Computational Algebra

October 24, 2026

Contents

1 Factorial (semi)rings **6**
- 1.1 Irreducible and prime elements . . . 6
- 1.2 Generalized primes: normalized prime elements . . . 17
- 1.3 In a semiring with GCD, each irreducible element is a prime element . . . 21
- 1.4 Factorial semirings: algebraic structures with unique prime factorizations . . . 23
- 1.5 GCD and LCM computation with unique factorizations . . . 44

2 Abstract euclidean algorithm in euclidean (semi)rings **56**
- 2.1 Generic construction of the (simple) euclidean algorithm . . . 56
- 2.2 The (simple) euclidean algorithm as gcd computation . . . 62
- 2.3 The extended euclidean algorithm . . . 65
- 2.4 Typical instances . . . 68

3 Primes **71**
- 3.1 Primes on *nat* and *int* . . . 71
- 3.2 Make prime naively executable . . . 76
- 3.3 Largest exponent of a prime factor . . . 78
- 3.4 Infinitely many primes . . . 80
- 3.5 Powers of Primes . . . 81
- 3.6 Chinese Remainder Theorem Variants . . . 82
- 3.7 Multiplicity and primality for natural numbers and integers . . . 84
- 3.8 Rings and fields with prime characteristic . . . 88
- 3.9 Finite fields . . . 89
- 3.10 The Freshman's Dream in rings of prime characteristic . . . 91

4 Polynomials as type over a ring structure **93**
- 4.1 Auxiliary: operations for lists (later) representing coefficients 93
- 4.2 Definition of type *poly* . . . 94
- 4.3 Degree of a polynomial . . . 94
- 4.4 The zero polynomial . . . 95

4.5 List-style constructor for polynomials . . . 97
4.6 Quickcheck generator for polynomials . . . 100
4.7 List-style syntax for polynomials . . . 100
4.8 Representation of polynomials by lists of coefficients . . . 101
4.9 Fold combinator for polynomials . . . 104
4.10 Canonical morphism on polynomials – evaluation . . . 104
4.11 Monomials . . . 106
4.12 Leading coefficient . . . 107
4.13 Addition and subtraction . . . 107
4.14 Multiplication by a constant, polynomial multiplication and the unit polynomial . . . 112
4.15 Mapping polynomials . . . 120
4.16 Conversions . . . 124
4.17 Lemmas about divisibility . . . 126
4.18 Polynomials form an integral domain . . . 127
4.19 Polynomials form an ordered integral domain . . . 130
4.20 Synthetic division and polynomial roots . . . 132
4.20.1 Synthetic division . . . 132
4.20.2 Polynomial roots . . . 133
4.20.3 Order of polynomial roots . . . 137
4.21 Additional induction rules on polynomials . . . 144
4.22 Composition of polynomials . . . 146
4.23 Divisibility . . . 152
4.24 Closure properties of coefficients . . . 154
4.25 Shifting polynomials . . . 155
4.26 Truncating polynomials . . . 156
4.27 Reflecting polynomials . . . 156
4.28 Derivatives . . . 159
4.29 Algebraic numbers . . . 173
4.30 Algebraic integers . . . 180
4.31 Division of polynomials . . . 185
4.31.1 Division in general . . . 185
4.31.2 Pseudo-Division . . . 191
4.31.3 Division in polynomials over fields . . . 195
4.31.4 List-based versions for fast implementation . . . 207
4.31.5 Improved Code-Equations for Polynomial (Pseudo) Division . . . 213
4.32 Primality and irreducibility in polynomial rings . . . 218
4.33 Content and primitive part of a polynomial . . . 221
4.34 A typeclass for algebraically closed fields . . . 226
4.35 Polynomials and limits . . . 233

5 A formalization of formal power series 236
5.1 The type of formal power series . . . 237
5.2 Subdegrees . . . 239
5.3 Ring structure . . . 245
5.4 Shifting and slicing . . . 257
5.5 Metrizability . . . 265
5.6 The topology of formal power series . . . 266
5.7 Division . . . 271
5.8 Computing reciprocals via Hensel lifting . . . 301
5.9 Euclidean division . . . 304
5.10 Formal Derivatives . . . 306
5.11 Powers . . . 312
5.12 Finite and infinite products . . . 316
5.13 Integration . . . 319
5.14 Composition . . . 323
5.15 Rules from Herbert Wilf's Generatingfunctionology . . . 324
5.15.1 Rule 1 . . . 324
5.15.2 Rule 2 . . . 324
5.15.3 Rule 3 . . . 325
5.15.4 Rule 5 — summation and "division" by $1 - X$ . . . 325
5.15.5 Rule 4 in its more general form . . . 326
5.16 Radicals . . . 334
5.17 Chain rule . . . 343
5.18 Compositional inverses . . . 345
5.19 Elementary series . . . 356
5.19.1 Exponential series . . . 356
5.19.2 Logarithmic series . . . 359
5.19.3 Binomial series . . . 361
5.19.4 Trigonometric functions . . . 369
5.20 Hypergeometric series . . . 375

6 Converting polynomials to formal power series 379

7 A formalization of formal Laurent series 387
7.1 The type of formal Laurent series . . . 387
7.1.1 Type definition . . . 387
7.1.2 Definition of basic Laurent series . . . 388
7.2 Subdegrees . . . 390
7.3 Shifting . . . 392
7.3.1 Shift definition . . . 392
7.3.2 Base factor . . . 394
7.4 Conversion between formal power and Laurent series . . . 395
7.4.1 Converting Laurent to power series . . . 395
7.4.2 Converting power to Laurent series . . . 400

7.5 Algebraic structures . . . 404
7.5.1 Addition . . . 404
7.5.2 Subtraction and negatives . . . 405
7.5.3 Multiplication . . . 407
7.5.4 Powers . . . 423
7.5.5 Inverses . . . 429
7.5.6 Division . . . 449
7.5.7 Units . . . 456
7.6 Composition . . . 456
7.7 Formal differentiation and integration . . . 464
7.7.1 Derivative . . . 464
7.7.2 Algebraic rules of the derivative . . . 467
7.7.3 Equality of derivatives . . . 471
7.7.4 Residues . . . 472
7.7.5 Integral definition and basic properties . . . 475
7.7.6 Algebraic rules of the integral . . . 480
7.7.7 Derivatives of integrals and vice versa . . . 481
7.8 Topology . . . 482
7.9 Notation . . . 484

8 The fraction field of any integral domain 484
8.1 General fractions construction . . . 484
8.1.1 Construction of the type of fractions . . . 484
8.1.2 Representation and basic operations . . . 485
8.1.3 The field of rational numbers . . . 488
8.1.4 The ordered field of fractions over an ordered idom . . . 488

9 Fundamental Theorem of Algebra 494
9.1 More lemmas about module of complex numbers . . . 494
9.2 Basic lemmas about polynomials . . . 494
9.3 Fundamental theorem of algebra . . . 496
9.4 Nullstellensatz, degrees and divisibility of polynomials . . . 507

10 n-th powers and roots of naturals 527
10.1 The set of n-th powers . . . 527
10.2 The n-root of a natural number . . . 531

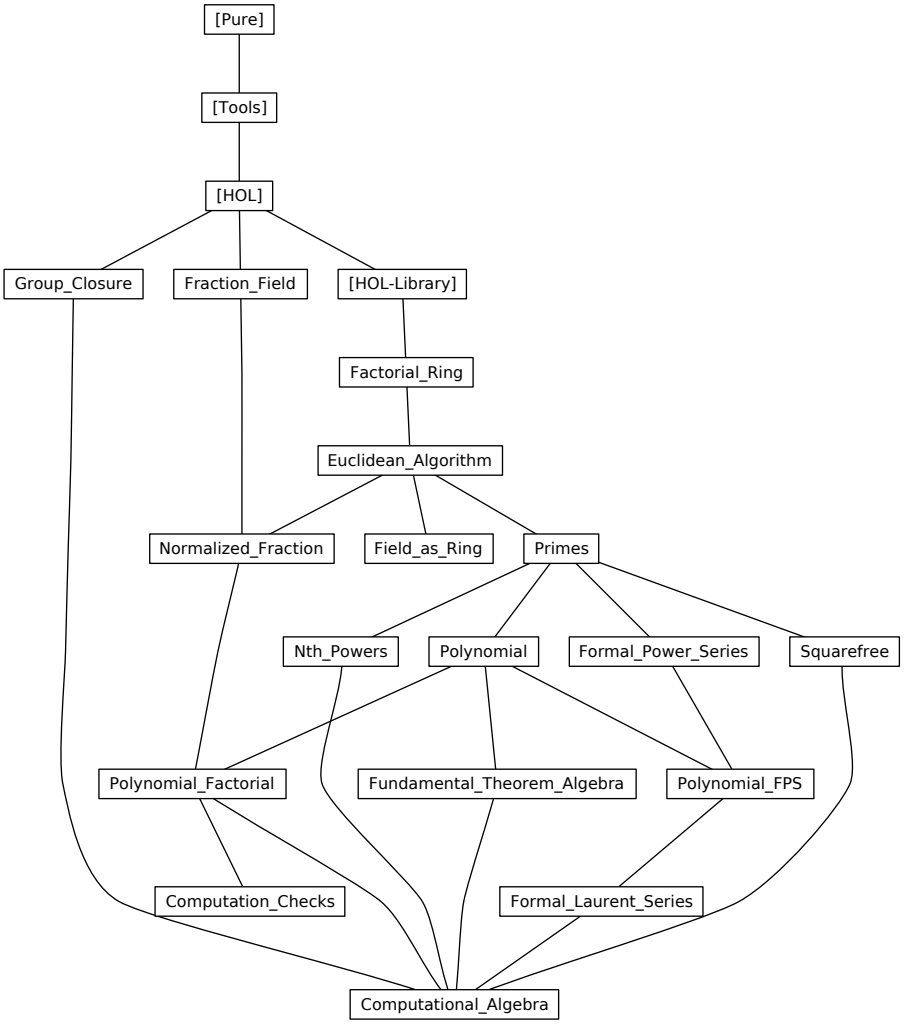
11 Polynomials, fractions and rings 534
11.1 Lifting elements into the field of fractions . . . 534
11.2 Lifting polynomial coefficients to the field of fractions . . . 535
11.3 Fractional Polynomial.content . . . 537
11.4 Polynomials over a field are a Euclidean ring . . . 540
11.5 Primality and irreducibility in polynomial rings . . . 541
11.6 Prime factorisation of polynomials . . . 545

11.7 Typeclass instances 547
11.8 Polynomial GCD 548
11.9 The isomorphism *int fract* → *rat* 551
11.10 The induced isomorphism on polynomials 552
11.11 Transfer of irreducibility from *int poly* to *rat poly* 553

12 Squarefreeness **554**

13 Pieces of computational Algebra **561**

14 Computation checks **564**



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* ⇒ *bool* **where**
irreducible p ≡ *p* ≠ *0* ∧ ¬*p dvd 1* ∧ (∀ *a b*. *p* = *a* ∗ *b* ⟶ *a dvd 1* ∨ *b dvd 1*)

lemma *not-irreducible-zero* [*simp*]: ¬*irreducible 0*
by (*simp add*: *irreducible-def*)

lemma *irreducible-not-unit*: *irreducible p* ⟹ ¬*p dvd 1*
by (*simp add*: *irreducible-def*)

lemma *not-irreducible-one* [*simp*]: ¬*irreducible 1*
by (*simp add*: *irreducible-def*)

lemma *irreducibleI*:
p ≠ *0* ⟹ ¬*p dvd 1* ⟹ (⋀*a b*. *p* = *a* ∗ *b* ⟹ *a dvd 1* ∨ *b dvd 1*) ⟹ *irreducible p*
by (*simp add*: *irreducible-def*)

lemma *irreducibleD*: *irreducible p* ⟹ *p* = *a* ∗ *b* ⟹ *a dvd 1* ∨ *b dvd 1*
by (*simp add*: *irreducible-def*)

lemma *irreducible-mono*:
assumes *irr*: *irreducible b* **and** *a dvd b* ¬*a dvd 1*
shows *irreducible a*
proof (*rule irreducibleI*)
fix *c d* **assume** *a* = *c* ∗ *d*
from *assms* **obtain** *k* **where** [*simp*]: *b* = *a* ∗ *k* **by** *auto*
from ‹*a* = *c* ∗ *d*› **have** *b* = *c* ∗ *d* ∗ *k*
by *simp*
hence *c dvd 1* ∨ (*d* ∗ *k*) *dvd 1*
using *irreducibleD*[*OF irr*, *of c d* ∗ *k*] **by** (*auto simp*: *mult.assoc*)
thus *c dvd 1* ∨ *d dvd 1*
by *auto*
qed (*use assms* **in** ‹*auto simp*: *irreducible-def*›)

lemma *irreducible-multD*:

```
  assumes l: irreducible (a*b)
  shows a dvd 1 ∧ irreducible b ∨ b dvd 1 ∧ irreducible a
proof−
  have *: irreducible b if l: irreducible (a*b) and a: a dvd 1 for a b :: 'a
  proof (rule irreducibleI)
    show ¬(b dvd 1)
    proof
      assume b dvd 1
      hence a * b dvd 1 * 1
        using ‹a dvd 1› by (intro mult-dvd-mono) auto
      with l show False
        by (auto simp: irreducible-def)
    qed
  next
    fix x y assume b = x * y
    have a * x dvd 1 ∨ y dvd 1
      using l by (rule irreducibleD) (use ‹b = x * y› in ‹auto simp: mult-ac›)
    thus x dvd 1 ∨ y dvd 1
      by auto
  qed (use l a in auto)

  from irreducibleD[OF assms refl] have a dvd 1 ∨ b dvd 1
    by (auto simp: irreducible-def)
  with *[of a b] *[of b a] l show ?thesis
    by (auto simp: mult.commute)
qed

lemma irreducible-power-iff [simp]:
  irreducible (p ^ n) ⟷ irreducible p ∧ n = 1
proof
  assume *: irreducible (p ^ n)
  have irreducible p
    using * by (induction n) (auto dest!: irreducible-multD)
  hence [simp]: ¬p dvd 1
    using * by (auto simp: irreducible-def)

  consider n = 0 | n = 1 | n > 1
    by linarith
  thus irreducible p ∧ n = 1
  proof cases
    assume n > 1
    hence p ^ n = p * p ^ (n − 1)
      by (cases n) auto
    with * ‹¬ p dvd 1› have p ^ (n − 1) dvd 1
      using irreducible-multD[of p p ^ (n − 1)] by auto
    with ‹¬p dvd 1› and ‹n > 1› have False
      by (meson dvd-power dvd-trans zero-less-diff)
    thus ?thesis ..
  qed (use * in auto)
```

qed *auto*

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

lemma *prime-elem-not-unit*: *prime-elem p* ⟹ ¬*p dvd 1*
by (*simp add*: *prime-elem-def*)

lemma *prime-elemI*:
p ≠ *0* ⟹ ¬*p dvd 1* ⟹ (⋀*a b*. *p dvd* (*a* ∗ *b*) ⟹ *p dvd a* ∨ *p dvd b*) ⟹ *prime-elem p*
by (*simp add*: *prime-elem-def*)

lemma *prime-elem-dvd-multD*:
prime-elem p ⟹ *p dvd* (*a* ∗ *b*) ⟹ *p dvd a* ∨ *p dvd b*
by (*simp add*: *prime-elem-def*)

lemma *prime-elem-dvd-mult-iff*:
prime-elem p ⟹ *p dvd* (*a* ∗ *b*) ⟷ *p dvd a* ∨ *p dvd b*
by (*auto simp*: *prime-elem-def*)

lemma *not-prime-elem-one* [*simp*]:
¬ *prime-elem 1*
by (*auto dest*: *prime-elem-not-unit*)

lemma *prime-elem-not-zeroI*:
assumes *prime-elem p*
shows *p* ≠ *0*
using *assms* **by** (*auto intro*: *ccontr*)

lemma *prime-elem-dvd-power*:
prime-elem p ⟹ *p dvd x* ^ *n* ⟹ *p dvd x*
by (*induction n*) (*auto dest*: *prime-elem-dvd-multD intro*: *dvd-trans*[*of - 1*])

lemma *prime-elem-dvd-power-iff*:
prime-elem p ⟹ *n* > *0* ⟹ *p dvd x* ^ *n* ⟷ *p dvd x*
by (*auto dest*: *prime-elem-dvd-power intro*: *dvd-trans*)

lemma *prime-elem-imp-nonzero* [*simp*]:
ASSUMPTION (*prime-elem x*) ⟹ *x* ≠ *0*
unfolding *ASSUMPTION-def* **by** (*rule prime-elem-not-zeroI*)

lemma *prime-elem-imp-not-one* [*simp*]:
ASSUMPTION (*prime-elem x*) ⟹ *x* ≠ *1*
unfolding *ASSUMPTION-def* **by** *auto*

end

context
fixes $f :: 'a$::*comm-semiring-1* $\Rightarrow 'b$::*comm-semiring-1*
begin

lemma *ring-iso-dvd*:
assumes *hom*: $\bigwedge x\ y.\ f\ (x * y) = f\ x * f\ y$ **and** *bij*: *bij f*
shows *f a dvd f b* $\longleftrightarrow$ *a dvd b*
using *bij-pointE* [*OF bij*] *hom* **unfolding** *dvd-def*
by *metis*

lemma *ring-iso-unit*:
assumes *hom*: $\bigwedge x\ y.\ f\ (x * y) = f\ x * f\ y$ **and** *one*: *f 1 = 1* **and** *bij*: *bij f*
shows *f a dvd 1* $\longleftrightarrow$ *a dvd 1*
using *ring-iso-dvd*[*OF hom bij*, *of a 1*] *one* **by** *simp*

theorem *ring-iso-irreducible*:
assumes *hom*: $\bigwedge x\ y.\ f\ (x * y) = f\ x * f\ y$ **and** *one*: *f 1 = 1*
and *zero*: *f 0 = 0* **and** *bij*: *bij f*
shows *irreducible* (*f a*) $\longleftrightarrow$ *irreducible a*
proof −
have *ne*: *f a = 0* $\longleftrightarrow$ *a = 0*
by (*metis bij bij-is-inj inj-eq zero*)
show *?thesis*
proof
assume *irreducible* (*f a*)
then show *irreducible a*
using *assms irreducibleI ring-iso-unit* **by** (*metis comm-semiring-1-class.irreducible-def*)
next
assume *irreducible a*
then have *a*: *a* $\neq$ *0* $\neg$ *a dvd 1* $\bigwedge u\ v.\ a = u * v \Longrightarrow u$ *dvd 1* $\vee$ *v dvd 1*
by (*auto simp*: *irreducible-def*)
show *irreducible* (*f a*)
proof (*rule irreducibleI*)
show *f a* $\neq$ *0* **using** *a*(*1*) *ne* **by** *simp*
show $\neg$ *f a dvd 1* **using** *a*(*2*) *ring-iso-unit*[*OF hom one bij*] **by** *simp*
next
fix *u v* **assume** *f a = u * v*
then obtain *u' v'* **where** *uv*: *u = f u' v = f v' a = u' * v'*
using *bij-pointE*[*OF bij*] **by** (*metis hom*)
then show *u dvd 1* $\vee$ *v dvd 1* **using** *uv ring-iso-unit*[*OF hom one bij*]
using *a*(*3*) **by** *blast*
qed
qed
qed

end

lemma (**in** *normalization-semidom*) *irreducible-cong*:
assumes *normalize a* = *normalize b*
shows *irreducible a* ⟷ *irreducible b*
proof (*cases a* = *0* ∨ *a dvd 1*)
case *True*
hence ¬*irreducible a* **by** (*auto simp*: *irreducible-def*)
from *True* **have** *normalize a* = *0* ∨ *normalize a dvd 1*
by *auto*
also note *assms*
finally have *b* = *0* ∨ *b dvd 1* **by** *simp*
hence ¬*irreducible b* **by** (*auto simp*: *irreducible-def*)
with ‹¬*irreducible a*› **show** *?thesis* **by** *simp*
next
case *False*
hence *b*: *b* ≠ *0* ¬*is-unit b* **using** *assms*
by (*auto simp*: *is-unit-normalize*[*of b*])
show *?thesis*
proof
assume *irreducible a*
thus *irreducible b*
by (*rule irreducible-mono*) (*use assms False b* **in** ‹*auto dest*: *associatedD2*›)
next
assume *irreducible b*
thus *irreducible a*
by (*rule irreducible-mono*) (*use assms False b* **in** ‹*auto dest*: *associatedD1*›)
qed
qed

lemma (**in** *normalization-semidom*) *associatedE1*:
assumes *normalize a* = *normalize b*
obtains *u* **where** *is-unit u a* = *u* ∗ *b*
proof (*cases a* = *0*)
case [*simp*]: *False*
from *assms* **have** [*simp*]: *b* ≠ *0* **by** *auto*
show *?thesis*
proof (*rule that*)
show *is-unit* (*unit-factor a div unit-factor b*)
by *auto*
have *unit-factor a div unit-factor b* ∗ *b* = *unit-factor a* ∗ (*b div unit-factor b*)
using ‹*b* ≠ *0*› *unit-div-commute unit-div-mult-swap unit-factor-is-unit* **by**
metis
also have *b div unit-factor b* = *normalize b* **by** *simp*
finally show *a* = *unit-factor a div unit-factor b* ∗ *b*
by (*metis assms unit-factor-mult-normalize*)
qed
next
case [*simp*]: *True*
hence [*simp*]: *b* = *0*

using *assms*[*symmetric*] **by** *auto*
show *?thesis*
by (*intro that*[*of 1*]) *auto*
qed

lemma (**in** *normalization-semidom*) *associatedE2*:
assumes *normalize a* = *normalize b*
obtains *u* **where** *is-unit u b* = *u* ∗ *a*
proof −
from *assms* **have** *normalize b* = *normalize a*
by *simp*
then obtain *u* **where** *is-unit u b* = *u* ∗ *a*
by (*elim associatedE1*)
thus *?thesis* **using** *that* **by** *blast*
qed

lemma (**in** *normalization-semidom*) *normalize-power-normalize*:
normalize (*normalize x* ^ *n*) = *normalize* (*x* ^ *n*)
proof (*induction n*)
case (*Suc n*)
have *normalize* (*normalize x* ^ *Suc n*) = *normalize* (*x* ∗ *normalize* (*normalize x* ^ *n*))
by *simp*
also note *Suc.IH*
finally show *?case* **by** *simp*
qed *auto*

context *algebraic-semidom*
begin

lemma *prime-elem-imp-irreducible*:
assumes *prime-elem p*
shows *irreducible p*
proof (*rule irreducibleI*)
fix *a b*
assume *p-eq*: *p* = *a* ∗ *b*
with *assms* **have** *nz*: *a* ≠ *0 b* ≠ *0* **by** *auto*
from *p-eq* **have** *p dvd a* ∗ *b* **by** *simp*
with ‹*prime-elem p*› **have** *p dvd a* ∨ *p dvd b* **by** (*rule prime-elem-dvd-multD*)
with ‹*p* = *a* ∗ *b*› **have** *a* ∗ *b dvd 1* ∗ *b* ∨ *a* ∗ *b dvd a* ∗ *1* **by** *auto*
thus *a dvd 1* ∨ *b dvd 1*
by (*simp only*: *dvd-times-left-cancel-iff*[*OF nz*(*1*)] *dvd-times-right-cancel-iff*[*OF nz*(*2*)])
qed (*insert assms*, *simp-all add*: *prime-elem-def*)

lemma (**in** *algebraic-semidom*) *unit-imp-no-irreducible-divisors*:
assumes *is-unit x irreducible p*

shows $\neg p$ *dvd* x
proof (*rule notI*)
assume p *dvd* x
with ‹*is-unit* x› **have** *is-unit* p
by (*auto intro*: *dvd-trans*)
with ‹*irreducible* p› **show** *False*
by (*simp add*: *irreducible-not-unit*)
qed

lemma *unit-imp-no-prime-divisors*:
assumes *is-unit* x *prime-elem* p
shows $\neg p$ *dvd* x
using *unit-imp-no-irreducible-divisors*[*OF assms*(*1*) *prime-elem-imp-irreducible*[*OF assms*(*2*)]] **.**

lemma *prime-elem-mono*:
assumes *prime-elem* p $\neg q$ *dvd* 1 q *dvd* p
shows *prime-elem* q
proof −
from ‹q *dvd* p› **obtain** r **where** r: $p = q * r$ **by** (*elim dvdE*)
hence p *dvd* $q * r$ **by** *simp*
with ‹*prime-elem* p› **have** p *dvd* $q \vee p$ *dvd* r **by** (*rule prime-elem-dvd-multD*)
hence p *dvd* q
proof
assume p *dvd* r
then obtain s **where** s: $r = p * s$ **by** (*elim dvdE*)
from r **have** $p * 1 = p * (q * s)$ **by** (*subst* (*asm*) s) (*simp add*: *mult-ac*)
with ‹*prime-elem* p› **have** q *dvd* 1
by (*subst* (*asm*) *mult-cancel-left*) *auto*
with ‹$\neg q$ *dvd* 1› **show** *?thesis* **by** *contradiction*
qed

show *?thesis*
proof (*rule prime-elemI*)
fix a b **assume** q *dvd* $(a * b)$
with ‹p *dvd* q› **have** p *dvd* $(a * b)$ **by** (*rule dvd-trans*)
with ‹*prime-elem* p› **have** p *dvd* $a \vee p$ *dvd* b **by** (*rule prime-elem-dvd-multD*)
with ‹q *dvd* p› **show** q *dvd* $a \vee q$ *dvd* b **by** (*blast intro*: *dvd-trans*)
qed (*insert assms*, *auto*)
qed

lemma *irreducibleD'*:
assumes *irreducible* a b *dvd* a
shows a *dvd* $b \vee$ *is-unit* b
proof −
from *assms* **obtain** c **where** c: $a = b * c$ **by** (*elim dvdE*)
from *irreducibleD*[*OF assms*(*1*) *this*] **have** *is-unit* $b \vee$ *is-unit* c **.**
thus *?thesis* **by** (*auto simp*: c *mult-unit-dvd-iff*)
qed

lemma *irreducibleI'*:
assumes $a \neq 0$ $\neg$*is-unit a* $\bigwedge b.$ *b dvd a* $\Longrightarrow$ *a dvd b* $\vee$ *is-unit b*
shows *irreducible a*
proof (*rule irreducibleI*)
fix *b c* **assume** *a-eq*: $a = b * c$
hence *a dvd b* $\vee$ *is-unit b* **by** (*intro assms*) *simp-all*
thus *is-unit b* $\vee$ *is-unit c*
proof
assume *a dvd b*
hence $b * c$ *dvd* $b * 1$ **by** (*simp add*: *a-eq*)
moreover from ‹$a \neq 0$› *a-eq* **have** $b \neq 0$ **by** *auto*
ultimately show *?thesis* **by** (*subst* (*asm*) *dvd-times-left-cancel-iff*) *auto*
qed *blast*
qed (*simp-all add*: *assms*(*1*,*2*))

lemma *irreducible-altdef*:
irreducible x $\longleftrightarrow$ $x \neq 0 \wedge \neg$*is-unit x* $\wedge$ $(\forall$ *b. b dvd x* $\longrightarrow$ *x dvd b* $\vee$ *is-unit b*)
using *irreducibleI'*[*of x*] *irreducibleD'*[*of x*] *irreducible-not-unit*[*of x*] **by** *auto*

lemma *prime-elem-multD*:
assumes *prime-elem* $(a * b)$
shows *is-unit a* $\vee$ *is-unit b*
proof $-$
from *assms* **have** $a \neq 0$ $b \neq 0$ **by** (*auto dest*!: *prime-elem-not-zeroI*)
moreover from *assms prime-elem-dvd-multD* [*of* $a * b$] **have** $a * b$ *dvd* $a \vee a *$
b dvd b
by *auto*
ultimately show *?thesis*
using *dvd-times-left-cancel-iff* [*of a b 1*]
dvd-times-right-cancel-iff [*of b a 1*]
by *auto*
qed

lemma *prime-elemD2*:
assumes *prime-elem p* **and** *a dvd p* **and** $\neg$ *is-unit a*
shows *p dvd a*
proof $-$
from ‹*a dvd p*› **obtain** *b* **where** $p = a * b$ **..**
with ‹*prime-elem p*› *prime-elem-multD* ‹$\neg$ *is-unit a*› **have** *is-unit b* **by** *auto*
with ‹$p = a * b$› **show** *?thesis*
by (*auto simp add*: *mult-unit-dvd-iff*)
qed

lemma *prime-elem-dvd-prod-msetE*:
assumes *prime-elem p*
assumes *dvd*: *p dvd prod-mset A*
obtains *a* **where** $a \in\# A$ **and** *p dvd a*
proof $-$

```
  from dvd have ∃ a. a ∈# A ∧ p dvd a
  proof (induct A)
    case empty then show ?case
    using ‹prime-elem p› by (simp add: prime-elem-not-unit)
  next
    case (add a A)
    then have p dvd a * prod-mset A by simp
    with ‹prime-elem p› consider (A) p dvd prod-mset A | (B) p dvd a
      by (blast dest: prime-elem-dvd-multD)
    then show ?case proof cases
      case B then show ?thesis by auto
    next
      case A
      with add.hyps obtain b where b ∈# A p dvd b
        by auto
      then show ?thesis by auto
    qed
  qed
  with that show thesis by blast

qed

context
begin

lemma prime-elem-powerD:
  assumes prime-elem (p ^ n)
  shows   prime-elem p ∧ n = 1
proof (cases n)
  case (Suc m)
  note assms
  also from Suc have p ^ n = p * p^m by simp
  finally have is-unit p ∨ is-unit (p^m) by (rule prime-elem-multD)
 moreover from assms have ¬is-unit p by (simp add: prime-elem-def is-unit-power-iff)
  ultimately have is-unit (p ^ m) by simp
  with ‹¬is-unit p› have m = 0 by (simp add: is-unit-power-iff)
  with Suc assms show ?thesis by simp
qed (insert assms, simp-all)

lemma prime-elem-power-iff:
  prime-elem (p ^ n) ⟷ prime-elem p ∧ n = 1
  by (auto dest: prime-elem-powerD)

end

lemma irreducible-mult-unit-left:
  is-unit a ⟹ irreducible (a * p) ⟷ irreducible p
  by (auto simp: irreducible-altdef mult.commute[of a] is-unit-mult-iff
        mult-unit-dvd-iff dvd-mult-unit-iff)
```

lemma *prime-elem-mult-unit-left*:
is-unit a $\Longrightarrow$ *prime-elem* (*a* ∗ *p*) $\longleftrightarrow$ *prime-elem p*
by (*auto simp*: *prime-elem-def mult.commute*[*of a*] *is-unit-mult-iff mult-unit-dvd-iff*)

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 −
have *p dvd m*∗*n* **using** *dvd-mult-left pk* **by** *blast*
then consider *p dvd m* | *p dvd n*
using *p prime-elem-dvd-mult-iff* **by** *blast*
then show *?thesis*
proof *cases*
case *1* **then obtain** *a* **where** *m* = *p* ∗ *a* **by** (*metis dvd-mult-div-cancel*)
then have ∃ *x*. *k dvd x* ∗ *n* ∧ *m* = *p* ∗ *x*
using *p pk* **by** (*auto simp*: *mult.assoc*)
then show *?thesis* **..**
next
case *2* **then obtain** *b* **where** *n* = *p* ∗ *b* **by** (*metis dvd-mult-div-cancel*)
with *p pk* **have** ∃ *y*. *k dvd m*∗*y* ∧ *n* = *p*∗*y*
by (*metis dvd-mult-right dvd-times-left-cancel-iff mult.left-commute mult-zero-left*)
then show *?thesis* **..**
qed
qed

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*
using *pc*
proof (*induct c arbitrary*: *m n*)
case *0* **show** *?case* **by** *simp*
next
case (*Suc c*)
consider *x* **where** *p*^*c dvd x*∗*n m* = *p*∗*x* | *y* **where** *p*^*c dvd m*∗*y n* = *p*∗*y*
using *prime-elem-dvd-cases* [*of* - *p*^*c*, *OF* - *p*] *Suc.prems* **by** *force*
then show *?case*
proof *cases*
case (*1 x*)
with *Suc.hyps*[*of x n*] **obtain** *a b* **where** *a* + *b* = *c* ∧ *p* ^ *a dvd x* ∧ *p* ^ *b dvd n* **by** *blast*
with *1* **have** *Suc a* + *b* = *Suc c* ∧ *p* ^ *Suc a dvd m* ∧ *p* ^ *b dvd n*
by (*auto intro*: *mult-dvd-mono*)
thus *?thesis* **by** *blast*
next
case (*2 y*)
with *Suc.hyps*[*of m y*] **obtain** *a b* **where** *a* + *b* = *c* ∧ *p* ^ *a dvd m* ∧ *p* ^ *b dvd y* **by** *blast*
with *2* **have** *a* + *Suc b* = *Suc c* ∧ *p* ^ *a dvd m* ∧ *p* ^ *Suc b dvd n*

```
    by (auto intro: mult-dvd-mono)
  with Suc.hyps [of m y] show ∃ a b. a + b = Suc c ∧ p ^ a dvd m ∧ p ^ b dvd
n
    by blast
 qed
qed

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 −
 from assms obtain r s
   where r + s = c ∧ p ^ r dvd m ∧ p ^ s dvd n
   by (blast dest: prime-elem-power-dvd-prod)
 moreover with assms have
   a ≤ r ∨ b ≤ s by arith
 ultimately show ?thesis by (auto intro: power-le-dvd)
qed

lemma prime-elem-not-unit' [simp]:
 ASSUMPTION (prime-elem x) ⟹ ¬is-unit x
 unfolding ASSUMPTION-def by (rule prime-elem-not-unit)

lemma prime-elem-dvd-power-iff:
 assumes prime-elem p
 shows p dvd a ^ n ⟷ p dvd a ∧ n > 0
 using assms by (induct n) (auto dest: prime-elem-not-unit prime-elem-dvd-multD)

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
 using ‹p ^ n dvd a * b› and ‹n > 0›
proof (induct n arbitrary: b)
 case 0 then show ?case by simp
next
 case (Suc n) show ?case
 proof (cases n = 0)
   case True with Suc ‹prime-elem p› ‹¬ p dvd a› show ?thesis
     by (simp add: prime-elem-dvd-mult-iff)
 next
   case False then have n > 0 by simp
   from ‹prime-elem p› have p ≠ 0 by auto
   from Suc.prems have *: p * p ^ n dvd a * b
     by simp
   then have p dvd a * b
     by (rule dvd-mult-left)
   with Suc ‹prime-elem p› ‹¬ p dvd a› have p dvd b
     by (simp add: prime-elem-dvd-mult-iff)
```

```
    moreover define c where c = b div p
    ultimately have b: b = p * c by simp
    with * have p * p ^ n dvd p * (a * c)
      by (simp add: ac-simps)
    with ‹p ≠ 0› have p ^ n dvd a * c
      by simp
    with Suc.hyps ‹n > 0› have p ^ n dvd c
      by blast
    with ‹p ≠ 0› show ?thesis
      by (simp add: b)
  qed
qed

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 −
  from assms have is-unit y ∨ x dvd y by (auto simp: irreducible-altdef)
  thus ?thesis
  proof (elim disjE)
    assume is-unit y
    hence normalize y = 1 by (simp add: is-unit-normalize)
    with assms show ?thesis by simp
  next
    assume x dvd y
    with ‹y dvd x› have normalize y = normalize x by (rule associatedI)
    with assms show ?thesis by simp
  qed
qed

lemma irreducible-normalize-iff [simp]: irreducible (normalize x) = irreducible x
  using irreducible-mult-unit-left[of 1 div unit-factor x x]
  by (cases x = 0) (simp-all add: unit-div-commute)

lemma prime-elem-normalize-iff [simp]: prime-elem (normalize x) = prime-elem
x
  using prime-elem-mult-unit-left[of 1 div unit-factor x x]
  by (cases x = 0) (simp-all add: unit-div-commute)

lemma prime-elem-associated:
  assumes prime-elem p and prime-elem q and q dvd p
  shows normalize q = normalize p
```

using ‹*q dvd p*› **proof** (*rule associatedI*)
  from ‹*prime-elem q*› **have** ¬ *is-unit q*
    by (*auto simp add*: *prime-elem-not-unit*)
  with ‹*prime-elem p*› ‹*q dvd p*› **show** *p dvd q*
    by (*blast intro*: *prime-elemD2*)
qed

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

lemma *not-prime-0* [*simp*]: ¬*prime 0* **by** (*simp add*: *prime-def*)

lemma *not-prime-unit*: *is-unit x* ⟹ ¬*prime x*
  using *prime-elem-not-unit*[*of x*] **by** (*auto simp add*: *prime-def*)

lemma *not-prime-1* [*simp*]: ¬*prime 1* **by** (*simp add*: *not-prime-unit*)

lemma *primeI*: *prime-elem x* ⟹ *normalize x* = *x* ⟹ *prime x*
  by (*simp add*: *prime-def*)

lemma *prime-imp-prime-elem* [*dest*]: *prime p* ⟹ *prime-elem p*
  by (*simp add*: *prime-def*)

lemma *normalize-prime*: *prime p* ⟹ *normalize p* = *p*
  by (*simp add*: *prime-def*)

lemma *prime-normalize-iff* [*simp*]: *prime* (*normalize p*) ⟷ *prime-elem p*
  by (*auto simp add*: *prime-def*)

lemma *prime-power-iff*:
  prime (*p* ^ *n*) ⟷ *prime p* ∧ *n* = *1*
  by (*auto simp*: *prime-def prime-elem-power-iff*)

lemma *prime-imp-nonzero* [*simp*]:
  ASSUMPTION (*prime x*) ⟹ *x* ≠ *0*
  unfolding *ASSUMPTION-def prime-def* **by** *auto*

lemma *prime-imp-not-one* [*simp*]:
  ASSUMPTION (*prime x*) ⟹ *x* ≠ *1*
  unfolding *ASSUMPTION-def* **by** *auto*

lemma *prime-not-unit'* [*simp*]:
  ASSUMPTION (*prime x*) ⟹ ¬*is-unit x*
  unfolding *ASSUMPTION-def prime-def* **by** *auto*

lemma *prime-normalize'* [*simp*]: *ASSUMPTION* (*prime x*) ⟹ *normalize x* = *x*
  unfolding *ASSUMPTION-def prime-def* **by** *simp*

lemma *unit-factor-prime*: *prime x* ⟹ *unit-factor x* = *1*

using *unit-factor-normalize*[*of x*] **unfolding** *prime-def* **by** *auto*

lemma *unit-factor-prime'* [*simp*]: *ASSUMPTION* (*prime x*) ⟹ *unit-factor x* = *1*
unfolding *ASSUMPTION-def* **by** (*rule unit-factor-prime*)

lemma *prime-imp-prime-elem'* [*simp*]: *ASSUMPTION* (*prime x*) ⟹ *prime-elem x*
by (*simp add*: *prime-def ASSUMPTION-def*)

lemma *prime-dvd-multD*: *prime p* ⟹ *p dvd a* ∗ *b* ⟹ *p dvd a* ∨ *p dvd b*
by (*intro prime-elem-dvd-multD*) *simp-all*

lemma *prime-dvd-mult-iff*: *prime p* ⟹ *p dvd a* ∗ *b* ⟷ *p dvd a* ∨ *p dvd b*
by (*auto dest*: *prime-dvd-multD*)

lemma *prime-dvd-power*:
prime p ⟹ *p dvd x* ^ *n* ⟹ *p dvd x*
by (*auto dest*!: *prime-elem-dvd-power simp*: *prime-def*)

lemma *prime-dvd-power-iff*:
prime p ⟹ *n* > *0* ⟹ *p dvd x* ^ *n* ⟷ *p dvd x*
by (*subst prime-elem-dvd-power-iff*) *simp-all*

lemma *prime-dvd-prod-mset-iff*: *prime p* ⟹ *p dvd prod-mset A* ⟷ (∃ *x*. *x* ∈# *A* ∧ *p dvd x*)
by (*induction A*) (*simp-all add*: *prime-elem-dvd-mult-iff prime-imp-prime-elem*, *blast*+)

lemma *prime-dvd-prod-iff*: *finite A* ⟹ *prime p* ⟹ *p dvd prod f A* ⟷ (∃ *x*∈*A*. *p dvd f x*)
by (*auto simp*: *prime-dvd-prod-mset-iff prod-unfold-prod-mset*)

lemma *primes-dvd-imp-eq*:
assumes *prime p prime q p dvd q*
shows *p* = *q*
proof −
from *assms* **have** *irreducible q* **by** (*simp add*: *prime-elem-imp-irreducible prime-def*)
from *irreducibleD'*[*OF this* ‹*p dvd q*›] *assms* **have** *q dvd p* **by** *simp*
with ‹*p dvd q*› **have** *normalize p* = *normalize q* **by** (*rule associatedI*)
with *assms* **show** *p* = *q* **by** *simp*
qed

lemma *prime-dvd-prod-mset-primes-iff*:
assumes *prime p* ⋀*q*. *q* ∈# *A* ⟹ *prime q*
shows *p dvd prod-mset A* ⟷ *p* ∈# *A*
proof −
from *assms*(*1*) **have** *p dvd prod-mset A* ⟷ (∃ *x*. *x* ∈# *A* ∧ *p dvd x*) **by** (*rule prime-dvd-prod-mset-iff*)

also from *assms* **have** $\ldots \longleftrightarrow p \in\# A$ **by** (*auto dest*: *primes-dvd-imp-eq*)
finally show *?thesis* **.**
qed

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 $A \subseteq\# B$
using *assms*
proof (*induction A arbitrary*: *B*)
case *empty*
thus *?case* **by** *simp*
next
case (*add p A B*)
hence *p*: *prime p* **by** *simp*
define B' **where** $B' = B - \{\#p\#\}$
from *add.prems* **have** *p dvd prod-mset B* **by** (*simp add*: *dvd-mult-left*)
with *add.prems* **have** $p \in\# B$
by (*subst* (*asm*) (*2*) *prime-dvd-prod-mset-primes-iff*) *simp-all*
hence *B*: $B = B' + \{\#p\#\}$ **by** (*simp add*: B'*-def*)
from *add.prems p* **have** $A \subseteq\# B'$ **by** (*intro add.IH*) (*simp-all add*: *B*)
thus *?case* **by** (*simp add*: *B*)
qed

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* $\longleftrightarrow A \subseteq\# B$
using *assms* **by** (*auto intro*: *prod-mset-subset-imp-dvd prod-mset-primes-dvd-imp-subset*)

lemma *is-unit-prod-mset-primes-iff*:
assumes $\bigwedge x.\ x \in\# A \Longrightarrow$ *prime x*
shows *is-unit* (*prod-mset A*) $\longleftrightarrow A = \{\#\}$
by (*auto simp add*: *is-unit-prod-mset-iff*)
(*meson all-not-in-conv assms not-prime-unit set-mset-eq-empty-iff*)

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 $-$
from *dvd* **have** *prod-mset A dvd prod-mset* $(B + C)$
by *simp*
with *A B C* **have** *subset*: $A \subseteq\# B + C$
by (*subst* (*asm*) *prod-mset-dvd-prod-mset-primes-iff*) *auto*
define *A1* **and** *A2* **where** $A1 = A \cap\# B$ **and** $A2 = A - A1$
have $A = A1 + A2$ **unfolding** *A1-def A2-def*

by (*rule sym*, *intro subset-mset.add-diff-inverse*) *simp-all*
from *subset* **have** *A1* $\subseteq\#$ *B A2* $\subseteq\#$ *C*
by (*auto simp*: *A1-def A2-def Multiset.subset-eq-diff-conv Multiset.union-commute*)
from ‹*A* = *A1* + *A2*› **have** *prod-mset A* = *prod-mset A1* ∗ *prod-mset A2* **by** *simp*
from *irred* **and** *this* **have** *is-unit* (*prod-mset A1*) ∨ *is-unit* (*prod-mset A2*)
by (*rule irreducibleD*)
with *A* **have** *A1* = {#} ∨ *A2* = {#} **unfolding** *A1-def A2-def*
by (*subst* (*asm*) (*1 2*) *is-unit-prod-mset-primes-iff*) (*auto dest*: *Multiset.in-diffD*)
with *dvd* ‹*A* = *A1* + *A2*› ‹*A1* $\subseteq\#$ *B*› ‹*A2* $\subseteq\#$ *C*› **show** *?thesis*
by (*auto intro*: *prod-mset-subset-imp-dvd*)
qed

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 −
from ‹$A \neq \{\#\}$› **obtain** *x* **where** *x*: $x \in\# A$ **by** *blast*
define *m* **where** *m* = *count B x*
have {*n*. *prod-mset A* ^ *n dvd prod-mset B*} ⊆ {..*m*}
proof *safe*
fix *n* **assume** *dvd*: *prod-mset A* ^ *n dvd prod-mset B*
from *x* **have** *x* ^ *n dvd prod-mset A* ^ *n* **by** (*intro dvd-power-same dvd-prod-mset*)
also note *dvd*
also have *x* ^ *n* = *prod-mset* (*replicate-mset n x*) **by** *simp*
finally have *replicate-mset n x* $\subseteq\#$ *B*
by (*rule prod-mset-primes-dvd-imp-subset*) (*insert A B x*, *simp-all split*: *if-splits*)
thus *n* ≤ *m* **by** (*simp add*: *count-le-replicate-mset-subset-eq m-def*)
qed
moreover have *finite* {..*m*} **by** *simp*
ultimately show *?thesis* **by** (*rule finite-subset*)
qed

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 (*rule prime-elemI*)
fix *a b* **assume** *x dvd a* ∗ *b*

from *dvd-productE*[*OF this*] **obtain** *y z* **where** *yz*: $x = y * z$ *y dvd a z dvd b* **.**
from ‹*irreducible x*› **and** ‹$x = y * z$› **have** *is-unit y* ∨ *is-unit z* **by** (*rule irreducibleD*)
with *yz* **show** *x dvd a* ∨ *x dvd b*
by (*auto simp*: *mult-unit-dvd-iff mult-unit-dvd-iff'*)
qed (*insert assms*, *auto simp*: *irreducible-not-unit*)

lemma *prime-elem-imp-coprime*:
assumes *prime-elem p* ¬*p dvd n*
shows *coprime p n*
proof (*rule coprimeI*)
fix *d* **assume** *d dvd p d dvd n*
show *is-unit d*
proof (*rule ccontr*)
assume ¬*is-unit d*
from ‹*prime-elem p*› **and** ‹*d dvd p*› **and** *this* **have** *p dvd d*
by (*rule prime-elemD2*)
from *this* **and** ‹*d dvd n*› **have** *p dvd n* **by** (*rule dvd-trans*)
with ‹¬*p dvd n*› **show** *False* **by** *contradiction*
qed
qed

lemma *prime-imp-coprime*:
assumes *prime p* ¬*p dvd n*
shows *coprime p n*
using *assms* **by** (*simp add*: *prime-elem-imp-coprime*)

lemma *prime-elem-imp-power-coprime*:
prime-elem p ⟹ ¬ *p dvd a* ⟹ *coprime a* (*p* ^ *m*)
by (*cases m* > *0*) (*auto dest*: *prime-elem-imp-coprime simp add*: *ac-simps*)

lemma *prime-imp-power-coprime*:
prime p ⟹ ¬ *p dvd a* ⟹ *coprime a* (*p* ^ *m*)
by (*rule prime-elem-imp-power-coprime*) *simp-all*

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* ∨ *p*^*n dvd b*
using *assms*
proof −
from *p* **have** ¬ *is-unit p*
by *simp*
with *ab p* **have** ¬ *p dvd a* ∨ ¬ *p dvd b*
using *not-coprimeI* **by** *blast*
with *p* **have** *coprime* (*p* ^ *n*) *a* ∨ *coprime* (*p* ^ *n*) *b*
by (*auto dest*: *prime-elem-imp-power-coprime simp add*: *ac-simps*)
with *pab* **show** *?thesis*
by (*auto simp add*: *coprime-dvd-mult-left-iff coprime-dvd-mult-right-iff*)
qed

lemma *primes-coprime*:
prime p $\Longrightarrow$ *prime q* $\Longrightarrow$ $p \neq q$ $\Longrightarrow$ *coprime p q*
using *prime-imp-coprime primes-dvd-imp-eq* **by** *blast*

end

1.4 Factorial semirings: algebraic structures with unique prime factorizations

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

Alternative characterization

lemma (**in** *normalization-semidom*) *factorial-semiring-altI-aux*:
assumes *finite-divisors*: $\bigwedge x.\ x \neq 0 \Longrightarrow \textit{finite}\ \{y.\ y\ \textit{dvd}\ x \wedge \textit{normalize}\ y = y\}$
assumes *irreducible-imp-prime-elem*: $\bigwedge x.\ \textit{irreducible}\ x \Longrightarrow \textit{prime-elem}\ x$
assumes $x \neq 0$
shows $\exists A.\ (\forall x.\ x \in\# A \longrightarrow \textit{prime-elem}\ x) \wedge \textit{normalize}\ (\textit{prod-mset}\ A) = \textit{normalize}\ x$
using ‹$x \neq 0$›
proof (*induction card* $\{b.\ b\ \textit{dvd}\ x \wedge \textit{normalize}\ b = b\}$ *arbitrary*: *x rule*: *less-induct*)
case (*less a*)
let *?fctrs* = $\lambda a.\ \{b.\ b\ \textit{dvd}\ a \wedge \textit{normalize}\ b = b\}$
show *?case*
proof (*cases is-unit a*)
case *True*
thus *?thesis* **by** (*intro exI*[*of* - {#}]) (*auto simp*: *is-unit-normalize*)
next
case *False*
show *?thesis*
proof (*cases* $\exists b.\ b\ \textit{dvd}\ a \wedge \neg \textit{is-unit}\ b \wedge \neg a\ \textit{dvd}\ b$)
case *False*
with ‹$\neg$*is-unit a*› *less.prems* **have** *irreducible a* **by** (*auto simp*: *irreducible-altdef*)
hence *prime-elem a* **by** (*rule irreducible-imp-prime-elem*)
thus *?thesis* **by** (*intro exI*[*of* - {#*normalize a*#}]) *auto*
next
case *True*
then obtain *b* **where** *b*: *b dvd a* $\neg$ *is-unit b* $\neg$ *a dvd b* **by** *auto*
from *b* **have** *?fctrs b* $\subseteq$ *?fctrs a* **by** (*auto intro*: *dvd-trans*)
moreover from *b* **have** *normalize a* $\notin$ *?fctrs b normalize a* $\in$ *?fctrs a* **by** *simp-all*
hence *?fctrs b* $\neq$ *?fctrs a* **by** *blast*
ultimately have *?fctrs b* $\subset$ *?fctrs a* **by** (*subst subset-not-subset-eq*) *blast*
with *finite-divisors*[*OF* ‹$a \neq 0$›] **have** *card* (*?fctrs b*) $<$ *card* (*?fctrs a*)
by (*rule psubset-card-mono*)
moreover from ‹$a \neq 0$› *b* **have** $b \neq 0$ **by** *auto*

```
    ultimately have ∃A. (∀x. x ∈# A ⟶ prime-elem x) ∧ normalize (prod-mset
A) = normalize b
      by (intro less) auto
    then obtain A where A: (∀x. x ∈# A ⟶ prime-elem x) ∧ normalize (∏#
A) = normalize b
      by auto

    define c where c = a div b
    from b have c: a = b * c by (simp add: c-def)
    from less.prems c have c ≠ 0 by auto
    from b c have ?fctrs c ⊆ ?fctrs a by (auto intro: dvd-trans)
    moreover have normalize a ∉ ?fctrs c
    proof safe
      assume normalize a dvd c
      hence b * c dvd 1 * c by (simp add: c)
      hence b dvd 1 by (subst (asm) dvd-times-right-cancel-iff) fact+
      with b show False by simp
    qed
    with ‹normalize a ∈ ?fctrs a› have ?fctrs a ≠ ?fctrs c by blast
    ultimately have ?fctrs c ⊂ ?fctrs a by (subst subset-not-subset-eq) blast
    with finite-divisors[OF ‹a ≠ 0›] have card (?fctrs c) < card (?fctrs a)
      by (rule psubset-card-mono)
      with ‹c ≠ 0› have ∃A. (∀x. x ∈# A ⟶ prime-elem x) ∧ normalize
(prod-mset A) = normalize c
      by (intro less) auto
    then obtain B where B: (∀x. x ∈# B ⟶ prime-elem x) ∧ normalize (∏#
B) = normalize c
      by auto

    show ?thesis
    proof (rule exI[of - A + B]; safe)
      have normalize (prod-mset (A + B)) =
            normalize (normalize (prod-mset A) * normalize (prod-mset B))
        by simp
      also have ... = normalize (b * c)
        by (simp only: A B) auto
      also have b * c = a
        using c by simp
      finally show normalize (prod-mset (A + B)) = normalize a .
    next
    qed (use A B in auto)
  qed
 qed
qed

lemma factorial-semiring-altI:
  assumes finite-divisors: ⋀x::'a. x ≠ 0 ⟹ finite {y. y dvd x ∧ normalize y =
y}
  assumes irreducible-imp-prime: ⋀x::'a. irreducible x ⟹ prime-elem x
```

shows *OFCLASS*('*a* :: *normalization-semidom*, *factorial-semiring-class*)
by *intro-classes* (*rule factorial-semiring-altI-aux*[*OF assms*])

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 −
from *prime-factorization-exists*[*OF assms*] **obtain** A
where A: $\bigwedge x.\ x \in\# A \Longrightarrow$ *prime-elem x normalize* (*prod-mset A*) = *normalize x* **by** *blast*
define A' **where** $A' =$ *image-mset normalize A*
have *normalize* (*prod-mset A'*) = *normalize* (*prod-mset A*)
by (*simp add*: *A'-def normalize-prod-mset-normalize*)
also note *A*(*2*)
finally have *normalize* (*prod-mset A'*) = *normalize x* **by** *simp*
moreover from *A*(*1*) **have** $\forall x.\ x \in\# A' \longrightarrow$ *prime x* **by** (*auto simp*: *prime-def A'-def*)
ultimately show *?thesis* **by** (*intro that*[*of A'*]) *blast*
qed

lemma *irreducible-imp-prime-elem*:
assumes *irreducible x*
shows *prime-elem x*
proof (*rule prime-elemI*)
fix *a b* **assume** *dvd*: *x dvd a* ∗ *b*
from *assms* **have** $x \neq 0$ **by** *auto*
show *x dvd a* ∨ *x dvd b*
proof (*cases a* = *0* ∨ *b* = *0*)
case *False*
hence $a \neq 0$ $b \neq 0$ **by** *blast*+
note *nz* = ‹*x* ≠ *0*› *this*
from *nz*[*THEN prime-factorization-exists'*] **obtain** *A B C*
where *ABC*:
$\bigwedge z.\ z \in\# A \Longrightarrow$ *prime z*
normalize ($\prod_{\#} A$) = *normalize x*
$\bigwedge z.\ z \in\# B \Longrightarrow$ *prime z*
normalize ($\prod_{\#} B$) = *normalize a*
$\bigwedge z.\ z \in\# C \Longrightarrow$ *prime z*
normalize ($\prod_{\#} C$) = *normalize b*
by *this blast*

have *irreducible* (*prod-mset A*)
by (*subst irreducible-cong*[*OF ABC*(*2*)]) *fact*
moreover have *normalize* (*prod-mset A*) *dvd*

normalize (*normalize* (*prod-mset B*) ∗ *normalize* (*prod-mset C*))
unfolding *ABC* **using** *dvd* **by** *simp*
hence *prod-mset A dvd prod-mset B ∗ prod-mset C*
unfolding *normalize-mult-normalize-left normalize-mult-normalize-right* **by** *simp*
ultimately have *prod-mset A dvd prod-mset B ∨ prod-mset A dvd prod-mset C*
by (*intro prod-mset-primes-irreducible-imp-prime*) (*use ABC* **in** *auto*)
hence *normalize* (*prod-mset A*) *dvd normalize* (*prod-mset B*) ∨
normalize (*prod-mset A*) *dvd normalize* (*prod-mset C*) **by** *simp*
thus *?thesis* **unfolding** *ABC* **by** *simp*
qed *auto*
qed (*use assms* **in** ‹*simp-all add*: *irreducible-def*›)

lemma *finite-divisor-powers*:
assumes *y* ≠ *0* ¬*is-unit x*
shows *finite* {*n*. *x* ^ *n dvd y*}
proof (*cases x* = *0*)
case *True*
with *assms* **have** {*n*. *x* ^ *n dvd y*} = {*0*} **by** (*auto simp*: *power-0-left*)
thus *?thesis* **by** *simp*
next
case *False*
note *nz* = *this* ‹*y* ≠ *0*›
from *nz*[*THEN prime-factorization-exists'*] **obtain** *A B*
where *AB*:
⋀*z*. *z* ∈# *A* ⟹ *prime z*
normalize (∏# *A*) = *normalize x*
⋀*z*. *z* ∈# *B* ⟹ *prime z*
normalize (∏# *B*) = *normalize y*
by *this blast*

from *AB assms* **have** *A* ≠ {#} **by** (*auto simp*: *normalize-1-iff*)
from *AB*(*2*,*4*) *prod-mset-primes-finite-divisor-powers* [*of A B*, *OF AB*(*1*,*3*) *this*]
have *finite* {*n*. *prod-mset A* ^ *n dvd prod-mset B*} **by** *simp*
also have {*n*. *prod-mset A* ^ *n dvd prod-mset B*} =
{*n*. *normalize* (*normalize* (*prod-mset A*) ^ *n*) *dvd normalize* (*prod-mset B*)}
unfolding *normalize-power-normalize* **by** *simp*
also have ... = {*n*. *x* ^ *n dvd y*}
unfolding *AB* **unfolding** *normalize-power-normalize* **by** *simp*
finally show *?thesis* **.**
qed

lemma *finite-prime-divisors*:
assumes *x* ≠ *0*
shows *finite* {*p*. *prime p* ∧ *p dvd x*}
proof −
from *prime-factorization-exists'*[*OF assms*] **obtain** *A*

where A: $\bigwedge z.\ z \in\# A \Longrightarrow$ *prime z normalize* $(\prod_{\#} A) =$ *normalize x* **by** *this blast*
have $\{p.$ *prime* $p \wedge p$ *dvd* $x\} \subseteq$ *set-mset A*
proof *safe*
fix p **assume** p: *prime p* **and** *dvd*: p *dvd x*
from *dvd* **have** p *dvd normalize x* **by** *simp*
also from A **have** *normalize x* = *normalize* (*prod-mset A*) **by** *simp*
finally have p *dvd prod-mset A*
by *simp*
thus $p \in\# A$ **using** p A
by (*subst* (*asm*) *prime-dvd-prod-mset-primes-iff*)
qed
moreover have *finite* (*set-mset A*) **by** *simp*
ultimately show *?thesis* **by** (*rule finite-subset*)
qed

lemma *infinite-unit-divisor-powers*:
assumes $y \neq 0$
assumes *is-unit x*
shows *infinite* $\{n.\ x\hat{}n$ *dvd* $y\}$
proof −
from ‹*is-unit x*› **have** *is-unit* $(x\hat{}n)$ **for** n
using *is-unit-power-iff* **by** *auto*
hence $x\hat{}n$ *dvd* y **for** n
by *auto*
hence $\{n.\ x\hat{}n$ *dvd* $y\} =$ *UNIV*
by *auto*
thus *?thesis*
by *auto*
qed

corollary *is-unit-iff-infinite-divisor-powers*:
assumes $y \neq 0$
shows *is-unit x* $\longleftrightarrow$ *infinite* $\{n.\ x\hat{}n$ *dvd* $y\}$
using *infinite-unit-divisor-powers finite-divisor-powers assms* **by** *auto*

lemma *prime-elem-iff-irreducible*: *prime-elem x* $\longleftrightarrow$ *irreducible x*
by (*blast intro*: *irreducible-imp-prime-elem prime-elem-imp-irreducible*)

lemma *prime-divisor-exists*:
assumes $a \neq 0$ $\neg$*is-unit a*
shows $\exists b.\ b$ *dvd* $a \wedge$ *prime b*
proof −
from *prime-factorization-exists'*[*OF assms*(*1*)]
obtain A **where** A: $\bigwedge z.\ z \in\# A \Longrightarrow$ *prime z normalize* $(\prod_{\#} A) =$ *normalize a*
by *this blast*
with *assms* **have** $A \neq \{\#\}$ **by** *auto*
then obtain x **where** $x \in\# A$ **by** *blast*
with $A(1)$ **have** $*$: x *dvd normalize* (*prod-mset A*) *prime x*

```
    by (auto simp: dvd-prod-mset)
  hence x dvd a by (simp add: A(2))
  with ∗ show ?thesis by blast
qed

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 (cases x = 0)
  case False
  from prime-factorization-exists'[OF this]
  obtain A where A: ⋀z. z ∈# A ⟹ prime z normalize (∏# A) = normalize x
    by this blast
  from A obtain u where u: is-unit u x = u ∗ prod-mset A
    by (elim associatedE2)

  from A(1) have P (u ∗ prod-mset A)
  proof (induction A)
    case (add p A)
    from add.prems have prime p by simp
     moreover from add.prems have P (u ∗ prod-mset A) by (intro add.IH)
simp-all
    ultimately have P (p ∗ (u ∗ prod-mset A)) by (rule assms(3))
    thus ?case by (simp add: mult-ac)
  qed (simp-all add: assms False u)
  with A u show ?thesis by simp
qed (simp-all add: assms(1))

lemma no-prime-divisors-imp-unit:
  assumes a ≠ 0 ⋀b. b dvd a ⟹ normalize b = b ⟹ ¬ prime-elem b
  shows is-unit a
proof (rule ccontr)
  assume ¬is-unit a
  from prime-divisor-exists[OF assms(1) this] obtain b where b dvd a prime b
by auto
  with assms(2)[of b] show False by (simp add: prime-def)
qed

lemma prime-divisorE:
  assumes a ≠ 0 and ¬ is-unit a
  obtains p where prime p and p dvd a
  using assms no-prime-divisors-imp-unit unfolding prime-def by blast

definition multiplicity :: 'a ⇒ 'a ⇒ 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 (cases finite {n. p ^ n dvd x})
  case True
```

hence *multiplicity p x = Max* {*n*. *p* ^ *n dvd x*}
by (*simp add*: *multiplicity-def*)
also have ... ∈ {*n*. *p* ^ *n dvd x*}
by (*rule Max-in*) (*auto intro*!: *True exI*[*of - 0*::*nat*])
finally show *?thesis* **by** *simp*
qed (*simp add*: *multiplicity-def*)

lemma *multiplicity-dvd'*: *n* ≤ *multiplicity p x* ⟹ *p* ^ *n dvd x*
by (*rule dvd-trans*[*OF le-imp-power-dvd multiplicity-dvd*])

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*}
using *finite-divisor-powers*[*OF xp*] **by** (*simp add*: *multiplicity-def*)

lemma *multiplicity-geI*:
assumes *p* ^ *n dvd x*
shows *multiplicity p x* ≥ *n*
proof −
from *assms* **have** *n* ≤ *Max* {*n*. *p* ^ *n dvd x*}
by (*intro Max-ge finite-divisor-powers xp*) *simp-all*
thus *?thesis* **by** (*subst multiplicity-eq-Max*)
qed

lemma *multiplicity-lessI*:
assumes ¬*p* ^ *n dvd x*
shows *multiplicity p x* < *n*
proof (*rule ccontr*)
assume ¬(*n* > *multiplicity p x*)
hence *p* ^ *n dvd x* **by** (*intro multiplicity-dvd'*) *simp*
with *assms* **show** *False* **by** *contradiction*
qed

lemma *power-dvd-iff-le-multiplicity*:
p ^ *n dvd x* ⟷ *n* ≤ *multiplicity p x*
using *multiplicity-geI*[*of n*] *multiplicity-lessI*[*of n*] **by** (*cases p* ^ *n dvd x*) *auto*

lemma *multiplicity-eq-zero-iff*:
shows *multiplicity p x = 0* ⟷ ¬*p dvd x*
using *power-dvd-iff-le-multiplicity*[*of 1*] **by** *auto*

lemma *multiplicity-gt-zero-iff*:
shows *multiplicity p x* > *0* ⟷ *p dvd x*
using *power-dvd-iff-le-multiplicity*[*of 1*] **by** *auto*

lemma *multiplicity-decompose*:

```
  ¬p dvd (x div p ^ multiplicity p x)
proof
  assume *: p dvd x div p ^ multiplicity p x
  have x = x div p ^ multiplicity p x * (p ^ multiplicity p x)
    using multiplicity-dvd[of p x] by simp
  also from * have x div p ^ multiplicity p x = (x div p ^ multiplicity p x div p)
* p by simp
  also have x div p ^ multiplicity p x div p * p * p ^ multiplicity p x =
              x div p ^ multiplicity p x div p * p ^ Suc (multiplicity p x)
    by (simp add: mult-assoc)
  also have p ^ Suc (multiplicity p x) dvd ... by (rule dvd-triv-right)
  finally show False by (subst (asm) power-dvd-iff-le-multiplicity) simp
qed

lemma multiplicity-decompose':
  obtains y where x = p ^ multiplicity p x * y ¬p dvd y
  using that[of x div p ^ multiplicity p x]
  by (simp add: multiplicity-decompose multiplicity-dvd)

end

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

lemma prime-elem-multiplicity-eq-zero-iff:
  prime-elem p ⟹ x ≠ 0 ⟹ multiplicity p x = 0 ⟷ ¬p dvd x
  by (rule multiplicity-eq-zero-iff) simp-all

lemma prime-multiplicity-other:
  assumes prime p prime q p ≠ q
  shows   multiplicity p q = 0
 using assms by (subst prime-elem-multiplicity-eq-zero-iff) (auto dest: primes-dvd-imp-eq)

lemma prime-multiplicity-gt-zero-iff:
  prime-elem p ⟹ x ≠ 0 ⟹ multiplicity p x > 0 ⟷ p dvd x
  by (rule multiplicity-gt-zero-iff) simp-all

lemma multiplicity-unit-left: is-unit p ⟹ multiplicity p x = 0
  by (simp add: multiplicity-def is-unit-power-iff unit-imp-dvd)

lemma multiplicity-unit-right:
  assumes is-unit x
  shows   multiplicity p x = 0
proof (cases is-unit p ∨ x = 0)
  case False
  with multiplicity-lessI[of x p 1] this assms
    show ?thesis by (auto dest: dvd-unit-imp-unit)
qed (auto simp: multiplicity-unit-left)
```

lemma *multiplicity-one* [*simp*]: *multiplicity p 1 = 0*
by (*rule multiplicity-unit-right*) *simp-all*

lemma *multiplicity-eqI*:
assumes *p ^ n dvd x ¬p ^ Suc n dvd x*
shows *multiplicity p x = n*
proof −
consider *x = 0* | *is-unit p* | *x ≠ 0 ¬is-unit p* **by** *blast*
thus *?thesis*
proof *cases*
assume *xp*: *x ≠ 0 ¬is-unit p*
from *xp assms*(*1*) **have** *multiplicity p x ≥ n* **by** (*intro multiplicity-geI*)
moreover from *assms*(*2*) *xp* **have** *multiplicity p x < Suc n* **by** (*intro multiplicity-lessI*)
ultimately show *?thesis* **by** *simp*
next
assume *is-unit p*
hence *is-unit* (*p ^ Suc n*) **by** (*simp add*: *is-unit-power-iff del*: *power-Suc*)
hence *p ^ Suc n dvd x* **by** (*rule unit-imp-dvd*)
with ‹*¬p ^ Suc n dvd x*› **show** *?thesis* **by** *contradiction*
qed (*insert assms*, *simp-all*)
qed

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

lemma *multiplicity-times-same*:
assumes *p ≠ 0*
shows *multiplicity p* (*p ∗ x*) = *Suc* (*multiplicity p x*)
proof (*rule multiplicity-eqI*)
show *p ^ Suc* (*multiplicity p x*) *dvd p ∗ x*
by (*auto intro*!: *mult-dvd-mono multiplicity-dvd*)
from *xp assms* **show** *¬ p ^ Suc* (*Suc* (*multiplicity p x*)) *dvd p ∗ x*
using *power-dvd-iff-le-multiplicity*[*OF xp*, *of Suc* (*multiplicity p x*)] **by** *simp*
qed

end

lemma *multiplicity-same-power'*: *multiplicity p* (*p ^ n*) = (*if p = 0 ∨ is-unit p then 0 else n*)
proof −
consider *p = 0* | *is-unit p* |*p ≠ 0 ¬is-unit p* **by** *blast*
thus *?thesis*
proof *cases*
assume *p ≠ 0 ¬is-unit p*
thus *?thesis* **by** (*induction n*) (*simp-all add*: *multiplicity-times-same*)

qed (*simp-all add*: *power-0-left multiplicity-unit-left*)
qed

lemma *multiplicity-same-power*:
p ≠ *0* ⟹ ¬*is-unit p* ⟹ *multiplicity p* (*p* ^ *n*) = *n*
by (*simp add*: *multiplicity-same-power'*)

lemma *multiplicity-prime-elem-times-other*:
assumes *prime-elem p* ¬*p dvd q*
shows *multiplicity p* (*q* * *x*) = *multiplicity p x*
proof (*cases x* = *0*)
case *False*
show *?thesis*
proof (*rule multiplicity-eqI*)
have *1* * *p* ^ *multiplicity p x dvd q* * *x*
by (*intro mult-dvd-mono multiplicity-dvd*) *simp-all*
thus *p* ^ *multiplicity p x dvd q* * *x* **by** *simp*
next
define *n* **where** *n* = *multiplicity p x*
from *assms* **have** ¬*is-unit p* **by** *simp*
from *multiplicity-decompose'*[*OF False this*]
obtain *y* **where** *y* [*folded n-def*]: *x* = *p* ^ *multiplicity p x* * *y* ¬ *p dvd y* **.**
from *y* **have** *p* ^ *Suc n dvd q* * *x* ⟷ *p* ^ *n* * *p dvd p* ^ *n* * (*q* * *y*) **by** (*simp add*: *mult-ac*)
also from *assms* **have** ... ⟷ *p dvd q* * *y* **by** *simp*
also have ... ⟷ *p dvd q* ∨ *p dvd y* **by** (*rule prime-elem-dvd-mult-iff*) *fact*+
also from *assms y* **have** ... ⟷ *False* **by** *simp*
finally show ¬(*p* ^ *Suc n dvd q* * *x*) **by** *blast*
qed
qed *simp-all*

lemma *multiplicity-self*:
assumes *p* ≠ *0* ¬*is-unit p*
shows *multiplicity p p* = *1*
proof −
from *assms* **have** *multiplicity p p* = *Max* {*n*. *p* ^ *n dvd p*}
by (*simp add*: *multiplicity-eq-Max*)
also from *assms* **have** *p* ^ *n dvd p* ⟷ *n* ≤ *1* **for** *n*
using *dvd-power-iff*[*of p n 1*] **by** *auto*
hence {*n*. *p* ^ *n dvd p*} = {..*1*} **by** *auto*
also have ... = {*0*,*1*} **by** *auto*
finally show *?thesis* **by** *simp*
qed

lemma *multiplicity-times-unit-left*:
assumes *is-unit c*
shows *multiplicity* (*c* * *p*) *x* = *multiplicity p x*
proof −
from *assms* **have** {*n*. (*c* * *p*) ^ *n dvd x*} = {*n*. *p* ^ *n dvd x*}

by (*subst mult.commute*) (*simp add*: *mult-unit-dvd-iff power-mult-distrib is-unit-power-iff*)
thus *?thesis* **by** (*simp add*: *multiplicity-def*)
qed

lemma *multiplicity-times-unit-right*:
assumes *is-unit c*
shows *multiplicity p* (*c* ∗ *x*) = *multiplicity p x*
proof −
from *assms* **have** {*n*. *p* ^ *n dvd c* ∗ *x*} = {*n*. *p* ^ *n dvd x*}
by (*subst mult.commute*) (*simp add*: *dvd-mult-unit-iff*)
thus *?thesis* **by** (*simp add*: *multiplicity-def*)
qed

lemma *multiplicity-normalize-left* [*simp*]:
multiplicity (*normalize p*) *x* = *multiplicity p x*
proof (*cases p* = *0*)
case [*simp*]: *False*
have *normalize p* = (*1 div unit-factor p*) ∗ *p*
by (*simp add*: *unit-div-commute is-unit-unit-factor*)
also have *multiplicity* ... *x* = *multiplicity p x*
by (*rule multiplicity-times-unit-left*) (*simp add*: *is-unit-unit-factor*)
finally show *?thesis* **.**
qed *simp-all*

lemma *multiplicity-normalize-right* [*simp*]:
multiplicity p (*normalize x*) = *multiplicity p x*
proof (*cases x* = *0*)
case [*simp*]: *False*
have *normalize x* = (*1 div unit-factor x*) ∗ *x*
by (*simp add*: *unit-div-commute is-unit-unit-factor*)
also have *multiplicity p* ... = *multiplicity p x*
by (*rule multiplicity-times-unit-right*) (*simp add*: *is-unit-unit-factor*)
finally show *?thesis* **.**
qed *simp-all*

lemma *multiplicity-prime* [*simp*]: *prime-elem p* ⟹ *multiplicity p p* = *1*
by (*rule multiplicity-self*) *auto*

lemma *multiplicity-prime-power* [*simp*]: *prime-elem p* ⟹ *multiplicity p* (*p* ^ *n*)
= *n*
by (*subst multiplicity-same-power′*) *auto*

lift-definition *prime-factorization* :: *′a* ⇒ *′a multiset* **is**
λ*x p*. *if prime p then multiplicity p x else 0*
proof −
fix *x* :: *′a*
show *finite* {*p*. *0* < (*if prime p then multiplicity p x else 0*)} (**is** *finite ?A*)
proof (*cases x* = *0*)
case *False*

```
    from False have ?A ⊆ {p. prime p ∧ p dvd x}
      by (auto simp: multiplicity-gt-zero-iff)
    moreover from False have finite {p. prime p ∧ p dvd x}
      by (rule finite-prime-divisors)
    ultimately show ?thesis by (rule finite-subset)
  qed simp-all
qed

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

lemma count-prime-factorization-nonprime:
  ¬prime p ⟹ count (prime-factorization x) p = 0
  by transfer simp

lemma count-prime-factorization-prime:
  prime p ⟹ count (prime-factorization x) p = multiplicity p x
  by transfer simp

lemma count-prime-factorization:
  count (prime-factorization x) p = (if prime p then multiplicity p x else 0)
  by transfer simp

lemma dvd-imp-multiplicity-le:
  assumes a dvd b b ≠ 0
  shows   multiplicity p a ≤ multiplicity p b
proof (cases is-unit p)
  case False
  with assms show ?thesis
   by (intro multiplicity-geI ) (auto intro: dvd-trans[OF multiplicity-dvd' assms(1)])
qed (insert assms, auto simp: multiplicity-unit-left)

lemma prime-power-inj:
  assumes prime a a ^ m = a ^ n
  shows   m = n
proof −
  have multiplicity a (a ^ m) = multiplicity a (a ^ n) by (simp only: assms)
 thus ?thesis using assms by (subst (asm) (1 2) multiplicity-prime-power) simp-all
qed

lemma prime-power-inj':
  assumes prime p prime q
  assumes p ^ m = q ^ n m > 0 n > 0
  shows   p = q m = n
proof −
  from assms have p ^ 1 dvd p ^ m by (intro le-imp-power-dvd) simp
  also have p ^ m = q ^ n by fact
  finally have p dvd q ^ n by simp
  with assms have p dvd q using prime-dvd-power[of p q] by simp
```

```
  with assms show p = q by (simp add: primes-dvd-imp-eq)
  with assms show m = n by (simp add: prime-power-inj)
qed

lemma prime-power-eq-one-iff [simp]: prime p ⟹ p ^ n = 1 ⟷ n = 0
  using prime-power-inj[of p n 0] by auto

lemma one-eq-prime-power-iff [simp]: prime p ⟹ 1 = p ^ n ⟷ n = 0
  using prime-power-inj[of p 0 n] by auto

lemma prime-power-inj'':
  assumes prime p prime q
  shows   p ^ m = q ^ n ⟷ (m = 0 ∧ n = 0) ∨ (p = q ∧ m = n)
  using assms
  by (cases m = 0; cases n = 0)
     (auto dest: prime-power-inj'[OF assms])

lemma prime-factorization-0 [simp]: prime-factorization 0 = {#}
  by (simp add: multiset-eq-iff count-prime-factorization)

lemma prime-factorization-empty-iff:
  prime-factorization x = {#} ⟷ x = 0 ∨ is-unit x
proof
  assume ∗: prime-factorization x = {#}
  {
    assume x: x ≠ 0 ¬is-unit x
    {
      fix p assume p: prime p
      have count (prime-factorization x) p = 0 by (simp add: ∗)
      also from p have count (prime-factorization x) p = multiplicity p x
        by (rule count-prime-factorization-prime)
       also from x p have ... = 0 ⟷ ¬p dvd x by (simp add: multiplic-
ity-eq-zero-iff)
      finally have ¬p dvd x .
    }
    with prime-divisor-exists[OF x] have False by blast
  }
  thus x = 0 ∨ is-unit x by blast
next
  assume x = 0 ∨ is-unit x
  thus prime-factorization x = {#}
  proof
    assume x: is-unit x
    {
      fix p assume p: prime p
      from p x have multiplicity p x = 0
        by (subst multiplicity-eq-zero-iff)
           (auto simp: multiplicity-eq-zero-iff dest: unit-imp-no-prime-divisors)
    }
```

```
    thus ?thesis by (simp add: multiset-eq-iff count-prime-factorization)
  qed simp-all
qed

lemma prime-factorization-unit:
  assumes is-unit x
  shows   prime-factorization x = {#}
proof (rule multiset-eqI)
  fix p :: 'a
  show count (prime-factorization x) p = count {#} p
  proof (cases prime p)
    case True
    with assms have multiplicity p x = 0
      by (subst multiplicity-eq-zero-iff)
         (auto simp: multiplicity-eq-zero-iff dest: unit-imp-no-prime-divisors)
    with True show ?thesis by (simp add: count-prime-factorization-prime)
  qed (simp-all add: count-prime-factorization-nonprime)
qed

lemma prime-factorization-1 [simp]: prime-factorization 1 = {#}
  by (simp add: prime-factorization-unit)

lemma prime-factorization-times-prime:
  assumes x ≠ 0 prime p
  shows   prime-factorization (p * x) = {#p#} + prime-factorization x
proof (rule multiset-eqI)
  fix q :: 'a
  consider ¬prime q | p = q | prime q p ≠ q by blast
  thus count (prime-factorization (p * x)) q = count ({#p#} + prime-factorization
x) q
  proof cases
    assume q: prime q p ≠ q
    with assms primes-dvd-imp-eq[of q p] have ¬q dvd p by auto
    with q assms show ?thesis
      by (simp add: multiplicity-prime-elem-times-other count-prime-factorization)
  qed (insert assms, auto simp: count-prime-factorization multiplicity-times-same)
qed

lemma prod-mset-prime-factorization-weak:
  assumes x ≠ 0
  shows   normalize (prod-mset (prime-factorization x)) = normalize x
  using assms
proof (induction x rule: prime-divisors-induct)
  case (factor p x)
  have normalize (prod-mset (prime-factorization (p * x))) =
         normalize (p * normalize (prod-mset (prime-factorization x)))
    using factor.prems factor.hyps by (simp add: prime-factorization-times-prime)
  also have normalize (prod-mset (prime-factorization x)) = normalize x
    by (rule factor.IH) (use factor in auto)
```

finally show *?case* **by** *simp*
qed (*auto simp*: *prime-factorization-unit is-unit-normalize*)

lemma *in-prime-factors-iff*:
p ∈ *prime-factors x* ⟷ *x* ≠ *0* ∧ *p dvd x* ∧ *prime p*
proof −
have *p* ∈ *prime-factors x* ⟷ *count* (*prime-factorization x*) *p* > *0* **by** *simp*
also have ... ⟷ *x* ≠ *0* ∧ *p dvd x* ∧ *prime p*
by (*subst count-prime-factorization*, *cases x* = *0*)
(*auto simp*: *multiplicity-eq-zero-iff multiplicity-gt-zero-iff*)
finally show *?thesis* **.**
qed

lemma *in-prime-factors-imp-prime* [*intro*]:
p ∈ *prime-factors x* ⟹ *prime p*
by (*simp add*: *in-prime-factors-iff*)

lemma *in-prime-factors-imp-dvd* [*dest*]:
p ∈ *prime-factors x* ⟹ *p dvd x*
by (*simp add*: *in-prime-factors-iff*)

lemma *prime-factorsI*:
x ≠ *0* ⟹ *prime p* ⟹ *p dvd x* ⟹ *p* ∈ *prime-factors x*
by (*auto simp*: *in-prime-factors-iff*)

lemma *prime-factors-dvd*:
x ≠ *0* ⟹ *prime-factors x* = {*p*. *prime p* ∧ *p dvd x*}
by (*auto intro*: *prime-factorsI*)

lemma *prime-factors-multiplicity*:
prime-factors n = {*p*. *prime p* ∧ *multiplicity p n* > *0*}
by (*cases n* = *0*) (*auto simp add*: *prime-factors-dvd prime-multiplicity-gt-zero-iff*)

lemma *prime-factorization-prime*:
assumes *prime p*
shows *prime-factorization p* = {#*p*#}
proof (*rule multiset-eqI*)
fix *q* :: *'a*
consider ¬*prime q* | *q* = *p* | *prime q q* ≠ *p* **by** *blast*
thus *count* (*prime-factorization p*) *q* = *count* {#*p*#} *q*
by *cases* (*insert assms*, *auto dest*: *primes-dvd-imp-eq*
simp: *count-prime-factorization multiplicity-self multiplicity-eq-zero-iff*)
qed

lemma *prime-factorization-prod-mset-primes*:
assumes ⋀*p*. *p* ∈# *A* ⟹ *prime p*
shows *prime-factorization* (*prod-mset A*) = *A*
using *assms*
proof (*induction A*)

case (*add p A*)
from *add.prems*[*of 0*] **have** *0* $\notin\#$ *A* **by** *auto*
hence *prod-mset A* $\neq$ *0* **by** *auto*
with *add* **show** *?case*
by (*simp-all add*: *mult-ac prime-factorization-times-prime Multiset.union-commute*)
qed *simp-all*

lemma *prime-factorization-cong*:
normalize x = *normalize y* $\Longrightarrow$ *prime-factorization x* = *prime-factorization y*
by (*simp add*: *multiset-eq-iff count-prime-factorization*
multiplicity-normalize-right [*of* - *x*, *symmetric*]
multiplicity-normalize-right [*of* - *y*, *symmetric*]
del: *multiplicity-normalize-right*)

lemma *prime-factorization-unique*:
assumes *x* $\neq$ *0 y* $\neq$ *0*
shows *prime-factorization x* = *prime-factorization y* $\longleftrightarrow$ *normalize x* = *normalize y*
proof
assume *prime-factorization x* = *prime-factorization y*
hence *prod-mset* (*prime-factorization x*) = *prod-mset* (*prime-factorization y*) **by** *simp*
hence *normalize* (*prod-mset* (*prime-factorization x*)) =
normalize (*prod-mset* (*prime-factorization y*))
by (*simp only*:)
with *assms* **show** *normalize x* = *normalize y*
by (*simp add*: *prod-mset-prime-factorization-weak*)
qed (*rule prime-factorization-cong*)

lemma *prime-factorization-normalize* [*simp*]:
prime-factorization (*normalize x*) = *prime-factorization x*
by (*cases x* = *0*, *simp*, *subst prime-factorization-unique*) *auto*

lemma *prime-factorization-eqI-strong*:
assumes $\bigwedge$*p*. *p* $\in\#$ *P* $\Longrightarrow$ *prime p prod-mset P* = *n*
shows *prime-factorization n* = *P*
using *prime-factorization-prod-mset-primes*[*of P*] *assms* **by** *simp*

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 −
have *P* = *prime-factorization* (*normalize* (*prod-mset P*))
using *prime-factorization-prod-mset-primes*[*of P*] *assms*(*1*) **by** *simp*
with *assms*(*2*) **show** *?thesis* **by** *simp*
qed

lemma *prime-factorization-mult*:
assumes *x* $\neq$ *0 y* $\neq$ *0*

shows *prime-factorization* $(x * y) =$ *prime-factorization* $x +$ *prime-factorization* y
proof −
have *normalize* (*prod-mset* (*prime-factorization* x) $*$ *prod-mset* (*prime-factorization* y)) =
normalize (*normalize* (*prod-mset* (*prime-factorization* x)) $*$
normalize (*prod-mset* (*prime-factorization* y)))
by (*simp only*: *normalize-mult-normalize-left normalize-mult-normalize-right*)
also have $\ldots =$ *normalize* $(x * y)$
by (*subst* (*1 2*) *prod-mset-prime-factorization-weak*) (*use assms* **in** *auto*)
finally show *?thesis*
by (*intro prime-factorization-eqI*) *auto*
qed

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* $(f\ n))$
using *assms* **by** (*induction* A *rule*: *finite-induct*)
(*auto simp*: *Sup-multiset-empty prime-factorization-mult*)

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 −
have *multiplicity* p $(x * y) =$ *count* (*prime-factorization* $(x * y)$) (*normalize* p)
by (*subst count-prime-factorization-prime*) (*simp-all add*: *assms*)
also from *assms*
have *prime-factorization* $(x * y) =$ *prime-factorization* $x +$ *prime-factorization* y
by (*intro prime-factorization-mult*)
also have *count* $\ldots$ (*normalize* p) =
count (*prime-factorization* x) (*normalize* p) + *count* (*prime-factorization* y) (*normalize* p)
by *simp*
also have $\ldots =$ *multiplicity* p $x +$ *multiplicity* p y
by (*subst* (*1 2*) *count-prime-factorization-prime*) (*simp-all add*: *assms*)
finally show *?thesis* **.**
qed

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)
using *assms* **by** (*induction* A) (*auto simp*: *prime-elem-multiplicity-mult-distrib*)

lemma *prime-elem-multiplicity-power-distrib*:
assumes *prime-elem* p $x \neq 0$
shows *multiplicity* p $(x \mathbin{\hat{}} n) = n *$ *multiplicity* p x
using *assms prime-elem-multiplicity-prod-mset-distrib* [*of p replicate-mset n x*]

by *simp*

lemma *prime-elem-multiplicity-prod-distrib*:
assumes *prime-elem p 0* $\notin$ *f ' A finite A*
shows *multiplicity p* (*prod f A*) = ($\sum$ *x*$\in$*A*. *multiplicity p* (*f x*))
proof −
have *multiplicity p* (*prod f A*) = ($\sum$ *x*$\in$#*mset-set A*. *multiplicity p* (*f x*))
using *assms* **by** (*subst prod-unfold-prod-mset*)
(*simp-all add*: *prime-elem-multiplicity-prod-mset-distrib sum-unfold-sum-mset*
multiset.map-comp o-def)
also from ‹*finite A*› **have** ... = ($\sum$ *x*$\in$*A*. *multiplicity p* (*f x*))
by (*induction A rule*: *finite-induct*) *simp-all*
finally show *?thesis* **.**
qed

lemma *multiplicity-distinct-prime-power*:
prime p $\Longrightarrow$ *prime q* $\Longrightarrow$ *p* $\neq$ *q* $\Longrightarrow$ *multiplicity p* (*q* ^ *n*) = *0*
by (*subst prime-elem-multiplicity-power-distrib*) (*auto simp*: *prime-multiplicity-other*)

lemma *prime-factorization-prime-power*:
prime p $\Longrightarrow$ *prime-factorization* (*p* ^ *n*) = *replicate-mset n p*
by (*induction n*)
(*simp-all add*: *prime-factorization-mult prime-factorization-prime Multiset.union-commute*)

lemma *prime-factorization-subset-iff-dvd*:
assumes [*simp*]: *x* $\neq$ *0 y* $\neq$ *0*
shows *prime-factorization x* $\subseteq$# *prime-factorization y* $\longleftrightarrow$ *x dvd y*
proof −
have *x dvd y* $\longleftrightarrow$
normalize (*prod-mset* (*prime-factorization x*)) *dvd normalize* (*prod-mset* (*prime-factorization y*))
using *assms* **by** (*subst* (*1 2*) *prod-mset-prime-factorization-weak*) *auto*
also have ... $\longleftrightarrow$ *prime-factorization x* $\subseteq$# *prime-factorization y*
by (*auto intro*!: *prod-mset-primes-dvd-imp-subset prod-mset-subset-imp-dvd*)
finally show *?thesis* **..**
qed

lemma *prime-factorization-subset-imp-dvd*:
x $\neq$ *0* $\Longrightarrow$ (*prime-factorization x* $\subseteq$# *prime-factorization y*) $\Longrightarrow$ *x dvd y*
by (*cases y* = *0*) (*simp-all add*: *prime-factorization-subset-iff-dvd*)

lemma *prime-factorization-divide*:
assumes *b dvd a*
shows *prime-factorization* (*a div b*) = *prime-factorization a* − *prime-factorization b*
proof (*cases a* = *0*)
case [*simp*]: *False*
from *assms* **have** [*simp*]: *b* $\neq$ *0* **by** *auto*
have *prime-factorization* ((*a div b*) ∗ *b*) = *prime-factorization* (*a div b*) +

prime-factorization b
  by (*intro prime-factorization-mult*) (*insert assms*, *auto elim*!: *dvdE*)
 with *assms* **show** *?thesis* **by** *simp*
qed *simp-all*

lemma (**in** *factorial-semiring*) *primepow-divisors-induct* [*case-names zero unit factor*]:
 assumes *P 0* $\bigwedge x.$ *is-unit x* $\Longrightarrow$ *P x*
      $\bigwedge p\ k\ x.$ *prime p* $\Longrightarrow$ *k > 0* $\Longrightarrow$ $\neg$*p dvd x* $\Longrightarrow$ *P x* $\Longrightarrow$ *P* (*p* ^ *k* * *x*)
 shows *P x*
proof −
 have *finite* (*prime-factors x*) **by** *simp*
 thus *?thesis*
 proof (*induction prime-factors x arbitrary*: *x rule*: *finite-induct*)
  case *empty*
  hence *prime-factors x* = {} **by** *metis*
  hence *prime-factorization x* = {#} **by** *simp*
  thus *?case* **using** *assms*(*1*,*2*) **by** (*auto simp*: *prime-factorization-empty-iff*)
 next
  case (*insert p A x*)
  define *k* **where** *k* = *multiplicity p x*
  have *k > 0* **using** *insert.hyps*
   by (*auto simp*: *prime-factors-multiplicity k-def*)
  have *p*: *p* ∈ *prime-factors x* **using** *insert.hyps* **by** *auto*
  from *p* **have** *x* ≠ *0* ¬*is-unit p* **by** (*auto simp*: *in-prime-factors-iff*)

  from *multiplicity-decompose'*[*OF this*] **obtain** *y* **where** *y*: *x* = *p* ^ *k* * *y* ¬*p dvd y*
   by (*auto simp*: *k-def*)
  have *prime-factorization x* = *replicate-mset k p* + *prime-factorization y*
   using *p* ‹*k > 0*› *y* **unfolding** *y*
   by (*subst prime-factorization-mult*)
     (*auto simp*: *prime-factorization-prime-power in-prime-factors-iff*)
  moreover from *y p* **have** *p* ∉ *prime-factors y*
   by (*auto simp*: *in-prime-factors-iff*)
  ultimately have *prime-factors y* = *prime-factors x* − {*p*}
   by *auto*
  also have ... = *A*
   using *insert.hyps* **by** *auto*
  finally have *P y* **using** *insert* **by** *auto*
  thus *P x*
  unfolding *y* **using** *y* ‹*k > 0*› *p* **by** (*intro assms*(*3*)) (*auto simp*: *in-prime-factors-iff*)
 qed
qed

lemma *zero-not-in-prime-factors* [*simp*]: *0* ∉ *prime-factors x*
 by (*auto dest*: *in-prime-factors-imp-prime*)

lemma *prime-prime-factors*:

prime p $\Longrightarrow$ *prime-factors p* = {*p*}
by (*simp add*: *local.prime-factorization-prime*)

lemma *prime-factors-product*:
$x \neq 0 \Longrightarrow y \neq 0 \Longrightarrow$ *prime-factors* $(x * y)$ = *prime-factors x* $\cup$ *prime-factors y*
by (*simp add*: *prime-factorization-mult*)

lemma *dvd-prime-factors* [*intro*]:
$y \neq 0 \Longrightarrow x$ *dvd y* $\Longrightarrow$ *prime-factors x* $\subseteq$ *prime-factors y*
by (*intro set-mset-mono*, *subst prime-factorization-subset-iff-dvd*) *auto*

lemma *multiplicity-le-imp-dvd*:
assumes $x \neq 0$ $\bigwedge p$. *prime p* $\Longrightarrow$ *multiplicity p x* $\leq$ *multiplicity p y*
shows *x dvd y*
proof (*cases y* = *0*)
case *False*
from *assms this* **have** *prime-factorization x* $\subseteq\#$ *prime-factorization y*
by (*intro mset-subset-eqI*) (*auto simp*: *count-prime-factorization*)
with *assms False* **show** *?thesis* **by** (*subst* (*asm*) *prime-factorization-subset-iff-dvd*)
qed *auto*

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*)
by (*auto intro*: *dvd-imp-multiplicity-le multiplicity-le-imp-dvd*)

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*
using *assms* **by** (*intro associatedI multiplicity-le-imp-dvd*) *simp-all*

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 −
have *prime-factorization* $(\prod i \in\# M.\ i)$ = *prime-factorization* $(\prod i \in\# N.\ i)$
by (*simp only*: *assms*)
also from *assms* **have** *prime-factorization* $(\prod i \in\# M.\ i) = M$
by (*subst prime-factorization-prod-mset-primes*) *simp-all*
also from *assms* **have** *prime-factorization* $(\prod i \in\# N.\ i) = N$
by (*subst prime-factorization-prod-mset-primes*) *simp-all*
finally show *?thesis* **.**
qed

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 −
have *prime-factorization* (*normalize* ($\prod i \in\# M.\ i$)) =
prime-factorization (*normalize* ($\prod i \in\# N.\ i$))
by (*simp only*: *assms*)
also from *assms* **have** *prime-factorization* (*normalize* ($\prod i \in\# M.\ i$)) = M
by (*subst prime-factorization-normalize*, *subst prime-factorization-prod-mset-primes*) *simp-all*
also from *assms* **have** *prime-factorization* (*normalize* ($\prod i \in\# N.\ i$)) = N
by (*subst prime-factorization-normalize*, *subst prime-factorization-prod-mset-primes*) *simp-all*
finally show *?thesis* **.**
qed

lemma *multiplicity-cong*:
($\bigwedge r.\ p \mathbin{\hat{}} r\ dvd\ a \longleftrightarrow p \mathbin{\hat{}} r\ dvd\ b$) $\Longrightarrow$ *multiplicity p a* = *multiplicity p b*
by (*simp add*: *multiplicity-def*)

lemma *not-dvd-imp-multiplicity-0*:
assumes $\neg p\ dvd\ x$
shows *multiplicity p x* = *0*
proof −
from *assms* **have** *multiplicity p x* < *1*
by (*intro multiplicity-lessI*) *auto*
thus *?thesis* **by** *simp*
qed

lemma *multiplicity-zero-left* [*simp*]: *multiplicity 0 x* = *0*
by (*cases x* = *0*) (*auto intro*: *not-dvd-imp-multiplicity-0*)

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 (*rule inj-onI*)
fix *P Q* **assume** *PQ*: $P \in A\ Q \in A\ \prod P = \prod Q$
with *prime-factorization-unique'*[*of mset-set P mset-set Q*] *assms*[*of P*] *assms*[*of Q*]
have *mset-set P* = *mset-set Q* **by** (*auto simp*: *prod-unfold-prod-mset*)
with *assms*[*of P*] *assms*[*of Q*] *PQ* **show** *P* = *Q* **by** *simp*
qed

lemma *divides-primepow-weak*:
assumes *prime p* **and** *a dvd p* $\mathbin{\hat{}}$ *n*
obtains *m* **where** $m \leq n$ **and** *normalize a* = *normalize* (*p* $\mathbin{\hat{}}$ *m*)
proof −
from *assms* **have** $a \neq 0$
by *auto*
with *assms*

have *normalize* (*prod-mset* (*prime-factorization a*)) *dvd*
normalize (*prod-mset* (*prime-factorization* (*p* ^ *n*)))
by (*subst* (*1 2*) *prod-mset-prime-factorization-weak*) *auto*
then have *prime-factorization a* ⊆# *prime-factorization* (*p* ^ *n*)
by (*simp add*: *in-prime-factors-imp-prime prod-mset-dvd-prod-mset-primes-iff*)
with *assms* **have** *prime-factorization a* ⊆# *replicate-mset n p*
by (*simp add*: *prime-factorization-prime-power*)
then obtain *m* **where** *m* ≤ *n* **and** *prime-factorization a* = *replicate-mset m p*
by (*rule msubseteq-replicate-msetE*)
then have ∗: *normalize* (*prod-mset* (*prime-factorization a*)) =
normalize (*prod-mset* (*replicate-mset m p*)) **by** *metis*
also have *normalize* (*prod-mset* (*prime-factorization a*)) = *normalize a*
using ‹*a* ≠ *0*› **by** (*simp add*: *prod-mset-prime-factorization-weak*)
also have *prod-mset* (*replicate-mset m p*) = *p* ^ *m*
by *simp*
finally show *?thesis* **using** ‹*m* ≤ *n*›
by (*intro that*[*of m*])
qed

lemma *divide-out-primepow-ex*:
assumes *n* ≠ *0* ∃*p*∈*prime-factors n*. *P p*
obtains *p k n*′ **where** *P p prime p p dvd n* ¬*p dvd n*′ *k* > *0 n* = *p* ^ *k* ∗ *n*′
proof −
from *assms* **obtain** *p* **where** *p*: *P p prime p p dvd n*
by *auto*
define *k* **where** *k* = *multiplicity p n*
define *n*′ **where** *n*′ = *n div p* ^ *k*
have *n*′: *n* = *p* ^ *k* ∗ *n*′ ¬*p dvd n*′
using *assms p multiplicity-decompose*[*of n p*]
by (*auto simp*: *n*′*-def k-def multiplicity-dvd*)
from *n*′ *p* **have** *k* > *0* **by** (*intro Nat.gr0I*) *auto*
with *n*′ *p that*[*of p n*′ *k*] **show** *?thesis* **by** *auto*
qed

lemma *divide-out-primepow*:
assumes *n* ≠ *0* ¬*is-unit n*
obtains *p k n*′ **where** *prime p p dvd n* ¬*p dvd n*′ *k* > *0 n* = *p* ^ *k* ∗ *n*′
using *divide-out-primepow-ex*[*OF assms*(*1*), *of* λ*-. True*] *prime-divisor-exists*[*OF assms*] *assms*
prime-factorsI **by** *metis*

