fls-right-inverse* f (g $$ *fls-subdegree* g) = g
⟨*proof*⟩

lemma *fls-lr-inverse-unique-divring*:
fixes f g :: $'a$ ::*division-ring fls*
assumes *fg*: $f * g = 1$
shows *fls-left-inverse* g (f $$ *fls-subdegree* f) = f
and *fls-right-inverse* f (g $$ *fls-subdegree* g) = g
⟨*proof*⟩

lemma *fls-lr-inverse-minus*:
fixes f :: $'a$::*ring-1 fls*
shows *fls-left-inverse* $(-f)$ $(-x)$ = $-$ *fls-left-inverse* f x
and *fls-right-inverse* $(-f)$ $(-x)$ = $-$ *fls-right-inverse* f x
⟨*proof*⟩

lemma *fls-inverse-minus* [*simp*]: *inverse* $(-f)$ = $-$*inverse* (f :: $'a$ :: *division-ring fls*)
⟨*proof*⟩

lemma *fls-lr-inverse-mult-ring1*:
fixes f g :: $'a$::*ring-1 fls*
assumes x: $x * f$ $$ *fls-subdegree* $f = 1$ f $$ *fls-subdegree* $f * x = 1$
and y: $y * g$ $$ *fls-subdegree* $g = 1$ g $$ *fls-subdegree* $g * y = 1$
shows *fls-left-inverse* $(f * g)$ $(y*x)$ = *fls-left-inverse* g y $*$ *fls-left-inverse* f x
and *fls-right-inverse* $(f * g)$ $(y*x)$ = *fls-right-inverse* g y $*$ *fls-right-inverse* f x
⟨*proof*⟩

lemma *fls-lr-inverse-power-ring1*:
fixes f :: $'a$::*ring-1 fls*
assumes x: $x * f$ $$ *fls-subdegree* $f = 1$ f $$ *fls-subdegree* $f * x = 1$
shows *fls-left-inverse* $(f \widehat{\ } n)$ $(x \widehat{\ } n)$ = (*fls-left-inverse* f x) $\widehat{\ } n$
fls-right-inverse $(f \widehat{\ } n)$ $(x \widehat{\ } n)$ = (*fls-right-inverse* f x) $\widehat{\ } n$
⟨*proof*⟩

lemma *fls-divide-convert-times-inverse*:
fixes f g :: $'a$::{*comm-monoid-add*,*inverse*,*mult-zero*,*uminus*} *fls*

shows *f / g = f ∗ inverse g*
⟨*proof*⟩

instance *fls* :: (*division-ring*) *division-ring*
⟨*proof*⟩

lemma *fls-lr-inverse-mult-divring*:
fixes *f g* :: *'a::division-ring fls*
and *df dg* :: *int*
defines *df ≡ fls-subdegree f*
and *dg ≡ fls-subdegree g*
shows *fls-left-inverse (f∗g) (inverse ((f∗g)$$(df+dg))) = fls-left-inverse g (inverse (g$$dg)) ∗ fls-left-inverse f (inverse (f$$df))*
and *fls-right-inverse (f∗g) (inverse ((f∗g)$$(df+dg))) = fls-right-inverse g (inverse (g$$dg)) ∗ fls-right-inverse f (inverse (f$$df))*
⟨*proof*⟩

lemma *fls-lr-inverse-power-divring*:
fls-left-inverse (f ^ n) ((inverse (f $$ fls-subdegree f)) ^ n) = (fls-left-inverse f (inverse (f $$ fls-subdegree f))) ^ n (**is** *?P*)
and *fls-right-inverse (f ^ n) ((inverse (f $$ fls-subdegree f)) ^ n) = (fls-right-inverse f (inverse (f $$ fls-subdegree f))) ^ n* (**is** *?Q*)
for *f* :: *'a::division-ring fls*
⟨*proof*⟩

instance *fls* :: (*field*) *field*
⟨*proof*⟩

instance *fls* :: ({*field-prime-char*,*comm-semiring-1*}) *field-prime-char*
⟨*proof*⟩

7.5.6 Division

lemma *fls-divide-nth-below*:
fixes *f g* :: *'a::{comm-monoid-add,uminus,times,inverse} fls*
shows *n < fls-subdegree f − fls-subdegree g ⟹ (f div g) $$ n = 0*
⟨*proof*⟩

lemma *fls-divide-nth-base*:
fixes *f g* :: *'a::division-ring fls*
shows
(f div g) $$ (fls-subdegree f − fls-subdegree g) = f $$ fls-subdegree f / g $$ fls-subdegree g
⟨*proof*⟩

lemma *fls-div-zero* [*simp*]:
0 div (g :: 'a :: {comm-monoid-add,inverse,mult-zero,uminus} fls) = 0
⟨*proof*⟩

lemma *fls-div-by-zero*:
fixes g :: $'a$::{*comm-monoid-add*,*inverse*,*mult-zero*,*uminus*} *fls*
assumes *inverse* $(0::'a) = 0$
shows g *div* $0 = 0$
⟨*proof*⟩

lemma *fls-divide-times*:
fixes f g :: $'a$::{*semiring-0*,*inverse*,*uminus*} *fls*
shows $(f * g) / h = f * (g / h)$
⟨*proof*⟩

lemma *fls-divide-times2*:
fixes f g :: $'a$::{*comm-semiring-0*,*inverse*,*uminus*} *fls*
shows $(f * g) / h = (f / h) * g$
⟨*proof*⟩

lemma *fls-divide-subdegree-ge*:
fixes f g :: $'a$::{*comm-monoid-add*,*uminus*,*times*,*inverse*} *fls*
assumes $f / g \neq 0$
shows *fls-subdegree* $(f / g) \geq$ *fls-subdegree* f $-$ *fls-subdegree* g
⟨*proof*⟩

lemma *fls-divide-subdegree*:
fixes f g :: $'a$::*division-ring* *fls*
assumes $f \neq 0$ $g \neq 0$
shows *fls-subdegree* $(f / g) =$ *fls-subdegree* f $-$ *fls-subdegree* g
⟨*proof*⟩

lemma *fls-divide-shift-numer-nonzero*:
fixes f g :: $'a$ :: {*comm-monoid-add*,*inverse*,*times*,*uminus*} *fls*
assumes $f \neq 0$
shows *fls-shift* m f / g = *fls-shift* m (f/g)
⟨*proof*⟩

lemma *fls-divide-shift-numer*:
fixes f g :: $'a$ :: {*comm-monoid-add*,*inverse*,*mult-zero*,*uminus*} *fls*
shows *fls-shift* m f / g = *fls-shift* m (f/g)
⟨*proof*⟩

lemma *fls-divide-shift-denom-nonzero*:
fixes f g :: $'a$ :: {*comm-monoid-add*,*inverse*,*times*,*uminus*} *fls*
assumes $g \neq 0$
shows f / *fls-shift* m g = *fls-shift* $(-m)$ (f/g)
⟨*proof*⟩

lemma *fls-divide-shift-denom*:
fixes f g :: $'a$ :: *division-ring* *fls*
shows f / *fls-shift* m g = *fls-shift* $(-m)$ (f/g)
⟨*proof*⟩

lemma *fls-divide-shift-both-nonzero*:
fixes *f g* :: *'a* :: {*comm-monoid-add*,*inverse*,*times*,*uminus*} *fls*
assumes $f \neq 0$ $g \neq 0$
shows *fls-shift n f* / *fls-shift m g* = *fls-shift* $(n-m)$ (f/g)
⟨*proof*⟩

lemma *fls-divide-shift-both* [*simp*]:
fixes *f g* :: *'a* :: *division-ring fls*
shows *fls-shift n f* / *fls-shift m g* = *fls-shift* $(n-m)$ (f/g)
⟨*proof*⟩

lemma *fls-divide-base-factor-numer*:
fls-base-factor f / *g* = *fls-shift* (*fls-subdegree f*) (f/g)
⟨*proof*⟩

lemma *fls-divide-base-factor-denom*:
f / *fls-base-factor g* = *fls-shift* (−*fls-subdegree g*) (f/g)
⟨*proof*⟩

lemma *fls-divide-base-factor'*:
fls-base-factor f / *fls-base-factor g* = *fls-shift* (*fls-subdegree f* − *fls-subdegree g*) (f/g)
⟨*proof*⟩

lemma *fls-divide-base-factor*:
fixes *f g* :: *'a* :: *division-ring fls*
shows *fls-base-factor f* / *fls-base-factor g* = *fls-base-factor* (f/g)
⟨*proof*⟩

lemma *fls-divide-regpart*:
fixes *f g* :: *'a*::{*inverse*,*comm-monoid-add*,*uminus*,*mult-zero*} *fls*
assumes *fls-subdegree f* ≥ *0* *fls-subdegree g* ≥ *0*
shows *fls-regpart* (*f* / *g*) = *fls-regpart f* / *fls-regpart g*
⟨*proof*⟩

lemma *fls-divide-fls-base-factor-to-fps'*:
fixes *f g* :: *'a*::{*comm-monoid-add*,*uminus*,*inverse*,*mult-zero*} *fls*
shows
fls-base-factor-to-fps f / *fls-base-factor-to-fps g* =
fls-regpart (*fls-shift* (*fls-subdegree f* − *fls-subdegree g*) (*f* / *g*))
⟨*proof*⟩

lemma *fls-divide-fls-base-factor-to-fps*:
fixes *f g* :: *'a*::*division-ring fls*
shows *fls-base-factor-to-fps f* / *fls-base-factor-to-fps g* = *fls-base-factor-to-fps* (*f* / *g*)
⟨*proof*⟩

lemma *fls-divide-fps-to-fls*:
fixes *f g* :: *'a*::{*inverse*,*ab-group-add*,*mult-zero*} *fps*
assumes *subdegree f* ≥ *subdegree g*
shows *fps-to-fls f* / *fps-to-fls g* = *fps-to-fls* (*f*/*g*)
⟨*proof*⟩

lemma *fls-divide-1'*:
fixes *f* :: *'a*::{*comm-monoid-add*,*inverse*,*mult-zero*,*uminus*,*zero-neq-one*,*monoid-mult*} *fls*
assumes *inverse* (*1*::*'a*) = *1*
shows *f* / *1* = *f*
⟨*proof*⟩

lemma *fls-divide-1* [*simp*]: *a* / *1* = (*a*::*'a*::*division-ring fls*)
⟨*proof*⟩

lemma *fls-const-divide-const*:
fixes *x y* :: *'a*::*division-ring*
shows *fls-const x* / *fls-const y* = *fls-const* (*x*/*y*)
⟨*proof*⟩

lemma *fls-divide-X'*:
fixes *f* :: *'a*::{*comm-monoid-add*,*inverse*,*mult-zero*,*uminus*,*zero-neq-one*,*monoid-mult*} *fls*
assumes *inverse* (*1*::*'a*) = *1*
shows *f* / *fls-X* = *fls-shift 1 f*
⟨*proof*⟩

lemma *fls-divide-X* [*simp*]:
fixes *f* :: *'a*::*division-ring fls*
shows *f* / *fls-X* = *fls-shift 1 f*
⟨*proof*⟩

lemma *fls-divide-X-power'*:
fixes *f* :: *'a*::{*semiring-1*,*inverse*,*uminus*} *fls*
assumes *inverse* (*1*::*'a*) = *1*
shows *f* / (*fls-X* ^ *n*) = *fls-shift n f*
⟨*proof*⟩

lemma *fls-divide-X-power* [*simp*]:
fixes *f* :: *'a*::*division-ring fls*
shows *f* / (*fls-X* ^ *n*) = *fls-shift n f*
⟨*proof*⟩

lemma *fls-divide-X-inv'*:
fixes *f* :: *'a*::{*comm-monoid-add*,*inverse*,*mult-zero*,*uminus*,*zero-neq-one*,*monoid-mult*} *fls*
assumes *inverse* (*1*::*'a*) = *1*
shows *f* / *fls-X-inv* = *fls-shift* (−*1*) *f*

⟨*proof*⟩

lemma *fls-divide-X-inv* [*simp*]:
fixes *f* :: *'a::division-ring fls*
shows *f* / *fls-X-inv* = *fls-shift* (−*1*) *f*
⟨*proof*⟩

lemma *fls-divide-X-inv-power'*:
fixes *f* :: *'a::{semiring-1,inverse,uminus} fls*
assumes *inverse* (*1::'a*) = *1*
shows *f* / (*fls-X-inv* ^ *n*) = *fls-shift* (−*int n*) *f*
⟨*proof*⟩

lemma *fls-divide-X-inv-power* [*simp*]:
fixes *f* :: *'a::division-ring fls*
shows *f* / (*fls-X-inv* ^ *n*) = *fls-shift* (−*int n*) *f*
⟨*proof*⟩

lemma *fls-divide-X-intpow'*:
fixes *f* :: *'a::{semiring-1,inverse,uminus} fls*
assumes *inverse* (*1::'a*) = *1*
shows *f* / (*fls-X-intpow i*) = *fls-shift i f*
⟨*proof*⟩

lemma *fls-divide-X-intpow-conv-times'*:
fixes *f* :: *'a::{semiring-1,inverse,uminus} fls*
assumes *inverse* (*1::'a*) = *1*
shows *f* / (*fls-X-intpow i*) = *f* ∗ *fls-X-intpow* (−*i*)
⟨*proof*⟩

lemma *fls-divide-X-intpow*:
fixes *f* :: *'a::division-ring fls*
shows *f* / (*fls-X-intpow i*) = *fls-shift i f*
⟨*proof*⟩

lemma *fls-divide-X-intpow-conv-times*:
fixes *f* :: *'a::division-ring fls*
shows *f* / (*fls-X-intpow i*) = *f* ∗ *fls-X-intpow* (−*i*)
⟨*proof*⟩

lemma *fls-X-intpow-div-fls-X-intpow-semiring1*:
assumes *inverse* (*1::'a::{semiring-1,inverse,uminus}*) = *1*
shows (*fls-X-intpow i* :: *'a fls*) / *fls-X-intpow j* = *fls-X-intpow* (*i*−*j*)
⟨*proof*⟩

lemma *fls-X-intpow-div-fls-X-intpow*:
(*fls-X-intpow i* :: *'a::division-ring fls*) / *fls-X-intpow j* = *fls-X-intpow* (*i*−*j*)
⟨*proof*⟩

lemma *fls-divide-add*:
fixes *f g h* :: *'a*::{*semiring-0*,*inverse*,*uminus*} *fls*
shows (*f* + *g*) / *h* = *f* / *h* + *g* / *h*
⟨*proof*⟩

lemma *fls-divide-diff*:
fixes *f g h* :: *'a*::{*ring*,*inverse*} *fls*
shows (*f* − *g*) / *h* = *f* / *h* − *g* / *h*
⟨*proof*⟩

lemma *fls-divide-uminus*:
fixes *f g h* :: *'a*::{*ring*,*inverse*} *fls*
shows (− *f*) / *g* = − (*f* / *g*)
⟨*proof*⟩

lemma *fls-divide-uminus'*:
fixes *f g h* :: *'a*::*division-ring fls*
shows *f* / (− *g*) = − (*f* / *g*)
⟨*proof*⟩

7.5.7 Units

lemma *fls-is-left-unit-iff-base-is-left-unit*:
fixes *f* :: *'a* :: *ring-1-no-zero-divisors fls*
shows (∃ *g*. *1* = *f* ∗ *g*) ⟷ (∃ *k*. *1* = *f* $$ *fls-subdegree f* ∗ *k*)
⟨*proof*⟩

lemma *fls-is-right-unit-iff-base-is-right-unit*:
fixes *f* :: *'a* :: *ring-1-no-zero-divisors fls*
shows (∃ *g*. *1* = *g* ∗ *f*) ⟷ (∃ *k*. *1* = *k* ∗ *f* $$ *fls-subdegree f*)
⟨*proof*⟩