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* ≠ *0 b* ≠ *0*
shows *prime-factorization* (*gcd-factorial a b*) = *prime-factorization a* ∩# *prime-factorization b*
proof −
have *prime-factorization* (*gcd-factorial a b*) =
prime-factorization (*prod-mset* (*prime-factorization a* ∩# *prime-factorization b*))
by (*simp add*: *gcd-factorial-def*)
also have ... = *prime-factorization a* ∩# *prime-factorization b*
by (*subst prime-factorization-prod-mset-primes*) *auto*
finally show *?thesis* **.**
qed

lemma *prime-factorization-lcm-factorial*:
assumes [*simp*]: *a* ≠ *0 b* ≠ *0*
shows *prime-factorization* (*lcm-factorial a b*) = *prime-factorization a* ∪# *prime-factorization b*
proof −
have *prime-factorization* (*lcm-factorial a b*) =
prime-factorization (*prod-mset* (*prime-factorization a* ∪# *prime-factorization b*))
by (*simp add*: *lcm-factorial-def*)
also have ... = *prime-factorization a* ∪# *prime-factorization b*
by (*subst prime-factorization-prod-mset-primes*) *auto*
finally show *?thesis* **.**
qed

lemma *prime-factorization-Gcd-factorial*:
assumes ¬*A* ⊆ {*0*}
shows *prime-factorization* (*Gcd-factorial A*) = *Inf* (*prime-factorization* ‘ (*A* − {*0*}))
proof −
from *assms* **obtain** *x* **where** *x*: *x* ∈ *A* − {*0*} **by** *auto*
hence *Inf* (*prime-factorization* ‘ (*A* − {*0*})) ⊆# *prime-factorization x*
by (*intro subset-mset.cInf-lower*) *simp-all*
hence ∀ *y*. *y* ∈# *Inf* (*prime-factorization* ‘ (*A* − {*0*})) ⟶ *y* ∈ *prime-factors x*

by (*auto dest*: *mset-subset-eqD*)
with *in-prime-factors-imp-prime*[*of* - *x*]
have $\forall p.\ p \in\# Inf\ (prime\text{-}factorization\ `\ (A - \{0\})) \longrightarrow prime\ p$ **by** *blast*
with *assms* **show** *?thesis*
by (*simp add*: *Gcd-factorial-def prime-factorization-prod-mset-primes*)
qed

lemma *prime-factorization-Lcm-factorial*:
assumes $0 \notin A$ *subset-mset.bdd-above* (*prime-factorization* ` *A*)
shows *prime-factorization* (*Lcm-factorial A*) = *Sup* (*prime-factorization* ` *A*)
proof (*cases* $A = \{\}$)
case *True*
hence *prime-factorization* ` $A = \{\}$ **by** *auto*
also have *Sup* ... = $\{\#\}$ **by** (*simp add*: *Sup-multiset-empty*)
finally show *?thesis* **by** (*simp add*: *Lcm-factorial-def*)
next
case *False*
have $\forall y.\ y \in\# Sup\ (prime\text{-}factorization\ `\ A) \longrightarrow prime\ y$
by (*auto simp*: *in-Sup-multiset-iff assms*)
with *assms False* **show** *?thesis*
by (*simp add*: *Lcm-factorial-def prime-factorization-prod-mset-primes*)
qed

lemma *gcd-factorial-commute*: *gcd-factorial a b* = *gcd-factorial b a*
by (*simp add*: *gcd-factorial-def multiset-inter-commute*)

lemma *gcd-factorial-dvd1*: *gcd-factorial a b dvd a*
proof (*cases* $a = 0 \vee b = 0$)
case *False*
hence *gcd-factorial a b* $\neq 0$ **by** (*auto simp*: *gcd-factorial-def*)
with *False* **show** *?thesis*
by (*subst prime-factorization-subset-iff-dvd* [*symmetric*])
(*auto simp*: *prime-factorization-gcd-factorial*)
qed (*auto simp*: *gcd-factorial-def*)

lemma *gcd-factorial-dvd2*: *gcd-factorial a b dvd b*
by (*subst gcd-factorial-commute*) (*rule gcd-factorial-dvd1*)

lemma *normalize-gcd-factorial* [*simp*]: *normalize* (*gcd-factorial a b*) = *gcd-factorial a b*
by (*simp add*: *gcd-factorial-def*)

lemma *normalize-lcm-factorial* [*simp*]: *normalize* (*lcm-factorial a b*) = *lcm-factorial a b*
by (*simp add*: *lcm-factorial-def*)

lemma *gcd-factorial-greatest*: *c dvd gcd-factorial a b* **if** *c dvd a c dvd b* **for** *a b c*
proof (*cases* $a = 0 \vee b = 0$)
case *False*

```
  with that have [simp]: c ≠ 0 by auto
  let ?p = prime-factorization
  from that False have ?p c ⊆# ?p a ?p c ⊆# ?p b
    by (simp-all add: prime-factorization-subset-iff-dvd)
  hence prime-factorization c ⊆#
        prime-factorization (prod-mset (prime-factorization a ∩# prime-factorization
b))
    using False by (subst prime-factorization-prod-mset-primes) auto
  with False show ?thesis
    by (auto simp: gcd-factorial-def prime-factorization-subset-iff-dvd [symmetric])
qed (auto simp: gcd-factorial-def that)

lemma lcm-factorial-gcd-factorial:
  lcm-factorial a b = normalize (a * b div gcd-factorial a b) for a b
proof (cases a = 0 ∨ b = 0)
  case False
  let ?p = prime-factorization
  have 1: normalize x * normalize y dvd z ⟷ x * y dvd z for x y z :: 'a
  proof -
    have normalize (normalize x * normalize y) dvd z ⟷ x * y dvd z
      unfolding normalize-mult-normalize-left normalize-mult-normalize-right by
simp
    thus ?thesis unfolding normalize-dvd-iff by simp
  qed

  have ?p (a * b) = (?p a ∪# ?p b) + (?p a ∩# ?p b)
    using False by (subst prime-factorization-mult) (auto intro!: multiset-eqI)
  hence normalize (prod-mset (?p (a * b))) =
          normalize (prod-mset ((?p a ∪# ?p b) + (?p a ∩# ?p b)))
    by (simp only:)
  hence *: normalize (a * b) = normalize (lcm-factorial a b * gcd-factorial a b)
using False
    by (subst (asm) prod-mset-prime-factorization-weak)
      (auto simp: lcm-factorial-def gcd-factorial-def)

  have [simp]: gcd-factorial a b dvd a * b lcm-factorial a b dvd a * b
    using associatedD2[OF *] by auto
  from False have [simp]: gcd-factorial a b ≠ 0 lcm-factorial a b ≠ 0
    by (auto simp: gcd-factorial-def lcm-factorial-def)

  show ?thesis
    by (rule associated-eqI)
       (use * in ‹auto simp: dvd-div-iff-mult div-dvd-iff-mult dest: associatedD1
associatedD2›)
qed (auto simp: lcm-factorial-def)

lemma normalize-Gcd-factorial:
  normalize (Gcd-factorial A) = Gcd-factorial A
  by (simp add: Gcd-factorial-def)
```

lemma *Gcd-factorial-eq-0-iff*:
Gcd-factorial A = 0 ⟷ A ⊆ {0}
by (*auto simp*: *Gcd-factorial-def in-Inf-multiset-iff split*: *if-splits*)

lemma *Gcd-factorial-dvd*:
assumes *x ∈ A*
shows *Gcd-factorial A dvd x*
proof (*cases x = 0*)
case *False*
with *assms* **have** *prime-factorization* (*Gcd-factorial A*) = *Inf* (*prime-factorization* ‘ (*A* − {*0*}))
by (*intro prime-factorization-Gcd-factorial*) *auto*
also from *False assms* **have** ... ⊆# *prime-factorization x*
by (*intro subset-mset.cInf-lower*) *auto*
finally show *?thesis*
by (*subst* (*asm*) *prime-factorization-subset-iff-dvd*)
(*insert assms False*, *auto simp*: *Gcd-factorial-eq-0-iff*)
qed *simp-all*

lemma *Gcd-factorial-greatest*:
assumes ⋀*y*. *y ∈ A* ⟹ *x dvd y*
shows *x dvd Gcd-factorial A*
proof (*cases A ⊆ {0}*)
case *False*
from *False* **obtain** *y* **where** *y ∈ A y ≠ 0* **by** *auto*
with *assms*[*of y*] **have** *nz*: *x ≠ 0* **by** *auto*
from *nz assms* **have** *prime-factorization x* ⊆# *prime-factorization y* **if** *y ∈ A* − {*0*} **for** *y*
using *that* **by** (*subst prime-factorization-subset-iff-dvd*) *auto*
with *False* **have** *prime-factorization x* ⊆# *Inf* (*prime-factorization* ‘ (*A* − {*0*}))
by (*intro subset-mset.cInf-greatest*) *auto*
also from *False* **have** ... = *prime-factorization* (*Gcd-factorial A*)
by (*rule prime-factorization-Gcd-factorial* [*symmetric*])
finally show *?thesis*
by (*subst* (*asm*) *prime-factorization-subset-iff-dvd*)
(*insert nz False*, *auto simp*: *Gcd-factorial-eq-0-iff*)
qed (*simp-all add*: *Gcd-factorial-def*)

lemma *normalize-Lcm-factorial*:
normalize (*Lcm-factorial A*) = *Lcm-factorial A*
by (*simp add*: *Lcm-factorial-def*)

lemma *Lcm-factorial-eq-0-iff*:
Lcm-factorial A = 0 ⟷ 0 ∈ A ∨ ¬*subset-mset.bdd-above* (*prime-factorization* ‘ *A*)
by (*auto simp*: *Lcm-factorial-def in-Sup-multiset-iff*)

lemma *dvd-Lcm-factorial*:

assumes $x \in A$
shows *x dvd Lcm-factorial A*
proof (*cases* $0 \notin A \wedge$ *subset-mset.bdd-above* (*prime-factorization* ‘ *A*))
case *True*
with *assms* **have** [*simp*]: $0 \notin A$ $x \neq 0$ $A \neq \{\}$ **by** *auto*
from *assms True* **have** *prime-factorization x* $\subseteq\#$ *Sup* (*prime-factorization* ‘ *A*)
by (*intro subset-mset.cSup-upper*) *auto*
also have ... = *prime-factorization* (*Lcm-factorial A*)
by (*rule prime-factorization-Lcm-factorial* [*symmetric*]) (*insert True*, *simp-all*)
finally show *?thesis*
by (*subst* (*asm*) *prime-factorization-subset-iff-dvd*)
(*insert True*, *auto simp*: *Lcm-factorial-eq-0-iff*)
qed (*insert assms*, *auto simp*: *Lcm-factorial-def*)

lemma *Lcm-factorial-least*:
assumes $\bigwedge y.\ y \in A \Longrightarrow$ *y dvd x*
shows *Lcm-factorial A dvd x*
proof −
consider $A = \{\} \mid 0 \in A \mid x = 0 \mid A \neq \{\}$ $0 \notin A$ $x \neq 0$ **by** *blast*
thus *?thesis*
proof *cases*
assume $*$: $A \neq \{\}$ $0 \notin A$ $x \neq 0$
hence *nz*: $x \neq 0$ **if** $x \in A$ **for** *x* **using** *that* **by** *auto*
from $*$ **have** *bdd*: *subset-mset.bdd-above* (*prime-factorization* ‘ *A*)
by (*intro subset-mset.bdd-aboveI*[*of - prime-factorization x*])
(*auto simp*: *prime-factorization-subset-iff-dvd nz dest*: *assms*)
have *prime-factorization* (*Lcm-factorial A*) = *Sup* (*prime-factorization* ‘ *A*)
by (*rule prime-factorization-Lcm-factorial*) *fact+*
also from $*$ **have** ... $\subseteq\#$ *prime-factorization x*
by (*intro subset-mset.cSup-least*)
(*auto simp*: *prime-factorization-subset-iff-dvd nz dest*: *assms*)
finally show *?thesis*
by (*subst* (*asm*) *prime-factorization-subset-iff-dvd*)
(*insert* $*$ *bdd*, *auto simp*: *Lcm-factorial-eq-0-iff*)
qed (*auto simp*: *Lcm-factorial-def dest*: *assms*)
qed

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 ≠ 0 b ≠ 0*
shows *prime-factorization (gcd a b) = prime-factorization a ∩# prime-factorization b*
by (*simp add*: *gcd-eq-gcd-factorial prime-factorization-gcd-factorial*)

lemma *prime-factorization-lcm*:
assumes [*simp*]: *a ≠ 0 b ≠ 0*
shows *prime-factorization (lcm a b) = prime-factorization a ∪# prime-factorization b*
by (*simp add*: *lcm-eq-lcm-factorial prime-factorization-lcm-factorial*)

lemma *prime-factorization-Gcd*:
assumes *Gcd A ≠ 0*
shows *prime-factorization (Gcd A) = Inf (prime-factorization ‘ (A − {0}))*
using *assms*
by (*simp add*: *prime-factorization-Gcd-factorial Gcd-eq-Gcd-factorial Gcd-factorial-eq-0-iff*)

lemma *prime-factorization-Lcm*:
assumes *Lcm A ≠ 0*
shows *prime-factorization (Lcm A) = Sup (prime-factorization ‘ A)*
using *assms*
by (*simp add*: *prime-factorization-Lcm-factorial Lcm-eq-Lcm-factorial Lcm-factorial-eq-0-iff*)

lemma *prime-factors-gcd* [*simp*]:
a ≠ 0 ⟹ b ≠ 0 ⟹ prime-factors (gcd a b) =
prime-factors a ∩ prime-factors b
by (*subst prime-factorization-gcd*) *auto*

lemma *prime-factors-lcm* [*simp*]:
a ≠ 0 ⟹ b ≠ 0 ⟹ prime-factors (lcm a b) =
prime-factors a ∪ prime-factors b
by (*subst prime-factorization-lcm*) *auto*

subclass *semiring-gcd*
by (*standard*, *unfold gcd-eq-gcd-factorial lcm-eq-lcm-factorial*)
(*rule gcd-lcm-factorial*; *assumption*)+

subclass *semiring-Gcd*
by (*standard*, *unfold Gcd-eq-Gcd-factorial Lcm-eq-Lcm-factorial*)
(*rule gcd-lcm-factorial*; *assumption*)+

lemma
assumes *x ≠ 0 y ≠ 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 −
  have gcd x y = gcd-factorial x y by (rule gcd-eq-gcd-factorial)
  also have ... = ?rhs1
    by (auto simp: gcd-factorial-def assms prod-mset-multiplicity
        count-prime-factorization-prime
         intro!: arg-cong[of - - normalize] dest: in-prime-factors-imp-prime intro!:
prod.cong)
  finally show gcd x y = ?rhs1 .
  have lcm x y = lcm-factorial x y by (rule lcm-eq-lcm-factorial)
  also have ... = ?rhs2
    by (auto simp: lcm-factorial-def assms prod-mset-multiplicity
        count-prime-factorization-prime intro!: arg-cong[of - - normalize]
        dest: in-prime-factors-imp-prime intro!: prod.cong)
  finally show lcm x y = ?rhs2 .
qed

lemma
  assumes x ≠ 0 y ≠ 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 −
  have gcd x y = gcd-factorial x y by (rule gcd-eq-gcd-factorial)
  also from assms have multiplicity p ... = min (multiplicity p x) (multiplicity p
y)
   by (simp add: count-prime-factorization-prime [symmetric] prime-factorization-gcd-factorial)
  finally show multiplicity p (gcd x y) = min (multiplicity p x) (multiplicity p y) .
  have lcm x y = lcm-factorial x y by (rule lcm-eq-lcm-factorial)
  also from assms have multiplicity p ... = max (multiplicity p x) (multiplicity
p y)
   by (simp add: count-prime-factorization-prime [symmetric] prime-factorization-lcm-factorial)
  finally show multiplicity p (lcm x y) = max (multiplicity p x) (multiplicity p y)
.
qed

lemma gcd-lcm-distrib:
  gcd x (lcm y z) = lcm (gcd x y) (gcd x z)
proof (cases x = 0 ∨ y = 0 ∨ z = 0)
  case True
  thus ?thesis
    by (auto simp: lcm-proj1-if-dvd lcm-proj2-if-dvd)
next
  case False
  hence normalize (gcd x (lcm y z)) = normalize (lcm (gcd x y) (gcd x z))
```

```
    by (intro associatedI prime-factorization-subset-imp-dvd)
      (auto simp: lcm-eq-0-iff prime-factorization-gcd prime-factorization-lcm
        subset-mset.inf-sup-distrib1)
  thus ?thesis by simp
qed

lemma lcm-gcd-distrib:
  lcm x (gcd y z) = gcd (lcm x y) (lcm x z)
proof (cases x = 0 ∨ y = 0 ∨ z = 0)
  case True
  thus ?thesis
    by (auto simp: lcm-proj1-if-dvd lcm-proj2-if-dvd)
next
  case False
  hence normalize (lcm x (gcd y z)) = normalize (gcd (lcm x y) (lcm x z))
    by (intro associatedI prime-factorization-subset-imp-dvd)
      (auto simp: lcm-eq-0-iff prime-factorization-gcd prime-factorization-lcm
        subset-mset.sup-inf-distrib1)
  thus ?thesis by simp
qed

end

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

subclass ring-gcd ..

subclass idom-divide ..

end


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

lemma normalize-prod-mset-primes:
  (⋀p. p ∈# A ⟹ prime p) ⟹ normalize (prod-mset A) = prod-mset A
proof (induction A)
  case (add p A)
  hence prime p by simp
  hence normalize p = p by simp
  with add show ?case by (simp add: normalize-mult)
qed simp-all

lemma prod-mset-prime-factorization:
  assumes x ≠ 0
  shows   prod-mset (prime-factorization x) = normalize x
```

using *assms*
by (*induction x rule*: *prime-divisors-induct*)
(*simp-all add*: *prime-factorization-unit prime-factorization-times-prime*
is-unit-normalize normalize-mult)

lemma *prime-decomposition*: *unit-factor x* $*$ *prod-mset* (*prime-factorization x*) = *x*
by (*cases x* = *0*) (*simp-all add*: *prod-mset-prime-factorization*)

lemma *prod-prime-factors*:
assumes $x \neq 0$
shows $(\prod p \in \text{prime-factors } x.\ p \mathbin{\hat{}} \text{multiplicity } p\ x) = \text{normalize } x$
proof −
have *normalize x* = *prod-mset* (*prime-factorization x*)
by (*simp add*: *prod-mset-prime-factorization assms*)
also have ... = $(\prod p \in \text{prime-factors } x.\ p \mathbin{\hat{}} \text{count } (\text{prime-factorization } x)\ p)$
by (*subst prod-mset-multiplicity*) *simp-all*
also have ... = $(\prod p \in \text{prime-factors } x.\ p \mathbin{\hat{}} \text{multiplicity } p\ x)$
by (*intro prod.cong*)
(*simp-all add*: *assms count-prime-factorization-prime in-prime-factors-imp-prime*)
finally show *?thesis* **..**
qed

lemma *prime-factorization-unique''*:
assumes *S-eq*: $S = \{p.\ 0 < f\ p\}$
and *finite S*
and *S*: $\forall p{\in}S.\ \text{prime } p$ *normalize n* = $(\prod p{\in}S.\ p \mathbin{\hat{}} f\ p)$
shows $S = \text{prime-factors } n \wedge (\forall p.\ \text{prime } p \longrightarrow f\ p = \text{multiplicity } p\ n)$
proof
define *A* **where** *A* = *Abs-multiset f*
from ‹*finite S*› *S*(*1*) **have** $(\prod p{\in}S.\ p \mathbin{\hat{}} f\ p) \neq 0$ **by** *auto*
with *S*(*2*) **have** *nz*: $n \neq 0$ **by** *auto*
from *S-eq* ‹*finite S*› **have** *count-A*: *count A* = *f*
unfolding *A-def* **by** (*subst multiset.Abs-multiset-inverse*) *simp-all*
from *S-eq count-A* **have** *set-mset-A*: *set-mset A* = *S*
by (*simp only*: *set-mset-def*)
from *S*(*2*) **have** *normalize n* = $(\prod p{\in}S.\ p \mathbin{\hat{}} f\ p)$ **.**
also have ... = *prod-mset A* **by** (*simp add*: *prod-mset-multiplicity S-eq set-mset-A count-A*)
also from *nz* **have** *normalize n* = *prod-mset* (*prime-factorization n*)
by (*simp add*: *prod-mset-prime-factorization*)
finally have *prime-factorization* (*prod-mset A*) =
prime-factorization (*prod-mset* (*prime-factorization n*)) **by** *simp*
also from *S*(*1*) **have** *prime-factorization* (*prod-mset A*) = *A*
by (*intro prime-factorization-prod-mset-primes*) (*auto simp*: *set-mset-A*)
also have *prime-factorization* (*prod-mset* (*prime-factorization n*)) = *prime-factorization n*
by (*intro prime-factorization-prod-mset-primes*) *auto*
finally show *S* = *prime-factors n* **by** (*simp add*: *set-mset-A* [*symmetric*])

```
  show (∀ p. prime p ⟶ f p = multiplicity p n)
  proof safe
    fix p :: 'a assume p: prime p
    have multiplicity p n = multiplicity p (normalize n) by simp
    also have normalize n = prod-mset A
      by (simp add: prod-mset-multiplicity S-eq set-mset-A count-A S)
    also from p set-mset-A S(1)
    have multiplicity p ... = sum-mset (image-mset (multiplicity p) A)
      by (intro prime-elem-multiplicity-prod-mset-distrib) auto
    also from S(1) p
    have image-mset (multiplicity p) A = image-mset (λq. if p = q then 1 else 0)
A
    by (intro image-mset-cong) (auto simp: set-mset-A multiplicity-self prime-multiplicity-other)
    also have sum-mset ... = f p
      by (simp add: semiring-1-class.sum-mset-delta' count-A)
    finally show f p = multiplicity p n ..
  qed
qed

lemma divides-primepow:
  assumes prime p and a dvd p ^ n
  obtains m where m ≤ n and normalize a = p ^ m
  using divides-primepow-weak[OF assms] that assms
  by (auto simp add: normalize-power)

lemma Ex-other-prime-factor:
  assumes n ≠ 0 and ¬(∃ k. normalize n = p ^ k) prime p
  shows  ∃ q∈prime-factors n. q ≠ p
proof (rule ccontr)
  assume ∗: ¬(∃ q∈prime-factors n. q ≠ p)
  have normalize n = (∏ p∈prime-factors n. p ^ multiplicity p n)
    using assms(1) by (intro prod-prime-factors [symmetric]) auto
  also from ∗ have ... = (∏ p∈{p}. p ^ multiplicity p n)
   using assms(3) by (intro prod.mono-neutral-left) (auto simp: prime-factors-multiplicity)
  finally have normalize n = p ^ multiplicity p n by auto
  with assms show False by auto
qed
```

Now a string of results due to Maya Kdzioka

```
lemma multiplicity-dvd-iff-dvd:
 assumes x ≠ 0
 shows p^k dvd x ⟷ p^k dvd p^multiplicity p x
proof (cases is-unit p)
 case True
 then have is-unit (p^k)
   using is-unit-power-iff by simp
 hence p^k dvd x
   by auto
```

```
 moreover from ‹is-unit p› have p^k dvd p^multiplicity p x
   using multiplicity-unit-left is-unit-power-iff by simp
 ultimately show ?thesis by simp
next
 case False
 show ?thesis
 proof (cases p = 0)
   case True
   then have p^multiplicity p x = 1
     by simp
   moreover have p^k dvd x ⟹ k = 0
   proof (rule ccontr)
     assume p^k dvd x and k ≠ 0
     with ‹p = 0› have p^k = 0 by auto
     with ‹p^k dvd x› have 0 dvd x by auto
     hence x = 0 by auto
     with ‹x ≠ 0› show False by auto
   qed
   ultimately show ?thesis
     by (auto simp add: is-unit-power-iff ‹¬ is-unit p›)
 next
   case False
   with ‹x ≠ 0› ‹¬ is-unit p› show ?thesis
    by (simp add: power-dvd-iff-le-multiplicity dvd-power-iff multiplicity-same-power)
 qed
qed

lemma multiplicity-decomposeI:
 assumes x = p^k * x' and ¬ p dvd x' and p ≠ 0
 shows multiplicity p x = k
  using assms local.multiplicity-eqI local.power-Suc2 by force

lemma multiplicity-sum-lt:
 assumes multiplicity p a < multiplicity p b a ≠ 0 b ≠ 0
 shows multiplicity p (a + b) = multiplicity p a
proof −
 let ?vp = multiplicity p
 have unit: ¬ is-unit p
 proof
   assume is-unit p
   then have ?vp a = 0 and ?vp b = 0 using multiplicity-unit-left by auto
   with assms show False by auto
 qed

 from multiplicity-decompose' obtain a' where a': a = p^?vp a * a' ¬ p dvd a'
   using unit assms by metis
 from multiplicity-decompose' obtain b' where b': b = p^?vp b * b'
   using unit assms by metis
```

show *?vp* $(a + b) =$ *?vp a*
proof (*rule multiplicity-decomposeI*)
let *?k* $=$ *?vp b* $-$ *?vp a*
from *assms* **have** *k*: *?k* > 0 **by** *simp*
with b' **have** $b = p\hat{}$*?vp a* $*$ $p\hat{}$*?k* $*$ b'
by (*simp flip*: *power-add*)
with a' **show** $*$: $a + b = p\hat{}$*?vp a* $*$ $(a' + p\hat{}$*?k* $*$ $b')$
by (*simp add*: *ac-simps distrib-left*)
moreover show $\neg$ *p dvd* $a' + p\hat{}$*?k* $*$ b'
using a' *k dvd-add-left-iff* **by** *auto*
show $p \neq 0$ **using** *assms* **by** *auto*
qed
qed

corollary *multiplicity-sum-min*:
assumes *multiplicity p a* $\neq$ *multiplicity p b a* $\neq 0$ *b* $\neq 0$
shows *multiplicity p* $(a + b) =$ *min* (*multiplicity p a*) (*multiplicity p b*)
proof $-$
let *?vp* $=$ *multiplicity p*
from *assms* **have** *?vp a* $<$ *?vp b* $\vee$ *?vp a* $>$ *?vp b*
by *auto*
then show *?thesis*
by (*metis assms multiplicity-sum-lt min.commute add-commute min.strict-order-iff*)

qed

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 (*cases a* $= 0$)
case *True*
then show *?thesis*

```
    by simp
next
  case [simp]: False
  have euclidean-size (normalize a) ≤ euclidean-size (normalize a * unit-factor a)
    by (rule size-mult-mono) simp
  moreover have euclidean-size a ≤ euclidean-size (a * (1 div unit-factor a))
    by (rule size-mult-mono) simp
  ultimately show ?thesis
    by simp
qed

context
begin

qualified function gcd :: 'a ⇒ 'a ⇒ 'a
  where gcd a b = (if b = 0 then normalize a else gcd b (a mod b))
  by pat-completeness simp
termination
  by (relation measure (euclidean-size ∘ snd)) (simp-all add: mod-size-less)

declare gcd.simps [simp del]

lemma eucl-induct [case-names zero mod]:
  assumes H1: ⋀b. P b 0
  and H2: ⋀a b. b ≠ 0 ⟹ P b (a mod b) ⟹ P a b
  shows P a b
proof (induct a b rule: gcd.induct)
  case (1 a b)
  show ?case
  proof (cases b = 0)
    case True then show P a b by simp (rule H1)
  next
    case False
    then have P b (a mod b)
      by (rule 1.hyps)
    with ‹b ≠ 0› show P a b
      by (blast intro: H2)
  qed
qed

qualified lemma gcd-0:
  gcd a 0 = normalize a
  by (simp add: gcd.simps [of a 0])

qualified lemma gcd-mod:
  a ≠ 0 ⟹ gcd a (b mod a) = gcd b a
  by (simp add: gcd.simps [of b 0] gcd.simps [of b a])

qualified 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 drop*]: *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 drop*]: *Gcd A = Lcm {d. ∀ a∈A. d dvd a}*

lemma *semiring-gcd*:
class.semiring-gcd one zero times gcd lcm
divide plus minus unit-factor normalize
proof
show *gcd a b dvd a*
and *gcd a b dvd b* **for** *a b*
by (*induct a b rule*: *eucl-induct*)
(*simp-all add*: *local.gcd-0 local.gcd-mod dvd-mod-iff*)
next
show *c dvd a ⟹ c dvd b ⟹ c dvd gcd a b* **for** *a b c*
proof (*induct a b rule*: *eucl-induct*)
case (*zero a*) **from** ‹*c dvd a*› **show** *?case*
by (*rule dvd-trans*) (*simp add*: *local.gcd-0*)
next
case (*mod a b*)
then show *?case*
by (*simp add*: *local.gcd-mod dvd-mod-iff*)
qed
next
show *normalize (gcd a b) = gcd a b* **for** *a b*
by (*induct a b rule*: *eucl-induct*)
(*simp-all add*: *local.gcd-0 local.gcd-mod*)
next
show *lcm a b = normalize (a * b div gcd a b)* **for** *a b*
by (*fact local.lcm-def*)
qed

interpretation *semiring-gcd one zero times gcd lcm*
divide plus minus unit-factor normalize
by (*fact semiring-gcd*)

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

```
  show ?thesis
  proof
    have (∀ a∈A. a dvd Lcm A) ∧ (∀ b. (∀ a∈A. a dvd b) ⟶ Lcm A dvd b) for A
    proof (cases ∃ l. l ≠  0 ∧ (∀ a∈A. a dvd l))
      case False
      then have Lcm A = 0
        by (auto simp add: local.Lcm-def)
      with False show ?thesis
        by auto
    next
      case True
      then obtain l₀ where l₀-props: l₀ ≠ 0 ∀ a∈A. a dvd l₀ by blast
     define n where n = (LEAST n. ∃ l. l ≠ 0 ∧ (∀ a∈A. a dvd l) ∧ euclidean-size
l = n)
      define l where l = (SOME l. l ≠ 0 ∧ (∀ a∈A. a dvd l) ∧ euclidean-size l =
n)
      have ∃ l. l ≠ 0 ∧ (∀ a∈A. a dvd l) ∧ euclidean-size l = n
        apply (subst n-def)
        apply (rule LeastI [of - euclidean-size l₀])
        apply (rule exI [of - l₀])
        apply (simp add: l₀-props)
        done
      from someI-ex [OF this] have l ≠ 0 and ∀ a∈A. a dvd l
        and euclidean-size l = n
        unfolding l-def by simp-all
      {
        fix l' assume ∀ a∈A. a dvd l'
        with ‹∀ a∈A. a dvd l› have ∀ a∈A. a dvd gcd l l'
          by (auto intro: gcd-greatest)
        moreover from ‹l ≠ 0› have gcd l l' ≠ 0
          by simp
        ultimately have ∃ b. b ≠ 0 ∧ (∀ a∈A. a dvd b) ∧
          euclidean-size b = euclidean-size (gcd l l')
          by (intro exI [of - gcd l l'], auto)
        then have euclidean-size (gcd l l') ≥ n
          by (subst n-def) (rule Least-le)
        moreover have euclidean-size (gcd l l') ≤ n
        proof −
          have gcd l l' dvd l
            by simp
          then obtain a where l = gcd l l' * a ..
          with ‹l ≠ 0› have a ≠ 0
            by auto
          hence euclidean-size (gcd l l') ≤ euclidean-size (gcd l l' * a)
            by (rule size-mult-mono)
          also have gcd l l' * a = l using ‹l = gcd l l' * a› ..
          also note ‹euclidean-size l = n›
          finally show euclidean-size (gcd l l') ≤ n .
        qed
```

```
      ultimately have ∗: euclidean-size l = euclidean-size (gcd l l′)
        by (intro le-antisym, simp-all add: ‹euclidean-size l = n›)
      from ‹l ≠ 0› have l dvd gcd l l′
        by (rule dvd-euclidean-size-eq-imp-dvd) (auto simp add: ∗)
      hence l dvd l′ by (rule dvd-trans [OF - gcd-dvd2])
    }
    with ‹∀ a∈A. a dvd l› and ‹l ≠ 0›
      have (∀ a∈A. a dvd normalize l) ∧
        (∀ l′. (∀ a∈A. a dvd l′) ⟶ normalize l dvd l′)
      by auto
    also from True have normalize l = Lcm A
      by (simp add: local.Lcm-def Let-def n-def l-def)
    finally show ?thesis .
  qed
  then show dvd-Lcm: a ∈ A ⟹ a dvd Lcm A
    and Lcm-least: (⋀a. a ∈ A ⟹ a dvd b) ⟹ Lcm A dvd b for A and a b
    by auto
  show a ∈ A ⟹ Gcd A dvd a for A and a
    by (auto simp add: local.Gcd-def intro: Lcm-least)
  show (⋀a. a ∈ A ⟹ b dvd a) ⟹ b dvd Gcd A for A and b
    by (auto simp add: local.Gcd-def intro: dvd-Lcm)
  show [simp]: normalize (Lcm A) = Lcm A for A
    by (simp add: local.Lcm-def)
  show normalize (Gcd A) = Gcd A for A
    by (simp add: local.Gcd-def)
 qed
qed

end

interpretation semiring-Gcd one zero times
  Euclidean-Algorithm.gcd Euclidean-Algorithm.lcm Euclidean-Algorithm.Gcd Euclidean-Algorithm.Lcm
  divide plus minus unit-factor normalize
 by (fact semiring-Gcd)

subclass factorial-semiring
proof −
 show class.factorial-semiring divide plus minus zero times one
   unit-factor normalize
 proof (standard, rule factorial-semiring-altI-aux) — FIXME rule
  fix x assume x ≠ 0
  thus finite {p. p dvd x ∧ normalize p = p}
  proof (induction euclidean-size x arbitrary: x rule: less-induct)
   case (less x)
   show ?case
   proof (cases ∃ y. y dvd x ∧ ¬x dvd y ∧ ¬is-unit y)
    case False
    have {p. p dvd x ∧ normalize p = p} ⊆ {1, normalize x}
```

```
      proof
        fix p assume p: p ∈ {p. p dvd x ∧ normalize p = p}
        with False have is-unit p ∨ x dvd p by blast
        thus p ∈ {1, normalize x}
        proof (elim disjE)
          assume is-unit p
          hence normalize p = 1 by (simp add: is-unit-normalize)
          with p show ?thesis by simp
        next
          assume x dvd p
          with p have normalize p = normalize x by (intro associatedI) simp-all
          with p show ?thesis by simp
        qed
      qed
      moreover have finite ... by simp
      ultimately show ?thesis by (rule finite-subset)
    next
      case True
      then obtain y where y: y dvd x ¬x dvd y ¬is-unit y by blast
      define z where z = x div y
      let ?fctrs = λx. {p. p dvd x ∧ normalize p = p}
      from y have x: x = y * z by (simp add: z-def)
      with less.prems have y ≠ 0 z ≠ 0 by auto
      have normalized-factors-product:
        {p. p dvd a * b ∧ normalize p = p} ⊆
          (λ(x,y). normalize (x * y)) ' ({p. p dvd a ∧ normalize p = p} × {p. p
dvd b ∧ normalize p = p})
        for a b
      proof safe
        fix p assume p: p dvd a * b normalize p = p
        from p(1) obtain x y where xy: p = x * y x dvd a y dvd b
          by (rule dvd-productE)
        define x' y' where x' = normalize x and y' = normalize y
        have p = normalize (x' * y')
          using p by (simp add: xy x'-def y'-def)
        moreover have x' dvd a ∧ normalize x' = x' and y' dvd b ∧ normalize
y' = y'
          using xy by (auto simp: x'-def y'-def)
        ultimately show p ∈ (λ(x, y). normalize (x * y)) '
            ({p. p dvd a ∧ normalize p = p} × {p. p dvd b ∧ normalize p = p})
by fast
      qed
      from x y have ¬is-unit z by (auto simp: mult-unit-dvd-iff)
      have ?fctrs x ⊆ (λ(p,p'). normalize (p * p')) ' (?fctrs y × ?fctrs z)
        by (subst x) (rule normalized-factors-product)
      moreover have ¬y * z dvd y * 1 ¬y * z dvd 1 * z
        by (subst dvd-times-left-cancel-iff dvd-times-right-cancel-iff; fact)+
      hence finite ((λ(p,p'). normalize (p * p')) ' (?fctrs y × ?fctrs z))
      by (intro finite-imageI finite-cartesian-product less dvd-proper-imp-size-less)
```

```
          (auto simp: x)
      ultimately show ?thesis by (rule finite-subset)
    qed
  qed
 next
  fix p
  assume irreducible p
  then show prime-elem p
   by (rule irreducible-imp-prime-elem-gcd)
 qed
qed

lemma Gcd-eucl-set [code]:
 Euclidean-Algorithm.Gcd (set xs) = fold Euclidean-Algorithm.gcd xs 0
 by (fact Gcd-set-eq-fold)

lemma Lcm-eucl-set [code]:
 Euclidean-Algorithm.Lcm (set xs) = fold Euclidean-Algorithm.lcm xs 1
 by (fact Lcm-set-eq-fold)

end

lemma prime-elem-int-abs-iff [simp]:
 fixes p :: int
 shows prime-elem |p| ⟷ prime-elem p
 using prime-elem-normalize-iff [of p] by simp

lemma prime-elem-int-minus-iff [simp]:
 fixes p :: int
 shows prime-elem (− p) ⟷ prime-elem p
 using prime-elem-normalize-iff [of − p] by simp

lemma prime-int-iff:
 fixes p :: int
 shows prime p ⟷ p > 0 ∧ prime-elem p
 by (auto simp add: prime-def dest: prime-elem-not-zeroI)
```

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
 unfolding gcd-eucl [symmetric] lcm-eucl [symmetric]
 by (fact semiring-gcd)
```

subclass *semiring-Gcd*
  unfolding *gcd-eucl* [*symmetric*] *lcm-eucl* [*symmetric*]
    Gcd-eucl [*symmetric*] *Lcm-eucl* [*symmetric*]
  by (*fact semiring-Gcd*)

subclass *factorial-semiring-gcd*
proof
  show *gcd a b* = *gcd-factorial a b* **for** *a b*
    apply (*rule sym*)
    apply (*rule gcdI*)
      apply (*fact gcd-lcm-factorial*)+
    done
  then show *lcm a b* = *lcm-factorial a b* **for** *a b*
    by (*simp add*: *lcm-factorial-gcd-factorial lcm-gcd*)
  show *Gcd A* = *Gcd-factorial A* **for** *A*
    apply (*rule sym*)
    apply (*rule GcdI*)
      apply (*fact gcd-lcm-factorial*)+
    done
  show *Lcm A* = *Lcm-factorial A* **for** *A*
    apply (*rule sym*)
    apply (*rule LcmI*)
      apply (*fact gcd-lcm-factorial*)+
    done
qed

lemma *gcd-mod-right* [*simp*]:
  a ≠ *0* ⟹ *gcd a* (*b mod a*) = *gcd a b*
  unfolding *gcd.commute* [*of a b*]
  by (*simp add*: *gcd-eucl* [*symmetric*] *local.gcd-mod*)

lemma *gcd-mod-left* [*simp*]:
  b ≠ *0* ⟹ *gcd* (*a mod b*) *b* = *gcd a b*
  by (*drule gcd-mod-right* [*of* - *a*]) (*simp add*: *gcd.commute*)

lemma *euclidean-size-gcd-le1* [*simp*]:
  assumes *a* ≠ *0*
  shows *euclidean-size* (*gcd a b*) ≤ *euclidean-size a*
proof −
  from *gcd-dvd1* **obtain** *c* **where** *A*: *a* = *gcd a b* ∗ *c* **..**
  with *assms* **have** *c* ≠ *0*
    by *auto*
  moreover from *this*
  have *euclidean-size* (*gcd a b*) ≤ *euclidean-size* (*gcd a b* ∗ *c*)
    by (*rule size-mult-mono*)
  with *A* **show** *?thesis*
    by *simp*
qed

lemma *euclidean-size-gcd-le2* [*simp*]:
  $b \neq 0 \Longrightarrow$ *euclidean-size* (*gcd a b*) $\leq$ *euclidean-size b*
  by (*subst gcd.commute*, *rule euclidean-size-gcd-le1*)

lemma *euclidean-size-gcd-less1*:
  assumes $a \neq 0$ **and** $\neg$ *a dvd b*
  shows *euclidean-size* (*gcd a b*) $<$ *euclidean-size a*
proof (*rule ccontr*)
  assume $\neg$*euclidean-size* (*gcd a b*) $<$ *euclidean-size a*
  with ‹$a \neq 0$› **have** *A*: *euclidean-size* (*gcd a b*) $=$ *euclidean-size a*
    by (*intro le-antisym*, *simp-all*)
  have *a dvd gcd a b*
    by (*rule dvd-euclidean-size-eq-imp-dvd*) (*simp-all add*: *assms A*)
  hence *a dvd b* **using** *dvd-gcdD2* **by** *blast*
  with ‹$\neg$ *a dvd b*› **show** *False* **by** *contradiction*
qed

lemma *euclidean-size-gcd-less2*:
  assumes $b \neq 0$ **and** $\neg$ *b dvd a*
  shows *euclidean-size* (*gcd a b*) $<$ *euclidean-size b*
  using *assms* **by** (*subst gcd.commute*, *rule euclidean-size-gcd-less1*)

lemma *euclidean-size-lcm-le1*:
  assumes $a \neq 0$ **and** $b \neq 0$
  shows *euclidean-size a* $\leq$ *euclidean-size* (*lcm a b*)
proof −
  have *a dvd lcm a b* **by** (*rule dvd-lcm1*)
  then obtain *c* **where** *A*: *lcm a b* $= a * c$ **..**
  with ‹$a \neq 0$› **and** ‹$b \neq 0$› **have** $c \neq 0$ **by** (*auto simp*: *lcm-eq-0-iff*)
  then show *?thesis* **by** (*subst A*, *intro size-mult-mono*)
qed

lemma *euclidean-size-lcm-le2*:
  $a \neq 0 \Longrightarrow b \neq 0 \Longrightarrow$ *euclidean-size b* $\leq$ *euclidean-size* (*lcm a b*)
  using *euclidean-size-lcm-le1* [*of b a*] **by** (*simp add*: *ac-simps*)

lemma *euclidean-size-lcm-less1*:
  assumes $b \neq 0$ **and** $\neg$ *b dvd a*
  shows *euclidean-size a* $<$ *euclidean-size* (*lcm a b*)
proof (*rule ccontr*)
  from *assms* **have** $a \neq 0$ **by** *auto*
  assume $\neg$*euclidean-size a* $<$ *euclidean-size* (*lcm a b*)
  with ‹$a \neq 0$› **and** ‹$b \neq 0$› **have** *euclidean-size* (*lcm a b*) $=$ *euclidean-size a*
    by (*intro le-antisym*, *simp*, *intro euclidean-size-lcm-le1*)
  with *assms* **have** *lcm a b dvd a*
    by (*rule-tac dvd-euclidean-size-eq-imp-dvd*) (*auto simp*: *lcm-eq-0-iff*)
  hence *b dvd a* **by** (*rule lcm-dvdD2*)
  with ‹$\neg$*b dvd a*› **show** *False* **by** *contradiction*

qed

lemma *euclidean-size-lcm-less2*:
assumes $a \neq 0$ **and** $\neg$ *a dvd b*
shows *euclidean-size b* $<$ *euclidean-size* (*lcm a b*)
using *assms euclidean-size-lcm-less1* [*of a b*] **by** (*simp add*: *ac-simps*)

end

lemma *factorial-euclidean-semiring-gcdI*:
OFCLASS('*a*::{*factorial-semiring-gcd*, *normalization-euclidean-semiring*}, *euclidean-semiring-gcd-class*)
proof
interpret *semiring-Gcd 1 0 times*
Euclidean-Algorithm.gcd Euclidean-Algorithm.lcm
Euclidean-Algorithm.Gcd Euclidean-Algorithm.Lcm
divide plus minus unit-factor normalize
rewrites *dvd.dvd* (∗) = *Rings.dvd*
by (*fact semiring-Gcd*) (*simp add*: *dvd.dvd-def dvd-def fun-eq-iff*)
show [*simp*]: *Euclidean-Algorithm.gcd* = (*gcd* :: '*a* ⇒ -)
proof (*rule ext*)+
fix *a b* :: '*a*
show *Euclidean-Algorithm.gcd a b* = *gcd a b*
proof (*induct a b rule*: *eucl-induct*)
case *zero*
then show *?case*
by *simp*
next
case (*mod a b*)
moreover have *gcd b* (*a mod b*) = *gcd b a*
using *GCD.gcd-add-mult* [*of b a div b a mod b*, *symmetric*]
by (*simp add*: *div-mult-mod-eq*)
ultimately show *?case*
by (*simp add*: *Euclidean-Algorithm.gcd-mod ac-simps*)
qed
qed
show [*simp*]: *Euclidean-Algorithm.Lcm* = (*Lcm* :: '*a set* ⇒ -)
by (*auto intro*!: *Lcm-eqI GCD.dvd-Lcm GCD.Lcm-least*)
show *Euclidean-Algorithm.lcm* = (*lcm* :: '*a* ⇒ -)
by (*simp add*: *fun-eq-iff Euclidean-Algorithm.lcm-def semiring-gcd-class.lcm-gcd*)
show *Euclidean-Algorithm.Gcd* = (*Gcd* :: '*a set* ⇒ -)
by (*simp add*: *fun-eq-iff Euclidean-Algorithm.Gcd-def semiring-Gcd-class.Gcd-Lcm*)
qed

2.3 The extended euclidean algorithm

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

subclass *euclidean-ring* **..**

subclass *ring-gcd* **..**
subclass *factorial-ring-gcd* **..**

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))*
  by *auto*
termination
  by (*relation measure* (λ(-, -, -, -, -, *b*). *euclidean-size b*))
    (*simp-all add*: *mod-size-less*)

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 −
  have *case euclid-ext-aux s' s t' t r' r of ((x, y), c) ⇒*
    *x * a + y * b = c ∧ c = gcd a b* (**is** *?P* (*euclid-ext-aux s' s t' t r' r*))
    using *assms*(*1−3*)
  proof (*induction s' s t' t r' r rule*: *euclid-ext-aux.induct*)
    case (*1 s' s t' t r' r*)
    show *?case*
    proof (*cases r = 0*)
      case *True*
      hence *euclid-ext-aux s' s t' t r' r =*
          ((s' div unit-factor r', t' div unit-factor r'), normalize r')
        by (*subst euclid-ext-aux.simps*) (*simp add*: *Let-def*)
      also have *?P* ...
      proof *safe*
        have *s' div unit-factor r' * a + t' div unit-factor r' * b =*
            *(s' * a + t' * b) div unit-factor r'*
          by (*cases r' = 0*) (*simp-all add*: *unit-div-commute*)
        also have *s' * a + t' * b = r'* **by** *fact*
        also have ... *div unit-factor r' = normalize r'* **by** *simp*
       finally show *s' div unit-factor r' * a + t' div unit-factor r' * b = normalize r'* **.**
      next
        from *1.prems True* **show** *normalize r' = gcd a b*
          by *simp*
      qed
      finally show *?thesis* **.**

```
    next
      case False
      hence euclid-ext-aux s' s t' t r' r =
            euclid-ext-aux s (s' − r' div r ∗ s) t (t' − r' div r ∗ t) r (r' mod r)
        by (subst euclid-ext-aux.simps) (simp add: Let-def)
      also from 1.prems False have ?P ...
      proof (intro 1.IH)
        have (s' − r' div r ∗ s) ∗ a + (t' − r' div r ∗ t) ∗ b =
            (s' ∗ a + t' ∗ b) − r' div r ∗ (s ∗ a + t ∗ b) by (simp add: algebra-simps)
        also have s' ∗ a + t' ∗ b = r' by fact
        also have s ∗ a + t ∗ b = r by fact
        also have r' − r' div r ∗ r = r' mod r using div-mult-mod-eq [of r' r]
          by (simp add: algebra-simps)
        finally show (s' − r' div r ∗ s) ∗ a + (t' − r' div r ∗ t) ∗ b = r' mod r .
      qed (auto simp: algebra-simps minus-mod-eq-div-mult [symmetric] gcd.commute)
      finally show ?thesis .
    qed
  qed
  with assms(4) show c = gcd a b x ∗ a + y ∗ b = gcd a b
    by simp-all
qed

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

definition bezout-coefficients :: 'a ⇒ 'a ⇒ 'a × '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)
  by (simp add: bezout-coefficients-def euclid-ext-aux.simps)

lemma bezout-coefficients-left-0:
  bezout-coefficients 0 a = (0, 1 div unit-factor a)
  by (simp add: bezout-coefficients-def euclid-ext-aux.simps)

lemma bezout-coefficients:
  assumes bezout-coefficients a b = (x, y)
  shows x ∗ a + y ∗ b = gcd a b
  using assms by (simp add: bezout-coefficients-def
    euclid-ext-aux-bezout [of a b a b 1 0 0 1 x y] prod-eq-iff)

lemma bezout-coefficients-fst-snd:
  fst (bezout-coefficients a b) ∗ a + snd (bezout-coefficients a b) ∗ b = gcd a b
  by (rule bezout-coefficients) simp

lemma euclid-ext-eq [simp]:
  euclid-ext a b = (bezout-coefficients a b, gcd a b) (is ?p = ?q)
proof
  show fst ?p = fst ?q
```

```
    by (simp add: bezout-coefficients-def)
  have snd (euclid-ext-aux 1 0 0 1 a b) = gcd a b
    by (rule euclid-ext-aux-eq-gcd [of a b a b 1 0 0 1])
      (simp-all add: prod-eq-iff)
  then show snd ?p = snd ?q
    by simp
qed

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

end

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

subclass factorial-semiring-multiplicative ..

end

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

subclass normalization-euclidean-semiring-multiplicative ..

subclass normalization-euclidean-semiring ..

subclass semiring-gcd-mult-normalize ..

end
```

2.4 Typical instances

```
instance nat :: normalization-euclidean-semiring ..

instance nat :: euclidean-semiring-gcd
proof
  interpret semiring-Gcd 1 0 times
    Euclidean-Algorithm.gcd Euclidean-Algorithm.lcm
    Euclidean-Algorithm.Gcd Euclidean-Algorithm.Lcm
    divide plus minus unit-factor normalize
    rewrites dvd.dvd (*) = Rings.dvd
    by (fact semiring-Gcd) (simp add: dvd.dvd-def dvd-def fun-eq-iff)
  show [simp]: (Euclidean-Algorithm.gcd :: nat ⇒ -) = gcd
  proof (rule ext)+
    fix m n :: nat
    show Euclidean-Algorithm.gcd m n = gcd m n
    proof (induct m n rule: eucl-induct)
```

```
    case zero
    then show ?case
      by simp
  next
    case (mod m n)
    then have gcd n (m mod n) = gcd n m
      using gcd-nat.simps [of m n] by (simp add: ac-simps)
    with mod show ?case
      by (simp add: Euclidean-Algorithm.gcd-mod ac-simps)
  qed
 qed
 show [simp]: (Euclidean-Algorithm.Lcm :: nat set ⇒ -) = Lcm
  by (auto intro!: ext Lcm-eqI)
 show (Euclidean-Algorithm.lcm :: nat ⇒ -) = lcm
  by (simp add: fun-eq-iff Euclidean-Algorithm.lcm-def semiring-gcd-class.lcm-gcd)
 show (Euclidean-Algorithm.Gcd :: nat set ⇒ -) = Gcd
  by (simp add: fun-eq-iff Euclidean-Algorithm.Gcd-def semiring-Gcd-class.Gcd-Lcm)
qed

instance nat :: normalization-euclidean-semiring-multiplicative ..

lemma prime-factorization-Suc-0 [simp]: prime-factorization (Suc 0) = {#}
 unfolding One-nat-def [symmetric] using prime-factorization-1 .

instance int :: normalization-euclidean-semiring ..

instance int :: euclidean-ring-gcd
proof
 interpret semiring-Gcd 1 0 times
  Euclidean-Algorithm.gcd Euclidean-Algorithm.lcm
  Euclidean-Algorithm.Gcd Euclidean-Algorithm.Lcm
  divide plus minus unit-factor normalize
  rewrites dvd.dvd (*) = Rings.dvd
  by (fact semiring-Gcd) (simp add: dvd.dvd-def dvd-def fun-eq-iff)
 show [simp]: (Euclidean-Algorithm.gcd :: int ⇒ -) = gcd
 proof (rule ext)+
  fix k l :: int
  show Euclidean-Algorithm.gcd k l = gcd k l
  proof (induct k l rule: eucl-induct)
    case zero
    then show ?case
      by simp
  next
    case (mod k l)
    have gcd l (k mod l) = gcd l k
    proof (cases l 0::int rule: linorder-cases)
      case less
      then show ?thesis
        using gcd-non-0-int [of − l − k] by (simp add: ac-simps)
```

```
    next
      case equal
      with mod show ?thesis
        by simp
    next
      case greater
      then show ?thesis
        using gcd-non-0-int [of l k] by (simp add: ac-simps)
    qed
    with mod show ?case
      by (simp add: Euclidean-Algorithm.gcd-mod ac-simps)
  qed
 qed
 show [simp]: (Euclidean-Algorithm.Lcm :: int set ⇒ -) = Lcm
  by (auto intro!: ext Lcm-eqI)
 show (Euclidean-Algorithm.lcm :: int ⇒ -) = lcm
  by (simp add: fun-eq-iff Euclidean-Algorithm.lcm-def semiring-gcd-class.lcm-gcd)
 show (Euclidean-Algorithm.Gcd :: int set ⇒ -) = Gcd
  by (simp add: fun-eq-iff Euclidean-Algorithm.Gcd-def semiring-Gcd-class.Gcd-Lcm)
qed

instance int :: normalization-euclidean-semiring-multiplicative ..

lemma (in idom) prime-CHAR-semidom:
 assumes CHAR('a) > 0
 shows   prime CHAR('a)
proof −
 have False if ab: a ≠ 1 b ≠ 1 CHAR('a) = a * b for a b
 proof −
  from assms ab have a > 0 b > 0
   by (auto intro!: Nat.gr0I)
  have of-nat (a * b) = (0 :: 'a)
   using ab by (metis of-nat-CHAR)
  also have of-nat (a * b) = (of-nat a :: 'a) * of-nat b
   by simp
  finally have of-nat a * of-nat b = (0 :: 'a) .
  moreover have of-nat a * of-nat b ≠ (0 :: 'a)
   using ab ‹a > 0› ‹b > 0›
   by (intro no-zero-divisors) (auto simp: of-nat-eq-0-iff-char-dvd)
  ultimately show False
   by contradiction
 qed
 moreover have CHAR('a) > 1
  using assms CHAR-not-1' by linarith
 ultimately have prime-elem CHAR('a)
  by (intro irreducible-imp-prime-elem) (auto simp: Factorial-Ring.irreducible-def)
 thus ?thesis
  by (auto simp: prime-def)
qed
```

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*)
  using *not-prime-1* [**where** *?'a* = *nat*] **by** *simp*

lemma *prime-ge-2-nat*:
  p ≥ *2* **if** *prime p* **for** *p* :: *nat*
proof −
  from *that* **have** *p* ≠ *0* **and** *p* ≠ *1*
    by (*auto dest*: *prime-elem-not-zeroI prime-elem-not-unit*)
  then show *?thesis*
    by *simp*
qed

lemma *prime-ge-2-int*:
  p ≥ *2* **if** *prime p* **for** *p* :: *int*
proof −
  from *that* **have** *prime-elem p* **and** |*p*| = *p*
    by (*auto dest*: *normalize-prime*)
  then have *p* ≠ *0* **and** |*p*| ≠ *1* **and** *p* ≥ *0*
    by (*auto dest*: *prime-elem-not-zeroI prime-elem-not-unit*)
  then show *?thesis*
    by *simp*
qed

lemma *prime-ge-0-int*: *prime p* ⟹ *p* ≥ (*0*::*int*)
  using *prime-ge-2-int* [*of p*] **by** *simp*

lemma *prime-gt-0-nat*: *prime p* ⟹ *p* > (*0*::*nat*)
  using *prime-ge-2-nat* [*of p*] **by** *simp*

lemma *prime-gt-0-int*: *prime p* ⟹ *p* > (*0*::*int*)
  using *prime-ge-2-int* [*of p*] **by** *simp*

lemma *prime-ge-1-nat*: *prime p* ⟹ *p* ≥ (*1*::*nat*)
  using *prime-ge-2-nat* [*of p*] **by** *simp*

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

```
  using prime-ge-1-nat [of p] by simp

lemma prime-ge-1-int: prime p ⟹ p ≥ (1::int)
  using prime-ge-2-int [of p] by simp

lemma prime-gt-1-nat: prime p ⟹ p > (1::nat)
  using prime-ge-2-nat [of p] by simp

lemma prime-gt-Suc-0-nat: prime p ⟹ p > Suc 0
  using prime-gt-1-nat [of p] by simp

lemma prime-gt-1-int: prime p ⟹ p > (1::int)
  using prime-ge-2-int [of p] by simp

lemma prime-natI:
  prime p if p ≥ 2 and ⋀m n. p dvd m * n ⟹ p dvd m ∨ p dvd n for p :: nat
  using that by (auto intro!: primeI prime-elemI)

lemma prime-intI:
  prime p if p ≥ 2 and ⋀m n. p dvd m * n ⟹ p dvd m ∨ p dvd n for p :: int
  using that by (auto intro!: primeI prime-elemI)

lemma prime-elem-nat-iff [simp]:
  prime-elem n ⟷ prime n for n :: nat
  by (simp add: prime-def)

lemma prime-elem-iff-prime-abs [simp]:
  prime-elem k ⟷ prime |k| for k :: int
  by (auto intro: primeI)

lemma prime-nat-int-transfer [simp]:
  prime (int n) ⟷ prime n (is ?P ⟷ ?Q)
proof
  assume ?P
  then have n ≥ 2
    by (auto dest: prime-ge-2-int)
  then show ?Q
  proof (rule prime-natI)
    fix r s
    assume n dvd r * s
    with of-nat-dvd-iff [of n r * s] have int n dvd int r * int s
      by simp
    with ‹?P› have int n dvd int r ∨ int n dvd int s
      using prime-dvd-mult-iff [of int n int r int s]
      by simp
    then show n dvd r ∨ n dvd s
      by simp
  qed
next
```

```
  assume ?Q
  then have int n ≥ 2
    by (auto dest: prime-ge-2-nat)
  then show ?P
  proof (rule prime-intI)
    fix r s
    assume int n dvd r * s
    then have n dvd nat |r * s|
      by simp
    then have n dvd nat |r| * nat |s|
      by (simp add: nat-abs-mult-distrib)
    with ‹?Q› have n dvd nat |r| ∨ n dvd nat |s|
      using prime-dvd-mult-iff [of n nat |r| nat |s|]
      by simp
    then show int n dvd r ∨ int n dvd s
      by simp
  qed
qed

lemma prime-nat-iff-prime [simp]:
  prime (nat k) ⟷ prime k
proof (cases k ≥ 0)
  case True
  then show ?thesis
    using prime-nat-int-transfer [of nat k] by simp
next
  case False
  then show ?thesis
    by (auto dest: prime-ge-2-int)
qed

lemma prime-int-nat-transfer:
  prime k ⟷ k ≥ 0 ∧ prime (nat k)
  by (auto dest: prime-ge-2-int)

lemma prime-nat-naiveI:
  prime p if p ≥ 2 and dvd: ⋀n. n dvd p ⟹ n = 1 ∨ n = p for p :: nat
proof (rule primeI, rule prime-elemI)
  fix m n :: nat
  assume p dvd m * n
  then obtain r s where p = r * s r dvd m s dvd n
    by (blast dest: division-decomp)
  moreover have r = 1 ∨ r = p
    using ‹r dvd m› ‹p = r * s› dvd [of r] by simp
  ultimately show p dvd m ∨ p dvd n
    by auto
qed (use ‹p ≥ 2› in simp-all)

lemma prime-int-naiveI:
```

```
  prime p if p ≥ 2 and dvd: ⋀k. k dvd p ⟹ |k| = 1 ∨ |k| = p for p :: int
proof −
  from ‹p ≥ 2› have nat p ≥ 2
    by simp
  then have prime (nat p)
  proof (rule prime-nat-naiveI)
    fix n
    assume n dvd nat p
    with ‹p ≥ 2› have n dvd nat |p|
      by simp
    then have int n dvd p
      by simp
    with dvd [of int n] show n = 1 ∨ n = nat p
      by auto
  qed
  then show ?thesis
    by simp
qed

lemma prime-nat-iff:
  prime (n :: nat) ⟷ (1 < n ∧ (∀ m. m dvd n ⟶ m = 1 ∨ m = n))
proof (safe intro!: prime-gt-1-nat)
  assume prime n
  then have ∗: prime-elem n
    by simp
  fix m assume m: m dvd n m ≠ n
  from ∗ ‹m dvd n› have n dvd m ∨ is-unit m
    by (intro irreducibleD′ prime-elem-imp-irreducible)
  with m show m = 1 by (auto dest: dvd-antisym)
next
  assume n > 1 ∀ m. m dvd n ⟶ m = 1 ∨ m = n
  then show prime n
    using prime-nat-naiveI [of n] by auto
qed

lemma prime-nat-iff′:
  prime (p :: nat) ⟷ p > 1 ∧ (∀ n ∈ {2..<p}. ¬ n dvd p)
proof safe
  assume p > 1 and ∗: ∀ n∈{2..<p}. ¬n dvd p
  show prime p unfolding prime-nat-iff
  proof (intro conjI allI impI)
    fix m assume m dvd p
    with ‹p > 1› have m ≠ 0 by (intro notI) auto
    hence m ≥ 1 by simp
    moreover from ‹m dvd p› and ∗ have m ∉ {2..<p} by blast
    with ‹m dvd p› and ‹p > 1› have m ≤ 1 ∨ m = p by (auto dest: dvd-imp-le)
    ultimately show m = 1 ∨ m = p by simp
  qed fact+
qed (auto simp: prime-nat-iff)
```

lemma *prime-int-iff*:
prime (*n*::*int*) ⟷ (*1* < *n* ∧ (∀ *m*. *m* ≥ *0* ∧ *m dvd n* ⟶ *m* = *1* ∨ *m* = *n*))
proof (*intro iffI conjI allI impI*; (*elim conjE*)*?*)
assume ∗: *prime n*
hence *irred*: *irreducible n* **by** (*auto intro*: *prime-elem-imp-irreducible*)
from ∗ **have** *n* ≥ *0 n* ≠ *0 n* ≠ *1*
by (*auto simp add*: *prime-ge-0-int*)
thus *n* > *1* **by** *presburger*
fix *m* **assume** *m dvd n* ‹*m* ≥ *0*›
with *irred* **have** *m dvd 1* ∨ *n dvd m* **by** (*auto simp*: *irreducible-altdef*)
with ‹*m dvd n*› ‹*m* ≥ *0*› ‹*n* > *1*› **show** *m* = *1* ∨ *m* = *n*
using *associated-iff-dvd*[*of m n*] **by** *auto*
next
assume *n*: *1* < *n* ∀ *m*. *m* ≥ *0* ∧ *m dvd n* ⟶ *m* = *1* ∨ *m* = *n*
hence *nat n* > *1* **by** *simp*
moreover have ∀ *m*. *m dvd nat n* ⟶ *m* = *1* ∨ *m* = *nat n*
proof (*intro allI impI*)
fix *m* **assume** *m dvd nat n*
with ‹*n* > *1*› **have** *m dvd nat* |*n*|
by *simp*
then have *int m dvd n*
by *simp*
with *n*(*2*) **have** *int m* = *1* ∨ *int m* = *n*
using *of-nat-0-le-iff* **by** *blast*
thus *m* = *1* ∨ *m* = *nat n* **by** *auto*
qed
ultimately show *prime n*
unfolding *prime-int-nat-transfer prime-nat-iff* **by** *auto*
qed

lemma *prime-int-iff′*:
prime (*p* :: *int*) ⟷ *p* > *1* ∧ (∀ *n* ∈ {*2*..<*p*}. ¬ *n dvd p*) (**is** *?P* ⟷ *?Q*)
proof (*cases p* ≥ *0*)
case *True*
have *?P* ⟷ *prime* (*nat p*)
by *simp*
also have ... ⟷ *p* > *1* ∧ (∀ *n*∈{*2*..<*nat p*}. ¬ *n dvd nat* |*p*|)
using *True* **by** (*simp add*: *prime-nat-iff′*)
also have {*2*..<*nat p*} = *nat* ‘ {*2*..<*p*}
using *True int-eq-iff* **by** *fastforce*
finally show *?P* ⟷ *?Q* **by** *simp*
next
case *False*
then show *?thesis*
by (*auto simp add*: *prime-ge-0-int*)
qed

lemma *prime-nat-not-dvd*:

assumes *prime p* $p > n$ $n \neq (1::nat)$
shows $\neg n$ *dvd p*
proof
assume *n dvd p*
from *assms*(*1*) **have** *irreducible p* **by** (*simp add*: *prime-elem-imp-irreducible*)
from *irreducibleD'*[*OF this* ‹*n dvd p*›] ‹*n dvd p*› ‹$p > n$› *assms* **show** *False*
by (*cases* $n = 0$) (*auto dest*!: *dvd-imp-le*)
qed

lemma *prime-int-not-dvd*:
assumes *prime p* $p > n$ $n > (1::int)$
shows $\neg n$ *dvd p*
proof
assume *n dvd p*
from *assms*(*1*) **have** *irreducible p* **by** (*auto intro*: *prime-elem-imp-irreducible*)
from *irreducibleD'*[*OF this* ‹*n dvd p*›] ‹*n dvd p*› ‹$p > n$› *assms* **show** *False*
by (*auto dest*!: *zdvd-imp-le*)
qed

lemma *prime-odd-nat*: *prime p* $\Longrightarrow$ $p > (2::nat)$ $\Longrightarrow$ *odd p*
by (*intro prime-nat-not-dvd*) *auto*

lemma *prime-odd-int*: *prime p* $\Longrightarrow$ $p > (2::int)$ $\Longrightarrow$ *odd p*
by (*intro prime-int-not-dvd*) *auto*

lemma *prime-int-altdef*:
prime p $= (1 < p \wedge (\forall m::int.\ m \geq 0 \longrightarrow m$ *dvd p* $\longrightarrow$
$m = 1 \vee m = p))$
unfolding *prime-int-iff* **by** *blast*

lemma *not-prime-eq-prod-nat*:
assumes $m > 1$ $\neg$ *prime* $(m::nat)$
shows $\exists n\ k.\ n = m * k \wedge 1 < m \wedge m < n \wedge 1 < k \wedge k < n$
using *assms irreducible-altdef*[*of m*]
by (*auto simp*: *prime-elem-iff-irreducible irreducible-altdef*)

3.2 Make prime naively executable

lemma *prime-int-numeral-eq* [*simp*]:
prime (*numeral m* :: *int*) $\longleftrightarrow$ *prime* (*numeral m* :: *nat*)
by (*simp add*: *prime-int-nat-transfer*)

class *check-prime-by-range* = *normalization-semidom* + *discrete-linordered-semidom*
+
assumes *prime-iff*: ‹*prime a* $\longleftrightarrow$ $1 < a \wedge (\forall d \in \{2..a\ div\ 2\}.\ \neg\ d$ *dvd a*)›
begin

lemma *two-is-prime* [*simp*]:
‹*prime 2*›

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

end

lemma *divisor-less-eq-half-nat*:
‹$m \leq n$ *div* 2› **if** ‹*m dvd n*› ‹$m < n$› **for** *m n* :: *nat*
using *that* **by** (*auto simp add*: *less-eq-div-iff-mult-less-eq*)

instance *nat* :: *check-prime-by-range*
apply *standard*
apply (*auto simp add*: *prime-nat-iff*)
apply (*rule ccontr*)
apply (*auto simp add*: *neq-iff*)
apply (*metis One-nat-def Suc-1 Suc-leI atLeastAtMost-iff divisor-less-eq-half-nat*)
done

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

lemma *divisor-less-eq-half-int*:
‹$k \leq l$ *div* 2› **if** ‹*k dvd l*› ‹$k < l$› ‹$l \geq 0$› ‹$k \geq 0$› **for** *k l* :: *int*
proof −
define *m n* **where** ‹m = *nat* |*k*|› ‹n = *nat* |*l*|›
with ‹$l \geq 0$› ‹$k \geq 0$› **have** ‹k = *int m*› ‹l = *int n*›
by *simp-all*
with *that* **show** *?thesis*
using *divisor-less-eq-half-nat* [*of m n*] **by** *simp*
qed

instance *int* :: *check-prime-by-range*
apply *standard*
apply (*auto simp add*: *prime-int-iff*)
apply (*smt* (*verit*) *int-div-less-self*)
apply (*rule ccontr*)
apply (*auto simp add*: *neq-iff zdvd-not-zless*)
apply (*metis div-by-0 dvd-div-eq-0-iff less-le-not-le one-dvd order-le-less zdvd-not-zless*)
apply (*metis atLeastAtMost-iff divisor-less-eq-half-int dvd-div-eq-0-iff int-one-le-iff-zero-less nle-le one-add-one pos-imp-zdiv-nonneg-iff zdiv-eq-0-iff zless-imp-add1-zle*)
done

lemma *prime-nat-numeral-eq* [*simp*]: — TODO Sieve Of Erathosthenes might speed this up
prime (*numeral m* :: *nat*) ⟷
(1::*nat*) < *numeral m* ∧
(∀ *n*::*nat* ∈ *set* [2..<*Suc* (*numeral m div 2*)]. ¬ *n dvd numeral m*)
using *prime-iff* [*of* ‹*numeral m* :: *nat*›]

by (*simp only*: *set-upt atLeastLessThanSuc-atLeastAtMost*)

context *check-prime-by-range*
begin

definition *check-divisors* :: ‹'*a* ⇒ '*a* ⇒ '*a* ⇒ *bool*›
where ‹*check-divisors l u a* ⟷ (∀ *d*∈{*l*..*u*}. ¬ *d dvd a*)›

lemma *check-divisors-rec* [*code*]:
‹*check-divisors l u a* ⟷ *u* < *l* ∨ (¬ *l dvd a* ∧ *check-divisors* (*l* + *1*) *u a*)›
apply (*auto simp add*: *check-divisors-def not-less*)
apply (*metis local.add-increasing2 local.atLeastAtMost-iff local.linear local.order-eq-iff local.zero-le-one*)
subgoal for *d*
apply (*cases* ‹*l* + *1* ≤ *d*›)
apply (*auto simp add*: *not-le*)
apply (*metis local.dual-order.antisym local.less-eq-iff-succ-less*)
done
done

lemma *prime-eq-check-divisors* [*code*]:
‹*prime a* ⟷ *a* > *1* ∧ *check-divisors 2* (*a div 2*) *a*›
by (*simp add*: *check-divisors-def prime-iff*)

end

3.3 Largest exponent of a prime factor

lemma *prime-factor-nat*:
n ≠ (*1*::*nat*) ⟹ ∃ *p*. *prime p* ∧ *p dvd n*
using *prime-divisor-exists*[*of n*]
by (*cases n* = *0*) (*auto intro*: *exI*[*of* - *2*::*nat*])

lemma *prime-factor-int*:
fixes *k* :: *int*
assumes |*k*| ≠ *1*
obtains *p* **where** *prime p p dvd k*
proof (*cases k* = *0*)
case *True*
then have *prime* (*2*::*int*) **and** *2 dvd k*
by *simp-all*
with *that* **show** *thesis*
by *blast*
next
case *False*
with *assms prime-divisor-exists* [*of k*] **obtain** *p* **where** *prime p p dvd k*
by *auto*
with *that* **show** *thesis*
by *blast*

qed

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

lemma *prime-power-cancel-less*:
assumes *prime p* **and** *eq*: $m * (p \hat{} k) = m' * (p \hat{} k')$ **and** *less*: $k < k'$ **and** $\neg$ *p dvd m*
shows *False*
proof −
obtain *l* **where** *l*: $k' = k + l$ **and** $l > 0$
using *less less-imp-add-positive* **by** *auto*
have $m = m * (p \hat{} k)$ *div* $(p \hat{} k)$
using ‹*prime p*› **by** *simp*
also have $\ldots = m' * (p \hat{} k')$ *div* $(p \hat{} k)$
using *eq* **by** *simp*
also have $\ldots = m' * (p \hat{} l) * (p \hat{} k)$ *div* $(p \hat{} k)$
by (*simp add*: *l mult.commute mult.left-commute power-add*)
also have ... $= m' * (p \hat{} l)$
using ‹*prime p*› **by** *simp*
finally have *p dvd m*
using ‹$l > 0$› **by** *simp*
with *assms* **show** *False*
by *simp*
qed

lemma *prime-power-cancel*:
assumes *prime p* **and** *eq*: $m * (p \hat{} k) = m' * (p \hat{} k')$ **and** $\neg$ *p dvd m* $\neg$ *p dvd* m'
shows $k = k'$
using *prime-power-cancel-less* [*OF* ‹*prime p*›] *assms*
by (*metis linorder-neqE-nat*)

lemma *prime-power-cancel2*:
assumes *prime p* $m * (p \hat{} k) = m' * (p \hat{} k')$ $\neg$ *p dvd m* $\neg$ *p dvd* m'
obtains $m = m'$ $k = k'$
using *prime-power-cancel* [*OF assms*] *assms* **by** *auto*

lemma *prime-power-canonical*:
fixes m :: *nat*
assumes *prime p* $m > 0$
shows $\exists k\ n.\ \neg$ *p dvd n* $\wedge$ $m = n * p \hat{} k$
using ‹$m > 0$›
proof (*induction m rule*: *less-induct*)
case (*less m*)
show *?case*
proof (*cases p dvd m*)
case *True*
then obtain m' **where** m': $m = p * m'$
using *dvdE* **by** *blast*
with ‹*prime p*› **have** $0 < m'$ $m' < m$

```
    using less.prems prime-nat-iff by auto
  with m' less show ?thesis
    by (metis power-Suc mult.left-commute)
 next
  case False
  then show ?thesis
    by (metis mult.right-neutral power-0)
 qed
qed
```

3.4 Infinitely many primes

```
lemma next-prime-bound: ∃ p::nat. prime p ∧ n < p ∧ p ≤ fact n + 1
proof−
 have f1: fact n + 1 ≠ (1::nat) using fact-ge-1 [of n, where 'a=nat] by arith
 from prime-factor-nat [OF f1]
 obtain p :: nat where prime p and p dvd fact n + 1 by auto
 then have p ≤ fact n + 1 apply (intro dvd-imp-le) apply auto done
 { assume p ≤ n
  from ‹prime p› have p ≥ 1
    by (cases p, simp-all)
  with ‹p <= n› have p dvd fact n
    by (intro dvd-fact)
  with ‹p dvd fact n + 1› have p dvd fact n + 1 − fact n
    by (rule dvd-diff-nat)
  then have p dvd 1 by simp
  then have p <= 1 by auto
  moreover from ‹prime p› have p > 1
    using prime-nat-iff by blast
  ultimately have False by auto}
 then have n < p by presburger
 with ‹prime p› and ‹p <= fact n + 1› show ?thesis by auto
qed

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

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

3.5 Powers of Primes

Versions for type nat only

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

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

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

lemma *divides-primepow-nat*:
fixes *p* :: *nat*
assumes *p*: *prime p*
shows *d dvd p* $\hat{}$ *k* $\longleftrightarrow$ ($\exists i{\leq}k.\ d = p\ \hat{}\ i$)
using *assms divides-primepow* [*of p d k*] **by** (*auto intro*: *le-imp-power-dvd*)

lemma *gcd-prime-int*:
assumes *prime* (*p* :: *int*)
shows *gcd p k* = (*if p dvd k then p else 1*)
proof −
have $p \geq 0$
using *assms prime-ge-0-int* **by** *auto*
show *?thesis*
proof (*cases p dvd k*)
case *True*
thus *?thesis* **using** *assms* ‹$p \geq 0$› **by** *auto*
next
case *False*
hence *coprime p k*
using *assms* **by** (*simp add*: *prime-imp-coprime*)
with *False* **show** *?thesis*
by *auto*
qed
qed

3.6 Chinese Remainder Theorem Variants

lemma *bezout-gcd-nat*:
fixes *a*::*nat* **shows** $\exists x\ y.\ a * x - b * y = gcd\ a\ b \vee b * x - a * y = gcd\ a\ b$
using *bezout-nat*[*of a b*]
by (*metis bezout-nat diff-add-inverse gcd-add-mult gcd.commute*

gcd-nat.right-neutral mult-0)

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

A binary form of the Chinese Remainder Theorem.

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

Primality

lemma *coprime-bezout-strong*:
fixes *a*::*nat* **assumes** *coprime a b* $b \neq 1$
shows $\exists x\ y.\ a * x = b * y + 1$
by (*metis add.commute add.right-neutral assms*(*1*) *assms*(*2*) *chinese-remainder coprime-1-left coprime-1-right coprime-crossproduct-nat mult.commute mult.right-neutral mult-cancel-left*)

lemma *bezout-prime*:
assumes *p*: *prime p* **and** *pa*: ¬ *p dvd a*
shows $\exists x\ y.\ a*x = Suc\ (p*y)$
proof −
have *ap*: *coprime a p*
using *coprime-commute p pa prime-imp-coprime* **by** *auto*
moreover from *p* **have** $p \neq 1$ **by** *auto*
ultimately have $\exists x\ y.\ a * x = p * y + 1$

by (*rule coprime-bezout-strong*)
then show *?thesis* **by** *simp*
qed

3.7 Multiplicity and primality for natural numbers and integers

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

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

lemma *prime-factors-ge-0-int* [*elim*]:
fixes $n :: int$
shows $p \in$ *prime-factors* $n \Longrightarrow p \geq 0$
by (*drule prime-factors-gt-0-int*) *simp*

lemma *prod-mset-prime-factorization-int*:
fixes $n :: int$
assumes $n > 0$
shows *prod-mset* (*prime-factorization* n) $= n$
using *assms* **by** (*simp add*: *prod-mset-prime-factorization*)

lemma *prime-factorization-exists-nat*:
$n > 0 \Longrightarrow (\exists M.\ (\forall p::nat \in$ *set-mset* $M.$ *prime* $p) \wedge n = (\prod i \in\# M.\ i))$
using *prime-factorization-exists*[*of* n] **by** *auto*

lemma *prod-mset-prime-factorization-nat* [*simp*]:
$(n::nat) > 0 \Longrightarrow$ *prod-mset* (*prime-factorization* n) $= n$
by (*subst prod-mset-prime-factorization*) *simp-all*

lemma *prime-factorization-nat*:
$n > (0::nat) \Longrightarrow n = (\prod p \in$ *prime-factors* $n.\ p \mathbin{\hat{}}$ *multiplicity* $p\ n)$
by (*simp add*: *prod-prime-factors*)

lemma *prime-factorization-int*:
$n > (0::int) \Longrightarrow n = (\prod p \in$ *prime-factors* $n.\ p \mathbin{\hat{}}$ *multiplicity* $p\ n)$
by (*simp add*: *prod-prime-factors*)

lemma *prime-factorization-unique-nat*:
fixes $f :: nat \Rightarrow$ -
assumes *S-eq*: $S = \{p.\ 0 < f\ p\}$
and *finite* S
and S: $\forall p{\in}S.$ *prime* $p\ n = (\prod p{\in}S.\ p \mathbin{\hat{}} f\ p)$
shows $S =$ *prime-factors* $n \wedge (\forall p.$ *prime* $p \longrightarrow f\ p =$ *multiplicity* $p\ n)$
using *assms* **by** (*intro prime-factorization-unique''*) *auto*

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 $S = prime\text{-}factors\ n \wedge (\forall\, p.\ prime\ p \longrightarrow f\ p = multiplicity\ p\ n)$
using *assms* **by** (*intro prime-factorization-unique''*) *auto*

lemma *prime-factors-characterization-nat*:
$S = \{p.\ 0 < f\ (p{::}nat)\} \Longrightarrow$
finite $S \Longrightarrow \forall\, p{\in}S.\ prime\ p \Longrightarrow n = (\prod p{\in}S.\ p\ \hat{}\ f\ p) \Longrightarrow prime\text{-}factors\ n = S$
by (*rule prime-factorization-unique-nat* [*THEN conjunct1*, *symmetric*])

lemma *prime-factors-characterization'-nat*:
finite $\{p.\ 0 < f\ (p{::}nat)\} \Longrightarrow$
$(\forall\, p.\ 0 < f\ p \longrightarrow prime\ p) \Longrightarrow$
$prime\text{-}factors\ (\prod p \mid 0 < f\ p.\ p\ \hat{}\ f\ p) = \{p.\ 0 < f\ p\}$
by (*rule prime-factors-characterization-nat*) *auto*

lemma *prime-factors-characterization-int*:
$S = \{p.\ 0 < f\ (p{::}int)\} \Longrightarrow$ *finite* $S \Longrightarrow$
$\forall\, p{\in}S.\ prime\ p \Longrightarrow abs\ n = (\prod p{\in}S.\ p\ \hat{}\ f\ p) \Longrightarrow prime\text{-}factors\ n = S$
by (*rule prime-factorization-unique-int* [*THEN conjunct1*, *symmetric*])

lemma *abs-prod*: $abs\ (prod\ f\ A :: 'a :: linordered\text{-}idom) = prod\ (\lambda x.\ abs\ (f\ x))\ A$
by (*cases finite A*, *induction A rule*: *finite-induct*) (*simp-all add*: *abs-mult*)

lemma *primes-characterization'-int* [*rule-format*]:
finite $\{p.\ p \geq 0 \wedge 0 < f\ (p{::}int)\} \Longrightarrow \forall\, p.\ 0 < f\ p \longrightarrow prime\ p \Longrightarrow$
$prime\text{-}factors\ (\prod p \mid p \geq 0 \wedge 0 < f\ p.\ p\ \hat{}\ f\ p) = \{p.\ p \geq 0 \wedge 0 < f\ p\}$
by (*rule prime-factors-characterization-int*) (*auto simp*: *abs-prod prime-ge-0-int*)

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

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

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

symmetric])
(*auto simp*: *abs-prod power-abs prime-ge-0-int intro*!: *prod.cong*)

lemma *multiplicity-characterization'-int* [*rule-format*]:
finite {*p*. *p* ≥ *0* ∧ *0* < *f* (*p*::*int*)} ⟹
(∀ *p*. *0* < *f p* ⟶ *prime p*) ⟹ *prime p* ⟹
multiplicity p (∏ *p* | *p* ≥ *0* ∧ *0* < *f p*. *p* ^ *f p*) = *f p*
by (*rule multiplicity-characterization-int*) (*auto simp*: *prime-ge-0-int*)

lemma *multiplicity-one-nat* [*simp*]: *multiplicity p* (*Suc 0*) = *0*
unfolding *One-nat-def* [*symmetric*] **by** (*rule multiplicity-one*)

lemma *multiplicity-eq-nat*:
fixes *x* **and** *y*::*nat*
assumes *x* > *0* *y* > *0* ⋀*p*. *prime p* ⟹ *multiplicity p x* = *multiplicity p y*
shows *x* = *y*
using *multiplicity-eq-imp-eq*[*of x y*] *assms* **by** *simp*

lemma *multiplicity-eq-int*:
fixes *x y* :: *int*
assumes *x* > *0* *y* > *0* ⋀*p*. *prime p* ⟹ *multiplicity p x* = *multiplicity p y*
shows *x* = *y*
using *multiplicity-eq-imp-eq*[*of x y*] *assms* **by** *simp*

lemma *multiplicity-prod-prime-powers*:
assumes *finite S* ⋀*x*. *x* ∈ *S* ⟹ *prime x prime p*
shows *multiplicity p* (∏ *p* ∈ *S*. *p* ^ *f p*) = (*if p* ∈ *S then f p else 0*)
proof −
define *g* **where** *g* = (λ*x*. *if x* ∈ *S then f x else 0*)
define *A* **where** *A* = *Abs-multiset g*
have {*x*. *g x* > *0*} ⊆ *S* **by** (*auto simp*: *g-def*)
from *finite-subset*[*OF this assms*(*1*)] **have** [*simp*]: *finite* {*x*. *0* < *g x*}
by *simp*
from *assms* **have** *count-A*: *count A x* = *g x* **for** *x* **unfolding** *A-def*
by *simp*
have *set-mset-A*: *set-mset A* = {*x*∈*S*. *f x* > *0*}
unfolding *set-mset-def count-A* **by** (*auto simp*: *g-def*)
with *assms* **have** *prime*: *prime x* **if** *x* ∈# *A* **for** *x* **using** *that* **by** *auto*
from *set-mset-A assms* **have** (∏ *p* ∈ *S*. *p* ^ *f p*) = (∏ *p* ∈ *S*. *p* ^ *g p*)
by (*intro prod.cong*) (*auto simp*: *g-def*)
also from *set-mset-A assms* **have** ... = (∏ *p* ∈ *set-mset A*. *p* ^ *g p*)
by (*intro prod.mono-neutral-right*) (*auto simp*: *g-def set-mset-A*)
also have ... = *prod-mset A*
by (*auto simp*: *prod-mset-multiplicity count-A set-mset-A intro*!: *prod.cong*)
also from *assms* **have** *multiplicity p* ... = *sum-mset* (*image-mset* (*multiplicity p*) *A*)
by (*subst prime-elem-multiplicity-prod-mset-distrib*) (*auto dest*: *prime*)
also from *assms* **have** *image-mset* (*multiplicity p*) *A* = *image-mset* (λ*x*. *if x* = *p then 1 else 0*) *A*

by (*intro image-mset-cong*) (*auto simp*: *prime-multiplicity-other dest*: *prime*)
also have *sum-mset* ... = (*if p* ∈ *S then f p else 0*) **by** (*simp add*: *sum-mset-delta count-A g-def*)
finally show *?thesis* **.**
qed

lemma *prime-factorization-prod-mset*:
assumes *0* ∉# *A*
shows *prime-factorization* (*prod-mset A*) = ∑ #(*image-mset prime-factorization A*)
using *assms* **by** (*induct A*) (*auto simp add*: *prime-factorization-mult*)

lemma *prime-factors-prod*:
assumes *finite A* **and** *0* ∉ *f* ‘ *A*
shows *prime-factors* (*prod f A*) = ⋃((*prime-factors* ∘ *f*) ‘ *A*)
using *assms* **by** (*simp add*: *prod-unfold-prod-mset prime-factorization-prod-mset*)

lemma *prime-factors-fact*:
prime-factors (*fact n*) = {*p* ∈ {*2*..*n*}. *prime p*} (**is** *?M* = *?N*)
proof (*rule set-eqI*)
fix *p*
{ fix *m* :: *nat*
assume *p* ∈ *prime-factors m*
then have *prime p* **and** *p dvd m* **by** *auto*
moreover assume *m* > *0*
ultimately have *2* ≤ *p* **and** *p* ≤ *m*
by (*auto intro*: *prime-ge-2-nat dest*: *dvd-imp-le*)
moreover assume *m* ≤ *n*
ultimately have *2* ≤ *p* **and** *p* ≤ *n*
by (*auto intro*: *order-trans*)
} note ∗ = *this*
show *p* ∈ *?M* ⟷ *p* ∈ *?N*
by (*auto simp add*: *fact-prod prime-factors-prod Suc-le-eq dest*!: *prime-prime-factors intro*: ∗)
qed

lemma *prime-dvd-fact-iff*:
assumes *prime p*
shows *p dvd fact n* ⟷ *p* ≤ *n*
using *assms*
by (*auto simp add*: *prime-factorization-subset-iff-dvd* [*symmetric*]
prime-factorization-prime prime-factors-fact prime-ge-2-nat)

lemma *dvd-choose-prime*:
assumes *kn*: *k* < *n* **and** *k*: *k* ≠ *0* **and** *n*: *n* ≠ *0* **and** *prime-n*: *prime n*
shows *n dvd* (*n choose k*)
proof −
have *n dvd* (*fact n*) **by** (*simp add*: *fact-num-eq-if n*)
moreover have ¬ *n dvd* (*fact k* ∗ *fact* (*n*−*k*))

by (*metis prime-dvd-fact-iff prime-dvd-mult-iff assms neq0-conv diff-less linorder-not-less*)
moreover have (*fact n*::*nat*) = *fact k* ∗ *fact* (*n*−*k*) ∗ (*n choose k*)
using *binomial-fact-lemma kn* **by** *auto*
ultimately show *?thesis* **using** *prime-n*
by (*auto simp add*: *prime-dvd-mult-iff*)
qed

lemma (**in** *ring-1*) *minus-power-prime-CHAR*:
assumes *p* = *CHAR*('*a*) *prime p*
shows (−*x* :: '*a*) ^ *p* = −(*x* ^ *p*)
proof (*cases p* = *2*)
case *False*
have *prime p*
using *assms* **by** *blast*
hence *odd p*
using *prime-imp-coprime assms False coprime-right-2-iff-odd gcd-nat.strict-iff-not*
by *blast*
thus *?thesis*
by *simp*
qed (*use assms* **in** ‹*auto simp*: *uminus-CHAR-2*›)

3.8 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*
using *local.CHAR-pos-iff local.prime-char-aux prime-gt-0-nat* **by** *blast*

lemma *CHAR-nonzero* [*simp*]: *CHAR*('*a*) ≠ *0*
using *CHAR-pos* **by** *auto*

lemma *CHAR-prime* [*intro*, *simp*]: *prime CHAR*('*a*)
by (*metis* (*mono-tags*, *lifting*) *gcd-nat.order-iff-strict local.of-nat-1 local.of-nat-eq-0-iff-char-dvd*
local.one-neq-zero local.prime-char-aux prime-nat-iff)

end

lemma *semiring-prime-charI* [*intro?*]:
prime CHAR('*a* :: *semiring-1*) ⟹ *OFCLASS*('*a*, *semiring-prime-char-class*)
by *standard auto*

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

using *assms prime-CHAR-semidom* **by** *blast*
qed

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* **..**
end
class *idom-prime-char* = *idom* + *semiring-prime-char*
begin
subclass *comm-ring-prime-char* **..**
end

class *field-prime-char* = *field* +
assumes *pos-char-exists*: ∃ *n*>*0*. *of-nat n* = (*0* :: *'a*)
begin
subclass *idom-prime-char*
apply *standard*
using *pos-char-exists local.CHAR-pos-iff local.of-nat-CHAR local.prime-CHAR-semidom*
by *blast*
end

lemma *field-prime-charI* [*intro?*]:
n > *0* ⟹ *of-nat n* = (*0* :: *'a* :: *field*) ⟹ *OFCLASS*(*'a*, *field-prime-char-class*)
by *standard auto*

lemma *field-prime-charI'* [*intro?*]:
CHAR(*'a* :: *field*) > *0* ⟹ *OFCLASS*(*'a*, *field-prime-char-class*)
by *standard auto*

3.9 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 *standard*
show ∃ *n*>*0*. *of-nat n* = (*0* :: *'a*)
using *assms prime-CHAR-semidom*[**where** *?'a* = *'a*] *finite-imp-CHAR-pos*[*OF assms*]
by (*intro exI*[*of* - *CHAR*(*'a*)]) *auto*
qed *fact+*

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 $x :: 'a :: finite\text{-}field$
shows $x \hat{} card\ (UNIV :: 'a\ set) = x$
proof (*cases* $x = 0$)
case *False*
have $x * (\prod y{\in}UNIV - \{0\}.\ x * y) = x * x \hat{} (card\ (UNIV :: 'a\ set) - 1) * \prod (UNIV - \{0\})$
by (*simp add*: *prod.distrib mult-ac*)
also have $x * x \hat{} (card\ (UNIV :: 'a\ set) - 1) = x \hat{} Suc\ (card\ (UNIV :: 'a\ set) - 1)$
by (*subst power-Suc*) *auto*
also have $Suc\ (card\ (UNIV :: 'a\ set) - 1) = card\ (UNIV :: 'a\ set)$
using *finite-UNIV-card-ge-0*[**where** $?'a = 'a$] **by** *simp*
also have $(\prod y{\in}UNIV - \{0\}.\ x * y) = (\prod y{\in}UNIV - \{0\}.\ y)$
by (*rule prod.reindex-bij-witness*[*of* - $\lambda y.\ y$ / x $\lambda y.\ x * y$]) (*use False* **in** *auto*)
finally show *?thesis*
by *simp*
qed (*use finite-UNIV-card-ge-0*[**where** $?'a = 'a$] **in** *auto*)

lemma *finite-field-power-card-power-eq-same*:
fixes $x :: 'a :: finite\text{-}field$
assumes $m = card\ (UNIV :: 'a\ set) \hat{} n$
shows $x \hat{} m = x$
unfolding *assms*
by (*induction n*) (*simp-all add*: *finite-field-power-card-eq-same power-mult*)

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

lemma *inj-on-enum-finite-field*: *inj-on enum-finite-field* $\{..<card\ (UNIV :: 'a\ set)\}$
using *enum-finite-field* **by** (*simp add*: *eq-card-imp-inj-on*)

end

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

typedef *gf2* = $\{0, 1 :: nat\}$
by *auto*

setup-lifting *type-definition-gf2*

instantiation *gf2* :: *field*
begin
lift-definition *zero-gf2* :: *gf2* **is** *0* **by** *auto*
lift-definition *one-gf2* :: *gf2* **is** *1* **by** *auto*

lift-definition *uminus-gf2* :: *gf2* ⇒ *gf2* **is** λ*x*. *x* **.**
lift-definition *plus-gf2* :: *gf2* ⇒ *gf2* ⇒ *gf2* **is** λ*x y*. *if x* = *y then 0 else 1* **by** *auto*
lift-definition *minus-gf2* :: *gf2* ⇒ *gf2* ⇒ *gf2* **is** λ*x y*. *if x* = *y then 0 else 1* **by** *auto*
lift-definition *times-gf2* :: *gf2* ⇒ *gf2* ⇒ *gf2* **is** λ*x y*. *x* ∗ *y* **by** *auto*
lift-definition *inverse-gf2* :: *gf2* ⇒ *gf2* **is** λ*x*. *x* **.**
lift-definition *divide-gf2* :: *gf2* ⇒ *gf2* ⇒ *gf2* **is** λ*x y*. *x* ∗ *y* **by** *auto*

instance
by *standard* (*transfer*; *fastforce*)+

end

instance *gf2* :: *finite-field*
proof
interpret *type-definition Rep-gf2 Abs-gf2* {*0*, *1* :: *nat*}
by (*rule type-definition-gf2*)
show *finite* (*UNIV* :: *gf2 set*)
by (*metis Abs-image finite.emptyI finite.insertI finite-imageI*)
qed

3.10 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 −
interpret *comm-semiring-prime-char*
by *standard* (*auto intro*!: *exI*[*of* - *CHAR*('*a*)] *assms*)
have *n* > *0*
unfolding *n-def* **by** *simp*
have (*x* + *y*) ^ *n* = (∑ *k*≤*n*. *of-nat* (*n choose k*) ∗ *x* ^ *k* ∗ *y* ^ (*n* − *k*))
by (*rule binomial-ring*)
also have ... = (∑ *k*∈{*0*,*n*}. *of-nat* (*n choose k*) ∗ *x* ^ *k* ∗ *y* ^ (*n* − *k*))
proof (*intro sum.mono-neutral-right ballI*)
fix *k* **assume** *k* ∈ {..*n*} − {*0*, *n*}
hence *k*: *k* > *0* *k* < *n*
by *auto*
have *CHAR*('*a*) *dvd* (*n choose k*)
unfolding *n-def*
by (*rule dvd-choose-prime*) (*use k* **in** ‹*auto simp*: *n-def*›)
hence *of-nat* (*n choose k*) = (*0* :: *'a*)
using *of-nat-eq-0-iff-char-dvd* **by** *blast*
thus *of-nat* (*n choose k*) ∗ *x* ^ *k* ∗ *y* ^ (*n* − *k*) = *0*
by *simp*
qed *auto*
finally show *?thesis*

using ‹$n > 0$› **by** (*simp add*: *add-ac*)
qed

lemma (**in** *comm-semiring-1*) *freshmans-dream'*:
assumes [*simp*]: *prime CHAR*('*a*) **and** $m = CHAR('a) \hat{} n$
shows $(x + y :: 'a) \hat{} m = x \hat{} m + y \hat{} m$
unfolding *assms*(*2*)
proof (*induction n*)
case (*Suc n*)
have $(x + y) \hat{} (CHAR('a) \hat{} n * CHAR('a)) = ((x + y) \hat{} (CHAR('a) \hat{} n)) \hat{} CHAR('a)$
by (*rule power-mult*)
thus *?case*
by (*simp add*: *Suc.IH freshmans-dream Groups.mult-ac flip*: *power-mult*)
qed *auto*

lemma (**in** *comm-semiring-1*) *freshmans-dream-sum*:
fixes $f :: 'b \Rightarrow 'a$
assumes *prime CHAR*('*a*) **and** $n = CHAR('a)$
shows $sum\ f\ A \hat{} n = sum\ (\lambda i.\ f\ i \hat{} n)\ A$
using *assms*
by (*induct A rule*: *infinite-finite-induct*)
(*auto simp add*: *power-0-left freshmans-dream*)

lemma (**in** *comm-semiring-1*) *freshmans-dream-sum'*:
fixes $f :: 'b \Rightarrow 'a$
assumes *prime CHAR*('*a*) $m = CHAR('a) \hat{} n$
shows $sum\ f\ A \hat{} m = sum\ (\lambda i.\ f\ i \hat{} m)\ A$
using *assms*
by (*induction A rule*: *infinite-finite-induct*)
(*auto simp*: *freshmans-dream' power-0-left*)

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*]

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*›
using *that mod-0-imp-dvd* [*of a b*] **by** *blast*

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 ##* [] = []
by (*simp add*: *cCons-def*)

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

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

lemma *cCons-not-0-eq* [*simp*]: *x* ≠ *0* ⟹ *x ## xs* = *x # xs*
by (*simp add*: *cCons-def*)

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 (*cases x* = *0*)
case *False*
then show *?thesis* **by** *simp*

next
case *True*
show *?thesis*
proof (*induct xs rule*: *rev-induct*)
case *Nil*
with *True* **show** *?case* **by** *simp*
next
case (*snoc y ys*)
then show *?case*
by (*cases y = 0*) (*simp-all add*: *append-Cons* [*symmetric*] *del*: *append-Cons*)
qed
qed

lemma *tl-cCons* [*simp*]: *tl* (*x ## xs*) = *xs*
by (*simp add*: *cCons-def*)

4.2 Definition of type *poly*

typedef (**overloaded**) *'a poly* = {*f* :: *nat* ⇒ *'a*::*zero*. ∀ ∞ *n*. *f n* = *0*}
morphisms *coeff Abs-poly*
by (*auto intro*!: *ALL-MOST*)

setup-lifting *type-definition-poly*

lemma *poly-eq-iff*: *p* = *q* ⟷ (∀ *n*. *coeff p n* = *coeff q n*)
by (*simp add*: *coeff-inject* [*symmetric*] *fun-eq-iff*)

lemma *poly-eqI*: (⋀*n*. *coeff p n* = *coeff q n*) ⟹ *p* = *q*
by (*simp add*: *poly-eq-iff*)

lemma *MOST-coeff-eq-0*: ∀ ∞ *n*. *coeff p n* = *0*
using *coeff* [*of p*] **by** *simp*

lemma *coeff-Abs-poly*:
assumes ⋀*i*. *i* > *n* ⟹ *f i* = *0*
shows *coeff* (*Abs-poly f*) = *f*
proof (*rule Abs-poly-inverse*, *clarify*)
have *eventually* (λ*i*. *i* > *n*) *cofinite*
by (*auto simp*: *MOST-nat*)
thus *eventually* (λ*i*. *f i* = *0*) *cofinite*
by *eventually-elim* (*use assms* **in** *auto*)
qed

4.3 Degree of a polynomial

definition *degree* :: *'a*::*zero poly* ⇒ *nat*
where *degree p* = (*LEAST n*. ∀ *i*>*n*. *coeff p i* = *0*)

lemma *degree-cong*:
assumes ⋀*i*. *coeff p i* = *0* ⟷ *coeff q i* = *0*

shows *degree p = degree q*
proof −
have (λn. ∀ i>n. *poly.coeff p i = 0*) = (λn. ∀ i>n. *poly.coeff q i = 0*)
using *assms* **by** (*auto simp*: *fun-eq-iff*)
thus *?thesis*
by (*simp only*: *degree-def*)
qed

lemma *coeff-Abs-poly-If-le*:
coeff (*Abs-poly* (λi. *if i ≤ n then f i else 0*)) = (λi. *if i ≤ n then f i else 0*)
proof (*rule Abs-poly-inverse*, *clarify*)
have *eventually* (λi. *i > n*) *cofinite*
by (*auto simp*: *MOST-nat*)
thus *eventually* (λi. (*if i ≤ n then f i else 0*) = *0*) *cofinite*
by *eventually-elim auto*
qed

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

lemma *le-degree*: *coeff p n ≠ 0* ⟹ *n ≤ degree p*
using *coeff-eq-0 linorder-le-less-linear* **by** *blast*

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

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

lemma *poly-eqI2*:
assumes *degree p = degree q* **and** ⋀i. *i ≤ degree p* ⟹ *coeff p i = coeff q i*
shows *p = q*
by (*metis assms le-degree poly-eqI*)

4.4 The zero polynomial

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

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

by (*rule MOST-I*) *simp*

instance ..

end

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

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

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

lemma *leading-coeff-0-iff* [*simp*]: *coeff p* (*degree p*) *= 0* ⟷ *p = 0*
by (*cases p = 0*) (*simp-all add*: *leading-coeff-neq-0*)

lemma *degree-lessI*:
assumes *p ≠ 0* ∨ *n > 0* ∀ *k*≥*n*. *coeff p k = 0*
shows *degree p < n*
proof (*cases p = 0*)
case *False*
show *?thesis*
proof (*rule ccontr*)

```
    assume *: ¬(degree p < n)
    define d where d = degree p
    from ‹p ≠ 0› have coeff p d ≠ 0
      by (auto simp: d-def)
    moreover have coeff p d = 0
      using assms(2) * by (auto simp: not-less)
    ultimately show False by contradiction
  qed
qed (use assms in auto)

lemma eq-zero-or-degree-less:
  assumes degree p ≤ n and coeff p n = 0
  shows p = 0 ∨ degree p < n
proof (cases n)
  case 0
  with ‹degree p ≤ n› and ‹coeff p n = 0› have coeff p (degree p) = 0
    by simp
  then have p = 0 by simp
  then show ?thesis ..
next
  case (Suc m)
  from ‹degree p ≤ n› have ∀ i>n. coeff p i = 0
    by (simp add: coeff-eq-0)
  with ‹coeff p n = 0› have ∀ i≥n. coeff p i = 0
    by (simp add: le-less)
  with ‹n = Suc m› have ∀ i>m. coeff p i = 0
    by (simp add: less-eq-Suc-le)
  then have degree p ≤ m
    by (rule degree-le)
  with ‹n = Suc m› have degree p < n
    by (simp add: less-Suc-eq-le)
  then show ?thesis ..
qed

lemma coeff-0-degree-minus-1: coeff rrr dr = 0 ⟹ degree rrr ≤ dr ⟹ degree
rrr ≤ dr − 1
  using eq-zero-or-degree-less by fastforce
```

4.5 List-style constructor for polynomials

```
lift-definition pCons :: 'a::zero ⇒ 'a poly ⇒ 'a poly
  is λa p. case-nat a (coeff p)
  by (rule MOST-SucD) (simp add: MOST-coeff-eq-0)

lemmas coeff-pCons = pCons.rep-eq

lemma coeff-pCons': poly.coeff (pCons c p) n = (if n = 0 then c else poly.coeff p
(n − 1))
  by transfer'(auto split: nat.splits)
```

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

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

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

lemma *degree-pCons-eq*: *p* ≠ *0* ⟹ *degree* (*pCons a p*) = *Suc* (*degree p*)
by (*simp add*: *degree-pCons-le le-antisym le-degree*)

lemma *degree-pCons-0*: *degree* (*pCons a 0*) = *0*
proof −
have *degree* (*pCons a 0*) ≤ *Suc 0*
by (*metis* (*no-types*) *degree-0 degree-pCons-le*)
then show *?thesis*
by (*metis coeff-0 coeff-pCons-Suc degree-0 eq-zero-or-degree-less less-Suc0*)
qed

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

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

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

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

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

```
  show p = pCons (coeff p 0) (Abs-poly (λn. coeff p (Suc n)))
    by transfer
      (simp-all add: MOST-inj[where f=Suc and P=λn. p n = 0 for p] fun-eq-iff
Abs-poly-inverse
        split: nat.split)
qed

lemma pCons-induct [case-names 0 pCons, induct type: poly]:
  assumes zero: P 0
  assumes pCons: ⋀a p. a ≠ 0 ∨ p ≠ 0 ⟹ P p ⟹ P (pCons a p)
  shows P p
proof (induct p rule: measure-induct-rule [where f=degree])
  case (less p)
  obtain a q where p = pCons a q by (rule pCons-cases)
  have P q
  proof (cases q = 0)
    case True
    then show P q by (simp add: zero)
  next
    case False
    then have degree (pCons a q) = Suc (degree q)
      by (rule degree-pCons-eq)
    with ‹p = pCons a q› have degree q < degree p
      by simp
    then show P q
      by (rule less.hyps)
  qed
  have P (pCons a q)
  proof (cases a ≠ 0 ∨ q ≠ 0)
    case True
    with ‹P q› show ?thesis by (auto intro: pCons)
  next
    case False
    with zero show ?thesis by simp
  qed
  with ‹p = pCons a q› show ?case
    by simp
qed

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

by *simp*
then show *thesis* **..**
qed

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* (‹(‹*indent=2 notation=‹mixfix polynomial enumeration›*›[:-:])›)
syntax-consts
-poly ⇌ *pCons*
translations
[:*x*, *xs*:] ⇌ *CONST pCons x* [:*xs*:]
[:*x*:] ⇌ *CONST pCons x 0*

lemma *degree-0-id*:
assumes *degree p = 0*
shows [: *coeff p 0* :] = *p*
by (*metis assms coeff-pCons-0 degree-eq-zeroE*)

lemma *degree0-coeffs*: *degree p = 0* ⟹ ∃ *a*. *p* = [: *a* :]
by (*meson degree-eq-zeroE*)

lemma *degree1-coeffs*:
fixes *p* :: *'a*::*zero poly*
assumes *degree p = 1*
obtains *a b* **where** *p* = [: *b*, *a* :] *a* ≠ *0*
proof −
obtain *b a q* **where** *p = pCons b q q = pCons a 0*
by (*metis assms degree0-coeffs degree-0 degree-pCons-eq-if lessI less-one pCons-cases*)
then show *thesis*
using *assms that* **by** *force*
qed

lemma *degree2-coeffs*:
fixes *p* :: *'a*::*zero poly*
assumes *degree p = 2*
obtains *a b c* **where** *p* = [: *c*, *b*, *a* :] *a* ≠ *0*
proof −
obtain *c q* **where** *p = pCons c q degree q = 1*
by (*metis One-nat-def assms degree-0 degree-pCons-eq-if fact-0 fact-2 nat.inject numeral-2-eq-2 pCons-cases*)
then show *thesis*
by (*metis degree1-coeffs that*)
qed

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*
by (*induct n*) *simp-all*

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

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

lemma *Poly-snoc-zero* [*simp*]: *Poly* (*as* @ [*0*]) = *Poly as*
using *Poly-append-replicate-0* [*of as 1*] **by** *simp*

lemma *Poly-cCons-eq-pCons-Poly* [*simp*]: *Poly* (*a* ## *p*) = *pCons a* (*Poly p*)
by (*simp add*: *cCons-def*)

lemma *Poly-on-rev-starting-with-0* [*simp*]: *hd as* = *0* ⟹ *Poly* (*rev* (*tl as*)) = *Poly* (*rev as*)
by (*cases as*) *simp-all*

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

lemma *coeff-Poly-eq* [*simp*]: *coeff* (*Poly xs*) = *nth-default 0 xs*
by (*induct xs*) (*simp-all add*: *fun-eq-iff coeff-pCons split*: *nat.splits*)

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

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

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

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

lemma *coeffs-pCons-eq-cCons* [*simp*]: *coeffs* (*pCons a p*) = *a* ## *coeffs p*
proof −
have ∗: ∀ *m*∈*set ms*. *m* > *0* ⟹ *map* (*case-nat x f*) *ms* = *map f* (*map* (λ*n*. *n* − *1*) *ms*)
for *ms* :: *nat list* **and** *f* :: *nat* ⇒ *'a* **and** *x* :: *'a*
by (*induct ms*) (*auto split*: *nat.split*)

show *?thesis*
by (*simp add*: ∗ *coeffs-def upt-conv-Cons coeff-pCons map-decr-upt del*: *upt-Suc*)
qed

lemma *length-coeffs*: $p \neq 0 \Longrightarrow$ *length* (*coeffs p*) = *degree p* + *1*
by (*simp add*: *coeffs-def*)

lemma *coeffs-nth*: $p \neq 0 \Longrightarrow n \leq$ *degree p* $\Longrightarrow$ *coeffs p* ! *n* = *coeff p n*
by (*auto simp*: *coeffs-def simp del*: *upt-Suc*)

lemma *coeff-in-coeffs*: $p \neq 0 \Longrightarrow n \leq$ *degree p* $\Longrightarrow$ *coeff p n* ∈ *set* (*coeffs p*)
using *coeffs-nth* [*of p n*, *symmetric*] **by** (*simp add*: *length-coeffs*)

lemma *not-0-cCons-eq* [*simp*]: $p \neq 0 \Longrightarrow a$ *##* *coeffs p* = *a* # *coeffs p*
by (*simp add*: *cCons-def*)

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

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

lemma *no-trailing-coeffs* [*simp*]:
no-trailing (*HOL.eq 0*) (*coeffs p*)
by (*induct p*) *auto*

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

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

lemma *nth-default-coeffs-eq*: *nth-default 0 (coeffs p) = coeff p*
by (*metis Poly-coeffs coeff-Poly-eq*)

lemma *range-coeff*: *range (coeff p) = insert 0 (set (coeffs p))*
by (*metis nth-default-coeffs-eq range-nth-default*)

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

lemma *coeffs-eqI*:
assumes *coeff*: $\bigwedge n$. *coeff p n = nth-default 0 xs n*
assumes *zero*: *no-trailing (HOL.eq 0) xs*
shows *coeffs p = xs*
proof −
from *coeff* **have** *p = Poly xs*
by (*simp add*: *poly-eq-iff*)
with *zero* **show** *?thesis* **by** *simp*
qed

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

lemma *length-coeffs-degree*: $p \neq 0 \Longrightarrow$ *length (coeffs p) = Suc (degree p)*
by (*induct p*) (*auto simp*: *cCons-def*)

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

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

lemma *set-coeffs-subset-singleton-0-iff* [*simp*]:
set (coeffs p) ⊆ {0} $\longleftrightarrow$ *p = 0*
by (*auto simp add*: *coeffs-def intro*: *classical*)

lemma *set-coeffs-not-only-0* [*simp*]:
set (coeffs p) ≠ {0}
by (*auto simp add*: *set-eq-subset*)

lemma *forall-coeffs-conv*:
$(\forall n.\ P\ (coeff\ p\ n)) \longleftrightarrow (\forall c \in set\ (coeffs\ p).\ P\ c)$ **if** *P 0*
using *that* **by** (*auto simp add*: *coeffs-def*)
(*metis atLeastLessThan-iff coeff-eq-0 not-less-iff-gr-or-eq zero-le*)

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

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

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

end

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

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

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

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*
by *simp*

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*
by (*simp add*: *fold-coeffs-def*)

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

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

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

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

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*)›
by (*induction p*) (*auto simp add*: *fun-eq-iff poly-def*)

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

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

lemma *poly-altdef*: *poly p x* = (∑ *i*≤*degree p. coeff p i * x ^ i*)
for *x* :: *'a*::{*comm-semiring-0*,*semiring-1*}
proof (*induction p rule*: *pCons-induct*)
case *0*
then show *?case*
by *simp*
next
case (*pCons a p*)
show *?case*
proof (*cases p = 0*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
let *?p' = pCons a p*
note *poly-pCons*[*of a p x*]
also note *pCons.IH*
also have *a + x * (*∑ *i*≤*degree p. coeff p i * x ^ i*) =
*coeff ?p' 0 * x^0 +* (∑ *i*≤*degree p. coeff ?p' (Suc i) * x^Suc i*)
by (*simp add*: *field-simps sum-distrib-left coeff-pCons*)
also note *sum.atMost-Suc-shift*[*symmetric*]
also note *degree-pCons-eq*[*OF* ‹*p ≠ 0*›, *of a*, *symmetric*]
finally show *?thesis* **.**
qed
qed

lemma *poly-as-sum*:
fixes *p* :: *'a*::*comm-semiring-1 poly*
shows *poly p x* = (∑ *i*≤*degree p. x ^ i * coeff p i*)
unfolding *poly-altdef* **by** (*simp add*: *ac-simps*)

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

lemma *poly-zero*:
fixes *p* :: *'a* :: *comm-ring-1 poly*
assumes *x*: *poly p x = 0* **shows** *p = 0* ⟷ *degree p = 0*

proof
assume *degp*: *degree p = 0*
hence *poly p x = coeff p (degree p)* **by**(*subst degree-0-id*[*OF degp*,*symmetric*], *simp*)
hence *coeff p (degree p) = 0* **using** *x* **by** *auto*
thus *p = 0* **by** *auto*
qed *auto*

4.11 Monomials

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

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

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

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

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

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

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

lemma *degree-monom-le*: *degree (monom a n) ≤ n*
by (*rule degree-le*, *simp*)

lemma *degree-monom-eq*: *a ≠ 0 ⟹ degree (monom a n) = n*
by (*metis coeff-monom leading-coeff-0-iff*)

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

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

lemma *poly-monom*: *poly (monom a n) x = a * x ^ n*
for *a x* :: *'a*::*comm-semiring-1*
by (*cases a = 0*, *simp-all*) (*induct n*, *simp-all add*: *mult.left-commute poly-eq-fold-coeffs*)

lemma *monom-eq-iff'*: *monom c n = monom d m* $\longleftrightarrow$ $c = d \wedge (c = 0 \vee n = m)$
by (*auto simp*: *poly-eq-iff*)

lemma *monom-eq-const-iff*: *monom c n* = [:*d*:] $\longleftrightarrow$ $c = d \wedge (c = 0 \vee n = 0)$
using *monom-eq-iff'*[*of c n d 0*] **by** (*simp add*: *monom-0*)

4.12 Leading coefficient

abbreviation *lead-coeff*:: *'a*::*zero poly* $\Rightarrow$ *'a*
where *lead-coeff p* $\equiv$ *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*
by *auto*

lemma *lead-coeff-monom* [*simp*]: *lead-coeff* (*monom c n*) = *c*
by (*cases c = 0*) (*simp-all add*: *degree-monom-eq*)

lemma *last-coeffs-eq-coeff-degree*:
last (*coeffs p*) = *lead-coeff p* **if** $p \neq 0$
using *that* **by** (*simp add*: *coeffs-def*)

lemma *lead-coeff-list-def*:
lead-coeff p = (*if coeffs p*=[] *then 0 else last* (*coeffs p*))
by (*simp add*: *last-coeffs-eq-coeff-degree*)

4.13 Addition and subtraction

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

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

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

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

```
  show p + q = q + p
    by (simp add: poly-eq-iff add.commute)
  show 0 + p = p
    by (simp add: poly-eq-iff)
qed

end

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

lift-definition minus-poly :: 'a poly ⇒ 'a poly ⇒ 'a poly
  is λp q n. coeff p n − coeff q n
proof −
  fix q p :: 'a poly
  show ∀∞n. coeff p n − coeff q n = 0
    using MOST-coeff-eq-0[of p] MOST-coeff-eq-0[of q] by eventually-elim simp
qed

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

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

end

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

lift-definition uminus-poly :: 'a poly ⇒ 'a poly
  is λp n. − coeff p n
proof −
  fix p :: 'a poly
  show ∀∞n. − coeff p n = 0
    using MOST-coeff-eq-0 by simp
qed

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

instance
proof
```

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

end

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

lemma minus-pCons [simp]: - pCons a p = pCons (- a) (- p)
  by (rule poly-eqI) (simp add: coeff-pCons split: nat.split)

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

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

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

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

lemma degree-add-eq-right: assumes degree p < degree q shows degree (p + q)
= degree q
proof (cases q = 0)
  case False
  show ?thesis
  proof (rule order-antisym)
    show degree (p + q) ≤ degree q
      by (simp add: assms degree-add-le order.strict-implies-order)
    show degree q ≤ degree (p + q)
      by (simp add: False assms coeff-eq-0 le-degree)
  qed
qed (use assms in auto)

lemma degree-add-eq-left: degree q < degree p ⟹ degree (p + q) = degree p
  using degree-add-eq-right [of q p] by (simp add: add.commute)

lemma degree-minus [simp]: degree (- p) = degree p
  by (simp add: degree-def)

lemma lead-coeff-add-le: degree p < degree q ⟹ lead-coeff (p + q) = lead-coeff q
 by (metis coeff-add coeff-eq-0 monoid-add-class.add.left-neutral degree-add-eq-right)
```

lemma *lead-coeff-minus*: *lead-coeff* (− *p*) = − *lead-coeff p*
by (*metis coeff-minus degree-minus*)

lemma *degree-diff-le-max*: *degree* (*p* − *q*) ≤ *max* (*degree p*) (*degree q*)
for *p q* :: *'a*::*ab-group-add poly*
using *degree-add-le* [**where** *p*=*p* **and** *q*=−*q*] **by** *simp*

lemma *degree-diff-le*: *degree p* ≤ *n* ⟹ *degree q* ≤ *n* ⟹ *degree* (*p* − *q*) ≤ *n*
for *p q* :: *'a*::*ab-group-add poly*
using *degree-add-le* [*of p n* − *q*] **by** *simp*

lemma *degree-diff-less*: *degree p* < *n* ⟹ *degree q* < *n* ⟹ *degree* (*p* − *q*) < *n*
for *p q* :: *'a*::*ab-group-add poly*
using *degree-add-less* [*of p n* − *q*] **by** *simp*

lemma *add-monom*: *monom a n* + *monom b n* = *monom* (*a* + *b*) *n*
by (*rule poly-eqI*) *simp*

lemma *diff-monom*: *monom a n* − *monom b n* = *monom* (*a* − *b*) *n*
by (*rule poly-eqI*) *simp*

lemma *minus-monom*: − *monom a n* = *monom* (− *a*) *n*
by (*rule poly-eqI*) *simp*

lemma *coeff-sum*: *coeff* (∑ *x*∈*A*. *p x*) *i* = (∑ *x*∈*A*. *coeff* (*p x*) *i*)
by (*induct A rule*: *infinite-finite-induct*) *simp-all*

lemma *monom-sum*: *monom* (∑ *x*∈*A*. *a x*) *n* = (∑ *x*∈*A*. *monom* (*a x*) *n*)
by (*rule poly-eqI*) (*simp add*: *coeff-sum*)

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 −
have ∗: *nth-default 0* (*plus-coeffs xs ys*) *n* = *nth-default 0 xs n* + *nth-default 0 ys n*
for *xs ys* :: *'a list* **and** *n*
proof (*induct xs ys arbitrary*: *n rule*: *plus-coeffs.induct*)
case (*3 x xs y ys n*)
then show *?case*
by (*cases n*) (*auto simp add*: *cCons-def*)
qed *simp-all*
have ∗∗: *no-trailing* (*HOL.eq 0*) (*plus-coeffs xs ys*)
if *no-trailing* (*HOL.eq 0*) *xs* **and** *no-trailing* (*HOL.eq 0*) *ys*

for *xs ys* :: *'a list*
using *that* **by** (*induct xs ys rule*: *plus-coeffs.induct*) (*simp-all add*: *cCons-def*)
show *?thesis*
by (*rule coeffs-eqI*) (*auto simp add*: ∗ *nth-default-coeffs-eq intro*: ∗∗)
qed

lemma *coeffs-uminus* [*code abstract*]:
coeffs (− *p*) = *map uminus* (*coeffs p*)
proof −
have *eq-0*: *HOL.eq 0* ∘ *uminus* = *HOL.eq* (*0*::*'a*)
by (*simp add*: *fun-eq-iff*)
show *?thesis*
by (*rule coeffs-eqI*) (*simp-all add*: *nth-default-map-eq nth-default-coeffs-eq no-trailing-map eq-0*)
qed

lemma [*code*]: *p* − *q* = *p* + − *q*
for *p q* :: *'a*::*ab-group-add poly*
by (*fact diff-conv-add-uminus*)

lemma *poly-add* [*simp*]: *poly* (*p* + *q*) *x* = *poly p x* + *poly q x*
proof (*induction p arbitrary*: *q*)
case (*pCons a p*)
then show *?case*
by (*cases q*) (*simp add*: *algebra-simps*)
qed *auto*

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

lemma *poly-diff* [*simp*]: *poly* (*p* − *q*) *x* = *poly p x* − *poly q x*
for *x* :: *'a*::*comm-ring*
using *poly-add* [*of p* − *q x*] **by** *simp*

lemma *poly-sum*: *poly* (∑ *k*∈*A*. *p k*) *x* = (∑ *k*∈*A*. *poly* (*p k*) *x*)
by (*induct A rule*: *infinite-finite-induct*) *simp-all*

lemma *poly-sum-list*: *poly* (∑ *p*←*ps*. *p*) *y* = (∑ *p*←*ps*. *poly p y*)
by (*induction ps*) *auto*

lemma *poly-sum-mset*: *poly* (∑ *x*∈#*A*. *p x*) *y* = (∑ *x*∈#*A*. *poly* (*p x*) *y*)
by (*induction A*) *auto*

lemma *degree-sum-le*: *finite S* ⟹ (⋀*p*. *p* ∈ *S* ⟹ *degree* (*f p*) ≤ *n*) ⟹ *degree* (*sum f S*) ≤ *n*
proof (*induct S rule*: *finite-induct*)
case *empty*
then show *?case* **by** *simp*

next
case (*insert p S*)
then have *degree* (*sum f S*) ≤ *n degree* (*f p*) ≤ *n*
by *auto*
then show *?case*
unfolding *sum.insert*[*OF insert*(*1*−*2*)] **by** (*metis degree-add-le*)
qed

lemma *degree-sum-less*:
assumes ⋀*x*. *x* ∈ *A* ⟹ *degree* (*f x*) < *n n* > *0*
shows *degree* (*sum f A*) < *n*
using *assms* **by** (*induction rule*: *infinite-finite-induct*) (*auto intro*!: *degree-add-less*)

lemma *poly-as-sum-of-monoms′*:
assumes *degree p* ≤ *n*
shows (∑ *i*≤*n*. *monom* (*coeff p i*) *i*) = *p*
proof −
have *eq*: ⋀*i*. {..*n*} ∩ {*i*} = (*if i* ≤ *n then* {*i*} *else* {})
by *auto*
from *assms* **show** *?thesis*
by (*simp add*: *poly-eq-iff coeff-sum coeff-eq-0 sum.If-cases eq*
if-distrib[**where** *f*=λ*x*. *x* ∗ *a* **for** *a*])
qed

lemma *poly-as-sum-of-monoms*: (∑ *i*≤*degree p*. *monom* (*coeff p i*) *i*) = *p*
by (*intro poly-as-sum-of-monoms′ order-refl*)

lemma *Poly-snoc*: *Poly* (*xs* @ [*x*]) = *Poly xs* + *monom x* (*length xs*)
by (*induct xs*) (*simp-all add*: *monom-0 monom-Suc*)

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

lift-definition *smult* :: *′a*::*comm-semiring-0* ⇒ *′a poly* ⇒ *′a poly*
is λ*a p n*. *a* ∗ *coeff p n*
proof −
fix *a* :: *′a* **and** *p* :: *′a poly*
show ∀ ∞ *i*. *a* ∗ *coeff p i* = *0*
using *MOST-coeff-eq-0*[*of p*] **by** *eventually-elim simp*
qed

lemma *coeff-smult* [*simp*]: *coeff* (*smult a p*) *n* = *a* ∗ *coeff p n*
by (*simp add*: *smult.rep-eq*)

lemma *degree-smult-le*: *degree* (*smult a p*) ≤ *degree p*
by (*rule degree-le*) (*simp add*: *coeff-eq-0*)

lemma *smult-smult* [*simp*]: *smult a* (*smult b p*) = *smult* (*a* ∗ *b*) *p*
by (*rule poly-eqI*) (*simp add*: *mult.assoc*)

lemma *smult-0-right* [*simp*]: *smult a 0 = 0*
by (*rule poly-eqI*) *simp*

lemma *smult-0-left* [*simp*]: *smult 0 p = 0*
by (*rule poly-eqI*) *simp*

lemma *smult-1-left* [*simp*]: *smult (1::'a::comm-semiring-1) p = p*
by (*rule poly-eqI*) *simp*

lemma *smult-add-right*: *smult a (p + q) = smult a p + smult a q*
by (*rule poly-eqI*) (*simp add*: *algebra-simps*)

lemma *smult-add-left*: *smult (a + b) p = smult a p + smult b p*
by (*rule poly-eqI*) (*simp add*: *algebra-simps*)

lemma *smult-minus-right* [*simp*]: *smult a (− p) = − smult a p*
for *a* :: *'a::comm-ring*
by (*rule poly-eqI*) *simp*

lemma *smult-minus-left* [*simp*]: *smult (− a) p = − smult a p*
for *a* :: *'a::comm-ring*
by (*rule poly-eqI*) *simp*

lemma *smult-diff-right*: *smult a (p − q) = smult a p − smult a q*
for *a* :: *'a::comm-ring*
by (*rule poly-eqI*) (*simp add*: *algebra-simps*)

lemma *smult-diff-left*: *smult (a − b) p = smult a p − smult b p*
for *a b* :: *'a::comm-ring*
by (*rule poly-eqI*) (*simp add*: *algebra-simps*)

lemmas *smult-distribs* =
smult-add-left smult-add-right
smult-diff-left smult-diff-right

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

lemma *smult-monom*: *smult a (monom b n) = monom (a * b) n*
by (*induct n*) (*simp-all add*: *monom-0 monom-Suc*)

lemma *smult-Poly*: *smult c (Poly xs) = Poly (map ((*) c) xs)*
by (*auto simp*: *poly-eq-iff nth-default-def*)

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}*
by (*cases a = 0*) (*simp-all add*: *degree-def*)

lemma *smult-eq-0-iff* [*simp*]: *smult a p = 0 ⟷ a = 0 ∨ p = 0*
for *a* :: *'a*::{*comm-semiring-0*,*semiring-no-zero-divisors*}
by (*simp add*: *poly-eq-iff*)

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 −
have *eq-0*: *HOL.eq 0* ∘ *times a* = *HOL.eq* (*0*::*'a*) **if** *a* ≠ *0*
using *that* **by** (*simp add*: *fun-eq-iff*)
show *?thesis*
by (*rule coeffs-eqI*) (*auto simp add*: *no-trailing-map nth-default-map-eq nth-default-coeffs-eq eq-0*)
qed

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
assume *?lhs*
also from *assms* **have** *smult* (*inverse b*) ... = *q*
by *simp*
finally show *?rhs*
by (*simp add*: *field-simps*)
next
assume *?rhs*
with *assms* **show** *?lhs* **by** *auto*
qed

lemma *smult-cancel*:
fixes *p*::*'a*::*idom poly*
assumes *c≠0* **and** *smult*: *smult c p* = *smult c q*
shows *p*=*q*
proof −
have *smult c* (*p*−*q*) = *0* **using** *smult* **by** (*metis diff-self smult-diff-right*)
thus *?thesis* **using** ‹*c≠0*› **by** *auto*
qed

instantiation *poly* :: (*comm-semiring-0*) *comm-semiring-0*
begin

definition *p* ∗ *q* = *fold-coeffs* (λ*a p*. *smult a q* + *pCons 0 p*) *p 0*

lemma *mult-poly-0-left*: (*0*::*'a poly*) ∗ *q* = *0*
by (*simp add*: *times-poly-def*)

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

by (*cases* $p = 0 \wedge a = 0$) (*auto simp add*: *times-poly-def*)

lemma *mult-poly-0-right*: $p * (0::'a\ poly) = 0$
by (*induct p*) (*simp-all add*: *mult-poly-0-left*)

lemma *mult-pCons-right* [*simp*]: $p * pCons\ a\ q = smult\ a\ p + pCons\ 0\ (p * q)$
by (*induct p*) (*simp-all add*: *mult-poly-0-left algebra-simps*)

lemmas *mult-poly-0 = mult-poly-0-left mult-poly-0-right*

lemma *mult-smult-left* [*simp*]: $smult\ a\ p * q = smult\ a\ (p * q)$
by (*induct p*) (*simp-all add*: *mult-poly-0 smult-add-right*)

lemma *mult-smult-right* [*simp*]: $p * smult\ a\ q = smult\ a\ (p * q)$
by (*induct q*) (*simp-all add*: *mult-poly-0 smult-add-right*)

lemma *mult-poly-add-left*: $(p + q) * r = p * r + q * r$
for $p\ q\ r :: 'a\ poly$
by (*induct r*) (*simp-all add*: *mult-poly-0 smult-distribs algebra-simps*)

instance
proof
fix $p\ q\ r :: 'a\ poly$
show *0*: $0 * p = 0$
by (*rule mult-poly-0-left*)
show $p * 0 = 0$
by (*rule mult-poly-0-right*)
show $(p + q) * r = p * r + q * r$
by (*rule mult-poly-add-left*)
show $(p * q) * r = p * (q * r)$
by (*induct p*) (*simp-all add*: *mult-poly-0 mult-poly-add-left*)
show $p * q = q * p$
by (*induct p*) (*simp-all add*: *mult-poly-0*)
qed

end

lemma *coeff-mult-degree-sum*:
coeff $(p * q)$ $(degree\ p + degree\ q) = coeff\ p\ (degree\ p) * coeff\ q\ (degree\ q)$
by (*induct p*) (*simp-all add*: *coeff-eq-0*)

instance *poly* :: ({*comm-semiring-0*,*semiring-no-zero-divisors*}) *semiring-no-zero-divisors*
proof
fix $p\ q :: 'a\ poly$
assume $p \neq 0$ **and** $q \neq 0$
have *coeff* $(p * q)$ $(degree\ p + degree\ q) = coeff\ p\ (degree\ p) * coeff\ q\ (degree\ q)$
by (*rule coeff-mult-degree-sum*)
also from ‹$p \neq 0$› ‹$q \neq 0$› **have** *coeff* $p\ (degree\ p) * coeff\ q\ (degree\ q) \neq 0$
by *simp*

finally have $\exists n.\ coeff\ (p * q)\ n \neq 0$ **..**
then show $p * q \neq 0$
by (*simp add*: *poly-eq-iff*)
qed

instance *poly* :: (*comm-semiring-0-cancel*) *comm-semiring-0-cancel* **..**

lemma *coeff-mult*: $coeff\ (p * q)\ n = (\sum i{\leq}n.\ coeff\ p\ i * coeff\ q\ (n{-}i))$
proof (*induct p arbitrary*: *n*)
case *0*
show *?case* **by** *simp*
next
case (*pCons a p n*)
then show *?case*
by (*cases n*) (*simp-all add*: *sum.atMost-Suc-shift del*: *sum.atMost-Suc*)
qed

lemma *coeff-mult-0*: $coeff\ (p * q)\ 0 = coeff\ p\ 0 * coeff\ q\ 0$
by (*simp add*: *coeff-mult*)

lemma *degree-mult-le*: $degree\ (p * q) \leq degree\ p + degree\ q$
proof (*rule degree-le*)
show $\forall i{>}degree\ p + degree\ q.\ coeff\ (p * q)\ i = 0$
by (*induct p*) (*simp-all add*: *coeff-eq-0 coeff-pCons split*: *nat.split*)
qed

lemma *mult-monom*: $monom\ a\ m * monom\ b\ n = monom\ (a * b)\ (m + n)$
by (*induct m*) (*simp add*: *monom-0 smult-monom*, *simp add*: *monom-Suc*)

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

lift-definition *one-poly* :: *'a poly*
is $\lambda n.\ of\text{-}bool\ (n = 0)$
by (*rule MOST-SucD*) *simp*

lemma *coeff-1* [*simp*]:
$coeff\ 1\ n = of\text{-}bool\ (n = 0)$
by (*simp add*: *one-poly.rep-eq*)

lemma *one-pCons*:
$1 = [:1:]$
by (*simp add*: *poly-eq-iff coeff-pCons split*: *nat.splits*)

lemma *pCons-one*:
$[:1:] = 1$
by (*simp add*: *one-pCons*)

instance

by *standard* (*simp-all add*: *one-pCons*)

end

lemma *poly-1* [*simp*]:
poly 1 x = 1
by (*simp add*: *one-pCons*)

lemma *one-poly-eq-simps* [*simp*]:
1 = [:1:] $\longleftrightarrow$ *True*
[:1:] = 1 $\longleftrightarrow$ *True*
by (*simp-all add*: *one-pCons*)

lemma *degree-1* [*simp*]:
degree 1 = 0
by (*simp add*: *one-pCons*)

lemma *coeffs-1-eq* [*simp*, *code abstract*]:
coeffs 1 = [1]
by (*simp add*: *one-pCons*)

lemma *smult-one* [*simp*]:
smult c 1 = [:c:]
by (*simp add*: *one-pCons*)

lemma *smult-sum*: *smult* $(\sum i \in S.\ f\ i)\ p = (\sum i \in S.\ smult\ (f\ i)\ p)$
by (*induct S rule*: *infinite-finite-induct*, *auto simp*: *smult-add-left*)

lemma *smult-power*: *(smult a p)* $\hat{}\ n = smult\ (a\ \hat{}\ n)\ (p\ \hat{}\ n)$
by (*induct n*, *auto simp*: *field-simps*)

lemma *monom-eq-1* [*simp*]:
monom 1 0 = 1
by (*simp add*: *monom-0 one-pCons*)

lemma *monom-eq-1-iff*:
monom c n = 1 $\longleftrightarrow$ *c = 1* $\wedge$ *n = 0*
using *monom-eq-const-iff* [*of c n 1*] **by** *auto*

lemma *monom-altdef*:
monom c n = smult c $([:0,\ 1:]\ \hat{}\ n)$
by (*induct n*) (*simp-all add*: *monom-0 monom-Suc*)

lemma *degree-sum-list-le*: $(\bigwedge p\ .\ p \in set\ ps \Longrightarrow degree\ p \leq n)$
$\Longrightarrow$ *degree* $(sum\text{-}list\ ps) \leq n$
proof (*induct ps*)
case (*Cons p ps*)
hence *degree* $(sum\text{-}list\ ps) \leq n$ *degree* $p \leq n$ **by** *auto*
thus *?case* **unfolding** *sum-list.Cons* **by** (*metis degree-add-le*)

qed *simp*

lemma *degree-prod-list-le*: *degree* (*prod-list ps*) ≤ *sum-list* (*map degree ps*)
proof (*induct ps*)
  case (*Cons p ps*)
  show *?case* **unfolding** *prod-list.Cons*
    by (*rule order.trans*[*OF degree-mult-le*], *insert Cons*, *auto*)
qed *simp*

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

lemma *prod-smult*: (∏ *x*∈*A*. *smult* (*c x*) (*p x*)) = *smult* (*prod c A*) (*prod p A*)
  by (*induction A rule*: *infinite-finite-induct*) (*auto simp*: *mult-ac*)

lemma *degree-power-le*: *degree* (*p* ^ *n*) ≤ *degree p* ∗ *n*
  by (*induct n*) (*auto intro*: *order-trans degree-mult-le*)

lemma *coeff-0-power*: *coeff* (*p* ^ *n*) *0* = *coeff p 0* ^ *n*
  by (*induct n*) (*simp-all add*: *coeff-mult*)

lemma *poly-smult* [*simp*]: *poly* (*smult a p*) *x* = *a* ∗ *poly p x*
  by (*induct p*) (*simp-all add*: *algebra-simps*)

lemma *poly-mult* [*simp*]: *poly* (*p* ∗ *q*) *x* = *poly p x* ∗ *poly q x*
  by (*induct p*) (*simp-all add*: *algebra-simps*)

lemma *poly-power* [*simp*]: *poly* (*p* ^ *n*) *x* = *poly p x* ^ *n*
  for *p* :: *'a*::*comm-semiring-1 poly*
  by (*induct n*) *simp-all*

lemma *poly-prod*: *poly* (∏ *k*∈*A*. *p k*) *x* = (∏ *k*∈*A*. *poly* (*p k*) *x*)
  by (*induct A rule*: *infinite-finite-induct*) *simp-all*

lemma *poly-prod-list*: *poly* (∏ *p*←*ps*. *p*) *y* = (∏ *p*←*ps*. *poly p y*)
  by (*induction ps*) *auto*

lemma *poly-prod-mset*: *poly* (∏ *x*∈#*A*. *p x*) *y* = (∏ *x*∈#*A*. *poly* (*p x*) *y*)
  by (*induction A*) *auto*

lemma *poly-const-pow*: [: *c* :] ^ *n* = [: *c* ^ *n* :]
  by (*induction n*) (*auto simp*: *algebra-simps*)

lemma *monom-power*: *monom c n* ^ *k* = *monom* (*c* ^ *k*) (*n* ∗ *k*)
  by (*induction k*) (*auto simp*: *mult-monom*)

lemma *degree-prod-sum-le*: *finite S* $\Longrightarrow$ *degree* (*prod f S*) $\leq$ *sum* (*degree* $\circ$ *f*) *S*
proof (*induct S rule*: *finite-induct*)
  case *empty*
  then show *?case* **by** *simp*
next
  case (*insert a S*)
  show *?case*
    unfolding *prod.insert*[*OF insert*(*1*−*2*)] *sum.insert*[*OF insert*(*1*−*2*)]
    by (*rule le-trans*[*OF degree-mult-le*]) (*use insert* **in** *auto*)
qed

lemma *coeff-0-prod-list*: *coeff* (*prod-list xs*) *0* = *prod-list* (*map* (λ*p*. *coeff p 0*) *xs*)
  by (*induct xs*) (*simp-all add*: *coeff-mult*)

lemma *coeff-monom-mult*: *coeff* (*monom c n* ∗ *p*) *k* = (*if k* < *n then 0 else c* ∗ *coeff p* (*k* − *n*))
proof −
  have *coeff* (*monom c n* ∗ *p*) *k* = (∑ *i*≤*k*. (*if n* = *i then c else 0*) ∗ *coeff p* (*k* − *i*))
    by (*simp add*: *coeff-mult*)
  also have ... = (∑ *i*≤*k*. (*if n* = *i then c* ∗ *coeff p* (*k* − *i*) *else 0*))
    by (*intro sum.cong*) *simp-all*
  also have ... = (*if k* < *n then 0 else c* ∗ *coeff p* (*k* − *n*))
    by *simp*
  finally show *?thesis* **.**
qed

lemma *coeff-monom-Suc*: *coeff* (*monom a* (*Suc d*) ∗ *p*) (*Suc i*) = *coeff* (*monom a d* ∗ *p*) *i*
  by (*simp add*: *monom-Suc*)

lemma *monom-1-dvd-iff'*: *monom 1 n dvd p* ⟷ (∀ *k*<*n*. *coeff p k* = *0*)
proof
  assume *monom 1 n dvd p*
  then obtain *r* **where** *p* = *monom 1 n* ∗ *r*
    by (*rule dvdE*)
  then show ∀ *k*<*n*. *coeff p k* = *0*
    by (*simp add*: *coeff-mult*)
next
  assume *zero*: (∀ *k*<*n*. *coeff p k* = *0*)
  define *r* **where** *r* = *Abs-poly* (λ*k*. *coeff p* (*k* + *n*))
  have ∀ ∞*k*. *coeff p* (*k* + *n*) = *0*
    by (*subst cofinite-eq-sequentially*, *subst eventually-sequentially-seg*,
        subst cofinite-eq-sequentially [*symmetric*]) *transfer*
  then have *coeff-r* [*simp*]: *coeff r k* = *coeff p* (*k* + *n*) **for** *k*
    unfolding *r-def* **by** (*subst poly.Abs-poly-inverse*) *simp-all*
  have *p* = *monom 1 n* ∗ *r*
    by (*rule poly-eqI*, *subst coeff-monom-mult*) (*simp-all add*: *zero*)
  then show *monom 1 n dvd p* **by** *simp*

qed

lemma *coeff-sum-monom*:
assumes *n*: $n \leq d$
shows *coeff* $(\sum i{\leq}d.\ monom\ (f\ i)\ i)\ n = f\ n$ (**is** *?l* = -)
proof −
have *?l* = $(\sum i{\leq}d.\ coeff\ (monom\ (f\ i)\ i)\ n)$ (**is** - = *sum ?cmf* -)
using *coeff-sum*.
also have $\{..d\} = insert\ n\ (\{..d\}-\{n\})$ **using** *n* **by** *auto*
hence *sum ?cmf* $\{..d\}$ = *sum ?cmf* ... **by** *auto*
also have ... = *sum ?cmf* $(\{..d\}-\{n\})$ + *?cmf n* **by** (*subst sum.insert*,*auto*)
also have *sum ?cmf* $(\{..d\}-\{n\}) = 0$ **by** (*subst sum.neutral*, *auto*)
finally show *?thesis* **by** *simp*
qed

4.15 Mapping polynomials

definition *map-poly* :: $('a :: zero \Rightarrow 'b :: zero) \Rightarrow 'a\ poly \Rightarrow 'b\ poly$
where *map-poly f p* = *Poly* (*map f* (*coeffs p*))

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

lemma *map-poly-1*: *map-poly f 1* = [:*f 1*:]
by (*simp add*: *map-poly-def*)

lemma *map-poly-1'* [*simp*]: *f 1* = *1* $\Longrightarrow$ *map-poly f 1* = *1*
by (*simp add*: *map-poly-def one-pCons*)

lemma *coeff-map-poly*:
assumes *f 0* = *0*
shows *coeff* (*map-poly f p*) *n* = *f* (*coeff p n*)
by (*auto simp*: *assms map-poly-def nth-default-def coeffs-def not-less Suc-le-eq coeff-eq-0*
simp del: *upt-Suc*)

lemma *lead-coeff-map-poly-nz*:
assumes *f* (*lead-coeff p*) ≠ *0 f 0* = *0*
shows *lead-coeff* (*map-poly f p*) = *f* (*lead-coeff p*)
by (*metis* (*no-types*, *lifting*) *antisym assms coeff-0 coeff-map-poly le-degree leading-coeff-0-iff*)

lemma *coeffs-map-poly* [*code abstract*]:
coeffs (*map-poly f p*) = *strip-while* ((=) *0*) (*map f* (*coeffs p*))
by (*simp add*: *map-poly-def*)

lemma *coeffs-map-poly'*:
assumes $\bigwedge x.\ x \neq 0 \Longrightarrow f\ x \neq 0$
shows *coeffs* (*map-poly f p*) = *map f* (*coeffs p*)

using *assms*
by (*auto simp add*: *coeffs-map-poly strip-while-idem-iff*
last-coeffs-eq-coeff-degree no-trailing-unfold last-map)

lemma *set-coeffs-map-poly*:
$(\bigwedge x.\ f\ x = 0 \longleftrightarrow x = 0) \Longrightarrow set\ (coeffs\ (map\text{-}poly\ f\ p)) = f\ `\ set\ (coeffs\ p)$
by (*simp add*: *coeffs-map-poly'*)

lemma *degree-map-poly*:
assumes $\bigwedge x.\ x \neq 0 \Longrightarrow f\ x \neq 0$
shows $degree\ (map\text{-}poly\ f\ p) = degree\ p$
by (*simp add*: *degree-eq-length-coeffs coeffs-map-poly' assms*)

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 −
have $(coeff\ (map\text{-}poly\ f\ p)\ n = 0) = (coeff\ p\ n = 0)$ **for** *n*
proof −
have $coeff\ (map\text{-}poly\ f\ p)\ n = f\ (coeff\ p\ n)$
by (*simp add*: *coeff-map-poly assms*)
also have $\ldots = 0 \longleftrightarrow coeff\ p\ n = 0$
proof $(cases\ n < length\ (coeffs\ p))$
case *True*
then have $coeff\ p\ n \in set\ (coeffs\ p)$
by (*auto simp*: *coeffs-def simp del*: *upt-Suc*)
with *assms* **show** $f\ (coeff\ p\ n) = 0 \longleftrightarrow coeff\ p\ n = 0$
by *auto*
next
case *False*
then show *?thesis*
by (*auto simp*: *assms length-coeffs nth-default-coeffs-eq* [*symmetric*] *nth-default-def*)
qed
finally show *?thesis* **.**
qed
then show *?thesis* **by** (*auto simp*: *poly-eq-iff*)
qed

lemma *map-poly-smult*:
assumes $f\ 0 = 0 \bigwedge c\ x.\ f\ (c * x) = f\ c * f\ x$
shows $map\text{-}poly\ f\ (smult\ c\ p) = smult\ (f\ c)\ (map\text{-}poly\ f\ p)$
by (*intro poly-eqI*) (*simp-all add*: *assms coeff-map-poly*)

lemma *map-poly-pCons*:
assumes $f\ 0 = 0$
shows $map\text{-}poly\ f\ (pCons\ c\ p) = pCons\ (f\ c)\ (map\text{-}poly\ f\ p)$
by (*intro poly-eqI*) (*simp-all add*: *assms coeff-map-poly coeff-pCons split*: *nat.splits*)

lemma *map-poly-map-poly*:

assumes $f\ 0 = 0\ g\ 0 = 0$
shows *map-poly f* (*map-poly g p*) = *map-poly* (*f* ∘ *g*) *p*
by (*intro poly-eqI*) (*simp add*: *coeff-map-poly assms*)

lemma *map-poly-id* [*simp*]: *map-poly id p* = *p*
by (*simp add*: *map-poly-def*)

lemma *map-poly-id′* [*simp*]: *map-poly* (λ*x*. *x*) *p* = *p*
by (*simp add*: *map-poly-def*)

lemma *map-poly-cong*:
assumes (⋀*x*. *x* ∈ *set* (*coeffs p*) ⟹ *f x* = *g x*)
shows *map-poly f p* = *map-poly g p*
proof −
from *assms* **have** *map f* (*coeffs p*) = *map g* (*coeffs p*)
by (*intro map-cong*) *simp-all*
then show *?thesis*
by (*simp only*: *coeffs-eq-iff coeffs-map-poly*)
qed

lemma *map-poly-monom*: *f 0* = *0* ⟹ *map-poly f* (*monom c n*) = *monom* (*f c*) *n*
by (*intro poly-eqI*) (*simp-all add*: *coeff-map-poly*)

lemma *map-poly-idI*:
assumes ⋀*x*. *x* ∈ *set* (*coeffs p*) ⟹ *f x* = *x*
shows *map-poly f p* = *p*
using *map-poly-cong*[*OF assms*, *of - id*] **by** *simp*

lemma *map-poly-idI′*:
assumes ⋀*x*. *x* ∈ *set* (*coeffs p*) ⟹ *f x* = *x*
shows *p* = *map-poly f p*
using *map-poly-cong*[*OF assms*, *of - id*] **by** *simp*

lemma *smult-conv-map-poly*: *smult c p* = *map-poly* (λ*x*. *c* ∗ *x*) *p*
by (*intro poly-eqI*) (*simp-all add*: *coeff-map-poly*)

lemma *poly-cnj*: *cnj* (*poly p z*) = *poly* (*map-poly cnj p*) (*cnj z*)
by (*simp add*: *poly-altdef degree-map-poly coeff-map-poly*)

lemma *monom-pCons-0-monom*:
monom (*pCons 0* (*monom a n*)) *d* = *map-poly* (*pCons 0*) (*monom* (*monom a n*) *d*)
by (*simp add*: *map-poly-monom*)

lemma *pCons-0-add*: *pCons 0* (*p* + *q*) = *pCons 0 p* + *pCons 0 q*
by *auto*

lemma *sum-pCons-0-commute*:
sum (λ*i*. *pCons 0* (*f i*)) *S* = *pCons 0* (*sum f S*)

by(*induct S rule*: *infinite-finite-induct*; *simp*)

lemma *pCons-0-as-mult*:
fixes *p*:: *'a* :: *comm-semiring-1 poly*
shows *pCons 0 p* = [:*0*,*1*:] ∗ *p*
by *auto*

lemma *poly-cnj-real*:
assumes ⋀*n*. *poly.coeff p n* ∈ ℝ
shows *cnj* (*poly p z*) = *poly p* (*cnj z*)
proof −
from *assms* **have** *map-poly cnj p* = *p*
by (*intro poly-eqI*) (*auto simp*: *coeff-map-poly Reals-cnj-iff*)
with *poly-cnj*[*of p z*] **show** *?thesis* **by** *simp*
qed

lemma *real-poly-cnj-root-iff*:
assumes ⋀*n*. *poly.coeff p n* ∈ ℝ
shows *poly p* (*cnj z*) = *0* ⟷ *poly p z* = *0*
proof −
have *poly p* (*cnj z*) = *cnj* (*poly p z*)
by (*simp add*: *poly-cnj-real assms*)
also have ... = *0* ⟷ *poly p z* = *0* **by** *simp*
finally show *?thesis* **.**
qed

lemma *sum-to-poly*: (∑ *x*∈*A*. [:*f x*:]) = [:∑ *x*∈*A*. *f x*:]
by (*induction A rule*: *infinite-finite-induct*) *auto*

lemma *diff-to-poly*: [:*c*:] − [:*d*:] = [:*c* − *d*:]
by (*simp add*: *poly-eq-iff mult-ac*)

lemma *mult-to-poly*: [:*c*:] ∗ [:*d*:] = [:*c* ∗ *d*:]
by (*simp add*: *poly-eq-iff mult-ac*)

lemma *prod-to-poly*: (∏ *x*∈*A*. [:*f x*:]) = [:∏ *x*∈*A*. *f x*:]
by (*induction A rule*: *infinite-finite-induct*) (*auto simp*: *mult-to-poly mult-ac*)

lemma *poly-map-poly-cnj* [*simp*]: *poly* (*map-poly cnj p*) *x* = *cnj* (*poly p* (*cnj x*))
using *complex-cnj-cnj poly-cnj* **by** *force*

lemma *map-poly-degree-eq*:
assumes *f* (*lead-coeff p*) ≠ *0*
shows *degree* (*map-poly f p*) = *degree p*
using *assms*
unfolding *map-poly-def degree-eq-length-coeffs coeffs-Poly lead-coeff-list-def*
by (*metis* (*full-types*) *last-conv-nth-default length-map no-trailing-unfold nth-default-coeffs-eq*

nth-default-map-eq strip-while-idem)

lemma *map-poly-degree-less*:
assumes *f* (*lead-coeff p*) =*0 degree p*≠*0*
shows *degree* (*map-poly f p*) < *degree p*
proof −
have *length* (*coeffs p*) >*1*
using ‹*degree p*≠*0*› **by** (*simp add*: *degree-eq-length-coeffs*)
then obtain *xs x* **where** *xs-def*:*coeffs p*=*xs*@[*x*] *length xs*>*0*
by (*metis One-nat-def add-0 append-Nil length-greater-0-conv list.size*(*4*) *nat-neq-iff not-less-zero rev-exhaust*)
have *f x*=*0* **using** *assms*(*1*) **by** (*simp add*: *lead-coeff-list-def xs-def*(*1*))
have *degree* (*map-poly f p*) = *length* (*strip-while* ((=) *0*) (*map f* (*xs*@[*x*]))) − *1*
unfolding *map-poly-def degree-eq-length-coeffs coeffs-Poly*
by (*subst xs-def*,*auto*)
also have ... = *length* (*strip-while* ((=) *0*) (*map f xs*)) − *1*
using ‹*f x*=*0*› **by** *simp*
also have ... ≤ *length xs* −*1*
using *length-strip-while-le* **by** (*metis diff-le-mono length-map*)
also have ... < *length* (*xs*@[*x*]) − *1*
using *xs-def*(*2*) **by** *auto*
also have ... = *degree p*
unfolding *degree-eq-length-coeffs xs-def* **by** *simp*
finally show *?thesis* **.**
qed

lemma *map-poly-degree-leq*:
shows *degree* (*map-poly f p*) ≤ *degree p*
unfolding *map-poly-def degree-eq-length-coeffs*
by (*metis coeffs-Poly diff-le-mono length-map length-strip-while-le*)

4.16 Conversions

lemma *of-nat-poly*: *of-nat n* = [:*of-nat n*:]
by (*induct n*) (*simp-all add*: *one-pCons*)

lemma *of-nat-monom*: *of-nat n* = *monom* (*of-nat n*) *0*
by (*simp add*: *of-nat-poly monom-0*)

lemma *degree-of-nat* [*simp*]: *degree* (*of-nat n*) = *0*
by (*simp add*: *of-nat-poly*)

lemma *lead-coeff-of-nat* [*simp*]: *lead-coeff* (*of-nat n*) = *of-nat n*
by (*simp add*: *of-nat-poly*)

lemma *of-int-poly*: *of-int k* = [:*of-int k*:]
by (*simp only*: *of-int-of-nat of-nat-poly*) *simp*

lemma *of-int-monom*: *of-int k* = *monom* (*of-int k*) *0*
by (*simp add*: *of-int-poly monom-0*)

lemma *degree-of-int* [*simp*]: *degree* (*of-int k*) = *0*
by (*simp add*: *of-int-poly*)

lemma *lead-coeff-of-int* [*simp*]: *lead-coeff* (*of-int k*) = *of-int k*
by (*simp add*: *of-int-poly*)

lemma *poly-of-nat* [*simp*]: *poly* (*of-nat n*) *x* = *of-nat n*
by (*simp add*: *of-nat-poly*)

lemma *poly-of-int* [*simp*]: *poly* (*of-int n*) *x* = *of-int n*
by (*simp add*: *of-int-poly*)

lemma *poly-numeral* [*simp*]: *poly* (*numeral n*) *x* = *numeral n*
by (*metis of-nat-numeral poly-of-nat*)

lemma *numeral-poly*: *numeral n* = [:*numeral n*:]
proof −
have *numeral n* = *of-nat* (*numeral n*)
by *simp*
also have ... = [:*of-nat* (*numeral n*):]
by (*simp add*: *of-nat-poly*)
finally show *?thesis*
by *simp*
qed

lemma *numeral-monom*:
numeral n = *monom* (*numeral n*) *0*
by (*simp add*: *numeral-poly monom-0*)

lemma *degree-numeral* [*simp*]:
degree (*numeral n*) = *0*
by (*simp add*: *numeral-poly*)

lemma *lead-coeff-numeral* [*simp*]:
lead-coeff (*numeral n*) = *numeral n*
by (*simp add*: *numeral-poly*)

lemma *coeff-linear-poly-power*:
fixes *c* :: *'a* :: *semiring-1*
assumes *i* ≤ *n*
shows *coeff* ([:*a*, *b*:] ^ *n*) *i* = *of-nat* (*n choose i*) ∗ *b* ^ *i* ∗ *a* ^ (*n* − *i*)
proof −
have [:*a*, *b*:] = *monom b 1* + [:*a*:]
by (*simp add*: *monom-altdef*)
also have *coeff* (... ^ *n*) *i* = (∑ *k*≤*n*. *a*^(*n*−*k*) ∗ *of-nat* (*n choose k*) ∗ (*if k* = *i then b* ^ *k else 0*))
by (*subst binomial-ring*) (*simp add*: *coeff-sum of-nat-poly monom-power poly-const-pow mult-ac*)

also have ... $= (\sum k{\in}\{i\}.\ a\ \hat{}\ (n - i) * b\ \hat{}\ i * \textit{of-nat}\ (n\ \textit{choose}\ k))$
using *assms* **by** (*intro sum.mono-neutral-cong-right*) (*auto simp*: *mult-ac*)
finally show $*$: *?thesis* **by** (*simp add*: *mult-ac*)
qed

4.17 Lemmas about divisibility

lemma *dvd-smult*:
assumes *p dvd q*
shows *p dvd smult a q*
proof −
from *assms* **obtain** *k* **where** $q = p * k$ **..**
then have *smult a q* $= p *$ *smult a k* **by** *simp*
then show *p dvd smult a q* **..**
qed

lemma *dvd-smult-cancel*: *p dvd smult a q* $\Longrightarrow$ $a \neq 0 \Longrightarrow$ *p dvd q*
for *a* :: *'a*::*field*
by (*drule dvd-smult* [**where** *a*=*inverse a*]) *simp*

lemma *dvd-smult-iff*: $a \neq 0 \Longrightarrow$ *p dvd smult a q* $\longleftrightarrow$ *p dvd q*
for *a* :: *'a*::*field*
by (*safe elim*!: *dvd-smult dvd-smult-cancel*)

lemma *smult-dvd-cancel*:
assumes *smult a p dvd q*
shows *p dvd q*
proof −
from *assms* **obtain** *k* **where** $q =$ *smult a p* $* k$ **..**
then have $q = p *$ *smult a k* **by** *simp*
then show *p dvd q* **..**
qed

lemma *smult-dvd*: *p dvd q* $\Longrightarrow$ $a \neq 0 \Longrightarrow$ *smult a p dvd q*
for *a* :: *'a*::*field*
by (*rule smult-dvd-cancel* [**where** *a*=*inverse a*]) *simp*

lemma *smult-dvd-iff*: *smult a p dvd q* $\longleftrightarrow$ (*if* $a = 0$ *then* $q = 0$ *else p dvd q*)
for *a* :: *'a*::*field*
by (*auto elim*: *smult-dvd smult-dvd-cancel*)

lemma *is-unit-smult-iff*: *smult c p dvd* $1 \longleftrightarrow$ *c dvd* $1 \wedge$ *p dvd* 1
proof −
have *smult c p* $= [:c:] * p$ **by** *simp*
also have ... *dvd* $1 \longleftrightarrow$ *c dvd* $1 \wedge$ *p dvd* 1
proof *safe*
assume $*$: $[:c:] * p$ *dvd* 1
then show *p dvd* 1
by (*rule dvd-mult-right*)

```
    from * obtain q where q: 1 = [:c:] * p * q
      by (rule dvdE)
    have c dvd c * (coeff p 0 * coeff q 0)
      by simp
    also have ... = coeff ([:c:] * p * q) 0
      by (simp add: mult.assoc coeff-mult)
    also note q [symmetric]
    finally have c dvd coeff 1 0 .
    then show c dvd 1 by simp
  next
    assume c dvd 1 p dvd 1
    from this(1) obtain d where 1 = c * d
      by (rule dvdE)
    then have 1 = [:c:] * [:d:]
      by (simp add: one-pCons ac-simps)
    then have [:c:] dvd 1
      by (rule dvdI)
    from mult-dvd-mono[OF this ‹p dvd 1›] show [:c:] * p dvd 1
      by simp
  qed
  finally show ?thesis .
qed
```

4.18 Polynomials form an integral domain

```
instance poly :: (idom) idom ..

instance poly :: ({ring-char-0, comm-ring-1}) ring-char-0
  by standard (auto simp add: of-nat-poly intro: injI)

lemma semiring-char-poly [simp]: CHAR('a :: comm-semiring-1 poly) = CHAR('a)
  by (rule CHAR-eqI) (auto simp: of-nat-poly of-nat-eq-0-iff-char-dvd)

instance poly :: ({semiring-prime-char,comm-semiring-1}) semiring-prime-char
  by (rule semiring-prime-charI) auto
instance poly :: ({comm-semiring-prime-char,comm-semiring-1}) comm-semiring-prime-char
  by standard
instance poly :: ({comm-ring-prime-char,comm-semiring-1}) comm-ring-prime-char
  by standard
instance poly :: ({idom-prime-char,comm-semiring-1}) idom-prime-char
  by standard

lemma linear-poly-root:
  (a :: 'a :: comm-ring-1) ∈ set as ⟹ poly (∏ a ← as. [: - a, 1:]) a = 0
proof (induct as)
  case (Cons b as)
  show ?case
  proof (cases a = b)
    case False
```

with *Cons* **have** $a \in set\ as$ **by** *auto*
from *Cons*(*1*)[*OF this*] **show** *?thesis* **by** *simp*
qed *simp*
qed *simp*

lemma *poly-sum-list-eq*: $poly\ (sum\text{-}list\ ps)\ x = sum\text{-}list\ (map\ (\lambda\ p.\ poly\ p\ x)\ ps)$
by (*induct ps*, *auto*)

lemma *poly-prod-list-eq*: $poly\ (prod\text{-}list\ ps)\ x = prod\text{-}list\ (map\ (\lambda\ p.\ poly\ p\ x)\ ps)$
by (*induct ps*, *auto*)

lemma *sum-list-neutral*: $(\bigwedge\ x.\ x \in set\ xs \Longrightarrow x = 0) \Longrightarrow sum\text{-}list\ xs = 0$
by (*induct xs*) *auto*

lemma *prod-list-neutral*: $(\bigwedge\ x.\ x \in set\ xs \Longrightarrow x = 1) \Longrightarrow prod\text{-}list\ xs = 1$
by (*induct xs*) *auto*

lemma (**in** *comm-monoid-mult*) *prod-list-map-remove1*:
$x \in set\ xs \Longrightarrow prod\text{-}list\ (map\ f\ xs) = f\ x * prod\text{-}list\ (map\ f\ (remove1\ x\ xs))$
by (*induct xs*) (*auto simp add*: *ac-simps*)

lemma *poly-prod-0*: *finite ps* $\Longrightarrow poly\ (prod\ f\ ps)\ x = (0 :: 'a :: field) \longleftrightarrow (\exists\ p \in ps.\ poly\ (f\ p)\ x = 0)$
by (*induct ps rule*: *finite-induct*, *auto*)

lemma *degree-mult-eq*: $p \neq 0 \Longrightarrow q \neq 0 \Longrightarrow degree\ (p * q) = degree\ p + degree\ q$
for $p\ q :: 'a::\{comm\text{-}semiring\text{-}0, semiring\text{-}no\text{-}zero\text{-}divisors\}\ poly$
by (*rule order-antisym* [*OF degree-mult-le le-degree*]) (*simp add*: *coeff-mult-degree-sum*)

lemma *degree-prod-sum-eq*:
$(\bigwedge x.\ x \in A \Longrightarrow f\ x \neq 0) \Longrightarrow$
$degree\ (prod\ f\ A :: 'a :: idom\ poly) = (\sum x \in A.\ degree\ (f\ x))$
by (*induction A rule*: *infinite-finite-induct*) (*auto simp*: *degree-mult-eq*)

lemma *dvd-imp-degree*:
‹*degree x* $\leq$ *degree y*› **if** ‹*x dvd y*› ‹$x \neq 0$› ‹$y \neq 0$›
for $x\ y :: ‹'a::\{comm\text{-}semiring\text{-}1, semiring\text{-}no\text{-}zero\text{-}divisors\}\ poly›$
proof −
from ‹*x dvd y*› **obtain** *z* **where** ‹$y = x * z$› **..**
with ‹$x \neq 0$› ‹$y \neq 0$› **show** *?thesis*
by (*simp add*: *degree-mult-eq*)
qed

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 $degree\ (\prod i \in A.\ (f\ i)) = (\sum i \in A.\ degree\ (f\ i))$
using *assms*

by (*induction A rule*: *infinite-finite-induct*) (*auto simp*: *degree-mult-eq*)

lemma *degree-mult-eq-0*:
degree (*p* ∗ *q*) = *0* ⟷ *p* = *0* ∨ *q* = *0* ∨ (*p* ≠ *0* ∧ *q* ≠ *0* ∧ *degree p* = *0* ∧ *degree q* = *0*)
for *p q* :: *'a*::{*comm-semiring-0*,*semiring-no-zero-divisors*} *poly*
by (*auto simp*: *degree-mult-eq*)

lemma *degree-power-eq*: *p* ≠ *0* ⟹ *degree* ((*p* :: *'a* :: *idom poly*) ^ *n*) = *n* ∗ *degree p*
by (*induction n*) (*simp-all add*: *degree-mult-eq*)

lemma *degree-mult-right-le*:
fixes *p q* :: *'a*::{*comm-semiring-0*,*semiring-no-zero-divisors*} *poly*
assumes *q* ≠ *0*
shows *degree p* ≤ *degree* (*p* ∗ *q*)
using *assms* **by** (*cases p* = *0*) (*simp-all add*: *degree-mult-eq*)

lemma *coeff-degree-mult*: *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*
by (*cases p* = *0* ∨ *q* = *0*) (*auto simp*: *degree-mult-eq coeff-mult-degree-sum mult-ac*)

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*
by (*erule dvdE*, *hypsubst*, *subst degree-mult-eq*) *auto*

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*
by (*metis dvd-imp-degree-le assms*)

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 (*cases c* = *0* ∨ *p* = *0*)
case *True*
then show *?thesis*
by (*auto intro*!: *poly-eqI*)
next
case *False*
show *?thesis*
proof
assume [:*c*:] *dvd p*
then show ∀ *n*. *c dvd coeff p n*
by (*auto simp*: *coeffs-def*)
next

assume $*$: $\forall n.\ c\ dvd\ coeff\ p\ n$
define *mydiv* **where** $mydiv\ x\ y = (SOME\ z.\ x = y * z)$ **for** $x\ y :: 'a$
have *mydiv*: $x = y * mydiv\ x\ y$ **if** $y\ dvd\ x$ **for** $x\ y$
using *that* **unfolding** *mydiv-def dvd-def* **by** (*rule someI-ex*)
define q **where** $q = Poly\ (map\ (\lambda a.\ mydiv\ a\ c)\ (coeffs\ p))$
from *False* $*$ **have** $p = q * [:c:]$
by (*intro poly-eqI*)
(*auto simp*: *q-def nth-default-def not-less length-coeffs-degree coeffs-nth intro*!: *coeff-eq-0 mydiv*)
then show $[:c:]\ dvd\ p$
by (*simp only*: *dvd-triv-right*)
qed
qed

lemma *const-poly-dvd-const-poly-iff* [*simp*]: $[:a:]\ dvd\ [:b:] \longleftrightarrow a\ dvd\ b$
for $a\ b :: 'a::\{comm\text{-}semiring\text{-}1, semiring\text{-}no\text{-}zero\text{-}divisors\}$
by (*subst const-poly-dvd-iff*) (*auto simp*: *coeff-pCons split*: *nat.splits*)

lemma *lead-coeff-mult*: $lead\text{-}coeff\ (p * q) = lead\text{-}coeff\ p * lead\text{-}coeff\ q$
for $p\ q :: 'a::\{comm\text{-}semiring\text{-}0, semiring\text{-}no\text{-}zero\text{-}divisors\}\ poly$
by (*cases* $p = 0 \vee q = 0$) (*auto simp*: *coeff-mult-degree-sum degree-mult-eq*)

lemma *lead-coeff-prod*: $lead\text{-}coeff\ (prod\ f\ A) = (\prod x{\in}A.\ lead\text{-}coeff\ (f\ x))$
for $f :: 'a \Rightarrow 'b::\{comm\text{-}semiring\text{-}1, semiring\text{-}no\text{-}zero\text{-}divisors\}\ poly$
by (*induction A rule*: *infinite-finite-induct*) (*auto simp*: *lead-coeff-mult*)

lemma *lead-coeff-smult*: $lead\text{-}coeff\ (smult\ c\ p) = c * lead\text{-}coeff\ p$
for $p :: 'a::\{comm\text{-}semiring\text{-}0, semiring\text{-}no\text{-}zero\text{-}divisors\}\ poly$
proof −
have $smult\ c\ p = [:c:] * p$ **by** *simp*
also have $lead\text{-}coeff\ \ldots = c * lead\text{-}coeff\ p$
by (*subst lead-coeff-mult*) *simp-all*
finally show *?thesis* **.**
qed

lemma *lead-coeff-1* [*simp*]: $lead\text{-}coeff\ 1 = 1$
by *simp*

lemma *lead-coeff-power*: $lead\text{-}coeff\ (p\ \hat{}\ n) = lead\text{-}coeff\ p\ \hat{}\ n$
for $p :: 'a::\{comm\text{-}semiring\text{-}1, semiring\text{-}no\text{-}zero\text{-}divisors\}\ poly$
by (*induct n*) (*simp-all add*: *lead-coeff-mult*)

4.19 Polynomials form an ordered integral domain

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

lemma *pos-poly-pCons*: $pos\text{-}poly\ (pCons\ a\ p) \longleftrightarrow pos\text{-}poly\ p \vee (p = 0 \wedge 0 < a)$
by (*simp add*: *pos-poly-def*)

lemma *not-pos-poly-0* [*simp*]: ¬ *pos-poly 0*
by (*simp add*: *pos-poly-def*)

lemma *pos-poly-add*: *pos-poly p* ⟹ *pos-poly q* ⟹ *pos-poly* (*p* + *q*)
proof (*induction p arbitrary*: *q*)
case (*pCons a p*)
then show *?case*
by (*cases q*; *force simp add*: *pos-poly-pCons add-pos-pos*)
qed *auto*

lemma *pos-poly-mult*: *pos-poly p* ⟹ *pos-poly q* ⟹ *pos-poly* (*p* ∗ *q*)
by (*simp add*: *pos-poly-def coeff-degree-mult*)

lemma *pos-poly-total*: *p* = *0* ∨ *pos-poly p* ∨ *pos-poly* (− *p*)
for *p* :: *'a*::*linordered-idom poly*
by (*induct p*) (*auto simp*: *pos-poly-pCons*)

lemma *pos-poly-coeffs* [*code*]: *pos-poly p* ⟷ (*let as* = *coeffs p in as* ≠ [] ∧ *last as* > *0*)
(**is** *?lhs* ⟷ *?rhs*)
proof
assume *?rhs*
then show *?lhs*
by (*auto simp add*: *pos-poly-def last-coeffs-eq-coeff-degree*)
next
assume *?lhs*
then have ∗: *0* < *coeff p* (*degree p*)
by (*simp add*: *pos-poly-def*)
then have *p* ≠ *0*
by *auto*
with ∗ **show** *?rhs*
by (*simp add*: *last-coeffs-eq-coeff-degree*)
qed

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

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

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

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

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

instance
proof
fix *x y z* :: *'a poly*

show $x < y \longleftrightarrow x \leq y \wedge \neg\ y \leq x$
unfolding *less-eq-poly-def less-poly-def*
using *pos-poly-add* **by** *force*
then show $x \leq y \Longrightarrow y \leq x \Longrightarrow x = y$
using *less-eq-poly-def less-poly-def* **by** *force*
show $x \leq x$
by (*simp add*: *less-eq-poly-def*)
show $x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z$
using *less-eq-poly-def pos-poly-add* **by** *fastforce*
show $x \leq y \Longrightarrow z + x \leq z + y$
by (*simp add*: *less-eq-poly-def*)
show $x \leq y \vee y \leq x$
unfolding *less-eq-poly-def*
using *pos-poly-total* [*of* $x - y$]
by *auto*
show $x < y \Longrightarrow 0 < z \Longrightarrow z * x < z * y$
by (*simp add*: *less-poly-def right-diff-distrib* [*symmetric*] *pos-poly-mult*)
show $|x| = (\mathit{if}\ x < 0\ \mathit{then} - x\ \mathit{else}\ x)$
by (*rule abs-poly-def*)
show $\mathit{sgn}\ x = (\mathit{if}\ x = 0\ \mathit{then}\ 0\ \mathit{else\ if}\ 0 < x\ \mathit{then}\ 1\ \mathit{else} - 1)$
by (*rule sgn-poly-def*)
qed

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*)
by (*simp add*: *synthetic-divmod-def*)

lemma *synthetic-divmod-pCons* [*simp*]:
synthetic-divmod (*pCons a p*) *c* = (λ(*q*, *r*). (*pCons r q*, *a* + *c* ∗ *r*)) (*synthetic-divmod p c*)
by (*cases p* = *0* ∧ *a* = *0*) (*auto simp add*: *synthetic-divmod-def*)

lemma *synthetic-div-0* [*simp*]: *synthetic-div 0 c* = *0*
by (*simp add*: *synthetic-div-def*)

lemma *synthetic-div-unique-lemma*: *smult c p = pCons a p ⟹ p = 0*
by (*induct p arbitrary*: *a*) *simp-all*

lemma *snd-synthetic-divmod*: *snd* (*synthetic-divmod p c*) = *poly p c*
by (*induct p*) (*simp-all add*: *split-def*)

lemma *synthetic-div-pCons* [*simp*]:
synthetic-div (*pCons a p*) *c* = *pCons* (*poly p c*) (*synthetic-div p c*)
by (*simp add*: *synthetic-div-def split-def snd-synthetic-divmod*)

lemma *synthetic-div-eq-0-iff*: *synthetic-div p c = 0 ⟷ degree p = 0*
proof (*induct p*)
case *0*
then show *?case* **by** *simp*
next
case (*pCons a p*)
then show *?case* **by** (*cases p*) *simp*
qed

lemma *degree-synthetic-div*: *degree* (*synthetic-div p c*) = *degree p − 1*
by (*induct p*) (*simp-all add*: *synthetic-div-eq-0-iff*)

lemma *synthetic-div-correct*:
p + smult c (*synthetic-div p c*) = *pCons* (*poly p c*) (*synthetic-div p c*)
by (*induct p*) *simp-all*

lemma *synthetic-div-unique*: *p + smult c q = pCons r q ⟹ r = poly p c ∧ q =*
synthetic-div p c
proof (*induction p arbitrary*: *q r*)
case *0*
then show *?case*
using *synthetic-div-unique-lemma* **by** *fastforce*
next
case (*pCons a p*)
then show *?case*
by (*cases q*; *force*)
qed

lemma *synthetic-div-correct′*: [:−*c*, *1*:] ∗ *synthetic-div p c* + [:*poly p c*:] = *p*
for *c* :: *′a*::*comm-ring-1*
using *synthetic-div-correct* [*of p c*] **by** (*simp add*: *algebra-simps*)

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

```
  assume ?lhs
  with synthetic-div-correct' [of c p] have p = [:−c, 1:] ∗ synthetic-div p c by simp
  then show ?rhs ..
next
  assume ?rhs
  then obtain k where p = [:−c, 1:] ∗ k by (rule dvdE)
  then show ?lhs by simp
qed

lemma dvd-iff-poly-eq-0: [:c, 1:] dvd p ⟷ poly p (− c) = 0
  for c :: 'a::comm-ring-1
  by (simp add: poly-eq-0-iff-dvd)

lemma poly-roots-finite: p ≠ 0 ⟹ finite {x. poly p x = 0}
  for p :: 'a::{comm-ring-1,ring-no-zero-divisors} poly
proof (induct n ≡ degree p arbitrary: p)
  case 0
  then obtain a where a ≠ 0 and p = [:a:]
    by (cases p) (simp split: if-splits)
  then show finite {x. poly p x = 0}
    by simp
next
  case (Suc n)
  show finite {x. poly p x = 0}
  proof (cases ∃ x. poly p x = 0)
    case False
    then show finite {x. poly p x = 0} by simp
  next
    case True
    then obtain a where poly p a = 0 ..
    then have [:−a, 1:] dvd p
      by (simp only: poly-eq-0-iff-dvd)
    then obtain k where k: p = [:−a, 1:] ∗ k ..
    with ‹p ≠ 0› have k ≠ 0
      by auto
    with k have degree p = Suc (degree k)
      by (simp add: degree-mult-eq del: mult-pCons-left)
    with ‹Suc n = degree p› have n = degree k
      by simp
    from this ‹k ≠ 0› have finite {x. poly k x = 0}
      by (rule Suc.hyps)
    then have finite (insert a {x. poly k x = 0})
      by simp
    then show finite {x. poly p x = 0}
      by (simp add: k Collect-disj-eq del: mult-pCons-left)
  qed
qed

lemma poly-eq-poly-eq-iff: poly p = poly q ⟷ p = q
```

```
  (is ?lhs ⟷ ?rhs)
  for p q :: 'a::{comm-ring-1,ring-no-zero-divisors,ring-char-0} poly
proof
  assume ?rhs
  then show ?lhs by simp
next
  assume ?lhs
  have poly p = poly 0 ⟷ p = 0 for p :: 'a poly
  proof (cases p = 0)
    case False
    then show ?thesis
      by (auto simp add: infinite-UNIV-char-0 dest: poly-roots-finite)
  qed auto
  from ‹?lhs› and this [of p − q] show ?rhs
    by auto
qed
```

A nice extension rule for polynomials.

```
lemma poly-ext:
  fixes p q :: 'a :: {ring-char-0, idom} poly
  assumes ⋀x. poly p x = poly q x shows p = q
  unfolding poly-eq-poly-eq-iff [symmetric]
  using assms by (rule ext)
```

Copied from non-negative variants.

```
lemma coeff-linear-power-neg[simp]:
  fixes a :: 'a::comm-ring-1
  shows coeff ([:a, −1:] ^ n) n = (−1) ^ n
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  then have degree ([:a, − 1:] ^ n) < Suc n
    by (auto intro: le-less-trans degree-power-le)
  with Suc show ?case
    by (simp add: coeff-eq-0)
qed

lemma degree-linear-power-neg[simp]:
  fixes a :: 'a::{idom,comm-ring-1}
  shows degree ([:a, −1:] ^ n) = n
  by (simp add: degree-power-eq)

lemma poly-all-0-iff-0: (∀ x. poly p x = 0) ⟷ p = 0
  for p :: 'a::{ring-char-0,comm-ring-1,ring-no-zero-divisors} poly
  by (auto simp add: poly-eq-poly-eq-iff [symmetric])

lemma poly-roots-degree:
```

fixes $p :: 'a::\{comm\text{-}ring\text{-}1,ring\text{-}no\text{-}zero\text{-}divisors\}\ poly$
assumes $p \neq 0$
shows $card\ \{x.\ poly\ p\ x = 0\} \leq degree\ p$
using *assms*
proof (*induction degree p arbitrary*: *p rule*: *less-induct*)
case (*less p*)
show *?case*
proof (*cases* $\exists x.\ poly\ p\ x = 0$)
case *False*
hence $\{x.\ poly\ p\ x = 0\} = \{\}$ **by** *blast*
thus *?thesis* **by** *simp*
next
case *True*
then obtain *x* **where** *x*: $poly\ p\ x = 0$ **by** *blast*
hence $[:-x,\ 1:]\ dvd\ p$ **by** (*subst* (*asm*) *poly-eq-0-iff-dvd*)
then obtain *q* **where** *q*: $p = [:-x,\ 1:] * q$ **by** (*auto simp*: *dvd-def*)
with ‹$p \neq 0$› **have** [*simp*]: $q \neq 0$ **by** *auto*
have *deg*: $degree\ p = Suc\ (degree\ q)$
by (*subst q*, *subst degree-mult-eq*) *auto*
have $card\ \{x.\ poly\ p\ x = 0\} \leq card\ (insert\ x\ \{x.\ poly\ q\ x = 0\})$
by (*intro card-mono*) (*auto intro*: *poly-roots-finite simp*: *q*)
also have $\ldots \leq Suc\ (card\ \{x.\ poly\ q\ x = 0\})$
by (*rule card-insert-le-m1*) *auto*
also from *deg* **have** $card\ \{x.\ poly\ q\ x = 0\} \leq degree\ q$
using ‹$p \neq 0$› **and** *q* **by** (*intro less*) *auto*
also have $Suc\ \ldots = degree\ p$ **by** (*simp add*: *deg*)
finally show *?thesis* **by** − *simp-all*
qed
qed

lemma *poly-eqI-degree*:
fixes $p\ q :: 'a :: \{comm\text{-}ring\text{-}1,\ ring\text{-}no\text{-}zero\text{-}divisors\}\ poly$
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 (*rule ccontr*)
assume *neq*: $p \neq q$
have $degree\ (p - q) \leq max\ (degree\ p)\ (degree\ q)$
by (*rule degree-diff-le-max*)
also from *assms* **have** $\ldots < card\ A$ **by** *linarith*
also have $\ldots \leq card\ \{x.\ poly\ (p - q)\ x = 0\}$
using *neq* **and** *assms* **by** (*intro card-mono poly-roots-finite*) *auto*
finally have $degree\ (p - q) < card\ \{x.\ poly\ (p - q)\ x = 0\}$ **.**
moreover have $degree\ (p - q) \geq card\ \{x.\ poly\ (p - q)\ x = 0\}$
using *neq* **by** (*intro poly-roots-degree*) *auto*
ultimately show *False* **by** *linarith*
qed

lemma *poly-eqI-degree-lead-coeff*:

```
  fixes p q :: 'a :: {comm-ring-1, ring-no-zero-divisors} poly
  assumes poly.coeff p n = poly.coeff q n card A ≥ n degree p ≤ n degree q ≤ n
  assumes ⋀z. z ∈ A ⟹ poly p z = poly q z
  shows   p = q
proof (rule ccontr)
  assume p ≠ q

  have n > 0
  proof (rule ccontr)
    assume ¬(n > 0)
    thus False
      using assms ‹p ≠ q› by (auto elim!: degree-eq-zeroE)
  qed

  have n ≤ card A
    by fact
  also have card A ≤ card {x. poly (p − q) x = 0}
    by (intro card-mono poly-roots-finite) (use ‹p ≠ q› assms in auto)
  also have card {x. poly (p − q) x = 0} ≤ degree (p − q)
    by (rule poly-roots-degree) (use ‹p ≠ q› in auto)
  also have degree (p − q) < n
  proof (intro degree-lessI allI impI)
    fix k assume k ≥ n
    show poly.coeff (p − q) k = 0
    proof (cases k = n)
      case False
      hence poly.coeff p k = 0 poly.coeff q k = 0
        using assms ‹k ≥ n› by (auto simp: coeff-eq-0)
      thus ?thesis
        by simp
    qed (use assms in auto)
  qed (use ‹n > 0› in auto)
  finally show False
    by simp
qed
```

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 (induct n)
  case (Suc n)
  have degree ([:a, 1:] ^ n) ≤ 1 * n
   by (metis One-nat-def degree-pCons-eq-if degree-power-le one-neq-zero one-pCons)
  then have coeff ([:a, 1:] ^ n) (Suc n) = 0
    by (simp add: coeff-eq-0)
```

then show *?case*
using *Suc.hyps* **by** *fastforce*
qed *auto*

lemma *degree-linear-power*: *degree* ([:*a*, *1*:] ^ *n*) = *n*
for *a* :: *'a*::*comm-semiring-1*
proof (*rule order-antisym*)
show *degree* ([:*a*, *1*:] ^ *n*) ≤ *n*
by (*metis One-nat-def degree-pCons-eq-if degree-power-le mult.left-neutral one-neq-zero one-pCons*)
qed (*simp add*: *coeff-linear-power le-degree*)

lemma *order-1*: [:−*a*, *1*:] ^ *order a p dvd p*
proof (*cases p* = *0*)
case *False*
show *?thesis*
proof (*cases order a p*)
case (*Suc n*)
then show *?thesis*
by (*metis lessI not-less-Least order-def*)
qed *auto*
qed *auto*

lemma *order-2*:
assumes *p* ≠ *0*
shows ¬ [:−*a*, *1*:] ^ *Suc* (*order a p*) *dvd p*
proof −
have *False* **if** [:− *a*, *1*:] ^ *Suc* (*degree p*) *dvd p*
using *dvd-imp-degree-le* [*OF that*]
by (*metis Suc-n-not-le-n assms degree-linear-power*)
then show *?thesis*
unfolding *order-def*
by (*metis* (*no-types*, *lifting*) *LeastI*)
qed

lemma *order*: *p* ≠ *0* ⟹ [:−*a*, *1*:] ^ *order a p dvd p* ∧ ¬ [:−*a*, *1*:] ^ *Suc* (*order a p*) *dvd p*
by (*rule conjI* [*OF order-1 order-2*])

lemma *order-degree*:
assumes *p*: *p* ≠ *0*
shows *order a p* ≤ *degree p*
proof −
have *order a p* = *degree* ([:−*a*, *1*:] ^ *order a p*)
by (*simp only*: *degree-linear-power*)
also from *order-1 p* **have** ... ≤ *degree p*
by (*rule dvd-imp-degree-le*)
finally show *?thesis* **.**
qed

lemma *order-root*: *poly p a = 0 ⟷ p = 0 ∨ order a p ≠ 0* (**is** *?lhs = ?rhs*)
proof
  show *?lhs ⟹ ?rhs*
    by (*metis One-nat-def order-2 poly-eq-0-iff-dvd power-one-right*)
  show *?rhs ⟹ ?lhs*
    by (*meson dvd-power dvd-trans neq0-conv order-1 poly-0 poly-eq-0-iff-dvd*)
qed

lemma *order-0I*: *poly p a ≠ 0 ⟹ order a p = 0*
  by (*subst* (*asm*) *order-root*) *auto*

lemma *order-unique-lemma*:
  fixes *p* :: *'a*::*idom poly*
  assumes [:−*a*, *1*:] ^ *n dvd p* ¬ [:−*a*, *1*:] ^ *Suc n dvd p*
  shows *order a p = n*
  unfolding *Polynomial.order-def*
  by (*metis* (*mono-tags*, *lifting*) *Least-equality assms not-less-eq-eq power-le-dvd*)

lemma *order-mult*:
  assumes *p ∗ q ≠ 0* **shows** *order a* (*p ∗ q*) = *order a p* + *order a q*
proof −
  define *i* **where** *i* ≡ *order a p*
  define *j* **where** *j* ≡ *order a q*
  define *t* **where** *t* ≡ [:−*a*, *1*:]
  have *t-dvd-iff*: ⋀*u*. *t dvd u ⟷ poly u a = 0*
    by (*simp add*: *t-def dvd-iff-poly-eq-0*)
  have *dvd*: *t* ^ *i dvd p t* ^ *j dvd q* **and** ¬ *t* ^ *Suc i dvd p* ¬ *t* ^ *Suc j dvd q*
    using *assms i-def j-def order-1 order-2 t-def* **by** *auto*
  then have ¬ *t* ^ *Suc*(*i* + *j*) *dvd p ∗ q*
    by (*elim dvdE*) (*simp add*: *power-add t-dvd-iff*)
  moreover have *t* ^ (*i* + *j*) *dvd p ∗ q*
    using *dvd* **by** (*simp add*: *mult-dvd-mono power-add*)
  ultimately show *order a* (*p ∗ q*) = *i* + *j*
    using *order-unique-lemma t-def* **by** *blast*
qed

lemma *order-smult*:
  assumes *c ≠ 0*
  shows *order x* (*smult c p*) = *order x p*
proof (*cases p = 0*)
  case *True*
  then show *?thesis*
    by *simp*
next
  case *False*
  have *smult c p* = [:*c*:] ∗ *p* **by** *simp*
  also from *assms False* **have** *order x* ... = *order x* [:*c*:] + *order x p*

by (*subst order-mult*) *simp-all*
also have *order x* [:*c*:] = *0*
by (*rule order-0I*) (*use assms* **in** *auto*)
finally show *?thesis*
by *simp*
qed

lemma *order-gt-0-iff*: $p \neq 0 \Longrightarrow order\ x\ p > 0 \longleftrightarrow poly\ p\ x = 0$
by (*subst order-root*) *auto*

lemma *order-eq-0-iff*: $p \neq 0 \Longrightarrow order\ x\ p = 0 \longleftrightarrow poly\ p\ x \neq 0$
by (*subst order-root*) *auto*

Next three lemmas contributed by Wenda Li

lemma *order-1-eq-0* [*simp*]:*order x 1 = 0*
by (*metis order-root poly-1 zero-neq-one*)

lemma *order-uminus*[*simp*]: *order x* (−*p*) = *order x p*
by (*metis neg-equal-0-iff-equal order-smult smult-1-left smult-minus-left*)

lemma *order-power-n-n*: *order a* ([:−*a*,*1*:]^*n*)=*n*
proof (*induct n*)
case *0*
then show *?case*
by (*metis order-root poly-1 power-0 zero-neq-one*)
next
case (*Suc n*)
have *order a* ([:− *a*, *1*:] ^ *Suc n*) = *order a* ([:− *a*, *1*:] ^ *n*) + *order a* [:−*a*,*1*:]
by (*metis* (*no-types*, *opaque-lifting*) *One-nat-def add-Suc-right monoid-add-class.add.right-neutral one-neq-zero order-mult pCons-eq-0-iff power-add power-eq-0-iff power-one-right*)
moreover have *order a* [:−*a*,*1*:] = *1*
unfolding *order-def*
proof (*rule Least-equality*, *rule notI*)
assume [:− *a*, *1*:] ^ *Suc 1 dvd* [:− *a*, *1*:]
then have *degree* ([:− *a*, *1*:] ^ *Suc 1*) ≤ *degree* ([:− *a*, *1*:])
by (*rule dvd-imp-degree-le*) *auto*
then show *False*
by *auto*
next
fix *y*
assume ∗: ¬ [:− *a*, *1*:] ^ *Suc y dvd* [:− *a*, *1*:]
show *1* ≤ *y*
proof (*rule ccontr*)
assume ¬ *1* ≤ *y*
then have *y* = *0* **by** *auto*
then have [:− *a*, *1*:] ^ *Suc y dvd* [:− *a*, *1*:] **by** *auto*
with ∗ **show** *False* **by** *auto*
qed
qed

ultimately show *?case*
using *Suc* **by** *auto*
qed

lemma *order-linear*[*simp*]: *order x* [:−*y*, *1*:] = (*if x*=*y then 1 else 0*)
by (*auto simp add*:*order-power-n-n*[**where** *n*=*1*,*simplified*] *order-0I*)

lemma *order-0-monom* [*simp*]: $c \neq 0 \Longrightarrow$ *order 0* (*monom c n*) = *n*
using *order-power-n-n*[*of 0 n*] **by** (*simp add*: *monom-altdef order-smult*)

lemma *dvd-imp-order-le*: $q \neq 0 \Longrightarrow p$ *dvd q* $\Longrightarrow$ *Polynomial.order a p* ≤ *Polynomial.order a q*
by (*auto simp*: *order-mult*)

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

lemma *order-divides*: [:−*a*, *1*:] ^ *n dvd p* ⟷ *p* = *0* ∨ *n* ≤ *order a p*
by (*meson dvd-0-right not-less-eq-eq order-1 order-2 power-le-dvd*)

lemma *order-decomp*:
assumes *p* ≠ *0*
shows ∃ *q*. *p* = [:− *a*, *1*:] ^ *order a p* ∗ *q* ∧ ¬ [:− *a*, *1*:] *dvd q*
proof −
from *assms* **have** ∗: [:− *a*, *1*:] ^ *order a p dvd p*
and ∗∗: ¬ [:− *a*, *1*:] ^ *Suc* (*order a p*) *dvd p*
by (*auto dest*: *order*)
from ∗ **obtain** *q* **where** *q*: *p* = [:− *a*, *1*:] ^ *order a p* ∗ *q* **..**
with ∗∗ **have** ¬ [:− *a*, *1*:] ^ *Suc* (*order a p*) *dvd* [:− *a*, *1*:] ^ *order a p* ∗ *q*
by *simp*
then have ¬ [:− *a*, *1*:] ^ *order a p* ∗ [:− *a*, *1*:] *dvd* [:− *a*, *1*:] ^ *order a p* ∗ *q*
by *simp*
with *idom-class.dvd-mult-cancel-left* [*of* [:− *a*, *1*:] ^ *order a p* [:− *a*, *1*:] *q*]
have ¬ [:− *a*, *1*:] *dvd q* **by** *auto*
with *q* **show** *?thesis* **by** *blast*
qed

lemma *monom-1-dvd-iff*: *p* ≠ *0* ⟹ *monom 1 n dvd p* ⟷ *n* ≤ *order 0 p*
using *order-divides*[*of 0 n p*] **by** (*simp add*: *monom-altdef*)

lemma *poly-root-order-induct* [*case-names 0 no-roots root*]:
fixes *p* :: *'a* :: *idom poly*
assumes *P 0* ⋀*p*. (⋀*x*. *poly p x* ≠ *0*) ⟹ *P p*
⋀*p x n*. *n* > *0* ⟹ *poly p x* ≠ *0* ⟹ *P p* ⟹ *P* ([:−*x*, *1*:] ^ *n* ∗ *p*)
shows *P p*
proof (*induction degree p arbitrary*: *p rule*: *less-induct*)
case (*less p*)
consider *p* = *0* | *p* ≠ *0* ∃ *x*. *poly p x* = *0* | ⋀*x*. *poly p x* ≠ *0* **by** *blast*
thus *?case*
proof *cases*
case *3*

```
    with assms(2)[of p] show ?thesis by simp
  next
    case 2
    then obtain x where x: poly p x = 0 by auto
    have [:−x, 1:] ^ order x p dvd p by (intro order-1)
    then obtain q where q: p = [:−x, 1:] ^ order x p * q by (auto simp: dvd-def)
    with 2 have [simp]: q ≠ 0 by auto
    have order-pos: order x p > 0
      using ‹p ≠ 0› and x by (auto simp: order-root)
    have order x p = order x p + order x q
      by (subst q, subst order-mult) (auto simp: order-power-n-n)
    hence [simp]: order x q = 0 by simp
    have deg: degree p = order x p + degree q
      by (subst q, subst degree-mult-eq) (auto simp: degree-power-eq)
    with order-pos have degree q < degree p by simp
    hence P q by (rule less)
    with order-pos have P ([:−x, 1:] ^ order x p * q)
      by (intro assms(3)) (auto simp: order-root)
    with q show ?thesis by simp
  qed (simp-all add: assms(1))
qed


context
  includes multiset.lifting
begin

lift-definition proots :: ('a :: idom) poly ⇒ 'a multiset is
  λ(p :: 'a poly) (x :: 'a). if p = 0 then 0 else order x p
proof −
  fix p :: 'a poly
  show finite {x. 0 < (if p = 0 then 0 else order x p)}
    by (cases p = 0)
       (auto simp: order-gt-0-iff intro: finite-subset[OF - poly-roots-finite[of p]])
qed

lemma proots-0 [simp]: proots (0 :: 'a :: idom poly) = {#}
  by transfer' auto

lemma proots-1 [simp]: proots (1 :: 'a :: idom poly) = {#}
  by transfer' auto

lemma proots-const [simp]: proots [: x :] = 0
  by transfer' (auto split: if-splits simp: fun-eq-iff order-eq-0-iff)

lemma proots-numeral [simp]: proots (numeral n) = 0
  by (simp add: numeral-poly)

lemma count-proots [simp]:
```

$p \neq 0 \Longrightarrow$ *count* (*proots p*) *a* = *order a p*
by *transfer′ auto*

lemma *set-count-proots* [*simp*]:
$p \neq 0 \Longrightarrow$ *set-mset* (*proots p*) = {*x*. *poly p x* = *0*}
by (*auto simp*: *set-mset-def order-gt-0-iff*)

lemma *proots-uminus* [*simp*]: *proots* (−*p*) = *proots p*
by (*cases p* = *0*; *rule multiset-eqI*) *auto*

lemma *proots-smult* [*simp*]: *c* ≠ *0* ⟹ *proots* (*smult c p*) = *proots p*
by (*cases p* = *0*; *rule multiset-eqI*) (*auto simp*: *order-smult*)

lemma *proots-mult*:
assumes *p* ≠ *0 q* ≠ *0*
shows *proots* (*p* ∗ *q*) = *proots p* + *proots q*
using *assms* **by** (*intro multiset-eqI*) (*auto simp*: *order-mult*)

lemma *proots-prod*:
assumes ⋀*x*. *x* ∈ *A* ⟹ *f x* ≠ *0*
shows *proots* (∏ *x*∈*A*. *f x*) = (∑ *x*∈*A*. *proots* (*f x*))
using *assms* **by** (*induction A rule*: *infinite-finite-induct*) (*auto simp*: *proots-mult*)

lemma *proots-prod-mset*:
assumes *0* ∉# *A*
shows *proots* (∏ *p*∈#*A*. *p*) = (∑ *p*∈#*A*. *proots p*)
using *assms* **by** (*induction A*) (*auto simp*: *proots-mult*)

lemma *proots-prod-list*:
assumes *0* ∉ *set ps*
shows *proots* (∏ *p*←*ps*. *p*) = (∑ *p*←*ps*. *proots p*)
using *assms* **by** (*induction ps*) (*auto simp*: *proots-mult prod-list-zero-iff*)

lemma *proots-power*: *proots* (*p* ^ *n*) = *repeat-mset n* (*proots p*)
proof (*cases p* = *0*)
case *False*
thus *?thesis*
by (*induction n*) (*auto simp*: *proots-mult*)
qed (*auto simp*: *power-0-left*)

lemma *proots-linear-factor* [*simp*]: *proots* [:*x*, *1*:] = {#−*x*#}
proof −
have *order* (−*x*) [:*x*, *1*:] > *0*
by (*subst order-gt-0-iff*) *auto*
moreover have *order* (−*x*) [:*x*, *1*:] ≤ *degree* [:*x*, *1*:]
by (*rule order-degree*) *auto*
moreover have *order y* [:*x*, *1*:] = *0* **if** *y* ≠ −*x* **for** *y*
by (*rule order-0I*) (*use that* **in** ‹*auto simp*: *add-eq-0-iff*›)
ultimately show *?thesis*

by (*intro multiset-eqI*) *auto*
qed

lemma *size-proots-le*: *size* (*proots p*) ≤ *degree p*
proof (*induction p rule*: *poly-root-order-induct*)
case (*no-roots p*)
hence *proots p* = *0*
by (*simp add*: *multiset-eqI order-root*)
thus *?case* **by** *simp*
next
case (*root p x n*)
have [*simp*]: *p* ≠ *0*
using *root.hyps* **by** *auto*
from *root.IH* **show** *?case*
by (*auto simp*: *proots-mult proots-power degree-mult-eq degree-power-eq*)
qed *auto*

end

lemma *proots-empty*: *proots p* = {#} ⟷ *p* = *0* ∨ (∀ *x*. *poly p x* ≠ *0*)
proof
show *proots p* = {#} ⟹ *p* = *0* ∨ (∀ *x*. *poly p x* ≠ *0*)
using *order-root count-empty count-proots* **by** *metis*
next
have (∀ *x*. *poly p x* ≠ *0*) ⟹ *proots p* = {#}
by (*simp add*: *multiset-eqI order-root*)
then show *p* = *0* ∨ (∀ *x*. *poly p x* ≠ *0*) ⟹ *proots p* = {#}
by *auto*
qed

lemma *proots-element*: *x* ∈# *proots p* ∨ *p* = *0* ⟷ *poly p x* = *0*
by (*cases p* = *0*) *auto*

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 ⋀*p*. (⋀*a*. *P a* ⟹ *poly p a* ≠ *0*) ⟹ *Q p*
and ⋀*a p*. *P a* ⟹ *Q p* ⟹ *Q* ([:*a*, −*1*:] ∗ *p*)
shows *Q p*
proof (*induction degree p arbitrary*: *p rule*: *less-induct*)
case (*less p*)
show *?case*
proof (*cases p* = *0*)
case *True*
with *assms*(*1*) **show** *?thesis* **by** *simp*

```
  next
    case False
    show ?thesis
    proof (cases ∃a. P a ∧ poly p a = 0)
      case False
      then show ?thesis by (intro assms(2)) blast
    next
      case True
      then obtain a where a: P a poly p a = 0
        by blast
      then have −[:−a, 1:] dvd p
        by (subst minus-dvd-iff) (simp add: poly-eq-0-iff-dvd)
      then obtain q where q: p = [:a, −1:] ∗ q by (elim dvdE) simp
      with False have q ≠ 0 by auto
      have degree p = Suc (degree q)
        by (subst q, subst degree-mult-eq) (simp-all add: ‹q ≠ 0›)
      then have Q q by (intro less) simp
      with a(1) have Q ([:a, −1:] ∗ q)
        by (rule assms(3))
      with q show ?thesis by simp
    qed
  qed
qed
```

Same proof as above. Could they be consolidated?