7.6 Composition

definition *fls-compose-fps* :: *'a* :: *field fls* ⇒ *'a fps* ⇒ *'a fls* **where**
fls-compose-fps F G =
fps-to-fls (*fps-compose* (*fls-base-factor-to-fps F*) *G*) ∗ *fps-to-fls G powi fls-subdegree F*

lemma *fps-compose-of-nat* [*simp*]: *fps-compose* (*of-nat n* :: *'a* :: *comm-ring-1 fps*) *H* = *of-nat n*
and *fps-compose-of-int* [*simp*]: *fps-compose* (*of-int i*) *H* = *of-int i*
⟨*proof*⟩

lemmas [*simp*] = *fps-to-fls-of-nat fps-to-fls-of-int*

lemma *fls-compose-fps-0* [*simp*]: *fls-compose-fps 0 H* = *0*
and *fls-compose-fps-1* [*simp*]: *fls-compose-fps 1 H* = *1*
and *fls-compose-fps-const* [*simp*]: *fls-compose-fps* (*fls-const c*) *H* = *fls-const c*
and *fls-compose-fps-of-nat* [*simp*]: *fls-compose-fps* (*of-nat n*) *H* = *of-nat n*

and *fls-compose-fps-of-int* [*simp*]: *fls-compose-fps* (*of-int i*) *H* = *of-int i*
and *fls-compose-fps-X* [*simp*]: *fls-compose-fps fls-X F* = *fps-to-fls F*
⟨*proof*⟩

lemma *fls-compose-fps-0-right*:
fls-compose-fps F 0 = (*if 0* ≤ *fls-subdegree F then fls-const* (*F* $$ *0*) *else 0*)
⟨*proof*⟩

lemma *fls-compose-fps-shift*:
assumes *H* ≠ *0*
shows *fls-compose-fps* (*fls-shift n F*) *H* = *fls-compose-fps F H* ∗ *fps-to-fls H powi* (−*n*)
⟨*proof*⟩

lemma *fls-compose-fps-to-fls* [*simp*]:
assumes [*simp*]: *G* ≠ *0 fps-nth G 0* = *0*
shows *fls-compose-fps* (*fps-to-fls F*) *G* = *fps-to-fls* (*fps-compose F G*)
⟨*proof*⟩

lemma *fls-compose-fps-mult*:
assumes [*simp*]: *H* ≠ *0 fps-nth H 0* = *0*
shows *fls-compose-fps* (*F* ∗ *G*) *H* = *fls-compose-fps F H* ∗ *fls-compose-fps G H*
⟨*proof*⟩

lemma *fls-compose-fps-power*:
assumes [*simp*]: *G* ≠ *0 fps-nth G 0* = *0*
shows *fls-compose-fps* (*F* ^ *n*) *G* = *fls-compose-fps F G* ^ *n*
⟨*proof*⟩

lemma *fls-compose-fps-add*:
assumes [*simp*]: *H* ≠ *0 fps-nth H 0* = *0*
shows *fls-compose-fps* (*F* + *G*) *H* = *fls-compose-fps F H* + *fls-compose-fps G H*
⟨*proof*⟩

lemma *fls-compose-fps-uminus* [*simp*]: *fls-compose-fps* (−*F*) *H* = −*fls-compose-fps F H*
⟨*proof*⟩

lemma *fls-compose-fps-diff*:
assumes [*simp*]: *H* ≠ *0 fps-nth H 0* = *0*
shows *fls-compose-fps* (*F* − *G*) *H* = *fls-compose-fps F H* − *fls-compose-fps G H*
⟨*proof*⟩

lemma *fps-compose-eq-0-iff*:
fixes *F G* :: ′*a* :: *idom fps*
assumes *fps-nth G 0* = *0*

shows *fps-compose F G = 0 ⟷ F = 0 ∨ (G = 0 ∧ fps-nth F 0 = 0)*
⟨*proof*⟩

lemma *fls-compose-fps-eq-0-iff*:
assumes *H ≠ 0 fps-nth H 0 = 0*
shows *fls-compose-fps F H = 0 ⟷ F = 0*
⟨*proof*⟩

lemma *fls-compose-fps-inverse*:
assumes [*simp*]: *H ≠ 0 fps-nth H 0 = 0*
shows *fls-compose-fps (inverse F) H = inverse (fls-compose-fps F H)*
⟨*proof*⟩

lemma *fls-compose-fps-divide*:
assumes [*simp*]: *H ≠ 0 fps-nth H 0 = 0*
shows *fls-compose-fps (F / G) H = fls-compose-fps F H / fls-compose-fps G H*
⟨*proof*⟩

lemma *fls-compose-fps-powi*:
assumes [*simp*]: *H ≠ 0 fps-nth H 0 = 0*
shows *fls-compose-fps (F powi n) H = fls-compose-fps F H powi n*
⟨*proof*⟩

lemma *fls-compose-fps-assoc*:
assumes [*simp*]: *G ≠ 0 fps-nth G 0 = 0 H ≠ 0 fps-nth H 0 = 0*
shows *fls-compose-fps (fls-compose-fps F G) H = fls-compose-fps F (fps-compose G H)*
⟨*proof*⟩

lemma *subdegree-pos-iff*: *subdegree F > 0 ⟷ F ≠ 0 ∧ fps-nth F 0 = 0*
⟨*proof*⟩

lemma *fls-X-power-int* [*simp*]: *fls-X powi n = (fls-X-intpow n :: 'a :: division-ring fls)*
⟨*proof*⟩

lemma *fls-const-power-int*: *fls-const (c powi n) = fls-const (c :: 'a :: division-ring) powi n*
⟨*proof*⟩

lemma *fls-nth-fls-compose-fps-linear*:
fixes *c :: 'a :: field*
assumes [*simp*]: *c ≠ 0*
shows *fls-compose-fps F (fps-const c * fps-X) $$ n = F $$ n * c powi n*
⟨*proof*⟩

lemma *fls-const-transfer* [*transfer-rule*]:
rel-fun (=) (pcr-fls (=))

(*λc n. if n = 0 then c else 0*) *fls-const*
⟨*proof*⟩

lemma *fls-shift-transfer* [*transfer-rule*]:
rel-fun (=) (*rel-fun* (*pcr-fls* (=)) (*pcr-fls* (=)))
(*λn f k. f* (*k*+*n*)) *fls-shift*
⟨*proof*⟩

lift-definition *fls-compose-power* :: *'a* :: *zero fls* ⇒ *nat* ⇒ *'a fls* **is**
λf d n. if d > 0 ∧ int d dvd n then f (*n div int d*) *else 0*
⟨*proof*⟩

lemma *fls-nth-compose-power*:
assumes *d > 0*
shows *fls-compose-power f d* $$ *n* = (*if int d dvd n then f* $$ (*n div int d*) *else 0*)
⟨*proof*⟩

lemma *fls-compose-power-0-left* [*simp*]: *fls-compose-power 0 d = 0*
⟨*proof*⟩

lemma *fls-compose-power-1-left* [*simp*]: *d > 0* ⟹ *fls-compose-power 1 d = 1*
⟨*proof*⟩

lemma *fls-compose-power-const-left* [*simp*]:
d > 0 ⟹ *fls-compose-power* (*fls-const c*) *d = fls-const c*
⟨*proof*⟩

lemma *fls-compose-power-shift* [*simp*]:
d > 0 ⟹ *fls-compose-power* (*fls-shift n f*) *d = fls-shift* (*d ∗ n*) (*fls-compose-power f d*)
⟨*proof*⟩

lemma *fls-compose-power-X-intpow* [*simp*]:
d > 0 ⟹ *fls-compose-power* (*fls-X-intpow n*) *d = fls-X-intpow* (*int d ∗ n*)
⟨*proof*⟩

lemma *fls-compose-power-X* [*simp*]:
d > 0 ⟹ *fls-compose-power fls-X d = fls-X-intpow* (*int d*)
⟨*proof*⟩

lemma *fls-compose-power-X-inv* [*simp*]:
d > 0 ⟹ *fls-compose-power fls-X-inv d = fls-X-intpow* (−*int d*)
⟨*proof*⟩

lemma *fls-compose-power-0-right* [*simp*]: *fls-compose-power f 0 = 0*
⟨*proof*⟩

lemma *fls-compose-power-add* [*simp*]:
fls-compose-power (*f* + *g*) *d* = *fls-compose-power f d* + *fls-compose-power g d*
⟨*proof*⟩

lemma *fls-compose-power-diff* [*simp*]:
fls-compose-power (*f* − *g*) *d* = *fls-compose-power f d* − *fls-compose-power g d*
⟨*proof*⟩

lemma *fls-compose-power-uminus* [*simp*]:
fls-compose-power (−*f*) *d* = −*fls-compose-power f d*
⟨*proof*⟩

lemma *fps-nth-compose-X-power*:
fps-nth (*f oo* (*fps-X* ^ *d*)) *n* = (*if d dvd n then fps-nth f* (*n div d*) *else 0*)
⟨*proof*⟩

lemma *fls-compose-power-fps-to-fls*:
assumes *d* > *0*
shows *fls-compose-power* (*fps-to-fls f*) *d* = *fps-to-fls* (*fps-compose f* (*fps-X* ^ *d*))
⟨*proof*⟩

lemma *fls-compose-power-mult* [*simp*]:
fls-compose-power (*f* ∗ *g* :: ′*a* :: *idom fls*) *d* = *fls-compose-power f d* ∗ *fls-compose-power g d*
⟨*proof*⟩

lemma *fls-compose-power-power* [*simp*]:
assumes *d* > *0* ∨ *n* > *0*
shows *fls-compose-power* (*f* ^ *n* :: ′*a* :: *idom fls*) *d* = *fls-compose-power f d* ^ *n*
⟨*proof*⟩

lemma *fls-nth-compose-power*′ [*simp*]:
d = *0* ∨ ¬*d dvd n* ⟹ *fls-compose-power f d* $$ *int n* = *0*
d dvd n ⟹ *d* > *0* ⟹ *fls-compose-power f d* $$ *int n* = *f* $$ *int* (*n div d*)
⟨*proof*⟩

7.7 Formal differentiation and integration

7.7.1 Derivative

definition *fls-deriv f* = *Abs-fls* (λ*n*. *of-int* (*n*+*1*) ∗ *f*$$(*n*+*1*))

lemma *fls-deriv-nth*[*simp*]: *fls-deriv f* $$ *n* = *of-int* (*n*+*1*) ∗ *f*$$(*n*+*1*)
⟨*proof*⟩

lemma *fls-deriv-residue*: *fls-deriv f* $$ −*1* = *0*
⟨*proof*⟩

lemma *fls-deriv-const*[*simp*]: *fls-deriv* (*fls-const x*) = *0*

⟨proof⟩

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

lemma *fls-deriv-of-int*[*simp*]: *fls-deriv* (*of-int i*) = *0*
⟨proof⟩

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

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

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

lemma *fls-deriv-subdegree′*:
assumes *of-int* (*fls-subdegree f*) ∗ *f* $$ *fls-subdegree f* ≠ *0*
shows *fls-subdegree* (*fls-deriv f*) = *fls-subdegree f* − *1*
⟨proof⟩

lemma *fls-deriv-subdegree0*:
assumes *fls-subdegree f* = *0*
shows *fls-subdegree* (*fls-deriv f*) ≥ *0*
⟨proof⟩

lemma *fls-subdegree-deriv′*:
fixes *f* :: ′*a*::*ring-1-no-zero-divisors fls*
assumes (*of-int* (*fls-subdegree f*) :: ′*a*) ≠ *0*
shows *fls-subdegree* (*fls-deriv f*) = *fls-subdegree f* − *1*
⟨proof⟩

lemma *fls-subdegree-deriv*:
fixes *f* :: ′*a*::{*ring-1-no-zero-divisors*,*ring-char-0*} *fls*
assumes *fls-subdegree f* ≠ *0*
shows *fls-subdegree* (*fls-deriv f*) = *fls-subdegree f* − *1*
⟨proof⟩

Shifting is like multiplying by a power of the implied variable, and so satisfies a product-like rule.

lemma *fls-deriv-shift*:
fls-deriv (*fls-shift n f*) = *of-int* (−*n*) ∗ *fls-shift* (*n*+*1*) *f* + *fls-shift n* (*fls-deriv f*)
⟨proof⟩

lemma *fls-deriv-X* [*simp*]: *fls-deriv fls-X* = *1*
⟨proof⟩

lemma *fls-deriv-X-inv* [*simp*]: *fls-deriv fls-X-inv* = − (*fls-X-inv*^{2})

⟨*proof*⟩

lemma *fls-deriv-delta*:
fls-deriv (*Abs-fls* (*λn. if n=m then c else 0*)) =
Abs-fls (*λn. if n=m−1 then of-int m ∗ c else 0*)
⟨*proof*⟩

lemma *fls-deriv-base-factor*:
fls-deriv (*fls-base-factor f*) =
of-int (−*fls-subdegree f*) ∗ *fls-shift* (*fls-subdegree f* + *1*) *f* +
fls-shift (*fls-subdegree f*) (*fls-deriv f*)
⟨*proof*⟩

lemma *fls-regpart-deriv*: *fls-regpart* (*fls-deriv f*) = *fps-deriv* (*fls-regpart f*)
⟨*proof*⟩

lemma *fls-prpart-deriv*:
fixes *f* :: *′a* :: {*comm-ring-1*,*ring-no-zero-divisors*} *fls*
— Commutivity and no zero divisors are required by the definition of *pderiv*.
shows *fls-prpart* (*fls-deriv f*) = − *pCons 0* (*pCons 0* (*pderiv* (*fls-prpart f*)))
⟨*proof*⟩

lemma *pderiv-fls-prpart*:
pderiv (*fls-prpart f*) = − *poly-shift 2* (*fls-prpart* (*fls-deriv f*))
⟨*proof*⟩

lemma *fls-deriv-fps-to-fls*: *fls-deriv* (*fps-to-fls f*) = *fps-to-fls* (*fps-deriv f*)
⟨*proof*⟩

7.7.2 Algebraic rules of the derivative

lemma *fls-deriv-add* [*simp*]: *fls-deriv* (*f*+*g*) = *fls-deriv f* + *fls-deriv g*
⟨*proof*⟩

lemma *fls-deriv-sub* [*simp*]: *fls-deriv* (*f*−*g*) = *fls-deriv f* − *fls-deriv g*
⟨*proof*⟩

lemma *fls-deriv-neg* [*simp*]: *fls-deriv* (−*f*) = − *fls-deriv f*
⟨*proof*⟩

lemma *fls-deriv-mult* [*simp*]:
fls-deriv (*f*∗*g*) = *f* ∗ *fls-deriv g* + *fls-deriv f* ∗ *g*
⟨*proof*⟩

lemma *fls-deriv-mult-const-left*:
fls-deriv (*fls-const c* ∗ *f*) = *fls-const c* ∗ *fls-deriv f*
⟨*proof*⟩

lemma *fls-deriv-linear*:

fls-deriv (*fls-const a* ∗ *f* + *fls-const b* ∗ *g*) =
fls-const a ∗ *fls-deriv f* + *fls-const b* ∗ *fls-deriv g*
⟨*proof*⟩

lemma *fls-deriv-mult-const-right*:
fls-deriv (*f* ∗ *fls-const c*) = *fls-deriv f* ∗ *fls-const c*
⟨*proof*⟩

lemma *fls-deriv-linear2*:
fls-deriv (*f* ∗ *fls-const a* + *g* ∗ *fls-const b*) =
fls-deriv f ∗ *fls-const a* + *fls-deriv g* ∗ *fls-const b*
⟨*proof*⟩

lemma *fls-deriv-sum*:
fls-deriv (*sum f S*) = *sum* (λ*i*. *fls-deriv* (*f i*)) *S*
⟨*proof*⟩

lemma *fls-deriv-power*:
fixes *f* :: *'a::comm-ring-1 fls*
shows *fls-deriv* (*f*^*n*) = *of-nat n* ∗ *f*^(*n*−*1*) ∗ *fls-deriv f*
⟨*proof*⟩

lemma *fls-deriv-X-power*:
fls-deriv (*fls-X* ^ *n*) = *of-nat n* ∗ *fls-X* ^ (*n*−*1*)
⟨*proof*⟩

lemma *fls-deriv-X-inv-power*:
fls-deriv (*fls-X-inv* ^ *n*) = − *of-nat n* ∗ *fls-X-inv* ^ (*Suc n*)
⟨*proof*⟩

lemma *fls-deriv-X-intpow*:
fls-deriv (*fls-X-intpow i*) = *of-int i* ∗ *fls-X-intpow* (*i*−*1*)
⟨*proof*⟩

lemma *fls-deriv-lr-inverse*:
assumes *x* ∗ *f* $$ *fls-subdegree f* = *1* *f* $$ *fls-subdegree f* ∗ *y* = *1*
— These assumptions imply x equals y, but no need to assume that.
shows *fls-deriv* (*fls-left-inverse f x*) =
− *fls-left-inverse f x* ∗ *fls-deriv f* ∗ *fls-left-inverse f x*
and *fls-deriv* (*fls-right-inverse f y*) =
− *fls-right-inverse f y* ∗ *fls-deriv f* ∗ *fls-right-inverse f y*
⟨*proof*⟩

lemma *fls-deriv-lr-inverse-comm*:
fixes *x y* :: *'a::comm-ring-1*
assumes *x* ∗ *f* $$ *fls-subdegree f* = *1*
shows *fls-deriv* (*fls-left-inverse f x*) = − *fls-deriv f* ∗ (*fls-left-inverse f x*)^2
and *fls-deriv* (*fls-right-inverse f x*) = − *fls-deriv f* ∗ (*fls-right-inverse f x*)^2
⟨*proof*⟩

lemma *fls-inverse-deriv-divring*:
fixes *a* :: *'a::division-ring fls*
shows *fls-deriv* (*inverse a*) = − *inverse a* ∗ *fls-deriv a* ∗ *inverse a*
⟨*proof*⟩

lemma *fls-inverse-deriv*:
fixes *a* :: *'a::field fls*
shows *fls-deriv* (*inverse a*) = − *fls-deriv a* ∗ (*inverse a*)^2
⟨*proof*⟩

lemma *fls-inverse-deriv'*:
fixes *a* :: *'a::field fls*
shows *fls-deriv* (*inverse a*) = − *fls-deriv a* / *a*^{2}
⟨*proof*⟩

7.7.3 Equality of derivatives

lemma *fls-deriv-eq-0-iff*:
fls-deriv f = *0* ⟷ *f* = *fls-const* (*f*$$*0* :: *'a::{ring-1-no-zero-divisors,ring-char-0}*)
⟨*proof*⟩

lemma *fls-deriv-eq-iff*:
fixes *f g* :: *'a::{ring-1-no-zero-divisors,ring-char-0} fls*
shows *fls-deriv f* = *fls-deriv g* ⟷ (*f* = *fls-const*(*f*$$*0* − *g*$$*0*) + *g*)
⟨*proof*⟩

lemma *fls-deriv-eq-iff-ex*:
fixes *f g* :: *'a::{ring-1-no-zero-divisors,ring-char-0} fls*
shows (*fls-deriv f* = *fls-deriv g*) ⟷ (∃ *c*. *f* = *fls-const c* + *g*)
⟨*proof*⟩

7.7.4 Residues

definition *fls-residue-def*[*simp*]: *fls-residue f* ≡ *f* $$ −*1*

lemma *fls-residue-deriv*: *fls-residue* (*fls-deriv f*) = *0*
⟨*proof*⟩

lemma *fls-residue-add*: *fls-residue* (*f*+*g*) = *fls-residue f* + *fls-residue g*
⟨*proof*⟩

lemma *fls-residue-times-deriv*:
fls-residue (*fls-deriv f* ∗ *g*) = − *fls-residue* (*f* ∗ *fls-deriv g*)
⟨*proof*⟩

lemma *fls-residue-power-series*: *fls-subdegree f* ≥ *0* ⟹ *fls-residue f* = *0*
⟨*proof*⟩

lemma *fls-residue-fls-X-intpow*:

fls-residue (*fls-X-intpow i*) = (*if i*=−*1 then 1 else 0*)
⟨*proof*⟩

lemma *fls-residue-shift-nth*:
fixes *f* :: ′*a*::*semiring-1 fls*
shows *f*$$*n* = *fls-residue* (*fls-X-intpow* (−*n*−*1*) ∗ *f*)
⟨*proof*⟩

lemma *fls-residue-fls-const-times*:
fixes *f* :: ′*a*::{*comm-monoid-add*, *mult-zero*} *fls*
shows *fls-residue* (*fls-const c* ∗ *f*) = *c* ∗ *fls-residue f*
and *fls-residue* (*f* ∗ *fls-const c*) = *fls-residue f* ∗ *c*
⟨*proof*⟩

lemma *fls-residue-of-int-times*:
fixes *f* :: ′*a*::*ring-1 fls*
shows *fls-residue* (*of-int i* ∗ *f*) = *of-int i* ∗ *fls-residue f*
and *fls-residue* (*f* ∗ *of-int i*) = *fls-residue f* ∗ *of-int i*
⟨*proof*⟩

lemma *fls-residue-deriv-times-lr-inverse-eq-subdegree*:
fixes *f g* :: ′*a*::*ring-1 fls*
assumes *y* ∗ (*f* $$ *fls-subdegree f*) = *1* (*f* $$ *fls-subdegree f*) ∗ *y* = *1*
shows *fls-residue* (*fls-deriv f* ∗ *fls-right-inverse f y*) = *of-int* (*fls-subdegree f*)
and *fls-residue* (*fls-deriv f* ∗ *fls-left-inverse f y*) = *of-int* (*fls-subdegree f*)
and *fls-residue* (*fls-left-inverse f y* ∗ *fls-deriv f*) = *of-int* (*fls-subdegree f*)
and *fls-residue* (*fls-right-inverse f y* ∗ *fls-deriv f*) = *of-int* (*fls-subdegree f*)
⟨*proof*⟩

lemma *fls-residue-deriv-times-inverse-eq-subdegree*:
fixes *f g* :: ′*a*::*division-ring fls*
shows *fls-residue* (*fls-deriv f* ∗ *inverse f*) = *of-int* (*fls-subdegree f*)
and *fls-residue* (*inverse f* ∗ *fls-deriv f*) = *of-int* (*fls-subdegree f*)
⟨*proof*⟩

7.7.5 Integral definition and basic properties

definition *fls-integral* :: ′*a*::{*ring-1*,*inverse*} *fls* ⇒ ′*a fls*
where *fls-integral a* = *Abs-fls* (λ*n*. *if n*=*0 then 0 else inverse* (*of-int n*) ∗ *a*$$(*n* − *1*))

lemma *fls-integral-nth* [*simp*]:
fls-integral a $$ *n* = (*if n*=*0 then 0 else inverse* (*of-int n*) ∗ *a*$$(*n*−*1*))
⟨*proof*⟩

lemma *fls-integral-conv-fps-zeroth-integral*:
assumes *fls-subdegree a* ≥ *0*
shows *fls-integral a* = *fps-to-fls* (*fps-integral0* (*fls-regpart a*))
⟨*proof*⟩

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

lemma *fls-integral-const′*:
fixes *x* :: ′*a*::{*ring-1*,*inverse*}
assumes *inverse* (*1*::′*a*) = *1*
shows *fls-integral* (*fls-const x*) = *fls-const x* ∗ *fls-X*
⟨*proof*⟩

lemma *fls-integral-const*:
fixes *x* :: ′*a*::*division-ring*
shows *fls-integral* (*fls-const x*) = *fls-const x* ∗ *fls-X*
⟨*proof*⟩

lemma *fls-integral-of-nat′*:
assumes *inverse* (*1*::′*a*::{*ring-1*,*inverse*}) = *1*
shows *fls-integral* (*of-nat n* :: ′*a fls*) = *of-nat n* ∗ *fls-X*
⟨*proof*⟩

lemma *fls-integral-of-nat*:
fls-integral (*of-nat n* :: ′*a*::*division-ring fls*) = *of-nat n* ∗ *fls-X*
⟨*proof*⟩

lemma *fls-integral-of-int′*:
assumes *inverse* (*1*::′*a*::{*ring-1*,*inverse*}) = *1*
shows *fls-integral* (*of-int i* :: ′*a fls*) = *of-int i* ∗ *fls-X*
⟨*proof*⟩

lemma *fls-integral-of-int*:
fls-integral (*of-int i* :: ′*a*::*division-ring fls*) = *of-int i* ∗ *fls-X*
⟨*proof*⟩

lemma *fls-integral-one′*:
assumes *inverse* (*1*::′*a*::{*ring-1*,*inverse*}) = *1*
shows *fls-integral* (*1*::′*a fls*) = *fls-X*
⟨*proof*⟩

lemma *fls-integral-one*: *fls-integral* (*1*::′*a*::*division-ring fls*) = *fls-X*
⟨*proof*⟩

lemma *fls-subdegree-integral-ge*:
fls-integral f ≠ *0* ⟹ *fls-subdegree* (*fls-integral f*) ≥ *fls-subdegree f* + *1*
⟨*proof*⟩

lemma *fls-subdegree-integral*:
fixes *f* :: ′*a*::{*division-ring*,*ring-char-0*} *fls*
assumes *f* ≠ *0 fls-subdegree f* ≠ −*1*
shows *fls-subdegree* (*fls-integral f*) = *fls-subdegree f* + *1*

⟨proof⟩

lemma *fls-integral-X* [*simp*]:
fls-integral (*fls-X*::′*a*::{*ring-1*,*inverse*} *fls*) =
fls-const (*inverse* (*of-int 2*)) ∗ *fls-X*^{2}
⟨proof⟩

lemma *fls-integral-X-power*:
fls-integral (*fls-X* ^ *n* ::′*a* :: {*ring-1*,*inverse*} *fls*) =
fls-const (*inverse* (*of-nat* (*Suc n*))) ∗ *fls-X* ^ *Suc n*
⟨proof⟩

lemma *fls-integral-X-power-char0*:
fls-integral (*fls-X* ^ *n* :: ′*a* :: {*ring-char-0*,*inverse*} *fls*) =
inverse (*of-nat* (*Suc n*)) ∗ *fls-X* ^ *Suc n*
⟨proof⟩

lemma *fls-integral-X-inv* [*simp*]: *fls-integral* (*fls-X-inv*::′*a*::{*ring-1*,*inverse*} *fls*) =
0
⟨proof⟩

lemma *fls-integral-X-inv-power*:
assumes *n* ≥ *2*
shows
fls-integral (*fls-X-inv* ^ *n* :: ′*a* :: {*ring-1*,*inverse*} *fls*) =
fls-const (*inverse* (*of-int* (*1* − *int n*))) ∗ *fls-X-inv* ^ (*n*−*1*)
⟨proof⟩

lemma *fls-integral-X-inv-power-char0*:
assumes *n* ≥ *2*
shows
fls-integral (*fls-X-inv* ^ *n* :: ′*a* :: {*ring-char-0*,*inverse*} *fls*) =
inverse (*of-int* (*1* − *int n*)) ∗ *fls-X-inv* ^ (*n*−*1*)
⟨proof⟩

lemma *fls-integral-X-inv-power′*:
assumes *n* ≥ *1*
shows
fls-integral (*fls-X-inv* ^ *n* :: ′*a* :: *division-ring fls*) =
− *fls-const* (*inverse* (*of-nat* (*n*−*1*))) ∗ *fls-X-inv* ^ (*n*−*1*)
⟨proof⟩

lemma *fls-integral-X-inv-power-char0′*:
assumes *n* ≥ *1*
shows
fls-integral (*fls-X-inv* ^ *n* :: ′*a* :: {*division-ring*,*ring-char-0*} *fls*) =
− *inverse* (*of-nat* (*n*−*1*)) ∗ *fls-X-inv* ^ (*n*−*1*)
⟨proof⟩

lemma *fls-integral-delta*:
assumes $m \neq -1$
shows
fls-integral (*Abs-fls* (λ*n. if n=m then c else 0*)) =
Abs-fls (λ*n. if n=m+1 then inverse* (*of-int* (*m+1*)) $*$ *c else 0*)
⟨*proof*⟩

lemma *fls-regpart-integral*:
fls-regpart (*fls-integral f*) = *fps-integral0* (*fls-regpart f*)
⟨*proof*⟩

lemma *fls-integral-fps-to-fls*:
fls-integral (*fps-to-fls f*) = *fps-to-fls* (*fps-integral0 f*)
⟨*proof*⟩

7.7.6 Algebraic rules of the integral

lemma *fls-integral-add* [*simp*]: *fls-integral* (*f+g*) = *fls-integral f* + *fls-integral g*
⟨*proof*⟩

lemma *fls-integral-sub* [*simp*]: *fls-integral* (*f−g*) = *fls-integral f* − *fls-integral g*
⟨*proof*⟩

lemma *fls-integral-neg* [*simp*]: *fls-integral* (−*f*) = − *fls-integral f*
⟨*proof*⟩

lemma *fls-integral-mult-const-left*:
fls-integral (*fls-const c* $*$ *f*) = *fls-const c* $*$ *fls-integral* (*f* :: ′*a*::*division-ring fls*)
⟨*proof*⟩

lemma *fls-integral-mult-const-left-comm*:
fixes *f* :: ′*a*::{*comm-ring-1*,*inverse*} *fls*
shows *fls-integral* (*fls-const c* $*$ *f*) = *fls-const c* $*$ *fls-integral f*
⟨*proof*⟩

lemma *fls-integral-linear*:
fixes *f g* :: ′*a*::*division-ring fls*
shows
fls-integral (*fls-const a* $*$ *f* + *fls-const b* $*$ *g*) =
fls-const a $*$ *fls-integral f* + *fls-const b* $*$ *fls-integral g*
⟨*proof*⟩

lemma *fls-integral-linear-comm*:
fixes *f g* :: ′*a*::{*comm-ring-1*,*inverse*} *fls*
shows
fls-integral (*fls-const a* $*$ *f* + *fls-const b* $*$ *g*) =
fls-const a $*$ *fls-integral f* + *fls-const b* $*$ *fls-integral g*
⟨*proof*⟩

lemma *fls-integral-mult-const-right*:
fls-integral (*f* ∗ *fls-const c*) = *fls-integral f* ∗ *fls-const c*
⟨*proof*⟩

lemma *fls-integral-linear2*:
fls-integral (*f* ∗ *fls-const a* + *g* ∗ *fls-const b*) =
fls-integral f ∗ *fls-const a* + *fls-integral g* ∗ *fls-const b*
⟨*proof*⟩

lemma *fls-integral-sum*:
fls-integral (*sum f S*) = *sum* (λ*i*. *fls-integral* (*f i*)) *S*
⟨*proof*⟩

7.7.7 Derivatives of integrals and vice versa

lemma *fls-integral-fls-deriv*:
fixes *a* :: ′*a*::{*division-ring*,*ring-char-0*} *fls*
shows *fls-integral* (*fls-deriv a*) + *fls-const* (*a*$$*0*) = *a*
⟨*proof*⟩

lemma *fls-deriv-fls-integral*:
fixes *a* :: ′*a*::{*division-ring*,*ring-char-0*} *fls*
assumes *fls-residue a* = *0*
shows *fls-deriv* (*fls-integral a*) = *a*
⟨*proof*⟩

Series with zero residue are precisely the derivatives.

lemma *fls-residue-nonzero-ex-antiderivative*:
fixes *f* :: ′*a*::{*division-ring*,*ring-char-0*} *fls*
assumes *fls-residue f* = *0*
shows ∃ *F*. *fls-deriv F* = *f*
⟨*proof*⟩

lemma *fls-ex-antiderivative-residue-nonzero*:
assumes ∃ *F*. *fls-deriv F* = *f*
shows *fls-residue f* = *0*
⟨*proof*⟩

lemma *fls-residue-nonzero-ex-anitderivative-iff*:
fixes *f* :: ′*a*::{*division-ring*,*ring-char-0*} *fls*
shows *fls-residue f* = *0* ⟷ (∃ *F*. *fls-deriv F* = *f*)
⟨*proof*⟩

7.8 Topology

instantiation *fls* :: (*group-add*) *metric-space*
begin

definition

dist-fls-def:
dist (*a* :: *'a fls*) *b* =
(*if a* = *b*
then 0
else if fls-subdegree (*a*−*b*) ≥ *0*
then inverse (*2* ^ *nat* (*fls-subdegree* (*a*−*b*)))
else 2 ^ *nat* (−*fls-subdegree* (*a*−*b*))
)

lemma *dist-fls-ge0*: *dist* (*a* :: *'a fls*) *b* ≥ *0*
⟨*proof*⟩

definition *uniformity-fls-def* [*code del*]:
(*uniformity* :: (*'a fls* × *'a fls*) *filter*) = (*INF e* ∈ {*0* <..}. *principal* {(*x*, *y*). *dist x y* < *e*})

definition *open-fls-def'* [*code del*]:
open (*U* :: *'a fls set*) ⟷ (∀ *x*∈*U*. *eventually* (λ(*x'*, *y*). *x'* = *x* ⟶ *y* ∈ *U*) *uniformity*)

lemma *dist-fls-sym*: *dist* (*a* :: *'a fls*) *b* = *dist b a*
⟨*proof*⟩

context
begin

private lemma *instance-helper*:
fixes *a b c* :: *'a fls*
assumes *neq*: *a*≠*b a*≠*c*
and *dist-ineq*: *dist a b* > *dist a c*
shows *fls-subdegree* (*a* − *b*) < *fls-subdegree* (*a* − *c*)
⟨*proof*⟩

instance
⟨*proof*⟩

end
end

declare *uniformity-Abort*[**where** *'a*=*'a* :: *group-add fls*, *code*]

lemma *open-fls-def*:
open (*S* :: *'a*::*group-add fls set*) = (∀ *a* ∈ *S*. ∃ *r*. *r* >*0* ∧ {*y*. *dist y a* < *r*} ⊆ *S*)
⟨*proof*⟩