```
lemma poly-root-induct-alt [case-names 0 no-proots root]:
  fixes p :: 'a :: idom poly
  assumes Q 0
  assumes ⋀p. (⋀a. P a ⟹ poly p a ≠ 0) ⟹ Q p
  assumes ⋀a p. P a ⟹ Q p ⟹ Q ([:−a, 1:] ∗ p)
  shows   Q p
proof (induction degree p arbitrary: p rule: less-induct)
  case (less p)
  show ?case
  proof (cases p = 0)
    case True
    with assms(1) show ?thesis by simp
  next
    case False
    show ?thesis
    proof (cases ∃a. P a ∧ poly p a = 0)
      case False
      then show ?thesis by (intro assms(2)) blast
    next
      case True
      then obtain a where a: P a poly p a = 0
        by blast
      then have −[:a, −1:] dvd p
        by (simp add: poly-eq-0-iff-dvd)
```

```
    then obtain q where q: p = [:−a, 1:] * q by (elim dvdE) simp
    with False have q ≠ 0 by auto
    have degree p = Suc (degree q)
      by (subst q, subst degree-mult-eq) (simp-all add: ‹q ≠ 0›)
    then have Q q by (intro less) simp
    with a(1) have Q ([:−a, 1:] * q)
      by (rule assms(3))
    with q show ?thesis by simp
  qed
 qed
qed

lemma dropWhile-replicate-append:
  dropWhile ((=) a) (replicate n a @ ys) = dropWhile ((=) a) ys
  by (induct n) simp-all
```

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 ⋀a p b q. P p q ⟹ P (pCons a p) (pCons b q)
  shows P p q
proof −
  define n where n = max (length (coeffs p)) (length (coeffs q))
  define xs where xs = coeffs p @ (replicate (n − length (coeffs p)) 0)
  define ys where ys = coeffs q @ (replicate (n − length (coeffs q)) 0)
  have length xs = length ys
    by (simp add: xs-def ys-def n-def)
  then have P (Poly xs) (Poly ys)
    by (induct rule: list-induct2) (simp-all add: assms)
  also have Poly xs = p
    by (simp add: xs-def Poly-append-replicate-0)
  also have Poly ys = q
    by (simp add: ys-def Poly-append-replicate-0)
  finally show ?thesis .
qed
```

4.22 Composition of polynomials

```
definition pcompose :: 'a::comm-semiring-0 poly ⇒ 'a poly ⇒ 'a poly
  where pcompose p q = fold-coeffs (λa c. [:a:] + q * c) p 0

notation pcompose (infixl ‹∘p› 71)

lemma pcompose-0 [simp]: pcompose 0 q = 0
  by (simp add: pcompose-def)

lemma pcompose-pCons: pcompose (pCons a p) q = [:a:] + q * pcompose p q
  by (cases p = 0 ∧ a = 0) (auto simp add: pcompose-def)
```

lemma *pcompose-altdef*: *pcompose p q = poly (map-poly (λx. [:x:]) p) q*
by (*induction p*) (*simp-all add*: *map-poly-pCons pcompose-pCons*)

lemma *coeff-pcompose-0* [*simp*]:
coeff (pcompose p q) 0 = poly p (coeff q 0)
by (*induction p*) (*simp-all add*: *coeff-mult-0 pcompose-pCons*)

lemma *pcompose-1*: *pcompose 1 p = 1*
for *p* :: *'a*::*comm-semiring-1 poly*
by (*auto simp*: *one-pCons pcompose-pCons*)

lemma *poly-pcompose*: *poly (pcompose p q) x = poly p (poly q x)*
by (*induct p*) (*simp-all add*: *pcompose-pCons*)

lemma *degree-pcompose-le*: *degree (pcompose p q) ≤ degree p * degree q*
proof (*induction p*)
case (*pCons a p*)
then show *?case*
proof (*clarsimp simp add*: *pcompose-pCons*)
assume *degree (p ∘$_p$ q) ≤ degree p * degree q p ≠ 0*
then have *degree (q * p ∘$_p$ q) ≤ degree q + degree p * degree q*
by (*meson add-le-cancel-left degree-mult-le dual-order.trans pCons.IH*)
then show *degree ([:a:] + q * p ∘$_p$ q) ≤ degree q + degree p * degree q*
by (*simp add*: *degree-add-le*)
qed
qed *auto*

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 (*induction p q rule*: *poly-induct2*)
case *0*
then show *?case* **by** *simp*
next
case (*pCons a p b q*)
have *pcompose (pCons a p + pCons b q) r = [:a + b:] + r * pcompose p r + r * pcompose q r*
by (*simp-all add*: *pcompose-pCons pCons.IH algebra-simps*)
also have *[:a + b:] = [:a:] + [:b:]* **by** *simp*
also have *... + r * pcompose p r + r * pcompose q r = pcompose (pCons a p) r + pcompose (pCons b q) r*
by (*simp only*: *pcompose-pCons add-ac*)
finally show *?case* **.**
qed

lemma *pcompose-uminus*: *pcompose (−p) r = −pcompose p r*
for *p r* :: *'a*::*comm-ring poly*
by (*induct p*) (*simp-all add*: *pcompose-pCons*)

lemma *pcompose-diff*: *pcompose (p − q) r = pcompose p r − pcompose q r*

for *p q r* :: *'a*::*comm-ring poly*
using *pcompose-add*[*of p* −*q*] **by** (*simp add*: *pcompose-uminus*)

lemma *pcompose-smult*: *pcompose* (*smult a p*) *r* = *smult a* (*pcompose p r*)
for *p r* :: *'a*::*comm-semiring-0 poly*
by (*induct p*) (*simp-all add*: *pcompose-pCons pcompose-add smult-add-right*)

lemma *pcompose-mult*: *pcompose* (*p* ∗ *q*) *r* = *pcompose p r* ∗ *pcompose q r*
for *p q r* :: *'a*::*comm-semiring-0 poly*
by (*induct p arbitrary*: *q*) (*simp-all add*: *pcompose-add pcompose-smult pcompose-pCons algebra-simps*)

lemma *pcompose-assoc*: *pcompose p* (*pcompose q r*) = *pcompose* (*pcompose p q*) *r*
for *p q r* :: *'a*::*comm-semiring-0 poly*
by (*induct p arbitrary*: *q*) (*simp-all add*: *pcompose-pCons pcompose-add pcompose-mult*)

lemma *pcompose-idR*[*simp*]: *pcompose p* [: *0*, *1* :] = *p*
for *p* :: *'a*::*comm-semiring-1 poly*
by (*induct p*) (*simp-all add*: *pcompose-pCons*)

lemma *pcompose-sum*: *pcompose* (*sum f A*) *p* = *sum* (λ*i*. *pcompose* (*f i*) *p*) *A*
by (*induct A rule*: *infinite-finite-induct*) (*simp-all add*: *pcompose-1 pcompose-add*)

lemma *pcompose-prod*: *pcompose* (*prod f A*) *p* = *prod* (λ*i*. *pcompose* (*f i*) *p*) *A*
by (*induct A rule*: *infinite-finite-induct*) (*simp-all add*: *pcompose-1 pcompose-mult*)

lemma *pcompose-const* [*simp*]: *pcompose* [:*a*:] *q* = [:*a*:]
by (*subst pcompose-pCons*) *simp*

lemma *pcompose-0'*: *pcompose p 0* = [:*coeff p 0*:]
by (*induct p*) (*auto simp add*: *pcompose-pCons*)

lemma *pcompose-coeff-0*:
coeff (*pcompose p q*) *0* = *poly p* (*coeff q 0*)
by (*metis poly-0-coeff-0 poly-pcompose*)

lemma *pcompose-pCons-0*: *pcompose p* [:*a*:] = [:*poly p a*:]
by (*metis* (*no-types*, *lifting*) *coeff-pCons-0 pcompose-0' pcompose-assoc poly-0-coeff-0 poly-pcompose*)

lemma *degree-pcompose*: *degree* (*pcompose p q*) = *degree p* ∗ *degree q*
for *p q* :: *'a*::{*comm-semiring-0*,*semiring-no-zero-divisors*} *poly*
proof (*induct p*)
case *0*
then show *?case* **by** *auto*
next
case (*pCons a p*)
consider *degree* (*q* ∗ *pcompose p q*) = *0* | *degree* (*q* ∗ *pcompose p q*) > *0*

```
    by blast
  then show ?case
  proof cases
    case prems: 1
    show ?thesis
    proof (cases p = 0)
      case True
      then show ?thesis by auto
    next
      case False
      from prems have degree q = 0 ∨ pcompose p q = 0
        by (auto simp add: degree-mult-eq-0)
      moreover have False if pcompose p q = 0 degree q ≠ 0
      proof −
        from pCons.hyps(2) that have degree p = 0
          by auto
        then obtain a1 where p = [:a1:]
          by (metis degree-pCons-eq-if old.nat.distinct(2) pCons-cases)
        with ‹pcompose p q = 0› ‹p ≠ 0› show False
          by auto
      qed
      ultimately have degree (pCons a p) * degree q = 0
        by auto
      moreover have degree (pcompose (pCons a p) q) = 0
      proof −
        from prems have 0 = max (degree [:a:]) (degree (q * pcompose p q))
          by simp
        also have ... ≥ degree ([:a:] + q * pcompose p q)
          by (rule degree-add-le-max)
        finally show ?thesis
          by (auto simp add: pcompose-pCons)
      qed
      ultimately show ?thesis by simp
    qed
  next
    case prems: 2
    then have p ≠ 0 q ≠ 0 pcompose p q ≠ 0
      by auto
    from prems degree-add-eq-right [of [:a:]]
    have degree (pcompose (pCons a p) q) = degree (q * pcompose p q)
      by (auto simp: pcompose-pCons)
   with pCons.hyps(2) degree-mult-eq[OF ‹q≠0› ‹pcompose p q≠0›] show ?thesis
      by auto
  qed
qed

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 −
  from assms degree-pcompose [of p q] have degree p = 0
    by auto
  then obtain a where p = [:a:]
    by (metis degree-pCons-eq-if gr0-conv-Suc neq0-conv pCons-cases)
  with assms(1) have a = 0
    by auto
  with ‹p = [:a:]› show ?thesis
    by simp
qed

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 ⟷ p = 0
  using pcompose-eq-0[OF - assms] by auto

lemma coeff-pcompose-linear:
  coeff (pcompose p [:0, a :: 'a :: comm-semiring-1:]) i = a ^ i * coeff p i
  by (induction p arbitrary: i) (auto simp: pcompose-pCons coeff-pCons mult-ac
split: nat.splits)

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 (induct p)
  case 0
  then show ?case by auto
next
  case (pCons a p)
  consider degree (q * pcompose p q) = 0 | degree (q * pcompose p q) > 0
    by blast
  then show ?case
  proof cases
    case prems: 1
    then have pcompose p q = 0
      by (metis assms degree-0 degree-mult-eq-0 neq0-conv)
    with pcompose-eq-0[OF - ‹degree q > 0›] have p = 0
      by simp
    then show ?thesis
      by auto
  next
    case prems: 2
    then have degree [:a:] < degree (q * pcompose p q)
      by simp
    then have lead-coeff ([:a:] + q * p ∘p q) = lead-coeff (q * p ∘p q)
      by (rule lead-coeff-add-le)
```

```
    then have lead-coeff (pcompose (pCons a p) q) = lead-coeff (q * pcompose p
q)
      by (simp add: pcompose-pCons)
    also have ... = lead-coeff q * (lead-coeff p * lead-coeff q ^ degree p)
      using pCons.hyps(2) lead-coeff-mult[of q pcompose p q] by simp
    also have ... = lead-coeff p * lead-coeff q ^ (degree p + 1)
      by (auto simp: mult-ac)
    finally show ?thesis by auto
  qed
qed

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
  by (induction p arbitrary: k)
     (auto simp: coeff-pCons coeff-monom-mult pcompose-pCons split: nat.splits)

lemma of-nat-mult-conv-smult: of-nat n * P = smult (of-nat n) P
  by (simp add: monom-0 of-nat-monom)

lemma numeral-mult-conv-smult: numeral n * P = smult (numeral n) P
  by (simp add: numeral-poly)

lemma sum-order-le-degree:
  assumes p ≠ 0
  shows   (∑ x | poly p x = 0. order x p) ≤ degree p
  using assms
proof (induction degree p arbitrary: p rule: less-induct)
  case (less p)
  show ?case
  proof (cases ∃ x. poly p x = 0)
    case False
    thus ?thesis
      by auto
  next
    case True
    then obtain x where x: poly p x = 0
      by auto
    have [:-x, 1:] ^ order x p dvd p
      by (simp add: order-1)
    then obtain q where q: p = [:-x, 1:] ^ order x p * q
      by (elim dvdE)
    have [simp]: q ≠ 0
      using q less.prems by auto
    have order x p = order x p + order x q
      by (subst q, subst order-mult) (auto simp: order-power-n-n)
    hence order x q = 0
      by auto
    hence [simp]: poly q x ≠ 0
```

by (*simp add*: *order-root*)
have *deg-p*: *degree p* = *degree q* + *order x p*
by (*subst q*, *subst degree-mult-eq*) (*auto simp*: *degree-power-eq*)
moreover have *order x p* > *0*
using *x less.prems* **by** (*simp add*: *order-root*)
ultimately have *degree q* < *degree p*
by *linarith*
hence (∑ *x* | *poly q x* = *0*. *order x q*) ≤ *degree q*
by (*intro less.hyps*) *auto*
hence *order x p* + (∑ *x* | *poly q x* = *0*. *order x q*) ≤ *degree p*
by (*simp add*: *deg-p*)
also have {*y*. *poly q y* = *0*} = {*y*. *poly p y* = *0*} − {*x*}
by (*subst q*) *auto*
also have (∑ *y* ∈ {*y*. *poly p y* = *0*} − {*x*}. *order y q*) =
(∑ *y* ∈ {*y*. *poly p y* = *0*} − {*x*}. *order y p*)
by (*intro sum.cong refl*, *subst q*)
(*auto simp*: *order-mult order-power-n-n intro*!: *order-0I*)
also have *order x p* + ... = (∑ *y* ∈ *insert x* ({*y*. *poly p y* = *0*} − {*x*}). *order y p*)
using ‹*p* ≠ *0*› **by** (*subst sum.insert*) (*auto simp*: *poly-roots-finite*)
also have *insert x* ({*y*. *poly p y* = *0*} − {*x*}) = {*y*. *poly p y* = *0*}
using ‹*poly p x* = *0*› **by** *auto*
finally show *?thesis* **.**
qed
qed

4.23 Divisibility

context
assumes *SORT-CONSTRAINT*('*a* :: *idom*)
begin
lemma *poly-linear-linear-factor*:
assumes *dvd*: [:*b*,*1*:] *dvd* (∏ (*a* :: '*a*) ← *as*. [: *a*, *1*:])
shows *b* ∈ *set as*
proof −
let *?p* = λ *as*. (∏ *a* ← *as*. [: *a*, *1*:])
let *?b* = [:*b*,*1*:]
from *assms*[*unfolded dvd-def*] **obtain** *p* **where** *id*: *?p as* = *?b* ∗ *p* **..**
from *arg-cong*[*OF id*, *of* λ *p*. *poly p* (−*b*)]
have *poly* (*?p as*) (−*b*) = *0* **by** *simp*
thus *?thesis*
proof (*induct as*)
case (*Cons a as*)
have *?p* (*a* # *as*) = [:*a*,*1*:] ∗ *?p as* **by** *simp*
from *Cons*(*2*)[*unfolded this*] **have** *poly* (*?p as*) (−*b*) = *0* ∨ (*a* − *b*) = *0* **by** *simp*
with *Cons*(*1*) **show** *?case* **by** *auto*
qed *simp*
qed

lemma *poly-linear-exp-linear-factors*:
assumes *dvd*: $([:b,1:])\hat{}n$ *dvd* $(\prod\ (a :: 'a) \leftarrow as.\ [:\ a,\ 1:])$
shows *length* (*filter* $((=)\ b)$ *as*) $\geq n$
proof −
let *?p* = λ *as*. $(\prod\ a \leftarrow as.\ [:\ a,\ 1:])$
let *?b* = [:*b*,*1*:]
from *dvd* **show** *?thesis*
proof (*induct n arbitrary*: *as*)
case (*Suc n as*)
have *bs*: *?b* $\hat{}$ *Suc n* = *?b* ∗ *?b* $\hat{}$ *n* **by** *simp*
from *poly-linear-linear-factor*[*OF dvd-mult-left*[*OF Suc*(*2*)[*unfolded bs*]],
unfolded in-set-conv-decomp]
obtain *as1 as2* **where** *as*: *as* = *as1* @ *b* # *as2* **by** *auto*
have *?p as* = [:*b*,*1*:] ∗ *?p* (*as1* @ *as2*) **unfolding** *as*
proof (*induct as1*)
case (*Cons a as1*)
have *?p* (*a* # *as1* @ *b* # *as2*) = [:*a*,*1*:] ∗ *?p* (*as1* @ *b* # *as2*) **by** *simp*
also have *?p* (*as1* @ *b* # *as2*) = [:*b*,*1*:] ∗ *?p* (*as1* @ *as2*) **unfolding** *Cons*
by *simp*
also have [:*a*,*1*:] ∗ ... = [:*b*,*1*:] ∗ ([:*a*,*1*:] ∗ *?p* (*as1* @ *as2*))
by (*metis* (*no-types*, *lifting*) *mult.left-commute*)
finally show *?case* **by** *simp*
qed *simp*
from *Suc*(*2*)[*unfolded bs this dvd-mult-cancel-left*]
have *?b* $\hat{}$ *n dvd ?p* (*as1* @ *as2*) **by** *simp*
from *Suc*(*1*)[*OF this*] **show** *?case* **unfolding** *as* **by** *simp*
qed *simp*
qed
end

lemma *const-poly-dvd*: ([:*a*:] *dvd* [:*b*:]) = (*a dvd b*)
proof
assume *a dvd b*
then obtain *c* **where** *b* = *a* ∗ *c* **unfolding** *dvd-def* **by** *auto*
hence [:*b*:] = [:*a*:] ∗ [:*c*:] **by** (*auto simp*: *ac-simps*)
thus [:*a*:] *dvd* [:*b*:] **unfolding** *dvd-def* **by** *blast*
next
assume [:*a*:] *dvd* [:*b*:]
then obtain *pc* **where** [:*b*:] = [:*a*:] ∗ *pc* **unfolding** *dvd-def* **by** *blast*
from *arg-cong*[*OF this*, *of* λ *p*. *coeff p 0*, *unfolded coeff-mult*]
have *b* = *a* ∗ *coeff pc 0* **by** *auto*
thus *a dvd b* **unfolding** *dvd-def* **by** *blast*
qed

lemma *const-poly-dvd-1* [*simp*]:
[:*a*:] *dvd 1* $\longleftrightarrow$ *a dvd 1*
by (*metis const-poly-dvd one-poly-eq-simps*(*2*))

lemma *poly-dvd-1*:
fixes $p :: 'a :: \{comm\text{-}semiring\text{-}1, semiring\text{-}no\text{-}zero\text{-}divisors\}$ *poly*
shows *p dvd 1* $\longleftrightarrow$ *degree p = 0* $\wedge$ *coeff p 0 dvd 1*
proof (*cases degree p = 0*)
case *False*
with *divides-degree*[*of p 1*] **show** *?thesis* **by** *auto*
next
case *True*
then obtain *a* **where** *p*: *p* = [:*a*:]
using *degree-eq-zeroE* **by** *blast*
show *?thesis* **unfolding** *p* **by** *auto*
qed

4.24 Closure properties of coefficients

context
fixes $R :: 'a :: comm\text{-}semiring\text{-}1\ set$
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 −
have *R-sum*: *sum f A* $\in$ *R* **if** $\bigwedge x.\ x \in A \Longrightarrow f\ x \in R$ **for** *A* **and** *f* :: *nat* $\Rightarrow$ $'a$
using *that* **by** (*induction A rule*: *infinite-finite-induct*) (*auto intro*: *R-0 R-plus*)
show *?thesis*
unfolding *coeff-mult* **by** (*auto intro*!: *R-sum R-mult assms*)
qed

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*
using *assms*(*1*)
proof (*induction p arbitrary*: *i*)
case (*pCons a p i*)
have [*simp*]: *a* $\in$ *R*
using *pCons.prems*[*of 0*] **by** *auto*
have *coeff p i* $\in$ *R* **for** *i*
using *pCons.prems*[*of Suc i*] **by** *auto*
hence *coeff* (*p* $\circ_p$ *q*) *i* $\in$ *R* **for** *i*
using *pCons.prems* **by** (*intro pCons.IH*)
thus *?case*
by (*auto simp*: *pcompose-pCons coeff-pCons split*: *nat.splits*
intro!: *assms R-plus coeff-mult-semiring-closed*)
qed *auto*

end

4.25 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*)
by (*auto simp add*: *nth-default-def add-ac*)

lemma *nth-default-take*: *nth-default x* (*take n xs*) *m* = (*if m* < *n then nth-default x xs m else x*)
by (*auto simp add*: *nth-default-def add-ac*)

lemma *coeff-poly-shift*: *coeff* (*poly-shift n p*) *i* = *coeff p* (*i* + *n*)
proof −
from *MOST-coeff-eq-0*[*of p*] **obtain** *m* **where** ∀ *k*>*m*. *coeff p k* = *0*
by (*auto simp*: *MOST-nat*)
then have ∀ *k*>*m*. *coeff p* (*k* + *n*) = *0*
by *auto*
then have ∀ ∞*k*. *coeff p* (*k* + *n*) = *0*
by (*auto simp*: *MOST-nat*)
then show *?thesis*
by (*simp add*: *poly-shift-def poly.Abs-poly-inverse*)
qed

lemma *poly-shift-id* [*simp*]: *poly-shift 0* = (λ*x*. *x*)
by (*simp add*: *poly-eq-iff fun-eq-iff coeff-poly-shift*)

lemma *poly-shift-0* [*simp*]: *poly-shift n 0* = *0*
by (*simp add*: *poly-eq-iff coeff-poly-shift*)

lemma *poly-shift-1*: *poly-shift n 1* = (*if n* = *0 then 1 else 0*)
by (*simp add*: *poly-eq-iff coeff-poly-shift*)

lemma *poly-shift-monom*: *poly-shift n* (*monom c m*) = (*if m* ≥ *n then monom c* (*m* − *n*) *else 0*)
by (*auto simp add*: *poly-eq-iff coeff-poly-shift*)

lemma *coeffs-shift-poly* [*code abstract*]:
coeffs (*poly-shift n p*) = *drop n* (*coeffs p*)
proof (*cases p* = *0*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
then show *?thesis*
by (*intro coeffs-eqI*)
(*simp-all add*: *coeff-poly-shift nth-default-drop nth-default-coeffs-eq*)

qed

4.26 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*)
unfolding *poly-cutoff-def*
by (*subst poly.Abs-poly-inverse*) (*auto simp*: *MOST-nat intro*: *exI*[*of - n*])

lemma *poly-cutoff-0* [*simp*]: *poly-cutoff n 0 = 0*
by (*simp add*: *poly-eq-iff coeff-poly-cutoff*)

lemma *poly-cutoff-1* [*simp*]: *poly-cutoff n 1* = (*if n = 0 then 0 else 1*)
by (*simp add*: *poly-eq-iff coeff-poly-cutoff*)

lemma *coeffs-poly-cutoff* [*code abstract*]:
coeffs (*poly-cutoff n p*) = *strip-while* ((=) *0*) (*take n* (*coeffs p*))
proof (*cases strip-while* ((=) *0*) (*take n* (*coeffs p*)) = [])
case *True*
then have *coeff* (*poly-cutoff n p*) *k = 0* **for** *k*
unfolding *coeff-poly-cutoff*
by (*auto simp*: *nth-default-coeffs-eq* [*symmetric*] *nth-default-def set-conv-nth*)
then have *poly-cutoff n p = 0*
by (*simp add*: *poly-eq-iff*)
then show *?thesis*
by (*subst True*) *simp-all*
next
case *False*
have *no-trailing* ((=) *0*) (*strip-while* ((=) *0*) (*take n* (*coeffs p*)))
by *simp*
with *False* **have** *last* (*strip-while* ((=) *0*) (*take n* (*coeffs p*))) ≠ *0*
unfolding *no-trailing-unfold* **by** *auto*
then show *?thesis*
by (*intro coeffs-eqI*)
(*simp-all add*: *coeff-poly-cutoff nth-default-take nth-default-coeffs-eq*)
qed

4.27 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*))
by (*simp add*: *reflect-poly-def*)

lemma *reflect-poly-0* [*simp*]: *reflect-poly 0 = 0*

by (*simp add*: *reflect-poly-def*)

lemma *reflect-poly-1* [*simp*]: *reflect-poly 1 = 1*
by (*simp add*: *reflect-poly-def one-pCons*)

lemma *coeff-reflect-poly*:
coeff (*reflect-poly p*) *n* = (*if n* > *degree p then 0 else coeff p* (*degree p* − *n*))
by (*cases p* = *0*)
(*auto simp add*: *reflect-poly-def nth-default-def*
rev-nth degree-eq-length-coeffs coeffs-nth not-less
dest: *le-imp-less-Suc*)

lemma *coeff-0-reflect-poly-0-iff* [*simp*]: *coeff* (*reflect-poly p*) *0* = *0* ⟷ *p* = *0*
by (*simp add*: *coeff-reflect-poly*)

lemma *reflect-poly-at-0-eq-0-iff* [*simp*]: *poly* (*reflect-poly p*) *0* = *0* ⟷ *p* = *0*
by (*simp add*: *coeff-reflect-poly poly-0-coeff-0*)

lemma *reflect-poly-pCons'*:
p ≠ *0* ⟹ *reflect-poly* (*pCons c p*) = *reflect-poly p* + *monom c* (*Suc* (*degree p*))
by (*intro poly-eqI*)
(*auto simp*: *coeff-reflect-poly coeff-pCons not-less Suc-diff-le split*: *nat.split*)

lemma *reflect-poly-const* [*simp*]: *reflect-poly* [:*a*:] = [:*a*:]
by (*cases a* = *0*) (*simp-all add*: *reflect-poly-def*)

lemma *poly-reflect-poly-nz*:
x ≠ *0* ⟹ *poly* (*reflect-poly p*) *x* = *x* ^ *degree p* ∗ *poly p* (*inverse x*)
for *x* :: *'a*::*field*
by (*induct rule*: *pCons-induct*) (*simp-all add*: *field-simps reflect-poly-pCons' poly-monom*)

lemma *coeff-0-reflect-poly* [*simp*]: *coeff* (*reflect-poly p*) *0* = *lead-coeff p*
by (*simp add*: *coeff-reflect-poly*)

lemma *poly-reflect-poly-0* [*simp*]: *poly* (*reflect-poly p*) *0* = *lead-coeff p*
by (*simp add*: *poly-0-coeff-0*)

lemma *reflect-poly-reflect-poly* [*simp*]: *coeff p 0* ≠ *0* ⟹ *reflect-poly* (*reflect-poly p*) = *p*
by (*cases p rule*: *pCons-cases*) (*simp add*: *reflect-poly-def*)

lemma *degree-reflect-poly-le*: *degree* (*reflect-poly p*) ≤ *degree p*
by (*simp add*: *degree-eq-length-coeffs coeffs-reflect-poly length-dropWhile-le diff-le-mono*)

lemma *reflect-poly-pCons*: *a* ≠ *0* ⟹ *reflect-poly* (*pCons a p*) = *Poly* (*rev* (*a* # *coeffs p*))
by (*subst coeffs-eq-iff*) (*simp add*: *coeffs-reflect-poly*)

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

```
degree p
 by (cases p rule: pCons-cases) (simp add: reflect-poly-pCons degree-eq-length-coeffs)

lemma reflect-poly-eq-0-iff [simp]: reflect-poly p = 0 ⟷ p = 0
  using coeff-0-reflect-poly-0-iff by fastforce


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 (cases p = 0 ∨ q = 0)
  case False
  then have [simp]: p ≠ 0 q ≠ 0 by auto
  show ?thesis
  proof (rule poly-eqI)
    show coeff (reflect-poly (p * q)) i = coeff (reflect-poly p * reflect-poly q) i for i
    proof (cases i ≤ degree (p * q))
      case True
      define A where A = {..i} ∩ {i - degree q..degree p}
      define B where B = {..degree p} ∩ {degree p - i..degree (p*q) - i}
      let ?f = λj. degree p - j

      from True have coeff (reflect-poly (p * q)) i = coeff (p * q) (degree (p * q)
- i)
        by (simp add: coeff-reflect-poly)
      also have ... = (∑j≤degree (p * q) - i. coeff p j * coeff q (degree (p * q)
- i - j))
        by (simp add: coeff-mult)
      also have ... = (∑j∈B. coeff p j * coeff q (degree (p * q) - i - j))
        by (intro sum.mono-neutral-right) (auto simp: B-def degree-mult-eq not-le
coeff-eq-0)
      also from True have ... = (∑j∈A. coeff p (degree p - j) * coeff q (degree
q - (i - j)))
        by (intro sum.reindex-bij-witness[of - ?f ?f])
          (auto simp: A-def B-def degree-mult-eq add-ac)
      also have ... =
        (∑j≤i.
          if j ∈ {i - degree q..degree p}
          then coeff p (degree p - j) * coeff q (degree q - (i - j))
          else 0)
        by (subst sum.inter-restrict [symmetric]) (simp-all add: A-def)
      also have ... = coeff (reflect-poly p * reflect-poly q) i
        by (fastforce simp: coeff-mult coeff-reflect-poly intro!: sum.cong)
      finally show ?thesis .
    qed (auto simp: coeff-mult coeff-reflect-poly coeff-eq-0 degree-mult-eq intro!:
sum.neutral)
  qed
qed auto

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*
using *reflect-poly-mult*[*of* [:*c*:] *p*] **by** *simp*

lemma *reflect-poly-power*: *reflect-poly* (p ^ n) = *reflect-poly* p ^ n
for p :: $'a$::{*comm-semiring-1*,*semiring-no-zero-divisors*} *poly*
by (*induct n*) (*simp-all add*: *reflect-poly-mult*)

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*
by (*induct A rule*: *infinite-finite-induct*) (*simp-all add*: *reflect-poly-mult*)

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*
by (*induct xs*) (*simp-all add*: *reflect-poly-mult*)

lemma *reflect-poly-Poly-nz*:
no-trailing (*HOL.eq 0*) *xs* ⟹ *reflect-poly* (*Poly xs*) = *Poly* (*rev xs*)
by (*simp add*: *reflect-poly-def*)

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

4.28 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*))
by (*auto intro*: *pCons-cases*)

termination *pderiv*
by (*relation measure degree*) *simp-all*

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

lemma *pderiv-0* [*simp*]: *pderiv 0* = *0*
using *pderiv.simps* [*of 0 0*] **by** *simp*

lemma *pderiv-pCons*: *pderiv* (*pCons a p*) = *p* + *pCons 0* (*pderiv p*)
by (*simp add*: *pderiv.simps*)

lemma *pderiv-1* [*simp*]: *pderiv 1* = *0*
by (*simp add*: *one-pCons pderiv-pCons*)

lemma *pderiv-of-nat* [*simp*]: *pderiv* (*of-nat n*) = *0*
and *pderiv-numeral* [*simp*]: *pderiv* (*numeral m*) = *0*
by (*simp-all add*: *of-nat-poly numeral-poly pderiv-pCons*)

```
lemma coeff-pderiv: coeff (pderiv p) n = of-nat (Suc n) * coeff p (Suc n)
  by (induct p arbitrary: n)
    (auto simp add: pderiv-pCons coeff-pCons algebra-simps split: nat.split)

fun pderiv-coeffs-code :: 'a::{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 (induct xs arbitrary: f n)
  case Nil
  then show ?case by simp
next
  case (Cons x xs)
  show ?case
  proof (cases n)
    case 0
    then show ?thesis
     by (cases pderiv-coeffs-code (f + 1) xs = [] ∧ f * x = 0) (auto simp: cCons-def)
  next
    case n: (Suc m)
    show ?thesis
    proof (cases pderiv-coeffs-code (f + 1) xs = [] ∧ f * x = 0)
      case False
      then have nth-default 0 (pderiv-coeffs-code f (x # xs)) n =
          nth-default 0 (pderiv-coeffs-code (f + 1) xs) m
        by (auto simp: cCons-def n)
      also have ... = (f + of-nat n) * nth-default 0 xs m
        by (simp add: Cons n add-ac)
      finally show ?thesis
        by (simp add: n)
    next
      case True
      have empty: pderiv-coeffs-code g xs = [] ⟹ g + of-nat m = 0 ∨ nth-default
0 xs m = 0 for g
      proof (induct xs arbitrary: g m)
        case Nil
        then show ?case by simp
      next
        case (Cons x xs)
        from Cons(2) have empty: pderiv-coeffs-code (g + 1) xs = [] and g: g =
```

```
0 ∨ x = 0
          by (auto simp: cCons-def split: if-splits)
        note IH = Cons(1)[OF empty]
        from IH[of m] IH[of m − 1] g show ?case
          by (cases m) (auto simp: field-simps)
      qed
      from True have nth-default 0 (pderiv-coeffs-code f (x # xs)) n = 0
        by (auto simp: cCons-def n)
      moreover from True have (f + of-nat n) ∗ nth-default 0 (x # xs) n = 0
        by (simp add: n) (use empty[of f+1] in ‹auto simp: field-simps›)
      ultimately show ?thesis by simp
    qed
  qed
qed

lemma coeffs-pderiv-code [code abstract]: coeffs (pderiv p) = pderiv-coeffs (coeffs
p)
  unfolding pderiv-coeffs-def
proof (rule coeffs-eqI, unfold pderiv-coeffs-code coeff-pderiv, goal-cases)
  case (1 n)
  have id: coeff p (Suc n) = nth-default 0 (map (λi. coeff p (Suc i)) [0..<degree
p]) n
    by (cases n < degree p) (auto simp: nth-default-def coeff-eq-0)
  show ?case
    unfolding coeffs-def map-upt-Suc by (auto simp: id)
next
  case 2
  obtain n :: 'a and xs where defs: tl (coeffs p) = xs 1 = n
    by simp
  from 2 show ?case
    unfolding defs by (induct xs arbitrary: n) (auto simp: cCons-def)
qed

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 (cases degree p)
  case 0
  then show ?thesis
    by (metis degree-eq-zeroE pderiv.simps)
next
  case (Suc n)
  then show ?thesis
   using coeff-0 coeff-pderiv degree-0 leading-coeff-0-iff mult-eq-0-iff nat.distinct(1)
of-nat-eq-0-iff
   by (metis coeff-0 coeff-pderiv degree-0 leading-coeff-0-iff mult-eq-0-iff nat.distinct(1)
of-nat-eq-0-iff)
qed

lemma degree-pderiv: degree (pderiv p) = degree p − 1
```

for $p :: 'a::\{comm\text{-}semiring\text{-}1,semiring\text{-}no\text{-}zero\text{-}divisors,semiring\text{-}char\text{-}0\}$ *poly*
proof −
have *degree p − 1 ≤ degree* (*pderiv p*)
proof (*cases degree p*)
case (*Suc n*)
then show *?thesis*
by (*metis coeff-pderiv degree-0 diff-Suc-1 le-degree leading-coeff-0-iff mult-eq-0-iff nat.distinct*(1) *of-nat-eq-0-iff*)
qed *auto*
moreover have ∀ *i*>*degree p − 1*. *coeff* (*pderiv p*) *i* = *0*
by (*simp add*: *coeff-eq-0 coeff-pderiv*)
ultimately show *?thesis*
using *order-antisym* [*OF degree-le*] **by** *blast*
qed

lemma *not-dvd-pderiv*:
fixes $p :: 'a::\{comm\text{-}semiring\text{-}1,semiring\text{-}no\text{-}zero\text{-}divisors,semiring\text{-}char\text{-}0\}$ *poly*
assumes *degree p ≠ 0*
shows ¬ *p dvd pderiv p*
proof
assume *dvd*: *p dvd pderiv p*
then obtain *q* **where** *p*: *pderiv p* = *p* ∗ *q*
unfolding *dvd-def* **by** *auto*
from *dvd* **have** *le*: *degree p ≤ degree* (*pderiv p*)
by (*simp add*: *assms dvd-imp-degree-le pderiv-eq-0-iff*)
from *assms* **and** *this* [*unfolded degree-pderiv*]
show *False* **by** *auto*
qed

lemma *dvd-pderiv-iff* [*simp*]: *p dvd pderiv p* ⟷ *degree p* = *0*
for $p :: 'a::\{comm\text{-}semiring\text{-}1,semiring\text{-}no\text{-}zero\text{-}divisors,semiring\text{-}char\text{-}0\}$ *poly*
using *not-dvd-pderiv*[*of p*] **by** (*auto simp*: *pderiv-eq-0-iff* [*symmetric*])

lemma *pderiv-singleton* [*simp*]: *pderiv* [:*a*:] = *0*
by (*simp add*: *pderiv-pCons*)

lemma *pderiv-add*: *pderiv* (*p* + *q*) = *pderiv p* + *pderiv q*
by (*rule poly-eqI*) (*simp add*: *coeff-pderiv algebra-simps*)

lemma *pderiv-minus*: *pderiv* (− *p* :: *'a* :: *idom poly*) = − *pderiv p*
by (*rule poly-eqI*) (*simp add*: *coeff-pderiv algebra-simps*)

lemma *pderiv-diff*: *pderiv* ((*p* :: - :: *idom poly*) − *q*) = *pderiv p* − *pderiv q*
by (*rule poly-eqI*) (*simp add*: *coeff-pderiv algebra-simps*)

lemma *pderiv-smult*: *pderiv* (*smult a p*) = *smult a* (*pderiv p*)
by (*rule poly-eqI*) (*simp add*: *coeff-pderiv algebra-simps*)

lemma *pderiv-mult*: *pderiv* (*p* ∗ *q*) = *p* ∗ *pderiv q* + *q* ∗ *pderiv p*

by (*induct p*) (*auto simp*: *pderiv-add pderiv-smult pderiv-pCons algebra-simps*)

lemma *pderiv-power-Suc*: *pderiv* (*p* ^ *Suc n*) = *smult* (*of-nat* (*Suc n*)) (*p* ^ *n*) ∗ *pderiv p*
proof (*induction n*)
case (*Suc n*)
then show *?case*
by (*simp add*: *pderiv-mult smult-add-left algebra-simps*)
qed *auto*

lemma *pderiv-power*:
pderiv (*p* ^ *n*) = *smult* (*of-nat n*) (*p* ^ (*n* − *1*) ∗ *pderiv p*)
by (*cases n*) (*simp-all add*: *pderiv-power-Suc del*: *power-Suc*)

lemma *pderiv-monom*:
pderiv (*monom c n*) = *monom* (*of-nat n* ∗ *c*) (*n* − *1*)
by (*cases n*)
(*simp-all add*: *monom-altdef pderiv-power-Suc pderiv-smult pderiv-pCons mult-ac del*: *power-Suc*)

lemma *pderiv-pcompose*: *pderiv* (*pcompose p q*) = *pcompose* (*pderiv p*) *q* ∗ *pderiv q*
by (*induction p rule*: *pCons-induct*)
(*auto simp*: *pcompose-pCons pderiv-add pderiv-mult pderiv-pCons pcompose-add algebra-simps*)

lemma *pderiv-prod*: *pderiv* (*prod f* (*as*)) = (∑ *a*∈*as*. *prod f* (*as* − {*a*}) ∗ *pderiv* (*f a*))
proof (*induct as rule*: *infinite-finite-induct*)
case (*insert a as*)
then have *id*: *prod f* (*insert a as*) = *f a* ∗ *prod f as*
⋀*g*. *sum g* (*insert a as*) = *g a* + *sum g as*
insert a as − {*a*} = *as*
by *auto*
have *prod f* (*insert a as* − {*b*}) = *f a* ∗ *prod f* (*as* − {*b*}) **if** *b* ∈ *as* **for** *b*
proof −
from ‹*a* ∉ *as*› *that* **have** ∗: *insert a as* − {*b*} = *insert a* (*as* − {*b*})
by *auto*
show *?thesis*
unfolding ∗ **by** (*subst prod.insert*) (*use insert* **in** *auto*)
qed
then show *?case*
unfolding *id pderiv-mult insert*(*3*) *sum-distrib-left*
by (*auto simp add*: *ac-simps intro*!: *sum.cong*)
qed *auto*

lemma *lead-coeff-pderiv*:
fixes *p* :: *'a*::{*comm-semiring-1*,*semiring-no-zero-divisors*,*semiring-char-0*} *poly*
shows *lead-coeff* (*pderiv p*) = *of-nat* (*degree p*) ∗ *lead-coeff p*

proof (*cases degree p*)
case *0*
then show *?thesis*
by (*simp add*: *pderiv-eq-0-iff*)
next
case (*Suc nat*)
then show *?thesis*
by (*simp add*: *coeff-pderiv degree-pderiv*)
qed

lemma *coeff-higher-pderiv*:
coeff ((*pderiv* ^^ *m*) *f*) *n* = *pochhammer* (*of-nat* (*Suc n*)) *m* ∗ *coeff f* (*n* + *m*)
by (*induction m arbitrary*: *n*) (*simp-all add*: *coeff-pderiv pochhammer-rec algebra-simps*)

lemma *higher-pderiv-0* [*simp*]: (*pderiv* ^^ *n*) *0* = *0*
by (*induction n*) *simp-all*

lemma *higher-pderiv-add*: (*pderiv* ^^ *n*) (*p* + *q*) = (*pderiv* ^^ *n*) *p* + (*pderiv* ^^ *n*) *q*
by (*induction n arbitrary*: *p q*) (*simp-all del*: *funpow.simps add*: *funpow-Suc-right pderiv-add*)

lemma *higher-pderiv-smult*: (*pderiv* ^^ *n*) (*smult c p*) = *smult c* ((*pderiv* ^^ *n*) *p*)
by (*induction n arbitrary*: *p*) (*simp-all del*: *funpow.simps add*: *funpow-Suc-right pderiv-smult*)

lemma *higher-pderiv-monom*:
m ≤ *n* + *1* ⟹ (*pderiv* ^^ *m*) (*monom c n*) = *monom* (*pochhammer* (*int n* − *int m* + *1*) *m* ∗ *c*) (*n* − *m*)
proof (*induction m arbitrary*: *c n*)
case (*Suc m*)
thus *?case*
by (*cases n*)
(*simp-all del*: *funpow.simps add*: *funpow-Suc-right pderiv-monom pochhammer-rec′ Suc.IH*)
qed *simp-all*

lemma *higher-pderiv-monom-eq-zero*:
m > *n* + *1* ⟹ (*pderiv* ^^ *m*) (*monom c n*) = *0*
proof (*induction m arbitrary*: *c n*)
case (*Suc m*)
thus *?case*
by (*cases n*)
(*simp-all del*: *funpow.simps add*: *funpow-Suc-right pderiv-monom pochhammer-rec′ Suc.IH*)
qed *simp-all*

lemma *higher-pderiv-sum*: (*pderiv* ^^ *n*) (*sum f A*) = (∑ *x*∈*A*. (*pderiv* ^^ *n*) (*f*

x))
by (*induction A rule*: *infinite-finite-induct*) (*simp-all add*: *higher-pderiv-add*)

lemma *higher-pderiv-sum-mset*: (*pderiv* ^^ *n*) (*sum-mset A*) = (∑ *p*∈#*A*. (*pderiv* ^^ *n*) *p*)
by (*induction A*) (*simp-all add*: *higher-pderiv-add*)

lemma *higher-pderiv-sum-list*: (*pderiv* ^^ *n*) (*sum-list ps*) = (∑ *p*←*ps*. (*pderiv* ^^ *n*) *p*)
by (*induction ps*) (*simp-all add*: *higher-pderiv-add*)

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*
by (*induction n*) (*auto simp*: *degree-pderiv*)

lemma *DERIV-pow2*: *DERIV* (λ*x*. *x* ^ *Suc n*) *x* :> *real* (*Suc n*) ∗ (*x* ^ *n*)
by (*rule DERIV-cong*, *rule DERIV-pow*) *simp*
declare *DERIV-pow2* [*simp*] *DERIV-pow* [*simp*]

lemma *DERIV-add-const*: *DERIV f x* :> *D* ⟹ *DERIV* (λ*x*. *a* + *f x* :: *'a*::*real-normed-field*) *x* :> *D*
by (*rule DERIV-cong*, *rule DERIV-add*) *auto*

lemma *poly-DERIV* [*simp*]: *DERIV* (λ*x*. *poly p x*) *x* :> *poly* (*pderiv p*) *x*
by (*induct p*) (*auto intro*!: *derivative-eq-intros simp add*: *pderiv-pCons*)

lemma *poly-isCont*[*simp*]:
fixes *x*::*'a*::*real-normed-field*
shows *isCont* (λ*x*. *poly p x*) *x*
by (*rule poly-DERIV* [*THEN DERIV-isCont*])

lemma *tendsto-poly* [*tendsto-intros*]: (*f* ⟶ *a*) *F* ⟹ ((λ*x*. *poly p* (*f x*)) ⟶ *poly p a*) *F*
for *f* :: - ⇒ *'a*::*real-normed-field*
by (*rule isCont-tendsto-compose* [*OF poly-isCont*])

lemma *continuous-within-poly*: *continuous* (*at z within s*) (*poly p*)
for *z* :: *'a*::{*real-normed-field*}
by (*simp add*: *continuous-within tendsto-poly*)

lemma *continuous-poly* [*continuous-intros*]: *continuous F f* ⟹ *continuous F* (λ*x*. *poly p* (*f x*))
for *f* :: - ⇒ *'a*::*real-normed-field*
unfolding *continuous-def* **by** (*rule tendsto-poly*)

lemma *continuous-on-poly* [*continuous-intros*]:
fixes *p* :: *'a* :: {*real-normed-field*} *poly*

assumes *continuous-on A f*
shows *continuous-on A* (λx. *poly p* (*f x*))
by (*metis DERIV-continuous-on assms continuous-on-compose2 poly-DERIV subset-UNIV*)

Consequences of the derivative theorem above.

lemma *poly-differentiable*[*simp*]: (λx. *poly p x*) *differentiable* (*at x*)
for *x* :: *real*
by (*simp add*: *real-differentiable-def*) (*blast intro*: *poly-DERIV*)

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*
using *IVT* [*of poly p a 0 b*] **by** (*auto simp add*: *order-le-less*)

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*
using *poly-IVT-pos* [**where** $p = -p$] **by** *simp*

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*
by (*metis less-not-sym mult-less-0-iff poly-IVT-neg poly-IVT-pos*)

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*
by (*simp add*: *MVT2*)

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 (*cases a b rule*: *linorder-cases*)
case *less*
from *poly-MVT*[*OF less*, *of p*] **obtain** *x*
where *a* $< x\ x <$ *b poly p b* $-$ *poly p a* $= (b - a) *$ *poly* (*pderiv p*) *x*
by *auto*
then show *?thesis* **by** (*intro bexI*[*of - x*]) (*auto intro*!: *subsetD*[*OF assms*])
next
case *greater*
from *poly-MVT*[*OF greater*, *of p*] **obtain** *x*
where *b* $< x\ x <$ *a poly p a* $-$ *poly p b* $= (a - b) *$ *poly* (*pderiv p*) *x* **by** *auto*
then show *?thesis* **by** (*intro bexI*[*of - x*]) (*auto simp*: *algebra-simps intro*!: *subsetD*[*OF assms*])
qed (*use assms* **in** *auto*)

lemma *poly-pinfty-gt-lc*:

```
  fixes p :: real poly
  assumes lead-coeff p > 0
  shows ∃ n. ∀ x ≥ n. poly p x ≥ lead-coeff p
  using assms
proof (induct p)
  case 0
  then show ?case by auto
next
  case (pCons a p)
  from this(1) consider a ≠ 0 p = 0 | p ≠ 0 by auto
  then show ?case
  proof cases
    case 1
    then show ?thesis by auto
  next
    case 2
    with pCons obtain n1 where gte-lcoeff: ∀ x≥n1. lead-coeff p ≤ poly p x
      by auto
    from pCons(3) ‹p ≠ 0› have gt-0: lead-coeff p > 0 by auto
    define n where n = max n1 (1 + |a| / lead-coeff p)
    have lead-coeff (pCons a p) ≤ poly (pCons a p) x if n ≤ x for x
    proof −
      from gte-lcoeff that have lead-coeff p ≤ poly p x
        by (auto simp: n-def)
      with gt-0 have |a| / lead-coeff p ≥ |a| / poly p x and poly p x > 0
        by (auto intro: frac-le)
      with ‹n ≤ x›[unfolded n-def] have x ≥ 1 + |a| / poly p x
        by auto
      with ‹lead-coeff p ≤ poly p x› ‹poly p x > 0› ‹p ≠ 0›
      show lead-coeff (pCons a p) ≤ poly (pCons a p) x
        by (auto simp: field-simps)
    qed
    then show ?thesis by blast
  qed
qed

lemma dvd-monic:
  fixes p q:: 'a :: idom poly
  assumes monic:lead-coeff p=1 and p dvd (smult c q) and c≠0
  shows p dvd q using assms
proof (cases q=0 ∨ degree p=0)
  case True
  thus ?thesis using assms
    by (auto elim!: degree-eq-zeroE simp add: const-poly-dvd-iff)
next
  case False
  hence q≠0 and degree p≠0 by auto
  obtain k where k:smult c q = p*k using assms dvd-def by metis
  hence k≠0 by (metis False assms(3) mult-zero-right smult-eq-0-iff)
```

hence *deg-eq*:*degree q*=*degree p* + *degree k*
by (*metis False assms*(*3*) *degree-0 degree-mult-eq degree-smult-eq k*)
have *c-dvd*:∀ *n*≤*degree k*. *c dvd coeff k* (*degree k* − *n*)
proof (*rule*,*rule*)
fix *n* **assume** *n* ≤ *degree k*
thus *c dvd coeff k* (*degree k* − *n*)
proof (*induct n rule*:*nat-less-induct*)
case (*1 n*)
define *T* **where** *T*≡(λ*i*. *coeff p i* ∗ *coeff k* (*degree p*+*degree k* − *n* − *i*))
have *c* ∗ *coeff q* (*degree q* − *n*) = (∑ *i*≤*degree q* − *n*. *coeff p i* ∗ *coeff k* (*degree q* − *n* − *i*))
using *coeff-mult*[*of p k degree q* − *n*] *k coeff-smult*[*of c q degree q* −*n*] **by** *auto*
also have ...=(∑ *i*≤*degree p*+*degree k* − *n*. *T i*)
using *deg-eq* **unfolding** *T-def* **by** *auto*
also have ...=(∑ *i*∈{*0*..<*degree p*}. *T i*) + *sum T* {(*degree p*)}+
sum T {*degree p* + *1*..*degree p* + *degree k* − *n*}
proof −
define *C* **where** *C*≡{{*0*..<*degree p*}, {*degree p*},{*degree p*+*1*..*degree p*+*degree k*−*n*}}
have ∀ *A*∈*C*. *finite A* **unfolding** *C-def* **by** *auto*
moreover have ∀ *A*∈*C*. ∀ *B*∈*C*. *A* ≠ *B* ⟶ *A* ∩ *B* = {}
unfolding *C-def* **by** *auto*
ultimately have *sum T* (⋃ *C*) = *sum* (*sum T*) *C*
using *sum.Union-disjoint* **by** *auto*
moreover have ⋃ *C*={..*degree p* + *degree k* − *n*}
using ‹*n* ≤ *degree k*› **unfolding** *C-def* **by** *auto*
moreover have *sum* (*sum T*) *C*= *sum T* {*0*..<*degree p*} + *sum T* {(*degree p*)} +
sum T {*degree p* + *1*..*degree p* + *degree k* − *n*}
proof −
have {*0*..<*degree p*}≠{*degree p*}
by (*metis atLeast0LessThan insertI1 lessThan-iff less-imp-not-eq*)
moreover have {*degree p*}≠{*degree p* + *1*..*degree p* + *degree k* − *n*}
by (*metis add.commute add-diff-cancel-right*′ *atLeastAtMost-singleton-iff diff-self-eq-0 eq-imp-le not-one-le-zero*)
moreover have {*0*..<*degree p*}≠{*degree p* + *1*..*degree p* + *degree k* − *n*}
using ‹*degree k*≥*n*› ‹*degree p*≠*0*› **by** *fastforce*
ultimately show *?thesis* **unfolding** *C-def* **by** *auto*
qed
ultimately show *?thesis* **by** *auto*
qed
also have ...=(∑ *i*∈{*0*..<*degree p*}. *T i*) + *coeff k* (*degree k* − *n*)
proof −
have ∀ *x*∈{*degree p* + *1*..*degree p* + *degree k* − *n*}. *T x*=*0*
using *coeff-eq-0*[*of p*] **unfolding** *T-def* **by** *simp*
hence *sum T* {*degree p* + *1*..*degree p* + *degree k* − *n*}=*0* **by** *auto*
moreover have *T* (*degree p*)=*coeff k* (*degree k* − *n*)
using *monic* **by** (*simp add*: *T-def*)

```
      ultimately show ?thesis by auto
    qed
    finally have c-coeff: c * coeff q (degree q - n) = sum T {0..<degree p}
          + coeff k (degree k - n) .
    moreover have n≠0⟹c dvd sum T {0..<degree p}
    proof (rule dvd-sum)
      fix i assume i:i ∈ {0..<degree p} and n≠0
      hence (n+i-degree p)≤degree k using ‹n ≤ degree k› by auto
      moreover have n + i - degree p <n using i ‹n≠0› by auto
      ultimately have c dvd coeff k (degree k - (n+i-degree p))
        using 1(1) by auto
      hence c dvd coeff k (degree p + degree k - n - i)
        by (metis add-diff-cancel-left' deg-eq diff-diff-left dvd-0-right le-degree
          le-diff-conv add.commute ordered-cancel-comm-monoid-diff-class.diff-diff-right)
      thus c dvd T i unfolding T-def by auto
    qed
    moreover have n=0 ⟹?case
    proof -
      assume n=0
      hence ∀ i∈{0..<degree p}. coeff k (degree p + degree k - n - i) =0
        using coeff-eq-0[of k] by simp
      hence c * coeff q (degree q - n) = coeff k (degree k - n)
        using c-coeff unfolding T-def by auto
      thus ?thesis by (metis dvdI)
    qed
    ultimately show ?case by (metis dvd-add-right-iff dvd-triv-left)
  qed
 qed
 hence ∀ n. c dvd coeff k n
  by (metis diff-diff-cancel dvd-0-right le-add2 le-add-diff-inverse le-degree)
 then obtain f where f:∀ n. c * f n=coeff k n unfolding dvd-def by metis
 have ∀∞ n. f n = 0
  by (metis (mono-tags, lifting) MOST-coeff-eq-0 MOST-mono assms(3) f mult-eq-0-iff)
 hence smult c (Abs-poly f)=k
  using f smult.abs-eq[of c Abs-poly f] Abs-poly-inverse[of f] coeff-inverse[of k]
  by simp
 hence q=p* Abs-poly f using k ‹c≠0› smult-cancel by auto
 thus ?thesis unfolding dvd-def by auto
qed

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)
 unfolding pderiv-mult pderiv-power-Suc
 by (simp del: power-Suc of-nat-Suc add: pderiv-pCons)

lemma order-pderiv:
 fixes p::'a::{idom,semiring-char-0} poly
 assumes p≠0 poly p x = 0
```

```
  shows order x p = Suc (order x (pderiv p)) using assms
proof −
  define xx op where xx=[:− x, 1:] and op = order x p
  have op ≠ 0 unfolding op-def using assms order-root by blast
  obtain pp where pp:p = xx ^ op ∗ pp ¬ xx dvd pp
    using order-decomp[OF ‹p≠0›,of x,folded xx-def op-def] by auto
  have p-der:pderiv p = smult (of-nat op) (xx^(op −1)) ∗ pp + xx^op∗pderiv pp
    unfolding pp(1) by (auto simp:pderiv-mult pderiv-power xx-def algebra-simps
pderiv-pCons)
  have xx^(op −1) dvd (pderiv p)
    unfolding p-der
    by (metis ‹op ≠ 0› dvd-add-left-iff dvd-mult2 dvd-refl dvd-smult dvd-triv-right
        power-eq-if)
  moreover have ¬ xx^op dvd (pderiv p)
  proof
    assume xx ^ op dvd pderiv p
    then have xx ^ op dvd smult (of-nat op) (xx^(op −1) ∗ pp)
      unfolding p-der by (simp add: dvd-add-left-iff)
    then have xx ^ op dvd (xx^(op −1)) ∗ pp
      apply (elim dvd-monic[rotated])
      using ‹op≠0› by (auto simp:lead-coeff-power xx-def)
    then have xx ^ (op−1) ∗ xx dvd (xx^(op −1))
      using ‹¬ xx dvd pp› by (simp add: ‹op ≠ 0› mult.commute power-eq-if)
    then have xx dvd 1
      using assms(1) pp(1) by auto
    then show False unfolding xx-def by (meson assms(1) dvd-trans one-dvd
order-decomp)
  qed
  ultimately have op − 1 = order x (pderiv p)
    using order-unique-lemma[of x op−1 pderiv p,folded xx-def] ‹op≠0›
    by auto
  then show ?thesis using ‹op≠0› unfolding op-def by auto
qed

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))
  by (metis add.right-neutral gr0-conv-Suc n nat.case nd order-mult order-pderiv
      order-power-n-n order-root pd pderiv-0 pe poly-eq-0-iff-dvd)

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 (*rule classical*)
assume *1*: $\neg$ *?thesis*
from ‹$pderiv\ p \neq 0$› **have** $p \neq 0$ **by** *auto*
with *p* **have** $order\ a\ p = order\ a\ q + order\ a\ d$
by (*simp add*: *order-mult*)
with *1* **have** $order\ a\ p \neq 0$
by (*auto split*: *if-splits*)
from ‹$pderiv\ p \neq 0$› ‹$pderiv\ p = e * d$› **have** *oapp*: $order\ a\ (pderiv\ p) = order\ a\ e + order\ a\ d$
by (*simp add*: *order-mult*)
from ‹$pderiv\ p \neq 0$› ‹$order\ a\ p \neq 0$› **have** *oap*: $order\ a\ p = Suc\ (order\ a\ (pderiv\ p))$
using ‹$p \neq 0$› *order-pderiv order-root* **by** *blast*
from ‹$p \neq 0$› ‹$p = q * d$› **have** $d \neq 0$
by *simp*
have $[:-\ a,\ 1:]\ \widehat{}\ order\ a\ (pderiv\ p)\ dvd\ r * p$
by (*metis dvd-trans dvd-triv-right oap order-1 power-Suc*)
then have $([:-a,\ 1:]\ \widehat{}\ (order\ a\ (pderiv\ p)))\ dvd\ d$
by (*simp add*: *d order-1*)
with ‹$d \neq 0$› **have** $order\ a\ (pderiv\ p) \leq order\ a\ d$
by (*simp add*: *order-divides*)
show *?thesis*
using ‹$order\ a\ p = order\ a\ q + order\ a\ d$›
and *oapp oap*
and ‹$order\ a\ (pderiv\ p) \leq order\ a\ d$›
by *auto*
qed

lemma *poly-squarefree-decomp-order2*:
$pderiv\ p \neq 0 \Longrightarrow p = q * d \Longrightarrow pderiv\ p = e * d \Longrightarrow$
$d = r * p + s * pderiv\ p \Longrightarrow \forall a.\ order\ a\ q = (if\ order\ a\ p = 0\ then\ 0\ else\ 1)$
for *p* :: *'a*::*field-char-0 poly*
by (*blast intro*: *poly-squarefree-decomp-order*)

lemma *order-pderiv2*:
$pderiv\ p \neq 0 \Longrightarrow order\ a\ p \neq 0 \Longrightarrow order\ a\ (pderiv\ p) = n \longleftrightarrow order\ a\ p = Suc\ n$
for *p* :: *'a*::*field-char-0 poly*
by (*metis nat.inject order-pderiv order-root pderiv-0*)

definition *rsquarefree* :: *'a*::*idom poly* $\Rightarrow$ *bool*
where $rsquarefree\ p \longleftrightarrow p \neq 0 \wedge (\forall a.\ order\ a\ p = 0 \vee order\ a\ p = 1)$

lemma *pderiv-iszero*: $pderiv\ p = 0 \Longrightarrow \exists h.\ p = [:h:]$
for *p* :: *'a*::{*semidom*,*semiring-char-0*} *poly*
by (*cases p*) (*auto simp*: *pderiv-eq-0-iff split*: *if-splits*)

lemma *rsquarefree-roots*: *rsquarefree p* $\longleftrightarrow$ ($\forall$ *a*. $\neg$ (*poly p a* = *0* $\wedge$ *poly* (*pderiv p*) *a* = *0*))
for *p* :: *'a*::*field-char-0 poly*
proof (*cases p* = *0*)
case *False*
show *?thesis*
proof (*cases pderiv p* = *0*)
case *True*
with ‹*p* ≠ *0*› *pderiv-iszero* **show** *?thesis*
by (*force simp add*: *order-0I rsquarefree-def*)
next
case *False*
with ‹*p* ≠ *0*› *order-pderiv2* **show** *?thesis*
by (*force simp add*: *rsquarefree-def order-root*)
qed
qed (*simp add*: *rsquarefree-def*)

lemma *rsquarefree-root-order*:
assumes *rsquarefree p poly p z* = *0*
shows *order z p* = *1*
by (*metis assms order-root rsquarefree-def*)

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* $\wedge$ ($\forall$ *a*. *poly q a* = *0* $\longleftrightarrow$ *poly p a* = *0*)
proof −
from ‹*pderiv p* ≠ *0*› **have** *p* ≠ *0* **by** *auto*
with ‹*p* = *q* ∗ *d*› **have** *q* ≠ *0* **by** *simp*
from *assms* **have** $\forall$ *a*. *order a q* = (*if order a p* = *0 then 0 else 1*)
by (*rule poly-squarefree-decomp-order2*)
with ‹*p* ≠ *0*› ‹*q* ≠ *0*› **show** *?thesis*
by (*simp add*: *rsquarefree-def order-root*)
qed

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*)
using *DERIV-chain*[*OF poly-DERIV assms*, *of p*] **by** (*simp add*: *o-def mult-ac*)

lemma *rsquarefree-single-root*[*simp*]: *rsquarefree* [:−*x*,*1*:]
proof −
have [:−*x*,*1*:] ≠ *0*
by *simp*
then show *?thesis*

unfolding *rsquarefree-def* **by** *auto*
qed

lemma *rsquarefree-mul*:
assumes *rsquarefree p rsquarefree q*
∀ *x*. *poly p x* ≠ *0* ∨ *poly q x* ≠ *0*
shows *rsquarefree*(*p* ∗ *q*)
proof −
have *11*: *p* ≠ *0 q* ≠ *0*
using *assms rsquarefree-def* **by** *auto*
then have *1*: *p* ∗ *q* ≠ *0*
by *simp*

have (∀ *x*. *order x p* = *0* ∨ *order x p* = *1*)
(∀ *x*. *order x q* = *0* ∨ *order x q* = *1*)
using *assms rsquarefree-def* **by** *auto*
then have *2*: (∀ *x*. *order x* (*p* ∗ *q*) = *0* ∨ *order x* (*p* ∗ *q*) = *1*)
using *11 1 order-mult assms*(*3*)
by (*metis comm-monoid-add-class.add-0 less-one order-gt-0-iff alethe-sum-simplify*)

show *?thesis* **unfolding** *rsquarefree-def*
using *1 2* **by** *auto*
qed

4.29 Algebraic numbers

lemma *intpolyE*:
assumes ⋀*i*. *poly.coeff p i* ∈ ℤ
obtains *q* **where** *p* = *map-poly of-int q*
proof −
have ∀ *i*∈{..*Polynomial.degree p*}. ∃ *x*. *poly.coeff p i* = *of-int x*
using *assms* **by** (*auto simp*: *Ints-def*)
from *bchoice*[*OF this*] **obtain** *f*
where *f*: ⋀*i*. *i* ≤ *Polynomial.degree p* ⟹ *poly.coeff p i* = *of-int* (*f i*) **by** *blast*
define *q* **where** *q* = *Poly* (*map f* [*0*..<*Suc* (*Polynomial.degree p*)])
have *p* = *map-poly of-int q*
by (*intro poly-eqI*)
(*auto simp*: *coeff-map-poly q-def nth-default-def f coeff-eq-0 simp del*: *upt-Suc*)
with *that* **show** *?thesis* **by** *blast*
qed

lemma *ratpolyE*:
assumes ⋀*i*. *poly.coeff p i* ∈ ℚ
obtains *q* **where** *p* = *map-poly of-rat q*
proof −
have ∀ *i*∈{..*Polynomial.degree p*}. ∃ *x*. *poly.coeff p i* = *of-rat x*
using *assms* **by** (*auto simp*: *Rats-def*)
from *bchoice*[*OF this*] **obtain** *f*
where *f*: ⋀*i*. *i* ≤ *Polynomial.degree p* ⟹ *poly.coeff p i* = *of-rat* (*f i*) **by** *blast*

define q **where** $q = Poly\ (map\ f\ [0..<Suc\ (Polynomial.degree\ p)])$
have $p = map\text{-}poly\ of\text{-}rat\ q$
by $(intro\ poly\text{-}eqI)$
$(auto\ simp: coeff\text{-}map\text{-}poly\ q\text{-}def\ nth\text{-}default\text{-}def\ f\ coeff\text{-}eq\text{-}0\ simp\ del: upt\text{-}Suc)$
with *that* **show** *?thesis* **by** *blast*
qed

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 $algebraicI: (\bigwedge i.\ coeff\ p\ i \in \mathbb{Z}) \Longrightarrow p \neq 0 \Longrightarrow poly\ p\ x = 0 \Longrightarrow algebraic\ x$
unfolding *algebraic-def* **by** *blast*

lemma *algebraicE*:
assumes *algebraic x*
obtains *p* **where** $\bigwedge i.\ coeff\ p\ i \in \mathbb{Z}\ p \neq 0\ poly\ p\ x = 0$
using *assms* **unfolding** *algebraic-def* **by** *blast*

lemma $algebraic\text{-}altdef: algebraic\ x \longleftrightarrow (\exists p.\ (\forall i.\ coeff\ p\ i \in \mathbb{Q}) \wedge p \neq 0 \wedge poly\ p\ x = 0)$
for $p :: 'a::field\text{-}char\text{-}0\ poly$
proof *safe*
fix *p*
assume $rat: \forall i.\ coeff\ p\ i \in \mathbb{Q}$ **and** $root: poly\ p\ x = 0$ **and** $nz: p \neq 0$
define *cs* **where** $cs = coeffs\ p$
from *rat* **have** $\forall c \in range\ (coeff\ p).\ \exists c'.\ c = of\text{-}rat\ c'$
unfolding *Rats-def* **by** *blast*
then obtain *f* **where** $f: coeff\ p\ i = of\text{-}rat\ (f\ (coeff\ p\ i))$ **for** *i*
by $(subst\ (asm)\ bchoice\text{-}iff)\ blast$
define cs' **where** $cs' = map\ (quotient\text{-}of \circ f)\ (coeffs\ p)$
define *d* **where** $d = Lcm\ (set\ (map\ snd\ cs'))$
define p' **where** $p' = smult\ (of\text{-}int\ d)\ p$

have $coeff\ p'\ n \in \mathbb{Z}$ **for** *n*
proof $(cases\ n \leq degree\ p)$
case *True*
define *c* **where** $c = coeff\ p\ n$
define *a* **where** $a = fst\ (quotient\text{-}of\ (f\ (coeff\ p\ n)))$
define *b* **where** $b = snd\ (quotient\text{-}of\ (f\ (coeff\ p\ n)))$
have $b\text{-}pos: b > 0$

```
    unfolding b-def using quotient-of-denom-pos' by simp
  have coeff p' n = of-int d * coeff p n
    by (simp add: p'-def)
  also have coeff p n = of-rat (of-int a / of-int b)
    unfolding a-def b-def
      by (subst quotient-of-div [of f (coeff p n), symmetric]) (simp-all add: f
[symmetric])
  also have of-int d * ... = of-rat (of-int (a*d) / of-int b)
    by (simp add: of-rat-mult of-rat-divide)
  also from nz True have b ∈ snd ` set cs'
    by (force simp: cs'-def o-def b-def coeffs-def simp del: upt-Suc)
  then have b dvd (a * d)
    by (simp add: d-def)
  then have of-int (a * d) / of-int b ∈ (ℤ :: rat set)
    by (rule of-int-divide-in-Ints)
  then have of-rat (of-int (a * d) / of-int b) ∈ ℤ by (elim Ints-cases) auto
  finally show ?thesis .
 next
  case False
  then show ?thesis
    by (auto simp: p'-def not-le coeff-eq-0)
 qed
 moreover have set (map snd cs') ⊆ {0<..}
  unfolding cs'-def using quotient-of-denom-pos' by (auto simp: coeffs-def simp
del: upt-Suc)
 then have d ≠ 0
  unfolding d-def by (induct cs') simp-all
 with nz have p' ≠ 0 by (simp add: p'-def)
 moreover from root have poly p' x = 0
  by (simp add: p'-def)
 ultimately show algebraic x
  unfolding algebraic-def by blast
next
 assume algebraic x
 then obtain p where p: coeff p i ∈ ℤ poly p x = 0 p ≠ 0 for i
  by (force simp: algebraic-def)
 moreover have coeff p i ∈ ℤ ⟹ coeff p i ∈ ℚ for i
  by (elim Ints-cases) simp
 ultimately show ∃ p. (∀ i. coeff p i ∈ ℚ) ∧ p ≠ 0 ∧ poly p x = 0 by auto
qed

lemma algebraicI': (⋀i. coeff p i ∈ ℚ) ⟹ p ≠ 0 ⟹ poly p x = 0 ⟹ algebraic
x
 unfolding algebraic-altdef by blast

lemma algebraicE':
 assumes algebraic (x :: 'a :: field-char-0)
 obtains p where p ≠ 0 poly (map-poly of-int p) x = 0
proof −
```

from *assms* **obtain** *q* **where** *q*: $\bigwedge i$. *coeff q i* $\in \mathbb{Z}$ *q* $\neq$ *0 poly q x* = *0*
by (*erule algebraicE*)
moreover from *this*(*1*) **obtain** *q'* **where** *q'*: *q* = *map-poly of-int q'* **by** (*erule intpolyE*)
moreover have *q'* $\neq$ *0*
using *q' q* **by** *auto*
ultimately show *?thesis* **by** (*intro that*[*of q'*]) *simp-all*
qed

lemma *algebraicE'-nonzero*:
assumes *algebraic* (*x* :: *'a* :: *field-char-0*) *x* $\neq$ *0*
obtains *p* **where** *p* $\neq$ *0 coeff p 0* $\neq$ *0 poly* (*map-poly of-int p*) *x* = *0*
proof −
from *assms*(*1*) **obtain** *p* **where** *p*: *p* $\neq$ *0 poly* (*map-poly of-int p*) *x* = *0*
by (*erule algebraicE'*)
define *n* :: *nat* **where** *n* = *order 0 p*
have *monom 1 n dvd p* **by** (*simp add*: *monom-1-dvd-iff p n-def*)
then obtain *q* **where** *q*: *p* = *monom 1 n* ∗ *q* **by** (*erule dvdE*)
have [*simp*]: *map-poly of-int* (*monom 1 n* ∗ *q*) = *monom* (*1* :: *'a*) *n* ∗ *map-poly of-int q*
by (*induction n*) (*auto simp*: *monom-0 monom-Suc map-poly-pCons*)
from *p* **have** *q* $\neq$ *0 poly* (*map-poly of-int q*) *x* = *0* **by** (*auto simp*: *q poly-monom assms*(*2*))
moreover from *this* **have** *order 0 p* = *n* + *order 0 q* **by** (*simp add*: *q order-mult*)
hence *order 0 q* = *0* **by** (*simp add*: *n-def*)
with ‹*q* $\neq$ *0*› **have** *poly q 0* $\neq$ *0* **by** (*simp add*: *order-root*)
ultimately show *?thesis* **using** *that*[*of q*] **by** (*auto simp*: *poly-0-coeff-0*)
qed

lemma *rat-imp-algebraic*: *x* $\in \mathbb{Q} \Longrightarrow$ *algebraic x*
proof (*rule algebraicI'*)
show *poly* [:−*x*, *1*:] *x* = *0*
by *simp*
qed (*auto simp*: *coeff-pCons split*: *nat.splits*)

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*)
by (*simp-all add*: *rat-imp-algebraic*)

lemma *algebraic-ii* [*simp*, *intro*]: *algebraic* i
proof (*rule algebraicI*)
show *poly* [:*1*, *0*, *1*:] i = *0*
by *simp*
qed (*auto simp*: *coeff-pCons split*: *nat.splits*)

lemma *algebraic-minus* [*intro*]:

assumes *algebraic x*
shows *algebraic* $(-x)$
proof −
from *assms* **obtain** *p* **where** *p*: $\forall\, i.$ *coeff p i* $\in \mathbb{Z}$ *poly p x* = *0 p* $\neq$ *0*
by (*elim algebraicE*) *auto*
define *s* **where** *s* = (*if even* (*degree p*) *then 1 else* -1 :: $'a$)

define *q* **where** *q* = *Polynomial.smult s* (*pcompose p* [:*0*, -1:])
have *poly q* $(-x)$ = *0*
using *p* **by** (*auto simp*: *q-def poly-pcompose s-def*)
moreover have *q* $\neq$ *0*
using *p* **by** (*auto simp*: *q-def s-def pcompose-eq-0-iff*)
find-theorems *pcompose* - - = *0*
moreover have *coeff q i* $\in \mathbb{Z}$ **for** *i*
proof −
have *coeff* (*pcompose p* [:*0*, -1:]) *i* $\in \mathbb{Z}$
using *p* **by** (*intro coeff-pcompose-semiring-closed*) (*auto simp*: *coeff-pCons split*: *nat.splits*)
thus *?thesis* **by** (*simp add*: *q-def s-def*)
qed
ultimately show *?thesis*
by (*auto simp*: *algebraic-def*)
qed

lemma *algebraic-minus-iff* [*simp*]:
algebraic $(-x) \longleftrightarrow$ *algebraic* (*x* :: $'a$ :: *field-char-0*)
using *algebraic-minus*[*of x*] *algebraic-minus*[*of* $-x$] **by** *auto*

lemma *algebraic-inverse* [*intro*]:
assumes *algebraic x*
shows *algebraic* (*inverse x*)
proof (*cases x* = *0*)
case [*simp*]: *False*
from *assms* **obtain** *p* **where** *p*: $\forall\, i.$ *coeff p i* $\in \mathbb{Z}$ *poly p x* = *0 p* $\neq$ *0*
by (*elim algebraicE*) *auto*
show *?thesis*
proof (*rule algebraicI*)
show *poly* (*reflect-poly p*) (*inverse x*) = *0*
using *assms p* **by** (*simp add*: *poly-reflect-poly-nz*)
qed (*use assms p* **in** ‹*auto simp*: *coeff-reflect-poly*›)
qed *auto*

lemma *algebraic-root*:
assumes *algebraic y*
and *poly p x* = *y* **and** $\forall\, i.$ *coeff p i* $\in \mathbb{Z}$ **and** *lead-coeff p* = *1* **and** *degree p* > *0*
shows *algebraic x*
proof −
from *assms* **obtain** *q* **where** *q*: *poly q y* = *0* $\forall\, i.$ *coeff q i* $\in \mathbb{Z}$ *q* $\neq$ *0*

```
    by (elim algebraicE) auto
  show ?thesis
  proof (rule algebraicI)
    from assms q show pcompose q p ≠ 0
      by (auto simp: pcompose-eq-0-iff)
    from assms q show coeff (pcompose q p) i ∈ ℤ for i
      by (intro allI coeff-pcompose-semiring-closed) auto
    show poly (pcompose q p) x = 0
      using assms q by (simp add: poly-pcompose)
  qed
qed

lemma algebraic-abs-real [simp]:
  algebraic |x :: real| ⟷ algebraic x
  by (auto simp: abs-if)

lemma algebraic-nth-root-real [intro]:
  assumes algebraic x
  shows   algebraic (root n x)
proof (cases n = 0)
  case False
  show ?thesis
  proof (rule algebraic-root)
    show poly (monom 1 n) (root n x) = (if even n then |x| else x)
      using sgn-power-root[of n x] False
      by (auto simp add: poly-monom sgn-if split: if-splits)
  qed (use False assms in ‹auto simp: degree-monom-eq›)
qed auto

lemma algebraic-sqrt [intro]: algebraic x ⟹ algebraic (sqrt x)
  by (auto simp: sqrt-def)

lemma algebraic-csqrt [intro]: algebraic x ⟹ algebraic (csqrt x)
  by (rule algebraic-root[where p = monom 1 2])
     (auto simp: poly-monom degree-monom-eq)

lemma algebraic-cnj [intro]:
  assumes algebraic x
  shows   algebraic (cnj x)
proof -
  from assms obtain p where p: poly p x = 0 ∀i. coeff p i ∈ ℤ p ≠ 0
    by (elim algebraicE) auto
  show ?thesis
  proof (rule algebraicI)
    show poly (map-poly cnj p) (cnj x) = 0
      using p by simp
    show map-poly cnj p ≠ 0
      using p by (auto simp: map-poly-eq-0-iff)
    show coeff (map-poly cnj p) i ∈ ℤ for i
```

```
    using p by (auto simp: coeff-map-poly)
  qed
qed

lemma algebraic-cnj-iff [simp]: algebraic (cnj x) ⟷ algebraic x
  using algebraic-cnj[of x] algebraic-cnj[of cnj x] by auto

lemma algebraic-of-real [intro]:
  assumes algebraic x
  shows   algebraic (of-real x)
proof −
  from assms obtain p where p: p ≠ 0 poly (map-poly of-int p) x = 0 by (erule
algebraicE')
  have 1: map-poly of-int p ≠ (0 :: 'a poly)
    using p by (metis coeff-0 coeff-map-poly leading-coeff-0-iff of-int-eq-0-iff)

  have poly (map-poly of-int p) (of-real x :: 'a) = of-real (poly (map-poly of-int p)
x)
    by (simp add: poly-altdef degree-map-poly coeff-map-poly)
  also note p(2)
  finally have 2: poly (map-poly of-int p) (of-real x :: 'a) = 0
    by simp

  from 1 2 show algebraic (of-real x :: 'a)
    by (intro algebraicI[of map-poly of-int p]) (auto simp: coeff-map-poly)
qed

lemma algebraic-of-real-iff [simp]:
   algebraic (of-real x :: 'a :: {real-algebra-1,field-char-0}) ⟷ algebraic x
proof
  assume algebraic (of-real x :: 'a)
  then obtain p where p: p ≠ 0 poly (map-poly of-int p) (of-real x :: 'a) = 0
    by (erule algebraicE')
  have 1: (map-poly of-int p :: real poly) ≠ 0
   using p by (metis coeff-0 coeff-map-poly leading-coeff-0-iff of-int-0 of-int-eq-iff)

  note p(2)
 also have poly (map-poly of-int p) (of-real x :: 'a) = of-real (poly (map-poly of-int
p) x)
    by (simp add: poly-altdef degree-map-poly coeff-map-poly)
  also have ... = 0 ⟷ poly (map-poly of-int p) x = 0
    using of-real-eq-0-iff by blast
  finally have 2: poly (map-poly real-of-int p) x = 0 .

  from 1 and 2 show algebraic x
    by (intro algebraicI[of map-poly of-int p]) (auto simp: coeff-map-poly)
qed auto
```

4.30 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
assume *algebraic-int x*
then obtain *p* **where** *p*: *lead-coeff p = 1* ∀ *i*. *coeff p i* ∈ ℤ *poly p x = 0*
by (*auto elim*: *algebraic-int.cases*)
define *the-int* **where** *the-int* = (λ*x*::*'a*. *THE r*. *x = of-int r*)
define *p'* **where** *p' = map-poly the-int p*
have *of-int-the-int*: *of-int* (*the-int x*) = *x* **if** *x* ∈ ℤ **for** *x*
unfolding *the-int-def* **by** (*rule sym*, *rule theI'*) (*insert that*, *auto simp*: *Ints-def*)
have *the-int-0-iff*: *the-int x = 0* ⟷ *x = 0* **if** *x* ∈ ℤ **for** *x*
using *of-int-the-int*[*OF that*] **by** *auto*
have [*simp*]: *the-int 0 = 0*
by (*subst the-int-0-iff*) *auto*
have *map-poly of-int p' = map-poly* (*of-int* ∘ *the-int*) *p*
by (*simp add*: *p'-def map-poly-map-poly*)
also from *p of-int-the-int* **have** ... = *p*
by (*subst poly-eq-iff*) (*auto simp*: *coeff-map-poly*)
finally have *p-p'*: *map-poly of-int p' = p* **.**

show (∃ *p*. *poly* (*map-poly of-int p*) *x = 0* ∧ *lead-coeff p = 1*)
proof (*intro exI conjI notI*)
from *p* **show** *poly* (*map-poly of-int p'*) *x = 0* **by** (*simp add*: *p-p'*)
next
show *lead-coeff p' = 1*
using *p* **by** (*simp flip*: *p-p' add*: *degree-map-poly coeff-map-poly*)
qed
next
assume ∃ *p*. *poly* (*map-poly of-int p*) *x = 0* ∧ *lead-coeff p = 1*
then obtain *p* **where** *p*: *poly* (*map-poly of-int p*) *x = 0 lead-coeff p = 1*
by *auto*
define *p'* **where** *p' =* (*map-poly of-int p* :: *'a poly*)
from *p* **have** *lead-coeff p' = 1 poly p' x = 0* ∀ *i*. *coeff p' i* ∈ ℤ
by (*auto simp*: *p'-def coeff-map-poly degree-map-poly*)
thus *algebraic-int x*
by (*intro algebraic-int.intros*)
qed

theorem *rational-algebraic-int-is-int*:
assumes *algebraic-int x* **and** *x* ∈ ℚ
shows *x* ∈ ℤ
proof −
from *assms*(*2*) **obtain** *a b* **where** *ab*: *b > 0 Rings.coprime a b* **and** *x-eq*: *x =*

```
of-int a / of-int b
    by (auto elim: Rats-cases')
  from ‹b > 0› have [simp]: b ≠ 0
    by auto
  from assms(1) obtain p
    where p: lead-coeff p = 1 ∀ i. coeff p i ∈ ℤ poly p x = 0
    by (auto simp: algebraic-int.simps)

  define q :: 'a poly where q = [:−of-int a, of-int b:]
  have poly q x = 0 q ≠ 0 ∀ i. coeff q i ∈ ℤ
    by (auto simp: x-eq q-def coeff-pCons split: nat.splits)
  define n where n = degree p
  have n > 0
    using p by (intro Nat.gr0I) (auto simp: n-def elim!: degree-eq-zeroE)
  have (∑ i<n. coeff p i * of-int (a ^ i * b ^ (n − i − 1))) ∈ ℤ
    using p by auto
  then obtain R where R: of-int R = (∑ i<n. coeff p i * of-int (a ^ i * b ^ (n
− i − 1)))
    by (auto simp: Ints-def)
  have [simp]: coeff p n = 1
    using p by (auto simp: n-def)

  have 0 = poly p x * of-int b ^ n
    using p by simp
  also have ... = (∑ i≤n. coeff p i * x ^ i * of-int b ^ n)
    by (simp add: poly-altdef n-def sum-distrib-right)
  also have ... = (∑ i≤n. coeff p i * of-int (a ^ i * b ^ (n − i)))
    by (intro sum.cong) (auto simp: x-eq field-simps simp flip: power-add)
  also have {..n} = insert n {..<n}
    using ‹n > 0› by auto
  also have (∑ i∈.... coeff p i * of-int (a ^ i * b ^ (n − i))) =
              coeff p n * of-int (a ^ n) + (∑ i<n. coeff p i * of-int (a ^ i * b ^ (n
− i)))
    by (subst sum.insert) auto
  also have (∑ i<n. coeff p i * of-int (a ^ i * b ^ (n − i))) =
            (∑ i<n. coeff p i * of-int (a ^ i * b * b ^ (n − i − 1)))
    by (intro sum.cong) (auto simp flip: power-add power-Suc simp: Suc-diff-Suc)
  also have ... = of-int (b * R)
    by (simp add: R sum-distrib-left sum-distrib-right mult-ac)
  finally have of-int (a ^ n) = (−of-int (b * R) :: 'a)
    by (auto simp: add-eq-0-iff)
  hence a ^ n = −b * R
    by (simp flip: of-int-mult of-int-power of-int-minus)
  hence b dvd a ^ n
    by simp
  with ‹Rings.coprime a b› have b dvd 1
    by (meson coprime-power-left-iff dvd-refl not-coprimeI)
  with x-eq and ‹b > 0› show ?thesis
    by auto
```

qed

lemma *algebraic-int-imp-algebraic* [*dest*]: *algebraic-int x* $\Longrightarrow$ *algebraic x*
by (*auto simp*: *algebraic-int.simps algebraic-def*)

lemma *int-imp-algebraic-int*:
assumes $x \in \mathbb{Z}$
shows *algebraic-int x*
proof
show $\forall$ *i*. *coeff* [:$-x$, *1*:] *i* $\in \mathbb{Z}$
using *assms* **by** (*auto simp*: *coeff-pCons split*: *nat.splits*)
qed *auto*

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*)
by (*simp-all add*: *int-imp-algebraic-int*)

lemma *algebraic-int-ii* [*simp*, *intro*]: *algebraic-int* i
proof
show *poly* [:*1*, *0*, *1*:] i = *0*
by *simp*
qed (*auto simp*: *coeff-pCons split*: *nat.splits*)

lemma *algebraic-int-minus* [*intro*]:
assumes *algebraic-int x*
shows *algebraic-int* ($-x$)
proof −
from *assms* **obtain** *p* **where** *p*: *lead-coeff p* = *1* $\forall$ *i*. *coeff p i* $\in \mathbb{Z}$ *poly p x* = *0*
by (*auto simp*: *algebraic-int.simps*)
define *s* **where** *s* = (*if even* (*degree p*) *then 1 else* -1 :: *'a*)

define *q* **where** *q* = *Polynomial.smult s* (*pcompose p* [:*0*, $-$*1*:])
have *lead-coeff q* = *s* $*$ *lead-coeff* (*pcompose p* [:*0*, $-$*1*:])
by (*simp add*: *q-def*)
also have *lead-coeff* (*pcompose p* [:*0*, $-$*1*:]) = *lead-coeff p* $*$ ($-$ *1*) $\hat{}$ *degree p*
by (*subst lead-coeff-comp*) *auto*
finally have *poly q* ($-x$) = *0* **and** *lead-coeff q* = *1*
using *p* **by** (*auto simp*: *q-def poly-pcompose s-def*)
moreover have *coeff q i* $\in \mathbb{Z}$ **for** *i*
proof −
have *coeff* (*pcompose p* [:*0*, $-$*1*:]) *i* $\in \mathbb{Z}$
using *p* **by** (*intro coeff-pcompose-semiring-closed*) (*auto simp*: *coeff-pCons split*: *nat.splits*)
thus *?thesis* **by** (*simp add*: *q-def s-def*)
qed
ultimately show *?thesis*

```
    by (auto simp: algebraic-int.simps)
qed

lemma algebraic-int-minus-iff [simp]:
  algebraic-int (−x) ⟷ algebraic-int (x :: 'a :: field-char-0)
  using algebraic-int-minus[of x] algebraic-int-minus[of −x] by auto

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
  from assms have [simp]: x ≠ 0
    by (auto simp: poly-0-coeff-0)
  show poly (reflect-poly p) (inverse x) = 0
    using assms by (simp add: poly-reflect-poly-nz)
qed (use assms in ‹auto simp: coeff-reflect-poly›)

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 −
  from assms obtain q where q: poly q y = 0 ∀ i. coeff q i ∈ ℤ lead-coeff q = 1
    by (auto simp: algebraic-int.simps)
  show ?thesis
  proof
    from assms q show lead-coeff (pcompose q p) = 1
      by (subst lead-coeff-comp) auto
    from assms q show ∀ i. coeff (pcompose q p) i ∈ ℤ
      by (intro allI coeff-pcompose-semiring-closed) auto
    show poly (pcompose q p) x = 0
      using assms q by (simp add: poly-pcompose)
  qed
qed

lemma algebraic-int-abs-real [simp]:
  algebraic-int |x :: real| ⟷ algebraic-int x
  by (auto simp: abs-if)

lemma algebraic-int-nth-root-real [intro]:
  assumes algebraic-int x
  shows   algebraic-int (root n x)
proof (cases n = 0)
  case False
  show ?thesis
  proof (rule algebraic-int-root)
    show poly (monom 1 n) (root n x) = (if even n then |x| else x)
      using sgn-power-root[of n x] False
```

```
    by (auto simp add: poly-monom sgn-if split: if-splits)
  qed (use False assms in ‹auto simp: degree-monom-eq›)
qed auto

lemma algebraic-int-sqrt [intro]: algebraic-int x ⟹ algebraic-int (sqrt x)
  by (auto simp: sqrt-def)

lemma algebraic-int-csqrt [intro]: algebraic-int x ⟹ algebraic-int (csqrt x)
  by (rule algebraic-int-root[where p = monom 1 2])
     (auto simp: poly-monom degree-monom-eq)

lemma algebraic-int-cnj [intro]:
  assumes algebraic-int x
  shows   algebraic-int (cnj x)
proof −
  from assms obtain p where p: lead-coeff p = 1 ∀ i. coeff p i ∈ ℤ poly p x = 0
    by (auto simp: algebraic-int.simps)
  show ?thesis
  proof
    show poly (map-poly cnj p) (cnj x) = 0
      using p by simp
    show lead-coeff (map-poly cnj p) = 1
      using p by (simp add: coeff-map-poly degree-map-poly)
    show ∀ i. coeff (map-poly cnj p) i ∈ ℤ
      using p by (auto simp: coeff-map-poly)
  qed
qed

lemma algebraic-int-cnj-iff [simp]: algebraic-int (cnj x) ⟷ algebraic-int x
  using algebraic-int-cnj[of x] algebraic-int-cnj[of cnj x] by auto

lemma algebraic-int-of-real [intro]:
  assumes algebraic-int x
  shows   algebraic-int (of-real x)
proof −
  from assms obtain p where p: poly p x = 0 ∀ i. coeff p i ∈ ℤ lead-coeff p = 1
    by (auto simp: algebraic-int.simps)
  show algebraic-int (of-real x :: 'a)
  proof
    have poly (map-poly of-real p) (of-real x) = (of-real (poly p x) :: 'a)
      by (induction p) (auto simp: map-poly-pCons)
    thus poly (map-poly of-real p) (of-real x) = (0 :: 'a)
      using p by simp
  qed (use p in ‹auto simp: coeff-map-poly degree-map-poly›)
qed

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

assume *algebraic-int* (*of-real x* :: *'a*)
then obtain *p*
where *p*: *poly* (*map-poly of-int p*) (*of-real x* :: *'a*) = *0 lead-coeff p* = *1*
by (*auto simp*: *algebraic-int-altdef-ipoly*)
show *algebraic-int x*
unfolding *algebraic-int-altdef-ipoly*
proof (*intro exI*[*of - p*] *conjI*)
have *of-real* (*poly* (*map-poly real-of-int p*) *x*) = *poly* (*map-poly of-int p*) (*of-real x* :: *'a*)
by (*induction p*) (*auto simp*: *map-poly-pCons*)
also note *p*(*1*)
finally show *poly* (*map-poly real-of-int p*) *x* = *0* **by** *simp*
qed (*use p* **in** *auto*)
qed *auto*

4.31 Division of polynomials

4.31.1 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*
using *assms*(*3*−)
proof (*induct n arbitrary*: *q r dr*)
case (*Suc n*)

let *?rr* = *d* ∗ *r*
let *?a* = *coeff ?rr dr*
let *?qq* = *?a div lc*
define *b* **where** [*simp*]: *b* = *monom ?qq n*
let *?rrr* = *d* ∗ (*r* − *b*)
let *?qqq* = *q* + *b*
note *res* = *Suc*(*3*)
from *Suc*(*4*) **have** *dr*: *dr* = *n* + *degree d* **by** *auto*
from *d* **have** *lc*: *lc* ≠ *0* **by** *auto*
have *coeff* (*b* ∗ *d*) *dr* = *coeff b n* ∗ *coeff d* (*degree d*)
proof (*cases ?qq* = *0*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
then have *n*: *n* = *degree b*
by (*simp add*: *degree-monom-eq*)
show *?thesis*
unfolding *n dr* **by** (*simp add*: *coeff-mult-degree-sum*)
qed
also have . . . = *lc* ∗ *coeff b n*
by (*simp add*: *d*)
finally have *c2*: *coeff* (*b* ∗ *d*) *dr* = *lc* ∗ *coeff b n* **.**
have *rrr*: *?rrr* = *?rr* − *b* ∗ *d*
by (*simp add*: *field-simps*)
have *c1*: *coeff* (*d* ∗ *r*) *dr* = *lc* ∗ *coeff r n*
proof (*cases degree r* = *n*)
case *True*
with *Suc*(*2*) **show** *?thesis*
unfolding *dr* **using** *coeff-mult-degree-sum*[*of d r*] *d* **by** (*auto simp*: *ac-simps*)
next
case *False*
from *dr Suc*(*2*) **have** *degree r* ≤ *n*
by *auto*
(*metis add.commute add-le-cancel-left d*(*1*) *degree-0 degree-mult-eq*
diff-is-0-eq diff-zero le-cases)
with *False* **have** *r-n*: *degree r* < *n*
by *auto*
then have *right*: *lc* ∗ *coeff r n* = *0*
by (*simp add*: *coeff-eq-0*)
have *coeff* (*d* ∗ *r*) *dr* = *coeff* (*d* ∗ *r*) (*degree d* + *n*)
by (*simp add*: *dr ac-simps*)
also from *r-n* **have** . . . = *0*
by (*metis False Suc.prems*(*1*) *add.commute add-left-imp-eq coeff-degree-mult*
coeff-eq-0
coeff-mult-degree-sum degree-mult-le dr le-eq-less-or-eq)
finally show *?thesis*
by (*simp only*: *right*)
qed

```
  have c0: coeff ?rrr dr = 0
    and id: lc * (coeff (d * r) dr div lc) = coeff (d * r) dr
    unfolding rrr coeff-diff c2
    unfolding b-def coeff-monom coeff-smult c1 using lc by auto
  from res[unfolded divide-poly-main.simps[of lc q] Let-def] id
  have res: divide-poly-main lc ?qqq ?rrr d (dr - 1) n = q'
    by (simp del: divide-poly-main.simps add: field-simps)
  note IH = Suc(1)[OF - res]
  from Suc(4) have dr: dr = n + degree d by auto
  from Suc(2) have deg-rr: degree ?rr ≤ dr by auto
  have deg-bd: degree (b * d) ≤ dr
    unfolding dr b-def by (rule order.trans[OF degree-mult-le]) (auto simp: de-
gree-monom-le)
  have degree ?rrr ≤ dr
    unfolding rrr by (rule degree-diff-le[OF deg-rr deg-bd])
  with c0 have deg-rrr: degree ?rrr ≤ (dr - 1)
    by (rule coeff-0-degree-minus-1)
  have n = 1 + (dr - 1) - degree d ∨ dr - 1 = 0 ∧ n = 0 ∧ ?rrr = 0
  proof (cases dr)
    case 0
    with Suc(4) have 0: dr = 0 n = 0 degree d = 0
      by auto
    with deg-rrr have degree ?rrr = 0
      by simp
    from degree-eq-zeroE[OF this] obtain a where rrr: ?rrr = [:a:]
      by metis
    show ?thesis
      unfolding 0 using c0 unfolding rrr 0 by simp
  next
    case -: Suc
    with Suc(4) show ?thesis by auto
  qed
  from IH[OF deg-rrr this] show ?case
    by simp
next
  case 0
  show ?case
  proof (cases r = 0)
    case True
    with 0 show ?thesis by auto
  next
    case False
    from d False have degree (d * r) = degree d + degree r
      by (subst degree-mult-eq) auto
    with 0 d show ?thesis by auto
  qed
qed

lemma divide-poly-main-0: divide-poly-main 0 0 r d dr n = 0
```

```
proof (induct n arbitrary: r d dr)
  case 0
  then show ?case by simp
next
  case Suc
  show ?case
    unfolding divide-poly-main.simps[of - - r] Let-def
    by (simp add: Suc del: divide-poly-main.simps)
qed

lemma divide-poly:
  assumes g: g ≠ 0
  shows (f * g) div g = (f :: 'a poly)
proof -
  have len: length (coeffs f) = Suc (degree f) if f ≠ 0 for f :: 'a poly
    using that unfolding degree-eq-length-coeffs by auto
  have divide-poly-main (coeff g (degree g)) 0 (g * f) g (degree (g * f))
    (1 + length (coeffs (g * f)) - length (coeffs  g)) = (f * g) div g
    by (simp add: divide-poly-def Let-def ac-simps)
  note main = divide-poly-main[OF g refl le-refl this]
  have (f * g) div g = 0 + f
  proof (rule main, goal-cases)
    case 1
    show ?case
    proof (cases f = 0)
      case True
      with g show ?thesis
        by (auto simp: degree-eq-length-coeffs)
    next
      case False
      with g have fg: g * f ≠ 0 by auto
      show ?thesis
        unfolding len[OF fg] len[OF g] by auto
    qed
  qed
  then show ?thesis by simp
qed

lemma divide-poly-0: f div 0 = 0
  for f :: 'a poly
  by (simp add: divide-poly-def Let-def divide-poly-main-0)

instance
  by standard (auto simp: divide-poly divide-poly-0)

end

instance poly :: (idom-divide) algebraic-semidom ..
```

lemma *div-const-poly-conv-map-poly*:
assumes [:*c*:] *dvd p*
shows *p div* [:*c*:] = *map-poly* (λ*x*. *x div c*) *p*
proof (*cases c* = *0*)
case *True*
then show *?thesis*
by (*auto intro*!: *poly-eqI simp*: *coeff-map-poly*)
next
case *False*
from *assms* **obtain** *q* **where** *p*: *p* = [:*c*:] ∗ *q* **by** (*rule dvdE*)
moreover {
have *smult c q* = [:*c*:] ∗ *q*
by *simp*
also have . . . *div* [:*c*:] = *q*
by (*rule nonzero-mult-div-cancel-left*) (*use False* **in** *auto*)
finally have *smult c q div* [:*c*:] = *q* **.**
}
ultimately show *?thesis* **by** (*intro poly-eqI*) (*auto simp*: *coeff-map-poly False*)
qed

lemma *is-unit-monom-0*:
fixes *a* :: *'a*::*field*
assumes *a* ≠ *0*
shows *is-unit* (*monom a 0*)
proof
from *assms* **show** *1* = *monom a 0* ∗ *monom* (*inverse a*) *0*
by (*simp add*: *mult-monom*)
qed

lemma *is-unit-triv*: *a* ≠ *0* ⟹ *is-unit* [:*a*:]
for *a* :: *'a*::*field*
by (*simp add*: *is-unit-monom-0 monom-0* [*symmetric*])

lemma *is-unit-iff-degree*:
fixes *p* :: *'a*::*field poly*
assumes *p* ≠ *0*
shows *is-unit p* ⟷ *degree p* = *0*
(**is** *?lhs* ⟷ *?rhs*)
by (*metis assms degree0-coeffs is-unit-triv pCons-0-0 poly-dvd-1*)

lemma *is-unit-pCons-iff*: *is-unit* (*pCons a p*) ⟷ *p* = *0* ∧ *a* ≠ *0*
for *p* :: *'a*::*field poly*
by (*metis degree-0-id is-unit-iff-degree is-unit-triv not-is-unit-0 pCons-eq-0-iff pCons-eq-iff*)

lemma *is-unit-monom-trivial*: *is-unit p* ⟹ *monom* (*coeff p* (*degree p*)) *0* = *p*
for *p* :: *'a*::*field poly*
by (*cases p*) (*simp-all add*: *monom-0 is-unit-pCons-iff*)

lemma *is-unit-const-poly-iff*: [:*c*:] *dvd 1* ⟷ *c dvd 1*
for *c* :: *'a*::{*comm-semiring-1*,*semiring-no-zero-divisors*}
by (*auto simp*: *one-pCons*)

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*
by (*metis assms degree-0-id poly-dvd-1*)

lemma *is-unit-polyE'*:
fixes *p* :: *'a*::*field poly*
assumes *is-unit p*
obtains *a* **where** *p* = *monom a 0* **and** *a* ≠ *0*
by (*metis assms degree-eq-zeroE monom-0 not-is-unit-0 pCons-0-0 poly-dvd-1*)

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*
by (*auto elim*: *is-unit-polyE simp add*: *is-unit-const-poly-iff*)

lemma *coprime-poly-0*:
poly p x ≠ *0* ∨ *poly q x* ≠ *0* **if** *coprime p q*
for *x* :: *'a* :: *field*
proof (*rule ccontr*)
assume ¬ (*poly p x* ≠ *0* ∨ *poly q x* ≠ *0*)
then have [:−*x*, *1*:] *dvd p* [:−*x*, *1*:] *dvd q*
by (*simp-all add*: *poly-eq-0-iff-dvd*)
with *that* **have** *is-unit* [:−*x*, *1*:]
by (*rule coprime-common-divisor*)
then show *False*
by (*auto simp add*: *is-unit-pCons-iff*)
qed

lemma *root-imp-reducible-poly*:
fixes *x* :: *'a* :: *field*
assumes *poly p x* = *0* **and** *degree p* > *1*
shows ¬*irreducible p*
proof −
from *assms* **have** *p* ≠ *0*
by *auto*
define *q* **where** *q* = [:−*x*, *1*:]
have *q dvd p*
using *assms* **by** (*simp add*: *poly-eq-0-iff-dvd q-def*)
then obtain *r* **where** *p-eq*: *p* = *q* ∗ *r*
by (*elim dvdE*)
have [*simp*]: *q* ≠ *0 r* ≠ *0*
using ‹*p* ≠ *0*› **by** (*auto simp*: *p-eq*)
have *degree p* = *Suc* (*degree r*)
unfolding *p-eq* **by** (*subst degree-mult-eq*) (*auto simp*: *q-def*)

with *assms*(*2*) **have** *degree r* > *0*
by *auto*
hence ¬*r dvd 1*
by (*auto simp*: *is-unit-poly-iff*)
moreover have ¬*q dvd 1*
by (*auto simp*: *is-unit-poly-iff q-def*)
ultimately show *?thesis* **using** *p-eq*
by (*auto simp*: *irreducible-def*)
qed

lemma *reducible-polyI*:
fixes *p* :: *'a* :: *field poly*
assumes *p* = *q* ∗ *r degree q* > *0 degree r* > *0*
shows ¬*irreducible p*
using *assms* **unfolding** *irreducible-def*
by (*metis* (*no-types*, *opaque-lifting*) *is-unitE is-unit-iff-degree not-gr0*)

4.31.2 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'*
using *assms*(*3*−)
proof (*induct n arbitrary*: *q r dr*)
case *0*
then show *?case* **by** *auto*

next
case (*Suc n*)
let *?rr* = *smult lc r*
let *?qq* = *coeff r dr*
define *b* **where** [*simp*]: *b* = *monom ?qq n*
let *?rrr* = *?rr* − *b* ∗ *d*
let *?qqq* = *smult lc q* + *b*
note *res* = *Suc*(*3*)
from *res*[*unfolded pseudo-divmod-main.simps*[*of lc q*] *Let-def*]
have *res*: *pseudo-divmod-main lc ?qqq ?rrr d* (*dr* − *1*) *n* = (*q*′,*r*′)
by (*simp del*: *pseudo-divmod-main.simps*)
from *Suc*(*4*) **have** *dr*: *dr* = *n* + *degree d* **by** *auto*
have *coeff* (*b* ∗ *d*) *dr* = *coeff b n* ∗ *coeff d* (*degree d*)
proof (*cases ?qq* = *0*)
case *True*
then show *?thesis* **by** *auto*
next
case *False*
then have *n*: *n* = *degree b*
by (*simp add*: *degree-monom-eq*)
show *?thesis*
unfolding *n dr* **by** (*simp add*: *coeff-mult-degree-sum*)
qed
also have ... = *lc* ∗ *coeff b n* **by** (*simp add*: *d*)
finally have *coeff* (*b* ∗ *d*) *dr* = *lc* ∗ *coeff b n* **.**
moreover have *coeff ?rr dr* = *lc* ∗ *coeff r dr*
by *simp*
ultimately have *c0*: *coeff ?rrr dr* = *0*
by *auto*
from *Suc*(*4*) **have** *dr*: *dr* = *n* + *degree d* **by** *auto*
have *deg-rr*: *degree ?rr* ≤ *dr*
using *Suc*(*2*) *degree-smult-le dual-order.trans* **by** *blast*
have *deg-bd*: *degree* (*b* ∗ *d*) ≤ *dr*
unfolding *dr* **by** (*rule order.trans*[*OF degree-mult-le*]) (*auto simp*: *degree-monom-le*)
have *degree ?rrr* ≤ *dr*
using *degree-diff-le*[*OF deg-rr deg-bd*] **by** *auto*
with *c0* **have** *deg-rrr*: *degree ?rrr* ≤ (*dr* − *1*)
by (*rule coeff-0-degree-minus-1*)
have *n* = *1* + (*dr* − *1*) − *degree d* ∨ *dr* − *1* = *0* ∧ *n* = *0* ∧ *?rrr* = *0*
proof (*cases dr*)
case *0*
with *Suc*(*4*) **have** *0*: *dr* = *0 n* = *0 degree d* = *0* **by** *auto*
with *deg-rrr* **have** *degree ?rrr* = *0* **by** *simp*
then have ∃ *a*. *?rrr* = [:*a*:]
by (*metis degree-pCons-eq-if old.nat.distinct*(*2*) *pCons-cases*)
from *this* **obtain** *a* **where** *rrr*: *?rrr* = [:*a*:]
by *auto*
show *?thesis*
unfolding *0* **using** *c0* **unfolding** *rrr 0* **by** *simp*

```
  next
    case -: Suc
    with Suc(4) show ?thesis by auto
  qed
  note IH = Suc(1)[OF deg-rrr res this]
  show ?case
  proof (intro conjI)
    from IH show r' = 0 ∨ degree r' < degree d
      by blast
    show smult (lc ^ Suc n) (d * q + r) = d * q' + r'
      unfolding IH[THEN conjunct2,symmetric]
      by (simp add: field-simps smult-add-right)
  qed
qed

lemma pseudo-divmod:
  assumes g: g ≠ 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 −
  from *[unfolded pseudo-divmod-def Let-def]
  have pseudo-divmod-main (coeff g (degree g)) 0 f g (degree f)
      (1 + length (coeffs f) − length (coeffs g)) = (q, r)
    by (auto simp: g)
  note main = pseudo-divmod-main[OF - - - this, OF g refl le-refl]
  from g have 1 + length (coeffs f) − length (coeffs g) = 1 + degree f − degree
g ∨
    degree f = 0 ∧ 1 + length (coeffs f) − length (coeffs g) = 0 ∧ f = 0
    by (cases f = 0; cases coeffs g) (auto simp: degree-eq-length-coeffs)
  note main' = main[OF this]
  then show r = 0 ∨ degree r < degree g by auto
  show smult (coeff g (degree g) ^ (Suc (degree f) − degree g)) f = g * q + r
   by (subst main'[THEN conjunct2, symmetric], simp add: degree-eq-length-coeffs,
       cases f = 0; cases coeffs g, use g in auto)
qed

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)
  by (induct n arbitrary: q q' lc r d dr) (simp-all add: Let-def)

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* ≠ *0*
shows ∃ *a q*. *a* ≠ *0* ∧ *smult a f* = *g* ∗ *q* + *r r* = *0* ∨ *degree r* < *degree g*
proof −
let *?cg* = *coeff g* (*degree g*)
let *?cge* = *?cg* ^ (*Suc* (*degree f*) − *degree g*)
define *a* **where** *a* = *?cge*
from *r-def*[*unfolded pseudo-mod-def*] **obtain** *q* **where** *pdm*: *pseudo-divmod f g* = (*q*, *r*)
by (*cases pseudo-divmod f g*) *auto*
from *pseudo-divmod*[*OF g pdm*] **have** *id*: *smult a f* = *g* ∗ *q* + *r* **and** *r* = *0* ∨ *degree r* < *degree g*
by (*auto simp*: *a-def*)
show *r* = *0* ∨ *degree r* < *degree g* **by** *fact*
from *g* **have** *a* ≠ *0*
by (*auto simp*: *a-def*)
with *id* **show** ∃ *a q*. *a* ≠ *0* ∧ *smult a f* = *g* ∗ *q* + *r*
by *auto*
qed

lemma *fst-pseudo-divmod-main-as-divide-poly-main*:
assumes *d*: *d* ≠ *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 (*induct n arbitrary*: *q r dr*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
note *lc0* = *leading-coeff-neq-0*[*OF d*, *folded lc*]
then have *pseudo-divmod-main lc q r d dr* (*Suc n*) =
pseudo-divmod-main lc (*smult lc q* + *monom* (*coeff r dr*) *n*)
(*smult lc r* − *monom* (*coeff r dr*) *n* ∗ *d*) *d* (*dr* − *1*) *n*
by (*simp add*: *Let-def ac-simps*)
also have *fst* ... = *divide-poly-main lc*
(*smult* (*lc*^*n*) (*smult lc q* + *monom* (*coeff r dr*) *n*))
(*smult* (*lc*^*n*) (*smult lc r* − *monom* (*coeff r dr*) *n* ∗ *d*))
d (*dr* − *1*) *n*
by (*simp only*: *Suc*[*unfolded divide-poly-main.simps Let-def*])
also have ... = *divide-poly-main lc* (*smult* (*lc* ^ *Suc n*) *q*) (*smult* (*lc* ^ *Suc n*) *r*) *d dr* (*Suc n*)
unfolding *smult-monom smult-distribs mult-smult-left*[*symmetric*]
using *lc0* **by** (*simp add*: *Let-def ac-simps*)
finally show *?case* **.**
qed

4.31.3 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 $c \equiv$ *coeff g* (*degree g*) ^ (*Suc* (*degree f*) − *degree g*)
shows $f = g * smult\ (1/c)\ q + smult\ (1/c)\ r$
proof −
from *leading-coeff-neq-0*[*OF g*] **have** *c0*: $c \neq 0$
by (*auto simp*: *c-def*)
from *pseudo-divmod*(*1*)[*OF g* $*$, *folded c-def*] **have** *smult c f* = $g * q + r$
by *auto*
also have *smult* (*1 / c*) ... = $g *$ *smult* (*1 / c*) *q* + *smult* (*1 / c*) *r*
by (*simp add*: *smult-add-right*)
finally show *?thesis*
using *c0* **by** *auto*
qed

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*)
unfolding *lc* **by** (*subst fst-pseudo-divmod-main-as-divide-poly-main*) (*auto simp*: *d power-one-over*)

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 (*cases g* = *0*)
case *True*
show *?thesis*
unfolding *divide-poly-def pseudo-divmod-def Let-def f'-def True*
by (*simp add*: *divide-poly-main-0*)
next
case *False*
from *leading-coeff-neq-0*[*OF False*] **have** *degree f'* = *degree f*
by (*auto simp*: *f'-def*)
then show *?thesis*
using *length-coeffs-degree*[*of f'*] *length-coeffs-degree*[*of f*]
unfolding *divide-poly-def pseudo-divmod-def Let-def*
divide-poly-main-field[*OF False*]
length-coeffs-degree[*OF False*]
f'-def
by *force*
qed

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
  fix *p* :: *'a poly*
  show *unit-factor p* ∗ *normalize p* = *p*
  proof (*cases p* = *0*)
    case *True*
    then show *?thesis*
      by (*simp add*: *unit-factor-poly-def normalize-poly-def*)
  next
    case *False*
    then have *lead-coeff p* ≠ *0*
      by *simp*
    then have ∗: *unit-factor* (*lead-coeff p*) ≠ *0*
      using *unit-factor-is-unit* [*of lead-coeff p*] **by** *auto*
    then have *unit-factor* (*lead-coeff p*) *dvd 1*
      by (*auto intro*: *unit-factor-is-unit*)
    then have ∗∗: *unit-factor* (*lead-coeff p*) *dvd c* **for** *c*
      by (*rule dvd-trans*) *simp*
    have ∗∗∗: *unit-factor* (*lead-coeff p*) ∗ (*c div unit-factor* (*lead-coeff p*)) = *c* **for**
c
    proof −
      from ∗∗ **obtain** *b* **where** *c* = *unit-factor* (*lead-coeff p*) ∗ *b* **..**
      with *False* ∗ **show** *?thesis* **by** *simp*
    qed
    have *p div* [:*unit-factor* (*lead-coeff p*):] =
      map-poly (λ*c*. *c div unit-factor* (*lead-coeff p*)) *p*
      by (*simp add*: *const-poly-dvd-iff div-const-poly-conv-map-poly* ∗∗)
    then show *?thesis*
      by (*simp add*: *normalize-poly-def unit-factor-poly-def*
        smult-conv-map-poly map-poly-map-poly o-def ∗∗∗)
  qed
next
  fix *p* :: *'a poly*
  assume *is-unit p*
  then obtain *c* **where** *p*: *p* = [:*c*:] *c dvd 1*
    by (*auto simp*: *is-unit-poly-iff*)
  then show *unit-factor p* = *p*
    by (*simp add*: *unit-factor-poly-def monom-0 is-unit-unit-factor*)
next

fix $p :: 'a\ poly$
assume $p \neq 0$
then show *is-unit* (*unit-factor* p)
by (*simp add*: *unit-factor-poly-def monom-0 is-unit-poly-iff unit-factor-is-unit*)
next
fix $a\ b :: 'a\ poly$ **assume** *is-unit a*
thus *unit-factor* $(a * b) = a *$ *unit-factor b*
by (*auto simp*: *unit-factor-poly-def lead-coeff-mult unit-factor-mult elim*!: *is-unit-polyE*)
qed (*simp-all add*: *normalize-poly-def unit-factor-poly-def monom-0 lead-coeff-mult unit-factor-mult*)

end

instance *poly* :: ({*semidom-divide-unit-factor*,*idom-divide*,*normalization-semidom-multiplicative*})
normalization-semidom-multiplicative
by *intro-classes* (*auto simp*: *unit-factor-poly-def lead-coeff-mult unit-factor-mult*)

lemma *normalize-poly-eq-map-poly*: *normalize p* = *map-poly* ($\lambda x.$ *x div unit-factor* (*lead-coeff p*)) *p*
proof −
have [:*unit-factor* (*lead-coeff p*):] *dvd p*
by (*metis unit-factor-poly-def unit-factor-self*)
then show *?thesis*
by (*simp add*: *normalize-poly-def div-const-poly-conv-map-poly*)
qed

lemma *coeff-normalize* [*simp*]:
coeff (*normalize p*) *n* = *coeff p n div unit-factor* (*lead-coeff p*)
by (*simp add*: *normalize-poly-eq-map-poly coeff-map-poly*)

lemma *lead-coeff-normalize-field*:
fixes *p*::'*a*::{*field*,*semidom-divide-unit-factor*} *poly*
assumes *p*≠*0*
shows *lead-coeff* (*normalize p*) = *1*
by (*metis* (*no-types*, *lifting*) *assms coeff-normalize divide-self-if dvd-field-iff*
is-unit-unit-factor leading-coeff-0-iff normalize-eq-0-iff normalize-idem)

lemma *smult-normalize-field-eq*:
fixes *p*::'*a*::{*field*,*semidom-divide-unit-factor*} *poly*
shows *smult* (*lead-coeff p*) (*normalize p*) = *p*
proof (*rule poly-eqI*)
fix *n*
have *unit-factor* (*lead-coeff p*) = *lead-coeff p*
by (*metis dvd-field-iff is-unit-unit-factor unit-factor-0*)
then show *coeff* (*smult* (*lead-coeff p*) (*normalize p*)) *n* = *coeff p n*
by *simp*
qed

class *field-unit-factor* = *field* + *unit-factor* +

assumes *unit-factor-field* [*simp*]: *unit-factor* = *id*
begin

subclass *semidom-divide-unit-factor*
proof
fix *a*
assume *a* ≠ *0*
then have *1* = *a* ∗ *inverse a* **by** *simp*
then have *a dvd 1* **..**
then show *unit-factor a dvd 1* **by** *simp*
qed *simp-all*

end

lemma *unit-factor-pCons*:
unit-factor (*pCons a p*) = (*if p* = *0 then* [:*unit-factor a*:] *else unit-factor p*)
by (*simp add*: *unit-factor-poly-def*)

lemma *normalize-monom* [*simp*]: *normalize* (*monom a n*) = *monom* (*normalize a*) *n*
by (*cases a* = *0*) (*simp-all add*: *map-poly-monom normalize-poly-eq-map-poly degree-monom-eq*)

lemma *unit-factor-monom* [*simp*]: *unit-factor* (*monom a n*) = [:*unit-factor a*:]
by (*cases a* = *0*) (*simp-all add*: *unit-factor-poly-def degree-monom-eq*)

lemma *normalize-const-poly*: *normalize* [:*c*:] = [:*normalize c*:]
by (*simp add*: *normalize-poly-eq-map-poly map-poly-pCons*)

lemma *normalize-smult*:
fixes *c* :: *'a* :: {*normalization-semidom-multiplicative*, *idom-divide*}
shows *normalize* (*smult c p*) = *smult* (*normalize c*) (*normalize p*)
proof −
have *smult c p* = [:*c*:] ∗ *p* **by** *simp*
also have *normalize* ... = *smult* (*normalize c*) (*normalize p*)
by (*subst normalize-mult*) (*simp add*: *normalize-const-poly*)
finally show *?thesis* **.**
qed

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

```
  fix x y :: 'a poly
  show x div y * y + x mod y = x
  proof (cases y = 0)
    case True
    then show ?thesis
      by (simp add: divide-poly-0 mod-poly-def)
  next
    case False
    then have pseudo-divmod (smult ((1 / lead-coeff y) ^ (Suc (degree x) - degree
y)) x) y =
        (x div y, x mod y)
      by (simp add: divide-poly-field mod-poly-def pseudo-mod-def)
    with False pseudo-divmod [OF False this] show ?thesis
      by (simp add: power-mult-distrib [symmetric] ac-simps)
  qed
qed

end

lemma pseudo-divmod-eq-div-mod:
  ‹pseudo-divmod f g = (f div g, f mod g)› if ‹lead-coeff g = 1›
  using that by (auto simp add: divide-poly-field mod-poly-def pseudo-mod-def)

lemma degree-mod-less-degree:
  ‹degree (x mod y) < degree y› if ‹y ≠ 0› ‹¬ y dvd x›
proof −
  from pseudo-mod(2) [of y] ‹y ≠ 0›
  have ∗: ‹pseudo-mod f y ≠ 0 ⟹ degree (pseudo-mod f y) < degree y› for f
    by blast
  from ‹¬ y dvd x› have ‹x mod y ≠ 0›
    by blast
  with ‹y ≠ 0› show ?thesis
    by (auto simp add: mod-poly-def intro: ∗)
qed

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
  show ‹(q * p + r) div p = q› if ‹p ≠ 0›
    and ‹euclidean-size r < euclidean-size p› for q p r :: ‹'a poly›
  proof (cases ‹r = 0›)
    case True
```

```
    with that show ?thesis
      by simp
  next
    case False
    with ‹p ≠ 0› ‹euclidean-size r < euclidean-size p›
    have ‹degree r < degree p›
      by (simp add: euclidean-size-poly-def)
    with ‹r ≠ 0› have ‹¬ p dvd r›
      by (auto dest: dvd-imp-degree)
    have ‹(q * p + r) div p = q ∧ (q * p + r) mod p = r›
    proof (rule ccontr)
      assume ‹¬ ?thesis›
      moreover have *: ‹((q * p + r) div p - q) * p = r - (q * p + r) mod p›
        by (simp add: algebra-simps)
      ultimately have ‹(q * p + r) div p ≠ q› and ‹(q * p + r) mod p ≠ r›
        using ‹p ≠ 0› by auto
      from ‹¬ p dvd r› have ‹¬ p dvd (q * p + r)›
        by simp
      with ‹p ≠ 0› have ‹degree ((q * p + r) mod p) < degree p›
        by (rule degree-mod-less-degree)
      with ‹degree r < degree p› ‹(q * p + r) mod p ≠ r›
      have ‹degree (r - (q * p + r) mod p) < degree p›
        by (auto intro: degree-diff-less)
      also have ‹degree p ≤ degree ((q * p + r) div p - q) + degree p›
        by simp
      also from ‹(q * p + r) div p ≠ q› ‹p ≠ 0›
      have ‹... = degree (((q * p + r) div p - q) * p)›
        by (simp add: degree-mult-eq)
      also from * have ‹... = degree (r - (q * p + r) mod p)›
        by simp
      finally have ‹degree (r - (q * p + r) mod p) < degree (r - (q * p + r) mod
p)› .
      then show False
        by simp
    qed
    then show ‹(q * p + r) div p = q› ..
  qed
qed (auto simp: euclidean-size-poly-def degree-mult-eq power-add intro: degree-mod-less-degree)

end

lemma euclidean-relation-polyI [case-names by0 divides euclidean-relation]:
  ‹(x div y, x mod y) = (q, r)›
    if by0: ‹y = 0 ⟹ q = 0 ∧ r = x›
    and divides: ‹y ≠ 0 ⟹ y dvd x ⟹ r = 0 ∧ x = q * y›
    and euclidean-relation: ‹y ≠ 0 ⟹ ¬ y dvd x ⟹ degree r < degree y ∧ x = q
* y + r›
  by (rule euclidean-relationI)
    (use that in ‹simp-all add: euclidean-size-poly-def›)
```

lemma *div-poly-eq-0-iff*:
‹*x div y = 0* ⟷ *x = 0* ∨ *y = 0* ∨ *degree x < degree y*› **for** *x y* :: ‹'*a*::*field poly*›
by (*simp add*: *unique-euclidean-semiring-class.div-eq-0-iff euclidean-size-poly-def*)

lemma *div-poly-less*:
‹*x div y = 0*› **if** ‹*degree x < degree y*› **for** *x y* :: ‹'*a*::*field poly*›
using *that* **by** (*simp add*: *div-poly-eq-0-iff*)

lemma *mod-poly-less*:
‹*x mod y = x*› **if** ‹*degree x < degree y*›
using *that* **by** (*simp add*: *mod-eq-self-iff-div-eq-0 div-poly-eq-0-iff*)

lemma *degree-div-less*:
‹*degree* (*x div y*) < *degree x*›
if ‹*degree x > 0*› ‹*degree y > 0*›
for *x y* :: ‹'*a*::*field poly*›
proof (*cases* ‹*x div y = 0*›)
case *True*
with ‹*degree x > 0*› **show** *?thesis*
by *simp*
next
case *False*
from *that* **have** ‹*x ≠ 0*› ‹*y ≠ 0*›
and ∗: ‹*degree* (*x div y ∗ y + x mod y*) > *0*›
by *auto*
show *?thesis*
proof (*cases* ‹*y dvd x*›)
case *True*
then obtain *z* **where** ‹*x = y ∗ z*› **..**
then have ‹*degree* (*x div y*) < *degree* (*x div y ∗ y*)›
using ‹*y ≠ 0*› ‹*x ≠ 0*› ‹*degree y > 0*› **by** (*simp add*: *degree-mult-eq*)
with ‹*y dvd x*› **show** *?thesis*
by *simp*
next
case *False*
with ‹*y ≠ 0*› **have** ‹*degree* (*x mod y*) < *degree y*›
by (*rule degree-mod-less-degree*)
with ‹*y ≠ 0*› ‹*x div y ≠ 0*› **have** ‹*degree* (*x mod y*) < *degree* (*x div y ∗ y*)›
by (*simp add*: *degree-mult-eq*)
then have ‹*degree* (*x div y ∗ y + x mod y*) = *degree* (*x div y ∗ y*)›
by (*rule degree-add-eq-left*)
with ‹*y ≠ 0*› ‹*x div y ≠ 0*› ‹*degree y > 0*› **show** *?thesis*
by (*simp add*: *degree-mult-eq*)
qed
qed

lemma *degree-mod-less'*: *b ≠ 0* ⟹ *a mod b ≠ 0* ⟹ *degree* (*a mod b*) < *degree b*
by (*rule degree-mod-less-degree*) *auto*

lemma *degree-mod-less*: $y \neq 0 \Longrightarrow x \bmod y = 0 \vee \text{degree } (x \bmod y) < \text{degree } y$
using *degree-mod-less'* **by** *blast*

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 −
have ‹(*smult a x div y*, *smult a x mod y*) = (*smult a* (*x div y*), *smult a* (*x mod y*))›
proof (*cases* ‹*a* = *0*›)
case *True*
then show *?thesis*
by *simp*
next
case *False*
show *?thesis*
by (*rule euclidean-relation-polyI*)
(*use False* **in** ‹*simp-all add*: *dvd-smult-iff degree-mod-less-degree flip*: *smult-add-right*›)
qed
then show *?Q* **and** *?R*
by *simp-all*
qed

lemma *poly-div-minus-left* [*simp*]: (− *x*) *div y* = − (*x div y*)
for *x y* :: '*a*::*field poly*
using *div-smult-left* [*of* − *1*::'*a*] **by** *simp*

lemma *poly-mod-minus-left* [*simp*]: (− *x*) *mod y* = − (*x mod y*)
for *x y* :: '*a*::*field poly*
using *mod-smult-left* [*of* − *1*::'*a*] **by** *simp*

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 −
have ‹((*x* + *y*) *div z*, (*x* + *y*) *mod z*) = (*x div z* + *y div z*, *x mod z* + *y mod z*)›
proof (*induction rule*: *euclidean-relation-polyI*)
case *by0*
then show *?case* **by** *simp*
next
case *divides*
then obtain *w* **where** ‹*x* + *y* = *z* ∗ *w*›
by *blast*
then have *y*: ‹*y* = *z* ∗ *w* − *x*›
by (*simp add*: *algebra-simps*)
from ‹*z* ≠ *0*› **show** *?case*
using *mod-mult-self4* [*of z w* ‹− *x*›] *div-mult-self4* [*of z w* ‹− *x*›]
by (*simp add*: *algebra-simps y*)

```
  next
    case euclidean-relation
    then have ‹degree (x mod z + y mod z) < degree z›
      using degree-mod-less-degree [of z x] degree-mod-less-degree [of z y]
        dvd-add-right-iff [of z x y] dvd-add-left-iff [of z y x]
      by (cases ‹z dvd x ∨ z dvd y›) (auto intro: degree-add-less)
    moreover have ‹x + y = (x div z + y div z) * z + (x mod z + y mod z)›
      by (simp add: algebra-simps)
    ultimately show ?case
      by simp
  qed
  then show ?Q and ?R
    by simp-all
qed

lemma poly-div-diff-left: (x - y) div z = x div z - y div z
  for x y z :: 'a::field poly
  by (simp only: diff-conv-add-uminus poly-div-add-left poly-div-minus-left)

lemma poly-mod-diff-left: (x - y) mod z = x mod z - y mod z
  for x y z :: 'a::field poly
  by (simp only: diff-conv-add-uminus poly-mod-add-left poly-mod-minus-left)

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 -
  have ‹(x div smult a y, x mod smult a y) = (smult (inverse a) (x div y), (if a =
0 then x else x mod y))›
  proof (induction rule: euclidean-relation-polyI)
    case by0
    then show ?case by auto
  next
    case divides
    moreover define w where ‹w = x div y›
    ultimately have ‹x = y * w›
      by (simp add: smult-dvd-iff)
    with divides show ?case
      by simp
  next
    case euclidean-relation
    then show ?case
      by (simp add: smult-dvd-iff degree-mod-less-degree)
  qed
  then show ?Q and ?R
    by simp-all
qed

lemma mod-mult-unit-eq:
  ‹x mod (z * y) = x mod y›
```

```
  if ‹is-unit z›
  for x y z :: ‹'a::field poly›
proof (cases ‹y = 0›)
  case True
  then show ?thesis
    by simp
next
  case False
  moreover have ‹z ≠ 0›
    using that by auto
  moreover define a where ‹a = lead-coeff z›
  ultimately have ‹z = [:a:]› ‹a ≠ 0›
    using that monom-0 [of a] by (simp-all add: is-unit-monom-trivial)
  then show ?thesis
    by (simp add: mod-smult-right)
qed

lemma poly-div-minus-right [simp]: x div (− y) = − (x div y)
  for x y :: 'a::field poly
  using div-smult-right [of - − 1::'a] by (simp add: nonzero-inverse-minus-eq)

lemma poly-mod-minus-right [simp]: x mod (− y) = x mod y
  for x y :: 'a::field poly
  using mod-smult-right [of - − 1::'a] by simp

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 −
  have ‹(x div (y * z), x mod (y * z)) = ((x div y) div z, y * (x div y mod z) + x
mod y)›
  proof (induction rule: euclidean-relation-polyI)
    case by0
    then show ?case by auto
  next
    case divides
    then show ?case by auto
  next
    case euclidean-relation
    then have ‹y ≠ 0› ‹z ≠ 0›
      by simp-all
    with ‹¬ y * z dvd x› have ‹degree (y * (x div y mod z) + x mod y) < degree
(y * z)›
      using degree-mod-less-degree [of y x] degree-mod-less-degree [of z ‹x div y›]
        degree-add-eq-left [of ‹x mod y› ‹y * (x div y mod z)›]
        by (cases ‹z dvd x div y›; cases ‹y dvd x›)
          (auto simp add: degree-mult-eq not-dvd-imp-mod-neq-0 dvd-div-iff-mult)
    moreover have ‹x = x div y div z * (y * z) + (y * (x div y mod z) + x mod
```

y)›
    by (*simp add*: *field-simps flip*: *distrib-left*)
  ultimately show *?case*
    by *simp*
 qed
 then show *?Q* **and** *?R*
  by *simp-all*
qed

lemma *dvd-pCons-imp-dvd-pCons-mod*:
 ‹*y dvd pCons a* (*x mod y*)› **if** ‹*y dvd pCons a x*›
proof −
 have ‹*pCons a x* = *pCons a* (*x div y* ∗ *y* + *x mod y*)›
  by *simp*
 also have ‹. . . = *pCons* 0 (*x div y* ∗ *y*) + *pCons a* (*x mod y*)›
  by *simp*
 also have ‹*pCons* 0 (*x div y* ∗ *y*) = (*x div y* ∗ *monom* 1 (*Suc* 0)) ∗ *y*›
  by (*simp add*: *monom-Suc*)
 finally show ‹*y dvd pCons a* (*x mod y*)›
  using ‹*y dvd pCons a x*› **by** *simp*
qed

lemma *degree-less-if-less-eqI*:
 ‹*degree x* < *degree y*› **if** ‹*degree x* ≤ *degree y*› ‹*coeff x* (*degree y*) = 0› ‹*x* ≠ 0›
proof (*cases* ‹*degree x* = *degree y*›)
 case *True*
 with ‹*coeff x* (*degree y*) = 0› **have** ‹*lead-coeff x* = 0›
  by *simp*
 then have ‹*x* = 0›
  by *simp*
 with ‹*x* ≠ 0› **show** *?thesis*
  by *simp*
next
 case *False*
 with ‹*degree x* ≤ *degree y*› **show** *?thesis*
  by *simp*
qed

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 −
 have ‹*?Q*› **and** ‹*?R*› **if** ‹*q* = 0›
  using *that* **by** *simp-all*
 moreover have ‹*?Q*› **and** ‹*?R*› **if** ‹*q* ≠ 0›

```
proof −
  define b where ‹b = coeff (pCons a (p mod q)) (degree q) / lead-coeff q›
  have ‹(pCons a p div q, pCons a p mod q) =
    (pCons b (p div q), (pCons a (p mod q) − smult b q))› (is ‹- = (?q, ?r)›)
  proof (induction rule: euclidean-relation-polyI)
    case by0
    with ‹q ≠ 0› show ?case by simp
  next
    case divides
    show ?case
    proof (cases ‹pCons a (p mod q) = 0›)
      case True
      then show ?thesis
        by (auto simp add: b-def)
    next
      case False
      have ‹q dvd pCons a (p mod q)›
        using ‹q dvd pCons a p› by (rule dvd-pCons-imp-dvd-pCons-mod)
      then obtain s where ∗: ‹pCons a (p mod q) = q ∗ s› ..
      with False have ‹s ≠ 0›
        by auto
      from ‹q ≠ 0› have ‹degree (pCons a (p mod q)) ≤ degree q›
        by (auto simp add: Suc-le-eq intro: degree-mod-less-degree)
      moreover from ‹s ≠ 0› have ‹degree q ≤ degree (pCons a (p mod q))›
        by (simp add: degree-mult-right-le ∗)
      ultimately have ‹degree (pCons a (p mod q)) = degree q›
        by (rule order.antisym)
      with ‹s ≠ 0› ‹q ≠ 0› have ‹degree s = 0›
        by (simp add: ∗ degree-mult-eq)
      then obtain c where ‹s = [:c:]›
        by (rule degree-eq-zeroE)
      also have ‹c = b›
        using ‹q ≠ 0› by (simp add: b-def ∗ ‹s = [:c:]›)
      finally have ‹smult b q = pCons a (p mod q)›
        by (simp add: ∗)
      then show ?thesis
        by simp
    qed
  next
    case euclidean-relation
    then have ‹degree q > 0›
      using is-unit-iff-degree by blast
    from ‹q ≠ 0› have ‹degree (pCons a (p mod q)) ≤ degree q›
      by (auto simp add: Suc-le-eq intro: degree-mod-less-degree)
    moreover have ‹degree (smult b q) ≤ degree q›
      by (rule degree-smult-le)
    ultimately have ‹degree (pCons a (p mod q) − smult b q) ≤ degree q›
      by (rule degree-diff-le)
    moreover have ‹coeff (pCons a (p mod q) − smult b q) (degree q) = 0›
```

```
      using ‹degree q > 0› by (auto simp add: b-def)
    ultimately have ‹degree (pCons a (p mod q) - smult b q) < degree q›
      using ‹degree q > 0›
      by (cases ‹pCons a (p mod q) = smult b q›)
        (auto intro: degree-less-if-less-eqI)
    then show ?case
      by simp
  qed
  with ‹q ≠ 0› show ?Q and ?R
    by (simp-all add: b-def)
 qed
 ultimately show ?Q and ?R
  by simp-all
qed

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))
 by (rule sym, induct p) (auto simp: div-pCons-eq mod-pCons-eq Let-def)

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
 unfolding b-def
 by (simp add: mod-pCons-eq)
```

4.31.4 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
  by (induct x y rule: minus-poly-rev-list.induct) auto

lemma length-minus-poly-rev-list [simp]: length (minus-poly-rev-list xs ys) = length
xs
  by (induct xs ys rule: minus-poly-rev-list.induct) auto

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
  by(cases a = 0) (simp-all add: minus-zero-does-nothing)

lemma Poly-append: Poly (a @ b) = Poly a + monom 1 (length a) * Poly b
  for a :: 'a::comm-semiring-1 list
  by (induct a) (auto simp: monom-0 monom-Suc)

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 (induct rev p rev q arbitrary: p q rule: minus-poly-rev-list.induct)
  case (1 x xs y ys)
  then have length (rev q) ≤ length (rev p)
    by simp
  from this[folded 1(2,3)] have ys-xs: length ys ≤ length xs
    by simp
  then have *: Poly (rev (minus-poly-rev-list xs ys)) =
      Poly (rev xs) − monom 1 (length xs − length ys) * Poly (rev ys)
    by (subst 1.hyps(1)[of rev xs rev ys, unfolded rev-rev-ident length-rev]) auto
  have Poly p − monom 1 (length p − length q) * Poly q =
    Poly (rev (rev p)) − monom 1 (length (rev (rev p)) − length (rev (rev q))) *
Poly (rev (rev q))
    by simp
  also have ... =
      Poly (rev (x # xs)) − monom 1 (length (x # xs) − length (y # ys)) * Poly
(rev (y # ys))
    unfolding 1(2,3) by simp
  also from ys-xs have ... =
    Poly (rev xs) + monom x (length xs) −
      (monom 1 (length xs − length ys) * Poly (rev ys) + monom y (length xs))
    by (simp add: Poly-append distrib-left mult-monom smult-monom)
  also have ... = Poly (rev (minus-poly-rev-list xs ys)) + monom (x − y) (length
```

xs)
  unfolding ∗ *diff-monom*[*symmetric*] **by** *simp*
 finally show *?case*
  by (*simp add*: *1*(*2*,*3*)[*symmetric*] *smult-monom Poly-append*)
qed *auto*

lemma *smult-monom-mult*: *smult a* (*monom b n* ∗ *f*) = *monom* (*a* ∗ *b*) *n* ∗ *f*
 using *smult-monom* [*of a - n*] **by** (*metis mult-smult-left*)

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 (*induct r*)
 case *Nil*
 then show *?case* **by** *simp*
next
 case (*Cons a rs*)
 then show *?case* **by** (*cases rev d*) (*simp-all add*: *ac-simps*)
qed

lemma *Poly-map*: *Poly* (*map* ((∗) *a*) *p*) = *smult a* (*Poly p*)
proof (*induct p*)
 case *Nil*
 then show *?case* **by** *simp*
next
 case (*Cons x xs*)
 then show *?case* **by** (*cases Poly xs* = *0*) *auto*
qed

lemma *last-coeff-is-hd*: *xs* ≠ [] ⟹ *coeff* (*Poly xs*) (*length xs* − *1*) = *hd* (*rev xs*)
 by (*simp-all add*: *hd-conv-nth rev-nth nth-default-nth nth-append*)

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'*)
 using *assms*(*4* −)
proof (*induct n arbitrary*: *r q*)
 case (*Suc n*)
 from *Suc.prems* **have** ∗: ¬ *Suc* (*length r*) ≤ *length d*
  by *simp*
 with ‹*d* ≠ []› **have** *r* ≠ []
  using *Suc-leI length-greater-0-conv list.size*(*3*) **by** *fastforce*

let *?a* = (*hd* (*rev r*))
let *?rr* = *map* ((∗) *lc*) (*rev r*)
let *?rrr* = *rev* (*tl* (*minus-poly-rev-list ?rr* (*map* ((∗) *?a*) (*rev d*))))
let *?qq* = *cCons ?a* (*map* ((∗) *lc*) *q*)
from ∗ *Suc*(*3*) **have** *n*: *n* = (*1* + *length r* − *length d* − *1*)
by *simp*
from ∗ **have** *rr-val*:(*length ?rrr*) = (*length r* − *1*)
by *auto*
with ‹*r* ≠ []› ∗ **have** *rr-smaller*: (*1* + *length r* − *length d* − *1*) = (*1* + *length ?rrr* − *length d*)
by *auto*
from ∗ **have** *id*: *Suc* (*length r*) − *length d* = *Suc* (*length r* − *length d*)
by *auto*
from *Suc.prems* ∗
have *pseudo-divmod-main-list lc ?qq* (*rev ?rrr*) (*rev d*) (*1* + *length r* − *length d* − *1*) = (*q′*, *rev r′*)
by (*simp add*: *Let-def if-0-minus-poly-rev-list id*)
with *n* **have** *v*: *pseudo-divmod-main-list lc ?qq* (*rev ?rrr*) (*rev d*) *n* = (*q′*, *rev r′*)
by *auto*
from ∗ **have** *sucrr*:*Suc* (*length r*) − *length d* = *Suc* (*length r* − *length d*)
using *Suc-diff-le not-less-eq-eq* **by** *blast*
from *Suc*(*3*) ‹*r* ≠ []› **have** *n-ok* : *n* = *1* + (*length ?rrr*) − *length d*
by *simp*
have *cong*: ⋀*x1 x2 x3 x4 y1 y2 y3 y4*. *x1* = *y1* ⟹ *x2* = *y2* ⟹ *x3* = *y3* ⟹ *x4* = *y4* ⟹
pseudo-divmod-main lc x1 x2 x3 x4 n = *pseudo-divmod-main lc y1 y2 y3 y4 n*
by *simp*
have *hd-rev*: *coeff* (*Poly r*) (*length r* − *Suc 0*) = *hd* (*rev r*)
using *last-coeff-is-hd*[*OF* ‹*r* ≠ []›] **by** *simp*
show *?case*
unfolding *Suc.hyps*(*1*)[*OF v n-ok*, *symmetric*] *pseudo-divmod-main.simps Let-def*
proof (*rule cong*[*OF - - refl*], *goal-cases*)
case *1*
show *?case*
by (*simp add*: *monom-Suc hd-rev*[*symmetric*] *smult-monom Poly-map*)
next
case *2*
show *?case*
proof (*subst Poly-on-rev-starting-with-0*, *goal-cases*)
show *hd* (*minus-poly-rev-list* (*map* ((∗) *lc*) (*rev r*)) (*map* ((∗) (*hd* (*rev r*))) (*rev d*))) = *0*
by (*fold lc*, *subst head-minus-poly-rev-list*, *insert* ∗ ‹*d* ≠ []›, *auto*)
from ∗ **have** *length d* ≤ *length r*
by *simp*
then show *smult lc* (*Poly r*) − *monom* (*coeff* (*Poly r*) (*length r* − *1*)) *n* ∗ *Poly d* =
Poly (*rev* (*minus-poly-rev-list* (*map* ((∗) *lc*) (*rev r*)) (*map* ((∗) (*hd* (*rev r*))) (*rev d*))))
by (*fold rev-map*) (*auto simp add*: *n smult-monom-mult Poly-map hd-rev*

[*symmetric*]
        minus-poly-rev-list)
  qed
 qed *simp*
qed *simp*

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 (*cases g = 0*)
 case *False*
 then have *last* (*coeffs g*) ≠ *0*
  and *last* (*coeffs g*) = *lead-coeff g*
  and *coeffs g* ≠ []
  by (*simp-all add*: *last-coeffs-eq-coeff-degree*)
 moreover obtain *q r* **where** *qr*: *pseudo-divmod-main-list*
  (*last* (*coeffs g*)) (*rev* [])
  (*rev* (*coeffs f*)) (*rev* (*coeffs g*))
  (*1* + *length* (*coeffs f*) −
  length (*coeffs g*)) = (*q*, *rev* (*rev r*))
  by *force*
 ultimately have (*Poly q*, *Poly* (*rev r*)) = *pseudo-divmod-main* (*lead-coeff g*) *0 f g*
  (*length* (*coeffs f*) − *Suc 0*) (*Suc* (*length* (*coeffs f*)) − *length* (*coeffs g*))
  by (*subst pseudo-divmod-main-list-invar* [*symmetric*]) *auto*
 moreover have *pseudo-divmod-main-list*
  (*hd* (*rev* (*coeffs g*))) []
  (*rev* (*coeffs f*)) (*rev* (*coeffs g*))
  (*1* + *length* (*coeffs f*) −
  length (*coeffs g*)) = (*q*, *r*)
  by (*metis hd-rev qr rev.simps*(*1*) *rev-swap*)
 ultimately show *?thesis*
  by (*simp add*: *degree-eq-length-coeffs pseudo-divmod-def pseudo-divmod-list-def*)
next
 case *True*
 then show *?thesis*
  by (*auto simp add*: *pseudo-divmod-def pseudo-divmod-list-def*)
qed

lemma *pseudo-mod-main-list*:
 snd (*pseudo-divmod-main-list l q xs ys n*) = *pseudo-mod-main-list l xs ys n*
 by (*induct n arbitrary*: *l q xs ys*) (*auto simp*: *Let-def*)

lemma *pseudo-mod-impl*[*code*]: *pseudo-mod f g = poly-of-list* (*pseudo-mod-list* (*coeffs f*) (*coeffs g*))
proof −
 have *snd-case*: ⋀*f g p*. *snd* ((λ(*x*,*y*). (*f x*, *g y*)) *p*) = *g* (*snd p*)
  by *auto*

show *?thesis*
unfolding *pseudo-mod-def pseudo-divmod-impl pseudo-divmod-list-def pseudo-mod-list-def Let-def*
by (*simp add*: *snd-case pseudo-mod-main-list*)
qed

4.31.5 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 (*cases* ‹*g* = *0*›)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
define *ilc* **where** ‹*ilc* = *inverse* (*lead-coeff g*)›
define *h* **where** ‹*h* = *smult ilc g*›
from *False* **have** ‹*lead-coeff h* = *1*›
and ‹*ilc* ≠ *0*›
by (*auto simp*: *h-def ilc-def*)
define *q r* **where** ‹*q* = *f div h*› **and** ‹*r* = *f mod h*›
with ‹*lead-coeff h* = *1*› **have** *p*: ‹*pseudo-divmod f h* = (*q*, *r*)›
by (*simp add*: *pseudo-divmod-eq-div-mod*)
from ‹*ilc* ≠ *0*› **have** ‹(*f div g*, *f mod g*) = (*smult ilc q*, *r*)›
by (*auto simp*: *h-def div-smult-right mod-smult-right q-def r-def*)
also have ‹(*smult ilc q*, *r*) = *?r*›
using ‹*g* ≠ *0*› **by** (*auto simp*: *Let-def p simp flip*: *h-def ilc-def*)
finally show *?thesis* **.**
qed

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 −
  note d = pdivmod-via-pseudo-divmod pseudo-divmod-impl pseudo-divmod-list-def
  show ?thesis
  proof (cases g = 0)
    case True
    with d show ?thesis by auto
  next
    case False
    define ilc where ilc = inverse (coeff g (degree g))
    from False have ilc: ilc ≠ 0
      by (auto simp: ilc-def)
    with False have id: g = 0 ⟷ False coeffs g = [] ⟷ False
      last (coeffs g) = coeff g (degree g)
      coeffs (smult ilc g) = [] ⟷ False
      by (auto simp: last-coeffs-eq-coeff-degree)
    have id2: hd (rev (coeffs (smult ilc g))) = 1
      by (subst hd-rev, insert id ilc, auto simp: coeffs-smult, subst last-map, auto
simp: id ilc-def)
    have id3: length (coeffs (smult ilc g)) = length (coeffs g)
      rev (coeffs (smult ilc g)) = rev (map ((*) ilc) (coeffs g))
      unfolding coeffs-smult using ilc by auto
    obtain q r where pair:
      pseudo-divmod-main-list 1 [] (rev (coeffs f)) (rev (map ((*) ilc) (coeffs g)))
        (1 + length (coeffs f) − length (coeffs g)) = (q, r)
      by force
    show ?thesis
      unfolding d Let-def id if-False ilc-def[symmetric] map-prod-def[symmetric]
id2
      unfolding id3 pair map-prod-def split
      by (auto simp: Poly-map)
  qed
qed

lemma pseudo-divmod-main-list-1: pseudo-divmod-main-list 1 = divmod-poly-one-main-list
proof (intro ext, goal-cases)
  case (1 q r d n)
  have ∗: map ((*) 1) xs = xs for xs :: 'a list
    by (induct xs) auto
  show ?case
    by (induct n arbitrary: q r d) (auto simp: ∗ Let-def)
qed

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 [])
  by (simp add: Let-def minus-zero-does-nothing)

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
  by (induct n arbitrary: q r d) (auto simp: Let-def)

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 −
  have snd (f div g, f mod g) = ?rhs
    unfolding pdivmod-via-divmod-list Let-def mod-poly-one-main-list [symmetric,
of - - - Nil]
    by (auto split: prod.splits)
  then show ?thesis by simp
qed

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 (intro ext)
  fix f g :: 'a poly
  have fst (f div g, f mod g) = div-field-poly-impl f g
    unfolding div-field-poly-impl-def pdivmod-via-divmod-list Let-def
    by (auto split: prod.splits)
  then show f div g =  div-field-poly-impl f g
    by simp
qed

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
  using assms(4 −)
proof (induct n arbitrary: r q)
  case (Suc n)
  from Suc.prems have ifCond: ¬ Suc (length r) ≤ length d
    by simp
  with d have r: r ≠ []
    using Suc-leI length-greater-0-conv list.size(3) by fastforce
```

```
  then obtain rr lcr where r: r = rr @ [lcr]
    by (cases r rule: rev-cases) auto
  from d lc obtain dd where d: d = dd @ [lc]
    by (cases d rule: rev-cases) auto
  from Suc(2) ifCond have n: n = 1 + length rr − length d
    by (auto simp: r)
  from ifCond have len: length dd ≤ length rr
    by (simp add: r d)
  show ?case
  proof (cases lcr div lc * lc = lcr)
    case False
    with r d show ?thesis
      unfolding Suc(2)[symmetric]
      by (auto simp add: Let-def nth-default-append)
  next
    case True
    with r d have id:
      ?thesis ⟷
        Poly (divide-poly-main-list lc (cCons (lcr div lc) q)
          (rev (rev (minus-poly-rev-list (rev rr) (rev (map ((*) (lcr div lc)) dd)))))
(rev d) n) =
          divide-poly-main lc
            (monom 1 (Suc n) * Poly q + monom (lcr div lc) n)
            (Poly r − monom (lcr div lc) n * Poly d)
            (Poly d) (length rr − 1) n
      by (cases r rule: rev-cases; cases d rule: rev-cases)
        (auto simp add: Let-def rev-map nth-default-append)
    have cong: ⋀x1 x2 x3 x4 y1 y2 y3 y4. x1 = y1 ⟹ x2 = y2 ⟹ x3 = y3 ⟹
x4 = y4 ⟹
        divide-poly-main lc x1 x2 x3 x4 n = divide-poly-main lc y1 y2 y3 y4 n
      by simp
    show ?thesis
      unfolding id
    proof (subst Suc(1), simp add: n,
        subst minus-poly-rev-list, force simp: len, rule cong[OF - - refl], goal-cases)
      case 2
      have monom lcr (length rr) = monom (lcr div lc) (length rr − length dd) *
monom lc (length dd)
        by (simp add: mult-monom len True)
      then show ?case unfolding r d Poly-append n ring-distribs
        by (auto simp: Poly-map smult-monom smult-monom-mult)
    qed (auto simp: len monom-Suc smult-monom)
  qed
qed simp

lemma divide-poly-list[code]: f div g = divide-poly-list f g
proof −
  note d = divide-poly-def divide-poly-list-def
  show ?thesis
```

```
  proof (cases g = 0)
    case True
    show ?thesis by (auto simp: d True)
  next
    case False
    then obtain cg lcg where cg: coeffs g = cg @ [lcg]
      by (cases coeffs g rule: rev-cases) auto
    with False have id: (g = 0) = False (cg @ [lcg] = []) = False
      by auto
    from cg False have lcg: coeff g (degree g) = lcg
      using last-coeffs-eq-coeff-degree last-snoc by force
    with False have lcg ≠ 0 by auto
    from cg Poly-coeffs [of g] have ltp: Poly (cg @ [lcg]) = g
      by auto
    show ?thesis
      unfolding d cg Let-def id if-False poly-of-list-def
      by (subst divide-poly-main-list, insert False cg ‹lcg ≠ 0›)
        (auto simp: lcg ltp, simp add: degree-eq-length-coeffs)
  qed
qed

lemma poly-mod:
  poly (p mod q) x = poly p x if poly q x = 0
proof −
  from that have poly (p mod q) x = poly (p div q * q) x + poly (p mod q) x
    by simp
  also have ... = poly p x
    by (simp only: poly-add [symmetric]) simp
  finally show ?thesis .
qed
```

4.32 Primality and irreducibility in polynomial rings

```
lemma prod-mset-const-poly: (∏ x∈#A. [:f x:]) = [:prod-mset (image-mset f A):]
  by (induct A) (simp-all add: ac-simps)

lemma irreducible-const-poly-iff:
  fixes c :: 'a :: {comm-semiring-1,semiring-no-zero-divisors}
  shows irreducible [:c:] ⟷ irreducible c
proof
  assume A: irreducible c
  show irreducible [:c:]
  proof (rule irreducibleI)
    fix a b assume ab: [:c:] = a * b
    hence degree [:c:] = degree (a * b) by (simp only: )
    also from A ab have a ≠ 0 b ≠ 0 by auto
    hence degree (a * b) = degree a + degree b by (simp add: degree-mult-eq)
    finally have degree a = 0 degree b = 0 by auto
   then obtain a' b' where ab': a = [:a':] b = [:b':] by (auto elim!: degree-eq-zeroE)
```

```
    from ab have coeff [:c:] 0 = coeff (a * b) 0 by (simp only: )
    hence c = a' * b' by (simp add: ab' mult-ac)
    from A and this have a' dvd 1 ∨ b' dvd 1 by (rule irreducibleD)
    with ab' show a dvd 1 ∨ b dvd 1
      by (auto simp add: is-unit-const-poly-iff)
  qed (insert A, auto simp: irreducible-def is-unit-poly-iff)
next
  assume A: irreducible [:c:]
  then have c ≠ 0 and ¬ c dvd 1
    by (auto simp add: irreducible-def is-unit-const-poly-iff)
  then show irreducible c
  proof (rule irreducibleI)
    fix a b assume ab: c = a * b
    hence [:c:] = [:a:] * [:b:] by (simp add: mult-ac)
    from A and this have [:a:] dvd 1 ∨ [:b:] dvd 1 by (rule irreducibleD)
    then show a dvd 1 ∨ b dvd 1
      by (auto simp add: is-unit-const-poly-iff)
  qed
qed

lemma lift-prime-elem-poly:
  assumes prime-elem (c :: 'a :: semidom)
  shows   prime-elem [:c:]
proof (rule prime-elemI)
  fix a b assume *: [:c:] dvd a * b
  from * have dvd: c dvd coeff (a * b) n for n
    by (subst (asm) const-poly-dvd-iff) blast
  {
    define m where m = (GREATEST m. ¬c dvd coeff b m)
    assume ¬[:c:] dvd b
    hence A: ∃ i. ¬c dvd coeff b i by (subst (asm) const-poly-dvd-iff) blast
    have B: ⋀i. ¬c dvd coeff b i ⟹ i ≤ degree b
      by (auto intro: le-degree)
    have coeff-m: ¬c dvd coeff b m unfolding m-def by (rule GreatestI-ex-nat[OF
A B])
    have i ≤ m if ¬c dvd coeff b i for i
      unfolding m-def by (metis (mono-tags, lifting) B Greatest-le-nat that)
    hence dvd-b: c dvd coeff b i if i > m for i using that by force

    have c dvd coeff a i for i
    proof (induction i rule: nat-descend-induct[of degree a])
      case (base i)
      thus ?case by (simp add: coeff-eq-0)
    next
      case (descend i)
      let ?A = {..i+m} - {i}
      have c dvd coeff (a * b) (i + m) by (rule dvd)
      also have coeff (a * b) (i + m) = (∑ k≤i + m. coeff a k * coeff b (i + m
- k))
```

```
    by (simp add: coeff-mult)
   also have {..i+m} = insert i ?A by auto
   also have (∑ k∈.... coeff a k * coeff b (i + m − k)) =
             coeff a i * coeff b m + (∑ k∈?A. coeff a k * coeff b (i + m − k))
    (is - = - + ?S)
    by (subst sum.insert) simp-all
   finally have eq: c dvd coeff a i * coeff b m + ?S .
   moreover have c dvd ?S
   proof (rule dvd-sum)
    fix k assume k: k ∈ {..i+m} − {i}
    show c dvd coeff a k * coeff b (i + m − k)
    proof (cases k < i)
     case False
     with k have c dvd coeff a k by (intro descend.IH) simp
     thus ?thesis by simp
    next
     case True
     hence c dvd coeff b (i + m − k) by (intro dvd-b) simp
     thus ?thesis by simp
    qed
   qed
   ultimately have c dvd coeff a i * coeff b m
    by (simp add: dvd-add-left-iff)
   with assms coeff-m show c dvd coeff a i
    by (simp add: prime-elem-dvd-mult-iff)
  qed
  hence [:c:] dvd a by (subst const-poly-dvd-iff) blast
 }
 then show [:c:] dvd a ∨ [:c:] dvd b by blast
next
 from assms show [:c:] ≠ 0 and ¬ [:c:] dvd 1
  by (simp-all add: prime-elem-def is-unit-const-poly-iff)
qed

lemma prime-elem-const-poly-iff:
 fixes c :: 'a :: semidom
 shows   prime-elem [:c:] ⟷ prime-elem c
proof
 assume A: prime-elem [:c:]
 show prime-elem c
 proof (rule prime-elemI)
  fix a b assume c dvd a * b
  hence [:c:] dvd [:a:] * [:b:] by (simp add: mult-ac)
  from A and this have [:c:] dvd [:a:] ∨ [:c:] dvd [:b:] by (rule prime-elem-dvd-multD)
  thus c dvd a ∨ c dvd b by simp
 qed (insert A, auto simp: prime-elem-def is-unit-poly-iff)
qed (auto intro: lift-prime-elem-poly)
```

4.33 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*
by (*simp add*: *content-def Gcd-fin.set-eq-fold fold-coeffs-def foldr-fold fun-eq-iff ac-simps*)

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

lemma *content-1* [*simp*]: *content 1* = *1*
by (*simp add*: *content-def*)

lemma *content-const* [*simp*]: *content* [:*c*:] = *normalize c*
by (*simp add*: *content-def cCons-def*)

lemma *const-poly-dvd-iff-dvd-content*: [:*c*:] *dvd p* ⟷ *c dvd content p*
for *c* :: *'a::semiring-gcd*
proof (*cases p* = *0*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
have [:*c*:] *dvd p* ⟷ (∀ *n*. *c dvd coeff p n*)
by (*rule const-poly-dvd-iff*)
also have ... ⟷ (∀ *a*∈*set* (*coeffs p*). *c dvd a*)
proof *safe*
fix *n* :: *nat*
assume ∀ *a*∈*set* (*coeffs p*). *c dvd a*
then show *c dvd coeff p n*
by (*cases n* ≤ *degree p*) (*auto simp*: *coeff-eq-0 coeffs-def split*: *if-splits*)
qed (*auto simp*: *coeffs-def simp del*: *upt-Suc split*: *if-splits*)
also have ... ⟷ *c dvd content p*
by (*simp add*: *content-def dvd-Gcd-fin-iff dvd-mult-unit-iff*)
finally show *?thesis* **.**
qed

lemma *content-dvd* [*simp*]: [:*content p*:] *dvd p*
by (*subst const-poly-dvd-iff-dvd-content*) *simp-all*

lemma *content-dvd-coeff* [*simp*]: *content p dvd coeff p n*
proof (*cases p* = *0*)
case *True*
then show *?thesis*
by *simp*
next
case *False*
then show *?thesis*

by (*cases* $n \leq$ *degree* p)
(*auto simp add*: *content-def not-le coeff-eq-0 coeff-in-coeffs intro*: *Gcd-fin-dvd*)
qed

lemma *content-dvd-coeffs*: $c \in$ *set* (*coeffs* p) $\Longrightarrow$ *content* p *dvd* c
by (*simp add*: *content-def Gcd-fin-dvd*)

lemma *normalize-content* [*simp*]: *normalize* (*content* p) = *content* p
by (*simp add*: *content-def*)

lemma *is-unit-content-iff* [*simp*]: *is-unit* (*content* p) $\longleftrightarrow$ *content* p = *1*
proof
assume *is-unit* (*content* p)
then have *normalize* (*content* p) = *1* **by** (*simp add*: *is-unit-normalize del*: *normalize-content*)
then show *content* p = *1* **by** *simp*
qed *auto*

lemma *content-smult* [*simp*]:
fixes c :: $'a$ :: {*normalization-semidom-multiplicative*, *semiring-gcd*}
shows *content* (*smult* c p) = *normalize* c $*$ *content* p
by (*simp add*: *content-def coeffs-smult Gcd-fin-mult normalize-mult*)

lemma *content-eq-zero-iff* [*simp*]: *content* p = *0* $\longleftrightarrow$ p = *0*
by (*auto simp*: *content-def simp*: *poly-eq-iff coeffs-def*)

definition *primitive-part* :: $'a$ :: *semiring-gcd poly* $\Rightarrow$ $'a$ *poly*
where *primitive-part* p = *map-poly* ($\lambda x.$ x *div content* p) p

lemma *primitive-part-0* [*simp*]: *primitive-part 0* = *0*
by (*simp add*: *primitive-part-def*)

lemma *content-times-primitive-part* [*simp*]: *smult* (*content* p) (*primitive-part* p) = p
for p :: $'a$ :: *semiring-gcd poly*
proof (*cases* p = *0*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
then show *?thesis*
unfolding *primitive-part-def*
by (*auto simp*: *smult-conv-map-poly map-poly-map-poly o-def content-dvd-coeffs intro*: *map-poly-idI*)
qed

lemma *primitive-part-eq-0-iff* [*simp*]: *primitive-part* p = *0* $\longleftrightarrow$ p = *0*
proof (*cases* p = *0*)
case *True*

then show *?thesis* **by** *simp*
next
case *False*
then have *primitive-part p* = *map-poly* (λ*x*. *x div content p*) *p*
by (*simp add*: *primitive-part-def*)
also from *False* **have** ... = *0* ⟷ *p* = *0*
by (*intro map-poly-eq-0-iff*) (*auto simp*: *dvd-div-eq-0-iff content-dvd-coeffs*)
finally show *?thesis*
using *False* **by** *simp*
qed

lemma *content-primitive-part* [*simp*]:
fixes *p* :: *'a* :: {*normalization-semidom-multiplicative*, *semiring-gcd*} *poly*
assumes *p* ≠ *0*
shows *content* (*primitive-part p*) = *1*
proof −
have *p* = *smult* (*content p*) (*primitive-part p*)
by *simp*
also have *content* ... = *content* (*primitive-part p*) ∗ *content p*
by (*simp del*: *content-times-primitive-part add*: *ac-simps*)
finally have *1* ∗ *content p* = *content* (*primitive-part p*) ∗ *content p*
by *simp*
then have *1* ∗ *content p div content p* = *content* (*primitive-part p*) ∗ *content p div content p*
by *simp*
with *assms* **show** *?thesis*
by *simp*
qed

lemma *content-decompose*:
obtains *p'* :: *'a* :: {*normalization-semidom-multiplicative*, *semiring-gcd*} *poly*
where *p* = *smult* (*content p*) *p' content p'* = *1*
proof (*cases p* = *0*)
case *True*
then have *p* = *smult* (*content p*) *1 content 1* = *1*
by *simp-all*
then show *?thesis* **..**
next
case *False*
then have *p* = *smult* (*content p*) (*primitive-part p*) *content* (*primitive-part p*) = *1*
by *simp-all*
then show *?thesis* **..**
qed

lemma *content-dvd-contentI* [*intro*]: *p dvd q* ⟹ *content p dvd content q*
using *const-poly-dvd-iff-dvd-content content-dvd dvd-trans* **by** *blast*

lemma *primitive-part-const-poly* [*simp*]: *primitive-part* [:*x*:] = [:*unit-factor x*:]

by (*simp add*: *primitive-part-def map-poly-pCons*)

lemma *primitive-part-prim*: *content p* = *1* ⟹ *primitive-part p* = *p*
by (*auto simp*: *primitive-part-def*)

lemma *degree-primitive-part* [*simp*]: *degree* (*primitive-part p*) = *degree p*
proof (*cases p* = *0*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
have *p* = *smult* (*content p*) (*primitive-part p*)
by *simp*
also from *False* **have** *degree* ... = *degree* (*primitive-part p*)
by (*subst degree-smult-eq*) *simp-all*
finally show *?thesis* **..**
qed

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 −
have *smult* (*content p*) (*normalize* (*primitive-part p*)) =
normalize ([:*content p*:] ∗ *primitive-part p*)
by (*subst normalize-mult*) (*simp-all add*: *normalize-const-poly*)
also have [:*content p*:] ∗ *primitive-part p* = *p* **by** *simp*
finally show *?thesis* **.**
qed

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 (*cases f* ∗ *g* = *0*)
case *False*
from *assms* **have** *f* ≠ *0 g* ≠ *0* **by** *auto*

hence *f* ∗ *g* ≠ *0* **by** *auto*
{
assume ¬*is-unit* (*content* (*f* ∗ *g*))
with *False* **have** ∃ *p*. *p dvd content* (*f* ∗ *g*) ∧ *prime p*
by (*intro prime-divisor-exists*) *simp-all*
then obtain *p* **where** *p dvd content* (*f* ∗ *g*) *prime p* **by** *blast*

```
    from ‹p dvd content (f * g)› have [:p:] dvd f * g
      by (simp add: const-poly-dvd-iff-dvd-content)
   moreover from ‹prime p› have prime-elem [:p:] by (simp add: lift-prime-elem-poly)
    ultimately have [:p:] dvd f ∨ [:p:] dvd g
      by (simp add: prime-elem-dvd-mult-iff)
    with assms have is-unit p by (simp add: const-poly-dvd-iff-dvd-content)
    with ‹prime p› have False by simp
  }
  hence is-unit (content (f * g)) by blast
   hence normalize (content (f * g)) = 1 by (simp add: is-unit-normalize del:
normalize-content)
  thus ?thesis by simp
qed (insert assms, auto)

lemma content-mult:
 fixes p q :: 'a :: {factorial-semiring, semiring-gcd, normalization-semidom-multiplicative}
poly
  shows content (p * q) = content p * content q
proof (cases p * q = 0)
  case False
  then have p ≠ 0 and q ≠ 0
    by simp-all
  then have *: content (primitive-part p * primitive-part q) = 1
    by (auto intro: content-1-mult)
 have p * q = smult (content p) (primitive-part p) * smult (content q) (primitive-part
q)
    by simp
  also have ... = smult (content p * content q) (primitive-part p * primitive-part
q)
    by (metis mult.commute mult-smult-right smult-smult)
  with * show ?thesis
  by (simp add: normalize-mult)
next
  case True
  then show ?thesis
    by auto
qed

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 -
  have primitive-part (p * q) = p * q div [:content (p * q):]
    by (simp add: primitive-part-def div-const-poly-conv-map-poly)
  also have ... = (p div [:content p:]) * (q div [:content q:])
    by (subst div-mult-div-if-dvd) (simp-all add: content-mult mult-ac)
```

```
  also have ... = primitive-part p * primitive-part q
    by (simp add: primitive-part-def div-const-poly-conv-map-poly)
  finally show ?thesis .
qed

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 -
  have smult a p = [:a:] * p by simp
  also have primitive-part ... = smult (unit-factor a) (primitive-part p)
    by (subst primitive-part-mult) simp-all
  finally show ?thesis .
qed

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 ⟹ primitive-part p dvd primitive-part q
  by (auto elim!: dvdE simp: primitive-part-mult)

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)
  by (induction A) (simp-all add: content-mult mult-ac)

lemma content-prod-eq-1-iff:
 fixes p q :: 'a :: {factorial-semiring, semiring-Gcd, normalization-semidom-multiplicative}
poly
  shows content (p * q) = 1 ⟷ content p = 1 ∧ content q = 1
proof safe
  assume A: content (p * q) = 1
  {
    fix p q :: 'a poly assume content p * content q = 1
    hence 1 = content p * content q by simp
    hence content p dvd 1 by (rule dvdI)
    hence content p = 1 by simp
  } note B = this
  from A B[of p q] B [of q p] show content p = 1 content q = 1
    by (simp-all add: content-mult mult-ac)
qed (auto simp: content-mult)
```

4.34 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* $(p :: 'a :: \textit{alg-closed-field poly}) > 0$
  shows $\exists x.\ \textit{poly}\ p\ x = 0$
proof −
  have $\exists x.\ (\sum k{\leq}\textit{degree}\ p.\ \textit{coeff}\ p\ k * x\ \hat{}\ k) = 0$
    using *assms* **by** (*intro alg-closed*) *auto*
  thus *?thesis*
    by (*simp add*: *poly-altdef*)
qed

lemma *alg-closedI* [*Pure.intro*]:
  assumes $\bigwedge p :: 'a\ \textit{poly}.\ \textit{degree}\ p > 0 \Longrightarrow \textit{lead-coeff}\ p = 1 \Longrightarrow \exists x.\ \textit{poly}\ p\ x = 0$
  shows *OFCLASS*('a :: *field*, *alg-closed-field-class*)
proof
  fix $n :: \textit{nat}$ **and** $f :: \textit{nat} \Rightarrow 'a$
  assume *n*: $n > 0\ f\ n \neq 0$
  define *p* **where** $p = \textit{Abs-poly}\ (\lambda k.\ \textit{if}\ k \leq n\ \textit{then}\ f\ k\ \textit{else}\ 0)$
  have *coeff-p*: $\textit{coeff}\ p\ k = (\textit{if}\ k \leq n\ \textit{then}\ f\ k\ \textit{else}\ 0)$ **for** *k*
  proof −
    have $\textit{eventually}\ (\lambda k.\ k > n)\ \textit{cofinite}$
      by (*auto simp*: *MOST-nat*)
    hence $\textit{eventually}\ (\lambda k.\ (\textit{if}\ k \leq n\ \textit{then}\ f\ k\ \textit{else}\ 0) = 0)\ \textit{cofinite}$
      by *eventually-elim auto*
    thus *?thesis*
      unfolding *p-def* **by** (*subst Abs-poly-inverse*) *auto*
  qed

  from *n* **have** $\textit{degree}\ p \geq n$
    by (*intro le-degree*) (*auto simp*: *coeff-p*)
  moreover have $\textit{degree}\ p \leq n$
    by (*intro degree-le*) (*auto simp*: *coeff-p*)
  ultimately have *deg-p*: $\textit{degree}\ p = n$
    by *linarith*
  from *deg-p* **and** *n* **have** [*simp*]: $p \neq 0$
    by *auto*

  define p' **where** $p' = \textit{smult}\ (\textit{inverse}\ (\textit{lead-coeff}\ p))\ p$
  have *deg-p'*: $\textit{degree}\ p' = \textit{degree}\ p$
    by (*auto simp*: p'-*def*)
  have *lead-coeff-p'* [*simp*]: $\textit{lead-coeff}\ p' = 1$
    by (*auto simp*: p'-*def*)

  from *deg-p* **and** *deg-p'* **and** *n* **have** $\textit{degree}\ p' > 0$
    by *simp*
  from *assms*[*OF this*] **obtain** *x* **where** $\textit{poly}\ p'\ x = 0$

```
    by auto
  hence poly p x = 0
    by (simp add: p'-def)
  also have poly p x = (∑ k≤n. f k * x ^ k)
    unfolding poly-altdef by (intro sum.cong) (auto simp: deg-p coeff-p)
  finally show ∃ x. (∑ k≤n. f k * x ^ k) = 0 ..
qed

lemma (in alg-closed-field) nth-root-exists:
  assumes n > 0
  shows   ∃ y. y ^ n = (x :: 'a)
proof −
  define f where f = (λi. if i = 0 then −x else if i = n then 1 else 0)
  have ∃ x. (∑ k≤n. f k * x ^ k) = 0
    by (rule alg-closed) (use assms in ‹auto simp: f-def›)
  also have (λx. ∑ k≤n. f k * x ^ k) = (λx. ∑ k∈{0,n}. f k * x ^ k)
    by (intro ext sum.mono-neutral-right) (auto simp: f-def)
  finally show ∃ y. y ^ n = x
    using assms by (simp add: f-def)
qed
```

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-closed-field poly
  assumes p ≠ 0
  shows ∃ A. size A = degree p ∧ p = smult (lead-coeff p) (∏ x∈#A. [:−x, 1:])
  using assms
proof (induction degree p arbitrary: p rule: less-induct)
  case (less p)
  show ?case
  proof (cases degree p = 0)
    case True
    thus ?thesis
      by (intro exI[of - {#}]) (auto elim!: degree-eq-zeroE)
  next
    case False
    then obtain x where x: poly p x = 0
      using alg-closed-imp-poly-has-root by blast
    hence [:−x, 1:] dvd p
      using poly-eq-0-iff-dvd by blast
    then obtain q where p-eq: p = [:−x, 1:] * q
      by (elim dvdE)
    have q ≠ 0
      using less.prems p-eq by auto
    moreover from this have deg: degree p = Suc (degree q)
      unfolding p-eq by (subst degree-mult-eq) auto
    ultimately obtain A where A: size A = degree q q = smult (lead-coeff q)
(∏ x∈#A. [:−x, 1:])
```

```
    using less.hyps[of q] by auto
  have smult (lead-coeff p) (∏ y∈#add-mset x A. [:− y, 1:]) =
      [:− x, 1:] ∗ smult (lead-coeff q) (∏ y∈#A. [:− y, 1:])
    unfolding p-eq lead-coeff-mult by simp
  also note A(2) [symmetric]
  also note p-eq [symmetric]
  finally show ?thesis using A(1)
    by (intro exI[of - add-mset x A]) (auto simp: deg)
 qed
qed
```

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-closed-field poly) > 1
 shows   ¬irreducible p
proof −
 have degree p > 0
   using assms by auto
 then obtain z where z: poly p z = 0
   using alg-closed-imp-poly-has-root[of p] by blast
 then have dvd: [:−z, 1:] dvd p
   by (subst dvd-iff-poly-eq-0) auto
 then obtain q where q: p = [:−z, 1:] ∗ q
   by (erule dvdE)
 have [simp]: q ≠ 0
   using assms q by auto

 show ?thesis
 proof (rule reducible-polyI)
   show p = [:−z, 1:] ∗ q
     by fact
 next
   have degree p = degree ([:−z, 1:] ∗ q)
     by (simp only: q)
   also have ... = degree q + 1
     by (subst degree-mult-eq) auto
   finally show degree q > 0
     using assms by linarith
 qed auto
qed
```

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 ⋀p :: 'a poly. degree p > 1 ⟹ lead-coeff p = 1 ⟹ coeff p 0 ≠ 0 ⟹
           ¬irreducible p
 shows   OFCLASS('a :: field, alg-closed-field-class)
proof
```

fix $p :: 'a\ poly$ **assume** p: $degree\ p > 0\ lead\text{-}coeff\ p = 1$
show $\exists x.\ poly\ p\ x = 0$
proof $(cases\ coeff\ p\ 0 = 0)$
  case *True*
  hence $poly\ p\ 0 = 0$
    by $(simp\ add: poly\text{-}0\text{-}coeff\text{-}0)$
  thus *?thesis* **by** *blast*
next
  case *False*
  from p **and** *this* **show** *?thesis*
  proof $(induction\ degree\ p\ arbitrary: p\ rule: less\text{-}induct)$
    case $(less\ p)$
    show *?case*
    proof $(cases\ degree\ p = 1)$
      case *True*
      then obtain $a\ b$ **where** p: $p = [:a, b:]$
        by $(cases\ p)$ $(auto\ split: if\text{-}splits\ elim!: degree\text{-}eq\text{-}zeroE)$
      from *True* **have** $[simp]$: $b \neq 0$
        by $(auto\ simp: p)$
      have $poly\ p\ (-a/b) = 0$
        by $(auto\ simp: p)$
      thus *?thesis* **by** *blast*
    next
      case *False*
      hence $degree\ p > 1$
        using *less.prems* **by** *auto*
      from $assms[OF\ \langle degree\ p > 1\rangle\ \langle lead\text{-}coeff\ p = 1\rangle\ \langle coeff\ p\ 0 \neq 0\rangle]$
      have $\neg irreducible\ p$ **by** *auto*
      then obtain $r\ s$ **where** rs: $degree\ r > 0\ degree\ s > 0\ p = r * s$
        using *less.prems* **unfolding** *irreducible-def*
        by $(metis\ is\text{-}unit\text{-}iff\text{-}degree\ mult\text{-}not\text{-}zero\ zero\text{-}less\text{-}iff\text{-}neq\text{-}zero)$
      hence $coeff\ r\ 0 \neq 0$
        using $\langle coeff\ p\ 0 \neq 0\rangle$ **by** $(auto\ simp: coeff\text{-}mult\text{-}0)$

      define r' **where** $r' = smult\ (inverse\ (lead\text{-}coeff\ r))\ r$
      have $[simp]$: $degree\ r' = degree\ r$
        by $(simp\ add: r'\text{-}def)$
      have lc: $lead\text{-}coeff\ r' = 1$
        using rs **by** $(auto\ simp: r'\text{-}def)$
      have nz: $coeff\ r'\ 0 \neq 0$
        using $\langle coeff\ r\ 0 \neq 0\rangle$ **by** $(auto\ simp: r'\text{-}def)$

      have $degree\ r < degree\ r + degree\ s$
        using rs **by** *linarith*
      also have $\ldots = degree\ (r * s)$
        using $rs(3)$ *less.prems* **by** $(subst\ degree\text{-}mult\text{-}eq)$ *auto*
      also have $r * s = p$
        using $rs(3)$ **by** *simp*
      finally have $\exists x.\ poly\ r'\ x = 0$

```
      by (intro less) (use lc rs nz in auto)
    thus ?thesis
      using rs(3) by (auto simp: r'-def)
  qed
 qed
qed
```

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 ⋀p :: 'a poly. degree p > 1 ⟹ lead-coeff p = 1 ⟹ coeff p (degree p
- 1) = 0 ⟹
          coeff p 0 ≠ 0 ⟹ ¬irreducible p
 shows   OFCLASS('a :: field-char-0, alg-closed-field-class)
proof (rule alg-closedI-reducible, goal-cases)
 case (1 p)
 define n where [simp]: n = degree p
 define a where a = coeff p (n - 1)
 define r where r = [: -a / of-nat n, 1 :]
 define s where s = [: a / of-nat n, 1 :]
 define q where q = pcompose p r

 have n > 0
   using 1 by simp
 have r-altdef: r = monom 1 1 + [:-a / of-nat n:]
   by (simp add: r-def monom-altdef)
 have deg-q: degree q = n
   by (simp add: q-def r-def degree-pcompose)
 have lc-q: lead-coeff q = 1
   unfolding q-def using 1 by (subst lead-coeff-comp) (simp-all add: r-def)
 have q ≠ 0
   using 1 deg-q by auto

 have coeff q (n - 1) =
        (∑i≤n. ∑k≤i. coeff p i * (of-nat (i choose k) *
          ((-a / of-nat n) ^ (i - k)) * (if k = n - 1 then 1 else 0))))
   unfolding q-def pcompose-altdef poly-altdef r-altdef
  by (simp-all add: degree-map-poly coeff-map-poly coeff-sum binomial-ring sum-distrib-left
poly-const-pow
            sum-distrib-right mult-ac monom-power coeff-monom-mult of-nat-poly
cong: if-cong)
 also have ... = (∑i≤n. ∑k∈(if i ≥ n - 1 then {n-1} else {}).
                  coeff p i * (of-nat (i choose k) * (-a / of-nat n) ^ (i - k)))
    by (rule sum.cong [OF refl], rule sum.mono-neutral-cong-right) (auto split:
if-splits)
 also have ... = (∑i∈{n-1,n}. ∑k∈(if i ≥ n - 1 then {n-1} else {}).
                  coeff p i * (of-nat (i choose k) * (-a / of-nat n) ^ (i - k)))
   by (rule sum.mono-neutral-right) auto
```

```
  also have ... = a - of-nat (n choose (n - 1)) * a / of-nat n
    using 1 by (simp add: a-def)
  also have n choose (n - 1) = n
    using ‹n > 0› by (subst binomial-symmetric) auto
  also have a - of-nat n * a / of-nat n = 0
    using ‹n > 0› by simp
  finally have coeff q (n - 1) = 0 .

  show ?case
  proof (cases coeff q 0 = 0)
    case True
    hence poly p (- (a / of-nat (degree p))) = 0
      by (auto simp: q-def r-def)
    thus ?thesis
      by (rule root-imp-reducible-poly) (use 1 in auto)
  next
    case False
    hence ¬irreducible q
      using assms[of q] and lc-q and 1 and ‹coeff q (n - 1) = 0›
      by (auto simp: deg-q)
    then obtain u v where uv: degree u > 0 degree v > 0 q = u * v
      using ‹q ≠ 0› 1 deg-q unfolding irreducible-def
     by (metis degree-mult-eq-0 is-unit-iff-degree n-def neq0-conv not-one-less-zero)

    have p = pcompose q s
      by (simp add: q-def r-def s-def pcompose-pCons flip: pcompose-assoc)
    also have q = u * v
      by fact
    finally have p = pcompose u s * pcompose v s
      by (simp add: pcompose-mult)
    moreover have degree (pcompose u s) > 0 degree (pcompose v s) > 0
      using uv by (simp-all add: s-def degree-pcompose)
    ultimately show ¬irreducible p
      using 1 by (intro reducible-polyI)
  qed
qed
```

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 -
  define n where n = order x p
  have n > 0
    using assms by (metis degree-0 gr0I n-def not-one-less-zero order-root)
  have [:-x, 1:] ^ n dvd p
```

```
    unfolding n-def by (simp add: order-1)
  then obtain q where p-eq: p = [:−x, 1:] ^ n * q
    by (elim dvdE)
  from assms have [simp]: q ≠ 0
    by (auto simp: p-eq)
  have order x p = n + Polynomial.order x q
    unfolding p-eq by (subst order-mult) (auto simp: order-power-n-n)
  hence Polynomial.order x q = 0
    by (simp add: n-def)
  hence poly q x ≠ 0
    by (simp add: order-root)

  show ?thesis
  proof (rule that)
    show coprime ([:−x, 1:] ^ n) q
    proof (rule coprimeI)
      fix d
      assume d: d dvd [:−x, 1:] ^ n d dvd q
      have degree d = 0
      proof (rule ccontr)
        assume ¬(degree d = 0)
        then obtain z where z: poly d z = 0
          using alg-closed-imp-poly-has-root by blast
        moreover from this and d(1) have poly ([:−x, 1:] ^ n) z = 0
          using dvd-trans poly-eq-0-iff-dvd by blast
        ultimately have poly d x = 0
          by auto
        with d(2) have poly q x = 0
          using dvd-trans poly-eq-0-iff-dvd by blast
        with ‹poly q x ≠ 0› show False by contradiction
      qed
      thus is-unit d using d
        by (metis ‹q ≠ 0› dvd-0-left is-unit-iff-degree)
    qed
  next
    have poly q y = 0
      using ‹poly p y = 0› ‹x ≠ y› by (auto simp: p-eq)
    with ‹q ≠ 0› show degree q > 0
      using order-degree order-gt-0-iff order-less-le-trans by blast
  qed (use ‹n > 0› in ‹simp-all add: p-eq degree-power-eq›)
qed
```

4.35 Polynomials and limits

```
lemma filterlim-poly-at-infinity:
  fixes p::'a::real-normed-field poly
  assumes degree p>0
  shows filterlim (poly p) at-infinity at-infinity
using assms
```

```
proof (induct p)
  case 0
  then show ?case by auto
next
  case (pCons a p)
  have ?case when degree p=0
  proof −
    obtain c where c-def:p=[:c:] using ‹degree p = 0› degree-eq-zeroE by blast
    then have c≠0 using ‹0 < degree (pCons a p)› by auto
    then show ?thesis unfolding c-def
      apply (auto intro!:tendsto-add-filterlim-at-infinity)
      apply (subst mult.commute)
      by (auto intro!:tendsto-mult-filterlim-at-infinity filterlim-ident)
  qed
  moreover have ?case when degree p≠0
  proof −
    have filterlim (poly p) at-infinity at-infinity
      using that by (auto intro:pCons)
    then show ?thesis
     by (auto intro!:tendsto-add-filterlim-at-infinity filterlim-at-infinity-times filter-
lim-ident)
  qed
  ultimately show ?case by auto
qed

lemma poly-divide-tendsto-aux:
  fixes p::'a::real-normed-field poly
  shows ((λx. poly p x/x^(degree p)) ⟶ lead-coeff p) at-infinity
proof (induct p)
  case 0
  then show ?case by (auto intro:tendsto-eq-intros)
next
  case (pCons a p)
  have ?case when p=0
    using that by auto
  moreover have ?case when p≠0
  proof −
    define g where g=(λx. a/(x*x^degree p))
    define f where f=(λx. poly p x/x^degree p)
    have ∀F x in at-infinity. poly (pCons a p) x / x ^ degree (pCons a p) = g x +
f x
    proof (rule eventually-at-infinityI[of 1])
      fix x::'a assume norm x≥1
      then have x≠0 by auto
      then show poly (pCons a p) x / x ^ degree (pCons a p) = g x + f x
        using that unfolding g-def f-def by (auto simp add:field-simps)
    qed
    moreover have ((λx. g x+f x) ⟶ lead-coeff (pCons a p)) at-infinity
    proof −
```

```
    have (g ⟶ 0) at-infinity
      unfolding g-def using filterlim-poly-at-infinity[of monom 1 (Suc (degree
p))]
    apply (auto intro!:tendsto-intros tendsto-divide-0 simp add: degree-monom-eq)
     apply (subst filterlim-cong[where g=poly (monom 1 (Suc (degree p)))])
     by (auto simp add:poly-monom)
    moreover have (f ⟶ lead-coeff (pCons a p)) at-infinity
     using pCons ‹p≠0› unfolding f-def by auto
    ultimately show ?thesis by (auto intro:tendsto-eq-intros)
   qed
   ultimately show ?thesis by (auto dest:tendsto-cong)
  qed
  ultimately show ?case by auto
qed

lemma filterlim-power-at-infinity:
  assumes n≠0
  shows filterlim (λx::'a::real-normed-field. x^n) at-infinity at-infinity
  using filterlim-poly-at-infinity[of monom 1 n] assms
  by (simp add: filterlim-ident filterlim-power-at-infinity)

lemma poly-divide-tendsto-0-at-infinity:
  fixes p::'a::real-normed-field poly
  assumes degree p > degree q
  shows ((λx. poly q x / poly p x) ⟶ 0 ) at-infinity
proof −
  define pp where pp ≡ (λx. x^(degree p) / poly p x)
  define qq where qq ≡ (λx. poly q x/x^(degree q))
  define dd where dd ≡ (λx::'a. 1/x^(degree p − degree q))
  have ∀ F x in at-infinity. poly q x / poly p x = qq x * pp x * dd x
  proof (rule eventually-at-infinityI[of 1])
   fix x::'a assume norm x≥1
   then have x≠0 by auto
   then show poly q x / poly p x = qq x * pp x * dd x
    unfolding qq-def pp-def dd-def using assms
    by (auto simp add:field-simps divide-simps power-diff)
  qed
  moreover have ((λx. qq x * pp x * dd x) ⟶ 0) at-infinity
  proof −
   have (qq ⟶ lead-coeff q) at-infinity
    unfolding qq-def using poly-divide-tendsto-aux[of q] .
   moreover have (pp ⟶ 1/lead-coeff p) at-infinity
   proof −
    have p≠0 using assms by auto
    then show ?thesis
     unfolding pp-def using poly-divide-tendsto-aux[of p]
     apply (drule-tac tendsto-inverse)
     by (auto simp add:inverse-eq-divide)
   qed
```