7.9 Notation

no-notation *fls-nth* (**infixl** $$ *75*)

bundle *fls-notation*
begin
notation *fls-nth* (**infixl** $$ *75*)
end

end

8 The fraction field of any integral domain

theory *Fraction-Field*
imports *Main*
begin

8.1 General fractions construction

8.1.1 Construction of the type of fractions

context *idom* **begin**

definition *fractrel* :: $'a \times 'a \Rightarrow 'a * 'a \Rightarrow bool$ **where**
$fractrel = (\lambda x\ y.\ snd\ x \neq 0 \wedge snd\ y \neq 0 \wedge fst\ x * snd\ y = fst\ y * snd\ x)$

lemma *fractrel-iff* [*simp*]:
$fractrel\ x\ y \longleftrightarrow snd\ x \neq 0 \wedge snd\ y \neq 0 \wedge fst\ x * snd\ y = fst\ y * snd\ x$
⟨*proof*⟩

lemma *symp-fractrel*: *symp fractrel*
⟨*proof*⟩

lemma *transp-fractrel*: *transp fractrel*
⟨*proof*⟩

lemma *part-equivp-fractrel*: *part-equivp fractrel*
⟨*proof*⟩

end

quotient-type (**overloaded**) $'a\ fract = 'a :: idom \times 'a\ /\ partial$: *fractrel*
⟨*proof*⟩

8.1.2 Representation and basic operations

lift-definition *Fract* :: $'a :: idom \Rightarrow 'a \Rightarrow 'a\ fract$
is $\lambda a\ b.\ if\ b = 0\ then\ (0,\ 1)\ else\ (a,\ b)$
⟨*proof*⟩

lemma *Fract-cases* [*cases type*: *fract*]:
obtains (*Fract*) *a b* **where** $q = Fract\ a\ b\ b \neq 0$
⟨*proof*⟩

lemma *Fract-induct* [*case-names Fract*, *induct type*: *fract*]:
$(\bigwedge a\ b.\ b \neq 0 \Longrightarrow P\ (Fract\ a\ b)) \Longrightarrow P\ q$
⟨*proof*⟩

lemma *eq-fract*:
shows $\bigwedge a\ b\ c\ d.\ b \neq 0 \Longrightarrow d \neq 0 \Longrightarrow Fract\ a\ b = Fract\ c\ d \longleftrightarrow a * d = c * b$
and $\bigwedge a.\ Fract\ a\ 0 = Fract\ 0\ 1$
and $\bigwedge a\ c.\ Fract\ 0\ a = Fract\ 0\ c$
⟨*proof*⟩

instantiation *fract* :: (*idom*) *comm-ring-1*
begin

lift-definition *zero-fract* :: *'a fract* **is** (*0*, *1*) ⟨*proof*⟩

lemma *Zero-fract-def*: $0 = Fract\ 0\ 1$
⟨*proof*⟩

lift-definition *one-fract* :: *'a fract* **is** (*1*, *1*) ⟨*proof*⟩

lemma *One-fract-def*: $1 = Fract\ 1\ 1$
⟨*proof*⟩

lift-definition *plus-fract* :: *'a fract* ⇒ *'a fract* ⇒ *'a fract*
is $\lambda q\ r.\ (fst\ q * snd\ r + fst\ r * snd\ q,\ snd\ q * snd\ r)$
⟨*proof*⟩

lemma *add-fract* [*simp*]:
$[\![\ b \neq 0;\ d \neq 0\ ]\!] \Longrightarrow Fract\ a\ b + Fract\ c\ d = Fract\ (a * d + c * b)\ (b * d)$
⟨*proof*⟩

lift-definition *uminus-fract* :: *'a fract* ⇒ *'a fract*
is $\lambda x.\ (-\ fst\ x,\ snd\ x)$
⟨*proof*⟩

lemma *minus-fract* [*simp*]:
fixes *a b* :: *'a::idom*
shows $-\ Fract\ a\ b = Fract\ (-\ a)\ b$
⟨*proof*⟩

lemma *minus-fract-cancel* [*simp*]: $Fract\ (-\ a)\ (-\ b) = Fract\ a\ b$
⟨*proof*⟩

definition *diff-fract-def*: $q - r = q + -\ (r::'a\ fract)$

lemma *diff-fract* [*simp*]:
$[\![\ b \neq 0;\ d \neq 0\ ]\!] \Longrightarrow Fract\ a\ b - Fract\ c\ d = Fract\ (a * d - c * b)\ (b * d)$
⟨*proof*⟩

lift-definition *times-fract* :: *'a fract* ⇒ *'a fract* ⇒ *'a fract*
is $\lambda q\ r.\ (fst\ q * fst\ r,\ snd\ q * snd\ r)$
⟨*proof*⟩

lemma *mult-fract* [*simp*]: $Fract\ (a::'a::idom)\ b * Fract\ c\ d = Fract\ (a * c)\ (b * d)$
⟨*proof*⟩

lemma *mult-fract-cancel*:
$c \neq 0 \Longrightarrow Fract\ (c * a)\ (c * b) = Fract\ a\ b$
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *of-nat-fract*: $of\text{-}nat\ k = Fract\ (of\text{-}nat\ k)\ 1$
⟨*proof*⟩

lemma *Fract-of-nat-eq*: $Fract\ (of\text{-}nat\ k)\ 1 = of\text{-}nat\ k$
⟨*proof*⟩

lemma *fract-collapse*:
$Fract\ 0\ k = 0$
$Fract\ 1\ 1 = 1$
$Fract\ k\ 0 = 0$
⟨*proof*⟩

lemma *fract-expand*:
$0 = Fract\ 0\ 1$
$1 = Fract\ 1\ 1$
⟨*proof*⟩

lemma *Fract-cases-nonzero*:
obtains (*Fract*) *a b* **where** $q = Fract\ a\ b$ **and** $b \neq 0$ **and** $a \neq 0$
| (*0*) $q = 0$
⟨*proof*⟩

8.1.3 The field of rational numbers

context *idom*
begin

subclass *ring-no-zero-divisors* ⟨*proof*⟩

end

instantiation *fract* :: (*idom*) *field*

begin

lift-definition *inverse-fract* :: $'a$ *fract* $\Rightarrow$ $'a$ *fract*
is $\lambda x.$ *if fst x = 0 then* $(0, 1)$ *else* $(snd\ x, fst\ x)$
⟨*proof*⟩

lemma *inverse-fract* [*simp*]: *inverse* $(Fract\ a\ b) = Fract\ (b::'a::idom)\ a$
⟨*proof*⟩

definition *divide-fract-def*: $q\ div\ r = q * inverse\ (r::\ 'a\ fract)$

lemma *divide-fract* [*simp*]: $Fract\ a\ b\ div\ Fract\ c\ d = Fract\ (a * d)\ (b * c)$
⟨*proof*⟩

instance
⟨*proof*⟩

end

8.1.4 The ordered field of fractions over an ordered idom

instantiation *fract* :: (*linordered-idom*) *linorder*
begin

lemma *less-eq-fract-respect*:
fixes $a\ b\ a'\ b'\ c\ d\ c'\ d' :: 'a$
assumes *neq*: $b \neq 0\ \ b' \neq 0\ \ d \neq 0\ \ d' \neq 0$
assumes *eq1*: $a * b' = a' * b$
assumes *eq2*: $c * d' = c' * d$
shows $((a * d) * (b * d) \leq (c * b) * (b * d)) \longleftrightarrow ((a' * d') * (b' * d') \leq (c' * b') * (b' * d'))$
⟨*proof*⟩

lift-definition *less-eq-fract* :: $'a$ *fract* $\Rightarrow$ $'a$ *fract* $\Rightarrow$ *bool*
is $\lambda q\ r.\ (fst\ q * snd\ r) * (snd\ q * snd\ r) \leq (fst\ r * snd\ q) * (snd\ q * snd\ r)$
⟨*proof*⟩

definition *less-fract-def*: $z < (w::'a\ fract) \longleftrightarrow z \leq w \wedge \neg\ w \leq z$

lemma *le-fract* [*simp*]:
$[\![\ b \neq 0;\ d \neq 0\ ]\!] \Longrightarrow Fract\ a\ b \leq Fract\ c\ d \longleftrightarrow (a * d) * (b * d) \leq (c * b) * (b * d)$
⟨*proof*⟩

lemma *less-fract* [*simp*]:
$[\![\ b \neq 0;\ d \neq 0\ ]\!] \Longrightarrow Fract\ a\ b < Fract\ c\ d \longleftrightarrow (a * d) * (b * d) < (c * b) * (b * d)$
⟨*proof*⟩

instance
⟨*proof*⟩

end

instantiation *fract* :: (*linordered-idom*) *linordered-field*
begin

definition *abs-fract-def2*:
$|q| = (\mathit{if}\ q < 0\ \mathit{then}\ -q\ \mathit{else}\ (q::'a\ \mathit{fract}))$

definition *sgn-fract-def*:
$\mathit{sgn}\ (q::'a\ \mathit{fract}) = (\mathit{if}\ q = 0\ \mathit{then}\ 0\ \mathit{else}\ \mathit{if}\ 0 < q\ \mathit{then}\ 1\ \mathit{else}\ -\ 1)$

theorem *abs-fract* [*simp*]: $|\mathit{Fract}\ a\ b| = \mathit{Fract}\ |a|\ |b|$
⟨*proof*⟩

instance ⟨*proof*⟩

end

instantiation *fract* :: (*linordered-idom*) *distrib-lattice*
begin

definition *inf-fract-def*:
$(\mathit{inf} :: 'a\ \mathit{fract} \Rightarrow 'a\ \mathit{fract} \Rightarrow 'a\ \mathit{fract}) = \mathit{min}$

definition *sup-fract-def*:
$(\mathit{sup} :: 'a\ \mathit{fract} \Rightarrow 'a\ \mathit{fract} \Rightarrow 'a\ \mathit{fract}) = \mathit{max}$

instance
⟨*proof*⟩

end

lemma *fract-induct-pos* [*case-names Fract*]:
fixes $P :: 'a::\mathit{linordered\text{-}idom}\ \mathit{fract} \Rightarrow \mathit{bool}$
assumes *step*: $\bigwedge a\ b.\ 0 < b \Longrightarrow P\ (\mathit{Fract}\ a\ b)$
shows $P\ q$
⟨*proof*⟩

lemma *zero-less-Fract-iff*: $0 < b \Longrightarrow 0 < \mathit{Fract}\ a\ b \longleftrightarrow 0 < a$
⟨*proof*⟩

lemma *Fract-less-zero-iff*: $0 < b \Longrightarrow \mathit{Fract}\ a\ b < 0 \longleftrightarrow a < 0$
⟨*proof*⟩

lemma *zero-le-Fract-iff*: $0 < b \Longrightarrow 0 \leq \mathit{Fract}\ a\ b \longleftrightarrow 0 \leq a$
⟨*proof*⟩

lemma *Fract-le-zero-iff*: $0 < b \Longrightarrow \mathit{Fract}\ a\ b \leq 0 \longleftrightarrow a \leq 0$
⟨*proof*⟩

lemma *one-less-Fract-iff*: $0 < b \Longrightarrow 1 < \mathit{Fract}\ a\ b \longleftrightarrow b < a$
⟨*proof*⟩

lemma *Fract-less-one-iff*: $0 < b \Longrightarrow \mathit{Fract}\ a\ b < 1 \longleftrightarrow a < b$
⟨*proof*⟩

lemma *one-le-Fract-iff*: $0 < b \Longrightarrow 1 \leq \mathit{Fract}\ a\ b \longleftrightarrow b \leq a$
⟨*proof*⟩

lemma *Fract-le-one-iff*: $0 < b \Longrightarrow \mathit{Fract}\ a\ b \leq 1 \longleftrightarrow a \leq b$
⟨*proof*⟩

end

9 Fundamental Theorem of Algebra

theory *Fundamental-Theorem-Algebra*
imports *Polynomial Complex-Main*
begin

9.1 More lemmas about module of complex numbers

The triangle inequality for cmod

lemma *complex-mod-triangle-sub*: $\mathit{cmod}\ w \leq \mathit{cmod}\ (w + z) + \mathit{norm}\ z$
⟨*proof*⟩

9.2 Basic lemmas about polynomials

lemma *poly-bound-exists*:
fixes $p :: 'a::\{\mathit{comm\text{-}semiring\text{-}0}, \mathit{real\text{-}normed\text{-}div\text{-}algebra}\}\ \mathit{poly}$
shows $\exists m.\ m > 0 \land (\forall z.\ \mathit{norm}\ z \leq r \longrightarrow \mathit{norm}\ (\mathit{poly}\ p\ z) \leq m)$
⟨*proof*⟩

Offsetting the variable in a polynomial gives another of same degree

definition $\mathit{offset\text{-}poly} :: 'a::\mathit{comm\text{-}semiring\text{-}0}\ \mathit{poly} \Rightarrow 'a \Rightarrow 'a\ \mathit{poly}$
where $\mathit{offset\text{-}poly}\ p\ h = \mathit{fold\text{-}coeffs}\ (\lambda a\ q.\ \mathit{smult}\ h\ q + \mathit{pCons}\ a\ q)\ p\ 0$

lemma *offset-poly-0*: $\mathit{offset\text{-}poly}\ 0\ h = 0$
⟨*proof*⟩

lemma *offset-poly-pCons*:
$\mathit{offset\text{-}poly}\ (\mathit{pCons}\ a\ p)\ h =$
$\mathit{smult}\ h\ (\mathit{offset\text{-}poly}\ p\ h) + \mathit{pCons}\ a\ (\mathit{offset\text{-}poly}\ p\ h)$
⟨*proof*⟩

lemma *offset-poly-single* [*simp*]: *offset-poly* [:*a*:] *h* = [:*a*:]
⟨*proof*⟩

lemma *poly-offset-poly*: *poly* (*offset-poly p h*) *x* = *poly p* (*h* + *x*)
⟨*proof*⟩

lemma *offset-poly-eq-0-lemma*: *smult c p* + *pCons a p* = *0* ⟹ *p* = *0*
⟨*proof*⟩

lemma *offset-poly-eq-0-iff* [*simp*]: *offset-poly p h* = *0* ⟷ *p* = *0*
⟨*proof*⟩

lemma *degree-offset-poly* [*simp*]: *degree* (*offset-poly p h*) = *degree p*
⟨*proof*⟩

definition *psize p* = (*if p* = *0 then 0 else Suc* (*degree p*))

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

lemma *poly-offset*:
fixes *p* :: *'a::comm-ring-1 poly*
shows ∃ *q*. *psize q* = *psize p* ∧ (∀ *x*. *poly q x* = *poly p* (*a* + *x*))
⟨*proof*⟩

An alternative useful formulation of completeness of the reals

lemma *real-sup-exists*:
assumes *ex*: ∃ *x*. *P x*
and *bz*: ∃ *z*. ∀ *x*. *P x* ⟶ *x* < *z*
shows ∃ *s::real*. ∀ *y*. (∃ *x*. *P x* ∧ *y* < *x*) ⟷ *y* < *s*
⟨*proof*⟩

9.3 Fundamental theorem of algebra

lemma *unimodular-reduce-norm*:
assumes *md*: *cmod z* = *1*
shows *cmod* (*z* + *1*) < *1* ∨ *cmod* (*z* − *1*) < *1* ∨ *cmod* (*z* + i) < *1* ∨ *cmod* (*z* − i) < *1*
⟨*proof*⟩

Hence we can always reduce modulus of *1* + *b z^n* if nonzero

lemma *reduce-poly-simple*:
assumes *b*: *b* ≠ *0*
and *n*: *n* ≠ *0*
shows ∃ *z*. *cmod* (*1* + *b* ∗ *z^n*) < *1*
⟨*proof*⟩

Bolzano-Weierstrass type property for closed disc in complex plane.

lemma *metric-bound-lemma*: $cmod\ (x - y) \leq |Re\ x - Re\ y| + |Im\ x - Im\ y|$
⟨*proof*⟩

lemma *Bolzano-Weierstrass-complex-disc*:
assumes r: $\forall n.\ cmod\ (s\ n) \leq r$
shows $\exists f\ z.\ strict\text{-}mono\ (f :: nat \Rightarrow nat) \wedge (\forall e >0.\ \exists N.\ \forall n \geq N.\ cmod\ (s\ (f\ n) - z) < e)$
⟨*proof*⟩

Polynomial is continuous.

lemma *poly-cont*:
fixes $p :: 'a::\{comm\text{-}semiring\text{-}0,real\text{-}normed\text{-}div\text{-}algebra\}\ poly$
assumes ep: $e > 0$
shows $\exists d >0.\ \forall w.\ 0 < norm\ (w - z) \wedge norm\ (w - z) < d \longrightarrow norm\ (poly\ p\ w - poly\ p\ z) < e$
⟨*proof*⟩

Hence a polynomial attains minimum on a closed disc in the complex plane.

lemma *poly-minimum-modulus-disc*: $\exists z.\ \forall w.\ cmod\ w \leq r \longrightarrow cmod\ (poly\ p\ z) \leq cmod\ (poly\ p\ w)$
⟨*proof*⟩

Nonzero polynomial in z goes to infinity as z does.

lemma *poly-infinity*:
fixes $p:: 'a::\{comm\text{-}semiring\text{-}0,real\text{-}normed\text{-}div\text{-}algebra\}\ poly$
assumes ex: $p \neq 0$
shows $\exists r.\ \forall z.\ r \leq norm\ z \longrightarrow d \leq norm\ (poly\ (pCons\ a\ p)\ z)$
⟨*proof*⟩

Hence polynomial's modulus attains its minimum somewhere.

lemma *poly-minimum-modulus*: $\exists z.\forall w.\ cmod\ (poly\ p\ z) \leq cmod\ (poly\ p\ w)$
⟨*proof*⟩

Constant function (non-syntactic characterization).

definition $constant\ f \longleftrightarrow (\forall x\ y.\ f\ x = f\ y)$

lemma *nonconstant-length*: $\neg\ constant\ (poly\ p) \Longrightarrow psize\ p \geq 2$
⟨*proof*⟩

lemma *poly-replicate-append*: $poly\ (monom\ 1\ n * p)\ (x::'a::comm\text{-}ring\text{-}1) = x\hat{}n * poly\ p\ x$
⟨*proof*⟩

Decomposition of polynomial, skipping zero coefficients after the first.

lemma *poly-decompose-lemma*:
assumes nz: $\neg\ (\forall z.\ z \neq 0 \longrightarrow poly\ p\ z = (0::'a::idom))$
shows $\exists k\ a\ q.\ a \neq 0 \wedge Suc\ (psize\ q + k) = psize\ p \wedge (\forall z.\ poly\ p\ z = z\hat{}k * poly\ (pCons\ a\ q)\ z)$

⟨*proof*⟩

lemma *poly-decompose*:
fixes p :: *'a::idom poly*
assumes *nc*: ¬ *constant* (*poly p*)
shows $\exists k\ a\ q.\ a \neq 0 \wedge k \neq 0 \wedge$
$psize\ q + k + 1 = psize\ p \wedge$
$(\forall z.\ poly\ p\ z = poly\ p\ 0 + z\hat{}k * poly\ (pCons\ a\ q)\ z)$
⟨*proof*⟩

Fundamental theorem of algebra

theorem *fundamental-theorem-of-algebra*:
assumes *nc*: ¬ *constant* (*poly p*)
shows $\exists z$::*complex*. $poly\ p\ z = 0$
⟨*proof*⟩

Alternative version with a syntactic notion of constant polynomial.

lemma *fundamental-theorem-of-algebra-alt*:
assumes *nc*: $\neg (\exists a\ l.\ a \neq 0 \wedge l = 0 \wedge p = pCons\ a\ l)$
shows $\exists z.\ poly\ p\ z = (0::complex)$
⟨*proof*⟩

9.4 Nullstellensatz, degrees and divisibility of polynomials

lemma *nullstellensatz-lemma*:
fixes p :: *complex poly*
assumes $\forall x.\ poly\ p\ x = 0 \longrightarrow poly\ q\ x = 0$
and $degree\ p = n$
and $n \neq 0$
shows $p\ dvd\ (q\ \hat{}\ n)$
⟨*proof*⟩

lemma *nullstellensatz-univariate*:
$(\forall x.\ poly\ p\ x = (0::complex) \longrightarrow poly\ q\ x = 0) \longleftrightarrow$
$p\ dvd\ (q\ \hat{}\ (degree\ p)) \vee (p = 0 \wedge q = 0)$
⟨*proof*⟩

Useful lemma

lemma *constant-degree*:
fixes p :: *'a::{idom,ring-char-0} poly*
shows *constant* (*poly p*) ⟷ $degree\ p = 0$ (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *complex-poly-decompose*:
$smult\ (lead\text{-}coeff\ p)\ (\prod z|poly\ p\ z = 0.\ [:-z,\ 1:]\ \hat{}\ order\ z\ p) = (p :: complex\ poly)$
⟨*proof*⟩

instance *complex* :: *alg-closed-field*
⟨*proof*⟩

lemma *size-proots-complex*: *size* (*proots* (*p* :: *complex poly*)) = *degree p*
⟨*proof*⟩

lemma *complex-poly-decompose-multiset*:
smult (*lead-coeff p*) (∏ *x*∈#*proots p*. [:−*x*, *1*:]) = (*p* :: *complex poly*)
⟨*proof*⟩

lemma *complex-poly-decompose′*:
obtains *root* **where** *smult* (*lead-coeff p*) (∏ *i*<*degree p*. [:−*root i*, *1*:]) = (*p* :: *complex poly*)
⟨*proof*⟩

lemma *complex-poly-decompose-rsquarefree*:
assumes *rsquarefree p*
shows *smult* (*lead-coeff p*) (∏ *z*|*poly p z* = *0*. [:−*z*, *1*:]) = (*p* :: *complex poly*)
⟨*proof*⟩

Arithmetic operations on multivariate polynomials.

lemma *mpoly-base-conv*:
fixes *x* :: ′*a*::*comm-ring-1*
shows *0* = *poly 0 x c* = *poly* [:*c*:] *x x* = *poly* [:*0*,*1*:] *x*
⟨*proof*⟩

lemma *mpoly-norm-conv*:
fixes *x* :: ′*a*::*comm-ring-1*
shows *poly* [:*0*:] *x* = *poly 0 x poly* [:*poly 0 y*:] *x* = *poly 0 x*
⟨*proof*⟩

lemma *mpoly-sub-conv*:
fixes *x* :: ′*a*::*comm-ring-1*
shows *poly p x* − *poly q x* = *poly p x* + −*1* ∗ *poly q x*
⟨*proof*⟩

lemma *poly-pad-rule*: *poly p x* = *0* ⟹ *poly* (*pCons 0 p*) *x* = *0*
⟨*proof*⟩

lemma *poly-cancel-eq-conv*:
fixes *x* :: ′*a*::*field*
shows *x* = *0* ⟹ *a* ≠ *0* ⟹ *y* = *0* ⟷ *a* ∗ *y* − *b* ∗ *x* = *0*
⟨*proof*⟩

lemma *poly-divides-pad-rule*:
fixes *p*:: (′*a*::*comm-ring-1*) *poly*
assumes *pq*: *p dvd q*
shows *p dvd* (*pCons 0 q*)
⟨*proof*⟩

lemma *poly-divides-conv0*:

fixes *p*:: *'a::field poly*
assumes *lgpq*: *degree q* < *degree p* **and** *lq*: *p* ≠ *0*
shows *p dvd q* ⟷ *q* = *0*
⟨*proof*⟩

lemma *poly-divides-conv1*:
fixes *p* :: *'a::field poly*
assumes *a0*: *a* ≠ *0*
and *pp'*: *p dvd p'*
and *qrp'*: *smult a q* − *p'* = *r*
shows *p dvd q* ⟷ *p dvd r*
⟨*proof*⟩

lemma *basic-cqe-conv1*:
(∃ *x*. *poly p x* = *0* ∧ *poly 0 x* ≠ *0*) ⟷ *False*
(∃ *x*. *poly 0 x* ≠ *0*) ⟷ *False*
(∃ *x*. *poly* [:*c*:] *x* ≠ *0*) ⟷ *c* ≠ *0*
(∃ *x*. *poly 0 x* = *0*) ⟷ *True*
(∃ *x*. *poly* [:*c*:] *x* = *0*) ⟷ *c* = *0*
⟨*proof*⟩

lemma *basic-cqe-conv2*:
assumes *l*: *p* ≠ *0*
shows ∃ *x*. *poly* (*pCons a* (*pCons b p*)) *x* = (*0*::*complex*)
⟨*proof*⟩

lemma *basic-cqe-conv-2b*: (∃ *x*. *poly p x* ≠ (*0*::*complex*)) ⟷ *p* ≠ *0*
⟨*proof*⟩

lemma *basic-cqe-conv3*:
fixes *p q* :: *complex poly*
assumes *l*: *p* ≠ *0*
shows (∃ *x*. *poly* (*pCons a p*) *x* = *0* ∧ *poly q x* ≠ *0*) ⟷ ¬ (*pCons a p*) *dvd* (*q* ^ *psize p*)
⟨*proof*⟩

lemma *basic-cqe-conv4*:
fixes *p q* :: *complex poly*
assumes *h*: ⋀*x*. *poly* (*q* ^ *n*) *x* = *poly r x*
shows *p dvd* (*q* ^ *n*) ⟷ *p dvd r*
⟨*proof*⟩

lemma *poly-const-conv*:
fixes *x* :: *'a::comm-ring-1*
shows *poly* [:*c*:] *x* = *y* ⟷ *c* = *y*
⟨*proof*⟩

end

theory *Group-Closure*
imports
Main
begin

context *ab-group-add*
begin

inductive-set *group-closure* :: *'a set* ⇒ *'a set* **for** *S*
where *base*: *s* ∈ *insert 0 S* ⟹ *s* ∈ *group-closure S*
| *diff*: *s* ∈ *group-closure S* ⟹ *t* ∈ *group-closure S* ⟹ *s* − *t* ∈ *group-closure S*

lemma *zero-in-group-closure* [*simp*]:
0 ∈ *group-closure S*
⟨*proof*⟩

lemma *group-closure-minus-iff* [*simp*]:
− *s* ∈ *group-closure S* ⟷ *s* ∈ *group-closure S*
⟨*proof*⟩

lemma *group-closure-add*:
s + *t* ∈ *group-closure S* **if** *s* ∈ *group-closure S* **and** *t* ∈ *group-closure S*
⟨*proof*⟩

lemma *group-closure-empty* [*simp*]:
group-closure {} = {*0*}
⟨*proof*⟩

lemma *group-closure-insert-zero* [*simp*]:
group-closure (*insert 0 S*) = *group-closure S*
⟨*proof*⟩

end

context *comm-ring-1*
begin

lemma *group-closure-scalar-mult-left*:
of-nat n ∗ *s* ∈ *group-closure S* **if** *s* ∈ *group-closure S*
⟨*proof*⟩

lemma *group-closure-scalar-mult-right*:
s ∗ *of-nat n* ∈ *group-closure S* **if** *s* ∈ *group-closure S*
⟨*proof*⟩

end

lemma *group-closure-abs-iff* [*simp*]:

$|s| \in$ *group-closure* $S \longleftrightarrow s \in$ *group-closure* S **for** s :: *int*
⟨*proof*⟩

lemma *group-closure-mult-left*:
$s * t \in$ *group-closure* S **if** $s \in$ *group-closure* S **for** $s\ t$ :: *int*
⟨*proof*⟩

lemma *group-closure-mult-right*:
$s * t \in$ *group-closure* S **if** $t \in$ *group-closure* S **for** $s\ t$ :: *int*
⟨*proof*⟩

context *idom*
begin

lemma *group-closure-mult-all-eq*:
group-closure (*times* k ‘ S) = *times* k ‘ *group-closure* S
⟨*proof*⟩

end

lemma *Gcd-group-closure-eq-Gcd*:
Gcd (*group-closure* S) = *Gcd* S **for** S :: *int set*
⟨*proof*⟩

lemma *group-closure-sum*:
fixes S :: *int set*
assumes X: *finite* X $X \neq \{\}$ $X \subseteq S$
shows $(\sum x{\in}X.\ a\ x * x) \in$ *group-closure* S
⟨*proof*⟩

lemma *Gcd-group-closure-in-group-closure*:
Gcd (*group-closure* S) $\in$ *group-closure* S **for** S :: *int set*
⟨*proof*⟩

lemma *Gcd-in-group-closure*:
Gcd $S \in$ *group-closure* S **for** S :: *int set*
⟨*proof*⟩

lemma *group-closure-eq*:
group-closure S = *range* (*times* (*Gcd* S)) **for** S :: *int set*
⟨*proof*⟩

end

theory *Normalized-Fraction*
imports
Main
Euclidean-Algorithm

Fraction-Field
begin

lemma *unit-factor-1-imp-normalized*: *unit-factor x = 1* ⟹ *normalize x = x*
⟨*proof*⟩

definition *quot-to-fract* :: *'a* × *'a* ⇒ *'a* :: *idom fract* **where**
quot-to-fract = (λ(*a*,*b*). *Fraction-Field.Fract a b*)

definition *normalize-quot* :: *'a* :: {*ring-gcd*,*idom-divide*,*semiring-gcd-mult-normalize*} × *'a* ⇒ *'a* × *'a* **where**
normalize-quot =
(λ(*a*,*b*). *if b = 0 then* (*0*,*1*) *else let d = gcd a b* ∗ *unit-factor b in* (*a div d*, *b div d*))

lemma *normalize-quot-zero* [*simp*]:
normalize-quot (*a*, *0*) = (*0*, *1*)
⟨*proof*⟩

lemma *normalize-quot-proj*:
fst (*normalize-quot* (*a*, *b*)) = *a div* (*gcd a b* ∗ *unit-factor b*)
snd (*normalize-quot* (*a*, *b*)) = *normalize b div gcd a b* **if** *b* ≠ *0*
⟨*proof*⟩

definition *normalized-fracts* :: (*'a* :: {*ring-gcd*,*idom-divide*} × *'a*) *set* **where**
normalized-fracts = {(*a*,*b*). *coprime a b* ∧ *unit-factor b = 1*}

lemma *not-normalized-fracts-0-denom* [*simp*]: (*a*, *0*) ∉ *normalized-fracts*
⟨*proof*⟩

lemma *unit-factor-snd-normalize-quot* [*simp*]:
unit-factor (*snd* (*normalize-quot x*)) = *1*
⟨*proof*⟩

lemma *snd-normalize-quot-nonzero* [*simp*]: *snd* (*normalize-quot x*) ≠ *0*
⟨*proof*⟩

lemma *normalize-quot-aux*:
fixes *a b*
assumes *b* ≠ *0*
defines *d* ≡ *gcd a b* ∗ *unit-factor b*
shows *a = fst* (*normalize-quot* (*a*,*b*)) ∗ *d b = snd* (*normalize-quot* (*a*,*b*)) ∗ *d*
d dvd a d dvd b d ≠ *0*
⟨*proof*⟩

lemma *normalize-quotE*:
assumes *b* ≠ *0*
obtains *d* **where** *a = fst* (*normalize-quot* (*a*,*b*)) ∗ *d b = snd* (*normalize-quot* (*a*,*b*)) ∗ *d*

d dvd a d dvd b d ≠ 0
⟨*proof*⟩

lemma *normalize-quotE′*:
assumes *snd x ≠ 0*
obtains *d* **where** *fst x = fst (normalize-quot x) ∗ d snd x = snd (normalize-quot x) ∗ d*
d dvd fst x d dvd snd x d ≠ 0
⟨*proof*⟩

lemma *coprime-normalize-quot*:
coprime (fst (normalize-quot x)) (snd (normalize-quot x))
⟨*proof*⟩

lemma *normalize-quot-in-normalized-fracts* [*simp*]: *normalize-quot x ∈ normalized-fracts*
⟨*proof*⟩

lemma *normalize-quot-eq-iff*:
assumes *b ≠ 0 d ≠ 0*
shows *normalize-quot (a,b) = normalize-quot (c,d) ⟷ a ∗ d = b ∗ c*
⟨*proof*⟩

lemma *normalize-quot-eq-iff′*:
assumes *snd x ≠ 0 snd y ≠ 0*
shows *normalize-quot x = normalize-quot y ⟷ fst x ∗ snd y = snd x ∗ fst y*
⟨*proof*⟩

lemma *normalize-quot-id*: *x ∈ normalized-fracts ⟹ normalize-quot x = x*
⟨*proof*⟩

lemma *normalize-quot-idem* [*simp*]: *normalize-quot (normalize-quot x) = normalize-quot x*
⟨*proof*⟩

lemma *fractrel-iff-normalize-quot-eq*:
fractrel x y ⟷ normalize-quot x = normalize-quot y ∧ snd x ≠ 0 ∧ snd y ≠ 0
⟨*proof*⟩

lemma *fractrel-normalize-quot-left*:
assumes *snd x ≠ 0*
shows *fractrel (normalize-quot x) y ⟷ fractrel x y*
⟨*proof*⟩

lemma *fractrel-normalize-quot-right*:
assumes *snd x ≠ 0*
shows *fractrel y (normalize-quot x) ⟷ fractrel y x*
⟨*proof*⟩

lift-definition *quot-of-fract* ::
  'a :: {*ring-gcd*,*idom-divide*,*semiring-gcd-mult-normalize*} *fract* ⇒ *'a* × *'a*
    is *normalize-quot*
  ⟨*proof*⟩