```
    moreover have (dd ⟶ 0) at-infinity
      unfolding dd-def
      apply (rule tendsto-divide-0)
      by (auto intro!: filterlim-power-at-infinity simp add:assms)
    ultimately show ?thesis by (auto intro:tendsto-eq-intros)
  qed
  ultimately show ?thesis by (auto dest:tendsto-cong)
qed

lemma poly-eventually-not-zero:
  fixes p::real poly
  assumes p≠0
  shows eventually (λx. poly p x ≠ 0) at-infinity
proof (rule eventually-at-infinityI[of Max (norm ` {x. poly p x = 0}) + 1])
  fix x::real assume §: Max (norm ` {x. poly p x = 0}) + 1 ≤ norm x
  have False when poly p x = 0
  proof −
    define S where S=norm `{x. poly p x = 0}
    have norm x∈S
      using that unfolding S-def by auto
    moreover have finite S
      using ‹p≠0› poly-roots-finite unfolding S-def by blast
    ultimately have norm x≤Max S
      by simp
    moreover have Max S + 1 ≤ norm x
      using § unfolding S-def by simp
    ultimately show False by argo
  qed
  then show poly p x ≠ 0 by auto
qed

no-notation cCons (infixr ‹##› 65)
hide-const (open) Polynomial.content

end
```

5 A formalization of formal power series

```
theory Formal-Power-Series
imports
  Complex-Main
  Euclidean-Algorithm
  Primes
  HOL−Library.FuncSet
  HOL−Library.Multiset
begin
```

5.1 The type of formal power series

typedef *'a fps* = {*f* :: *nat* ⇒ *'a*. *True*}
  morphisms *fps-nth Abs-fps*
  by *simp*

notation *fps-nth* (**infixl** ‹$› *75*)

lemma *expand-fps-eq*: *p* = *q* ⟷ (∀ *n*. *p* $ *n* = *q* $ *n*)
  by (*simp add*: *fps-nth-inject* [*symmetric*] *fun-eq-iff*)

lemmas *fps-eq-iff* = *expand-fps-eq*

lemma *fps-ext*: (⋀*n*. *p* $ *n* = *q* $ *n*) ⟹ *p* = *q*
  by (*simp add*: *expand-fps-eq*)

lemma *fps-nth-Abs-fps* [*simp*]: *Abs-fps f* $ *n* = *f n*
  by (*simp add*: *Abs-fps-inverse*)

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 ..
end

lemma *fps-zero-nth* [*simp*]: *0* $ *n* = *0*
  unfolding *fps-zero-def* **by** *simp*

lemma *fps-nonzero-nth*: *f* ≠ *0* ⟷ (∃ *n*. *f* $ *n* ≠ *0*)
  by (*simp add*: *expand-fps-eq*)

lemma *fps-nonzero-nth-minimal*: *f* ≠ *0* ⟷ (∃ *n*. *f* $ *n* ≠ *0* ∧ (∀ *m* < *n*. *f* $ *m* = *0*))
  (**is** *?lhs* ⟷ *?rhs*)
proof
  let *?n* = *LEAST n*. *f* $ *n* ≠ *0*
  show *?rhs* **if** *?lhs*
  proof −
    from *that* **have** ∃ *n*. *f* $ *n* ≠ *0*
      by (*simp add*: *fps-nonzero-nth*)
    then have *f* $ *?n* ≠ *0*
      by (*rule LeastI-ex*)
    moreover have ∀ *m*<*?n*. *f* $ *m* = *0*
      by (*auto dest*: *not-less-Least*)
    ultimately show *?thesis* **by** *metis*
  qed
qed (*auto simp*: *expand-fps-eq*)

lemma *fps-nonzeroI*: $f\$n \neq 0 \Longrightarrow f \neq 0$
by *auto*

instantiation *fps* :: ({*one*, *zero*}) *one*
begin
definition *fps-one-def*: $1 = Abs\text{-}fps\ (\lambda n.\ if\ n = 0\ then\ 1\ else\ 0)$
instance ..
end

lemma *fps-one-nth* [*simp*]: $1\ \$\ n = (if\ n = 0\ then\ 1\ else\ 0)$
unfolding *fps-one-def* **by** *simp*

instantiation *fps* :: (*plus*) *plus*
begin
definition *fps-plus-def*: $(+) = (\lambda f\ g.\ Abs\text{-}fps\ (\lambda n.\ f\ \$\ n + g\ \$\ n))$
instance ..
end

lemma *fps-add-nth* [*simp*]: $(f + g)\ \$\ n = f\ \$\ n + g\ \$\ n$
unfolding *fps-plus-def* **by** *simp*

instantiation *fps* :: (*minus*) *minus*
begin
definition *fps-minus-def*: $(-) = (\lambda f\ g.\ Abs\text{-}fps\ (\lambda n.\ f\ \$\ n - g\ \$\ n))$
instance ..
end

lemma *fps-sub-nth* [*simp*]: $(f - g)\ \$\ n = f\ \$\ n - g\ \$\ n$
unfolding *fps-minus-def* **by** *simp*

instantiation *fps* :: (*uminus*) *uminus*
begin
definition *fps-uminus-def*: $uminus = (\lambda f.\ Abs\text{-}fps\ (\lambda n.\ -\ (f\ \$\ n)))$
instance ..
end

lemma *fps-neg-nth* [*simp*]: $(-\ f)\ \$\ n = -\ (f\ \$\ n)$
unfolding *fps-uminus-def* **by** *simp*

lemma *fps-neg-0* [*simp*]: $-(0::'a::group\text{-}add\ fps) = 0$
by (*rule iffD2*, *rule fps-eq-iff*, *auto*)

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 ..
end

lemma *fps-mult-nth*: $(f * g)\ \$\ n = (\sum i{=}0..n.\ f\$i * g\$(n - i))$
unfolding *fps-times-def* **by** *simp*

lemma *fps-mult-nth-0* [*simp*]: $(f * g)\ \$\ 0 = f\ \$\ 0 * g\ \$\ 0$
unfolding *fps-times-def* **by** *simp*

lemma *fps-mult-nth-1*: $(f * g)\ \$\ 1 = f\$0 * g\$1 + f\$1 * g\$0$
by (*simp add*: *fps-mult-nth*)

lemma *fps-mult-nth-1'* [*simp*]: $(f * g)\ \$\ Suc\ 0 = f\$0 * g\$Suc\ 0 + f\$Suc\ 0 * g\$0$
by (*simp add*: *fps-mult-nth*)

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
fix *a* :: *'a fps*
show *0 * a = 0* **by** (*simp add*: *fps-ext fps-mult-nth*)
show *a * 0 = 0* **by** (*simp add*: *fps-ext fps-mult-nth*)
qed

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*)
by *simp*

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*)
by *simp*

lemma *fps-one-mult*:
fixes *f* :: *'a*::{*comm-monoid-add*, *mult-zero*, *monoid-mult*} *fps*
shows *1 * f = f*
and *f * 1 = f*
by (*simp-all add*: *fps-ext fps-mult-nth mult-delta-left mult-delta-right*)

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 ≠ 0* **and** ⋀*i*. *i < d* ⟹ *f $ i = 0*

shows *subdegree f = d*
by (*smt* (*verit*) *LeastI-ex assms fps-zero-nth linorder-cases not-less-Least subdegree-def*)

lemma *nth-subdegree-nonzero* [*simp*,*intro*]: *f ≠ 0 ⟹ f $ subdegree f ≠ 0*
using *fps-nonzero-nth-minimal subdegreeI* **by** *blast*

lemma *nth-less-subdegree-zero* [*dest*]: *n < subdegree f ⟹ f $ n = 0*
by (*metis fps-nonzero-nth-minimal fps-zero-nth subdegreeI*)

lemma *subdegree-geI*:
assumes *f ≠ 0* ⋀*i*. *i < n ⟹ f$i = 0*
shows *subdegree f ≥ n*
by (*meson assms leI nth-subdegree-nonzero*)

lemma *subdegree-greaterI*:
assumes *f ≠ 0* ⋀*i*. *i ≤ n ⟹ f$i = 0*
shows *subdegree f > n*
by (*meson assms leI nth-subdegree-nonzero*)

lemma *subdegree-leI*:
f $ n ≠ 0 ⟹ subdegree f ≤ n
using *linorder-not-less* **by** *blast*

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

lemma *subdegree-1* [*simp*]: *subdegree 1 = 0*
by (*metis fps-one-nth nth-subdegree-nonzero subdegree-0*)

lemma *subdegree-eq-0-iff*: *subdegree f = 0 ⟷ f = 0 ∨ f $ 0 ≠ 0*
using *nth-subdegree-nonzero subdegree-leI* **by** *fastforce*

lemma *subdegree-eq-0* [*simp*]: *f $ 0 ≠ 0 ⟹ subdegree f = 0*
by (*simp add*: *subdegree-eq-0-iff*)

lemma *nth-subdegree-zero-iff* [*simp*]: *f $ subdegree f = 0 ⟷ f = 0*
by (*cases f = 0*) *auto*

lemma *fps-nonzero-subdegree-nonzeroI*: *subdegree f > 0 ⟹ f ≠ 0*
by *auto*

lemma *subdegree-uminus* [*simp*]:
subdegree (−(*f*::(*'a*::*group-add*) *fps*)) = *subdegree f*
proof (*cases f=0*)
case *False* **thus** *?thesis* **by** (*force intro*: *subdegreeI*)
qed *simp*

lemma *subdegree-minus-commute* [*simp*]:

fixes f :: $'a$::*group-add fps*
shows *subdegree* $(f-g)$ = *subdegree* $(g - f)$
proof (*cases* $g-f=0$)
case *True* **then show** *?thesis*
by (*metis fps-sub-nth nth-subdegree-nonzero right-minus-eq*)
next
case *False* **show** *?thesis*
using *nth-subdegree-nonzero*[*OF False*] **by** (*fastforce intro*: *subdegreeI*)
qed

lemma *subdegree-add-ge'*:
fixes $f\ g$ :: $'a$::*monoid-add fps*
assumes $f + g \neq 0$
shows *subdegree* $(f + g) \geq$ *min* (*subdegree* f) (*subdegree* g)
using *assms*
by (*force intro*: *subdegree-geI*)

lemma *subdegree-add-ge*:
assumes $f \neq -(g :: ('a :: group\text{-}add)\ fps)$
shows *subdegree* $(f + g) \geq$ *min* (*subdegree* f) (*subdegree* g)
proof (*rule subdegree-add-ge'*)
have $f + g = 0 \Longrightarrow False$
proof−
assume *fg*: $f + g = 0$
have $\bigwedge n.\ f\ \$\ n = -\ g\ \$\ n$
by (*metis add-eq-0-iff equation-minus-iff fg fps-add-nth fps-neg-nth fps-zero-nth*)
with *assms* **show** *False* **by** (*auto intro*: *fps-ext*)
qed
thus $f + g \neq 0$ **by** *fast*
qed

lemma *subdegree-add-eq1*:
assumes $f \neq 0$
and *subdegree* $f <$ *subdegree* $(g :: 'a$::*monoid-add fps*)
shows *subdegree* $(f + g)$ = *subdegree* f
using *assms* **by**(*auto intro*: *subdegreeI simp*: *nth-less-subdegree-zero*)

lemma *subdegree-add-eq2*:
assumes $g \neq 0$
and *subdegree* $g <$ *subdegree* $(f :: 'a :: monoid\text{-}add\ fps)$
shows *subdegree* $(f + g)$ = *subdegree* g
using *assms* **by** (*auto intro*: *subdegreeI simp*: *nth-less-subdegree-zero*)

lemma *subdegree-diff-eq1*:
assumes $f \neq 0$
and *subdegree* $f <$ *subdegree* $(g :: 'a :: group\text{-}add\ fps)$
shows *subdegree* $(f - g)$ = *subdegree* f
using *assms* **by** (*auto intro*: *subdegreeI simp*: *nth-less-subdegree-zero*)

lemma *subdegree-diff-eq1-cancel*:
assumes $f \neq 0$
and *subdegree* $f <$ *subdegree* $(g :: 'a :: cancel\text{-}comm\text{-}monoid\text{-}add\ fps)$
shows *subdegree* $(f - g) =$ *subdegree* f
using *assms* **by** (*auto intro*: *subdegreeI simp*: *nth-less-subdegree-zero*)

lemma *subdegree-diff-eq2*:
assumes $g \neq 0$
and *subdegree* $g <$ *subdegree* $(f :: 'a :: group\text{-}add\ fps)$
shows *subdegree* $(f - g) =$ *subdegree* g
using *assms* **by** (*auto intro*: *subdegreeI simp*: *nth-less-subdegree-zero*)

lemma *subdegree-diff-ge* [*simp*]:
assumes $f \neq (g :: 'a :: group\text{-}add\ fps)$
shows *subdegree* $(f - g) \geq min$ (*subdegree* f) (*subdegree* g)
proof−
have $f \neq -(-g)$
using *assms expand-fps-eq* **by** *fastforce*
moreover have $f + -g = f - g$ **by** (*simp add*: *fps-ext*)
ultimately show *?thesis*
using *subdegree-add-ge*[*of f* $-g$] **by** *simp*
qed

lemma *subdegree-diff-ge'*:
fixes $f\ g :: 'a :: comm\text{-}monoid\text{-}diff\ fps$
assumes $f - g \neq 0$
shows *subdegree* $(f - g) \geq$ *subdegree* f
using *assms* **by** (*auto intro*: *subdegree-geI simp*: *nth-less-subdegree-zero*)

lemma *nth-subdegree-mult-left* [*simp*]:
fixes $f\ g :: ('a :: \{mult\text{-}zero, comm\text{-}monoid\text{-}add\})\ fps$
shows $(f * g)\ \$$ (*subdegree* f) $= f\ \$$ *subdegree* $f * g\ \$\ 0$
by (*cases subdegree f*) (*simp-all add*: *fps-mult-nth nth-less-subdegree-zero*)

lemma *nth-subdegree-mult-right* [*simp*]:
fixes $f\ g :: ('a :: \{mult\text{-}zero, comm\text{-}monoid\text{-}add\})\ fps$
shows $(f * g)\ \$$ (*subdegree* g) $= f\ \$\ 0 * g\ \$$ *subdegree* g
by (*cases subdegree g*) (*simp-all add*: *fps-mult-nth nth-less-subdegree-zero sum.atLeast-Suc-atMost*)

lemma *nth-subdegree-mult* [*simp*]:
fixes $f\ g :: ('a :: \{mult\text{-}zero, comm\text{-}monoid\text{-}add\})\ fps$
shows $(f * g)\ \$$ (*subdegree* $f +$ *subdegree* g) $= f\ \$$ *subdegree* $f * g\ \$$ *subdegree* g
proof−
let *?n* $=$ *subdegree* $f +$ *subdegree* g
have $(f * g)\ \$\ ?n = (\sum i{=}0..?n.\ f\$i * g\$(?n-i))$
by (*simp add*: *fps-mult-nth*)
also have ... $= (\sum i{=}0..?n.$ *if* $i =$ *subdegree* f *then* $f\$i * g\$(?n-i)$ *else* $0)$
proof (*intro sum.cong*)

fix x **assume** x: $x \in \{0..?n\}$
hence $x = subdegree\ f \vee x < subdegree\ f \vee ?n - x < subdegree\ g$ **by** *auto*
thus $f\ \$\ x * g\ \$\ (?n - x) = (if\ x = subdegree\ f\ then\ f\ \$\ x * g\ \$\ (?n - x)\ else\ 0)$
by (*elim disjE conjE*) *auto*
qed *auto*
also have ... = $f\ \$\ subdegree\ f * g\ \$\ subdegree\ g$ **by** *simp*
finally show *?thesis* **.**
qed

lemma *fps-mult-nth-eq0*:
fixes $f\ g :: 'a::\{comm\text{-}monoid\text{-}add,mult\text{-}zero\}\ fps$
assumes $n < subdegree\ f + subdegree\ g$
shows $(f*g)\ \$\ n = 0$
proof−
have $\bigwedge i.\ i \in \{0..n\} \Longrightarrow f\$i * g\$(n - i) = 0$
proof−
fix i **assume** i: $i \in \{0..n\}$
show $f\$i * g\$(n - i) = 0$
proof (*cases* $i < subdegree\ f \vee n - i < subdegree\ g$)
case *False* **with** *assms i* **show** *?thesis* **by** *auto*
qed (*auto simp*: *nth-less-subdegree-zero*)
qed
thus $(f * g)\ \$\ n = 0$ **by** (*simp add*: *fps-mult-nth*)
qed

lemma *fps-mult-subdegree-ge*:
fixes $f\ g :: 'a::\{comm\text{-}monoid\text{-}add,mult\text{-}zero\}\ fps$
assumes $f*g \neq 0$
shows $subdegree\ (f*g) \geq subdegree\ f + subdegree\ g$
using *assms fps-mult-nth-eq0*
by (*intro subdegree-geI*) *simp*

lemma *subdegree-mult'*:
fixes $f\ g :: 'a::\{comm\text{-}monoid\text{-}add,mult\text{-}zero\}\ fps$
assumes $f\ \$\ subdegree\ f * g\ \$\ subdegree\ g \neq 0$
shows $subdegree\ (f*g) = subdegree\ f + subdegree\ g$
proof−
from *assms* **have** $(f * g)\ \$\ (subdegree\ f + subdegree\ g) \neq 0$ **by** *simp*
hence $f*g \neq 0$ **by** *fastforce*
hence $subdegree\ (f*g) \geq subdegree\ f + subdegree\ g$ **using** *fps-mult-subdegree-ge* **by** *fast*
moreover from *assms* **have** $subdegree\ (f*g) \leq subdegree\ f + subdegree\ g$
by (*intro subdegree-leI*) *simp*
ultimately show *?thesis* **by** *simp*
qed

lemma *subdegree-mult* [*simp*]:
fixes $f\ g :: 'a :: \{semiring\text{-}no\text{-}zero\text{-}divisors\}\ fps$

assumes $f \neq 0$ $g \neq 0$
shows *subdegree* $(f * g) =$ *subdegree* $f +$ *subdegree* g
using *assms*
by (*intro subdegree-mult'*) *simp*

lemma *fps-mult-nth-conv-upto-subdegree-left*:
fixes $f\ g :: ('a :: \{$*mult-zero,comm-monoid-add*$\})$ *fps*
shows $(f * g)\ \$\ n = (\sum i$=*subdegree* $f..n.\ f\ \$\ i * g\ \$\ (n - i))$
proof (*cases subdegree* $f \leq n$)
case *True*
hence $\{0..n\} = \{0..<$*subdegree* $f\} \cup \{$*subdegree* $f..n\}$ **by** *auto*
moreover have $\{0..<$*subdegree* $f\} \cap \{$*subdegree* $f..n\} = \{\}$ **by** *auto*
ultimately show *?thesis*
using *nth-less-subdegree-zero*[*of* - f]
by (*simp add*: *fps-mult-nth sum.union-disjoint*)
qed (*simp add*: *fps-mult-nth nth-less-subdegree-zero*)

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−
have $\{0..n\} = \{0..n -$ *subdegree* $g\} \cup \{n -$ *subdegree* $g<..n\}$ **by** *auto*
moreover have $\{0..n -$ *subdegree* $g\} \cap \{n -$ *subdegree* $g<..n\} = \{\}$ **by** *auto*
moreover have $\forall\, i \in \{n -$ *subdegree* $g<..n\}.\ g\ \$\ (n - i) = 0$
using *nth-less-subdegree-zero*[*of* - g] **by** *auto*
ultimately show *?thesis* **by** (*simp add*: *fps-mult-nth sum.union-disjoint*)
qed

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 (*cases subdegree* $f \leq n -$ *subdegree* g)
case *True*
hence $\{$*subdegree* $f..n\} = \{$*subdegree* $f..n -$ *subdegree* $g\} \cup \{n -$ *subdegree* $g<..n\}$
by *auto*
moreover have $\{$*subdegree* $f..n -$ *subdegree* $g\} \cap \{n -$ *subdegree* $g<..n\} = \{\}$
by *auto*
moreover have $\forall\, i \in \{n -$ *subdegree* $g<..n\}.\ f\ \$\ i * g\ \$\ (n - i) = 0$
using *nth-less-subdegree-zero*[*of* - g] **by** *auto*
ultimately show *?thesis*
using *fps-mult-nth-conv-upto-subdegree-left*[*of f g n*]
by (*simp add*: *sum.union-disjoint*)
next
case *False*
hence *1*: *subdegree* $f >$ *n* − *subdegree* g **by** *simp*
show *?thesis*
proof (*cases* $f*g = 0$)
case *False*
with *1* **have** $n <$ *subdegree* $(f*g)$ **using** *fps-mult-subdegree-ge*[*of f g*] **by** *simp*

```
    with 1 show ?thesis by auto
  qed (simp add: 1)
qed

lemma fps-mult-nth-outside-subdegrees:
  fixes f g :: ('a :: {mult-zero,comm-monoid-add}) fps
  shows n < subdegree f ⟹ (f * g) $ n = 0
  and   n < subdegree g ⟹ (f * g) $ n = 0
  by    (auto simp: fps-mult-nth-conv-inside-subdegrees)
```

5.3 Ring structure

```
instance fps :: (semigroup-add) semigroup-add
proof
  fix a b c :: 'a fps
  show a + b + c = a + (b + c)
    by (simp add: fps-ext add.assoc)
qed

instance fps :: (ab-semigroup-add) ab-semigroup-add
proof
  fix a b :: 'a fps
  show a + b = b + a
    by (simp add: fps-ext add.commute)
qed

instance fps :: (monoid-add) monoid-add
proof
  fix a :: 'a fps
  show 0 + a = a by (simp add: fps-ext)
  show a + 0 = a by (simp add: fps-ext)
qed

instance fps :: (comm-monoid-add) comm-monoid-add
proof
  fix a :: 'a fps
  show 0 + a = a by (simp add: fps-ext)
qed

instance fps :: (cancel-semigroup-add) cancel-semigroup-add
proof
  fix a b c :: 'a fps
  show b = c if a + b = a + c
    using that by (simp add: expand-fps-eq)
  show b = c if b + a = c + a
    using that by (simp add: expand-fps-eq)
qed

instance fps :: (cancel-ab-semigroup-add) cancel-ab-semigroup-add
```

proof
fix $a\ b\ c :: 'a\ fps$
show $a + b - a = b$
by (*simp add*: *expand-fps-eq*)
show $a - b - c = a - (b + c)$
by (*simp add*: *expand-fps-eq diff-diff-eq*)
qed

instance *fps* :: (*cancel-comm-monoid-add*) *cancel-comm-monoid-add* **..**

instance *fps* :: (*group-add*) *group-add*
proof
fix $a\ b :: 'a\ fps$
show $- a + a = 0$ **by** (*simp add*: *fps-ext*)
show $a + - b = a - b$ **by** (*simp add*: *fps-ext*)
qed

instance *fps* :: (*ab-group-add*) *ab-group-add*
proof
fix $a\ b :: 'a\ fps$
show $- a + a = 0$ **by** (*simp add*: *fps-ext*)
show $a - b = a + - b$ **by** (*simp add*: *fps-ext*)
qed

instance *fps* :: (*zero-neq-one*) *zero-neq-one*
by *standard* (*simp add*: *expand-fps-eq*)

lemma *fps-mult-assoc-lemma*:
fixes $k :: nat$
and $f :: nat \Rightarrow nat \Rightarrow nat \Rightarrow 'a$::*comm-monoid-add*
shows $(\sum j{=}0..k.\ \sum i{=}0..j.\ f\ i\ (j - i)\ (n - j)) =$
$(\sum j{=}0..k.\ \sum i{=}0..k - j.\ f\ j\ i\ (n - j - i))$
by (*induct k*) (*simp-all add*: *Suc-diff-le sum.distrib add.assoc*)

instance *fps* :: (*semiring-0*) *semiring-0*
proof
fix $a\ b\ c :: 'a\ fps$
show $(a + b) * c = a * c + b * c$
by (*simp add*: *expand-fps-eq fps-mult-nth distrib-right sum.distrib*)
show $a * (b + c) = a * b + a * c$
by (*simp add*: *expand-fps-eq fps-mult-nth distrib-left sum.distrib*)
show $(a * b) * c = a * (b * c)$
proof (*rule fps-ext*)
fix $n :: nat$
have $(\sum j{=}0..n.\ \sum i{=}0..j.\ a\$i * b\$(j - i) * c\$(n - j)) =$
$(\sum j{=}0..n.\ \sum i{=}0..n - j.\ a\$j * b\$i * c\$(n - j - i))$
by (*rule fps-mult-assoc-lemma*)
then show $((a * b) * c)\ \$\ n = (a * (b * c))\ \$\ n$
by (*simp add*: *fps-mult-nth sum-distrib-left sum-distrib-right mult.assoc*)

```
  qed
qed

instance fps :: (semiring-0-cancel) semiring-0-cancel ..

lemma fps-mult-commute-lemma:
  fixes n :: nat
    and f :: nat ⇒ nat ⇒ 'a::comm-monoid-add
  shows (∑ i=0..n. f i (n − i)) = (∑ i=0..n. f (n − i) i)
  by (rule sum.reindex-bij-witness[where i=(−) n and j=(−) n]) auto

instance fps :: (comm-semiring-0) comm-semiring-0
proof
  fix a b c :: 'a fps
  show a * b = b * a
  proof (rule fps-ext)
    fix n :: nat
    have (∑ i=0..n. a$i * b$(n − i)) = (∑ i=0..n. a$(n − i) * b$i)
      by (rule fps-mult-commute-lemma)
    then show (a * b) $ n = (b * a) $ n
      by (simp add: fps-mult-nth mult.commute)
  qed
qed (simp add: distrib-right)

instance fps :: (comm-semiring-0-cancel) comm-semiring-0-cancel ..

instance fps :: (semiring-1) semiring-1
proof
  fix a :: 'a fps
  show 1 * a = a a * 1 = a by (simp-all add: fps-one-mult)
qed

instance fps :: (comm-semiring-1) comm-semiring-1
  by standard simp

instance fps :: (semiring-1-cancel) semiring-1-cancel ..

lemma fps-square-nth: (f^2) $ n = (∑ k≤n. f $ k * f $ (n − k))
  by (simp add: power2-eq-square fps-mult-nth atLeast0AtMost)

lemma fps-sum-nth: sum f S $ n = sum (λk. (f k) $ n) S
proof (cases finite S)
  case True
  then show ?thesis by (induct set: finite) auto
next
  case False
  then show ?thesis by simp
qed
```

definition *fps-const c = Abs-fps* (λ*n. if n = 0 then c else 0*)

lemma *fps-nth-fps-const* [*simp*]: *fps-const c $ n* = (*if n = 0 then c else 0*)
unfolding *fps-const-def* **by** *simp*

lemma *fps-const-0-eq-0* [*simp*]: *fps-const 0 = 0*
by (*simp add*: *fps-ext*)

lemma *fps-const-nonzero-eq-nonzero*: *c* ≠ *0* ⟹ *fps-const c* ≠ *0*
using *fps-nonzeroI*[*of fps-const c 0*] **by** *simp*

lemma *fps-const-eq-0-iff* [*simp*]: *fps-const c = 0* ⟷ *c = 0*
by (*auto simp*: *fps-eq-iff*)

lemma *fps-const-1-eq-1* [*simp*]: *fps-const 1 = 1*
by (*simp add*: *fps-ext*)

lemma *fps-const-eq-1-iff* [*simp*]: *fps-const c = 1* ⟷ *c = 1*
by (*auto simp*: *fps-eq-iff*)

lemma *subdegree-fps-const* [*simp*]: *subdegree* (*fps-const c*) = *0*
by (*cases c = 0*) (*auto intro*!: *subdegreeI*)

lemma *fps-const-neg* [*simp*]: − (*fps-const* (*c*::′*a*::*group-add*)) = *fps-const* (− *c*)
by (*simp add*: *fps-ext*)

lemma *fps-const-add* [*simp*]: *fps-const* (*c*::′*a*::*monoid-add*) + *fps-const d* = *fps-const* (*c* + *d*)
by (*simp add*: *fps-ext*)

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*)
by (*simp add*: *fps-ext*)

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*)
by (*simp add*: *fps-ext*)

lemma *fps-const-sub* [*simp*]: *fps-const* (*c*::′*a*::*group-add*) − *fps-const d* = *fps-const* (*c* − *d*)
by (*simp add*: *fps-ext*)

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*)
by (*simp add*: *fps-eq-iff fps-mult-nth sum.neutral*)

lemma *fps-const-mult-left*:
fps-const (*c*::*'a*::{*comm-monoid-add*,*mult-zero*}) ∗ *f* = *Abs-fps* (λ*n*. *c* ∗ *f*$*n*)
unfolding *fps-eq-iff fps-mult-nth*
by (*simp add*: *fps-const-def mult-delta-left*)

lemma *fps-const-mult-right*:
f ∗ *fps-const* (*c*::*'a*::{*comm-monoid-add*,*mult-zero*}) = *Abs-fps* (λ*n*. *f*$*n* ∗ *c*)
unfolding *fps-eq-iff fps-mult-nth*
by (*simp add*: *fps-const-def mult-delta-right*)

lemma *fps-mult-left-const-nth* [*simp*]:
(*fps-const* (*c*::*'a*::{*comm-monoid-add*,*mult-zero*}) ∗ *f*)$*n* = *c*∗ *f*$*n*
by (*simp add*: *fps-mult-nth mult-delta-left*)

lemma *fps-mult-right-const-nth* [*simp*]:
(*f* ∗ *fps-const* (*c*::*'a*::{*comm-monoid-add*,*mult-zero*}))$*n* = *f*$*n* ∗ *c*
by (*simp add*: *fps-mult-nth mult-delta-right*)

lemma *fps-const-power* [*simp*]: *fps-const c* ^ *n* = *fps-const* (*c*^*n*)
by (*induct n*) *auto*

instance *fps* :: (*ring*) *ring* **..**

instance *fps* :: (*comm-ring*) *comm-ring* **..**

instance *fps* :: (*ring-1*) *ring-1* **..**

instance *fps* :: (*comm-ring-1*) *comm-ring-1* **..**

instance *fps* :: (*semiring-no-zero-divisors*) *semiring-no-zero-divisors*
proof
fix *a b* :: *'a fps*
assume *a* ≠ *0* **and** *b* ≠ *0*
hence (*a* ∗ *b*) $ (*subdegree a* + *subdegree b*) ≠ *0* **by** *simp*
thus *a* ∗ *b* ≠ *0* **using** *fps-nonzero-nth* **by** *fast*
qed

instance *fps* :: (*semiring-1-no-zero-divisors*) *semiring-1-no-zero-divisors* **..**

instance *fps* :: ({*cancel-semigroup-add*,*semiring-no-zero-divisors-cancel*})
semiring-no-zero-divisors-cancel
proof
fix *a b c* :: *'a fps*
show (*a* ∗ *c* = *b* ∗ *c*) = (*c* = *0* ∨ *a* = *b*)
proof
assume *ab*: *a* ∗ *c* = *b* ∗ *c*
have *c* ≠ *0* ⟹ *a* = *b*

```
    proof (rule fps-ext)
      fix n
      assume c: c ≠ 0
      show a $ n = b $ n
      proof (induct n rule: nat-less-induct)
        case (1 n)
        with ab c show ?case
          using fps-mult-nth-conv-upto-subdegree-right[of a c subdegree c + n]
                fps-mult-nth-conv-upto-subdegree-right[of b c subdegree c + n]
          by    (cases n) auto
      qed
    qed
    thus c = 0 ∨ a = b by fast
  qed auto
  show (c * a = c * b) = (c = 0 ∨ a = b)
  proof
    assume ab: c * a = c * b
    have c ≠ 0 ⟹ a = b
    proof (rule fps-ext)
      fix n
      assume c: c ≠ 0
      show a $ n = b $ n
      proof (induct n rule: nat-less-induct)
        case (1 n)
        moreover have ∀ i∈{Suc (subdegree c)..subdegree c + n}. subdegree c + n
− i < n by auto
        ultimately show ?case
          using ab c fps-mult-nth-conv-upto-subdegree-left[of c a subdegree c + n]
                fps-mult-nth-conv-upto-subdegree-left[of c b subdegree c + n]
          by    (simp add: sum.atLeast-Suc-atMost)
      qed
    qed
    thus c = 0 ∨ a = b by fast
  qed auto
qed

instance fps :: (ring-no-zero-divisors) ring-no-zero-divisors ..

instance fps :: (ring-1-no-zero-divisors) ring-1-no-zero-divisors ..

instance fps :: (idom) idom ..

lemma fps-of-nat: fps-const (of-nat c) = of-nat c
  by (induction c) (simp-all add: fps-const-add [symmetric] del: fps-const-add)

lemma fps-of-int: fps-const (of-int c) = of-int c
  by (induction c) (simp-all add: fps-const-minus [symmetric] fps-of-nat fps-const-neg
[symmetric]
                                  del: fps-const-minus fps-const-neg)
```

lemma *semiring-char-fps* [*simp*]: *CHAR*('*a* :: *comm-semiring-1 fps*) = *CHAR*('*a*)
by (*rule CHAR-eqI*) (*auto simp flip*: *fps-of-nat simp*: *of-nat-eq-0-iff-char-dvd*)

instance *fps* :: ({*semiring-prime-char*,*comm-semiring-1*}) *semiring-prime-char*
by (*rule semiring-prime-charI*) *auto*
instance *fps* :: ({*comm-semiring-prime-char*,*comm-semiring-1*}) *comm-semiring-prime-char*
by *standard*
instance *fps* :: ({*comm-ring-prime-char*,*comm-semiring-1*}) *comm-ring-prime-char*
by *standard*
instance *fps* :: ({*idom-prime-char*,*comm-semiring-1*}) *idom-prime-char*
by *standard*

lemma *fps-numeral-fps-const*: *numeral k* = *fps-const* (*numeral k*)
by (*induct k*) (*simp-all only*: *numeral.simps fps-const-1-eq-1 fps-const-add* [*symmetric*])

lemmas *numeral-fps-const* = *fps-numeral-fps-const*

lemma *neg-numeral-fps-const*:
(− *numeral k* :: '*a* :: *ring-1 fps*) = *fps-const* (− *numeral k*)
by (*simp add*: *numeral-fps-const*)

lemma *fps-numeral-nth*: *numeral n* $ *i* = (*if i* = *0 then numeral n else 0*)
by (*simp add*: *numeral-fps-const*)

lemma *fps-numeral-nth-0* [*simp*]: *numeral n* $ *0* = *numeral n*
by (*simp add*: *numeral-fps-const*)

lemma *subdegree-numeral* [*simp*]: *subdegree* (*numeral n*) = *0*
by (*simp add*: *numeral-fps-const*)

lemma *fps-nth-of-nat* [*simp*]:
(*of-nat c*) $ *n* = (*if n=0 then of-nat c else 0*)
by (*simp add*: *fps-of-nat*[*symmetric*])

lemma *fps-nth-of-int* [*simp*]:
(*of-int c*) $ *n* = (*if n=0 then of-int c else 0*)
by (*simp add*: *fps-of-int*[*symmetric*])

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*
by (*simp-all add*: *fps-of-nat*[*symmetric*])

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*
by (*simp-all add*: *fps-of-int*[*symmetric*])

lemma *numeral-neq-fps-zero* [*simp*]: (*numeral f* :: *'a* :: *field-char-0 fps*) ≠ *0*
proof
  assume *numeral f* = (*0* :: *'a fps*)
  from *arg-cong*[*of - - λF. F $ 0, OF this*] **show** *False* **by** *simp*
qed

instance *fps* :: (*semiring-char-0*) *semiring-char-0*
proof
  show *inj* (*of-nat* :: *nat* ⇒ *'a fps*)
  proof
    fix *m n* :: *nat*
    assume *of-nat m* = (*of-nat n* :: *'a fps*)
    hence *fps-nth* (*of-nat m*) *0* = (*fps-nth* (*of-nat n*) *0* :: *'a*)
      by (*simp only*:)
    thus *m* = *n*
      by *simp*
  qed
qed

lemma *subdegree-power-ge*:
  f^n ≠ *0* ⟹ *subdegree* (*f^n*) ≥ *n* ∗ *subdegree f*
proof (*induct n*)
  case (*Suc n*) **thus** *?case* **using** *fps-mult-subdegree-ge* **by** *fastforce*
qed *simp*

lemma *fps-pow-nth-below-subdegree*:
  k < *n* ∗ *subdegree f* ⟹ (*f^n*) $ *k* = *0*
proof (*cases f^n* = *0*)
  case *False*
  assume *k* < *n* ∗ *subdegree f*
  with *False* **have** *k* < *subdegree* (*f^n*) **using** *subdegree-power-ge*[*of f n*] **by** *simp*
  thus (*f^n*) $ *k* = *0* **by** *auto*
qed *simp*

lemma *fps-pow-base* [*simp*]:
  (*f ^ n*) $ (*n* ∗ *subdegree f*) = (*f* $ *subdegree f*) ^ *n*
proof (*induct n*)
  case (*Suc n*)
  show *?case*
  proof (*cases Suc n* ∗ *subdegree f* < *subdegree f* + *subdegree* (*f^n*))
    case *True* **with** *Suc* **show** *?thesis*
      by (*auto simp*: *fps-mult-nth-eq0 distrib-right*)
  next
    case *False*
    hence ∀ *i*∈{*Suc* (*subdegree f*)..*Suc n* ∗ *subdegree f* − *subdegree* (*f ^ n*)}.
          f ^ n $ (*Suc n* ∗ *subdegree f* − *i*) = *0*
     by (*auto simp*: *fps-pow-nth-below-subdegree*)
   with *False Suc* **show** *?thesis*
      using *fps-mult-nth-conv-inside-subdegrees*[*of f f^n Suc n* ∗ *subdegree f*]

```
          sum.atLeast-Suc-atMost[of
            subdegree f
            Suc n * subdegree f - subdegree (f ^ n)
            λi. f $ i * f ^ n $ (Suc n * subdegree f - i)
          ]
    by    simp
  qed
qed simp

lemma subdegree-power-eqI:
  fixes f :: 'a::semiring-1 fps
  shows (f $ subdegree f) ^ n ≠ 0 ⟹ subdegree (f ^ n) = n * subdegree f
proof (induct n)
  case (Suc n)
  from Suc have 1: subdegree (f ^ n) = n * subdegree f by fastforce
  with Suc(2) have f $ subdegree f * f ^ n $ subdegree (f ^ n) ≠ 0 by simp
  with 1 show ?case using subdegree-mult'[of f f^n] by simp
qed simp

lemma subdegree-power [simp]:
  subdegree ((f :: ('a :: semiring-1-no-zero-divisors) fps) ^ n) = n * subdegree f
  by (cases f = 0; induction n) simp-all

lemma subdegree-prod:
  fixes f :: 'a ⇒ 'b :: idom fps
  assumes ⋀x. x ∈ A ⟹ f x ≠ 0
  shows   subdegree (∏ x∈A. f x) = (∑ x∈A. subdegree (f x))
  using assms by (induction A rule: infinite-finite-induct) auto


lemma minus-one-power-iff: (- (1::'a::ring-1)) ^ n = (if even n then 1 else - 1)
  by (induct n) auto

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
  by (auto intro!: subdegreeI simp: fps-X-def)

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 (cases n)
  case (Suc m)
  moreover have (fps-X * f) $ Suc m = f $ (Suc m - 1)
  proof (cases m)
    case 0 thus ?thesis using fps-mult-nth-1[of fps-X f] by (simp add: fps-X-def)
  next
   case (Suc k) thus ?thesis by (simp add: fps-mult-nth fps-X-def sum.atLeast-Suc-atMost)
  qed
```

ultimately show *?thesis* **by** *simp*
qed (*simp add*: *fps-X-def*)

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 (*cases n*)
case (*Suc m*)
moreover have (*a* ∗ *fps-X*) $ *Suc m* = *a* $ (*Suc m* − *1*)
proof (*cases m*)
case *0* **thus** *?thesis* **using** *fps-mult-nth-1*[*of a fps-X*] **by** (*simp add*: *fps-X-def*)
next
case (*Suc k*)
hence (*a* ∗ *fps-X*) $ *Suc m* = (∑ *i*=*0*..*k*. *a*$*i* ∗ *fps-X*$(*Suc m* − *i*)) + *a*$(*Suc k*)
by (*simp add*: *fps-mult-nth fps-X-def*)
moreover have ∀ *i*∈{*0*..*k*}. *a*$*i* ∗ *fps-X*$(*Suc m* − *i*) = *0* **by** (*auto simp*: *Suc fps-X-def*)
ultimately show *?thesis* **by** (*simp add*: *Suc*)
qed
ultimately show *?thesis* **by** *simp*
qed (*simp add*: *fps-X-def*)

lemma *fps-mult-fps-X-commute*:
fixes *a* :: *'a*::{*comm-monoid-add*,*mult-zero*,*monoid-mult*} *fps*
shows *fps-X* ∗ *a* = *a* ∗ *fps-X*
by (*simp add*: *fps-eq-iff*)

lemma *fps-mult-fps-X-power-commute*: *fps-X* ^ *k* ∗ *a* = *a* ∗ *fps-X* ^ *k*
proof (*induct k*)
case (*Suc k*)
hence *fps-X* ^ *Suc k* ∗ *a* = *a* ∗ *fps-X* ∗ *fps-X* ^ *k*
by (*simp add*: *mult.assoc fps-mult-fps-X-commute*[*symmetric*])
thus *?case* **by** (*simp add*: *mult.assoc*)
qed *simp*

lemma *fps-subdegree-mult-fps-X*:
fixes *f* :: *'a*::{*comm-monoid-add*,*mult-zero*,*monoid-mult*} *fps*
assumes *f* ≠ *0*
shows *subdegree* (*fps-X* ∗ *f*) = *subdegree f* + *1*
and *subdegree* (*f* ∗ *fps-X*) = *subdegree f* + *1*
proof−
show *subdegree* (*fps-X* ∗ *f*) = *subdegree f* + *1*
proof (*intro subdegreeI*)
fix *i* :: *nat* **assume** *i*: *i* < *subdegree f* + *1*
show (*fps-X* ∗ *f*) $ *i* = *0*
proof (*cases i*=*0*)
case *False* **with** *i* **show** *?thesis* **by** (*simp add*: *nth-less-subdegree-zero*)
next

```
    case True thus ?thesis using fps-X-mult-nth[of f i] by simp
   qed
  qed (simp add: assms)
  thus subdegree (f * fps-X) = subdegree f + 1
   by (simp add: fps-mult-fps-X-commute)
qed

lemma fps-mult-fps-X-nonzero:
  fixes  f :: 'a::{comm-monoid-add,mult-zero,monoid-mult} fps
  assumes f ≠ 0
  shows   fps-X * f ≠ 0
  and     f * fps-X ≠ 0
  using   assms fps-subdegree-mult-fps-X[of f]
          fps-nonzero-subdegree-nonzeroI[of fps-X * f]
          fps-nonzero-subdegree-nonzeroI[of f * fps-X]
  by      auto

lemma fps-mult-fps-X-power-nonzero:
  assumes f ≠ 0
  shows   fps-X ^ n * f ≠ 0
  and     f * fps-X ^ n ≠ 0
proof -
  show fps-X ^ n * f ≠ 0
   by (induct n) (simp-all add: assms mult.assoc fps-mult-fps-X-nonzero(1))
  thus f * fps-X ^ n ≠ 0
   by (simp add: fps-mult-fps-X-power-commute)
qed

lemma fps-X-power-iff: fps-X ^ n = Abs-fps (λm. if m = n then 1 else 0)
  by (induction n) (auto simp: fps-eq-iff)

lemma fps-X-nth[simp]: fps-X$n = (if n = 1 then 1 else 0)
  by (simp add: fps-X-def)

lemma fps-X-power-nth[simp]: (fps-X^k) $n = (if n = k then 1 else 0)
  by (simp add: fps-X-power-iff)

lemma fps-X-power-subdegree: subdegree (fps-X^n) = n
  by (auto intro: subdegreeI)

lemma fps-X-power-mult-nth:
  (fps-X^k * f) $ n = (if n < k then 0 else f $ (n - k))
  by  (cases n<k)
        (simp-all add: fps-mult-nth-conv-upto-subdegree-left fps-X-power-subdegree
sum.atLeast-Suc-atMost)

lemma fps-X-power-mult-right-nth:
  (f * fps-X^k) $ n = (if n < k then 0 else f $ (n - k))
  using fps-mult-fps-X-power-commute[of k f] fps-X-power-mult-nth[of k f] by simp
```

lemma *fps-subdegree-mult-fps-X-power*:
assumes $f \neq 0$
shows *subdegree* (*fps-X* ^ n * f) = *subdegree* f + n
and *subdegree* (f * *fps-X* ^ n) = *subdegree* f + n
proof −
from *assms* **show** *subdegree* (*fps-X* ^ n * f) = *subdegree* f + n
by (*induct* n)
(*simp-all add*: *algebra-simps fps-subdegree-mult-fps-X*(*1*) *fps-mult-fps-X-power-nonzero*(*1*))
thus *subdegree* (f * *fps-X* ^ n) = *subdegree* f + n
by (*simp add*: *fps-mult-fps-X-power-commute*)
qed

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 (*cases* n)
case *0*
then show *?thesis*
by (*simp add*: *fps-mult-nth*)
next
case (*Suc m*)
have ((*1* + *fps-X*)*a) $ n = *sum* (λ*i*. (*1* + *fps-X*) $ i * a $ (n − i)) {*0*..n}
by (*simp add*: *fps-mult-nth*)
also have ... = *sum* (λ*i*. (*1*+*fps-X*)$$i$ * $a$$(n−i)) {*0*.. *1*}
unfolding *Suc* **by** (*rule sum.mono-neutral-right*) *auto*
also have ... = (*if* n = *0 then* $a$$$n$ *else* $a$$$n$ + $a$$(n − *1*))
by (*simp add*: *Suc*)
finally show *?thesis* **.**
qed

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*))
using *fps-mult-fps-X-plus-1-nth*
by (*simp add*: *distrib-left fps-mult-fps-X-commute distrib-right*)

lemma *fps-X-neq-fps-const* [*simp*]: (*fps-X* :: *'a* :: *zero-neq-one fps*) ≠ *fps-const c*
proof
assume (*fps-X*::*'a fps*) = *fps-const* (*c*::*'a*)
hence *fps-X*$*1* = (*fps-const* (*c*::*'a*))$*1* **by** (*simp only*:)
thus *False* **by** *auto*
qed

lemma *fps-X-neq-zero* [*simp*]: (*fps-X* :: *'a* :: *zero-neq-one fps*) ≠ *0*
by (*simp only*: *fps-const-0-eq-0*[*symmetric*] *fps-X-neq-fps-const*) *simp*

lemma *fps-X-neq-one* [*simp*]: (*fps-X* :: *'a* :: *zero-neq-one fps*) ≠ *1*
by (*simp only*: *fps-const-1-eq-1*[*symmetric*] *fps-X-neq-fps-const*) *simp*

lemma *fps-X-neq-numeral* [*simp*]: *fps-X* $\neq$ *numeral c*
by (*simp only*: *numeral-fps-const fps-X-neq-fps-const*) *simp*

lemma *fps-X-pow-eq-fps-X-pow-iff* [*simp*]: *fps-X* $\hat{}$ *m* = *fps-X* $\hat{}$ *n* $\longleftrightarrow$ *m* = *n*
proof
assume (*fps-X* :: *'a fps*) $\hat{}$ *m* = *fps-X* $\hat{}$ *n*
hence (*fps-X* :: *'a fps*) $\hat{}$ *m* $ *m* = *fps-X* $\hat{}$ *n* $ *m* **by** (*simp only*:)
thus *m* = *n* **by** (*simp split*: *if-split-asm*)
qed *simp-all*

5.4 Shifting and slicing

definition *fps-shift* :: *nat* $\Rightarrow$ *'a fps* $\Rightarrow$ *'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*)
by (*simp add*: *fps-shift-def*)

lemma *fps-shift-0* [*simp*]: *fps-shift 0 f* = *f*
by (*intro fps-ext*) (*simp add*: *fps-shift-def*)

lemma *fps-shift-zero* [*simp*]: *fps-shift n 0* = *0*
by (*intro fps-ext*) (*simp add*: *fps-shift-def*)

lemma *fps-shift-one*: *fps-shift n 1* = (*if n* = *0 then 1 else 0*)
by (*intro fps-ext*) (*simp add*: *fps-shift-def*)

lemma *fps-shift-fps-const*: *fps-shift n* (*fps-const c*) = (*if n* = *0 then fps-const c else 0*)
by (*intro fps-ext*) (*simp add*: *fps-shift-def*)

lemma *fps-shift-numeral*: *fps-shift n* (*numeral c*) = (*if n* = *0 then numeral c else 0*)
by (*simp add*: *numeral-fps-const fps-shift-fps-const*)

lemma *fps-shift-fps-X* [*simp*]:
n $\geq$ *1* $\Longrightarrow$ *fps-shift n fps-X* = (*if n* = *1 then 1 else 0*)
by (*intro fps-ext*) (*auto simp*: *fps-X-def*)

lemma *fps-shift-fps-X-power* [*simp*]:
n $\leq$ *m* $\Longrightarrow$ *fps-shift n* (*fps-X* $\hat{}$ *m*) = *fps-X* $\hat{}$ (*m* $-$ *n*)
by (*intro fps-ext*) *auto*

lemma *fps-shift-subdegree* [*simp*]:
n $\leq$ *subdegree f* $\Longrightarrow$ *subdegree* (*fps-shift n f*) = *subdegree f* $-$ *n*
by (*cases f*=*0*) (*auto intro*: *subdegreeI simp*: *nth-less-subdegree-zero*)

lemma *fps-shift-fps-shift*:

fps-shift $(m + n)$ f = *fps-shift* m (*fps-shift* n f)
by (*rule fps-ext*) (*simp add*: *add-ac*)

lemma *fps-shift-fps-shift-reorder*:
fps-shift m (*fps-shift* n f) = *fps-shift* n (*fps-shift* m f)
using *fps-shift-fps-shift*[*of* m n f] *fps-shift-fps-shift*[*of* n m f] **by** (*simp add*: *add.commute*)

lemma *fps-shift-rev-shift*:
$m \leq n \Longrightarrow$ *fps-shift* n (*Abs-fps* (λk. *if* $k<m$ *then 0 else* f \$ $(k-m)$)) = *fps-shift* $(n-m)$ f
$m > n \Longrightarrow$ *fps-shift* n (*Abs-fps* (λk. *if* $k<m$ *then 0 else* f \$ $(k-m)$)) =
Abs-fps (λk. *if* $k<m-n$ *then 0 else* f \$ $(k-(m-n))$)
proof −
assume $m \leq n$
thus *fps-shift* n (*Abs-fps* (λk. *if* $k<m$ *then 0 else* f \$ $(k-m)$)) = *fps-shift* $(n-m)$ f
by (*intro fps-ext*) *auto*
next
assume *mn*: $m > n$
hence $\bigwedge k.\ k \geq m-n \Longrightarrow k+n-m = k - (m-n)$ **by** *auto*
thus
fps-shift n (*Abs-fps* (λk. *if* $k<m$ *then 0 else* f \$ $(k-m)$)) =
Abs-fps (λk. *if* $k<m-n$ *then 0 else* f \$ $(k-(m-n))$)
by (*intro fps-ext*) *auto*
qed

lemma *fps-shift-add*:
fps-shift n $(f + g)$ = *fps-shift* n f + *fps-shift* n g
by (*simp add*: *fps-eq-iff*)

lemma *fps-shift-diff*:
fps-shift n $(f - g)$ = *fps-shift* n f − *fps-shift* n g
by (*auto intro*: *fps-ext*)

lemma *fps-shift-uminus*:
fps-shift n $(-f)$ = − *fps-shift* n f
by (*auto intro*: *fps-ext*)

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−
have *case1*: $\bigwedge a\ b{::}'b$ *fps*. $1 \leq$ *subdegree* $b \Longrightarrow$ *fps-shift* 1 $(a*b)$ = $a *$ *fps-shift* 1 b
proof (*rule fps-ext*)
fix a $b :: 'b$ *fps*
and $n :: nat$
assume *b*: $1 \leq$ *subdegree* b

```
    have ⋀i. i ≤ n ⟹ n + 1 − i = (n−i) + 1
      by (simp add: algebra-simps)
    with b show fps-shift 1 (a∗b) $ n = (a ∗ fps-shift 1 b) $ n
      by (simp add: fps-mult-nth nth-less-subdegree-zero)
  qed
  have n ≤ subdegree g ⟹ fps-shift n (h∗g) = h ∗ fps-shift n g
  proof (induct n)
    case (Suc n)
    have fps-shift (Suc n) (h∗g) = fps-shift 1 (fps-shift n (h∗g))
      by (simp add: fps-shift-fps-shift[symmetric])
    also have ... = h ∗ (fps-shift 1 (fps-shift n g))
      using Suc case1 by force
    finally show ?case by (simp add: fps-shift-fps-shift[symmetric])
  qed simp
  with assms show ?thesis by fast
qed

lemma fps-shift-mult-right-noncomm:
  assumes n ≤ subdegree (g :: 'b :: {comm-monoid-add, mult-zero} fps)
  shows fps-shift n (g∗h) = fps-shift n g ∗ h
proof−
  have case1: ⋀a b::'b fps. 1 ≤ subdegree a ⟹ fps-shift 1 (a∗b) = fps-shift 1 a ∗
b
  proof (rule fps-ext)
    fix a b :: 'b fps
    and n :: nat
    assume 1 ≤ subdegree a
    hence fps-shift 1 (a∗b) $ n = (∑ i=Suc 0..Suc n. a$i ∗ b$(n+1−i))
      using sum.atLeast-Suc-atMost[of 0 n+1 λi. a$i ∗ b$(n+1−i)]
      by    (simp add: fps-mult-nth nth-less-subdegree-zero)
    thus fps-shift 1 (a∗b) $ n = (fps-shift 1 a ∗ b) $ n
      using sum.shift-bounds-cl-Suc-ivl[of λi. a$i ∗ b$(n+1−i) 0 n]
      by    (simp add: fps-mult-nth)
  qed
  have n ≤ subdegree g ⟹ fps-shift n (g∗h) = fps-shift n g ∗ h
  proof (induct n)
    case (Suc n)
    have fps-shift (Suc n) (g∗h) = fps-shift 1 (fps-shift n (g∗h))
      by (simp add: fps-shift-fps-shift[symmetric])
    also have ... = (fps-shift 1 (fps-shift n g)) ∗ h
      using Suc case1 by force
    finally show ?case by (simp add: fps-shift-fps-shift[symmetric])
  qed simp
  with assms show ?thesis by fast
qed

lemma fps-shift-mult-right:
  assumes n ≤ subdegree (g :: 'b :: comm-semiring-0 fps)
  shows   fps-shift n (g∗h) = h ∗ fps-shift n g
```

by (*simp add*: *assms fps-shift-mult-right-noncomm mult.commute*)

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* $(m+n)$ $(f*g)$
using *assms*
by (*simp add*: *fps-shift-mult fps-shift-mult-right-noncomm fps-shift-fps-shift*)

lemma *fps-shift-subdegree-zero-iff* [*simp*]:
fps-shift (*subdegree f*) *f* = *0* $\longleftrightarrow$ *f* = *0*
by (*subst* (*1*) *nth-subdegree-zero-iff*[*symmetric*], *cases f* = *0*)
(*simp-all del*: *nth-subdegree-zero-iff*)

lemma *fps-shift-times-fps-X*:
fixes *f g* :: *'a*::{*comm-monoid-add*,*mult-zero*,*monoid-mult*} *fps*
shows *1* $\leq$ *subdegree f* $\Longrightarrow$ *fps-shift 1 f* $*$ *fps-X* = *f*
by (*intro fps-ext*) (*simp add*: *nth-less-subdegree-zero*)

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*
by (*intro fps-ext*) (*simp add*: *nth-less-subdegree-zero*)

lemma *fps-shift-times-fps-X''*:
fixes *f* :: *'a*::{*comm-monoid-add*,*mult-zero*,*monoid-mult*} *fps*
shows *1* $\leq$ *n* $\Longrightarrow$ *fps-shift n* (*f* $*$ *fps-X*) = *fps-shift* $(n - 1)$ *f*
by (*intro fps-ext*) (*simp add*: *nth-less-subdegree-zero*)

lemma *fps-shift-times-fps-X-power*:
n $\leq$ *subdegree f* $\Longrightarrow$ *fps-shift n f* $*$ *fps-X* ^ *n* = *f*
by (*intro fps-ext*) (*simp add*: *fps-X-power-mult-right-nth nth-less-subdegree-zero*)

lemma *fps-shift-times-fps-X-power'* [*simp*]:
fps-shift n (*f* $*$ *fps-X*^*n*) = *f*
by (*intro fps-ext*) (*simp add*: *fps-X-power-mult-right-nth nth-less-subdegree-zero*)

lemma *fps-shift-times-fps-X-power''*:
m $\leq$ *n* $\Longrightarrow$ *fps-shift n* (*f* $*$ *fps-X*^*m*) = *fps-shift* $(n - m)$ *f*
by (*intro fps-ext*) (*simp add*: *fps-X-power-mult-right-nth nth-less-subdegree-zero*)

lemma *fps-shift-times-fps-X-power'''*:
m $>$ *n* $\Longrightarrow$ *fps-shift n* (*f* $*$ *fps-X*^*m*) = *f* $*$ *fps-X*^$(m - n)$
proof (*cases f*=*0*)
case *False*
assume *m*: *m*>*n*
hence *m* = *n* + (*m*−*n*) **by** *auto*
with *False m* **show** *?thesis*
using *power-add*[*of fps-X*::*'a fps n m*−*n*]

fps-shift-mult-right-noncomm[*of n f* ∗ *fps-X*^*n fps-X*^(*m*−*n*)]
by (*simp add*: *mult.assoc fps-subdegree-mult-fps-X-power*(*2*))
qed *simp*

lemma *subdegree-decompose*:
f = *fps-shift* (*subdegree f*) *f* ∗ *fps-X* ^ *subdegree f*
by (*rule fps-ext*) (*auto simp*: *fps-X-power-mult-right-nth*)

lemma *subdegree-decompose′*:
n ≤ *subdegree f* ⟹ *f* = *fps-shift n f* ∗ *fps-X*^*n*
by (*rule fps-ext*) (*auto simp*: *fps-X-power-mult-right-nth intro*!: *nth-less-subdegree-zero*)

instantiation *fps* :: (*zero*) *unit-factor*
begin
definition *fps-unit-factor-def* [*simp*]:
unit-factor f = *fps-shift* (*subdegree f*) *f*
instance ..
end

lemma *fps-unit-factor-zero-iff*: *unit-factor* (*f*::′*a*::*zero fps*) = *0* ⟷ *f* = *0*
by *simp*

lemma *fps-unit-factor-nth-0*: *f* ≠ *0* ⟹ *unit-factor f* $ *0* ≠ *0*
by *simp*

lemma *fps-X-unit-factor*: *unit-factor* (*fps-X* :: ′*a* :: *zero-neq-one fps*) = *1*
by (*intro fps-ext*) *auto*

lemma *fps-X-power-unit-factor*: *unit-factor* (*fps-X* ^ *n*) = *1*
proof−
define *X* :: ′*a fps* **where** *X* ≡ *fps-X*
hence *unit-factor* (*X*^*n*) = *fps-shift n* (*X*^*n*)
by (*simp add*: *fps-X-power-subdegree*)
moreover have *fps-shift n* (*X*^*n*) = *1*
by (*auto intro*: *fps-ext simp*: *fps-X-power-iff X-def*)
ultimately show *?thesis* **by** (*simp add*: *X-def*)
qed

lemma *fps-unit-factor-decompose*:
f = *unit-factor f* ∗ *fps-X* ^ *subdegree f*
by (*simp add*: *subdegree-decompose*)

lemma *fps-unit-factor-decompose′*:
f = *fps-X* ^ *subdegree f* ∗ *unit-factor f*
using *fps-unit-factor-decompose* **by** (*simp add*: *fps-mult-fps-X-power-commute*)

lemma *fps-unit-factor-uminus*:
unit-factor (−*f*) = − *unit-factor* (*f*::′*a*::*group-add fps*)
by (*simp add*: *fps-shift-uminus*)

lemma *fps-unit-factor-shift*:
assumes $n \leq$ *subdegree f*
shows *unit-factor* (*fps-shift n f*) = *unit-factor f*
by (*simp add*: *assms fps-shift-fps-shift*[*symmetric*])

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 −
show *unit-factor* (*fps-X* $*$ *f*) = *unit-factor f*
by (*cases f*=*0*) (*auto intro*: *fps-ext simp*: *fps-subdegree-mult-fps-X*(*1*))
thus *unit-factor* (*f* $*$ *fps-X*) = *unit-factor f* **by** (*simp add*: *fps-mult-fps-X-commute*)
qed

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 −
show *unit-factor* (*fps-X* ^ *n* $*$ *f*) = *unit-factor f*
proof (*induct n*)
case (*Suc m*) **thus** *?case*
using *fps-unit-factor-mult-fps-X*(*1*)[*of fps-X* ^ *m* $*$ *f*] **by** (*simp add*: *mult.assoc*)
qed *simp*
thus *unit-factor* (*f* $*$ *fps-X* ^ *n*) = *unit-factor f*
by (*simp add*: *fps-mult-fps-X-power-commute*)
qed

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 −
show *unit-factor* (*f* $*$ *unit-factor g*) = *unit-factor* (*f* $*$ *g*)
proof (*cases f*$*$*g* = *0*)
case *False* **thus** *?thesis*
using *fps-mult-subdegree-ge*[*of f g*] *fps-unit-factor-shift*[*of subdegree g f*$*$*g*]
by (*simp add*: *fps-shift-mult*)
next
case *True*
moreover have *f* $*$ *unit-factor g* = *fps-shift* (*subdegree g*) (*f*$*$*g*)
by (*simp add*: *fps-shift-mult*)
ultimately show *?thesis* **by** *simp*
qed
show *unit-factor* (*unit-factor f* $*$ *g*) = *unit-factor* (*f* $*$ *g*)
proof (*cases f*$*$*g* = *0*)
case *False* **thus** *?thesis*
using *fps-mult-subdegree-ge*[*of f g*] *fps-unit-factor-shift*[*of subdegree f f*$*$*g*]

```
    by    (simp add: fps-shift-mult-right-noncomm)
  next
    case True
    moreover have unit-factor f * g = fps-shift (subdegree f) (f*g)
      by (simp add: fps-shift-mult-right-noncomm)
    ultimately show ?thesis by simp
  qed
qed

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)
  using fps-unit-factor-mult-unit-factor(1)[of unit-factor f g]
        fps-unit-factor-mult-unit-factor(2)[of f g]
  by    simp

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
  using   assms
  by      (simp add: subdegree-mult' fps-shift-mult-both)

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
  using fps-unit-factor-mult'[of f g]
  by    (cases f=0 ∨ g=0) auto

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)
  unfolding fps-cutoff-def by simp

lemma fps-cutoff-zero-iff: fps-cutoff n f = 0 ⟷ (f = 0 ∨ n ≤ subdegree f)
proof
  assume A: fps-cutoff n f = 0
  thus f = 0 ∨ n ≤ subdegree f
  proof (cases f = 0)
    assume f ≠ 0
    with A have n ≤ subdegree f
      by (intro subdegree-geI) (simp-all add: fps-eq-iff split: if-split-asm)
    thus ?thesis ..
  qed simp
qed (auto simp: fps-eq-iff intro: nth-less-subdegree-zero)

lemma fps-cutoff-0 [simp]: fps-cutoff 0 f = 0
  by (simp add: fps-eq-iff)
```

lemma *fps-cutoff-zero* [*simp*]: *fps-cutoff n 0 = 0*
by (*simp add*: *fps-eq-iff*)

lemma *fps-cutoff-one*: *fps-cutoff n 1 = (if n = 0 then 0 else 1)*
by (*simp add*: *fps-eq-iff*)

lemma *fps-cutoff-fps-const*: *fps-cutoff n (fps-const c) = (if n = 0 then 0 else fps-const c)*
by (*simp add*: *fps-eq-iff*)

lemma *fps-cutoff-numeral*: *fps-cutoff n (numeral c) = (if n = 0 then 0 else numeral c)*
by (*simp add*: *numeral-fps-const fps-cutoff-fps-const*)

lemma *fps-shift-cutoff*:
*fps-shift n f * fps-X^n + fps-cutoff n f = f*
by (*simp add*: *fps-eq-iff fps-X-power-mult-right-nth*)

lemma *fps-shift-cutoff'*:
*fps-X^n * fps-shift n f + fps-cutoff n f = f*
by (*simp add*: *fps-eq-iff fps-X-power-mult-nth*)

lemma *fps-cutoff-left-mult-nth*:
*k < n ⟹ (fps-cutoff n f * g) $ k = (f * g) $ k*
by (*simp add*: *fps-mult-nth*)

lemma *fps-cutoff-add*: *fps-cutoff n (f + g :: 'a :: monoid-add fps) = fps-cutoff n f + fps-cutoff n g*
by (*auto simp*: *fps-eq-iff*)

lemma *fps-cutoff-diff*: *fps-cutoff n (f − g :: 'a :: group-add fps) = fps-cutoff n f − fps-cutoff n g*
by (*auto simp*: *fps-eq-iff*)

lemma *fps-cutoff-uminus*: *fps-cutoff n (−f :: 'a :: group-add fps) = −fps-cutoff n f*
by (*auto simp*: *fps-eq-iff*)

lemma *fps-cutoff-right-mult-nth*:
assumes *k < n*
shows *(f * fps-cutoff n g) $ k = (f * g) $ k*
proof−
from *assms* **have** ∀ *i*∈{*0..k*}. *fps-cutoff n g $ (k − i) = g $ (k − i)* **by** *auto*
thus *?thesis* **by** (*simp add*: *fps-mult-nth*)
qed

lemma *fps-cutoff-eq-fps-cutoff-iff*:
fps-cutoff n f = fps-cutoff n g ⟷ (∀ *k*<*n. fps-nth f k = fps-nth g k)*
by (*subst fps-eq-iff*) *auto*

lemma *fps-conv-fps-X-power-mult-fps-shift*:
assumes $f = 0 \vee subdegree\ f \geq n$
shows $f = fps\text{-}X\ \hat{}\ n * fps\text{-}shift\ n\ f$
proof −
have $f = fps\text{-}X\ \hat{}\ n * fps\text{-}shift\ n\ f + fps\text{-}cutoff\ n\ f$
by (*auto simp*: *fps-eq-iff fps-X-power-mult-nth*)
also have $fps\text{-}cutoff\ n\ f = 0$
by (*subst fps-cutoff-zero-iff*) (*use assms* **in** *auto*)
finally show *?thesis* **by** *simp*
qed

5.5 Metrizability

instantiation *fps* :: ({*minus*,*zero*}) *dist*
begin

definition
dist-fps-def: $dist\ (a :: 'a\ fps)\ b = (if\ a = b\ then\ 0\ else\ inverse\ (2\ \hat{}\ subdegree\ (a - b)))$

lemma *dist-fps-ge0*: $dist\ (a :: 'a\ fps)\ b \geq 0$
by (*simp add*: *dist-fps-def*)

instance ..

end

instantiation *fps* :: (*group-add*) *metric-space*
begin

definition *uniformity-fps-def*:
$(uniformity :: ('a\ fps \times 'a\ fps)\ filter) = (INF\ e{\in}\{0 <..\}.\ principal\ \{(x, y).\ dist\ x\ y < e\})$

definition *open-fps-def'*:
$open\ (U :: 'a\ fps\ set) \longleftrightarrow (\forall x{\in}U.\ eventually\ (\lambda(x', y).\ x' = x \longrightarrow y \in U)\ uniformity)$

lemma *dist-fps-sym*: $dist\ (a :: 'a\ fps)\ b = dist\ b\ a$
by (*simp add*: *dist-fps-def*)

instance
proof
show *th*: $dist\ a\ b = 0 \longleftrightarrow a = b$ **for** $a\ b :: 'a\ fps$
by (*simp add*: *dist-fps-def split*: *if-split-asm*)
then have *th'*[*simp*]: $dist\ a\ a = 0$ **for** $a :: 'a\ fps$ **by** *simp*

fix $a\ b\ c :: 'a\ fps$

```
  consider a = b | c = a ∨ c = b | a ≠ b a ≠ c b ≠ c by blast
  then show dist a b ≤ dist a c + dist b c
  proof cases
    case 1
    then show ?thesis by (simp add: dist-fps-def)
  next
    case 2
    then show ?thesis
      by (cases c = a) (simp-all add: th dist-fps-sym)
  next
    case neq: 3
    have False if dist a b > dist a c + dist b c
    proof −
      let ?n = subdegree (a − b)
      from neq have dist a b > 0 dist b c > 0 and dist a c > 0 by (simp-all add:
dist-fps-def)
      with that have dist a b > dist a c and dist a b > dist b c by simp-all
      with neq have ?n < subdegree (a − c) and ?n < subdegree (b − c)
        by (simp-all add: dist-fps-def field-simps)
      hence (a − c) $ ?n = 0 and (b − c) $ ?n = 0
        by (simp-all only: nth-less-subdegree-zero)
      hence (a − b) $ ?n = 0 by simp
     moreover from neq have (a − b) $ ?n ≠ 0 by (intro nth-subdegree-nonzero)
simp-all
      ultimately show False by contradiction
    qed
    thus ?thesis by (auto simp add: not-le[symmetric])
  qed
qed (rule open-fps-def' uniformity-fps-def)+

end

lemma open-fps-def: open (S :: 'a::group-add fps set) = (∀ a ∈ S. ∃ r. r >0 ∧ {y.
dist y a < r} ⊆ S)
  unfolding open-dist subset-eq by simp
```

Topology

5.6 The topology of formal power series

A set of formal power series is open iff for any power series f in it, there exists some number n such that all power series that agree with f on the first n components are also in it.

```
lemma open-fps-iff:
  open A ⟷ (∀ F∈A. ∃ n. {G. fps-cutoff n G = fps-cutoff n F} ⊆ A)
proof
  assume open A
  show ∀ F∈A. ∃ n. {G. fps-cutoff n G = fps-cutoff n F} ⊆ A
```

```
  proof
    fix F :: 'a fps
    assume F: F ∈ A
    with ‹open A› obtain e where e: e > 0 ⋀G. dist G F < e ⟹ G ∈ A
      by (force simp: open-fps-def)
    thm dist-fps-def
    have filterlim (λn. (1/2)^n :: real) (nhds 0) at-top
      by (intro LIMSEQ-realpow-zero) auto
    from order-tendstoD(2)[OF this e(1)] have eventually (λn. 1 / 2 ^ n < e)
at-top
      by (simp add: power-divide)
    then obtain n where n: 1 / 2 ^ n < e
      by (auto simp: eventually-sequentially)
    show ∃n. {G. fps-cutoff n G = fps-cutoff n F} ⊆ A
    proof (rule exI[of - n], safe)
      fix G assume ∗: fps-cutoff n G = fps-cutoff n F
      show G ∈ A
      proof (cases G = F)
        case False
        hence dist G F = inverse (2 ^ subdegree (G − F))
          by (auto simp: dist-fps-def)
        also have subdegree (G − F) ≥ n
        proof (rule subdegree-geI)
          fix i assume i < n
          hence fps-nth (G − F) i = fps-nth (fps-cutoff n G − fps-cutoff n F) i
            by (auto simp: fps-eq-iff)
          also from ∗ have ... = 0
            by simp
          finally show fps-nth (G − F) i = 0 .
        qed (use False in auto)
        hence inverse (2 ^ subdegree (G − F) :: real) ≤ inverse (2 ^ n)
          by (intro le-imp-inverse-le power-increasing) auto
        also have ... < e
          using n by (simp add: field-simps)
        finally show G ∈ A
          using e(2)[of G] by auto
      qed (use ‹F ∈ A› in auto)
    qed
  qed
next
  assume ∗: ∀F∈A. ∃n. {G. fps-cutoff n G = fps-cutoff n F} ⊆ A
  show open A
    unfolding open-fps-def
  proof safe
    fix F assume F: F ∈ A
    with ∗ obtain n where n: ⋀G. fps-cutoff n G = fps-cutoff n F ⟹ G ∈ A
      by blast
    show ∃r>0. {G. dist G F < r} ⊆ A
    proof (rule exI[of - 1 / 2 ^ n], safe)
```

```
    fix G assume dist: dist G F < 1 / 2 ^ n
    show G ∈ A
    proof (cases G = F)
      case False
      hence dist G F = inverse (2 ^ subdegree (F − G))
        by (simp add: dist-fps-def)
      with dist have n < subdegree (F − G)
        by (auto simp: field-simps)
      hence fps-nth (F − G) i = 0 if i ≤ n for i
        using that nth-less-subdegree-zero[of i F − G] by simp
      hence fps-cutoff n G = fps-cutoff n F
        by (auto simp: fps-eq-iff fps-cutoff-def)
      thus G ∈ A
        by (rule n)
    qed (use ‹F ∈ A› in auto)
  qed auto
 qed
qed

lemma open-fps-cutoff: open {H. fps-cutoff N H = fps-cutoff N G}
 unfolding open-fps-iff
proof safe
 fix F assume F: fps-cutoff N F = fps-cutoff N G
 show ∃ n. {G. fps-cutoff n G = fps-cutoff n F}
          ⊆ {H. fps-cutoff N H = fps-cutoff N G}
   by (rule exI[of - N]) (use F in ‹auto simp: fps-eq-iff›)
qed

lemma eventually-fps-nth-eq-nhds-fps-strong:
 eventually (λg. ∀ k≤n. fps-nth g k = fps-nth f k) (nhds f)
proof −
 have eventually (λg. g ∈ {g. fps-cutoff (n+1) g = fps-cutoff (n+1) f}) (nhds
f)
   by (rule eventually-nhds-in-open, rule open-fps-cutoff) auto
 thus ?thesis
   by eventually-elim (auto simp: fps-cutoff-eq-fps-cutoff-iff)
qed

lemma eventually-fps-nth-eq-nhds-fps: eventually (λg. fps-nth g k = fps-nth f k)
(nhds f)
 using eventually-fps-nth-eq-nhds-fps-strong[of k] by eventually-elim auto
```

A family of formal power series f_x tends to a limit series g at some filter F iff for any $N \geq 0$, the set of x for which f_x and G agree on the first N coefficients is in F.

For a sequence $(f_i)_{n\geq 0}$ this means that $f_i \longrightarrow G$ iff for any $N \geq 0$, f_x and G agree for all but finitely many x.

```
lemma tendsto-fps-iff:
 filterlim f (nhds (g :: 'a :: group-add fps)) F ⟷
```

```
    (∀ n. eventually (λx. fps-nth (f x) n = fps-nth g n) F)
proof safe
  assume lim: filterlim f (nhds (g :: 'a :: group-add fps)) F
  show eventually (λx. fps-nth (f x) n = fps-nth g n) F for n
  proof −
    define S where S = {H. fps-cutoff (n+1) H = fps-cutoff (n+1) g}
    have S: open S g ∈ S
      unfolding S-def using open-fps-cutoff[of n+1 g] by (auto simp: S-def)
    from lim and S have eventually (λx. f x ∈ S) F
      using topological-tendstoD by blast
    thus eventually (λx. fps-nth (f x) n = fps-nth g n) F
      by eventually-elim (auto simp: S-def fps-cutoff-eq-fps-cutoff-iff)
  qed
next
  assume ∗: ∀ n. eventually (λx. fps-nth (f x) n = fps-nth g n) F
  show filterlim f (nhds (g :: 'a :: group-add fps)) F
  proof (rule topological-tendstoI)
    fix S :: 'a fps set
    assume S: open S g ∈ S
    then obtain N where N: {H. fps-cutoff N H = fps-cutoff N g} ⊆ S
      unfolding open-fps-iff by blast
    have eventually (λx. ∀ n∈{..<N}. fps-nth (f x) n = fps-nth g n) F
      by (subst eventually-ball-finite-distrib) (use ∗ in auto)
    hence eventually (λx. f x ∈ {H. fps-cutoff N H = fps-cutoff N g}) F
      by eventually-elim (auto simp: fps-cutoff-eq-fps-cutoff-iff)
    thus eventually (λx. f x ∈ S) F
      by eventually-elim (use N in auto)
  qed
qed

lemma tendsto-fpsI:
  assumes ⋀n. eventually (λx. fps-nth (f x) n = fps-nth G n) F
  shows   filterlim f (nhds (G :: 'a :: group-add fps)) F
  unfolding tendsto-fps-iff using assms by blast
```

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 ∃ k>0. (1/y)^k < x
proof −
  have yp: y > 0
    using y1 by simp
  from reals-Archimedean2[of max 0 (− log y x) + 1]
  obtain k :: nat where k: real k > max 0 (− log y x) + 1
    by blast
  from k have kp: k > 0
    by simp
```

```
  from k have real k > - log y x
    by simp
  then have ln y * real k > - ln x
    unfolding log-def
    using ln-gt-zero-iff[OF yp] y1
    by (simp add: minus-divide-left field-simps del: minus-divide-left[symmetric])
  then have ln y * real k + ln x > 0
    by simp
  then have exp (real k * ln y + ln x) > exp 0
    by (simp add: ac-simps)
  then have y ^ k * x > 1
    unfolding exp-zero exp-add exp-of-nat-mult exp-ln [OF xp] exp-ln [OF yp]
    by simp
  then have x > (1 / y)^k using yp
    by (simp add: field-simps)
  then show ?thesis
    using kp by blast
qed

lemma fps-sum-rep-nth: (sum (λi. fps-const(a$i)*fps-X^i) {0..m})$n = (if n ≤
m then a$n else 0)
  by (simp add: fps-sum-nth if-distrib cong del: if-weak-cong)

lemma fps-notation: (λn. sum (λi. fps-const(a$i) * fps-X^i) {0..n}) ⟶ a
  (is ?s ⟶ a)
proof −
  have ∃n0. ∀n ≥ n0. dist (?s n) a < r if r > 0 for r
  proof −
    obtain n0 where n0: (1/2)^n0 < r n0 > 0
      using reals-power-lt-ex[OF ‹r > 0›, of 2] by auto
    show ?thesis
    proof −
      have dist (?s n) a < r if nn0: n ≥ n0 for n
      proof −
        from that have thnn0: (1/2)^n ≤ (1/2 :: real)^n0
          by (simp add: field-split-simps)
        show ?thesis
        proof (cases ?s n = a)
          case True
          then show ?thesis
            unfolding dist-eq-0-iff[of ?s n a, symmetric]
            using ‹r > 0› by (simp del: dist-eq-0-iff)
        next
          case False
          from False have dth: dist (?s n) a = (1/2)^subdegree (?s n − a)
            by (simp add: dist-fps-def field-simps)
          from False have kn: subdegree (?s n − a) > n
            by (intro subdegree-greaterI) (simp-all add: fps-sum-rep-nth)
          then have dist (?s n) a < (1/2)^n
```

by (*simp add*: *field-simps dist-fps-def*)
also have ... ≤ (*1*/*2*)^*n0*
using *nn0* **by** (*simp add*: *field-split-simps*)
also have ... < *r*
using *n0* **by** *simp*
finally show *?thesis* **.**
qed
qed
then show *?thesis* **by** *blast*
qed
qed
then show *?thesis*
unfolding *lim-sequentially* **by** *blast*
qed

5.7 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*
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*
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* = *0*,

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 = 0*, then div by zero will equal zero.

instance ..

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*
by *auto*

lemma *fps-inverse-0-iff'*: (*inverse f*) $ *0* = *0* ⟷ *inverse* (*f* $ *0*) = *0*
by (*simp add*: *fps-inverse-def fps-lr-inverse-0-iff*(*2*))

lemma *fps-inverse-0-iff*[*simp*]: (*inverse f*) $ *0* = (*0*::*'a*::*division-ring*) ⟷ *f* $ *0* = *0*
by (*simp add*: *fps-inverse-0-iff'*)

lemma *fps-lr-inverse-nth-0*:
(*fps-left-inverse f x*) $ *0* = *x* (*fps-right-inverse f x*) $ *0* = *x*
by *auto*

lemma *fps-inverse-nth-0* [*simp*]: (*inverse f*) $ *0* = *inverse* (*f* $ *0*)
by (*simp add*: *fps-inverse-def*)

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−
show *fps-left-inverse f 0* = *0*
proof (*rule fps-ext*)
fix *n* **show** *fps-left-inverse f 0* $ *n* = *0* $ *n*
by (*cases n*) (*simp-all add*: *fps-inverse-def*)
qed
show *fps-right-inverse g 0* = *0*
proof (*rule fps-ext*)
fix *n* **show** *fps-right-inverse g 0* $ *n* = *0* $ *n*
by (*cases n*) (*simp-all add*: *fps-inverse-def*)
qed
qed

lemma *fps-lr-inverse-eq0-imp-starting0*:

fps-left-inverse f x = 0 ⟹ x = 0
fps-right-inverse f x = 0 ⟹ x = 0
proof−
assume *A*: *fps-left-inverse f x = 0*
have *0 = fps-left-inverse f x $ 0* **by** (*subst A*) *simp*
thus *x = 0* **by** *simp*
next
assume *A*: *fps-right-inverse f x = 0*
have *0 = fps-right-inverse f x $ 0* **by** (*subst A*) *simp*
thus *x = 0* **by** *simp*
qed

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*
using *fps-lr-inverse-starting0 fps-lr-inverse-eq0-imp-starting0*
by *auto*

lemma *fps-inverse-eq-0-iff'*:
fixes *f* :: *'a*::{*ab-group-add*,*inverse*,*mult-zero*} *fps*
shows *inverse f = 0 ⟷ inverse* (*f $ 0*) *= 0*
by (*simp add*: *fps-inverse-def fps-lr-inverse-eq-0-iff*(*2*))

lemma *fps-inverse-eq-0-iff*[*simp*]: *inverse f* = (*0*:: (*'a*::*division-ring*) *fps*) ⟷ *f* $ *0 = 0*
using *fps-inverse-eq-0-iff'*[*of f*] **by** *simp*

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−
show *fps-left-inverse* (*fps-const a*) *x = fps-const x*
proof (*rule fps-ext*)
fix *n* **show** *fps-left-inverse* (*fps-const a*) *x $ n = fps-const x $ n*
by (*cases n*) *auto*
qed
show *fps-right-inverse* (*fps-const b*) *y = fps-const y*
proof (*rule fps-ext*)
fix *n* **show** *fps-right-inverse* (*fps-const b*) *y $ n = fps-const y $ n*
by (*cases n*) *auto*
qed
qed

lemma *fps-const-inverse*:
fixes $a :: 'a::\{comm\text{-}monoid\text{-}add,inverse,mult\text{-}zero,uminus\}$
shows $inverse\ (fps\text{-}const\ a) = fps\text{-}const\ (inverse\ a)$
unfolding *fps-inverse-def*
by (*simp add*: *fps-const-lr-inverse*(2))

lemma *fps-lr-inverse-zero*:
fixes $x :: 'a::\{ab\text{-}group\text{-}add,mult\text{-}zero\}$
and $y :: 'b::\{comm\text{-}monoid\text{-}add,mult\text{-}zero,uminus\}$
shows *fps-left-inverse 0 x* = *fps-const x*
and *fps-right-inverse 0 y* = *fps-const y*
using *fps-const-lr-inverse*[*of 0*]
by *simp-all*

lemma *fps-inverse-zero-conv-fps-const*:
$inverse\ (0::'a::\{comm\text{-}monoid\text{-}add,mult\text{-}zero,uminus,inverse\}\ fps) = fps\text{-}const\ (inverse\ 0)$
using *fps-lr-inverse-zero*(2)[*of inverse* (0::'*a*)] **by** (*simp add*: *fps-inverse-def*)

lemma *fps-inverse-zero'*:
assumes $inverse\ (0::'a::\{comm\text{-}monoid\text{-}add,inverse,mult\text{-}zero,uminus\}) = 0$
shows *inverse* (0::'*a fps*) = *0*
by (*simp add*: *assms fps-inverse-zero-conv-fps-const*)

lemma *fps-inverse-zero* [*simp*]:
$inverse\ (0::'a::division\text{-}ring\ fps) = 0$
by (*rule fps-inverse-zero'*[*OF inverse-zero*])

lemma *fps-lr-inverse-one*:
fixes $x :: 'a::\{ab\text{-}group\text{-}add,mult\text{-}zero,one\}$
and $y :: 'b::\{comm\text{-}monoid\text{-}add,mult\text{-}zero,uminus,one\}$
shows *fps-left-inverse 1 x* = *fps-const x*
and *fps-right-inverse 1 y* = *fps-const y*
using *fps-const-lr-inverse*[*of 1*]
by *simp-all*

lemma *fps-lr-inverse-one-one*:
$fps\text{-}left\text{-}inverse\ 1\ 1 = (1::'a::\{ab\text{-}group\text{-}add,mult\text{-}zero,one\}\ fps)$
$fps\text{-}right\text{-}inverse\ 1\ 1 = (1::'b::\{comm\text{-}monoid\text{-}add,mult\text{-}zero,uminus,one\}\ fps)$
by (*simp-all add*: *fps-lr-inverse-one*)

lemma *fps-inverse-one'*:
assumes $inverse\ (1::'a::\{comm\text{-}monoid\text{-}add,inverse,mult\text{-}zero,uminus,one\}) = 1$
shows *inverse* (*1* :: '*a fps*) = *1*
using *assms fps-lr-inverse-one-one*(2)
by (*simp add*: *fps-inverse-def*)

lemma *fps-inverse-one* [*simp*]: *inverse* (*1* :: '*a* :: *division-ring fps*) = *1*

```
  by (rule fps-inverse-one'[OF inverse-1])

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−

  show fps-left-inverse (−f) (−x) = − fps-left-inverse f x
  proof (intro fps-ext)
    fix n show fps-left-inverse (−f) (−x) $ n = − fps-left-inverse f x $ n
    proof (induct n rule: nat-less-induct)
      case (1 n) thus ?case by (cases n) (simp-all add: sum-negf algebra-simps)
    qed
  qed

  show fps-right-inverse (−f) (−x) = − fps-right-inverse f x
  proof (intro fps-ext)
    fix n show fps-right-inverse (−f) (−x) $ n = − fps-right-inverse f x $ n
    proof (induct n rule: nat-less-induct)
      case (1 n) show ?case
      proof (cases n)
        case (Suc m)
        with 1 have
          ∀ i∈{1..Suc m}. fps-right-inverse (−f) (−x) $ (Suc m − i) =
            − fps-right-inverse f x $ (Suc m − i)
          by auto
        with Suc show ?thesis by (simp add: sum-negf algebra-simps)
      qed simp
    qed
  qed

qed

lemma fps-inverse-minus [simp]: inverse (−f) = −inverse (f :: 'a :: division-ring
fps)
  by (simp add: fps-inverse-def fps-lr-inverse-minus(2))

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 (rule fps-ext)
  fix n show (fps-left-inverse f x * f) $ n = 1 $ n
    by (cases n) (simp-all add: f0 fps-mult-nth mult.assoc)
qed

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 (rule fps-ext)
  fix n
  show (f * fps-right-inverse f y) $ n = 1 $ n
  proof (cases n)
    case (Suc k)
    moreover from Suc have fps-right-inverse f y $ n =
          - y * sum (λi. f$i * fps-right-inverse-constructor f y (n - i)) {1..n}
      by simp
    hence
      (f * fps-right-inverse f y) $ n =
        - 1 * sum (λi. f$i * fps-right-inverse-constructor f y (n - i)) {1..n} +
        sum (λi. f$i * (fps-right-inverse-constructor f y (n - i))) {1..n}
     by (simp add: fps-mult-nth sum.atLeast-Suc-atMost mult.assoc f0[symmetric])
    thus (f * fps-right-inverse f y) $ n = 1 $ n by (simp add: Suc)
  qed (simp add: f0 fps-inverse-def)
qed
```

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−
  from f0(2) have f * fps-right-inverse f y = 1
     by (simp add: fps-right-inverse)
  hence fps-left-inverse f x * f * fps-right-inverse f y = fps-left-inverse f x
    by (simp add: mult.assoc)
  moreover from f0(1) have
    fps-left-inverse f x * f * fps-right-inverse f y = fps-right-inverse f y
    by (simp add: fps-left-inverse)
  ultimately show ?thesis by simp
qed

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
  using   assms fps-left-inverse-eq-fps-right-inverse[of x f x]
  by      (simp add: mult.commute)

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*
using *assms fps-left-inverse-eq-fps-right-inverse[of x f y] fps-left-inverse[of x f]*
by *simp*

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*
using *assms fps-left-inverse-eq-fps-right-inverse[of x f y] fps-right-inverse[of f y]*
by *simp*

lemma *inverse-mult-eq-1* [*intro*]:
assumes *f$0 ≠ (0::'a::division-ring)*
shows *inverse f * f = 1*
using *fps-left-inverse'[of inverse (f$0)]*
by (*simp add*: *assms fps-inverse-def*)

lemma *inverse-mult-eq-1'*:
assumes *f$0 ≠ (0::'a::division-ring)*
shows *f * inverse f = 1*
using *assms fps-right-inverse*
by (*force simp*: *fps-inverse-def*)

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−
have
*fps-left-inverse (unit-factor f) x * f =*
*fps-left-inverse (unit-factor f) x * unit-factor f * fps-X ^ subdegree f*
using *fps-unit-factor-decompose[of f]* **by** (*simp add*: *mult.assoc*)
with *assms* **show** *?thesis* **by** (*simp add*: *fps-left-inverse*)
qed

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−
have
*f * fps-right-inverse (unit-factor f) y =*
*fps-X ^ subdegree f * (unit-factor f * fps-right-inverse (unit-factor f) y)*
using *fps-unit-factor-decompose'[of f]* **by** (*simp add*: *mult.assoc[symmetric]*)
with *assms* **show** *?thesis* **by** (*simp add*: *fps-right-inverse*)
qed

lemma *fps-mult-right-inverse-unit-factor-divring*:

(*f* :: 'a::*division-ring fps*) ≠ *0* ⟹ *f* * *inverse* (*unit-factor f*) = *fps-X* ^ *subdegree f*
using *fps-mult-right-inverse-unit-factor*[*of f*]
by (*simp add*: *fps-inverse-def*)

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−
from *assms*(*1*) **have**
fps-left-inverse (*fps-left-inverse f x*) *y* * *fps-left-inverse f x* * *f* =
fps-left-inverse (*fps-left-inverse f x*) *y*
by (*simp add*: *fps-left-inverse mult.assoc*)
moreover from *assms*(*2*) **have**
fps-left-inverse (*fps-left-inverse f x*) *y* * *fps-left-inverse f x* = *1*
by (*simp add*: *fps-left-inverse*)
ultimately show *?thesis* **by** *simp*
qed

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*
using *assms fps-left-inverse-idempotent-ring1*[*of x f f*$*0*]
by (*simp add*: *mult.commute*)

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−
from *assms*(*1*) **have** *f* * (*fps-right-inverse f x* * *fps-right-inverse* (*fps-right-inverse f x*) *y*) =
fps-right-inverse (*fps-right-inverse f x*) *y*
by (*simp add*: *fps-right-inverse mult.assoc*[*symmetric*])
moreover from *assms*(*2*) **have**
fps-right-inverse f x * *fps-right-inverse* (*fps-right-inverse f x*) *y* = *1*
by (*simp add*: *fps-right-inverse*)
ultimately show *?thesis* **by** *simp*
qed

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*
using *assms fps-right-inverse-idempotent-ring1*[*of f x f*$*0*]

```
  by      (simp add: mult.commute)

lemma fps-inverse-idempotent[intro, simp]:
  f$0 ≠ (0::'a::division-ring) ⟹ inverse (inverse f) = f
  using fps-right-inverse-idempotent-ring1[of f]
  by    (simp add: fps-inverse-def)

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−

  show fps-left-inverse g (f$0) = f
  proof (intro fps-ext)
    fix n show fps-left-inverse g (f$0) $ n = f $ n
    proof (induct n rule: nat-less-induct)
      case (1 n) show ?case
      proof (cases n)
        case (Suc k)
        hence ∀ i∈{0..k}. fps-left-inverse g (f$0) $ i = f $ i using 1 by simp
        hence fps-left-inverse g (f$0) $ Suc k = f $ Suc k − 1 $ Suc k * f$0
         by (simp add: fps-mult-nth fg(1)[symmetric] distrib-right mult.assoc fg(2))
        with Suc show ?thesis by simp
      qed simp
    qed
  qed

  show fps-right-inverse f (g$0) = g
  proof (intro fps-ext)
    fix n show fps-right-inverse f (g$0) $ n = g $ n
    proof (induct n rule: nat-less-induct)
      case (1 n) show ?case
      proof (cases n)
        case (Suc k)
        hence ∀ i∈{1..Suc k}. fps-right-inverse f (g$0) $ (Suc k − i) = g $ (Suc k
− i)
          using 1 by auto
        hence
          fps-right-inverse f (g$0) $ Suc k = 1 * g $ Suc k − g$0 * 1 $ Suc k
        by (simp add: fps-mult-nth fg(1)[symmetric] algebra-simps fg(2)[symmetric]
sum.atLeast-Suc-atMost)
        with Suc show ?thesis by simp
      qed simp
    qed
  qed

qed
```

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−
from *fg* **have** *f*$*0* ∗ *g*$*0* = *1* **using** *fps-mult-nth-0*[*of f g*] **by** *simp*
hence *g*$*0* ∗ *f*$*0* = *1* **using** *inverse-unique*[*of f*$*0*] *left-inverse*[*of f*$*0*] **by** *force*
thus *fps-left-inverse g* (*f*$*0*) = *f fps-right-inverse f* (*g*$*0*) = *g*
using *fg fps-lr-inverse-unique-ring1* **by** *auto*
qed

lemma *fps-inverse-unique*:
fixes *f g* :: *'a* :: *division-ring fps*
assumes *fg*: *f* ∗ *g* = *1*
shows *inverse f* = *g*
proof −
from *fg* **have** *if0*: *inverse* (*f*$*0*) = *g*$*0 f*$*0* ≠ *0*
using *inverse-unique*[*of f*$*0*] *fps-mult-nth-0*[*of f g*] **by** *auto*
with *fg* **have** *fps-right-inverse f* (*g*$*0*) = *g*
using *left-inverse*[*of f*$*0*] **by** (*intro fps-lr-inverse-unique-ring1*(*2*)) *simp-all*
with *if0*(*1*) **show** *?thesis* **by** (*simp add*: *fps-inverse-def*)
qed

lemma *inverse-fps-numeral*:
inverse (*numeral n* :: (*'a* :: *field-char-0*) *fps*) = *fps-const* (*inverse* (*numeral n*))
by (*intro fps-inverse-unique fps-ext*) (*simp-all add*: *fps-numeral-nth*)

lemma *inverse-fps-of-nat*:
inverse (*of-nat n* :: *'a* :: {*semiring-1*,*times*,*uminus*,*inverse*} *fps*) =
fps-const (*inverse* (*of-nat n*))
by (*simp add*: *fps-of-nat fps-const-inverse*[*symmetric*])