lemma *quot-to-fract-quot-of-fract* [*simp*]: *quot-to-fract* (*quot-of-fract x*) = *x*
  ⟨*proof*⟩

lemma *quot-of-fract-quot-to-fract*: *quot-of-fract* (*quot-to-fract x*) = *normalize-quot x*
⟨*proof*⟩

lemma *quot-of-fract-quot-to-fract'*:
  x ∈ *normalized-fracts* ⟹ *quot-of-fract* (*quot-to-fract x*) = *x*
  ⟨*proof*⟩

lemma *quot-of-fract-in-normalized-fracts* [*simp*]: *quot-of-fract x* ∈ *normalized-fracts*
  ⟨*proof*⟩

lemma *normalize-quotI*:
  assumes *a* ∗ *d* = *b* ∗ *c b* ≠ *0* (*c*, *d*) ∈ *normalized-fracts*
  shows *normalize-quot* (*a*, *b*) = (*c*, *d*)
⟨*proof*⟩

lemma *td-normalized-fract*:
  type-definition quot-of-fract quot-to-fract normalized-fracts
  ⟨*proof*⟩

lemma *quot-of-fract-add-aux*:
  assumes *snd x* ≠ *0 snd y* ≠ *0*
  shows (*fst x* ∗ *snd y* + *fst y* ∗ *snd x*) ∗ (*snd* (*normalize-quot x*) ∗ *snd* (*normalize-quot y*)) =
    snd x ∗ *snd y* ∗ (*fst* (*normalize-quot x*) ∗ *snd* (*normalize-quot y*) +
    snd (*normalize-quot x*) ∗ *fst* (*normalize-quot y*))
⟨*proof*⟩

locale *fract-as-normalized-quot*
begin
setup-lifting *td-normalized-fract*
end

lemma *quot-of-fract-add*:
  quot-of-fract (*x* + *y*) =
    (*let* (*a*,*b*) = *quot-of-fract x*; (*c*,*d*) = *quot-of-fract y*
     in normalize-quot (*a* ∗ *d* + *b* ∗ *c*, *b* ∗ *d*))
  ⟨*proof*⟩

lemma *quot-of-fract-uminus*:
quot-of-fract $(-x)$ = (*let* (a,b) = *quot-of-fract* x *in* $(-a, b)$)
⟨*proof*⟩

lemma *quot-of-fract-diff*:
quot-of-fract $(x - y)$ =
(*let* (a,b) = *quot-of-fract* x; (c,d) = *quot-of-fract* y
in *normalize-quot* $(a * d - b * c, b * d)$) (**is** - = *?rhs*)
⟨*proof*⟩

lemma *normalize-quot-mult-coprime*:
assumes *coprime a b coprime c d unit-factor b* = *1 unit-factor d* = *1*
defines $e \equiv$ *fst* (*normalize-quot* (a, d)) **and** $f \equiv$ *snd* (*normalize-quot* (a, d))
and $g \equiv$ *fst* (*normalize-quot* (c, b)) **and** $h \equiv$ *snd* (*normalize-quot* (c, b))
shows *normalize-quot* $(a * c, b * d) = (e * g, f * h)$
⟨*proof*⟩

lemma *normalize-quot-mult*:
assumes *snd* $x \neq 0$ *snd* $y \neq 0$
shows *normalize-quot* (*fst* x * *fst* y, *snd* x * *snd* y) = *normalize-quot*
(*fst* (*normalize-quot* x) * *fst* (*normalize-quot* y),
snd (*normalize-quot* x) * *snd* (*normalize-quot* y))
⟨*proof*⟩

lemma *quot-of-fract-mult*:
quot-of-fract $(x * y)$ =
(*let* (a,b) = *quot-of-fract* x; (c,d) = *quot-of-fract* y;
(e,f) = *normalize-quot* (a,d); (g,h) = *normalize-quot* (c,b)
in $(e*g, f*h)$)
⟨*proof*⟩

lemma *normalize-quot-0* [*simp*]:
normalize-quot $(0, x) = (0, 1)$ *normalize-quot* $(x, 0) = (0, 1)$
⟨*proof*⟩

lemma *normalize-quot-eq-0-iff* [*simp*]: *fst* (*normalize-quot* x) = $0 \longleftrightarrow$ *fst* $x = 0$ $\vee$ *snd* $x = 0$
⟨*proof*⟩

lemma *fst-quot-of-fract-0-imp*: *fst* (*quot-of-fract* x) = $0 \Longrightarrow$ *snd* (*quot-of-fract* x) = *1*
⟨*proof*⟩

lemma *normalize-quot-swap*:
assumes $a \neq 0$ $b \neq 0$
defines $a' \equiv$ *fst* (*normalize-quot* (a, b)) **and** $b' \equiv$ *snd* (*normalize-quot* (a, b))
shows *normalize-quot* (b, a) = (b' *div unit-factor* a', a' *div unit-factor* a')
⟨*proof*⟩

lemma *quot-of-fract-inverse*:
quot-of-fract (*inverse x*) =
(*let* (*a*,*b*) = *quot-of-fract x*; *d* = *unit-factor a*
in *if d* = *0* *then* (*0*, *1*) *else* (*b div d*, *a div d*))
⟨*proof*⟩

lemma *normalize-quot-div-unit-left*:
fixes *x y u*
assumes *is-unit u*
defines $x' \equiv$ *fst* (*normalize-quot* (*x*, *y*)) **and** $y' \equiv$ *snd* (*normalize-quot* (*x*, *y*))
shows *normalize-quot* (*x div u*, *y*) = (x' *div u*, y')
⟨*proof*⟩

lemma *normalize-quot-div-unit-right*:
fixes *x y u*
assumes *is-unit u*
defines $x' \equiv$ *fst* (*normalize-quot* (*x*, *y*)) **and** $y' \equiv$ *snd* (*normalize-quot* (*x*, *y*))
shows *normalize-quot* (*x*, *y div u*) = ($x' * u$, y')
⟨*proof*⟩

lemma *normalize-quot-normalize-left*:
fixes *x y u*
defines $x' \equiv$ *fst* (*normalize-quot* (*x*, *y*)) **and** $y' \equiv$ *snd* (*normalize-quot* (*x*, *y*))
shows *normalize-quot* (*normalize x*, *y*) = (x' *div unit-factor x*, y')
⟨*proof*⟩

lemma *normalize-quot-normalize-right*:
fixes *x y u*
defines $x' \equiv$ *fst* (*normalize-quot* (*x*, *y*)) **and** $y' \equiv$ *snd* (*normalize-quot* (*x*, *y*))
shows *normalize-quot* (*x*, *normalize y*) = ($x' *$ *unit-factor y*, y')
⟨*proof*⟩

lemma *quot-of-fract-0* [*simp*]: *quot-of-fract 0* = (*0*, *1*)
⟨*proof*⟩

lemma *quot-of-fract-1* [*simp*]: *quot-of-fract 1* = (*1*, *1*)
⟨*proof*⟩

lemma *quot-of-fract-divide*:
quot-of-fract (*x* / *y*) = (*if y* = *0* *then* (*0*, *1*) *else*
(*let* (*a*,*b*) = *quot-of-fract x*; (*c*,*d*) = *quot-of-fract y*;
(*e*,*f*) = *normalize-quot* (*a*,*c*); (*g*,*h*) = *normalize-quot* (*d*,*b*)
in (*e* ∗ *g*, *f* ∗ *h*))) (**is** - = *?rhs*)
⟨*proof*⟩

lemma *snd-quot-of-fract-nonzero* [*simp*]: *snd* (*quot-of-fract x*) ≠ *0*
⟨*proof*⟩

lemma *Fract-quot-of-fract* [*simp*]: *Fract* (*fst* (*quot-of-fract* *x*)) (*snd* (*quot-of-fract* *x*)) = *x*
⟨*proof*⟩

lemma *snd-quot-of-fract-Fract-whole*:
assumes *y dvd x*
shows *snd* (*quot-of-fract* (*Fract x y*)) = *1*
⟨*proof*⟩

lemma *fst-quot-of-fract-eq-0-iff* [*simp*]: *fst* (*quot-of-fract* *x*) = *0* ⟷ *x* = *0*
⟨*proof*⟩

lemma *coprime-quot-of-fract*:
coprime (*fst* (*quot-of-fract* *x*)) (*snd* (*quot-of-fract* *x*))
⟨*proof*⟩

lemma *unit-factor-snd-quot-of-fract*: *unit-factor* (*snd* (*quot-of-fract* *x*)) = *1*
⟨*proof*⟩

lemma *normalize-snd-quot-of-fract*: *normalize* (*snd* (*quot-of-fract* *x*)) = *snd* (*quot-of-fract* *x*)
⟨*proof*⟩

end

10 n-th powers and roots of naturals

theory *Nth-Powers*
imports *Primes*
begin

10.1 The set of n-th powers

definition *is-nth-power* :: *nat* ⇒ *'a* :: *monoid-mult* ⇒ *bool* **where**
is-nth-power n x ⟷ (∃ *y*. *x* = *y* ^ *n*)

lemma *is-nth-power-nth-power* [*simp*, *intro*]: *is-nth-power n* (*x* ^ *n*)
⟨*proof*⟩

lemma *is-nth-powerI* [*intro?*]: *x* = *y* ^ *n* ⟹ *is-nth-power n x*
⟨*proof*⟩

lemma *is-nth-powerE*: *is-nth-power n x* ⟹ (⋀*y*. *x* = *y* ^ *n* ⟹ *P*) ⟹ *P*
⟨*proof*⟩

abbreviation *is-square* **where** *is-square* ≡ *is-nth-power 2*

lemma *is-zeroth-power* [*simp*]: *is-nth-power 0 x* ⟷ *x* = *1*

⟨*proof*⟩

lemma *is-first-power* [*simp*]: *is-nth-power 1 x*
⟨*proof*⟩

lemma *is-first-power′* [*simp*]: *is-nth-power* (*Suc 0*) *x*
⟨*proof*⟩

lemma *is-nth-power-0* [*simp*]: $n > 0 \Longrightarrow$ *is-nth-power n* (*0* :: *′a* :: *semiring-1*)
⟨*proof*⟩

lemma *is-nth-power-0-iff* [*simp*]: *is-nth-power n* (*0* :: *′a* :: *semiring-1*) $\longleftrightarrow n > 0$
⟨*proof*⟩

lemma *is-nth-power-1* [*simp*]: *is-nth-power n 1*
⟨*proof*⟩

lemma *is-nth-power-Suc-0* [*simp*]: *is-nth-power n* (*Suc 0*)
⟨*proof*⟩

lemma *is-nth-power-conv-multiplicity*:
fixes *x* :: *′a* :: {*factorial-semiring*, *normalization-semidom-multiplicative*}
assumes $n > 0$
shows *is-nth-power n* (*normalize x*) $\longleftrightarrow$ ($\forall$ *p. prime p* $\longrightarrow$ *n dvd multiplicity p x*)
⟨*proof*⟩

lemma *is-nth-power-conv-multiplicity-nat*:
assumes $n > 0$
shows *is-nth-power n* (*x* :: *nat*) $\longleftrightarrow$ ($\forall$ *p. prime p* $\longrightarrow$ *n dvd multiplicity p x*)
⟨*proof*⟩

lemma *is-nth-power-mult*:
assumes *is-nth-power n a is-nth-power n b*
shows *is-nth-power n* (*a* ∗ *b* :: *′a* :: *comm-monoid-mult*)
⟨*proof*⟩

lemma *is-nth-power-mult-coprime-natD*:
fixes *a b* :: *nat*
assumes *coprime a b is-nth-power n* (*a* ∗ *b*) $a > 0$ $b > 0$
shows *is-nth-power n a is-nth-power n b*
⟨*proof*⟩

lemma *is-nth-power-mult-coprime-nat-iff*:
fixes *a b* :: *nat*
assumes *coprime a b*
shows *is-nth-power n* (*a* ∗ *b*) $\longleftrightarrow$ *is-nth-power n a* $\wedge$*is-nth-power n b*
⟨*proof*⟩

lemma *is-nth-power-prime-power-nat-iff*:
fixes p :: *nat* **assumes** *prime* p
shows *is-nth-power* n $(p \mathbin{\char`\^} k) \longleftrightarrow n$ *dvd* k
⟨*proof*⟩

lemma *is-nth-power-nth-power'*:
assumes n *dvd* n'
shows *is-nth-power* n $(m \mathbin{\char`\^} n')$
⟨*proof*⟩

definition *is-nth-power-nat* :: *nat* ⇒ *nat* ⇒ *bool*
where [*code-abbrev*]: *is-nth-power-nat* = *is-nth-power*

lemma *is-nth-power-nat-code* [*code*]:
is-nth-power-nat n m =
(*if* $n = 0$ *then* $m = 1$
else if $m = 0$ *then* $n > 0$
else if $n = 1$ *then* *True*
else $(\exists k \in \{1..m\}.\ k \mathbin{\char`\^} n = m)$)
⟨*proof*⟩

10.2 The n-root of a natural number

definition *nth-root-nat* :: *nat* ⇒ *nat* ⇒ *nat* **where**
nth-root-nat k n = (*if* $k = 0$ *then* 0 *else* *Max* $\{m.\ m \mathbin{\char`\^} k \le n\}$)

lemma *zeroth-root-nat* [*simp*]: *nth-root-nat* 0 $n = 0$
⟨*proof*⟩

lemma *nth-root-nat-aux1*:
assumes $k > 0$
shows $\{m{::}nat.\ m \mathbin{\char`\^} k \le n\} \subseteq \{..n\}$
⟨*proof*⟩

lemma *nth-root-nat-aux2*:
assumes $k > 0$
shows *finite* $\{m{::}nat.\ m \mathbin{\char`\^} k \le n\}$ $\{m{::}nat.\ m \mathbin{\char`\^} k \le n\} \neq \{\}$
⟨*proof*⟩

lemma
assumes $k > 0$
shows *nth-root-nat-power-le*: *nth-root-nat* k $n \mathbin{\char`\^} k \le n$
and *nth-root-nat-ge*: $x \mathbin{\char`\^} k \le n \Longrightarrow x \le$ *nth-root-nat* k n
⟨*proof*⟩

lemma *nth-root-nat-less*:
assumes $k > 0$ $x \mathbin{\char`\^} k > n$
shows *nth-root-nat* k $n < x$
⟨*proof*⟩

lemma *nth-root-nat-unique*:
  assumes $m \,\hat{}\, k \leq n$ $(m + 1) \,\hat{}\, k > n$
  shows *nth-root-nat k n* = *m*
⟨*proof*⟩

lemma *nth-root-nat-0* [*simp*]: *nth-root-nat k 0* = *0* ⟨*proof*⟩
lemma *nth-root-nat-1* [*simp*]: $k > 0 \Longrightarrow$ *nth-root-nat k 1* = *1*
  ⟨*proof*⟩
lemma *nth-root-nat-Suc-0* [*simp*]: $k > 0 \Longrightarrow$ *nth-root-nat k* (*Suc 0*) = *Suc 0*
  ⟨*proof*⟩

lemma *first-root-nat* [*simp*]: *nth-root-nat 1 n* = *n*
  ⟨*proof*⟩

lemma *first-root-nat′* [*simp*]: *nth-root-nat* (*Suc 0*) *n* = *n*
  ⟨*proof*⟩

lemma *nth-root-nat-code-naive′*:
  nth-root-nat k n = (*if k* = *0 then 0 else Max* (*Set.filter* ($\lambda m.\ m \,\hat{}\, k \leq n$) {..*n*}))
⟨*proof*⟩

function *nth-root-nat-aux* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* **where**
  nth-root-nat-aux m k acc n =
    (*let acc′* = (*k* + *1*) ^ *m*
     in if $k \geq n \vee acc' > n$ *then k else nth-root-nat-aux m* (*k*+*1*) *acc′ n*)
  ⟨*proof*⟩
termination ⟨*proof*⟩

lemma *nth-root-nat-aux-le*:
  assumes $k \,\hat{}\, m \leq n$ $m > 0$
  shows *nth-root-nat-aux m k* (*k* ^ *m*) *n* ^ *m* ≤ *n*
  ⟨*proof*⟩

lemma *nth-root-nat-aux-gt*:
  assumes $m > 0$
  shows (*nth-root-nat-aux m k* (*k* ^ *m*) *n* + *1*) ^ *m* > *n*
  ⟨*proof*⟩

lemma *nth-root-nat-aux-correct*:
  assumes $k \,\hat{}\, m \leq n$ $m > 0$
  shows *nth-root-nat-aux m k* (*k* ^ *m*) *n* = *nth-root-nat m n*
  ⟨*proof*⟩

lemma *nth-root-nat-naive-code* [*code*]:
  nth-root-nat m n = (*if m* = *0* ∨ *n* = *0 then 0 else if m* = *1* ∨ *n* = *1 then n else*
      nth-root-nat-aux m 1 1 n)
  ⟨*proof*⟩

lemma *nth-root-nat-nth-power* [*simp*]: $k > 0 \Longrightarrow$ *nth-root-nat* k $(n \mathbin{\char`\^} k) = n$
⟨*proof*⟩

lemma *nth-root-nat-nth-power'*:
assumes $k > 0$ k *dvd* m
shows *nth-root-nat* k $(n \mathbin{\char`\^} m) = n \mathbin{\char`\^} (m$ *div* $k)$
⟨*proof*⟩

lemma *nth-root-nat-mono*:
assumes $m \leq n$
shows *nth-root-nat* k $m \leq$ *nth-root-nat* k n
⟨*proof*⟩

end

11 Polynomials, fractions and rings

theory *Polynomial-Factorial*
imports
Complex-Main
Polynomial
Normalized-Fraction
begin

11.1 Lifting elements into the field of fractions

definition *to-fract* :: *'a* :: *idom* ⇒ *'a fract*
where *to-fract* x = *Fract* x *1*
— FIXME: more idiomatic name, abbreviation

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

lemma *to-fract-1* [*simp*]: *to-fract* *1* = *1*
⟨*proof*⟩

lemma *to-fract-add* [*simp*]: *to-fract* $(x + y)$ = *to-fract* x + *to-fract* y
⟨*proof*⟩

lemma *to-fract-diff* [*simp*]: *to-fract* $(x - y)$ = *to-fract* x − *to-fract* y
⟨*proof*⟩

lemma *to-fract-uminus* [*simp*]: *to-fract* $(-x)$ = −*to-fract* x
⟨*proof*⟩

lemma *to-fract-mult* [*simp*]: *to-fract* $(x * y)$ = *to-fract* x ∗ *to-fract* y
⟨*proof*⟩

lemma *to-fract-eq-iff* [*simp*]: *to-fract* x = *to-fract* $y \longleftrightarrow x = y$
⟨*proof*⟩

lemma *to-fract-eq-0-iff* [*simp*]: *to-fract* $x = 0 \longleftrightarrow x = 0$
⟨*proof*⟩

lemma *to-fract-quot-of-fract*:
assumes *snd* (*quot-of-fract* x) = *1*
shows *to-fract* (*fst* (*quot-of-fract* x)) = x
⟨*proof*⟩

lemma *Fract-conv-to-fract*: *Fract* a b = *to-fract* a / *to-fract* b
⟨*proof*⟩

lemma *quot-of-fract-to-fract* [*simp*]: *quot-of-fract* (*to-fract* x) = (x, *1*)
⟨*proof*⟩

lemma *snd-quot-of-fract-to-fract* [*simp*]: *snd* (*quot-of-fract* (*to-fract* x)) = *1*
⟨*proof*⟩

11.2 Lifting polynomial coefficients to the field of fractions

abbreviation (*input*) *fract-poly* :: ‹*'a*::*idom poly* ⇒ *'a fract poly*›
where *fract-poly* ≡ *map-poly to-fract*

abbreviation (*input*) *unfract-poly* :: ‹*'a*::{*ring-gcd*,*semiring-gcd-mult-normalize*,*idom-divide*} *fract poly* ⇒ *'a poly*›
where *unfract-poly* ≡ *map-poly* (*fst* ∘ *quot-of-fract*)

lemma *fract-poly-smult* [*simp*]: *fract-poly* (*smult c p*) = *smult* (*to-fract c*) (*fract-poly p*)
⟨*proof*⟩

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

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

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

lemma *fract-poly-diff* [*simp*]:
fract-poly ($p - q$) = *fract-poly* p − *fract-poly* q
⟨*proof*⟩

lemma *to-fract-sum* [*simp*]: *to-fract* (*sum f A*) = *sum* (λx. *to-fract* (*f x*)) *A*

⟨*proof*⟩

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

lemma *fract-poly-eq-iff* [*simp*]: *fract-poly p* = *fract-poly q* ⟷ *p* = *q*
⟨*proof*⟩

lemma *fract-poly-eq-0-iff* [*simp*]: *fract-poly p* = *0* ⟷ *p* = *0*
⟨*proof*⟩

lemma *fract-poly-dvd*: *p dvd q* ⟹ *fract-poly p dvd fract-poly q*
⟨*proof*⟩

lemma *prod-mset-fract-poly*:
(∏ *x*∈#*A*. *map-poly to-fract* (*f x*)) = *fract-poly* (*prod-mset* (*image-mset f A*))
⟨*proof*⟩

lemma *is-unit-fract-poly-iff*:
p dvd 1 ⟷ *fract-poly p dvd 1* ∧ *content p* = *1*
⟨*proof*⟩

lemma *fract-poly-is-unit*: *p dvd 1* ⟹ *fract-poly p dvd 1*
⟨*proof*⟩

lemma *fract-poly-smult-eqE*:
fixes *c* :: ′*a* :: {*idom-divide*,*ring-gcd*,*semiring-gcd-mult-normalize*} *fract*
assumes *fract-poly p* = *smult c* (*fract-poly q*)
obtains *a b*
where *c* = *to-fract b* / *to-fract a smult a p* = *smult b q coprime a b normalize a* = *a*
⟨*proof*⟩

11.3 Fractional content

abbreviation (*input*) *Lcm-coeff-denoms*
:: ′*a* :: {*semiring-Gcd*,*idom-divide*,*ring-gcd*,*semiring-gcd-mult-normalize*} *fract poly* ⇒ ′*a*
where *Lcm-coeff-denoms p* ≡ *Lcm* (*snd* ‘ *quot-of-fract* ‘ *set* (*coeffs p*))

definition *fract-content* ::
′*a* :: {*factorial-semiring*,*semiring-Gcd*,*ring-gcd*,*idom-divide*,*semiring-gcd-mult-normalize*} *fract poly* ⇒ ′*a fract* **where**
fract-content p =
(*let d* = *Lcm-coeff-denoms p in Fract* (*content* (*unfract-poly* (*smult* (*to-fract d*) *p*))) *d*)

definition *primitive-part-fract* ::

$'a$ :: {*factorial-semiring*,*semiring-Gcd*,*ring-gcd*,*idom-divide*,*semiring-gcd-mult-normalize*} *fract poly* ⇒ $'a$ *poly* **where**
primitive-part-fract p =
primitive-part (*unfract-poly* (*smult* (*to-fract* (*Lcm-coeff-denoms p*)) *p*))

lemma *primitive-part-fract-0* [*simp*]: *primitive-part-fract 0* = *0*
⟨*proof*⟩

lemma *fract-content-eq-0-iff* [*simp*]:
fract-content p = *0* ⟷ *p* = *0*
⟨*proof*⟩

lemma *content-primitive-part-fract* [*simp*]:
fixes *p* :: $'a$ :: {*semiring-gcd-mult-normalize*,
factorial-semiring, *ring-gcd*, *semiring-Gcd*,*idom-divide*} *fract poly*
shows *p* ≠ *0* ⟹ *content* (*primitive-part-fract p*) = *1*
⟨*proof*⟩

lemma *content-times-primitive-part-fract*:
smult (*fract-content p*) (*fract-poly* (*primitive-part-fract p*)) = *p*
⟨*proof*⟩

lemma *fract-content-fract-poly* [*simp*]: *fract-content* (*fract-poly p*) = *to-fract* (*content p*)
⟨*proof*⟩

lemma *content-decompose-fract*:
fixes *p* :: $'a$ :: {*factorial-semiring*,*semiring-Gcd*,*ring-gcd*,*idom-divide*,
semiring-gcd-mult-normalize} *fract poly*
obtains *c p'* **where** *p* = *smult c* (*map-poly to-fract p'*) *content p'* = *1*
⟨*proof*⟩

lemma *fract-poly-dvdD*:
fixes *p* :: $'a$ :: {*factorial-semiring*,*semiring-Gcd*,*ring-gcd*,*idom-divide*,
semiring-gcd-mult-normalize} *poly*
assumes *fract-poly p dvd fract-poly q content p* = *1*
shows *p dvd q*
⟨*proof*⟩

11.4 Polynomials over a field are a Euclidean ring

context
begin

interpretation *field-poly*:
normalization-euclidean-semiring-multiplicative **where** *zero* = *0* :: $'a$ :: *field poly*
and *one* = *1* **and** *plus* = *plus* **and** *minus* = *minus*
and *times* = *times*
and *normalize* = λ*p*. *smult* (*inverse* (*lead-coeff p*)) *p*

and *unit-factor* = λp. [:*lead-coeff p*:]
and *euclidean-size* = λp. *if p = 0 then 0 else 2 ^ degree p*
and *divide* = *divide* **and** *modulo* = *modulo*
rewrites *dvd.dvd* (*times* :: *'a poly* ⇒ -) = *Rings.dvd*
and *comm-monoid-mult.prod-mset times 1* = *prod-mset*
and *comm-semiring-1.irreducible times 1 0* = *irreducible*
and *comm-semiring-1.prime-elem times 1 0* = *prime-elem*
⟨*proof*⟩

lemma *field-poly-irreducible-imp-prime*:
prime-elem p **if** *irreducible p* **for** *p* :: *'a* :: *field poly*
⟨*proof*⟩

lemma *field-poly-prod-mset-prime-factorization*:
prod-mset (*field-poly.prime-factorization p*) = *smult* (*inverse* (*lead-coeff p*)) *p*
if *p* ≠ *0* **for** *p* :: *'a* :: *field poly*
⟨*proof*⟩

lemma *field-poly-in-prime-factorization-imp-prime*:
prime-elem p **if** *p* ∈# *field-poly.prime-factorization x*
for *p* :: *'a* :: *field poly*
⟨*proof*⟩

11.5 Primality and irreducibility in polynomial rings

lemma *nonconst-poly-irreducible-iff*:
fixes *p* :: *'a* :: {*factorial-semiring*,*semiring-Gcd*,*ring-gcd*,*idom-divide*,*semiring-gcd-mult-normalize*} *poly*
assumes *degree p* ≠ *0*
shows *irreducible p* ⟷ *irreducible* (*fract-poly p*) ∧ *content p* = *1*
⟨*proof*⟩

lemma *irreducible-imp-prime-poly*:
fixes *p* :: *'a* :: {*factorial-semiring*,*semiring-Gcd*,*ring-gcd*,*idom-divide*,*semiring-gcd-mult-normalize*} *poly*
assumes *irreducible p*
shows *prime-elem p*
⟨*proof*⟩

lemma *degree-primitive-part-fract* [*simp*]:
degree (*primitive-part-fract p*) = *degree p*
⟨*proof*⟩

lemma *irreducible-primitive-part-fract*:
fixes *p* :: *'a* :: {*idom-divide*, *ring-gcd*, *factorial-semiring*, *semiring-Gcd*,*semiring-gcd-mult-normalize*} *fract poly*
assumes *irreducible p*
shows *irreducible* (*primitive-part-fract p*)
⟨*proof*⟩

lemma *prime-elem-primitive-part-fract*:
fixes *p* :: *'a* :: {*idom-divide*, *ring-gcd*, *factorial-semiring*, *semiring-Gcd*,*semiring-gcd-mult-normalize*} *fract poly*
shows *irreducible p* ⟹ *prime-elem* (*primitive-part-fract p*)
⟨*proof*⟩

lemma *irreducible-linear-field-poly*:
fixes *a b* :: *'a*::*field*
assumes *b* ≠ *0*
shows *irreducible* [:*a*,*b*:]
⟨*proof*⟩

lemma *prime-elem-linear-field-poly*:
(*b* :: *'a* :: *field*) ≠ *0* ⟹ *prime-elem* [:*a*,*b*:]
⟨*proof*⟩

lemma *irreducible-linear-poly*:
fixes *a b* :: *'a*::{*idom-divide*,*ring-gcd*,*factorial-semiring*,*semiring-Gcd*,*semiring-gcd-mult-normalize*}
shows *b* ≠ *0* ⟹ *coprime a b* ⟹ *irreducible* [:*a*,*b*:]
⟨*proof*⟩

lemma *prime-elem-linear-poly*:
fixes *a b* :: *'a*::{*idom-divide*,*ring-gcd*,*factorial-semiring*,*semiring-Gcd*,*semiring-gcd-mult-normalize*}
shows *b* ≠ *0* ⟹ *coprime a b* ⟹ *prime-elem* [:*a*,*b*:]
⟨*proof*⟩

11.6 Prime factorisation of polynomials

lemma *poly-prime-factorization-exists-content-1*:
fixes *p* :: *'a* :: {*factorial-semiring*,*semiring-Gcd*,*ring-gcd*,*idom-divide*,*semiring-gcd-mult-normalize*} *poly*
assumes *p* ≠ *0* *content p* = *1*
shows ∃ *A*. (∀ *p*. *p* ∈# *A* ⟶ *prime-elem p*) ∧ *prod-mset A* = *normalize p*
⟨*proof*⟩

lemma *poly-prime-factorization-exists*:
fixes *p* :: *'a* :: {*factorial-semiring*,*semiring-Gcd*,*ring-gcd*,*idom-divide*,*semiring-gcd-mult-normalize*} *poly*
assumes *p* ≠ *0*
shows ∃ *A*. (∀ *p*. *p* ∈# *A* ⟶ *prime-elem p*) ∧ *normalize* (*prod-mset A*) = *normalize p*
⟨*proof*⟩

end

11.7 Typeclass instances

instance *poly* :: ({*factorial-ring-gcd*,*semiring-gcd-mult-normalize*}) *factorial-semiring*
⟨*proof*⟩

instantiation *poly* :: ({*factorial-ring-gcd*, *semiring-gcd-mult-normalize*}) *factorial-ring-gcd*
begin

definition *gcd-poly* :: *'a poly* ⇒ *'a poly* ⇒ *'a poly* **where**
[*code del*]: *gcd-poly* = *gcd-factorial*

definition *lcm-poly* :: *'a poly* ⇒ *'a poly* ⇒ *'a poly* **where**
[*code del*]: *lcm-poly* = *lcm-factorial*

definition *Gcd-poly* :: *'a poly set* ⇒ *'a poly* **where**
[*code del*]: *Gcd-poly* = *Gcd-factorial*

definition *Lcm-poly* :: *'a poly set* ⇒ *'a poly* **where**
[*code del*]: *Lcm-poly* = *Lcm-factorial*

instance ⟨*proof*⟩

end

instance *poly* :: ({*factorial-ring-gcd*, *semiring-gcd-mult-normalize*}) *semiring-gcd-mult-normalize*
⟨*proof*⟩

instance *poly* :: ({*field*,*factorial-ring-gcd*,*semiring-gcd-mult-normalize*})
normalization-euclidean-semiring ⟨*proof*⟩

instance *poly* :: ({*field*, *normalization-euclidean-semiring*, *factorial-ring-gcd*,
semiring-gcd-mult-normalize}) *euclidean-ring-gcd*
⟨*proof*⟩

instance *poly* :: ({*field*, *normalization-euclidean-semiring*, *factorial-ring-gcd*,
semiring-gcd-mult-normalize}) *factorial-semiring-multiplicative*
⟨*proof*⟩

11.8 Polynomial GCD

lemma *gcd-poly-decompose*:
fixes *p q* :: *'a* :: {*factorial-ring-gcd*,*semiring-gcd-mult-normalize*} *poly*
shows *gcd p q* =
smult (*gcd* (*content p*) (*content q*)) (*gcd* (*primitive-part p*) (*primitive-part q*))
⟨*proof*⟩

lemma *gcd-poly-pseudo-mod*:
fixes *p q* :: *'a* :: {*factorial-ring-gcd*,*semiring-gcd-mult-normalize*} *poly*
assumes *nz*: *q* ≠ *0* **and** *prim*: *content p* = *1 content q* = *1*
shows *gcd p q* = *gcd q* (*primitive-part* (*pseudo-mod p q*))
⟨*proof*⟩

lemma *degree-pseudo-mod-less*:
  assumes *q* ≠ *0 pseudo-mod p q* ≠ *0*
  shows *degree* (*pseudo-mod p q*) < *degree q*
  ⟨*proof*⟩

function *gcd-poly-code-aux* :: *'a* :: *factorial-ring-gcd poly* ⇒ *'a poly* ⇒ *'a poly*
where
  gcd-poly-code-aux p q =
    (*if q* = *0 then normalize p else gcd-poly-code-aux q* (*primitive-part* (*pseudo-mod p q*)))
⟨*proof*⟩
termination
  ⟨*proof*⟩

declare *gcd-poly-code-aux.simps* [*simp del*]

lemma *gcd-poly-code-aux-correct*:
  assumes *content p* = *1 q* = *0* ∨ *content q* = *1*
  shows *gcd-poly-code-aux p q* = *gcd p q*
  ⟨*proof*⟩

definition *gcd-poly-code*
  :: *'a* :: *factorial-ring-gcd poly* ⇒ *'a poly* ⇒ *'a poly*
  where *gcd-poly-code p q* =
    (*if p* = *0 then normalize q else if q* = *0 then normalize p else*
      smult (*gcd* (*content p*) (*content q*))
        (*gcd-poly-code-aux* (*primitive-part p*) (*primitive-part q*)))

lemma *gcd-poly-code* [*code*]: *gcd p q* = *gcd-poly-code p q*
  ⟨*proof*⟩

lemma *lcm-poly-code* [*code*]:
  fixes *p q* :: *'a* :: {*factorial-ring-gcd*,*semiring-gcd-mult-normalize*} *poly*
  shows *lcm p q* = *normalize* (*p* ∗ *q div gcd p q*)
  ⟨*proof*⟩

lemmas *Gcd-poly-set-eq-fold* [*code*] =
  Gcd-set-eq-fold [**where** *?'a* = *'a* :: {*factorial-ring-gcd*,*semiring-gcd-mult-normalize*} *poly*]
lemmas *Lcm-poly-set-eq-fold* [*code*] =
  Lcm-set-eq-fold [**where** *?'a* = *'a* :: {*factorial-ring-gcd*,*semiring-gcd-mult-normalize*} *poly*]