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 −
define *h* **where** *h* ≡ *fps-left-inverse g y* ∗ *fps-left-inverse f x*
hence *h0*: *h*$*0* = *y*∗*x* **by** *simp*
have *fps-left-inverse* (*f*∗*g*) (*h*$*0*) = *h*
proof (*intro fps-lr-inverse-unique-ring1*(*1*))
from *h-def*
have *h* ∗ (*f* ∗ *g*) = *fps-left-inverse g y* ∗ (*fps-left-inverse f x* ∗ *f*) ∗ *g*
by (*simp add*: *mult.assoc*)

```
    thus h * (f * g) = 1
      using fps-left-inverse[OF x(1)] fps-left-inverse[OF y(1)] by simp
    from h-def have (f*g)$0 * h$0 = f$0 * 1 * x
      by (simp add: mult.assoc y(2)[symmetric])
    with x(2) show (f * g) $ 0 * h $ 0 = 1 by simp
  qed
  with h-def
    show  fps-left-inverse (f * g) (y*x) = fps-left-inverse g y * fps-left-inverse f x
    by    simp
next
  define h where h ≡ fps-right-inverse g y * fps-right-inverse f x
  hence h0: h$0 = y*x by simp
  have fps-right-inverse (f*g) (h$0) = h
  proof (intro fps-lr-inverse-unique-ring1(2))
    from h-def
      have  f * g * h = f * (g * fps-right-inverse g y) * fps-right-inverse f x
      by    (simp add: mult.assoc)
    thus f * g * h = 1
      using fps-right-inverse[OF x(2)] fps-right-inverse[OF y(2)] by simp
    from h-def have h$0 * (f*g)$0 = y * 1 * g$0
      by (simp add: mult.assoc x(1)[symmetric])
    with y(1) show h$0 * (f*g)$0  = 1 by simp
  qed
  with h-def
    show  fps-right-inverse (f * g) (y*x) = fps-right-inverse g y * fps-right-inverse
f x
    by    simp
qed

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−
  show fps-left-inverse (f * g) (inverse ((f*g)$0)) =
        fps-left-inverse g (inverse (g$0)) * fps-left-inverse f (inverse (f$0))
  proof (cases f$0 = 0 ∨ g$0 = 0)
    case True
    hence fps-left-inverse (f * g) (inverse ((f*g)$0)) = 0
      by (simp add: fps-lr-inverse-eq-0-iff(1))
    moreover from True have
      fps-left-inverse g (inverse (g$0)) * fps-left-inverse f (inverse (f$0)) = 0
      by (auto simp: fps-lr-inverse-eq-0-iff(1))
    ultimately show ?thesis by simp
  next
    case False
    hence fps-left-inverse (f * g) (inverse (g$0) * inverse (f$0)) =
```

```
        fps-left-inverse g (inverse (g$0)) * fps-left-inverse f (inverse (f$0))
    by (intro fps-lr-inverse-mult-ring1(1)) simp-all
   with False show ?thesis by (simp add: nonzero-inverse-mult-distrib)
  qed
  show fps-right-inverse (f * g) (inverse ((f*g)$0)) =
       fps-right-inverse g (inverse (g$0)) * fps-right-inverse f (inverse (f$0))
  proof (cases f$0 = 0 ∨ g$0 = 0)
   case True
   from True have fps-right-inverse (f * g) (inverse ((f*g)$0)) = 0
    by (simp add: fps-lr-inverse-eq-0-iff(2))
   moreover from True have
    fps-right-inverse g (inverse (g$0)) * fps-right-inverse f (inverse (f$0)) = 0
    by (auto simp: fps-lr-inverse-eq-0-iff(2))
   ultimately show ?thesis by simp
  next
   case False
   hence fps-right-inverse (f * g) (inverse (g$0) * inverse (f$0)) =
         fps-right-inverse g (inverse (g$0)) * fps-right-inverse f (inverse (f$0))
    by (intro fps-lr-inverse-mult-ring1(2)) simp-all
   with False show ?thesis by (simp add: nonzero-inverse-mult-distrib)
  qed
qed

lemma fps-inverse-mult-divring:
  inverse (f * g) = inverse g * inverse (f :: 'a::division-ring fps)
  using fps-lr-inverse-mult-divring(2) by (simp add: fps-inverse-def)

lemma fps-inverse-mult: inverse (f * g :: 'a::field fps) = inverse f * inverse g
  by (simp add: fps-inverse-mult-divring)

lemma inverse-prod-fps: inverse (prod f A) = (∏x∈A. inverse (f x) :: 'a :: field
fps)
  by (induction A rule: infinite-finite-induct) (auto simp: fps-inverse-mult)

lemma fps-lr-inverse-gp-ring1:
  fixes   ones ones-inv :: 'a :: ring-1 fps
  defines ones ≡ Abs-fps (λn. 1)
  and     ones-inv ≡ Abs-fps (λn. 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−
  show fps-left-inverse ones 1 = ones-inv
  proof (rule fps-ext)
   fix n
   show fps-left-inverse ones 1 $ n = ones-inv $ n
   proof (induct n rule: nat-less-induct)
    case (1 n) show ?case
    proof (cases n)
     case (Suc m)
```

```
      have m: n = Suc m by fact
      moreover have fps-left-inverse ones 1 $ Suc m = ones-inv $ Suc m
      proof (cases m)
        case (Suc k) thus ?thesis
       using Suc m 1 by (simp add: ones-def ones-inv-def sum.atLeast-Suc-atMost)
      qed (simp add: ones-def ones-inv-def)
      ultimately show ?thesis by simp
    qed (simp add: ones-inv-def)
  qed
  moreover have fps-right-inverse ones 1 = fps-left-inverse ones 1
    by (auto intro: fps-left-inverse-eq-fps-right-inverse[symmetric] simp: ones-def)
  ultimately show fps-right-inverse ones 1 = ones-inv by simp
qed

lemma fps-lr-inverse-gp-ring1':
  fixes   ones :: 'a :: ring-1 fps
  defines ones ≡ Abs-fps (λn. 1)
  shows   fps-left-inverse ones 1 = 1 − fps-X
  and     fps-right-inverse ones 1 = 1 − fps-X
proof−
  define ones-inv :: 'a :: ring-1 fps
    where ones-inv ≡ Abs-fps (λn. if n=0 then 1 else if n=1 then − 1 else 0)
  hence fps-left-inverse ones 1 = ones-inv
  and   fps-right-inverse ones 1 = ones-inv
    using ones-def fps-lr-inverse-gp-ring1 by auto
  thus fps-left-inverse ones 1 = 1 − fps-X
  and   fps-right-inverse ones 1 = 1 − fps-X
    by (auto intro: fps-ext simp: ones-inv-def)
qed

lemma fps-inverse-gp:
  inverse (Abs-fps(λn. (1::'a::division-ring))) =
    Abs-fps (λn. if n= 0 then 1 else if n=1 then − 1 else 0)
  using fps-lr-inverse-gp-ring1(2) by (simp add: fps-inverse-def)

lemma fps-inverse-gp': inverse (Abs-fps (λn. 1::'a::division-ring)) = 1 − fps-X
  by (simp add: fps-inverse-def fps-lr-inverse-gp-ring1'(2))

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−
  have fps-left-inverse ones 1 = 1 − fps-X
    using fps-lr-inverse-gp-ring1'(1) by (simp add: ones-def)
  thus fps-left-inverse (1 − fps-X) 1 = ones
    using fps-left-inverse-idempotent-ring1[of 1 ones 1] by (simp add: ones-def)
```

```
  have fps-right-inverse ones 1 = 1 − fps-X
    using fps-lr-inverse-gp-ring1'(2) by (simp add: ones-def)
  thus fps-right-inverse (1 − fps-X) 1 = ones
    using fps-right-inverse-idempotent-ring1[of ones 1 1] by (simp add: ones-def)
qed

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
  by      (simp add: fps-inverse-def assms fps-lr-inverse-one-minus-fps-X(2))

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−
  define  f invf2 :: 'a fps
    where f ≡ (1 − fps-X)
    and   invf2 ≡ Abs-fps (λn. of-nat (n+1))

  have f2-nth-simps:
    f^2 $ 1 = − of-nat 2 f^2 $ 2 = 1 ⋀n. n>2 ⟹ f^2 $ n = 0
     by (simp-all add: power2-eq-square f-def fps-mult-nth sum.atLeast-Suc-atMost)

  show fps-left-inverse (f^2) 1 = invf2
  proof (intro fps-ext)
    fix n show fps-left-inverse (f^2) 1 $ n = invf2 $ n
    proof (induct n rule: nat-less-induct)
      case (1 t)
      hence induct-assm:
        ⋀m. m < t ⟹ fps-left-inverse (f^2) 1 $ m = invf2 $ m
        by fast
      show ?case
      proof (cases t)
        case (Suc m)
        have m: t = Suc m by fact
        moreover have fps-left-inverse (f^2) 1 $ Suc m = invf2 $ Suc m
        proof (cases m)
          case 0 thus ?thesis using f2-nth-simps(1) by (simp add: invf2-def)
        next
          case (Suc l)
          have l: m = Suc l by fact
         moreover have fps-left-inverse (f^2) 1 $ Suc (Suc l) = invf2 $ Suc (Suc
l)
          proof (cases l)
        case 0 thus ?thesis using f2-nth-simps(1,2) by (simp add: Suc-1[symmetric]
invf2-def)
          next
```

```
        case (Suc k)
        from Suc l m
          have A: fps-left-inverse (f^2) 1 $ Suc (Suc k) = invf2 $ Suc (Suc k)
          and  B: fps-left-inverse (f^2) 1 $ Suc k = invf2 $ Suc k
          using induct-assm[of Suc k] induct-assm[of Suc (Suc k)]
          by    auto
        have times2: ⋀a::nat. 2*a = a + a by simp
        have ∀ i∈{0..k}. (f^2)$(Suc (Suc (Suc k)) - i) = 0
          using f2-nth-simps(3) by auto
        hence
          fps-left-inverse (f^2) 1 $ Suc (Suc (Suc k)) =
            fps-left-inverse (f^2) 1 $ Suc (Suc k) * of-nat 2 -
            fps-left-inverse (f^2) 1 $ Suc k
          using sum.ub-add-nat f2-nth-simps(1,2) by simp
        also have ... = of-nat (2 * Suc (Suc (Suc k))) - of-nat (Suc (Suc k))
          by (subst A, subst B) (simp add: invf2-def mult.commute)
        also have ... = of-nat (Suc (Suc (Suc k)) + 1)
          by (subst times2[of Suc (Suc (Suc k))]) simp
        finally have
          fps-left-inverse (f^2) 1 $ Suc (Suc (Suc k)) = invf2 $ Suc (Suc (Suc
k))
          by (simp add: invf2-def)
        with Suc show ?thesis by simp
      qed
      ultimately show ?thesis by simp
    qed
    ultimately show ?thesis by simp
  qed (simp add: invf2-def)
 qed
qed

 moreover have fps-right-inverse (f^2) 1 = fps-left-inverse (f^2) 1
  by  (auto
       intro: fps-left-inverse-eq-fps-right-inverse[symmetric]
       simp: f-def power2-eq-square
      )
 ultimately show fps-right-inverse (f^2) 1 = invf2
  by simp

qed

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))
 using   assms fps-lr-one-over-one-minus-fps-X-squared(2)
 by      (simp add: fps-inverse-def power2-eq-square)

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))
```

by (*rule fps-one-over-one-minus-fps-X-squared'*[*OF inverse-1*])

lemma *fps-lr-inverse-fps-X-plus1*:
fps-left-inverse (*1* + *fps-X*) (*1*::*'a*::*ring-1*) = *Abs-fps* (λ*n*. (−*1*)^*n*)
fps-right-inverse (*1* + *fps-X*) (*1*::*'a*) = *Abs-fps* (λ*n*. (−*1*)^*n*)
proof−

show *fps-left-inverse* (*1* + *fps-X*) (*1*::*'a*) = *Abs-fps* (λ*n*. (−*1*)^*n*)
proof (*rule fps-ext*)
fix *n* **show** *fps-left-inverse* (*1* + *fps-X*) (*1*::*'a*) $ *n* = *Abs-fps* (λ*n*. (−*1*)^*n*) $ *n*
proof (*induct n rule*: *nat-less-induct*)
case (*1 n*) **show** *?case*
proof (*cases n*)
case (*Suc m*)
have *m*: *n* = *Suc m* **by** *fact*
from *Suc 1* **have**
A: *fps-left-inverse* (*1* + *fps-X*) (*1*::*'a*) $ *n* =
− (∑ *i*=*0*..*m*. (− *1*)^*i* ∗ (*1* + *fps-X*) $ (*Suc m* − *i*))
by *simp*
show *?thesis*
proof (*cases m*)
case (*Suc l*)
have ∀ *i*∈{*0*..*l*}. ((*1*::*'a fps*) + *fps-X*) $ (*Suc* (*Suc l*) − *i*) = *0* **by** *auto*
with *Suc A m* **show** *?thesis* **by** *simp*
qed (*simp add*: *m A*)
qed *simp*
qed

moreover have
fps-right-inverse (*1* + *fps-X*) (*1*::*'a*) = *fps-left-inverse* (*1* + *fps-X*) *1*
by (*intro fps-left-inverse-eq-fps-right-inverse*[*symmetric*]) *simp-all*
ultimately show *fps-right-inverse* (*1* + *fps-X*) (*1*::*'a*) = *Abs-fps* (λ*n*. (−*1*)^*n*)
by *simp*

qed

lemma *fps-inverse-fps-X-plus1'*:
assumes *inverse* (*1*::*'a*::{*ring-1*,*inverse*}) = *1*
shows *inverse* (*1* + *fps-X*) = *Abs-fps* (λ*n*. (− (*1*::*'a*)) ^ *n*)
using *assms fps-lr-inverse-fps-X-plus1*(*2*)
by (*simp add*: *fps-inverse-def*)

lemma *fps-inverse-fps-X-plus1*:
inverse (*1* + *fps-X*) = *Abs-fps* (λ*n*. (− (*1*::*'a*::*division-ring*)) ^ *n*)
by (*rule fps-inverse-fps-X-plus1'*[*OF inverse-1*])

lemma *subdegree-lr-inverse*:

fixes $x :: 'a::\{comm\text{-}monoid\text{-}add,mult\text{-}zero,uminus\}$
and $y :: 'b::\{ab\text{-}group\text{-}add,mult\text{-}zero\}$
shows *subdegree* (*fps-left-inverse f x*) = *0*
and *subdegree* (*fps-right-inverse g y*) = *0*
proof−
show *subdegree* (*fps-left-inverse f x*) = *0*
using *fps-lr-inverse-eq-0-iff*(*1*) *subdegree-eq-0-iff* **by** *fastforce*
show *subdegree* (*fps-right-inverse g y*) = *0*
using *fps-lr-inverse-eq-0-iff*(*2*) *subdegree-eq-0-iff* **by** *fastforce*
qed

lemma *subdegree-inverse* [*simp*]:
fixes $f :: 'a::\{ab\text{-}group\text{-}add,inverse,mult\text{-}zero\}$ *fps*
shows *subdegree* (*inverse f*) = *0*
using *subdegree-lr-inverse*(*2*)
by (*simp add*: *fps-inverse-def*)

lemma *fps-right-inverse-constructor-rec*:
n > *0* ⟹ *fps-right-inverse-constructor f a n* =
−*a* ∗ *sum* (λ*i*. *fps-nth f i* ∗ *fps-right-inverse-constructor f a* (*n* − *i*))
{*1*..*n*}
by (*cases n*) *auto*

lemma *fps-right-inverse-constructor-cong*:
assumes ⋀*k*. *k* ≤ *n* ⟹ *fps-nth f k* = *fps-nth g k*
shows *fps-right-inverse-constructor f c n* = *fps-right-inverse-constructor g c n*
using *assms*
proof (*induction n rule*: *less-induct*)
case (*less n*)
show *?case*
proof (*cases n* > *0*)
case *n*: *True*
have *fps-right-inverse-constructor f c n* =
−*c* ∗ *sum* (λ*i*. *fps-nth f i* ∗ *fps-right-inverse-constructor f c* (*n* − *i*))
{*1*..*n*}
by (*subst fps-right-inverse-constructor-rec*) (*use n* **in** *auto*)
also have *sum* (λ*i*. *fps-nth f i* ∗ *fps-right-inverse-constructor f c* (*n* − *i*)) {*1*..*n*}
=
sum (λ*i*. *fps-nth g i* ∗ *fps-right-inverse-constructor g c* (*n* − *i*)) {*1*..*n*}
by (*intro sum.cong refl arg-cong2*[*of* - - - - (∗)] *less*) (*use assms* **in** *auto*)
also have −*c* ∗ ... = *fps-right-inverse-constructor g c n*
by (*subst* (*2*) *fps-right-inverse-constructor-rec*) (*use n* **in** *auto*)
finally show *?thesis* **.**
qed *auto*
qed

lemma *fps-cutoff-inverse*:
fixes $f :: 'a :: field\ fps$
assumes *fps-nth f 0* ≠ *0*

```
  shows   fps-cutoff n (inverse (fps-cutoff n f)) = fps-cutoff n (inverse f)
proof (cases n = 0)
  case True
  show ?thesis
    by (simp add: True)
next
  case False
  show ?thesis
  proof (subst fps-cutoff-eq-fps-cutoff-iff, safe)
    fix k assume k < n
    have fps-nth (inverse (fps-cutoff n f)) k =
          fps-right-inverse-constructor (fps-cutoff n f) (inverse (fps-nth f 0)) k
      using False by (simp add: fps-inverse-def)
    also have ... = fps-right-inverse-constructor f (inverse (fps-nth f 0)) k
      by (rule fps-right-inverse-constructor-cong) (use ‹k < n› in auto)
    also have ... = fps-nth (inverse f) k
      using False by (simp add: fps-inverse-def)
    finally show fps-nth (inverse (fps-cutoff n f)) k = fps-nth (inverse f) k .
  qed
qed

lemma tendsto-inverse-fps-aux:
  fixes f :: 'a :: field fps
  assumes fps-nth f 0 ≠ 0
  shows   ((λf. inverse f) ⟶ inverse f) (at f)
  unfolding tendsto-fps-iff
proof
  fix n :: nat
  have eventually (λg. ∀k≤n. fps-nth g k = fps-nth f k) (nhds f)
    by (rule eventually-fps-nth-eq-nhds-fps-strong)
  hence eventually (λg. ∀k≤n. fps-nth g k = fps-nth f k) (at f)
    using eventually-nhds-conv-at by blast
  thus eventually (λg. fps-nth (inverse g) n = fps-nth (inverse f) n) (at f)
  proof eventually-elim
    case (elim g)
    from elim have fps-nth g 0 = fps-nth f 0
      by auto
    with assms have [simp]: fps-nth g 0 ≠ 0
      by simp
   have fps-cutoff (n+1) (inverse f) = fps-cutoff (n+1) (inverse (fps-cutoff (n+1)
f))
      by (rule fps-cutoff-inverse [symmetric]) fact
    also have fps-cutoff (n+1) f = fps-cutoff (n+1) g
      by (subst fps-cutoff-eq-fps-cutoff-iff) (use elim in auto)
    also have fps-cutoff (n+1) (inverse ...) = fps-cutoff (n+1) (inverse g)
      by (rule fps-cutoff-inverse) auto
    finally show ?case
      by (subst (asm) fps-cutoff-eq-fps-cutoff-iff) auto
  qed
```

qed

lemma *tendsto-inverse-fps* [*tendsto-intros*]:
fixes $g :: 'a :: field\ fps$
assumes $(f \longrightarrow g)\ F$
assumes $fps\text{-}nth\ g\ 0 \neq 0$
shows $((\lambda x.\ inverse\ (f\ x)) \longrightarrow inverse\ g)\ F$
by (*rule tendsto-compose*[*OF tendsto-inverse-fps-aux assms*(*1*)]) *fact*

lemma *fps-div-zero* [*simp*]:
$0\ div\ (g :: 'a :: \{comm\text{-}monoid\text{-}add,inverse,mult\text{-}zero,uminus\}\ fps) = 0$
by (*simp add*: *fps-divide-def*)

lemma *fps-div-by-zero'*:
fixes $g :: 'a::\{comm\text{-}monoid\text{-}add,inverse,mult\text{-}zero,uminus\}\ fps$
assumes $inverse\ (0::'a) = 0$
shows $g\ div\ 0 = 0$
by (*simp add*: *fps-divide-def assms fps-inverse-zero'*)

lemma *fps-div-by-zero* [*simp*]: $(g::'a::division\text{-}ring\ fps)\ div\ 0 = 0$
by (*rule fps-div-by-zero'*[*OF inverse-zero*])

lemma *fps-divide-unit'*: $subdegree\ g = 0 \Longrightarrow f\ div\ g = f * inverse\ g$
by (*simp add*: *fps-divide-def*)

lemma *fps-divide-unit*: $g\$0 \neq 0 \Longrightarrow f\ div\ g = f * inverse\ g$
by (*intro fps-divide-unit'*) (*simp add*: *subdegree-eq-0-iff*)

lemma *fps-divide-nth-0'*:
$subdegree\ (g::'a::division\text{-}ring\ fps) = 0 \Longrightarrow (f\ div\ g)\ \$\ 0 = f\ \$\ 0\ /\ (g\ \$\ 0)$
by (*simp add*: *fps-divide-unit' divide-inverse*)

lemma *fps-divide-nth-0* [*simp*]:
$g\ \$\ 0 \neq 0 \Longrightarrow (f\ div\ g)\ \$\ 0 = f\ \$\ 0\ /\ (g\ \$\ 0 :: - :: division\text{-}ring)$
by (*simp add*: *fps-divide-nth-0'*)

lemma *fps-divide-nth-below*:
fixes $f\ g :: 'a::\{comm\text{-}monoid\text{-}add,uminus,mult\text{-}zero,inverse\}\ fps$
shows $n < subdegree\ f - subdegree\ g \Longrightarrow (f\ div\ g)\ \$\ n = 0$
by (*simp add*: *fps-divide-def fps-mult-nth-eq0*)

lemma *fps-divide-nth-base*:
fixes $f\ g :: 'a::division\text{-}ring\ fps$
assumes $subdegree\ g \leq subdegree\ f$
shows $(f\ div\ g)\ \$\ (subdegree\ f - subdegree\ g) = f\ \$\ subdegree\ f * inverse\ (g\ \$\ subdegree\ g)$
by (*simp add*: *assms fps-divide-def fps-divide-unit'*)

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*
by (*intro subdegree-geI*) (*simp-all add*: *assms fps-divide-nth-below*)

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 (*intro antisym*)
from *assms* **have** *1*: (*f div g*) $ (*subdegree f* − *subdegree g*) ≠ *0*
using *fps-divide-nth-base*[*of g f*] **by** *simp*
thus *subdegree* (*f / g*) ≤ *subdegree f* − *subdegree g* **by** (*intro subdegree-leI*) *simp*
from *1* **have** *f / g* ≠ *0* **by** (*auto intro*: *fps-nonzeroI*)
thus *subdegree f* − *subdegree g* ≤ *subdegree* (*f / g*) **by** (*rule fps-divide-subdegree-ge*)
qed

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*)
using *assms fps-shift-mult-right-noncomm*[*of n f inverse* (*unit-factor g*)]
fps-shift-fps-shift-reorder[*of subdegree g n f* ∗ *inverse* (*unit-factor g*)]
by (*simp add*: *fps-divide-def*)

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 (*intro fps-ext*)
fix *k*
from *assms*(*1*) **have** *LHS*:
(*f / fps-shift n g*) $ *k* = (*f* ∗ *inverse* (*unit-factor g*)) $ (*k* + (*subdegree g* − *n*))
using *fps-unit-factor-shift*[*of n g*]
by (*simp add*: *fps-divide-def*)
show (*f / fps-shift n g*) $ *k* = *Abs-fps* (λ*k*. *if k*<*n then 0 else* (*f*/*g*) $ (*k*−*n*)) $ *k*
proof (*cases k*<*n*)
case *True* **with** *assms LHS* **show** *?thesis* **using** *fps-mult-nth-eq0*[*of - f*] **by** *simp*
next
case *False*
hence (*f*/*g*) $ (*k*−*n*) = (*f* ∗ *inverse* (*unit-factor g*)) $ ((*k*−*n*) + *subdegree g*)
by (*simp add*: *fps-divide-def*)
with *False LHS assms*(*1*) **show** *?thesis* **by** *auto*
qed
qed

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)*
by *(simp add: fps-divide-shift-numer)*

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 (λk. if k<subdegree g then 0 else (f/g) $ (k−subdegree g))
by *(simp add: assms fps-divide-shift-denom)*

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)*
using *assms fps-divide-unit-factor-numer[of f unit-factor g]*
fps-divide-unit-factor-denom[of g f]
fps-shift-rev-shift(1)[of subdegree g subdegree f f/g]
by *simp*

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)*
using *assms fps-divide-unit-factor-both'[of g f] fps-divide-subdegree[of f g]*
by *(cases f=0 ∨ g=0) auto*

lemma *fps-divide-self*:
(f::'a::division-ring fps) ≠ 0 ⟹ f / f = 1
using *fps-mult-right-inverse-unit-factor-divring[of f]*
by *(simp add: fps-divide-def)*

lemma *fps-divide-add*:
fixes *f g h :: 'a::{semiring-0,inverse,uminus} fps*
shows *(f + g) / h = f / h + g / h*
by *(simp add: fps-divide-def algebra-simps fps-shift-add)*

lemma *fps-divide-diff*:
fixes *f g h :: 'a::{ring,inverse} fps*
shows *(f − g) / h = f / h − g / h*
by *(simp add: fps-divide-def algebra-simps fps-shift-diff)*

lemma *fps-divide-uminus*:
fixes *f g h :: 'a::{ring,inverse} fps*
shows *(− f) / g = − (f / g)*
by *(simp add: fps-divide-def algebra-simps fps-shift-uminus)*

lemma *fps-divide-uminus'*:

fixes *f g h* :: *'a*::*division-ring fps*
shows *f* / (− *g*) = − (*f* / *g*)
by (*simp add*: *fps-divide-def fps-unit-factor-uminus fps-shift-uminus*)

lemma *fps-divide-times*:
fixes *f g h* :: *'a*::{*semiring-0*,*inverse*,*uminus*} *fps*
assumes *subdegree h* ≤ *subdegree g*
shows (*f* ∗ *g*) / *h* = *f* ∗ (*g* / *h*)
using *assms fps-mult-subdegree-ge*[*of g inverse* (*unit-factor h*)]
fps-shift-mult[*of subdegree h g* ∗ *inverse* (*unit-factor h*) *f*]
by (*fastforce simp add*: *fps-divide-def mult.assoc*)

lemma *fps-divide-times2*:
fixes *f g h* :: *'a*::{*comm-semiring-0*,*inverse*,*uminus*} *fps*
assumes *subdegree h* ≤ *subdegree f*
shows (*f* ∗ *g*) / *h* = (*f* / *h*) ∗ *g*
using *assms fps-divide-times*[*of h f g*]
by (*simp add*: *mult.commute*)

lemma *fps-times-divide-eq*:
fixes *f g* :: *'a*::*field fps*
assumes *g* ≠ *0* **and** *subdegree f* ≥ *subdegree g*
shows *f div g* ∗ *g* = *f*
using *assms fps-divide-times2*[*of g f g*]
by (*simp add*: *fps-divide-times fps-divide-self*)

lemma *fps-divide-times-eq*:
(*g* :: *'a*::*division-ring fps*) ≠ *0* ⟹ (*f* ∗ *g*) *div g* = *f*
by (*simp add*: *fps-divide-times fps-divide-self*)

lemma *fps-divide-by-mult'*:
fixes *f g h* :: *'a* :: *division-ring fps*
assumes *subdegree h* ≤ *subdegree f*
shows *f* / (*g* ∗ *h*) = *f* / *h* / *g*
proof (*cases f*=*0* ∨ *g*=*0* ∨ *h*=*0*)
case *False* **with** *assms* **show** *?thesis*
using *fps-unit-factor-mult*[*of g h*]
by (*auto simp*:
fps-divide-def fps-shift-fps-shift fps-inverse-mult-divring mult.assoc
fps-shift-mult-right-noncomm
)
qed *auto*

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−
have *f* / (*g* ∗ *h*) = *f* / (*h* ∗ *g*) **by** (*simp add*: *mult.commute*)

also have ... = *f* / *g* / *h* **using** *fps-divide-by-mult'*[*OF assms*] **by** *simp*
finally show *?thesis* **by** *simp*
qed

lemma *fps-divide-cancel*:
fixes *f g h* :: *'a* :: *division-ring fps*
shows *h* ≠ *0* ⟹ (*f* ∗ *h*) *div* (*g* ∗ *h*) = *f div g*
by (*cases f=0*)
(*auto simp*: *fps-divide-by-mult' fps-divide-times-eq*)

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*
using *assms fps-inverse-one' fps-one-mult*(*2*)[*of a*]
by (*force simp*: *fps-divide-def*)

lemma *fps-divide-1* [*simp*]: (*a* :: *'a*::*division-ring fps*) / *1* = *a*
by (*rule fps-divide-1'*[*OF inverse-1*])

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*
using *assms fps-one-mult*(*2*)[*of f*]
by (*simp add*: *fps-divide-def fps-X-unit-factor fps-inverse-one'*)

lemma *fps-divide-X* [*simp*]: *a* / *fps-X* = *fps-shift 1* (*a*::*'a*::*division-ring fps*)
by (*rule fps-divide-X'*[*OF inverse-1*])

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*
using *fps-inverse-one'*[*OF assms*] *fps-one-mult*(*2*)[*of f*]
by (*simp add*: *fps-divide-def fps-X-power-subdegree*)

lemma *fps-divide-X-power* [*simp*]: *a* / (*fps-X* ^ *n*) = *fps-shift n* (*a*::*'a*::*division-ring fps*)
by (*rule fps-divide-X-power'*[*OF inverse-1*])

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*
using *assms*
by (*intro fps-ext*) (*simp-all add*: *fps-divide-shift-denom fps-X-power-mult-right-nth*)

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*
by (*simp add*: *assms fps-divide-shift-denom-conv-times-fps-X-power*)

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*
using *assms*
by (*simp add*:
fps-divide-def fps-X-power-subdegree fps-X-power-unit-factor fps-inverse-one'
)

lemma *fps-shift-altdef*:
fps-shift n f = (*f* :: *'a* :: *division-ring fps*) *div fps-X*^*n*
by (*rule fps-shift-altdef'*[*OF inverse-1*])

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*)
using *assms*
by (*simp add*: *fps-shift-altdef'* [*symmetric*])

lemma *fps-div-fps-X-power-nth*: ((*f* :: *'a* :: *division-ring fps*) *div fps-X*^*n*) $ *k* = *f* $ (*k* + *n*)
by (*rule fps-div-fps-X-power-nth'*[*OF inverse-1*])

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*
using *assms fps-div-fps-X-power-nth'*[*of f 1*]
by *simp*

lemma *fps-div-fps-X-nth*: ((*f* :: *'a* :: *division-ring fps*) *div fps-X*) $ *k* = *f* $ *Suc k*
by (*rule fps-div-fps-X-nth'*[*OF inverse-1*])

lemma *divide-fps-const'*:
fixes *c* :: *'a* :: {*inverse*,*comm-monoid-add*,*uminus*,*mult-zero*}
shows *f* / *fps-const c* = *f* ∗ *fps-const* (*inverse c*)
by (*simp add*: *fps-divide-def fps-const-inverse*)

lemma *divide-fps-const* [*simp*]:
fixes *c* :: *'a* :: {*comm-semiring-0*,*inverse*,*uminus*}
shows *f* / *fps-const c* = *fps-const* (*inverse c*) ∗ *f*

by (*simp add*: *divide-fps-const' mult.commute*)

lemma *fps-const-divide*: *fps-const* (*x* :: - :: *division-ring*) / *fps-const y* = *fps-const* (*x* / *y*)
by (*simp add*: *fps-divide-def fps-const-inverse divide-inverse*)

lemma *fps-numeral-divide-divide*:
x / *numeral b* / *numeral c* = (*x* / *numeral* (*b* * *c*) :: 'a :: *field fps*)
by (*simp add*: *fps-divide-by-mult*[*symmetric*])

lemma *fps-numeral-mult-divide*:
numeral b * *x* / *numeral c* = (*numeral b* / *numeral c* * *x* :: 'a :: *field fps*)
by (*simp add*: *fps-divide-times2*)

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 (∃ *g*. *1* = *f* * *g*) ⟷ (∃ *k*. *1* = *f*$*0* * *k*)
proof
assume ∃ *g*. *1* = *f* * *g*
then obtain *g* **where** *1* = *f* * *g* **by** *fast*
hence *1* = *f*$*0* * *g*$*0* **using** *fps-mult-nth-0*[*of f g*] **by** *simp*
thus ∃ *k*. *1* = *f*$*0* * *k* **by** *auto*
next
assume ∃ *k*. *1* = *f*$*0* * *k*
then obtain *k* **where** *1* = *f*$*0* * *k* **by** *fast*
hence *1* = *f* * *fps-right-inverse f k*
using *fps-right-inverse* **by** *simp*
thus ∃ *g*. *1* = *f* * *g* **by** *fast*
qed

lemma *fps-is-right-unit-iff-zeroth-is-right-unit*:
fixes *f* :: 'a :: *ring-1 fps*
shows (∃ *g*. *1* = *g* * *f*) ⟷ (∃ *k*. *1* = *k* * *f*$*0*)
proof
assume ∃ *g*. *1* = *g* * *f*
then obtain *g* **where** *1* = *g* * *f* **by** *fast*
hence *1* = *g*$*0* * *f*$*0* **using** *fps-mult-nth-0*[*of g f*] **by** *simp*
thus ∃ *k*. *1* = *k* * *f*$*0* **by** *auto*
next
assume ∃ *k*. *1* = *k* * *f*$*0*
then obtain *k* **where** *1* = *k* * *f*$*0* **by** *fast*
hence *1* = *fps-left-inverse f k* * *f*
using *fps-left-inverse* **by** *simp*
thus ∃ *g*. *1* = *g* * *f* **by** *fast*
qed

lemma *fps-is-unit-iff* [*simp*]: (*f* :: *'a* :: *field fps*) *dvd 1* ⟷ *f* $ *0* ≠ *0*
proof
  assume *f dvd 1*
  then obtain *g* **where** *1* = *f* ∗ *g* **by** (*elim dvdE*)
  from *this*[*symmetric*] **have** (*f*∗*g*) $ *0* = *1* **by** *simp*
  thus *f* $ *0* ≠ *0* **by** *auto*
next
  assume *A*: *f* $ *0* ≠ *0*
  thus *f dvd 1* **by** (*simp add*: *inverse-mult-eq-1*[*OF A*, *symmetric*])
qed

lemma *subdegree-eq-0-left*:
  fixes *f* :: *'a*::{*comm-monoid-add*,*zero-neq-one*,*mult-zero*} *fps*
  assumes ∃ *g*. *1* = *f* ∗ *g*
  shows *subdegree f* = *0*
proof (*intro subdegree-eq-0*)
  from *assms* **obtain** *g* **where** *1* = *f* ∗ *g* **by** *fast*
  hence *f*$*0* ∗ *g*$*0* = *1* **using** *fps-mult-nth-0*[*of f g*] **by** *simp*
  thus *f*$*0* ≠ *0* **by** *auto*
qed

lemma *subdegree-eq-0-right*:
  fixes *f* :: *'a*::{*comm-monoid-add*,*zero-neq-one*,*mult-zero*} *fps*
  assumes ∃ *g*. *1* = *g* ∗ *f*
  shows *subdegree f* = *0*
proof (*intro subdegree-eq-0*)
  from *assms* **obtain** *g* **where** *1* = *g* ∗ *f* **by** *fast*
  hence *g*$*0* ∗ *f*$*0* = *1* **using** *fps-mult-nth-0*[*of g f*] **by** *simp*
  thus *f*$*0* ≠ *0* **by** *auto*
qed

lemma *subdegree-eq-0'* [*simp*]: (*f* :: *'a* :: *field fps*) *dvd 1* ⟹ *subdegree f* = *0*
  by *simp*

lemma *fps-dvd1-left-trivial-unit-factor*:
  fixes *f* :: *'a*::{*comm-monoid-add*, *zero-neq-one*, *mult-zero*} *fps*
  assumes ∃ *g*. *1* = *f* ∗ *g*
  shows *unit-factor f* = *f*
  using *assms subdegree-eq-0-left*
  by *fastforce*

lemma *fps-dvd1-right-trivial-unit-factor*:
  fixes *f* :: *'a*::{*comm-monoid-add*, *zero-neq-one*, *mult-zero*} *fps*
  assumes ∃ *g*. *1* = *g* ∗ *f*
  shows *unit-factor f* = *f*
  using *assms subdegree-eq-0-right*
  by *fastforce*

lemma *fps-dvd1-trivial-unit-factor*:

(*f* :: *'a*::*comm-semiring-1 fps*) *dvd 1* ⟹ *unit-factor f* = *f*
unfolding *dvd-def* **by** (*rule fps-dvd1-left-trivial-unit-factor*) *simp*

lemma *fps-unit-dvd-left*:
fixes *f* :: *'a* :: *division-ring fps*
assumes *f* $ *0* ≠ *0*
shows ∃ *g*. *1* = *f* ∗ *g*
using *assms fps-is-left-unit-iff-zeroth-is-left-unit right-inverse*
by *fastforce*

lemma *fps-unit-dvd-right*:
fixes *f* :: *'a* :: *division-ring fps*
assumes *f* $ *0* ≠ *0*
shows ∃ *g*. *1* = *g* ∗ *f*
using *assms fps-is-right-unit-iff-zeroth-is-right-unit left-inverse*
by *fastforce*

lemma *fps-unit-dvd* [*simp*]: (*f* $ *0* :: *'a* :: *field*) ≠ *0* ⟹ *f dvd g*
using *fps-unit-dvd-left dvd-trans*[*of f 1*] **by** *simp*

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*
using *assms fps-mult-subdegree-ge*
by *fastforce*

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*
using *assms fps-mult-subdegree-ge*
by *fastforce*

lemma *dvd-imp-subdegree-le*:
f dvd g ⟹ *g* ≠ *0* ⟹ *subdegree f* ≤ *subdegree g*
using *dvd-left-imp-subdegree-le* **by** *fast*

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−
define *h* :: *'a fps* **where** *h* ≡ *fps-X* ^ (*subdegree g* − *subdegree f*)
from *assms*(*1*) **obtain** *y* **where** *f* $ *subdegree f* ∗ *y* = *1* **by** *fast*
hence *unit-factor f* $ *0* ∗ *y* = *1* **by** *simp*
from *this* **obtain** *k* **where** *1* = *unit-factor f* ∗ *k*
using *fps-is-left-unit-iff-zeroth-is-left-unit*[*of unit-factor f*] **by** *auto*
hence *fps-X* ^ *subdegree f* = *fps-X* ^ *subdegree f* ∗ *unit-factor f* ∗ *k*

```
    by (simp add: mult.assoc)
  moreover have fps-X ^ subdegree f * unit-factor f = f
    by (rule fps-unit-factor-decompose'[symmetric])
  ultimately have
    fps-X ^ (subdegree f + (subdegree g − subdegree f)) = f * k * h
    by (simp add: power-add h-def)
  hence g = f * (k * h * unit-factor g)
    using fps-unit-factor-decompose'[of g]
    by    (simp add: assms(2) mult.assoc)
  thus ?thesis by fast
qed

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 (intro subdegree-le-imp-dvd-left-ring1)
  from assms(1) have f $ subdegree f ≠ 0 by simp
  thus ∃y. f $ subdegree f * y = 1 using right-inverse by blast
qed (rule assms(2))

lemma subdegree-le-imp-dvd-right-ring1:
  fixes   f g :: 'a :: ring-1 fps
  assumes ∃x. x * f $ subdegree f = 1 subdegree f ≤ subdegree g
  shows   ∃k. g = k * f
proof−
  define h :: 'a fps where h ≡ fps-X ^ (subdegree g − subdegree f)
  from assms(1) obtain x where x * f $ subdegree f = 1 by fast
  hence x * unit-factor f $ 0 = 1 by simp
  from this obtain k where 1 = k * unit-factor f
    using fps-is-right-unit-iff-zeroth-is-right-unit[of unit-factor f] by auto
  hence fps-X ^ subdegree f = k * (unit-factor f * fps-X ^ subdegree f)
    by (simp add: mult.assoc[symmetric])
  moreover have unit-factor f * fps-X ^ subdegree f = f
    by (rule fps-unit-factor-decompose[symmetric])
  ultimately have fps-X ^ (subdegree g − subdegree f + subdegree f) = h * k * f
    by (simp add: power-add h-def mult.assoc)
  hence g = unit-factor g * h * k * f
    using fps-unit-factor-decompose[of g]
    by    (simp add: assms(2) mult.assoc)
  thus ?thesis by fast
qed

lemma subdegree-le-imp-dvd-right-divring:
  fixes   f g :: 'a :: division-ring fps
  assumes f ≠ 0 subdegree f ≤ subdegree g
  shows   ∃k. g = k * f
proof (intro subdegree-le-imp-dvd-right-ring1)
  from assms(1) have f $ subdegree f ≠ 0 by simp
```

thus $\exists x.\ x * f\ \$\ subdegree\ f = 1$ **using** *left-inverse* **by** *blast*
qed (*rule assms*(*2*))

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
assume $subdegree\ f \leq subdegree\ g$
with *assms* **show** $f\ dvd\ g$
using *subdegree-le-imp-dvd-left-divring*
by (*auto intro*: *dvdI*)
qed (*simp add*: *assms dvd-imp-subdegree-le*)

lemma *subdegree-div'*:
fixes $p\ q :: 'a::division\text{-}ring\ fps$
assumes $\exists k.\ p = k * q$
shows $subdegree\ (p\ div\ q) = subdegree\ p - subdegree\ q$
proof (*cases* $p = 0$)
case *False*
from *assms*(*1*) **obtain** k **where** k: $p = k * q$ **by** *blast*
with *False* **have** $subdegree\ (p\ div\ q) = subdegree\ k$ **by** (*simp add*: *fps-divide-times-eq*)
moreover have $k\ \$\ subdegree\ k * q\ \$\ subdegree\ q \neq 0$
proof
assume $k\ \$\ subdegree\ k * q\ \$\ subdegree\ q = 0$
hence $k\ \$\ subdegree\ k * q\ \$\ subdegree\ q * inverse\ (q\ \$\ subdegree\ q) = 0$ **by** *simp*
with *False* k **show** *False* **by** (*simp add*: *mult.assoc*)
qed
ultimately show *?thesis* **by** (*simp add*: k *subdegree-mult'*)
qed *simp*

lemma *subdegree-div*:
fixes $p\ q :: 'a :: field\ fps$
assumes $q\ dvd\ p$
shows $subdegree\ (p\ div\ q) = subdegree\ p - subdegree\ q$
using *assms*
unfolding *dvd-def*
by (*auto intro*: *subdegree-div'*)

lemma *subdegree-div-unit'*:
fixes $p\ q :: 'a :: \{ab\text{-}group\text{-}add,mult\text{-}zero,inverse\}\ fps$
assumes $q\ \$\ 0 \neq 0\ p\ \$\ subdegree\ p * inverse\ (q\ \$\ 0) \neq 0$
shows $subdegree\ (p\ div\ q) = subdegree\ p$
using *assms subdegree-mult'*[*of p inverse q*]
by (*auto simp add*: *fps-divide-unit*)

lemma *subdegree-div-unit''*:
fixes $p\ q :: 'a :: \{ring\text{-}no\text{-}zero\text{-}divisors,inverse\}\ fps$
assumes $q\ \$\ 0 \neq 0\ inverse\ (q\ \$\ 0) \neq 0$

shows *subdegree* (*p div q*) = *subdegree p*
by (*cases p* = *0*) (*auto intro*: *subdegree-div-unit' simp*: *assms*)

lemma *subdegree-div-unit*:
fixes *p q* :: *'a* :: *division-ring fps*
assumes *q* $ *0* ≠ *0*
shows *subdegree* (*p div q*) = *subdegree p*
by (*intro subdegree-div-unit''*) (*simp-all add*: *assms*)

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 ..

end

lemma *fps-mod-zero* [*simp*]:
(*f*::*'a*::{*comm-semiring-1*,*inverse*,*uminus*} *fps*) *mod* *0* = *f*
by (*simp add*: *fps-mod-def*)

lemma *fps-mod-eq-zero*:
assumes *g* ≠ *0* **and** *subdegree f* ≥ *subdegree g*
shows *f mod g* = *0*
proof (*cases f* ∗ *inverse* (*unit-factor g*) = *0*)
case *False*
have *fps-cutoff* (*subdegree g*) (*f* ∗ *inverse* (*unit-factor g*)) = *0*
using *False assms*(*2*) *fps-mult-subdegree-ge fps-cutoff-zero-iff* **by** *force*
with *assms*(*1*) **show** *?thesis* **by** (*simp add*: *fps-mod-def Let-def*)
qed (*simp add*: *assms fps-mod-def*)

lemma *fps-mod-unit* [*simp*]: *g*$*0* ≠ *0* ⟹ *f mod g* = *0*
by (*intro fps-mod-eq-zero*) *auto*

lemma *subdegree-mod*:
assumes *subdegree* (*f*::*'a*::*field fps*) < *subdegree g*
shows *subdegree* (*f mod g*) = *subdegree f*
proof (*cases f* = *0*)
case *False*
with *assms* **show** *?thesis*
by (*intro subdegreeI*)
(*auto simp*: *inverse-mult-eq-1 fps-mod-def Let-def fps-cutoff-left-mult-nth mult.assoc*)
qed (*simp add*: *fps-mod-def*)

instance *fps* :: (*field*) *idom-modulo*

proof

fix *f g* :: *'a fps*

define *n* **where** *n* = *subdegree g*
define *h* **where** *h* = *f* ∗ *inverse* (*unit-factor g*)

show *f div g* ∗ *g* + *f mod g* = *f*
proof (*cases g* = *0*)
case *False*
with *n-def h-def* **have**
f div g ∗ *g* + *f mod g* = (*fps-shift n h* ∗ *fps-X* ^ *n* + *fps-cutoff n h*) ∗ *unit-factor g*
by (*simp add*: *fps-divide-def fps-mod-def Let-def subdegree-decompose algebra-simps*)
with *False* **show** *?thesis*
by (*simp add*: *fps-shift-cutoff h-def inverse-mult-eq-1*)
qed *auto*

qed (*rule fps-divide-times-eq*, *simp-all add*: *fps-divide-def*)

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
fix *f g* :: *'a fps*
assume *is-unit f*
thus *unit-factor* (*f* ∗ *g*) = *f* ∗ *unit-factor g*
using *fps-unit-factor-mult*[*of f g*] **by** *simp*
next
fix *f g* :: *'a fps*
show *unit-factor f* ∗ *normalize f* = *f*
by (*simp add*: *fps-shift-times-fps-X-power*)
next
fix *f g* :: *'a fps*
show *unit-factor* (*f* ∗ *g*) = *unit-factor f* ∗ *unit-factor g*
using *fps-unit-factor-mult*[*of f g*] **by** *simp*
qed (*simp-all add*: *fps-divide-def Let-def*)

end

5.8 Computing reciprocals via Hensel lifting

lemma *inverse-fps-hensel-lifting*:
fixes *F G* :: *'a* :: *field fps* **and** *n* :: *nat*
assumes *G-eq*: *fps-cutoff n G* = *fps-cutoff n* (*inverse F*)

assumes *unit*: *fps-nth F 0* $\neq$ *0*
shows *fps-cutoff* (*2*∗*n*) (*inverse F*) = *fps-cutoff* (*2*∗*n*) (*G* ∗ (*2* − *F* ∗ *G*))
proof −
define *R* **where** *R* = *inverse F* − *G*
have *eq*: *G* = *inverse F* − *R*
by (*simp add*: *R-def*)
from *assms* **have** *fps-cutoff n R* = *0*
by (*simp add*: *R-def fps-cutoff-diff*)
hence *R*: *R* = *0* ∨ *subdegree R* ≥ *n*
by (*simp add*: *fps-cutoff-zero-iff*)

have *G* ∗ (*2* − *F* ∗ *G*) − *inverse F* =
inverse F + *F* ∗ *inverse F* ∗ *R* ∗ *2* − *F* ∗ R^2 − *R* ∗ *2* − *F* ∗ *inverse F* ∗ *inverse F*
by (*simp add*: *eq algebra-simps power2-eq-square*)
also have *F* ∗ *inverse F* = *1*
using *unit* **by** (*simp add*: *inverse-mult-eq-1* ′)
also have *inverse F* + *1* ∗ *R* ∗ *2* − *F* ∗ R^2 − *R* ∗ *2* − *1* ∗ *inverse F* = −*F* ∗ R^2
by (*simp add*: *algebra-simps*)
finally have *fps-cutoff* (*2*∗*n*) (*G* ∗ (*2* − *F* ∗ *G*) − *inverse F*) = *fps-cutoff* (*2*∗*n*) (−*F* ∗ R^2)
by (*rule arg-cong*)
also have ... = *0*
proof (*cases* −*F* ∗ R^2 = *0*)
case *False*
have *2* ∗ *n* ≤ *subdegree* (−*F* ∗ R^2)
using *False R unit* **by** *simp*
thus *?thesis*
by (*simp add*: *fps-cutoff-zero-iff*)
qed *auto*
finally show *?thesis*
by (*simp add*: *fps-cutoff-diff*)
qed

lemma *inverse-fps-hensel-lifting* ′:
fixes *F G* :: ′*a* :: *field fps* **and** *n* :: *nat*
assumes *G-eq*: *fps-cutoff n G* = *fps-cutoff n* (*inverse F*)
assumes *unit*: *fps-nth F 0* $\neq$ *0*
defines *P* ≡ *fps-shift n* (*F* ∗ *G* − *1*)
shows *fps-cutoff* (*2*∗*n*) (*inverse F*) = *fps-cutoff* (*2*∗*n*) (*G* ∗ (*1* − *fps-X* ^ *n* ∗ *P*))
proof −
define *R* **where** *R* = *inverse F* − *G*
have *eq*: *G* = *inverse F* − *R*
by (*simp add*: *R-def*)
from *assms* **have** *fps-cutoff n R* = *0*
by (*simp add*: *R-def fps-cutoff-diff*)
hence *R*: *R* = *0* ∨ *subdegree R* ≥ *n*

```
    by (simp add: fps-cutoff-zero-iff)

  have FG-eq: F * G = 1 + fps-X ^ n * P
  proof (cases F * G - 1 = 0)
    case False
    have eq: F * G - 1 = F * (G - inverse F)
      using unit by (simp add: inverse-mult-eq-1' ring-distribs)
    have subdegree (F * (G - inverse F)) ≥ n
    proof -
      have fps-cutoff n (G - inverse F) = 0
        using G-eq by (simp add: fps-cutoff-diff)
      hence n ≤ subdegree (G - inverse F)
        using False unfolding eq by (simp add: fps-cutoff-zero-iff)
      also have subdegree (G - inverse F) = subdegree (F * (G - inverse F))
        by (subst subdegree-mult) (use unit False in ‹auto simp: eq›)
      finally have n ≤ subdegree (F * (G - inverse F)) .
      thus ?thesis
        by blast
    qed
    hence F * G - 1 = fps-X ^ n * P
      unfolding eq P-def by (intro fps-conv-fps-X-power-mult-fps-shift) auto
    thus ?thesis
      by (simp add: algebra-simps)
  qed (auto simp: P-def)

  have G * (1 - fps-X ^ n * P) - inverse F = G * (2 - F * G) - inverse F
    by (auto simp: FG-eq)
  also have G * (2 - F * G) - inverse F =
        inverse F + F * inverse F * R * 2 - F * R^2 - R * 2 - F * inverse F *
inverse F
    by (simp add: eq algebra-simps power2-eq-square)
  also have F * inverse F = 1
    using unit by (simp add: inverse-mult-eq-1')
  also have inverse F + 1 * R * 2 - F * R^2 - R * 2 - 1 * inverse F = -F *
R^2
    by (simp add: algebra-simps)
  finally have fps-cutoff (2*n) (G * (1 - fps-X ^ n * P) - inverse F) = fps-cutoff
(2*n) (-F * R^2)
    by (rule arg-cong)
  also have ... = 0
  proof (cases -F * R^2 = 0)
    case False
    have 2 * n ≤ subdegree (-F * R^2)
      using False R unit by simp
    thus ?thesis
      by (simp add: fps-cutoff-zero-iff)
  qed auto
  finally show ?thesis
    by (simp add: fps-cutoff-diff)
```

qed

5.9 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
fix *f g* :: *'a fps* **assume** [*simp*]: *g* ≠ *0*
show *euclidean-size f* ≤ *euclidean-size* (*f* ∗ *g*)
by (*cases f* = *0*) (*simp-all add*: *fps-euclidean-size-def*)
show *euclidean-size* (*f mod g*) < *euclidean-size g*
proof (*cases f* = *0*)
case *True*
then show *?thesis*
by (*simp add*: *fps-euclidean-size-def*)
next
case *False*
then show *?thesis*
using *le-less-linear*[*of subdegree g subdegree f*]
by (*force simp add*: *fps-mod-eq-zero fps-euclidean-size-def subdegree-mod*)
qed
next
fix *f g h* :: *'a fps* **assume** [*simp*]: *h* ≠ *0*
show (*h* ∗ *f*) *div* (*h* ∗ *g*) = *f div g*
by (*simp add*: *fps-divide-cancel mult.commute*)
show (*f* + *g* ∗ *h*) *div h* = *g* + *f div h*
by (*simp add*: *fps-divide-add fps-divide-times-eq*)
qed (*simp add*: *fps-euclidean-size-def*)

end

instance *fps* :: (*field*) *normalization-euclidean-semiring* **..**

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 by *standard* (*simp-all add*: *fps-gcd-def fps-lcm-def fps-Gcd-def fps-Lcm-def*)
end

lemma *fps-gcd*:
assumes [*simp*]: *f* ≠ *0 g* ≠ *0*
shows *gcd f g* = *fps-X* ^ *min* (*subdegree f*) (*subdegree g*)

proof −
let *?m* = *min* (*subdegree f*) (*subdegree g*)
show *gcd f g* = *fps-X* ^ *?m*
proof (*rule sym*, *rule gcdI*)
fix *d* **assume** *d dvd f d dvd g*
thus *d dvd fps-X* ^ *?m* **by** (*cases d* = *0*) (*simp-all add*: *fps-dvd-iff*)
qed (*simp-all add*: *fps-dvd-iff*)
qed

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*))
by (*simp add*: *fps-gcd*)

lemma *fps-lcm*:
assumes [*simp*]: *f* ≠ *0 g* ≠ *0*
shows *lcm f g* = *fps-X* ^ *max* (*subdegree f*) (*subdegree g*)
proof −
let *?m* = *max* (*subdegree f*) (*subdegree g*)
show *lcm f g* = *fps-X* ^ *?m*
proof (*rule sym*, *rule lcmI*)
fix *d* **assume** *f dvd d g dvd d*
thus *fps-X* ^ *?m dvd d* **by** (*cases d* = *0*) (*simp-all add*: *fps-dvd-iff*)
qed (*simp-all add*: *fps-dvd-iff*)
qed

lemma *fps-lcm-altdef*: *lcm f g* =
(*if f* = *0* ∨ *g* = *0 then 0 else fps-X* ^ *max* (*subdegree f*) (*subdegree g*))
by (*simp add*: *fps-lcm*)

lemma *fps-Gcd*:
assumes *A* − {*0*} ≠ {}
shows *Gcd A* = *fps-X* ^ (*INF f*∈*A*−{*0*}. *subdegree f*)
proof (*rule sym*, *rule GcdI*)
fix *f* **assume** *f* ∈ *A*
thus *fps-X* ^ (*INF f*∈*A* − {*0*}. *subdegree f*) *dvd f*
by (*cases f* = *0*) (*auto simp*: *fps-dvd-iff intro*!: *cINF-lower*)
next
fix *d* **assume** *d*: ⋀*f*. *f* ∈ *A* ⟹ *d dvd f*
from *assms* **obtain** *f* **where** *f* ∈ *A* − {*0*} **by** *auto*
with *d*[*of f*] **have** [*simp*]: *d* ≠ *0* **by** *auto*
from *d assms* **have** *subdegree d* ≤ (*INF f*∈*A*−{*0*}. *subdegree f*)
by (*intro cINF-greatest*) (*simp-all add*: *fps-dvd-iff*[*symmetric*])
with *d assms* **show** *d dvd fps-X* ^ (*INF f*∈*A*−{*0*}. *subdegree f*) **by** (*simp add*: *fps-dvd-iff*)
qed *simp-all*

lemma *fps-Gcd-altdef*: *Gcd A* =
(*if A* ⊆ {*0*} *then 0 else fps-X* ^ (*INF f*∈*A*−{*0*}. *subdegree f*))
using *fps-Gcd* **by** *auto*

lemma *fps-Lcm*:
assumes *A* ≠ {} *0* ∉ *A bdd-above* (*subdegree*‘*A*)
shows *Lcm A* = *fps-X* ^ (*SUP f*∈*A*. *subdegree f*)
proof (*rule sym*, *rule LcmI*)
fix *f* **assume** *f* ∈ *A*
moreover from *assms*(*3*) **have** *bdd-above* (*subdegree* ‘ *A*) **by** *auto*
ultimately show *f dvd fps-X* ^ (*SUP f*∈*A*. *subdegree f*) **using** *assms*(*2*)
by (*cases f* = *0*) (*auto simp*: *fps-dvd-iff intro*!: *cSUP-upper*)
next
fix *d* **assume** *d*: ⋀*f*. *f* ∈ *A* ⟹ *f dvd d*
from *assms* **obtain** *f* **where** *f*: *f* ∈ *A f* ≠ *0* **by** *auto*
show *fps-X* ^ (*SUP f*∈*A*. *subdegree f*) *dvd d*
proof (*cases d* = *0*)
assume *d* ≠ *0*
moreover from *d* **have** ⋀*f*. *f* ∈ *A* ⟹ *f* ≠ *0* ⟹ *f dvd d* **by** *blast*
ultimately have *subdegree d* ≥ (*SUP f*∈*A*. *subdegree f*) **using** *assms*
by (*intro cSUP-least*) (*auto simp*: *fps-dvd-iff*)
with ‹*d* ≠ *0*› **show** *?thesis* **by** (*simp add*: *fps-dvd-iff*)
qed *simp-all*
qed *simp-all*

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 (*cases bdd-above* (*subdegree*‘*A*))
assume *unbounded*: ¬*bdd-above* (*subdegree*‘*A*)
have *Lcm A* = *0*
proof (*rule ccontr*)
assume *Lcm A* ≠ *0*
from *unbounded* **obtain** *f* **where** *f*: *f* ∈ *A subdegree* (*Lcm A*) < *subdegree f*
unfolding *bdd-above-def* **by** (*auto simp*: *not-le*)
moreover from *f* **and** ‹*Lcm A* ≠ *0*› **have** *subdegree f* ≤ *subdegree* (*Lcm A*)
by (*intro dvd-imp-subdegree-le dvd-Lcm*) *simp-all*
ultimately show *False* **by** *simp*
qed
with *unbounded* **show** *?thesis* **by** *simp*
qed (*simp-all add*: *fps-Lcm Lcm-eq-0-I*)

5.10 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*)
by (*simp add*: *fps-deriv-def*)

lemma *fps-0th-higher-deriv*:
(*fps-deriv* ^^ *n*) *f* $ *0* = *fact n* * *f* $ *n*
by (*induction n arbitrary*: *f*)
(*simp-all add*: *funpow-Suc-right mult-of-nat-commute algebra-simps del*: *funpow.simps*)

lemma *fps-deriv-mult*[*simp*]:
fps-deriv (*f* * *g*) = *f* * *fps-deriv g* + *fps-deriv f* * *g*
proof (*intro fps-ext*)
fix *n*
have *LHS*: *fps-deriv* (*f* * *g*) $ *n* = (∑ *i*=*0*..*Suc n*. *of-nat* (*n*+*1*) * *f*$*i* * *g*$(*Suc n* − *i*))
by (*simp add*: *fps-mult-nth sum-distrib-left algebra-simps*)

have ∀ *i*∈{*1*..*n*}. *n* − (*i* − *1*) = *n* − *i* + *1* **by** *auto*
moreover have
(∑ *i*=*0*..*n*. *of-nat* (*i*+*1*) * *f*$(*i*+*1*) * *g*$(*n* − *i*)) =
(∑ *i*=*1*..*Suc n*. *of-nat i* * *f*$*i* * *g*$(*n* − (*i* − *1*)))
by (*intro sum.reindex-bij-witness*[**where** *i*=λ*x*. *x*−*1* **and** *j*=λ*x*. *x*+*1*]) *auto*
ultimately have
(*f* * *fps-deriv g* + *fps-deriv f* * *g*) $ *n* =
of-nat (*Suc n*) * *f*$*0* * *g*$(*Suc n*) +
(∑ *i*=*1*..*n*. (*of-nat* (*n* − *i* + *1*) + *of-nat i*) * *f* $ *i* * *g* $ (*n* − *i* + *1*)) +
of-nat (*Suc n*) * *f*$(*Suc n*) * *g*$*0*
by (*simp add*: *fps-mult-nth algebra-simps mult-of-nat-commute sum.atLeast-Suc-atMost sum.distrib*)
moreover have
∀ *i*∈{*1*..*n*}.
(*of-nat* (*n* − *i* + *1*) + *of-nat i*) * *f* $ *i* * *g* $ (*n* − *i* + *1*) =
of-nat (*n* + *1*) * *f* $ *i* * *g* $ (*Suc n* − *i*)
proof
fix *i* **assume** *i*: *i* ∈ {*1*..*n*}
from *i* **have** *of-nat* (*n* − *i* + *1*) + (*of-nat i* :: *'a*) = *of-nat* (*n* + *1*)
using *of-nat-add*[*of n*−*i*+*1 i*,*symmetric*] **by** *simp*
moreover from *i* **have** *Suc n* − *i* = *n* − *i* + *1* **by** *auto*
ultimately show (*of-nat* (*n* − *i* + *1*) + *of-nat i*) * *f* $ *i* * *g* $ (*n* − *i* + *1*) =
of-nat (*n* + *1*) * *f* $ *i* * *g* $ (*Suc n* − *i*)
by *simp*
qed
ultimately have
(*f* * *fps-deriv g* + *fps-deriv f* * *g*) $ *n* =
(∑ *i*=*0*..*Suc n*. *of-nat* (*Suc n*) * *f* $ *i* * *g* $ (*Suc n* − *i*))
by (*simp add*: *sum.atLeast-Suc-atMost*)
with *LHS* **show** *fps-deriv* (*f* * *g*) $ *n* = (*f* * *fps-deriv g* + *fps-deriv f* * *g*) $ *n*
by *simp*
qed

lemma *fps-deriv-fps-X*[*simp*]: *fps-deriv fps-X* = *1*

by (*simp add*: *fps-deriv-def fps-X-def fps-eq-iff*)

lemma *fps-deriv-neg*[*simp*]:
fps-deriv (− (*f*:: *'a*::*ring-1 fps*)) = − (*fps-deriv f*)
by (*simp add*: *fps-eq-iff fps-deriv-def*)

lemma *fps-deriv-add*[*simp*]: *fps-deriv* (*f* + *g*) = *fps-deriv f* + *fps-deriv g*
by (*auto intro*: *fps-ext simp*: *algebra-simps*)

lemma *fps-deriv-sub*[*simp*]:
fps-deriv ((*f*:: *'a*::*ring-1 fps*) − *g*) = *fps-deriv f* − *fps-deriv g*
using *fps-deriv-add* [*of f* − *g*] **by** *simp*

lemma *fps-deriv-const*[*simp*]: *fps-deriv* (*fps-const c*) = *0*
by (*simp add*: *fps-ext fps-deriv-def fps-const-def*)

lemma *fps-deriv-of-nat* [*simp*]: *fps-deriv* (*of-nat n*) = *0*
by (*simp add*: *fps-of-nat* [*symmetric*])

lemma *fps-deriv-of-int* [*simp*]: *fps-deriv* (*of-int n*) = *0*
by (*simp add*: *fps-of-int* [*symmetric*])

lemma *fps-deriv-numeral* [*simp*]: *fps-deriv* (*numeral n*) = *0*
by (*simp add*: *numeral-fps-const*)

lemma *fps-deriv-mult-const-left*[*simp*]:
fps-deriv (*fps-const c* ∗ *f*) = *fps-const c* ∗ *fps-deriv f*
by *simp*

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*
by *simp*

lemma *fps-deriv-0*[*simp*]: *fps-deriv 0* = *0*
by (*simp add*: *fps-deriv-def fps-eq-iff*)

lemma *fps-deriv-1*[*simp*]: *fps-deriv 1* = *0*
by (*simp add*: *fps-deriv-def fps-eq-iff*)

lemma *fps-deriv-mult-const-right*[*simp*]:
fps-deriv (*f* ∗ *fps-const c*) = *fps-deriv f* ∗ *fps-const c*
by *simp*

lemma *fps-deriv-sum*:
fps-deriv (*sum f S*) = *sum* (λ*i*. *fps-deriv* (*f i*)) *S*
proof (*cases finite S*)
case *False*
then show *?thesis* **by** *simp*

next
  case *True*
  show *?thesis* **by** (*induct rule*: *finite-induct* [*OF True*]) *simp-all*
qed

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
  assume *f*: *fps-deriv f = 0*
  show *f = fps-const* (*f$0*)
  proof (*intro fps-ext*)
    fix *n* **show** *f $ n = fps-const* (*f$0*) *$ n*
    proof (*cases n*)
      case (*Suc m*)
      have (*of-nat* (*Suc m*) :: *'a*) ≠ *0* **by** (*rule of-nat-neq-0*)
      with *f Suc* **show** *?thesis* **using** *fps-deriv-nth*[*of f*] **by** *auto*
    qed *simp*
  qed
next
  show *f = fps-const* (*f$0*) ⟹ *fps-deriv f = 0* **using** *fps-deriv-const*[*of f$0*] **by**
simp
qed

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 −
  have *fps-deriv f = fps-deriv g* ⟷ *fps-deriv* (*f − g*) = *0*
    using *fps-deriv-sub*[*of f g*]
    by *simp*
  also have ... ⟷ *f − g = fps-const* ((*f − g*) $ *0*)
    unfolding *fps-deriv-eq-0-iff* **..**
  finally show *?thesis*
    by (*simp add*: *field-simps*)
qed

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*)
  by (*auto simp*: *fps-deriv-eq-iff*)

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)

by (*induct n arbitrary*: *f*) *auto*

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*
by (*induct n arbitrary*: *f g*) *auto*

lemma *fps-nth-deriv-neg*[*simp*]:
fps-nth-deriv n (− (*f* :: *'a*::*ring-1 fps*)) = − (*fps-nth-deriv n f*)
by (*induct n arbitrary*: *f*) *simp-all*

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*
using *fps-nth-deriv-linear*[*of n 1 f 1 g*] **by** *simp*

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*
using *fps-nth-deriv-add* [*of n f* − *g*] **by** *simp*

lemma *fps-nth-deriv-0*[*simp*]: *fps-nth-deriv n 0* = *0*
by (*induct n*) *simp-all*

lemma *fps-nth-deriv-1*[*simp*]: *fps-nth-deriv n 1* = (*if n* = *0 then 1 else 0*)
by (*induct n*) *simp-all*

lemma *fps-nth-deriv-const*[*simp*]:
fps-nth-deriv n (*fps-const c*) = (*if n* = *0 then fps-const c else 0*)
by (*cases n*) *simp-all*

lemma *fps-nth-deriv-mult-const-left*[*simp*]:
fps-nth-deriv n (*fps-const c* ∗ *f*) = *fps-const c* ∗ *fps-nth-deriv n f*
using *fps-nth-deriv-linear*[*of n c f 0 0*] **by** *simp*

lemma *fps-nth-deriv-mult-const-right*[*simp*]:
fps-nth-deriv n (*f* ∗ *fps-const c*) = *fps-nth-deriv n f* ∗ *fps-const c*
by (*induct n arbitrary*: *f*) *auto*

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 (*cases finite S*)
case *True*
show *?thesis* **by** (*induct rule*: *finite-induct* [*OF True*]) *simp-all*
next
case *False*
then show *?thesis* **by** *simp*
qed

lemma *fps-deriv-maclauren-0*:
(*fps-nth-deriv k* (*f* :: *'a*::*comm-semiring-1 fps*)) $ *0* = *of-nat* (*fact k*) ∗ *f* $ *k*

by (*induct k arbitrary*: *f*) (*simp-all add*: *field-simps*)

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−

define *L* **where** *L* ≡ *fps-left-inverse f x*
hence *fps-deriv* (*L* ∗ *f*) = *0* **using** *fps-left-inverse*[*OF assms*(*1*)] **by** *simp*
with *assms* **show** *fps-deriv L* = − *L* ∗ *fps-deriv f* ∗ *L*
using *fps-right-inverse'*[*OF assms*]
by (*simp add*: *minus-unique mult.assoc L-def*)

define *R* **where** *R* ≡ *fps-right-inverse f y*
hence *fps-deriv* (*f* ∗ *R*) = *0* **using** *fps-right-inverse*[*OF assms*(*2*)] **by** *simp*
hence *1*: *f* ∗ *fps-deriv R* + *fps-deriv f* ∗ *R* = *0* **by** *simp*
have *R* ∗ *f* ∗ *fps-deriv R* = − *R* ∗ *fps-deriv f* ∗ *R*
using *iffD2*[*OF eq-neg-iff-add-eq-0*, *OF 1*] **by** (*simp add*: *mult.assoc*)
thus *fps-deriv R* = − *R* ∗ *fps-deriv f* ∗ *R*
using *fps-left-inverse'*[*OF assms*] **by** (*simp add*: *R-def*)

qed

lemma *fps-deriv-lr-inverse-comm*:
fixes *x* :: *'a*::*comm-ring-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
using *assms fps-deriv-lr-inverse*[*of x f x*]
by (*simp-all add*: *mult.commute power2-eq-square*)

lemma *fps-inverse-deriv-divring*:
fixes *a* :: *'a*::*division-ring fps*
assumes *a*$*0* ≠ *0*
shows *fps-deriv* (*inverse a*) = − *inverse a* ∗ *fps-deriv a* ∗ *inverse a*
using *assms fps-deriv-lr-inverse*(*2*)[*of inverse* (*a*$*0*) *a inverse* (*a*$*0*)]
by (*simp add*: *fps-inverse-def*)

lemma *fps-inverse-deriv*:
fixes *a* :: *'a*::*field fps*
assumes *a*$*0* ≠ *0*
shows *fps-deriv* (*inverse a*) = − *fps-deriv a* ∗ (*inverse a*)^2
using *assms fps-deriv-lr-inverse-comm*(*2*)[*of inverse* (*a*$*0*) *a*]
by (*simp add*: *fps-inverse-def*)

lemma *fps-inverse-deriv'*:
fixes *a* :: *'a*::*field fps*
assumes *a0*: *a $ 0 ≠ 0*
shows *fps-deriv* (*inverse a*) = − *fps-deriv a* / a^2
using *fps-inverse-deriv*[*OF a0*] *a0*
by (*simp add*: *fps-divide-unit power2-eq-square fps-inverse-mult*)

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 −
have *eq-divide-imp*: *c* ≠ *0* ⟹ *a* ∗ *c* = *b* ⟹ *a* = *b div c* **for** *a b c* :: *'a* :: *field fps*
by (*drule sym*) (*simp add*: *mult.assoc*)
from *assms* **have** *a* = *a* / *b* ∗ *b* **by** *simp*
also have *fps-deriv* (*a* / *b* ∗ *b*) = *fps-deriv* (*a* / *b*) ∗ *b* + *a* / *b* ∗ *fps-deriv b* **by** *simp*
finally have *fps-deriv* (*a* / *b*) ∗ *b*^2 = *fps-deriv a* ∗ *b* − *a* ∗ *fps-deriv b* **using** *assms*
by (*simp add*: *power2-eq-square algebra-simps*)
thus *?thesis* **by** (*cases b* = *0*) (*simp-all add*: *eq-divide-imp*)
qed

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*)
by (*cases n*) *simp-all*

5.11 Powers

lemma *fps-power-zeroth*: (*a*^*n*) $ *0* = (*a$0*)^*n*
by (*induct n*) *auto*

lemma *fps-power-zeroth-eq-one*: *a$0* = *1* ⟹ *a*^*n* $ *0* = *1*
by (*simp add*: *fps-power-zeroth*)

lemma *fps-power-first*:
fixes *a* :: *'a*::*comm-semiring-1 fps*
shows (*a*^*n*) $ *1* = *of-nat n* ∗ (*a$0*)^(*n*−*1*) ∗ *a$1*
proof (*cases n*)
case (*Suc m*)
have (*a* ^ *Suc m*) $ *1* = *of-nat* (*Suc m*) ∗ (*a$0*)^(*Suc m* − *1*) ∗ *a$1*
proof (*induct m*)
case (*Suc k*)
hence (*a* ^ *Suc* (*Suc k*)) $ *1* =
a$0 ∗ *of-nat* (*Suc k*) ∗ (*a* $ *0*)^*k* ∗ *a$1* + *a$1* ∗ ((*a$0*)^(*Suc k*))
using *fps-mult-nth-1*[*of a*] **by** (*simp add*: *fps-power-zeroth*[*symmetric*] *mult.assoc*)
thus *?case* **by** (*simp add*: *algebra-simps*)

qed *simp*
with *Suc* **show** *?thesis* **by** *simp*
qed *simp*

lemma *fps-power-first-eq*: $a\ \$\ 0 = 1 \Longrightarrow a\hat{}n\ \$\ 1 = of\text{-}nat\ n * a\$1$
proof (*induct n*)
case (*Suc n*)
show *?case* **unfolding** *power-Suc fps-mult-nth*
using *Suc.hyps*[*OF* ‹$a\$0 = 1$›] ‹$a\$0 = 1$› *fps-power-zeroth-eq-one*[*OF* ‹$a\$0=1$›]
by (*simp add*: *algebra-simps*)
qed *simp*

lemma *fps-power-first-eq'*:
assumes $a\ \$\ 1 = 1$
shows $a\hat{}n\ \$\ 1 = of\text{-}nat\ n * (a\$0)\hat{}(n-1)$
proof (*cases n*)
case (*Suc m*)
from *assms* **have** $(a\ \hat{}\ Suc\ m)\ \$\ 1 = of\text{-}nat\ (Suc\ m) * (a\$0)\hat{}(Suc\ m - 1)$
using *fps-mult-nth-1*[*of a*]
by (*induct m*)
(*simp-all add*: *algebra-simps mult-of-nat-commute fps-power-zeroth*)
with *Suc* **show** *?thesis* **by** *simp*
qed *simp*

lemmas *startsby-one-power* = *fps-power-zeroth-eq-one*

lemma *startsby-zero-power*: $a\ \$\ 0 = 0 \Longrightarrow n > 0 \Longrightarrow a\hat{}n\ \$0 = 0$
by (*simp add*: *fps-power-zeroth zero-power*)

lemma *startsby-power*: $a\ \$0 = v \Longrightarrow a\hat{}n\ \$0 = v\hat{}n$
by (*simp add*: *fps-power-zeroth*)

lemma *startsby-nonzero-power*:
fixes $a :: 'a{::}semiring\text{-}1\text{-}no\text{-}zero\text{-}divisors\ fps$
shows $a\ \$\ 0 \neq 0 \Longrightarrow a\hat{}n\ \$\ 0 \neq 0$
by (*simp add*: *startsby-power*)

lemma *startsby-zero-power-iff*[*simp*]:
$a\hat{}n\ \$0 = (0{::}'a{::}semiring\text{-}1\text{-}no\text{-}zero\text{-}divisors) \longleftrightarrow n \neq 0 \wedge a\$0 = 0$
proof
show $a\ \hat{}\ n\ \$\ 0 = 0 \Longrightarrow n \neq 0 \wedge a\ \$\ 0 = 0$
proof
assume *a*: $a\hat{}n\ \$\ 0 = 0$
thus $a\ \$\ 0 = 0$ **using** *startsby-nonzero-power* **by** *auto*
have $n = 0 \Longrightarrow a\hat{}n\ \$\ 0 = 1$ **by** *simp*
with *a* **show** $n \neq 0$ **by** *fastforce*
qed
show $n \neq 0 \wedge a\ \$\ 0 = 0 \Longrightarrow a\ \hat{}\ n\ \$\ 0 = 0$
by (*cases n*) *auto*

qed

lemma *startsby-zero-power-prefix*:
assumes *a0*: $a \$ 0 = 0$
shows $\forall n < k.\ a \hat{} k \$ n = 0$
proof (*induct k rule*: *nat-less-induct*, *clarify*)
case (*1 k*)
fix $j :: nat$ **assume** *j*: $j < k$
show $a \hat{} k \$ j = 0$
proof (*cases k*)
case *0* **with** *j* **show** *?thesis* **by** *simp*
next
case (*Suc i*)
with *1 j* **have** $\forall m \in \{0<..j\}.\ a \hat{} i \$ (j - m) = 0$ **by** *auto*
with *Suc a0* **show** *?thesis* **by** (*simp add*: *fps-mult-nth sum.atLeast-Suc-atMost*)
qed
qed

lemma *startsby-zero-sum-depends*:
assumes *a0*: $a \$0 = 0$
and *kn*: $n \geq k$
shows $sum\ (\lambda i.\ (a \hat{} i)\$k)\ \{0\ ..\ n\} = sum\ (\lambda i.\ (a \hat{} i)\$k)\ \{0\ ..\ k\}$
proof (*intro strip sum.mono-neutral-right*)
show $\bigwedge i.\ i \in \{0..n\} - \{0..k\} \Longrightarrow a \hat{} i \$ k = 0$
by (*simp add*: *a0 startsby-zero-power-prefix*)
qed (*use kn* **in** *auto*)

lemma *startsby-zero-power-nth-same*:
assumes *a0*: $a\$0 = 0$
shows $a \hat{} n \$ n = (a\$1) \hat{} n$
proof (*induct n*)
case (*Suc n*)
have $\forall i \in \{Suc\ 1..Suc\ n\}.\ a \hat{} n \$ (Suc\ n - i) = 0$
using *a0 startsby-zero-power-prefix*[*of a n*] **by** *auto*
thus *?case*
using *a0 Suc sum.atLeast-Suc-atMost*[*of 0 Suc n* $\lambda i.\ a \$ i * a \hat{} n \$ (Suc\ n - i)$]
sum.atLeast-Suc-atMost[*of 1 Suc n* $\lambda i.\ a \$ i * a \hat{} n \$ (Suc\ n - i)$]
by (*simp add*: *fps-mult-nth*)
qed *simp*

lemma *fps-lr-inverse-power*:
fixes $a :: 'a::ring\text{-}1\ fps$
assumes $x * a\$0 = 1\ a\$0 * x = 1$
shows *fps-left-inverse* $(a \hat{} n)\ (x \hat{} n) = fps\text{-}left\text{-}inverse\ a\ x \hat{} n$
and *fps-right-inverse* $(a \hat{} n)\ (x \hat{} n) = fps\text{-}right\text{-}inverse\ a\ x \hat{} n$
proof−

from *assms* **have** *xn*: $\bigwedge n.\ x \hat{} n * (a \hat{} n \$ 0) = 1\ \bigwedge n.\ (a \hat{} n \$ 0) * x \hat{} n = 1$

```
    by (simp-all add: left-right-inverse-power fps-power-zeroth)

  show fps-left-inverse (a^n) (x^n) = fps-left-inverse a x ^ n
  proof (induct n)
    case 0
    then show ?case by (simp add: fps-lr-inverse-one-one(1))
  next
    case (Suc n)
    with assms show ?case
      using xn fps-lr-inverse-mult-ring1(1)[of x a x^n a^n]
      by    (simp add: power-Suc2[symmetric])
  qed

  moreover have fps-right-inverse (a^n) (x^n) = fps-left-inverse (a^n) (x^n)
    using xn by (intro fps-left-inverse-eq-fps-right-inverse[symmetric])
  moreover have fps-right-inverse a x = fps-left-inverse a x
    using assms by (intro fps-left-inverse-eq-fps-right-inverse[symmetric])
  ultimately show fps-right-inverse (a^n) (x^n) = fps-right-inverse a x ^ n
    by simp

qed

lemma fps-inverse-power:
  fixes a :: 'a::division-ring fps
  shows inverse (a^n) = inverse a ^ n
proof (cases n=0 a$0 = 0 rule: case-split[case-product case-split])
  case False-True
  hence LHS: inverse (a^n) = 0 and RHS: inverse a ^ n = 0
    by (simp-all add: startsby-zero-power)
  show ?thesis using trans-sym[OF LHS RHS] by fast
next
  case False-False
  from False-False(2) show ?thesis
    by  (simp add:
       fps-inverse-def fps-power-zeroth power-inverse fps-lr-inverse-power(2)[symmetric]
        )
qed auto

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 (cases n)
  case (Suc m)
  moreover have fps-deriv (a^Suc m) = of-nat (Suc m) * fps-deriv a * a^m
    by (induct m) (simp-all add: algebra-simps)
  ultimately show ?thesis by simp
qed simp

lemma fps-deriv-power:
```

fixes $a :: 'a::comm\text{-}semiring\text{-}1\ fps$
shows $fps\text{-}deriv\ (a\ \hat{}\ n) = fps\text{-}const\ (of\text{-}nat\ n) * fps\text{-}deriv\ a * a\ \hat{}\ (n - 1)$
by (*simp add*: *fps-deriv-power' fps-of-nat*)

5.12 Finite and infinite products

lemma *fps-prod-nth'*:
assumes *finite A*
shows $fps\text{-}nth\ (\prod x{\in}A.\ f\ x)\ n = (\sum X{\in}multisets\text{-}of\text{-}size\ A\ n.\ \prod x{\in}A.\ fps\text{-}nth\ (f\ x)\ (count\ X\ x))$
using *assms*
proof (*induction A arbitrary*: *n rule*: *finite-induct*)
case (*insert a A n*)
note [*simp*] = ‹$a \notin A$›
note [*intro*, *simp*] = ‹*finite A*›
have $(\sum X{\in}multisets\text{-}of\text{-}size\ (insert\ a\ A)\ n.\ \prod x{\in}insert\ a\ A.\ fps\text{-}nth\ (f\ x)\ (count\ X\ x)) =$
$(\sum (m,X){\in}(SIGMA\ m{:}\{0..n\}.\ multisets\text{-}of\text{-}size\ A\ (n{-}m)).$
$\prod x{\in}insert\ a\ A.\ fps\text{-}nth\ (f\ x)\ (count\ (X + replicate\text{-}mset\ m\ a)\ x))$
by (*subst sum.reindex-bij-betw*[*OF bij-betw-multisets-of-size-insert*, *symmetric*]) (*simp-all add*: *case-prod-unfold*)
also have $\ldots = (\sum m{=}0..n.\ \sum X{\in}multisets\text{-}of\text{-}size\ A\ (n{-}m).$
$\prod x{\in}insert\ a\ A.\ fps\text{-}nth\ (f\ x)\ (count\ (X + replicate\text{-}mset\ m\ a)\ x))$
by (*rule sum.Sigma* [*symmetric*]) *auto*
also have $\ldots = (\sum m{=}0..n.\ fps\text{-}nth\ (f\ a)\ m * fps\text{-}nth\ (\prod x{\in}A.\ f\ x)\ (n - m))$
proof (*rule sum.cong*)
fix *m*
assume *m*: $m \in \{0..n\}$
have $(\sum X{\in}multisets\text{-}of\text{-}size\ A\ (n{-}m).$
$\prod x{\in}insert\ a\ A.\ fps\text{-}nth\ (f\ x)\ (count\ (X + replicate\text{-}mset\ m\ a)\ x)) =$
$(\sum X{\in}multisets\text{-}of\text{-}size\ A\ (n{-}m).\ fps\text{-}nth\ (f\ a)\ (count\ X\ a + m) *$
$(\prod x{\in}A.\ fps\text{-}nth\ (f\ x)\ (count\ (X + replicate\text{-}mset\ m\ a)\ x)))$
by *simp*
also have $\ldots = (\sum X{\in}multisets\text{-}of\text{-}size\ A\ (n{-}m).\ fps\text{-}nth\ (f\ a)\ m *$
$(\prod x{\in}A.\ fps\text{-}nth\ (f\ x)\ (count\ (X + replicate\text{-}mset\ m\ a)\ x)))$
by (*intro sum.cong arg-cong2*[*of - - - - (*)*] *arg-cong2*[*of - - - - fps-nth*] *refl*) (*auto simp*: *multisets-of-size-def simp flip*: *not-in-iff*)
also have $\ldots = fps\text{-}nth\ (f\ a)\ m * (\sum X{\in}multisets\text{-}of\text{-}size\ A\ (n{-}m).$
$(\prod x{\in}A.\ fps\text{-}nth\ (f\ x)\ (count\ (X + replicate\text{-}mset\ m\ a)\ x)))$
by (*simp add*: *sum-distrib-left*)
also have $(\sum X{\in}multisets\text{-}of\text{-}size\ A\ (n{-}m).\ \prod x{\in}A.\ fps\text{-}nth\ (f\ x)\ (count\ (X + replicate\text{-}mset\ m\ a)\ x)) =$
$(\sum X{\in}multisets\text{-}of\text{-}size\ A\ (n{-}m).\ \prod x{\in}A.\ fps\text{-}nth\ (f\ x)\ (count\ X\ x))$
by (*intro sum.cong prod.cong*) *auto*
also have $\ldots = fps\text{-}nth\ (\prod x{\in}A.\ f\ x)\ (n - m)$
by (*rule insert.IH* [*symmetric*])
finally show $(\sum X{\in}multisets\text{-}of\text{-}size\ A\ (n{-}m).\ \prod x{\in}insert\ a\ A.\ fps\text{-}nth\ (f\ x)\ (count\ (X + replicate\text{-}mset\ m\ a)\ x)) =$
$fps\text{-}nth\ (f\ a)\ m * fps\text{-}nth\ (\prod x{\in}A.\ f\ x)\ (n - m)$.

```
  qed auto
  also have ... = fps-nth (∏ x∈insert a A. f x) n
    by (simp add: fps-mult-nth)
  finally show ?case ..
qed auto

theorem tendsto-prod-fps:
  fixes f :: nat ⇒ 'a :: {idom, t2-space} fps
  assumes [simp]: ⋀k. f k ≠ 0
  assumes g: ⋀n k. k > g n ⟹ subdegree (f k - 1) > n
  defines P ≡ Abs-fps (λn. (∑ X∈multisets-of-size {..g n} n. ∏ i≤g n. fps-nth (f
i) (count X i)))
  shows   (λn. ∏ k≤n. f k) ⟶ P
proof (rule tendsto-fpsI)
  fix n :: nat
  show eventually (λN. fps-nth (prod f {..N}) n = fps-nth P n) at-top
    using eventually-ge-at-top[of g n]
  proof eventually-elim
    case (elim N)
     have fps-nth (prod f {..N}) n = (∑ X∈multisets-of-size {..N} n. ∏ x≤N.
fps-nth (f x) (count X x))
      by (subst fps-prod-nth') auto
    also have ... = (∑ X | X ∈ multisets-of-size {..N} n ∧ (∀ x≤N. fps-nth (f x)
(count X x) ≠ 0).
                    ∏ x≤N. fps-nth (f x) (count X x))
      by (intro sum.mono-neutral-right) auto

    also have {X. X ∈ multisets-of-size {..N} n ∧ (∀ x≤N. fps-nth (f x) (count X
x) ≠ 0)} =
              {X. X ∈ multisets-of-size {..g n} n ∧ (∀ x≤N. fps-nth (f x) (count X
x) ≠ 0)}
      (is ?lhs = ?rhs)
    proof (intro equalityI subsetI)
      fix X assume X ∈ ?rhs
      thus X ∈ ?lhs using elim multisets-of-size-mono[of {..g n} {..N}] by auto
    next
      fix X assume X ∈ ?lhs
     hence X: set-mset X ⊆ {..N} size X = n ⋀x. x ≤ N ⟹ fps-nth (f x) (count
X x) ≠ 0
        by (auto simp: multisets-of-size-def)
      have set-mset X ⊆ {..g n}
      proof
        fix x assume *: x ∈ set-mset X
        show x ∈ {..g n}
        proof (rule ccontr)
          assume x ∉ {..g n}
          hence x: x > g n x ≤ N
            using X(1) * by auto
          have count X x ≤ n
```

using X x *count-le-size*[*of* X x] **by** (*auto simp*: *Pi-def*)
also have $n < subdegree\ (f\ x - 1)$
by (*rule g*) (*use x* **in** *auto*)
finally have *fps-nth* $(f\ x - 1)$ (*count* X x) $= 0$
by *blast*
hence *fps-nth* $(f\ x)$ (*count* X x) $= 0$
using $*$ **by** *simp*
moreover have *fps-nth* $(f\ x)$ (*count* X x) $\neq 0$
by (*intro X*(*3*)) (*use x* **in** *auto*)
ultimately show *False* **by** *contradiction*
qed
qed
thus $X \in$ *?rhs* **using** X
by (*auto simp*: *multisets-of-size-def*)
qed

also have $(\sum X \mid X \in$ *multisets-of-size* $\{..g\ n\}\ n \wedge (\forall x{\leq}N.$ *fps-nth* $(f\ x)$ (*count* $X\ x) \neq 0).$
$\prod x{\leq}N.$ *fps-nth* $(f\ x)$ (*count* $X\ x)) =$
$(\sum X \mid X \in$ *multisets-of-size* $\{..g\ n\}\ n \wedge (\forall x{\leq}N.$ *fps-nth* $(f\ x)$ (*count* $X\ x) \neq 0).$
$\prod i{\leq}g\ n.$ *fps-nth* $(f\ i)$ (*count* $X\ i))$
proof (*intro sum.cong prod.mono-neutral-right ballI*)
fix $X\ i$
assume *i*: $i \in \{..N\} - \{..g\ n\}$
assume $X \in \{X.\ X \in$ *multisets-of-size* $\{..g\ n\}\ n \wedge (\forall x{\leq}N.$ *fps-nth* $(f\ x)$ $(count\ X\ x) \neq 0)\}$
hence *h*: $X \in$ *multisets-of-size* $\{..g\ n\}\ n \bigwedge x.\ x \leq N \Longrightarrow$ *fps-nth* $(f\ x)$ (*count* $X\ x) \neq 0$
by *blast*+
have $i \notin\# X$
using $i\ h$ **unfolding** *multisets-of-size-def* **by** *auto*
have $n < subdegree\ (f\ i - 1)$
by (*intro g*) (*use i* **in** *auto*)
moreover have *count* $X\ i \leq n$
using $\langle i \notin\# X\rangle$ **by** (*simp add*: *not-in-iff*)
ultimately have *fps-nth* $(f\ i - 1)$ (*count* $X\ i) = 0$
by (*intro nth-less-subdegree-zero*) *auto*
thus *fps-nth* $(f\ i)$ (*count* $X\ i) = 1$
using *h*(*2*) $i\ \langle i \notin\# X\rangle$ **by** (*auto split*: *if-splits*)
qed (*use elim* **in** *auto*)

also have $(\sum X \mid X \in$ *multisets-of-size* $\{..g\ n\}\ n \wedge (\forall x{\leq}N.$ *fps-nth* $(f\ x)$ (*count* $X\ x) \neq 0).$
$\prod i{\leq}g\ n.$ *fps-nth* $(f\ i)$ (*count* $X\ i)) =$
$(\sum X \in$ *multisets-of-size* $\{..g\ n\}\ n.\ \prod i{\leq}g\ n.$ *fps-nth* $(f\ i)$ (*count* $X\ i))$
proof (*intro sum.mono-neutral-left ballI*)
fix X **assume** $X \in$ *multisets-of-size* $\{..g\ n\}\ n -$
$\{X{\in}$*multisets-of-size* $\{..g\ n\}\ n.\ \forall x{\leq}N.$ *fps-nth* $(f\ x)$ (*count* $X\ x)$

```
≠ 0}
      then obtain i
        where h: X ∈ multisets-of-size {..g n} n
        and i: i ≤ N fps-nth (f i) (count X i) = 0
        by blast
      have ¬(i > g n)
      proof
        assume i': i > g n
        hence count X i = 0
          using h by (auto simp: multisets-of-size-def simp flip: not-in-iff)
        have subdegree (f i − 1) > n
          by (intro g) (use i' in auto)
        hence subdegree (f i − 1) > 0
          by simp
        hence fps-nth (f i − 1) 0 = 0
          by blast
        hence fps-nth (f i) (count X i) = 1
          using ‹count X i = 0› by simp
        thus False using i by simp
      qed
      thus  (∏ x≤g n. fps-nth (f x) (count X x)) = 0
        using i by auto
    qed auto

    also have ... = fps-nth P n
      by (simp add: P-def)
    finally show fps-nth (∏ k≤N. f k) n = fps-nth P n .
  qed
qed
```

5.13 Integration

```
definition fps-integral :: 'a::{semiring-1,inverse} fps ⇒ 'a ⇒ 'a fps
  where fps-integral a a0 =
        Abs-fps (λn. 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
  by    (auto simp: fps-integral-def)

lemma fps-integral-conv-plus-const:
  fps-integral a a0 = fps-integral a 0 + fps-const a0
  unfolding fps-integral-def by (intro fps-ext) simp

lemma fps-deriv-fps-integral:
```

fixes $a :: 'a::\{division\text{-}ring,ring\text{-}char\text{-}0\}$ *fps*
shows *fps-deriv* (*fps-integral a a0*) = *a*
proof (*intro fps-ext*)
fix *n*
have (*of-nat* (*Suc n*) :: *'a*) ≠ *0* **by** (*rule of-nat-neq-0*)
hence *of-nat* (*Suc n*) ∗ *inverse* (*of-nat* (*Suc n*) :: *'a*) = *1* **by** *simp*
moreover have
fps-deriv (*fps-integral a a0*) $ *n* = *of-nat* (*Suc n*) ∗ *inverse* (*of-nat* (*Suc n*)) ∗ *a* $ *n*
by (*simp add*: *mult.assoc*)
ultimately show *fps-deriv* (*fps-integral a a0*) $ *n* = *a* $ *n* **by** *simp*
qed

lemma *fps-integral0-deriv*:
fixes $a :: 'a::\{division\text{-}ring,ring\text{-}char\text{-}0\}$ *fps*
shows *fps-integral0* (*fps-deriv a*) = *a* − *fps-const* (*a$0*)
proof (*intro fps-ext*)
fix *n*
show *fps-integral0* (*fps-deriv a*) $ *n* = (*a* − *fps-const* (*a$0*)) $ *n*
proof (*cases n*)
case (*Suc m*)
have (*of-nat* (*Suc m*) :: *'a*) ≠ *0* **by** (*rule of-nat-neq-0*)
hence *inverse* (*of-nat* (*Suc m*) :: *'a*) ∗ *of-nat* (*Suc m*) = *1* **by** *simp*
moreover have
fps-integral0 (*fps-deriv a*) $ *Suc m* =
inverse (*of-nat* (*Suc m*)) ∗ *of-nat* (*Suc m*) ∗ *a* $ (*Suc m*)
by (*simp add*: *mult.assoc*)
ultimately show *?thesis* **using** *Suc* **by** *simp*
qed *simp*
qed

lemma *fps-integral-deriv*:
fixes $a :: 'a::\{division\text{-}ring,ring\text{-}char\text{-}0\}$ *fps*
shows *fps-integral* (*fps-deriv a*) (*a$0*) = *a*
using *fps-integral-conv-plus-const*[*of fps-deriv a a$0*]
by (*simp add*: *fps-integral0-deriv*)

lemma *fps-integral0-zero*:
fps-integral0 (*0*::*'a*::\{*semiring-1*,*inverse*\} *fps*) = *0*
by (*intro fps-ext*) (*simp add*: *fps-integral-def*)

lemma *fps-integral0-fps-const'*:
fixes $c :: 'a::\{semiring\text{-}1,inverse\}$
assumes *inverse* (*1*::*'a*) = *1*
shows *fps-integral0* (*fps-const c*) = *fps-const c* ∗ *fps-X*
proof (*intro fps-ext*)
fix *n*
show *fps-integral0* (*fps-const c*) $ *n* = (*fps-const c* ∗ *fps-X*) $ *n*
by (*cases n*) (*simp-all add*: *assms mult-delta-right*)

qed

lemma *fps-integral0-fps-const*:
fixes *c* :: *'a*::*division-ring*
shows *fps-integral0* (*fps-const c*) = *fps-const c* ∗ *fps-X*
by (*rule fps-integral0-fps-const'*[*OF inverse-1*])

lemma *fps-integral0-one'*:
assumes *inverse* (*1*::*'a*::{*semiring-1*,*inverse*}) = *1*
shows *fps-integral0* (*1*::*'a fps*) = *fps-X*
using *assms fps-integral0-fps-const'*[*of 1*::*'a*]
by *simp*

lemma *fps-integral0-one*:
fps-integral0 (*1*::*'a*::*division-ring fps*) = *fps-X*
by (*rule fps-integral0-one'*[*OF inverse-1*])

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 (*intro fps-ext*)
fix *n*
show *fps-integral0* (*fps-const c* ∗ *a*) $ *n* = (*fps-const c* ∗ *fps-integral0 a*) $ *n*
using *mult-inverse-of-nat-commute*[*of n c*, *symmetric*]
mult.assoc[*of inverse* (*of-nat n*) *c a*$(*n*−*1*)]
mult.assoc[*of c inverse* (*of-nat n*) *a*$(*n*−*1*)]
by (*simp add*: *fps-integral-def*)
qed

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*
by (*intro fps-ext*) (*simp add*: *fps-integral-def algebra-simps*)

lemma *fps-integral0-neg*:
fixes *a* :: *'a*::{*ring-1*,*inverse*} *fps*
shows *fps-integral0* (−*a*) = − *fps-integral0 a*
using *fps-integral0-fps-const-mult-right*[*of a* −*1*]
by (*simp add*: *fps-const-neg*[*symmetric*])

lemma *fps-integral0-add*:
fps-integral0 (*a*+*b*) = *fps-integral0 a* + *fps-integral0 b*
by (*intro fps-ext*) (*simp add*: *fps-integral-def algebra-simps*)

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*
by (*simp add*: *fps-integral0-add fps-integral0-fps-const-mult-left*)

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*
by (*simp add*: *fps-integral0-add fps-integral0-fps-const-mult-right*)

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*
using *fps-integral-conv-plus-const*[*of*
fps-const a ∗ *f* + *fps-const b* ∗ *g*
a∗*a0* + *b*∗*b0*
]
fps-integral-conv-plus-const[*of f a0*] *fps-integral-conv-plus-const*[*of g b0*]
by (*simp add*: *fps-integral0-linear algebra-simps*)

lemma *fps-integral0-sub*:
fixes *a b* :: *'a*::{*ring-1*,*inverse*} *fps*
shows *fps-integral0* (*a*−*b*) = *fps-integral0 a* − *fps-integral0 b*
using *fps-integral0-linear2*[*of a 1 b* −*1*]
by (*simp add*: *fps-const-neg*[*symmetric*])

lemma *fps-integral0-of-nat*:
fps-integral0 (*of-nat n* :: *'a*::*division-ring fps*) = *of-nat n* ∗ *fps-X*
using *fps-integral0-fps-const*[*of of-nat n* :: *'a*] **by** (*simp add*: *fps-of-nat*)

lemma *fps-integral0-sum*:
fps-integral0 (*sum f S*) = *sum* (λ*i*. *fps-integral0* (*f i*)) *S*
proof (*cases finite S*)
case *True* **show** *?thesis*
by (*induct rule*: *finite-induct* [*OF True*])
(*simp-all add*: *fps-integral0-zero fps-integral0-add*)
qed (*simp add*: *fps-integral0-zero*)

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−
have *fps-integral0* (*fps-deriv* (*a* ∗ *fps-integral0 b*)) = *a* ∗ *fps-integral0 b*
using *fps-integral0-deriv*[*of* (*a* ∗ *fps-integral0 b*)] **by** *simp*
moreover have
fps-integral0 (*a* ∗ *b*) =
fps-integral0 (*fps-deriv* (*a* ∗ *fps-integral0 b*)) −
fps-integral0 (*fps-deriv a* ∗ *fps-integral0 b*)
by (*auto simp*: *fps-deriv-fps-integral fps-integral0-sub*[*symmetric*])

ultimately show *?thesis* **by** *simp*
qed

lemma *fps-integral0-fps-X*:
fps-integral0 (*fps-X*::*'a*::{*semiring-1*,*inverse*} *fps*) =
fps-const (*inverse* (*of-nat* 2)) ∗ *fps-X*^{2}
by (*intro fps-ext*) (*auto simp*: *fps-integral-def*)

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 (*intro fps-ext*)
fix *k* **show**
fps-integral0 ((*fps-X*::*'a fps*) ^ *n*) $ *k* =
(*fps-const* (*inverse* (*of-nat* (*Suc n*))) ∗ *fps-X* ^ *Suc n*) $ *k*
by (*cases k*) *simp-all*
qed

5.14 Composition

definition *fps-compose* :: *'a*::*semiring-1 fps* ⇒ *'a fps* ⇒ *'a fps* (**infixl** ‹*oo*› *55*)
where *a oo b* = *Abs-fps* (λ*n*. *sum* (λ*i*. *a*$*i* ∗ (*b*^*i*$*n*)) {*0*..*n*})

lemma *fps-compose-nth*: (*a oo b*)$*n* = *sum* (λ*i*. *a*$*i* ∗ (*b*^*i*$*n*)) {*0*..*n*}
by (*simp add*: *fps-compose-def*)

lemma *fps-compose-nth-0* [*simp*]: (*f oo g*) $ *0* = *f* $ *0*
by (*simp add*: *fps-compose-nth*)

lemma *fps-compose-fps-X*[*simp*]: *a oo fps-X* = (*a* :: *'a*::*comm-ring-1 fps*)
by (*simp add*: *fps-ext fps-compose-def mult-delta-right*)

lemma *fps-const-compose*[*simp*]: *fps-const* (*a*::*'a*::*comm-ring-1*) *oo b* = *fps-const a*
by (*simp add*: *fps-eq-iff fps-compose-nth mult-delta-left*)

lemma *numeral-compose*[*simp*]: (*numeral k* :: *'a*::*comm-ring-1 fps*) *oo b* = *numeral k*
unfolding *numeral-fps-const* **by** *simp*

lemma *neg-numeral-compose*[*simp*]: (− *numeral k* :: *'a*::*comm-ring-1 fps*) *oo b* = − *numeral k*
unfolding *neg-numeral-fps-const* **by** *simp*

lemma *fps-X-fps-compose-startby0*[*simp*]: *a*$*0* = *0* ⟹ *fps-X oo a* = (*a* :: *'a*::*comm-ring-1 fps*)
by (*simp add*: *fps-eq-iff fps-compose-def mult-delta-left not-le*)

5.15 Rules from Herbert Wilf's Generatingfunctionology

5.15.1 Rule 1

lemma *fps-power-mult-eq-shift*:
*fps-X^Suc k * Abs-fps (λn. a (n + Suc k)) =*
*Abs-fps a − sum (λi. fps-const (a i :: 'a::comm-ring-1) * fps-X^i) {0 .. k}*
(**is** *?lhs = ?rhs*)
proof −
have *?lhs $ n = ?rhs $ n* **for** *n :: nat*
proof −
have *?lhs $ n = (if n < Suc k then 0 else a n)*
unfolding *fps-X-power-mult-nth* **by** *auto*
also have ... = *?rhs $ n*
proof (*induct k*)
case *0*
then show *?case*
by (*simp add*: *fps-sum-nth*)
next
case (*Suc k*)
have (*Abs-fps a − sum (λi. fps-const (a i :: 'a) * fps-X^i) {0 .. Suc k}*)*$n =*
(*Abs-fps a − sum (λi. fps-const (a i :: 'a) * fps-X^i) {0 .. k} −*
*fps-const (a (Suc k)) * fps-X^ Suc k) $ n*
by (*simp add*: *field-simps*)
also have ... = (*if n < Suc k then 0 else a n*) − (*fps-const (a (Suc k)) * fps-X^ Suc k*)*$n*
using *Suc.hyps*[*symmetric*] **unfolding** *fps-sub-nth* **by** *simp*
also have ... = (*if n < Suc (Suc k) then 0 else a n*)
unfolding *fps-X-power-mult-right-nth*
by (*simp add*: *not-less le-less-Suc-eq*)
finally show *?case*
by *simp*
qed
finally show *?thesis* **.**
qed
then show *?thesis*
by (*simp add*: *fps-eq-iff*)
qed

5.15.2 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)*
by (*simp add*: *fps-XD-def field-simps*)

lemma *fps-XD-mult-const*[*simp*]:*fps-XD (fps-const (c::'a::comm-ring-1) * a) = fps-const c * fps-XD a*
by (*simp add*: *fps-XD-def field-simps*)

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*)
by *simp*

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*)
by (*induct n*) *simp-all*

lemma *fps-mult-fps-X-deriv-shift*: *fps-X*∗ *fps-deriv a* = *Abs-fps* (λ*n*. *of-nat n*∗ *a*$*n*)
by (*simp add*: *fps-eq-iff*)

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*)
by (*induct k arbitrary*: *a*) (*simp-all add*: *fps-XD-def fps-eq-iff field-simps del*: *One-nat-def*)

5.15.3 Rule 3

Rule 3 is trivial and is given by `fps_times_def`.