Example: *Lcm* {[:*1*, *2*, *3*:], [:*2*, *3*, *4*:]} = [:[:*2*:], [:*7*:], [:*16*:], [:*17*:], [:*12*:]:]

end

12 Squarefreeness

theory *Squarefree*
imports *Primes*
begin

definition *squarefree* :: *'a* :: *comm-monoid-mult* ⇒ *bool* **where**
squarefree n ⟷ (∀ *x*. *x* ^ *2 dvd n* ⟶ *x dvd 1*)

lemma *squarefreeI*: (⋀*x*. *x* ^ *2 dvd n* ⟹ *x dvd 1*) ⟹ *squarefree n*
⟨*proof*⟩

lemma *squarefreeD*: *squarefree n* ⟹ *x* ^ *2 dvd n* ⟹ *x dvd 1*
⟨*proof*⟩

lemma *not-squarefreeI*: *x* ^ *2 dvd n* ⟹ ¬*x dvd 1* ⟹ ¬*squarefree n*
⟨*proof*⟩

lemma *not-squarefreeE* [*case-names square-dvd*]:
¬*squarefree n* ⟹ (⋀*x*. *x* ^ *2 dvd n* ⟹ ¬*x dvd 1* ⟹ *P*) ⟹ *P*
⟨*proof*⟩

lemma *not-squarefree-0* [*simp*]: ¬*squarefree* (*0* :: *'a* :: *comm-semiring-1*)
⟨*proof*⟩

lemma *squarefree-factorial-semiring*:
assumes *n* ≠ *0*
shows *squarefree* (*n* :: *'a* :: *factorial-semiring*) ⟷ (∀ *p*. *prime p* ⟶ ¬*p* ^ *2 dvd n*)
⟨*proof*⟩

lemma *squarefree-factorial-semiring'*:
assumes *n* ≠ *0*
shows *squarefree* (*n* :: *'a* :: *factorial-semiring*) ⟷
(∀ *p*∈*prime-factors n*. *multiplicity p n* = *1*)
⟨*proof*⟩

lemma *squarefree-factorial-semiring''*:
assumes *n* ≠ *0*
shows *squarefree* (*n* :: *'a* :: *factorial-semiring*) ⟷
(∀ *p*. *prime p* ⟶ *multiplicity p n* ≤ *1*)
⟨*proof*⟩

lemma *squarefree-unit* [*simp*]: *is-unit n* ⟹ *squarefree n*
⟨*proof*⟩

lemma *squarefree-1* [*simp*]: *squarefree* (*1* :: *'a* :: *algebraic-semidom*)

⟨*proof*⟩

lemma *squarefree-minus* [*simp*]: *squarefree* (−*n* :: ′*a* :: *comm-ring-1*) ⟷ *squarefree n*
⟨*proof*⟩

lemma *squarefree-mono*: *a dvd b* ⟹ *squarefree b* ⟹ *squarefree a*
⟨*proof*⟩

lemma *squarefree-multD*:
assumes *squarefree* (*a* ∗ *b*)
shows *squarefree a squarefree b*
⟨*proof*⟩

lemma *squarefree-prime-elem*:
assumes *prime-elem* (*p* :: ′*a* :: *factorial-semiring*)
shows *squarefree p*
⟨*proof*⟩

lemma *squarefree-prime*:
assumes *prime* (*p* :: ′*a* :: *factorial-semiring*)
shows *squarefree p*
⟨*proof*⟩

lemma *squarefree-mult-coprime*:
fixes *a b* :: ′*a* :: *factorial-semiring-gcd*
assumes *coprime a b squarefree a squarefree b*
shows *squarefree* (*a* ∗ *b*)
⟨*proof*⟩

lemma *squarefree-prod-coprime*:
fixes *f* :: ′*a* ⇒ ′*b* :: *factorial-semiring-gcd*
assumes ⋀*a b*. *a* ∈ *A* ⟹ *b* ∈ *A* ⟹ *a* ≠ *b* ⟹ *coprime* (*f a*) (*f b*)
assumes ⋀*a*. *a* ∈ *A* ⟹ *squarefree* (*f a*)
shows *squarefree* (*prod f A*)
⟨*proof*⟩

lemma *squarefree-powerD*: *m* > *0* ⟹ *squarefree* (*n* ^ *m*) ⟹ *squarefree n*
⟨*proof*⟩

lemma *squarefree-power-iff*:
squarefree (*n* ^ *m*) ⟷ *m* = *0* ∨ *is-unit n* ∨ (*squarefree n* ∧ *m* = *1*)
⟨*proof*⟩

definition *squarefree-nat* :: *nat* ⇒ *bool* **where**
[*code-abbrev*]: *squarefree-nat* = *squarefree*

lemma *squarefree-nat-code-naive* [*code*]:
squarefree-nat n ⟷ *n* ≠ *0* ∧ (∀ *k*∈{*2*..*n*}. ¬*k* ^ *2 dvd n*)

⟨*proof*⟩

definition *square-part* :: *'a* :: *factorial-semiring* ⇒ *'a* **where**
square-part n = (*if n* = *0 then 0 else*
normalize (∏ *p*∈*prime-factors n*. *p* ^ (*multiplicity p n div 2*)))

lemma *square-part-nonzero*:
n ≠ *0* ⟹ *square-part n* = *normalize* (∏ *p*∈*prime-factors n*. *p* ^ (*multiplicity p n div 2*))
⟨*proof*⟩

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

lemma *square-part-unit* [*simp*]: *is-unit x* ⟹ *square-part x* = *1*
⟨*proof*⟩

lemma *square-part-1* [*simp*]: *square-part 1* = *1*
⟨*proof*⟩

lemma *square-part-0-iff* [*simp*]: *square-part n* = *0* ⟷ *n* = *0*
⟨*proof*⟩

lemma *normalize-uminus* [*simp*]:
normalize (−*x* :: *'a* :: {*normalization-semidom*, *comm-ring-1*}) = *normalize x*
⟨*proof*⟩

lemma *multiplicity-uminus-right* [*simp*]:
multiplicity (*x* :: *'a* :: {*factorial-semiring*, *comm-ring-1*}) (−*y*) = *multiplicity x y*
⟨*proof*⟩

lemma *multiplicity-uminus-left* [*simp*]:
multiplicity (−*x* :: *'a* :: {*factorial-semiring*, *comm-ring-1*}) *y* = *multiplicity x y*
⟨*proof*⟩

lemma *prime-factorization-uminus* [*simp*]:
prime-factorization (−*x* :: *'a* :: {*factorial-semiring*, *comm-ring-1*}) = *prime-factorization x*
⟨*proof*⟩

lemma *square-part-uminus* [*simp*]:
square-part (−*x* :: *'a* :: {*factorial-semiring*, *comm-ring-1*}) = *square-part x*
⟨*proof*⟩

lemma *prime-multiplicity-square-part*:
assumes *prime p*
shows *multiplicity p* (*square-part n*) = *multiplicity p n div 2*

⟨*proof*⟩

lemma *square-part-square-dvd* [*simp*, *intro*]: *square-part n ^ 2 dvd n*
⟨*proof*⟩

lemma *prime-multiplicity-le-imp-dvd*:
assumes $x \neq 0$ $y \neq 0$
shows *x dvd y* ⟷ (∀ *p*. *prime p* ⟶ *multiplicity p x* ≤ *multiplicity p y*)
⟨*proof*⟩

lemma *dvd-square-part-iff*: *x dvd square-part n* ⟷ *x ^ 2 dvd n*
⟨*proof*⟩

definition *squarefree-part* :: *'a* :: *factorial-semiring* ⇒ *'a* **where**
squarefree-part n = (*if n* = *0 then 1 else n div square-part n ^ 2*)

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

lemma *squarefree-part-unit* [*simp*]: *is-unit n* ⟹ *squarefree-part n* = *n*
⟨*proof*⟩

lemma *squarefree-part-1* [*simp*]: *squarefree-part 1* = *1*
⟨*proof*⟩

lemma *squarefree-decompose*: *n* = *squarefree-part n* ∗ *square-part n ^ 2*
⟨*proof*⟩

lemma *squarefree-part-uminus* [*simp*]:
assumes $x \neq 0$
shows *squarefree-part* (−*x* :: *'a* :: {*factorial-semiring*, *comm-ring-1*}) = −*squarefree-part x*
⟨*proof*⟩

lemma *squarefree-part-nonzero* [*simp*]: *squarefree-part n* ≠ *0*
⟨*proof*⟩

lemma *prime-multiplicity-squarefree-part*:
assumes *prime p*
shows *multiplicity p* (*squarefree-part n*) = *multiplicity p n mod 2*
⟨*proof*⟩

lemma *prime-multiplicity-squarefree-part-le-Suc-0* [*intro*]:
assumes *prime p*
shows *multiplicity p* (*squarefree-part n*) ≤ *Suc 0*
⟨*proof*⟩

lemma *squarefree-squarefree-part* [*simp*, *intro*]: *squarefree* (*squarefree-part n*)

⟨*proof*⟩

lemma *squarefree-decomposition-unique*:
assumes *square-part* m = *square-part* n
assumes *squarefree-part* m = *squarefree-part* n
shows $m = n$
⟨*proof*⟩

lemma *normalize-square-part* [*simp*]: *normalize* (*square-part* x) = *square-part* x
⟨*proof*⟩

lemma *square-part-even-power'*: *square-part* (x ^ ($2 * n$)) = *normalize* (x ^ n)
⟨*proof*⟩

lemma *square-part-even-power*: *even* n ⟹ *square-part* (x ^ n) = *normalize* (x ^ (n *div* 2))
⟨*proof*⟩

lemma *square-part-odd-power'*: *square-part* (x ^ (*Suc* ($2 * n$))) = *normalize* (x ^ n * *square-part* x)
⟨*proof*⟩

lemma *square-part-odd-power*:
odd n ⟹ *square-part* (x ^ n) = *normalize* (x ^ (n *div* 2) * *square-part* x)
⟨*proof*⟩

end

13 Pieces of computational Algebra

theory *Computational-Algebra*
imports
Euclidean-Algorithm
Factorial-Ring
Formal-Laurent-Series
Fraction-Field
Fundamental-Theorem-Algebra
Group-Closure
Normalized-Fraction
Nth-Powers
Polynomial-FPS
Polynomial
Polynomial-Factorial
Primes
Squarefree
begin

end

theory *Field-as-Ring*
imports
Complex-Main
Euclidean-Algorithm
begin

context *field*
begin

subclass *idom-divide* ⟨*proof*⟩

definition *normalize-field* :: *'a* ⇒ *'a*
where [*simp*]: *normalize-field x* = (*if x* = *0 then 0 else 1*)
definition *unit-factor-field* :: *'a* ⇒ *'a*
where [*simp*]: *unit-factor-field x* = *x*
definition *euclidean-size-field* :: *'a* ⇒ *nat*
where [*simp*]: *euclidean-size-field x* = (*if x* = *0 then 0 else 1*)
definition *mod-field* :: *'a* ⇒ *'a* ⇒ *'a*
where [*simp*]: *mod-field x y* = (*if y* = *0 then x else 0*)

end

instantiation *real* ::
{*unique-euclidean-ring*, *normalization-euclidean-semiring*, *normalization-semidom-multiplicative*}
begin

definition [*simp*]: *normalize-real* = (*normalize-field* :: *real* ⇒ -)
definition [*simp*]: *unit-factor-real* = (*unit-factor-field* :: *real* ⇒ -)
definition [*simp*]: *modulo-real* = (*mod-field* :: *real* ⇒ -)
definition [*simp*]: *euclidean-size-real* = (*euclidean-size-field* :: *real* ⇒ -)
definition [*simp*]: *division-segment* (*x* :: *real*) = *1*

instance
⟨*proof*⟩

end

instantiation *real* :: *euclidean-ring-gcd*
begin

definition *gcd-real* :: *real* ⇒ *real* ⇒ *real* **where**
gcd-real = *Euclidean-Algorithm.gcd*
definition *lcm-real* :: *real* ⇒ *real* ⇒ *real* **where**
lcm-real = *Euclidean-Algorithm.lcm*
definition *Gcd-real* :: *real set* ⇒ *real* **where**
Gcd-real = *Euclidean-Algorithm.Gcd*
definition *Lcm-real* :: *real set* ⇒ *real* **where**
Lcm-real = *Euclidean-Algorithm.Lcm*

instance ⟨*proof*⟩

end

instance *real* :: *field-gcd* ⟨*proof*⟩

instantiation *rat* ::
{*unique-euclidean-ring*, *normalization-euclidean-semiring*, *normalization-semidom-multiplicative*}
begin

definition [*simp*]: *normalize-rat* = (*normalize-field* :: *rat* ⇒ -)
definition [*simp*]: *unit-factor-rat* = (*unit-factor-field* :: *rat* ⇒ -)
definition [*simp*]: *modulo-rat* = (*mod-field* :: *rat* ⇒ -)
definition [*simp*]: *euclidean-size-rat* = (*euclidean-size-field* :: *rat* ⇒ -)
definition [*simp*]: *division-segment* (*x* :: *rat*) = *1*

instance
⟨*proof*⟩

end

instantiation *rat* :: *euclidean-ring-gcd*
begin

definition *gcd-rat* :: *rat* ⇒ *rat* ⇒ *rat* **where**
gcd-rat = *Euclidean-Algorithm.gcd*
definition *lcm-rat* :: *rat* ⇒ *rat* ⇒ *rat* **where**
lcm-rat = *Euclidean-Algorithm.lcm*
definition *Gcd-rat* :: *rat set* ⇒ *rat* **where**
Gcd-rat = *Euclidean-Algorithm.Gcd*
definition *Lcm-rat* :: *rat set* ⇒ *rat* **where**
Lcm-rat = *Euclidean-Algorithm.Lcm*

instance ⟨*proof*⟩

end

instance *rat* :: *field-gcd* ⟨*proof*⟩

instantiation *complex* ::
{*unique-euclidean-ring*, *normalization-euclidean-semiring*, *normalization-semidom-multiplicative*}
begin

definition [*simp*]: *normalize-complex* = (*normalize-field* :: *complex* ⇒ -)
definition [*simp*]: *unit-factor-complex* = (*unit-factor-field* :: *complex* ⇒ -)
definition [*simp*]: *modulo-complex* = (*mod-field* :: *complex* ⇒ -)

definition [*simp*]: *euclidean-size-complex* = (*euclidean-size-field* :: *complex* ⇒ -)
definition [*simp*]: *division-segment* (*x* :: *complex*) = *1*

instance
⟨*proof*⟩

end

instantiation *complex* :: *euclidean-ring-gcd*
begin

definition *gcd-complex* :: *complex* ⇒ *complex* ⇒ *complex* **where**
gcd-complex = *Euclidean-Algorithm.gcd*
definition *lcm-complex* :: *complex* ⇒ *complex* ⇒ *complex* **where**
lcm-complex = *Euclidean-Algorithm.lcm*
definition *Gcd-complex* :: *complex set* ⇒ *complex* **where**
Gcd-complex = *Euclidean-Algorithm.Gcd*
definition *Lcm-complex* :: *complex set* ⇒ *complex* **where**
Lcm-complex = *Euclidean-Algorithm.Lcm*

instance ⟨*proof*⟩

end

instance *complex* :: *field-gcd* ⟨*proof*⟩

end

References

[1] K. J. Nowak. Some elementary proofs of Puiseuxs theorems. *Univ. Iagel. Acta Math*, 38:279–282, 2000.