5.15.4 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 (*rule fps-ext*)
define *f g* :: *'a fps*
where *f* ≡ *1* − *fps-X*
and *g* ≡ *Abs-fps* (λ*n*. *sum* (λ*i*. *a* $ *i*) {*0*..*n*})
fix *n* **show** *a* $ *n*= (*f* ∗ *g*) $ *n*
proof (*cases n*)
case (*Suc m*)
hence (*f* ∗ *g*) $ *n* = *g* $ *Suc m* − *g* $ *m*
using *fps-mult-nth*[*of f g Suc m*]
sum.atLeast-Suc-atMost[*of 0 Suc m* λ*i*. *f* $ *i* ∗ *g* $ (*Suc m* − *i*)]
sum.atLeast-Suc-atMost[*of 1 Suc m* λ*i*. *f* $ *i* ∗ *g* $ (*Suc m* − *i*)]
by (*simp add*: *f-def*)
with *Suc* **show** *?thesis* **by** (*simp add*: *g-def*)
qed (*simp add*: *f-def g-def*)
qed

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−
from *assms* **have** *a* /((*1*::*'a fps*) − *fps-X*) = *a* ∗ *Abs-fps* (λ*n*. *1*)
by (*simp add*: *fps-divide-def fps-inverse-def fps-lr-inverse-one-minus-fps-X*(*2*))

thus *?thesis* **by** (*auto intro*: *fps-ext simp*: *fps-mult-nth*)
qed

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*})
using *fps-divide-fps-X-minus1-sum-ring1*[*of a*] **by** *simp*

5.15.5 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 −
have ⟦*length xs* = *1*; *n* = *sum-list xs*⟧ ⟹ *xs* = [*sum-list xs*] **for** *xs*
by (*cases xs*) *auto*
then show *?thesis*
by (*auto simp add*: *natpermute-def*)
qed

lemma *natlist-trivial-Suc0* [*simp*]: *natpermute n* (*Suc 0*) = {[*n*]}
using *natlist-trivial-1* **by** *force*

lemma *append-natpermute-less-eq*:
assumes *xs* @ *ys* ∈ *natpermute n k*
shows *sum-list xs* ≤ *n*
and *sum-list ys* ≤ *n*
proof −
from *assms* **have** *sum-list* (*xs* @ *ys*) = *n*
by (*simp add*: *natpermute-def*)
then have *sum-list xs* + *sum-list ys* = *n*
by *simp*
then show *sum-list xs* ≤ *n* **and** *sum-list ys* ≤ *n*
by *simp-all*
qed

lemma *natpermute-split*:
assumes *h* ≤ *k*
shows *natpermute n k* =
(⋃ *m* ∈{0..*n*}. {*l1* @ *l2* |*l1 l2*. *l1* ∈ *natpermute m h* ∧ *l2* ∈ *natpermute* (*n* − *m*) (*k* − *h*)})
(**is** *?L* = *?R* **is** - = (⋃ *m* ∈{0..*n*}. *?S m*))
proof
show *?R* ⊆ *?L*
proof
fix *l*
assume *l*: *l* ∈ *?R*
from *l* **obtain** *m xs ys* **where** *h*: *m* ∈ {0..*n*}

```
    and xs: xs ∈ natpermute m h
    and ys: ys ∈ natpermute (n − m) (k − h)
    and leq: l = xs@ys by blast
  from xs have xs': sum-list xs = m
    by (simp add: natpermute-def)
  from ys have ys': sum-list ys = n − m
    by (simp add: natpermute-def)
  show l ∈ ?L using leq xs ys h
    using assms by (force simp add: natpermute-def)
 qed
 show ?L ⊆ ?R
 proof
  fix l
  assume l: l ∈ natpermute n k
  let ?xs = take h l
  let ?ys = drop h l
  let ?m = sum-list ?xs
  from l have ls: sum-list (?xs @ ?ys) = n
    by (simp add: natpermute-def)
  have xs: ?xs ∈ natpermute ?m h using l assms
    by (simp add: natpermute-def)
  have l-take-drop: sum-list l = sum-list (take h l @ drop h l)
    by simp
  then have ys: ?ys ∈ natpermute (n − ?m) (k − h)
   using l assms ls by (auto simp add: natpermute-def simp del: append-take-drop-id)
  from ls have m: ?m ∈ {0..n}
    by (simp add: l-take-drop del: append-take-drop-id)
  have sum-list (take h l) ≤ sum-list l
    using l-take-drop ls m by presburger
  with xs ys ls l show l ∈ ?R
    by simp (metis append-take-drop-id m)
 qed
qed

lemma natpermute-0: natpermute n 0 = (if n = 0 then {[]} else {})
 by (auto simp add: natpermute-def)

lemma natpermute-0'[simp]: natpermute 0 k = (if k = 0 then {[]} else {replicate
k 0})
 by (auto simp add: set-replicate-conv-if natpermute-def replicate-length-same)

lemma natpermute-finite: finite (natpermute n k)
proof (induct k arbitrary: n)
 case 0
 then show ?case
   by (simp add: natpermute-0)
next
 case (Suc k)
 then show ?case
```

```
    using natpermute-split [of k Suc k] finite-UN-I by simp
qed

lemma natpermute-contain-maximal:
  {xs ∈ natpermute n (k + 1). n ∈ set xs} = (⋃ i∈{0 .. k}. {(replicate (k + 1) 0)
[i:=n]})
  (is ?A = ?B)
proof
  show ?A ⊆ ?B
  proof
    fix xs
    assume xs ∈ ?A
    then have H: xs ∈ natpermute n (k + 1) and n: n ∈ set xs
      by blast+
    then obtain i where i: i ∈ {0.. k} xs!i = n
      unfolding in-set-conv-nth by (auto simp add: less-Suc-eq-le natpermute-def)
    have eqs: ({0..k} − {i}) ∪ {i} = {0..k}
      using i by auto
    have f: finite({0..k} − {i}) finite {i}
      by auto
    have d: ({0..k} − {i}) ∩ {i} = {}
      using i by auto
    from H have n = sum (nth xs) {0..k}
    by (auto simp add: natpermute-def atLeastLessThanSuc-atLeastAtMost sum-list-sum-nth)
    also have ... = n + sum (nth xs) ({0..k} − {i})
      unfolding sum.union-disjoint[OF f d, unfolded eqs] using i by simp
    finally have zxs: ∀ j∈ {0..k} − {i}. xs!j = 0
      by auto
    from H have xsl: length xs = k+1
      by (simp add: natpermute-def)
    from i have i': i < length (replicate (k+1) 0)   i < k+1
      unfolding length-replicate by presburger+
    have xs = (replicate (k+1) 0) [i := n]
    proof (rule nth-equalityI)
      show length xs = length ((replicate (k + 1) 0)[i := n])
        by (metis length-list-update length-replicate xsl)
      show xs ! j = (replicate (k + 1) 0)[i := n] ! j if j < length xs for j
      proof (cases j = i)
        case True
        then show ?thesis
          by (metis i'(1) i(2) nth-list-update)
      next
        case False
        with that show ?thesis
          by (simp add: xsl zxs del: replicate.simps split: nat.split)
      qed
    qed
    then show xs ∈ ?B using i by blast
  qed
```

```
  show ?B ⊆ ?A
  proof
    fix xs
    assume xs ∈ ?B
    then obtain i where i: i ∈ {0..k} and xs: xs = (replicate (k + 1) 0) [i:=n]
      by auto
    have nxs: n ∈ set xs
      unfolding xs using set-update-memI i
     by (metis Suc-eq-plus1 atLeast0AtMost atMost-iff le-simps(2) length-replicate)
    have xsl: length xs = k + 1
      by (simp only: xs length-replicate length-list-update)
    have sum-list xs = sum (nth xs) {0..<k+1}
      unfolding sum-list-sum-nth xsl ..
    also have ... = sum (λj. if j = i then n else 0) {0..< k+1}
      by (rule sum.cong) (simp-all add: xs del: replicate.simps)
    also have ... = n using i by simp
    finally have xs ∈ natpermute n (k + 1)
      using xsl unfolding natpermute-def mem-Collect-eq by blast
    then show xs ∈ ?A
      using nxs by blast
  qed
qed
```

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 (λv. prod (λj. (a j) $ (v!j)) {0..m}) (natpermute n (m+1))
  (is ?P m n)
proof (induct m arbitrary: n rule: nat-less-induct)
  fix m n assume H: ∀ m' < m. ∀ n. ?P m' n
  show ?P m n
  proof (cases m)
    case 0
    then show ?thesis
      by simp
  next
    case (Suc k)
    then have km: k < m by arith
    have u0: {0 .. k} ∪ {m} = {0..m}
      using Suc by (simp add: set-eq-iff) presburger
    have f0: finite {0 .. k} finite {m} by auto
    have d0: {0 .. k} ∩ {m} = {} using Suc by auto
    have (prod a {0 .. m}) $ n = (prod a {0 .. k} * a m) $ n
      unfolding prod.union-disjoint[OF f0 d0, unfolded u0] by simp
    also have ... = (∑ i = 0..n. (∑ v∈natpermute i (k + 1).
          (∏ j = 0..k. a j $ v ! j) * a m $ (n − i)))
      unfolding fps-mult-nth H[rule-format, OF km] sum-distrib-right ..
```

also have ... = ($\sum i = 0..n.$
$\sum v{\in}(\lambda l1.\ l1\ @\ [n - i])\ `\ natpermute\ i\ (Suc\ k).$
$(\prod j = 0..k.\ a\ j\ \$\ v\ !\ j) * a\ (Suc\ k)\ \$\ v\ !\ Suc\ k)$
by (*intro sum.cong* [*OF refl sym*] *sum.reindex-cong*) (*auto simp*: *inj-on-def natpermute-def nth-append Suc*)
also have ... = ($\sum v{\in}(\bigcup x{\in}\{0..n\}.\ \{l1\ @\ [n - x]\ |l1.\ l1 \in natpermute\ x\ (Suc\ k)\}$).
$(\prod j = 0..k.\ a\ j\ \$\ v\ !\ j) * a\ (Suc\ k)\ \$\ v\ !\ Suc\ k)$
by (*subst sum.UNION-disjoint*) (*auto simp add*: *natpermute-finite setcompr-eq-image*)
also have ... = ($\sum v{\in}natpermute\ n\ (m + 1).\ \prod j{\in}\{0..m\}.\ a\ j\ \$\ v\ !\ j$)
using *natpermute-split*[*of m m + 1*] **by** (*simp add*: *Suc*)
finally show *?thesis* **.**
qed
qed

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 −
have *th0*: *a*^*Suc m* = *prod* (λ*i*. *a*) {*0..m*}
by (*simp add*: *prod-constant*)
show *?thesis* **unfolding** *th0 fps-prod-nth* **..**
qed

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*))
by (*cases m*) (*simp-all add*: *fps-power-nth-Suc del*: *power-Suc*)

lemmas *fps-nth-power-0* = *fps-power-zeroth*

lemma *natpermute-max-card*:
assumes *n0*: *n* ≠ *0*
shows *card* {*xs* ∈ *natpermute n* (*k* + *1*). *n* ∈ *set xs*} = *k* + *1*
unfolding *natpermute-contain-maximal*
proof −
let *?A* = λ*i*. {(*replicate* (*k* + *1*) *0*)[*i* := *n*]}
let *?K* = {*0* ..*k*}
have *fK*: *finite ?K*
by *simp*
have *fAK*: ∀ *i*∈*?K*. *finite* (*?A i*)
by *auto*
have *d*: ∀ *i*∈ *?K*. ∀ *j*∈ *?K*. *i* ≠ *j* ⟶

```
    {(replicate (k + 1) 0)[i := n]} ∩ {(replicate (k + 1) 0)[j := n]} = {}
  proof clarify
    fix i j
    assume i: i ∈ ?K and j: j ∈ ?K and ij: i ≠ j
    have False if eq: (replicate (k+1) 0)[i:=n] = (replicate (k+1) 0)[j:= n]
    proof −
      have (replicate (k+1) 0) [i:=n] ! i = n
        using i by (simp del: replicate.simps)
      moreover
      have (replicate (k+1) 0) [j:=n] ! i = 0
        using i ij by (simp del: replicate.simps)
      ultimately show ?thesis
        using eq n0 by (simp del: replicate.simps)
    qed
    then show {(replicate (k + 1) 0)[i := n]} ∩ {(replicate (k + 1) 0)[j := n]} =
{}
      by auto
  qed
  from card-UN-disjoint[OF fK fAK d]
  show card (⋃ i∈{0..k}. {(replicate (k + 1) 0)[i := n]}) = k + 1
    by simp
qed

lemma fps-power-Suc-nth:
  fixes f :: 'a :: comm-ring-1 fps
  assumes k: k > 0
  shows (f ^ Suc m) $ k =
          of-nat (Suc m) * (f $ k * (f $ 0) ^ m) +
          (∑ v∈{v∈natpermute k (m+1). k ∉ set v}. ∏ j = 0..m. f $ v ! j)
proof −
  define A B
    where A = {v∈natpermute k (m+1). k ∈ set v}
      and  B = {v∈natpermute k (m+1). k ∉ set v}
  have [simp]: finite A finite B A ∩ B = {} by (auto simp: A-def B-def natper-
mute-finite)

  from natpermute-max-card[of k m] k have card-A: card A = m + 1 by (simp
add: A-def)
  {
    fix v assume v: v ∈ A
    from v have [simp]: length v = Suc m by (simp add: A-def natpermute-def)
    from v have ∃j. j ≤ m ∧ v ! j = k
      by (auto simp: set-conv-nth A-def natpermute-def less-Suc-eq-le)
    then obtain j where j: j ≤ m v ! j = k by auto

    from v have k = sum-list v by (simp add: A-def natpermute-def)
    also have ... = (∑ i=0..m. v ! i)
    by (simp add: sum-list-sum-nth atLeastLessThanSuc-atLeastAtMost del: sum.op-ivl-Suc)
    also from j have {0..m} = insert j ({0..m}−{j}) by auto
```

```
    also from j have (∑ i∈.... v ! i) = k + (∑ i∈{0..m}−{j}. v ! i)
      by (subst sum.insert) simp-all
    finally have (∑ i∈{0..m}−{j}. v ! i) = 0 by simp
    hence zero: v ! i = 0 if i ∈ {0..m}−{j} for i using that
      by (subst (asm) sum-eq-0-iff) auto

    from j have {0..m} = insert j ({0..m} − {j}) by auto
    also from j have (∏ i∈.... f $ (v ! i)) = f $ k * (∏ i∈{0..m} − {j}. f $ (v !
i))
      by (subst prod.insert) auto
    also have (∏ i∈{0..m} − {j}. f $ (v ! i)) = (∏ i∈{0..m} − {j}. f $ 0)
      by (intro prod.cong) (simp-all add: zero)
    also from j have ... = (f $ 0) ^ m by (subst prod-constant) simp-all
    finally have (∏ j = 0..m. f $ (v ! j)) = f $ k * (f $ 0) ^ m .
  } note A = this

  have (f ^ Suc m) $ k = (∑ v∈natpermute k (m + 1). ∏ j = 0..m. f $ v ! j)
    by (rule fps-power-nth-Suc)
  also have natpermute k (m+1) = A ∪ B unfolding A-def B-def by blast
  also have (∑ v∈.... ∏ j = 0..m. f $ (v ! j)) =
              (∑ v∈A. ∏ j = 0..m. f $ (v ! j)) + (∑ v∈B. ∏ j = 0..m. f $ (v ! j))
    by (intro sum.union-disjoint) simp-all
  also have (∑ v∈A. ∏ j = 0..m. f $ (v ! j)) = of-nat (Suc m) * (f $ k * (f $ 0)
^ m)
    by (simp add: A card-A)
  finally show ?thesis by (simp add: B-def)
qed

lemma fps-power-Suc-eqD:
  fixes f g :: 'a :: {idom,semiring-char-0} fps
  assumes f ^ Suc m = g ^ Suc m f $ 0 = g $ 0 f $ 0 ≠ 0
  shows   f = g
proof (rule fps-ext)
  fix k :: nat
  show f $ k = g $ k
  proof (induction k rule: less-induct)
    case (less k)
    show ?case
    proof (cases k = 0)
      case False
      let ?h = λf. (∑ v | v ∈ natpermute k (m + 1) ∧ k ∉ set v. ∏ j = 0..m. f $
v ! j)
      from False fps-power-Suc-nth[of k f m] fps-power-Suc-nth[of k g m]
        have f $ k * (of-nat (Suc m) * (f $ 0) ^ m) + ?h f =
              g $ k * (of-nat (Suc m) * (f $ 0) ^ m) + ?h g using assms
        by (simp add: mult-ac del: power-Suc of-nat-Suc)
      also have v ! i < k if v ∈ {v∈natpermute k (m+1). k ∉ set v} i ≤ m for v i
        using that elem-le-sum-list[of i v] unfolding natpermute-def
        by (auto simp: set-conv-nth dest!: spec[of - i])
```

hence *?h f = ?h g*
by (*intro sum.cong refl prod.cong less lessI*) (*simp add*: *natpermute-def*)
finally have *f $ k * (of-nat (Suc m) * (f $ 0) ^ m) = g $ k * (of-nat (Suc m) * (f $ 0) ^ m)*
by *simp*
with *assms* **show** *f $ k = g $ k*
by (*subst* (*asm*) *mult-right-cancel*) (*auto simp del*: *of-nat-Suc*)
qed (*simp-all add*: *assms*)
qed
qed

lemma *fps-power-Suc-eqD'*:
fixes *f g* :: *'a* :: {*idom,semiring-char-0*} *fps*
assumes *f ^ Suc m = g ^ Suc m f $ subdegree f = g $ subdegree g*
shows *f = g*
proof (*cases f = 0*)
case *False*
have *Suc m * subdegree f = subdegree (f ^ Suc m)*
by (*rule subdegree-power* [*symmetric*])
also have *f ^ Suc m = g ^ Suc m* **by** *fact*
also have *subdegree ... = Suc m * subdegree g* **by** (*rule subdegree-power*)
finally have [*simp*]: *subdegree f = subdegree g*
by (*subst* (*asm*) *Suc-mult-cancel1*)
have *fps-shift* (*subdegree f*) *f * fps-X ^ subdegree f = f*
by (*rule subdegree-decompose* [*symmetric*])
also have *... ^ Suc m = g ^ Suc m* **by** *fact*
also have *g = fps-shift* (*subdegree g*) *g * fps-X ^ subdegree g*
by (*rule subdegree-decompose*)
also have *subdegree f = subdegree g* **by** *fact*
finally have *fps-shift* (*subdegree g*) *f ^ Suc m = fps-shift* (*subdegree g*) *g ^ Suc m*
by (*simp add*: *algebra-simps power-mult-distrib del*: *power-Suc*)
hence *fps-shift* (*subdegree g*) *f = fps-shift* (*subdegree g*) *g*
by (*rule fps-power-Suc-eqD*) (*insert assms False*, *auto*)
with *subdegree-decompose*[*of f*] *subdegree-decompose*[*of g*] **show** *?thesis* **by** *simp*
qed (*insert assms*, *simp-all*)

lemma *fps-power-eqD'*:
fixes *f g* :: *'a* :: {*idom,semiring-char-0*} *fps*
assumes *f ^ m = g ^ m f $ subdegree f = g $ subdegree g m > 0*
shows *f = g*
using *fps-power-Suc-eqD'*[*of f m−1 g*] *assms* **by** *simp*

lemma *fps-power-eqD*:
fixes *f g* :: *'a* :: {*idom,semiring-char-0*} *fps*
assumes *f ^ m = g ^ m f $ 0 = g $ 0 f $ 0 ≠ 0 m > 0*
shows *f = g*
by (*rule fps-power-eqD'*[*of f m g*]) (*insert assms*, *simp-all*)

```
lemma fps-compose-inj-right:
  assumes a0: a$0 = (0::'a::idom)
    and a1: a$1 ≠ 0
  shows (b oo a = c oo a) ⟷ b = c
  (is ?lhs ⟷ ?rhs)
proof
  show ?lhs if ?rhs using that by simp
  show ?rhs if ?lhs
  proof −
    have b$n = c$n for n
    proof (induct n rule: nat-less-induct)
      fix n
      assume H: ∀ m<n. b$m = c$m
      show b$n = c$n
      proof (cases n)
        case 0
        from ‹?lhs› have (b oo a)$n = (c oo a)$n
          by simp
        then show ?thesis
          using 0 by (simp add: fps-compose-nth)
      next
        case (Suc n1)
        have f: finite {0 .. n1} finite {n} by simp-all
        have eq: {0 .. n1} ∪ {n} = {0 .. n} using Suc by auto
        have d: {0 .. n1} ∩ {n} = {} using Suc by auto
        have seq: (∑ i = 0..n1. b $ i * a ^ i $ n) = (∑ i = 0..n1. c $ i * a ^ i $
n)
          using H Suc by auto
        have th0: (b oo a) $n = (∑ i = 0..n1. c $ i * a ^ i $ n) + b$n * (a$1)^n
          unfolding fps-compose-nth sum.union-disjoint[OF f d, unfolded eq] seq
          using startsby-zero-power-nth-same[OF a0]
          by simp
        have th1: (c oo a) $n = (∑ i = 0..n1. c $ i * a ^ i $ n) + c$n * (a$1)^n
          unfolding fps-compose-nth sum.union-disjoint[OF f d, unfolded eq]
          using startsby-zero-power-nth-same[OF a0]
          by simp
        from ‹?lhs›[unfolded fps-eq-iff, rule-format, of n] th0 th1 a1
        show ?thesis by auto
      qed
    qed
    then show ?rhs by (simp add: fps-eq-iff)
  qed
qed
```

5.16 Radicals

```
declare prod.cong [fundef-cong]

function radical :: (nat ⇒ 'a ⇒ 'a) ⇒ nat ⇒ 'a::field fps ⇒ nat ⇒ '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)
  by pat-completeness auto

termination radical
proof
  let ?R = measure (λ(r, k, a, n). n)
  {
    show wf ?R by auto
  next
    fix r :: nat ⇒ 'a ⇒ 'a
    and a :: 'a fps
    and k n xs i
    assume xs: xs ∈ {xs ∈ natpermute (Suc n) (Suc k). Suc n ∉ set xs} and i: i
∈ {0..k}
    have False if c: Suc n ≤ xs ! i
    proof −
      from xs i have xs !i ≠ Suc n
        by (simp add: in-set-conv-nth natpermute-def)
      with c have c': Suc n < xs!i by arith
      have fths: finite {0 ..< i} finite {i} finite {i+1..<Suc k}
        by simp-all
      have d: {0 ..< i} ∩ ({i} ∪ {i+1 ..< Suc k}) = {} {i} ∩ {i+1..< Suc k} =
{}
        by auto
      have eqs: {0..<Suc k} = {0 ..< i} ∪ ({i} ∪ {i+1 ..< Suc k})
        using i by auto
      from xs have Suc n = sum-list xs
        by (simp add: natpermute-def)
      also have ... = sum (nth xs) {0..<Suc k} using xs
        by (simp add: natpermute-def sum-list-sum-nth)
      also have ... = xs!i + sum (nth xs) {0..<i} + sum (nth xs) {i+1..<Suc k}
       unfolding eqs sum.union-disjoint[OF fths(1) finite-UnI[OF fths(2,3)] d(1)]
        unfolding sum.union-disjoint[OF fths(2) fths(3) d(2)]
        by simp
      finally show ?thesis using c' by simp
    qed
    then show ((r, Suc k, a, xs!i), r, Suc k, a, Suc n) ∈ ?R
      using not-less by auto
  next
    fix r :: nat ⇒ 'a ⇒ 'a
    and a :: 'a fps
    and k n
```

```
    show ((r, Suc k, a, 0), r, Suc k, a, Suc n) ∈ ?R by simp
  }
qed

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
  using radical.elims by blast

lemma fps-radical0[simp]: fps-radical r 0 a = 1
  by (auto simp add: fps-eq-iff fps-radical-def)

lemma fps-radical-nth-0[simp]: fps-radical r n a $ 0 = (if n = 0 then 1 else r n
(a$0))
  by (cases n) (simp-all add: fps-radical-def)

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 (cases k)
  case 0
  then show ?thesis by simp
next
  case (Suc h)
  have eq1: fps-radical r k a ^ k $ 0 = (∏j∈{0..h}. fps-radical r k a $ (replicate
k 0) ! j)
    unfolding fps-power-nth Suc by simp
  also have ... = (∏j∈{0..h}. r k (a$0))
  proof (rule prod.cong [OF refl])
    show fps-radical r k a $ replicate k 0 ! j = r k (a $ 0) if j ∈ {0..h} for j
    proof −
      have j < Suc h
        using that by presburger
      then show ?thesis
        by (metis Suc fps-radical-nth-0 nth-replicate old.nat.distinct(2))
    qed
  qed
  also have ... = a$0
    using r Suc by simp
  finally show ?thesis
    using Suc by simp
qed

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
  let ?r = fps-radical r (Suc k) a
  show ?rhs if r0: ?lhs
  proof −
    from a0 r0 have r00: r (Suc k) (a$0) ≠ 0 by auto
    have ?r ^ Suc k $ z = a$z for z
    proof (induct z rule: nat-less-induct)
      fix n
      assume H: ∀ m<n. ?r ^ Suc k $ m = a$m
      show ?r ^ Suc k $ n = a $n
      proof (cases n)
        case 0
        then show ?thesis
          using fps-radical-power-nth[of r Suc k a, OF r0] by simp
      next
        case (Suc n1)
        then have n ≠ 0 by simp
        let ?Pnk = natpermute n (k + 1)
        let ?Pnkn = {xs ∈ ?Pnk. n ∈ set xs}
        let ?Pnknn = {xs ∈ ?Pnk. n ∉ set xs}
        have eq: ?Pnkn ∪ ?Pnknn = ?Pnk by blast
        have d: ?Pnkn ∩ ?Pnknn = {} by blast
        have f: finite ?Pnkn finite ?Pnknn
          using finite-Un[of ?Pnkn ?Pnknn, unfolded eq]
          by (metis natpermute-finite)+
        let ?f = λv. ∏j∈{0..k}. ?r $ v ! j
        have sum ?f ?Pnkn = sum (λv. ?r $ n * r (Suc k) (a $ 0) ^ k) ?Pnkn
        proof (rule sum.cong)
          fix v assume v: v ∈ {xs ∈ natpermute n (k + 1). n ∈ set xs}
          let ?ths = (∏j∈{0..k}. fps-radical r (Suc k) a $ v ! j) =
            fps-radical r (Suc k) a $ n * r (Suc k) (a $ 0) ^ k
          from v obtain i where i: i ∈ {0..k} v = (replicate (k+1) 0) [i:= n]
            unfolding natpermute-contain-maximal by auto
          have (∏j∈{0..k}. fps-radical r (Suc k) a $ v ! j) =
                (∏j∈{0..k}. if j = i then fps-radical r (Suc k) a $ n else r (Suc k)
(a$0))
              using i r0 by (auto simp del: replicate.simps intro: prod.cong)
          also have ... = (fps-radical r (Suc k) a $ n) * r (Suc k) (a$0) ^ k
            using i r0 by (simp add: prod-gen-delta)
          finally show ?ths .
        qed rule
        then have sum ?f ?Pnkn = of-nat (k+1) * ?r $ n * r (Suc k) (a $ 0) ^ k
          by (simp add: natpermute-max-card[OF ‹n ≠ 0›, simplified])
        also have ... = a$n − sum ?f ?Pnknn
         unfolding Suc using r00 a0 by (simp add: field-simps fps-radical-def del:
of-nat-Suc)
        finally have fn: sum ?f ?Pnkn = a$n − sum ?f ?Pnknn .
        have (?r ^ Suc k)$n = sum ?f ?Pnkn + sum ?f ?Pnknn
          unfolding fps-power-nth-Suc sum.union-disjoint[OF f d, unfolded eq] ..
```

```
      also have ... = a$n unfolding fn by simp
      finally show ?thesis .
    qed
  qed
  then show ?thesis using r0 by (simp add: fps-eq-iff)
 qed
 show ?lhs if ?rhs
 proof −
  from that have ((fps-radical r (Suc k) a) ^ (Suc k))$0 = a$0
   by simp
  then show ?thesis
   unfolding fps-power-nth-Suc
   by (simp add: prod-constant del: replicate.simps)
 qed
qed

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
 show ?lhs if ?rhs
  using that using power-radical[OF b0, of r k, unfolded r0] by simp
 show ?rhs if ?lhs
 proof −
  have r00: r (Suc k) (b$0) ≠ 0 using b0 r0 by auto
  have ceq: card {0..k} = Suc k by simp
  from a0 have a0r0: a$0 = ?r$0 by simp
  have a $ n = ?r $ n for n
  proof (induct n rule: nat-less-induct)
   fix n
   assume h: ∀ m<n. a$m = ?r $m
   show a$n = ?r $ n
   proof (cases n)
    case 0
    then show ?thesis using a0 by simp
   next
    case (Suc n1)
    have fK: finite {0..k} by simp
    have nz: n ≠ 0 using Suc by simp
    let ?Pnk = natpermute n (Suc k)
    let ?Pnkn = {xs ∈ ?Pnk. n ∈ set xs}
    let ?Pnknn = {xs ∈ ?Pnk. n ∉ set xs}
    have eq: ?Pnkn ∪ ?Pnknn = ?Pnk by blast
    have d: ?Pnkn ∩ ?Pnknn = {} by blast
    have f: finite ?Pnkn finite ?Pnknn
     using finite-Un[of ?Pnkn ?Pnknn, unfolded eq]
```

```
    by (metis natpermute-finite)+
  let ?f = λv. ∏j∈{0..k}. ?r $ v ! j
  let ?g = λv. ∏j∈{0..k}. a $ v ! j
  have sum ?g ?Pnkn = sum (λv. a $ n * (?r$0)^k) ?Pnkn
  proof (rule sum.cong)
    fix v
    assume v: v ∈ {xs ∈ natpermute n (Suc k). n ∈ set xs}
    let ?ths = (∏j∈{0..k}. a $ v ! j) = a $ n * (?r$0)^k
    from v obtain i where i: i ∈ {0..k} v = (replicate (k+1) 0) [i:= n]
      unfolding Suc-eq-plus1 natpermute-contain-maximal
      by (auto simp del: replicate.simps)
   have (∏j∈{0..k}. a $ v ! j) = (∏j∈{0..k}. if j = i then a $ n else r (Suc
k) (b$0))
      using i a0 by (auto simp del: replicate.simps intro: prod.cong)
    also have ... = a $ n * (?r $ 0)^k
      using i by (simp add: prod-gen-delta)
    finally show ?ths .
  qed rule
  then have th0: sum ?g ?Pnkn = of-nat (k+1) * a $ n * (?r $ 0)^k
    by (simp add: natpermute-max-card[OF nz, simplified])
  have th1: sum ?g ?Pnknn = sum ?f ?Pnknn
  proof (rule sum.cong, rule refl, rule prod.cong, simp)
    fix xs i
    assume xs: xs ∈ ?Pnknn and i: i ∈ {0..k}
    have False if c: n ≤ xs ! i
    proof −
      from xs i have xs ! i ≠ n
        by (simp add: in-set-conv-nth natpermute-def)
      with c have c': n < xs!i by arith
      have fths: finite {0 ..< i} finite {i} finite {i+1..<Suc k}
        by simp-all
       have d: {0 ..< i} ∩ ({i} ∪ {i+1 ..< Suc k}) = {} {i} ∩ {i+1..< Suc
k} = {}
        by auto
      have eqs: {0..<Suc k} = {0 ..< i} ∪ ({i} ∪ {i+1 ..< Suc k})
        using i by auto
      from xs have n = sum-list xs
        by (simp add: natpermute-def)
      also have ... = sum (nth xs) {0..<Suc k}
        using xs by (simp add: natpermute-def sum-list-sum-nth)
     also have ... = xs!i + sum (nth xs) {0..<i} + sum (nth xs) {i+1..<Suc
k}
       unfolding eqs  sum.union-disjoint[OF fths(1) finite-UnI[OF fths(2,3)]
d(1)]
        unfolding sum.union-disjoint[OF fths(2) fths(3) d(2)]
        by simp
      finally show ?thesis using c' by simp
    qed
    then have thn: xs!i < n by presburger
```

```
      from h[rule-format, OF thn] show a$(xs !i) = ?r$(xs!i) .
    qed
    have th00: ⋀x::'a. of-nat (Suc k) * (x * inverse (of-nat (Suc k))) = x
      by (simp add: field-simps del: of-nat-Suc)
    from ‹?lhs› have b$n = a^Suc k $ n
      by (simp add: fps-eq-iff)
    also have a ^ Suc k$n = sum ?g ?Pnkn + sum ?g ?Pnknn
      unfolding fps-power-nth-Suc
      using sum.union-disjoint[OF f d, unfolded Suc-eq-plus1[symmetric],
        unfolded eq, of ?g] by simp
    also have ... = of-nat (k+1) * a $ n * (?r $ 0)^k + sum ?f ?Pnknn
      unfolding th0 th1 ..
    finally have §: of-nat (k+1) * a $ n * (?r $ 0)^k = b$n - sum ?f ?Pnknn
      by simp
    have a$n = (b$n - sum ?f ?Pnknn) / (of-nat (k+1) * (?r $ 0)^k)
      apply (rule eq-divide-imp)
      using r00 § by (simp-all add: ac-simps del: of-nat-Suc)
    then show ?thesis
      unfolding fps-radical-def Suc
      by (simp del: of-nat-Suc)
  qed
 qed
 then show ?rhs by (simp add: fps-eq-iff)
 qed
qed


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 −
 let ?ak = a^ Suc k
 have ak0: ?ak $ 0 = (a$0) ^ Suc k
   by (simp add: fps-nth-power-0 del: power-Suc)
 from r0 have th0: r (Suc k) (a ^ Suc k $ 0) ^ Suc k = a ^ Suc k $ 0
   using ak0 by auto
 from r0 ak0 have th1: r (Suc k) (a ^ Suc k $ 0) = a $ 0
   by auto
 from ak0 a0 have ak00: ?ak $ 0 ≠0
   by auto
 from radical-unique[of r k ?ak a, OF th0 th1 ak00] show ?thesis
   by metis
qed

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 -
  let ?r = fps-radical r (Suc k) a
  let ?w = (of-nat (Suc k)) * ?r ^ k
  from a0 r0 have r0': r (Suc k) (a$0) ≠ 0
    by auto
  from r0' have w0: ?w $ 0 ≠ 0
    by (simp del: of-nat-Suc)
  note th0 = inverse-mult-eq-1[OF w0]
  let ?iw = inverse ?w
  from iffD1[OF power-radical[of a r], OF a0 r0]
  have fps-deriv (?r ^ Suc k) = fps-deriv a
    by simp
  then have fps-deriv ?r * ?w = fps-deriv a
    by (simp add: fps-deriv-power' ac-simps del: power-Suc)
  then have ?iw * fps-deriv ?r * ?w = ?iw * fps-deriv a
    by simp
  with a0 r0 have fps-deriv ?r * (?iw * ?w) = fps-deriv a / ?w
    by (subst fps-divide-unit) (auto simp del: of-nat-Suc)
  then show ?thesis unfolding th0 by simp
qed

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)
  using fps-deriv-radical'[of r k a, OF r0 a0]
  by (simp add: fps-of-nat[symmetric])

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
  show ?rhs if r0': ?lhs
  proof -
    from r0' have r0: (r k ((a * b) $ 0)) ^ k = (a * b) $ 0
      by (simp add: fps-mult-nth ra0 rb0 power-mult-distrib)
    show ?thesis
    proof (cases k)
```

```
    case 0
    then show ?thesis using r0' by simp
   next
    case (Suc h)
    let ?ra = fps-radical r (Suc h) a
    let ?rb = fps-radical r (Suc h) b
    have th0: r (Suc h) ((a * b) $ 0) = (fps-radical r (Suc h) a * fps-radical r
(Suc h) b) $ 0
     using r0' Suc by (simp add: fps-mult-nth)
    have ab0: (a*b) $ 0 ≠ 0
     using a0 b0 by (simp add: fps-mult-nth)
    from radical-unique[of r h a*b fps-radical r (Suc h) a * fps-radical r (Suc h)
b, OF r0[unfolded Suc] th0 ab0, symmetric]
    iffD1[OF power-radical[of - r], OF a0 ra0[unfolded Suc]] iffD1[OF power-radical[of
- r], OF b0 rb0[unfolded Suc]] Suc r0'
    show ?thesis
     by (auto simp add: power-mult-distrib simp del: power-Suc)
   qed
  qed
  show ?lhs if ?rhs
  proof −
   from that have (fps-radical r k (a * b)) $ 0 = (fps-radical r k a * fps-radical r
k b) $ 0
    by simp
   then show ?thesis
    using k by (simp add: fps-mult-nth)
  qed
qed



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
  let ?r = fps-radical r k
  from kp obtain h where k: k = Suc h
   by (cases k) auto
  have ra0': r k (a$0) ≠ 0 using a0 ra0 k by auto
  have rb0': r k (b$0) ≠ 0 using b0 rb0 k by auto

  show ?lhs if ?rhs
```

```
  proof −
    from that have ?r (a/b) $ 0 = (?r a / ?r b)$0
      by simp
    then show ?thesis
      using k a0 b0 rb0′ by (simp add: fps-divide-unit fps-mult-nth fps-inverse-def
divide-inverse)
  qed
  show ?rhs if ?lhs
  proof −
    from a0 b0 have ab0[simp]: (a/b)$0 = a$0 / b$0
      by (simp add: fps-divide-def fps-mult-nth divide-inverse fps-inverse-def)
    have th0: r k ((a/b)$0) ^ k = (a/b)$0
      by (simp add: ‹?lhs› power-divide ra0 rb0)
    from a0 b0 ra0′ rb0′ kp ‹?lhs›
    have th1: r k ((a / b) $ 0) = (fps-radical r k a / fps-radical r k b) $ 0
      by (simp add: fps-divide-unit fps-mult-nth fps-inverse-def divide-inverse)
    from a0 b0 ra0′ rb0′ kp have ab0′: (a / b) $ 0 ≠ 0
    by (simp add: fps-divide-unit fps-mult-nth fps-inverse-def nonzero-imp-inverse-nonzero)
   note tha[simp] = iffD1[OF power-radical[where r=r and k=h], OF a0 ra0[unfolded
k], unfolded k[symmetric]]
   note thb[simp] = iffD1[OF power-radical[where r=r and k=h], OF b0 rb0[unfolded
k], unfolded k[symmetric]]
    from b0 rb0′ have th2: (?r a / ?r b)^k = a/b
     by (simp add: fps-divide-unit power-mult-distrib fps-inverse-power[symmetric])

    from iffD1[OF radical-unique[where r=r and a=?r a / ?r b and b=a/b and
k=h], symmetric, unfolded k[symmetric], OF th0 th1 ab0′ th2]
    show ?thesis .
  qed
qed

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)^k = 1
    and a0: a$0 ≠ 0
  shows r k (inverse (a $ 0)) = r k 1 / (r k (a $ 0)) ⟷
    fps-radical r k (inverse a) = fps-radical r k 1 / fps-radical r k a
  using radical-divide[where k=k and r=r and a=1 and b=a, OF k ] ra0 r1 a0
  by (simp add: divide-inverse fps-divide-def)
```

5.17 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 −
```

```
  have (fps-deriv (a oo b))$n = (((fps-deriv a) oo b) * (fps-deriv b)) $n for n
  proof −
    have (fps-deriv (a oo b))$n = sum (λi. a $ i * (fps-deriv (b^i))$n) {0.. Suc n}
      by (simp add: fps-compose-def field-simps sum-distrib-left del: of-nat-Suc)
    also have ... = sum (λi. a$i * ((fps-const (of-nat i)) * (fps-deriv b * (b^(i −
1))))$n) {0.. Suc n}
      by (simp add: field-simps fps-deriv-power del: fps-mult-left-const-nth of-nat-Suc)
    also have ... = sum (λi. of-nat i * a$i * (((b^(i − 1)) * fps-deriv b))$n) {0..
Suc n}
      unfolding fps-mult-left-const-nth  by (simp add: field-simps)
    also have ... = sum (λi. of-nat i * a$i * (sum (λj. (b^(i − 1))$j * (fps-deriv
b)$(n − j)) {0..n})) {0.. Suc n}
      unfolding fps-mult-nth ..
    also have ... = sum (λi. of-nat i * a$i * (sum (λj. (b^(i − 1))$j * (fps-deriv
b)$(n − j)) {0..n})) {1.. Suc n}
      by (intro sum.mono-neutral-right) (auto simp add: mult-delta-left not-le)
     also have ... = sum (λi. of-nat (i + 1) * a$(i+1) * (sum (λj. (b^ i)$j *
of-nat (n − j + 1) * b$(n − j + 1)) {0..n})) {0.. n}
      unfolding fps-deriv-nth
      by (rule sum.reindex-cong [of Suc]) (simp-all add: mult.assoc)
    finally have th0: (fps-deriv (a oo b))$n =
      sum (λi. of-nat (i + 1) * a$(i+1) * (sum (λj. (b^ i)$j * of-nat (n − j + 1)
* b$(n − j + 1)) {0..n})) {0.. n} .

    have (((fps-deriv a) oo b) * (fps-deriv b))$n = sum (λi. (fps-deriv b)$ (n − i)
* ((fps-deriv a) oo b)$i) {0..n}
      unfolding fps-mult-nth by (simp add: ac-simps)
    also have ... = sum (λi. sum (λj. of-nat (n − i +1) * b$(n − i + 1) * of-nat
(j + 1) * a$(j+1) * (b^j)$i) {0..n}) {0..n}
      unfolding fps-deriv-nth fps-compose-nth sum-distrib-left mult.assoc
       by (auto simp: subset-eq b0 startsby-zero-power-prefix sum.mono-neutral-left
intro: sum.cong)
     also have ... = sum (λi. of-nat (i + 1) * a$(i+1) * (sum (λj. (b^ i)$j *
of-nat (n − j + 1) * b$(n − j + 1)) {0..n})) {0.. n}
      unfolding sum-distrib-left
      by (subst sum.swap) (force intro: sum.cong)
    finally show ?thesis
      unfolding th0 by simp
  qed
  then show ?thesis by (simp add: fps-eq-iff)
qed

lemma fps-poly-sum-fps-X:
  assumes ∀ i > n. a$i = 0
  shows a = sum (λi. fps-const (a$i) * fps-X^i) {0..n} (is a = ?r)
proof −
  have a$i = ?r$i for i
    unfolding fps-sum-nth fps-mult-left-const-nth fps-X-power-nth
    by (simp add: mult-delta-right assms)
```

then show *?thesis*
unfolding *fps-eq-iff* **by** *blast*
qed

5.18 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* (λ*i*. (*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* ≠ *0*
shows *fps-inv a oo a* = *fps-X*
proof −
let *?i* = *fps-inv a oo a*
have *?i* $*n* = *fps-X*$*n* **for** *n*
proof (*induct n rule*: *nat-less-induct*)
fix *n*
assume *h*: ∀ *m*<*n*. *?i*$*m* = *fps-X*$*m*
show *?i* $ *n* = *fps-X*$*n*
proof (*cases n*)
case *0*
then show *?thesis* **using** *a0*
by (*simp add*: *fps-compose-nth fps-inv-def*)
next
case (*Suc n1*)
have *?i* $ *n* = *sum* (λ*i*. (*fps-inv a* $ *i*) ∗ (*a*^*i*)$*n*) {*0* .. *n1*} + *fps-inv a* $ *Suc n1* ∗ (*a* $ *1*)^ *Suc n1*
by (*simp only*: *fps-compose-nth*) (*simp add*: *Suc startsby-zero-power-nth-same* [*OF a0*] *del*: *power-Suc*)
also have . . . = *sum* (λ*i*. (*fps-inv a* $ *i*) ∗ (*a*^*i*)$*n*) {*0* .. *n1*} +
(*fps-X$ Suc n1 − sum* (λ*i*. (*fps-inv a* $ *i*) ∗ (*a*^*i*)$*n*) {*0* .. *n1*})
using *a0 a1 Suc* **by** (*simp add*: *fps-inv-def*)
also have . . . = *fps-X*$*n* **using** *Suc* **by** *simp*
finally show *?thesis* **.**
qed
qed
then show *?thesis*
by (*simp add*: *fps-eq-iff*)
qed

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 ≠ 0
  shows fps-ginv b a oo a = b
proof −
  let ?i = fps-ginv b a oo a
  have ?i $n = b$n for n
  proof (induct n rule: nat-less-induct)
    fix n
    assume h: ∀ m<n. ?i$m = b$m
    show ?i $ n = b$n
    proof (cases n)
      case 0
      then show ?thesis using a0
        by (simp add: fps-compose-nth fps-ginv-def)
    next
      case (Suc n1)
      have ?i $ n = sum (λi. (fps-ginv b a $ i) ∗ (a^i)$n) {0 .. n1} + fps-ginv b
a $ Suc n1 ∗ (a $ 1)^ Suc n1
        by (simp only: fps-compose-nth) (simp add: Suc startsby-zero-power-nth-same
[OF a0] del: power-Suc)
      also have ... = sum (λi. (fps-ginv b a $ i) ∗ (a^i)$n) {0 .. n1} +
        (b$ Suc n1 − sum (λi. (fps-ginv b a $ i) ∗ (a^i)$n) {0 .. n1})
        using a0 a1 Suc by (simp add: fps-ginv-def)
      also have ... = b$n using Suc by simp
      finally show ?thesis .
    qed
  qed
  then show ?thesis
    by (simp add: fps-eq-iff)
qed

lemma fps-inv-ginv: fps-inv = fps-ginv fps-X
proof −
  have compinv x n = gcompinv fps-X x n for n and x :: 'a fps
  proof (induction n rule: nat-less-induct)
    case (1 n)
    then show ?case
      by (cases n) auto
  qed
  then show ?thesis
```

by (*auto simp add*: *fun-eq-iff fps-eq-iff fps-inv-def fps-ginv-def*)
qed

lemma *fps-compose-1*[*simp*]: *1 oo a = 1*
by (*simp add*: *fps-eq-iff fps-compose-nth mult-delta-left*)

lemma *fps-compose-0*[*simp*]: *0 oo a = 0*
by (*simp add*: *fps-eq-iff fps-compose-nth*)

lemma *fps-compose-0-right*[*simp*]: *a oo 0 = fps-const* (*a $ 0*)
by (*simp add*: *fps-eq-iff fps-compose-nth power-0-left sum.neutral*)

lemma *fps-compose-add-distrib*: (*a + b*) *oo c* = (*a oo c*) + (*b oo c*)
by (*simp add*: *fps-eq-iff fps-compose-nth field-simps sum.distrib*)

lemma *fps-compose-sum-distrib*: (*sum f S*) *oo a* = *sum* (*λi. f i oo a*) *S*
proof (*cases finite S*)
case *True*
show *?thesis*
proof (*rule finite-induct*[*OF True*])
show *sum f* {} *oo a* = (∑ *i*∈{}. *f i oo a*)
by *simp*
next
fix *x F*
assume *fF*: *finite F*
and *xF*: *x* ∉ *F*
and *h*: *sum f F oo a* = *sum* (*λi. f i oo a*) *F*
show *sum f* (*insert x F*) *oo a* = *sum* (*λi. f i oo a*) (*insert x F*)
using *fF xF h* **by** (*simp add*: *fps-compose-add-distrib*)
qed
next
case *False*
then show *?thesis* **by** *simp*
qed

lemma *convolution-eq*:
sum (*λi. 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*}
by (*rule sum.reindex-bij-witness*[**where** *i*=*fst* **and** *j*=*λi.* (*i*, *n* − *i*)]) *auto*

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 −
let *?S* = {(*k*::*nat*, *m*::*nat*). *k* + *m* ≤ *n*}
have *s*: *?S* ⊆ {*0*..*n*} × {*0*..*n*} **by** (*simp add*: *subset-eq*)
have *f*: *finite* {(*k*::*nat*, *m*::*nat*). *k* + *m* ≤ *n*}

by (*auto intro*: *finite-subset*[*OF s*])
have *?r* = ($\sum$ (*k*, *m*) ∈ {(*k*, *m*). *k* + *m* ≤ *n*}. $\sum$ *j* = *0*..*n*. *a* $ *k* * *b* $ *m* * (*c* ^ *k* $ *j* * *d* ^ *m* $ (*n* − *j*)))
by (*simp add*: *fps-mult-nth sum-distrib-left*)
also have ... = ($\sum$ *i* = *0*..*n*. $\sum$ (*k*,*m*)∈{(*k*,*m*). *k*+*m* ≤ *n*}. *a* $ *k* * *c* ^ *k* $ *i* * *b* $ *m* * *d* ^ *m* $ (*n*−*i*))
unfolding *sum.swap* [**where** *A* = {*0*..*n*}] **by** (*auto simp add*: *field-simps intro*: *sum.cong*)
also have ... = ($\sum$ *i* = *0*..*n*.
$\sum$ *q* = *0*..*i*. $\sum$ *j* = *0*..*n* − *i*. *a* $ *q* * *c* ^ *q* $ *i* * (*b* $ *j* * *d* ^ *j* $ (*n* − *i*)))
apply (*rule sum.cong* [*OF refl*])
apply (*simp add*: *sum.cartesian-product mult.assoc*)
apply (*rule sum.mono-neutral-right*[*OF f*], *force*)
by *clarsimp* (*meson c0 d0 leI startsby-zero-power-prefix*)
also have ... = *?l*
by (*simp add*: *fps-mult-nth fps-compose-nth sum-product*)
finally show *?thesis* **by** *simp*
qed

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 −
have *th0*: {(*k*, *m*). *k* + *m* ≤ *n*} = ($\bigcup$ *s*∈{*0*..*n*}. $\bigcup$ *i*∈{*0*..*s*}. {(*i*, *s* − *i*)})
by *auto*
show *?l* = *?r*
unfolding *th0*
by (*simp add*: *sum.UNION-disjoint eq-diff-iff disjoint-iff*)
qed

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*}
unfolding *product-composition-lemma*[*OF c0 c0*] *power-add*[*symmetric*]
unfolding *sum-pair-less-iff*[**where** *a* = λ*k*. *a*$*k* **and** *b*=λ*m*. *b*$*m* **and** *c*=λ*s*. (*c* ^ *s*)$*n* **and** *n* = *n*] **..**

lemma *fps-compose-mult-distrib*:
assumes *c0*: *c* $ *0* = (*0*::*'a*::*idom*)
shows (*a* * *b*) *oo c* = (*a oo c*) * (*b oo c*)
proof (*clarsimp simp add*: *fps-eq-iff fps-compose-mult-distrib-lemma* [*OF c0*])
show (*a* * *b oo c*) $ *n* = ($\sum$ *s* = *0*..*n*. $\sum$ *i* = *0*..*s*. *a* $ *i* * *b* $ (*s* − *i*) * *c* ^ *s* $ *n*) **for** *n*
by (*simp add*: *fps-compose-nth fps-mult-nth sum-distrib-right*)
qed

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 (*induct S rule*: *infinite-finite-induct*)
next
case (*insert*)
then show *?case*
by (*simp add*: *fps-compose-mult-distrib*[*OF c0*])
qed *auto*

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 −
have *f = (f / g) * g* **by** *simp*
also have *fps-compose ... h = fps-compose (f / g) h * fps-compose g h*
by (*subst fps-compose-mult-distrib*) *simp-all*
finally show *?thesis* **.**
qed

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*
using *fps-compose-divide*[*OF assms*(*1*,*2*)] *assms*(*3*) **by** *simp*

lemma *fps-compose-power*:
assumes *c0*: *c$0 = (0::'a::idom)*
shows *(a oo c)^n = a^n oo c*
proof (*cases n*)
case *0*
then show *?thesis* **by** *simp*
next
case (*Suc m*)
have *(∏ n = 0..m. a) oo c = (∏ n = 0..m. a oo c)*
using *c0 fps-compose-prod-distrib* **by** *blast*
moreover have *th0*: *a^n = prod (λk. a) {0..m} (a oo c) ^ n = prod (λk. a oo c) {0..m}*
by (*simp-all add*: *prod-constant Suc*)
ultimately show *?thesis*
by *presburger*
qed

lemma *fps-compose-uminus*: *− (a::'a::ring-1 fps) oo c = − (a oo c)*
by (*simp add*: *fps-eq-iff fps-compose-nth field-simps sum-negf*[*symmetric*])

lemma *fps-compose-sub-distrib*: *(a − b) oo (c::'a::ring-1 fps) = (a oo c) − (b oo*

```
c)
  using fps-compose-add-distrib [of a − b c] by (simp add: fps-compose-uminus)

lemma fps-X-fps-compose: fps-X oo a = Abs-fps (λn. if n = 0 then (0::'a::comm-ring-1)
else a$n)
  by (simp add: fps-eq-iff fps-compose-nth mult-delta-left)

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 safe
  assume ∗: fps-compose F G = 0 F ≠ 0
  have fps-nth (fps-compose F G) 0 = fps-nth F 0
    by simp
  also have fps-compose F G = 0
    by (simp add: ∗)
  finally show fps-nth F 0 = 0
    by simp
  show G = 0
  proof (rule ccontr)
    assume G ≠ 0
    hence subdegree G > 0 using assms
      using subdegree-eq-0-iff by blast
    define N where N = subdegree F ∗ subdegree G
    have fps-nth (fps-compose F G) N = (∑ i = 0..N. fps-nth F i ∗ fps-nth (G ^
i) N)
      unfolding fps-compose-def by (simp add: N-def)
    also have ... = (∑ i∈{subdegree F}. fps-nth F i ∗ fps-nth (G ^ i) N)
    proof (intro sum.mono-neutral-right ballI)
      fix i assume i: i ∈ {0..N} − {subdegree F}
      show fps-nth F i ∗ fps-nth (G ^ i) N = 0
      proof (cases i subdegree F rule: linorder-cases)
        assume i > subdegree F
        hence fps-nth (G ^ i) N = 0
          using i ‹subdegree G > 0› by (intro fps-pow-nth-below-subdegree) (auto
simp: N-def)
        thus ?thesis by simp
      qed (use i in ‹auto simp: N-def›)
    qed (use ‹subdegree G > 0› in ‹auto simp: N-def›)
    also have ... = fps-nth F (subdegree F) ∗ fps-nth (G ^ subdegree F) N
      by simp
    also have ... ≠ 0
      using ‹G ≠ 0› ‹F ≠ 0› by (auto simp: N-def)
    finally show False using ∗ by auto
  qed
qed auto

lemma subdegree-fps-compose [simp]:
```

fixes $F\ G :: 'a :: idom\ fps$
assumes [*simp*]: *fps-nth G 0 = 0*
shows *subdegree* (*fps-compose F G*) = *subdegree F* ∗ *subdegree G*
proof (*cases G = 0*; *cases F = 0*)
assume [*simp*]: $G \neq 0\ F \neq 0$
define m **where** m = *subdegree F*
define F' **where** F' = *fps-shift m F*
have *F-eq*: $F = F' * fps\text{-}X\ \hat{}\ m$
unfolding *F'-def* **by** (*simp add*: *fps-shift-times-fps-X-power m-def*)
have [*simp*]: $F' \neq 0$
using ‹$F \neq 0$› **unfolding** *F-eq* **by** *auto*
have *subdegree* (*fps-compose F G*) = *subdegree* (*fps-compose F' G*) + *m* ∗ *subdegree G*
by (*simp add*: *F-eq fps-compose-mult-distrib fps-compose-eq-0-iff flip*: *fps-compose-power*)
also have *subdegree* (*fps-compose F' G*) = *0*
by (*intro subdegree-eq-0*) (*auto simp*: *F'-def m-def*)
finally show *?thesis* **by** (*simp add*: *m-def*)
qed *auto*

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 −
let *?ia* = *inverse a*
let *?ab* = *a oo b*
let *?iab* = *inverse ?ab*

from *a0* **have** *ia0*: *?ia $ 0* ≠ *0* **by** *simp*
from *a0* **have** *ab0*: *?ab $ 0* ≠ *0* **by** (*simp add*: *fps-compose-def*)
have (*?ia oo b*) ∗ (*a oo b*) = *1*
unfolding *fps-compose-mult-distrib*[*OF b0*, *symmetric*]
unfolding *inverse-mult-eq-1*[*OF a0*]
fps-compose-1 **..**

then have (*?ia oo b*) ∗ (*a oo b*) ∗ *?iab* = *1* ∗ *?iab* **by** *simp*
then have (*?ia oo b*) ∗ (*?iab* ∗ (*a oo b*)) = *?iab* **by** *simp*
then show *?thesis* **unfolding** *inverse-mult-eq-1*[*OF ab0*] **by** *simp*
qed

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*)
using *b0 c0* **by** (*simp add*: *fps-divide-unit fps-inverse-compose fps-compose-mult-distrib*)

lemma *gp*:
assumes *a0*: *a$0* = (*0*::'a::*field*)
shows (*Abs-fps* (λ*n*. *1*)) *oo a* = *1*/(*1* − *a*)

```
    (is ?one oo a = -)
proof −
  have o0: ?one $ 0 ≠ 0 by simp
  have th0: (1 − fps-X) $ 0 ≠ (0::'a) by simp
  from fps-inverse-gp[where ?'a = 'a]
  have inverse ?one = 1 − fps-X by (simp add: fps-eq-iff)
  then have inverse (inverse ?one) = inverse (1 − fps-X) by simp
  then have th: ?one = 1/(1 − fps-X) unfolding fps-inverse-idempotent[OF o0]
    by (simp add: fps-divide-def)
  show ?thesis
    unfolding th
    unfolding fps-divide-compose[OF a0 th0]
    fps-compose-1 fps-compose-sub-distrib fps-X-fps-compose-startby0[OF a0] ..
qed

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 −
  let ?r = fps-radical r (Suc k)
  let ?ab = a oo b
  have ab0: ?ab $ 0 = a$0
    by (simp add: fps-compose-def)
  from ab0 a0 ra0 have rab0: ?ab $ 0 ≠ 0 r (Suc k) (?ab $ 0) ^ Suc k = ?ab $ 0
    by simp-all
  have th00: r (Suc k) ((a oo b) $ 0) = (fps-radical r (Suc k) a oo b) $ 0
    by (simp add: ab0 fps-compose-def)
  have th0: (?r a oo b) ^ (Suc k) = a  oo b
    unfolding fps-compose-power[OF b0]
    unfolding iffD1[OF power-radical[of a r k], OF a0 ra0]  ..
  from iffD1[OF radical-unique[where r=r and k=k and b= ?ab and a = ?r a
oo b, OF rab0(2) th00 rab0(1)], OF th0]
  show ?thesis  .
qed

lemma fps-const-mult-apply-left: fps-const c * (a oo b) = (fps-const c * a) oo b
  by (simp add: fps-eq-iff fps-compose-nth sum-distrib-left mult.assoc)

lemma fps-const-mult-apply-right:
  (a oo b) * fps-const (c::'a::comm-semiring-1) = (fps-const c * a) oo b
  by (simp add: fps-const-mult-apply-left mult.commute)

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 −
```

```
  have ?l$n = ?r$n for n
  proof −
    have ?l$n = (sum (λi. (fps-const (a$i) * b^i) oo c) {0..n})$n
    by (simp add: fps-compose-nth fps-compose-power[OF c0] fps-const-mult-apply-left
        sum-distrib-left mult.assoc fps-sum-nth)
    also have ... = ((sum (λi. fps-const (a$i) * b^i) {0..n}) oo c)$n
      by (simp add: fps-compose-sum-distrib)
    also have ... = (∑ i = 0..n. ∑ j = 0..n. a $ j * (b ^ j $ i * c ^ i $ n))
      by (simp add: fps-compose-nth fps-sum-nth sum-distrib-right mult.assoc)
    also have ... = (∑ i = 0..n. ∑ j = 0..i. a $ j * (b ^ j $ i * c ^ i $ n))
     by (intro sum.cong [OF refl] sum.mono-neutral-right; simp add: b0 startsby-zero-power-prefix)
    also have ... = ?r$n
      by (simp add: fps-compose-nth  sum-distrib-right mult.assoc)
    finally show ?thesis .
  qed
  then show ?thesis
    by (simp add: fps-eq-iff)
qed


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 (cases k)
  case 0
  then show ?thesis by simp
next
  case (Suc h)
  have ?l $ n = ?r $n for n
  proof −
    consider k > n | k ≤ n by arith
    then show ?thesis
    proof cases
      case 1
      then show ?thesis
        using a0 startsby-zero-power-prefix[OF a0] Suc
        by (simp add: fps-compose-nth del: power-Suc)
    next
      case 2
      then show ?thesis
        by (simp add: fps-compose-nth mult-delta-left)
    qed
  qed
  then show ?thesis
    unfolding fps-eq-iff by blast
qed

lemma fps-inv-right:
```

```
  assumes a0: a$0 = 0
    and a1: a$1 ≠ 0
  shows a oo fps-inv a = fps-X
proof −
  let ?ia = fps-inv a
  let ?iaa = a oo fps-inv a
  have th0: ?ia $ 0 = 0
    by (simp add: fps-inv-def)
  have th1: ?iaa $ 0 = 0
    using a0 a1 by (simp add: fps-inv-def fps-compose-nth)
  have th2: fps-X$0 = 0
    by simp
  from fps-inv[OF a0 a1] have a oo (fps-inv a oo a) = a oo fps-X
    by simp
  then have (a oo fps-inv a) oo a = fps-X oo a
     by (simp add: fps-compose-assoc[OF a0 th0] fps-X-fps-compose-startby0[OF
a0])
  with fps-compose-inj-right[OF a0 a1] show ?thesis
    by simp
qed

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 −
  let ?ia = fps-inv a
  let ?d = fps-deriv a oo ?ia
  let ?dia = fps-deriv ?ia
  have ia0: ?ia$0 = 0
    by (simp add: fps-inv-def)
  have th0: ?d$0 ≠ 0
    using a1 by (simp add: fps-compose-nth)
  from fps-inv-right[OF a0 a1] have ?d * ?dia = 1
    by (simp add: fps-compose-deriv[OF ia0, of a, symmetric] )
  then have inverse ?d * ?d * ?dia = inverse ?d * 1
    by simp
  with inverse-mult-eq-1 [OF th0] show ?dia = inverse ?d
    by simp
qed

lemma fps-inv-idempotent:
  assumes a0: a$0 = 0
    and a1: a$1 ≠ 0
  shows fps-inv (fps-inv a) = a
proof −
  let ?r = fps-inv
  have ra0: ?r a $ 0 = 0
    by (simp add: fps-inv-def)
```

```
  from a1 have ra1: ?r a $ 1 ≠ 0
    by (simp add: fps-inv-def field-simps)
  have fps-X0: fps-X$0 = 0
    by simp
  from fps-inv[OF ra0 ra1] have ?r (?r a) oo ?r a = fps-X .
  then have ?r (?r a) oo ?r a oo a = fps-X oo a
    by simp
  then have ?r (?r a) oo (?r a oo a) = a
    unfolding fps-X-fps-compose-startby0[OF a0]
    unfolding fps-compose-assoc[OF a0 ra0, symmetric] .
  then show ?thesis
    unfolding fps-inv[OF a0 a1] by simp
qed

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 −
  let ?r = fps-ginv
  from c0 have rca0: ?r c a $0 = 0
    by (simp add: fps-ginv-def)
  from a1 c1 have rca1: ?r c a $ 1 ≠ 0
    by (simp add: fps-ginv-def field-simps)
  from fps-ginv[OF rca0 rca1]
  have ?r b (?r c a) oo ?r c a = b .
  then have ?r b (?r c a) oo ?r c a oo a = b oo a
    by simp
  then have ?r b (?r c a) oo (?r c a oo a) = b oo a
    by (simp add: a0 fps-compose-assoc rca0)
  then have ?r b (?r c a) oo c = b oo a
    unfolding fps-ginv[OF a0 a1] .
  then have ?r b (?r c a) oo c oo fps-inv c= b oo a oo fps-inv c
    by simp
  then have ?r b (?r c a) oo (c oo fps-inv c) = b oo a oo fps-inv c
    by (metis c0 c1 fps-compose-assoc fps-compose-nth-0 fps-inv fps-inv-right)
  then show ?thesis
    unfolding fps-inv-right[OF c0 c1] by simp
qed

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 −
  let ?ia = fps-ginv b a
  let ?ifps-Xa = fps-ginv fps-X a
```

```
  let ?d = fps-deriv
  let ?dia = ?d ?ia
  have ifps-Xa0: ?ifps-Xa $ 0 = 0
    by (simp add: fps-ginv-def)
  have da0: ?d a $ 0 ≠ 0
    using a1 by simp
  from fps-ginv[OF a0 a1, of b] have ?d (?ia oo a) = fps-deriv b
    by simp
  then have (?d ?ia oo a) * ?d a = ?d b
    unfolding fps-compose-deriv[OF a0] .
  then have (?d ?ia oo a) * ?d a * inverse (?d a) = ?d b * inverse (?d a)
    by simp
  with a1 have (?d ?ia oo a) * (inverse (?d a) * ?d a) = ?d b / ?d a
    by (simp add: fps-divide-unit)
  then have (?d ?ia oo a) oo ?ifps-Xa = (?d b / ?d a) oo ?ifps-Xa
    unfolding inverse-mult-eq-1[OF da0] by simp
  then have ?d ?ia oo (a oo ?ifps-Xa) = (?d b / ?d a) oo ?ifps-Xa
    unfolding fps-compose-assoc[OF ifps-Xa0 a0] .
  then show ?thesis unfolding fps-inv-ginv[symmetric]
    unfolding fps-inv-right[OF a0 a1] by simp
qed

lemma fps-compose-linear:
  fps-compose (f :: 'a :: comm-ring-1 fps) (fps-const c * fps-X) = Abs-fps (λn. c^n
* f $ n)
  by (simp add: fps-eq-iff fps-compose-def power-mult-distrib
                if-distrib cong: if-cong)

lemma fps-compose-uminus':
  fps-compose f (−fps-X :: 'a :: comm-ring-1 fps) = Abs-fps (λn. (−1)^n * f $ n)
  using fps-compose-linear[of f −1]
  by (simp only: fps-const-neg [symmetric] fps-const-1-eq-1) simp
lemma fps-nth-compose-linear [simp]:
  fixes f :: 'a :: comm-ring-1 fps
  shows fps-nth (fps-compose f (fps-const c * fps-X)) n = c ^ n * fps-nth f n
proof −
  have fps-nth (fps-compose f (fps-const c * fps-X)) n =
        (∑ i∈{n}. fps-nth f i * fps-nth ((fps-const c * fps-X) ^ i) n)
    unfolding fps-compose-nth
    by (intro sum.mono-neutral-cong-right) (auto simp: power-mult-distrib)
  also have ... = c ^ n * fps-nth f n
    by (simp add: power-mult-distrib)
  finally show ?thesis .
qed
```

5.19 Elementary series

5.19.1 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 −
have *?l*$*n* = *?r* $ *n* **for** *n*
using *of-nat-neq-0* **by** (*auto simp add*: *fps-exp-def divide-simps*)
then show *?thesis*
by (*simp add*: *fps-eq-iff*)
qed

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
show *?rhs* **if** *?lhs*
proof −
from *that* **have** *th*: ⋀*n*. *a* $ *Suc n* = *c* ∗ *a*$*n* / *of-nat* (*Suc n*)
by (*simp add*: *fps-deriv-def fps-eq-iff field-simps del*: *of-nat-Suc*)
have *th'*: *a*$*n* = *a*$*0* ∗ *c* ^ *n*/ (*fact n*) **for** *n*
proof (*induct n*)
case *0*
then show *?case* **by** *simp*
next
case *Suc*
then show *?case*
by (*simp add*: *th divide-simps*)
qed
show *?thesis*
by (*auto simp add*: *fps-eq-iff fps-const-mult-left fps-exp-def intro*: *th'*)
qed
show *?lhs* **if** *?rhs*
using *that* **by** (*metis fps-exp-deriv fps-deriv-mult-const-left mult.left-commute*)
qed

lemma *fps-exp-add-mult*: *fps-exp* (*a* + *b*) = *fps-exp* (*a*::'*a*::*field-char-0*) ∗ *fps-exp b* (**is** *?l* = *?r*)
proof −
have *fps-deriv ?r* = *fps-const* (*a* + *b*) ∗ *?r*
by (*simp add*: *fps-const-add*[*symmetric*] *field-simps del*: *fps-const-add*)
then have *?r* = *?l*
by (*simp only*: *fps-exp-unique-ODE*) (*simp add*: *fps-mult-nth fps-exp-def*)
then show *?thesis* **..**
qed

lemma *fps-exp-nth*[*simp*]: *fps-exp a* $ *n* = *a*^*n* / *of-nat* (*fact n*)
by (*simp add*: *fps-exp-def*)

lemma *fps-exp-0*[*simp*]: *fps-exp* (*0*::'*a*::*field*) = *1*

by (*simp add*: *fps-eq-iff power-0-left*)

lemma *fps-exp-neg*: *fps-exp* (− *a*) = *inverse* (*fps-exp* (*a*::′*a*::*field-char-0*))
proof −
from *fps-exp-add-mult*[*of a* − *a*] **have** *th0*: *fps-exp a* ∗ *fps-exp* (− *a*) = *1* **by** *simp*
from *fps-inverse-unique*[*OF th0*] **show** *?thesis* **by** *simp*
qed

lemma *fps-exp-nth-deriv*[*simp*]:
fps-nth-deriv n (*fps-exp* (*a*::′*a*::*field-char-0*)) = (*fps-const a*)^*n* ∗ (*fps-exp a*)
by (*induct n*) *auto*

lemma *fps-X-compose-fps-exp*[*simp*]: *fps-X oo fps-exp* (*a*::′*a*::*field*) = *fps-exp a* − *1*
by (*simp add*: *fps-eq-iff fps-X-fps-compose*)

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 −
let *?b* = *fps-exp a* − *1*
have *b0*: *?b* $ *0* = *0*
by *simp*
have *b1*: *?b* $ *1* ≠ *0*
by (*simp add*: *a*)
from *fps-inv*[*OF b0 b1*] **show** *fps-inv* (*fps-exp a* − *1*) *oo* (*fps-exp a* − *1*) = *fps-X*
.
from *fps-inv-right*[*OF b0 b1*] **show** (*fps-exp a* − *1*) *oo fps-inv* (*fps-exp a* − *1*) = *fps-X* .
qed

lemma *fps-exp-power-mult*: (*fps-exp* (*c*::′*a*::*field-char-0*))^*n* = *fps-exp* (*of-nat n* ∗ *c*)
by (*induct n*) (*simp-all add*: *field-simps fps-exp-add-mult*)

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 −
let *?ck* = (*c* / *of-nat* (*Suc k*))
let *?r* = *fps-radical r* (*Suc k*)
have *eq0*[*simp*]: *?ck* ∗ *of-nat* (*Suc k*) = *c of-nat* (*Suc k*) ∗ *?ck* = *c*
by (*simp-all del*: *of-nat-Suc*)
have *th0*: *fps-exp ?ck* ^ (*Suc k*) = *fps-exp c* **unfolding** *fps-exp-power-mult eq0*
..
have *th*: *r* (*Suc k*) (*fps-exp c* $*0*) ^ *Suc k* = *fps-exp c* $ *0*

r (Suc k) (fps-exp c $ 0) = fps-exp ?ck $ 0 fps-exp c $ 0 ≠ 0 **using** *r* **by** *simp-all*
from *th0 radical-unique*[**where** *r=r* **and** *k=k*, *OF th*] **show** *?thesis*
by *auto*
qed

lemma *fps-exp-compose-linear* [*simp*]:
fps-exp (*d*::*'a*::*field-char-0*) *oo* (*fps-const c * fps-X*) = *fps-exp* (*c * d*)
by (*simp add*: *fps-compose-linear fps-exp-def fps-eq-iff power-mult-distrib*)

lemma *fps-fps-exp-compose-minus* [*simp*]:
fps-compose (*fps-exp c*) (−*fps-X*) = *fps-exp* (−*c* :: *'a* :: *field-char-0*)
using *fps-exp-compose-linear*[*of c −1* :: *'a*]
unfolding *fps-const-neg* [*symmetric*] *fps-const-1-eq-1* **by** *simp*

lemma *fps-exp-eq-iff* [*simp*]: *fps-exp c = fps-exp d* ⟷ *c* = (*d* :: *'a* :: *field-char-0*)
proof
assume *fps-exp c = fps-exp d*
from *arg-cong*[*of - - λF. F $ 1*, *OF this*] **show** *c = d* **by** *simp*
qed *simp-all*

lemma *fps-exp-eq-fps-const-iff* [*simp*]:
fps-exp (*c* :: *'a* :: *field-char-0*) = *fps-const c'* ⟷ *c = 0 ∧ c' = 1*
proof
assume *c = 0 ∧ c' = 1*
thus *fps-exp c = fps-const c'* **by** (*simp add*: *fps-eq-iff*)
next
assume *fps-exp c = fps-const c'*
from *arg-cong*[*of - - λF. F $ 1*, *OF this*] *arg-cong*[*of - - λF. F $ 0*, *OF this*]
show *c = 0 ∧ c' = 1* **by** *simp-all*
qed

lemma *fps-exp-neq-0* [*simp*]: ¬*fps-exp* (*c* :: *'a* :: *field-char-0*) = *0*
unfolding *fps-const-0-eq-0* [*symmetric*] *fps-exp-eq-fps-const-iff* **by** *simp*

lemma *fps-exp-eq-1-iff* [*simp*]: *fps-exp* (*c* :: *'a* :: *field-char-0*) = *1* ⟷ *c = 0*
unfolding *fps-const-1-eq-1* [*symmetric*] *fps-exp-eq-fps-const-iff* **by** *simp*

lemma *fps-exp-neq-numeral-iff* [*simp*]:
fps-exp (*c* :: *'a* :: *field-char-0*) = *numeral n* ⟷ *c = 0 ∧ n = Num.One*
unfolding *numeral-fps-const fps-exp-eq-fps-const-iff* **by** *simp*

5.19.2 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*))
by (*simp add*: *fps-eq-iff*)

definition *fps-ln* :: *'a*::*field-char-0* ⇒ *'a fps*
where *fps-ln c* = *fps-const* (*1*/*c*) ∗ *Abs-fps* (λ*n*. *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*)
unfolding *fps-inverse-fps-X-plus1*
by (*simp add*: *fps-ln-def fps-eq-iff del*: *of-nat-Suc*)

lemma *fps-ln-nth*: *fps-ln c* $ *n* = (*if n* = *0 then 0 else 1*/*c* ∗ ((− *1*) ^ (*n* − *1*) / *of-nat n*))
by (*simp add*: *fps-ln-def field-simps*)

lemma *fps-ln-0* [*simp*]: *fps-ln c* $ *0* = *0* **by** (*simp add*: *fps-ln-def*)

lemma *fps-ln-fps-exp-inv*:
fixes *a* :: *'a*::*field-char-0*
assumes *a*: *a* ≠ *0*
shows *fps-ln a* = *fps-inv* (*fps-exp a* − *1*) (**is** *?l* = *?r*)
proof −
let *?b* = *fps-exp a* − *1*
have *b0*: *?b* $ *0* = *0* **by** *simp*
have *b1*: *?b* $ *1* ≠ *0* **by** (*simp add*: *a*)
have *fps-deriv* (*fps-exp a* − *1*) *oo fps-inv* (*fps-exp a* − *1*) =
(*fps-const a* ∗ (*fps-exp a* − *1*) + *fps-const a*) *oo fps-inv* (*fps-exp a* − *1*)
by (*simp add*: *field-simps*)
also have ... = *fps-const a* ∗ (*fps-X* + *1*)
by (*simp add*: *fps-compose-add-distrib fps-inv-right*[*OF b0 b1*] *distrib-left flip*: *fps-const-mult-apply-left*)
finally have *eq*: *fps-deriv* (*fps-exp a* − *1*) *oo fps-inv* (*fps-exp a* − *1*) = *fps-const a* ∗ (*fps-X* + *1*) .
from *fps-inv-deriv*[*OF b0 b1*, *unfolded eq*]
have *fps-deriv* (*fps-inv ?b*) = *fps-const* (*inverse a*) / (*fps-X* + *1*)
using *a* **by** (*simp add*: *fps-const-inverse eq fps-divide-def fps-inverse-mult*)
then have *fps-deriv ?l* = *fps-deriv ?r*
by (*simp add*: *fps-ln-deriv add.commute fps-divide-def divide-inverse*)
then show *?thesis* **unfolding** *fps-deriv-eq-iff*
by (*simp add*: *fps-ln-nth fps-inv-def*)
qed

lemma *fps-ln-mult-add*:
assumes *c0*: *c*≠*0*
and *d0*: *d*≠*0*
shows *fps-ln c* + *fps-ln d* = *fps-const* (*c*+*d*) ∗ *fps-ln* (*c*∗*d*)
(**is** *?r* = *?l*)
proof−
from *c0 d0* **have** *eq*: *1*/*c* + *1*/*d* = (*c*+*d*)/(*c*∗*d*) **by** (*simp add*: *field-simps*)
have *fps-deriv ?r* = *fps-const* (*1*/*c* + *1*/*d*) ∗ *inverse* (*1* + *fps-X*)
by (*simp add*: *fps-ln-deriv fps-const-add*[*symmetric*] *algebra-simps del*: *fps-const-add*)
also have ... = *fps-deriv ?l*

by (*simp add*: *eq fps-ln-deriv*)
finally show *?thesis*
unfolding *fps-deriv-eq-iff* **by** *simp*
qed

lemma *fps-X-dvd-fps-ln* [*simp*]: *fps-X dvd fps-ln c*
proof −
have *fps-ln c* = *fps-X* ∗ *Abs-fps* (λ*n*. (−*1*) ^ *n* / (*of-nat* (*Suc n*) ∗ *c*))
by (*intro fps-ext*) (*simp add*: *fps-ln-def of-nat-diff*)
thus *?thesis* **by** *simp*
qed

5.19.3 Binomial series

definition *fps-binomial a* = *Abs-fps* (λ*n*. *a gchoose n*)

lemma *fps-binomial-nth*[*simp*]: *fps-binomial a* $ *n* = *a gchoose n*
by (*simp add*: *fps-binomial-def*)

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
let *?da* = *fps-deriv a*
let *?x1* = (*1* + *fps-X*):: *'a fps*
let *?l* = *?x1* ∗ *?da*
let *?r* = *fps-const c* ∗ *a*

have *eq*: *?l* = *?r* ⟷ *?lhs*
proof −
have *x10*: *?x1* $ *0* ≠ *0* **by** *simp*
have *?l* = *?r* ⟷ *inverse ?x1* ∗ *?l* = *inverse ?x1* ∗ *?r* **by** *simp*
also have ... ⟷ *?da* = (*fps-const c* ∗ *a*) / *?x1*
unfolding *fps-divide-def mult.assoc*[*symmetric*] *inverse-mult-eq-1*[*OF x10*]
by (*simp add*: *field-simps*)
finally show *?thesis* **.**
qed

show *?rhs* **if** *?lhs*
proof −
from *eq that* **have** *h*: *?l* = *?r* **..**
have *th0*: *a*$ *Suc n* = ((*c* − *of-nat n*) / *of-nat* (*Suc n*)) ∗ *a* $*n* **for** *n*
proof −
from *h* **have** *?l* $ *n* = *?r* $ *n* **by** *simp*
then show *?thesis*
by (*simp add*: *field-simps del*: *of-nat-Suc split*: *if-split-asm*)
qed

```
    have th1: a $ n = (c gchoose n) * a $ 0 for n
    proof (induct n)
      case 0
      then show ?case by simp
    next
      case (Suc m)
      have (c - of-nat m) * (c gchoose m) = (c gchoose Suc m) * of-nat (Suc m)
        by (metis gbinomial-absorb-comp gbinomial-absorption mult.commute)
      with Suc show ?case
        unfolding th0
        by (simp add: divide-simps del: of-nat-Suc)
    qed
    show ?thesis
    by (metis expand-fps-eq fps-binomial-nth fps-mult-right-const-nth mult.commute
th1)
  qed

  show ?lhs if ?rhs
  proof -
    have th00: x * (a $ 0 * y) = a $ 0 * (x * y) for x y
      by (simp add: mult.commute)
    have ?l = (1 + fps-X) * fps-deriv (fps-const (a $ 0) * fps-binomial c)
      using that by auto
    also have ... = fps-const c * (fps-const (a $ 0) * fps-binomial c)
    proof (clarsimp simp add: fps-eq-iff algebra-simps)
       show a $ 0 * (c gchoose Suc n) + (of-nat n * ((c gchoose n) * a $ 0) +
of-nat n * (a $ 0 * (c gchoose Suc n)))
         = c * ((c gchoose n) * a $ 0) for n
      unfolding mult.assoc[symmetric]
      by (simp add: field-simps gbinomial-mult-1)
  qed
    also have ... = ?r
      using that by auto
    finally have ?l = ?r .
    with eq show ?thesis ..
  qed
qed

lemma fps-binomial-ODE-unique':
  (fps-deriv a = fps-const c * a / (1 + fps-X) ∧ a $ 0 = 1) ⟷ (a = fps-binomial
c)
  by (subst fps-binomial-ODE-unique) auto

lemma fps-binomial-deriv: fps-deriv (fps-binomial c) = fps-const c * fps-binomial
c / (1 + fps-X)
proof -
  let ?a = fps-binomial c
  have th0: ?a = fps-const (?a$0) * ?a by (simp)
  from iffD2[OF fps-binomial-ODE-unique, OF th0] show ?thesis .
```

qed

lemma *fps-binomial-add-mult*: *fps-binomial* (*c*+*d*) = *fps-binomial c* ∗ *fps-binomial d* (**is** *?l* = *?r*)
proof −
  let *?P* = *?r* − *?l*
  let *?b* = *fps-binomial*
  let *?db* = λ*x*. *fps-deriv* (*?b x*)
  have *fps-deriv ?P* = *?db c* ∗ *?b d* + *?b c* ∗ *?db d* − *?db* (*c* + *d*) **by** *simp*
  also have ... = *inverse* (*1* + *fps-X*) ∗
    (*fps-const c* ∗ *?b c* ∗ *?b d* + *fps-const d* ∗ *?b c* ∗ *?b d* − *fps-const* (*c*+*d*) ∗ *?b* (*c* + *d*))
    unfolding *fps-binomial-deriv*
    by (*simp add*: *fps-divide-def field-simps*)
  also have ... = (*fps-const* (*c* + *d*)/ (*1* + *fps-X*)) ∗ *?P*
   by (*simp add*: *field-simps fps-divide-unit fps-const-add*[*symmetric*] *del*: *fps-const-add*)
  finally have *th0*: *fps-deriv ?P* = *fps-const* (*c*+*d*) ∗ *?P* / (*1* + *fps-X*)
    by (*simp add*: *fps-divide-def*)
  have *?P* = *fps-const* (*?P*$*0*) ∗ *?b* (*c* + *d*)
    unfolding *fps-binomial-ODE-unique*[*symmetric*]
    using *th0* **by** *simp*
  then have *?P* = *0* **by** (*simp add*: *fps-mult-nth*)
  then show *?thesis* **by** *simp*
qed

lemma *fps-binomial-minus-one*: *fps-binomial* (− *1*) = *inverse* (*1* + *fps-X*)
  (**is** *?l* = *inverse ?r*)
proof−
  have *th*: *?r*$*0* ≠ *0* **by** *simp*
  have *th'*: *fps-deriv* (*inverse ?r*) = *fps-const* (− *1*) ∗ *inverse ?r* / (*1* + *fps-X*)
    by (*simp add*: *fps-inverse-deriv*[*OF th*] *fps-divide-def*
      power2-eq-square mult.commute fps-const-neg[*symmetric*] *del*: *fps-const-neg*)
  have *eq*: *inverse ?r* $ *0* = *1*
    by (*simp add*: *fps-inverse-def*)
  from *iffD1*[*OF fps-binomial-ODE-unique*[*of inverse* (*1* + *fps-X*) − *1*] *th'*] *eq*
  show *?thesis* **by** (*simp add*: *fps-inverse-def*)
qed

lemma *fps-binomial-of-nat*: *fps-binomial* (*of-nat n*) = (*1* + *fps-X* :: *'a* :: *field-char-0 fps*) ^ *n*
proof (*cases n* = *0*)
  case [*simp*]: *True*
  have *fps-deriv* ((*1* + *fps-X*) ^ *n* :: *'a fps*) = *0* **by** *simp*
  also have ... = *fps-const* (*of-nat n*) ∗ (*1* + *fps-X*) ^ *n* / (*1* + *fps-X*) **by** (*simp add*: *fps-binomial-def*)
 finally show *?thesis* **by** (*subst sym*, *subst fps-binomial-ODE-unique'* [*symmetric*]) *simp-all*
next
  case *False*

have *fps-deriv* ((*1* + *fps-X*) ^ *n* :: *'a fps*) = *fps-const* (*of-nat n*) * (*1* + *fps-X*) ^ (*n* − *1*)
by (*simp add*: *fps-deriv-power*)
also have (*1* + *fps-X* :: *'a fps*) $ *0* ≠ *0* **by** *simp*
hence (*1* + *fps-X* :: *'a fps*) ≠ *0* **by** (*intro notI*) (*simp only*: , *simp*)
with *False* **have** (*1* + *fps-X* :: *'a fps*) ^ (*n* − *1*) = (*1* + *fps-X*) ^ *n* / (*1* + *fps-X*)
by (*cases n*) (*simp-all*)
also have *fps-const* (*of-nat n* :: *'a*) * ((*1* + *fps-X*) ^ *n* / (*1* + *fps-X*)) =
fps-const (*of-nat n*) * (*1* + *fps-X*) ^ *n* / (*1* + *fps-X*)
by (*simp add*: *unit-div-mult-swap*)
finally show *?thesis*
by (*subst sym*, *subst fps-binomial-ODE-unique'* [*symmetric*]) (*simp-all add*: *fps-power-nth*)
qed

lemma *fps-binomial-0* [*simp*]: *fps-binomial 0* = *1*
using *fps-binomial-of-nat*[*of 0*] **by** *simp*

lemma *fps-binomial-power*: *fps-binomial a* ^ *n* = *fps-binomial* (*of-nat n* * *a*)
by (*induction n*) (*simp-all add*: *fps-binomial-add-mult ring-distribs*)

lemma *fps-binomial-1*: *fps-binomial 1* = *1* + *fps-X*
using *fps-binomial-of-nat*[*of 1*] **by** *simp*

lemma *fps-binomial-minus-of-nat*:
fps-binomial (− *of-nat n*) = *inverse* ((*1* + *fps-X* :: *'a* :: *field-char-0 fps*) ^ *n*)
by (*rule sym*, *rule fps-inverse-unique*)
(*simp add*: *fps-binomial-of-nat* [*symmetric*] *fps-binomial-add-mult* [*symmetric*])

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*)
by (*subst fps-binomial-of-nat*)
(*simp add*: *fps-compose-power* [*symmetric*] *fps-compose-add-distrib fps-const-neg* [*symmetric*]
del: *fps-const-neg*)

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 (*cases c* = *0*)
case *False*
thus *?thesis*
by (*subst fps-binomial-minus-of-nat*)
(*simp add*: *fps-compose-power* [*symmetric*] *fps-inverse-compose fps-compose-add-distrib*
fps-const-neg [*symmetric*] *del*: *fps-const-neg*)
qed *simp*

lemma *fps-X-plus-const-power*:
  $c \neq 0 \Longrightarrow (\mathit{fps\text{-}X} + \mathit{fps\text{-}const}\ c)\ \hat{}\ n =$
    fps-const $(c\hat{}n) *$ *fps-compose* (*fps-binomial* (*of-nat* n)) (*fps-const* (*inverse* c)
$*$ *fps-X*)
  by (*subst fps-binomial-of-nat*)
   (*simp add*: *fps-compose-power* [*symmetric*] *fps-binomial-of-nat fps-compose-add-distrib*
            fps-const-power [*symmetric*] *power-mult-distrib* [*symmetric*]
            algebra-simps inverse-mult-eq-1' *del*: *fps-const-power*)

lemma *fps-X-plus-const-neg-power*:
  $c \neq 0 \Longrightarrow$ *inverse* $((\mathit{fps\text{-}X} + \mathit{fps\text{-}const}\ c)\ \hat{}\ n) =$
     fps-const (*inverse* $c\hat{}n$) $*$ *fps-compose* (*fps-binomial* ($-$*of-nat* n)) (*fps-const*
(*inverse* c) $*$ *fps-X*)
  by (*subst fps-binomial-minus-of-nat*)
   (*simp add*: *fps-compose-power* [*symmetric*] *fps-binomial-of-nat fps-compose-add-distrib*
       fps-const-power [*symmetric*] *power-mult-distrib* [*symmetric*] *fps-inverse-compose*

        algebra-simps fps-const-inverse [*symmetric*] *fps-inverse-mult* [*symmetric*]
          fps-inverse-power [*symmetric*] *inverse-mult-eq-1'*
       del: *fps-const-power*)

lemma *one-minus-const-fps-X-neg-power'*:
  fixes $c :: 'a :: \mathit{field\text{-}char\text{-}0}$
  assumes $n > 0$
  shows *inverse* $((1 - \mathit{fps\text{-}const}\ c * \mathit{fps\text{-}X})\ \hat{}\ n) = \mathit{Abs\text{-}fps}\ (\lambda k.\ \mathit{of\text{-}nat}\ ((n + k -$
$1)$ *choose* $k) * c\hat{}k)$
proof $-$
  have §: $\bigwedge j.\ \mathit{Abs\text{-}fps}\ (\lambda na.\ (-\ c)\ \hat{}\ na * \mathit{fps\text{-}binomial}\ (-\ \mathit{of\text{-}nat}\ n)\ \$\ na)\ \$\ j =$
       Abs-fps $(\lambda k.$ *of-nat* $(n + k - 1$ *choose* $k) * c\ \hat{}\ k)\ \$\ j$
    using *assms*
   by (*simp add*: *gbinomial-minus binomial-gbinomial of-nat-diff flip*: *power-mult-distrib*
mult.assoc)
  show *?thesis*
    apply (*rule fps-ext*)
    using §
     by (*metis* (*no-types*, *lifting*) *one-minus-fps-X-const-neg-power fps-const-neg*
fps-compose-linear fps-nth-Abs-fps)
qed

Vandermonde's Identity as a consequence.

lemma *gbinomial-Vandermonde*:
  sum $(\lambda k.$ (a *gchoose* k) $*$ (b *gchoose* $(n - k)$)) $\{0..n\} = (a + b)$ *gchoose* n
proof $-$
  let *?ba* = *fps-binomial* a
  let *?bb* = *fps-binomial* b
  let *?bab* = *fps-binomial* $(a + b)$
  from *fps-binomial-add-mult*[*of* a b] **have** *?bab* \$ n = (*?ba* $*$ *?bb*)\$$n$ **by** *simp*
  then show *?thesis* **by** (*simp add*: *fps-mult-nth*)

qed

lemma *binomial-Vandermonde*:
sum (λ*k*. (*a choose k*) ∗ (*b choose* (*n* − *k*))) {*0*..*n*} = (*a* + *b*) *choose n*
using *gbinomial-Vandermonde*[*of* (*of-nat a*) *of-nat b n*]
by (*simp only*: *binomial-gbinomial*[*symmetric*] *of-nat-mult*[*symmetric*]
of-nat-sum[*symmetric*] *of-nat-add*[*symmetric*] *of-nat-eq-iff*)

lemma *binomial-Vandermonde-same*: *sum* (λ*k*. (*n choose k*)^2) {*0*..*n*} = (*2* ∗ *n*) *choose n*
using *binomial-Vandermonde*[*of n n n*, *symmetric*]
unfolding *mult-2*
by (*metis atMost-atLeast0 choose-square-sum mult-2*)

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 −
let *?m1* = λ*m*. (− *1* :: *'a*) ^ *m*
let *?f* = λ*m*. *of-nat* (*fact m*)
let *?p* = λ(*x*::*'a*). *pochhammer* (− *x*)
from *b* **have** *bn0*: *?p b n* ≠ *0*
unfolding *pochhammer-eq-0-iff* **by** *simp*
have *th00*:
b gchoose (*n* − *k*) =
(*?m1 n* ∗ *?p b n* ∗ *?m1 k* ∗ *?p* (*of-nat n*) *k*) / (*?f n* ∗ *pochhammer* (*b* − *of-nat n* + *1*) *k*)
(**is** *?gchoose*)
pochhammer (*1* + *b* − *of-nat n*) *k* ≠ *0*
(**is** *?pochhammer*)
if *kn*: *k* ∈ {*0*..*n*} **for** *k*
proof −
from *kn* **have** *k* ≤ *n* **by** *simp*
have *nz*: *pochhammer* (*1* + *b* − *of-nat n*) *n* ≠ *0*
proof
assume *pochhammer* (*1* + *b* − *of-nat n*) *n* = *0*
then have *c*: *pochhammer* (*b* − *of-nat n* + *1*) *n* = *0*
by (*simp add*: *algebra-simps*)
then obtain *j* **where** *j*: *j* < *n b* − *of-nat n* + *1* = − *of-nat j*
unfolding *pochhammer-eq-0-iff* **by** *blast*
from *j* **have** *b* = *of-nat n* − *of-nat j* − *of-nat 1*
by (*simp add*: *algebra-simps*)
then show *False*
using ‹*j* < *n*› *j b*
by (*metis bn0 c mult-cancel-right2 pochhammer-minus*)

```
  qed

  from nz kn [simplified] have nz': pochhammer (1 + b − of-nat n) k ≠ 0
    by (rule pochhammer-neq-0-mono)

  consider k = 0 ∨ n = 0 | k ≠ 0 n ≠ 0
    by blast
  then have b gchoose (n − k) =
   (?m1 n * ?p b n * ?m1 k * ?p (of-nat n) k) / (?f n * pochhammer (b − of-nat
n + 1) k)
  proof cases
    case 1
    then show ?thesis
      using kn by (cases k = 0) (simp-all add: gbinomial-pochhammer)
  next
    case neq: 2
    then obtain m where m: n = Suc m
      by (cases n) auto
    from neq(1) obtain h where h: k = Suc h
      by (cases k) auto
    show ?thesis
    proof (cases k = n)
      case True
      with pochhammer-minus'[where k=k and b=b] bn0 show ?thesis
        by (simp add: pochhammer-same)
    next
      case False
      with kn have kn': k < n
        by simp
      have h ≤ m
        using ‹k ≤ n› h m by blast
      have m1nk: ?m1 n = prod (λi. − 1) {..m} ?m1 k = prod (λi. − 1) {0..h}
        by (simp-all add: m h)
      have bnz0: pochhammer (b − of-nat n + 1) k ≠ 0
        using bn0 kn
        unfolding pochhammer-eq-0-iff
        by (metis add.commute add-diff-eq nz' pochhammer-eq-0-iff)
      have eq1: prod (λk. (1::'a) + of-nat m − of-nat k) {..h} =
        prod of-nat {Suc (m − h) .. Suc m}
        using kn' h m
       by (intro prod.reindex-bij-witness[where i=λk. Suc m − k and j=λk. Suc
m − k])
          (auto simp: of-nat-diff)
       have (∏ i = 0..<k. 1 + of-nat n − of-nat k + of-nat i) = (∏ x = n −
k..<n. (1::'a) + of-nat x)
        using ‹k ≤ n›
          using prod.atLeastLessThan-shift-bounds [where ?'a = 'a, of λi. 1 +
of-nat i 0 n − k k]
        by (auto simp add: of-nat-diff field-simps)
```

```
      then have fact (n − k) ∗ pochhammer ((1::'a) + of-nat n − of-nat k) k =
fact n
        using ‹k ≤ n›
        by (auto simp add: fact-split [of k n] pochhammer-prod field-simps)
      then have th1: (?m1 k ∗ ?p (of-nat n) k) / ?f n = 1 / of-nat(fact (n − k))
        by (simp add: pochhammer-minus field-simps)
       have ?m1 n ∗ ?p b n = pochhammer (b − of-nat m) (Suc m)
        by (simp add: pochhammer-minus field-simps m)
       also have ... = (∏ i = 0..m. b − of-nat i)
      by (auto simp add: pochhammer-prod-rev of-nat-diff prod.atLeast-Suc-atMost-Suc-shift
simp del: prod.cl-ivl-Suc)
       finally have th20: ?m1 n ∗ ?p b n = prod (λi. b − of-nat i) {0..m} .
       have (∏ x = 0..h. b − of-nat m + of-nat (h − x)) = (∏ i = m − h..m. b
− of-nat i)
        using ‹h ≤ m› prod.atLeastAtMost-shift-0 [of m − h m, where ?'a = 'a]
        by (auto simp add: of-nat-diff field-simps)
       then have th21:pochhammer (b − of-nat n + 1) k = prod (λi. b − of-nat
i) {n − k .. n − 1}
      using kn by (simp add: pochhammer-prod-rev m h prod.atLeast-Suc-atMost-Suc-shift
del: prod.op-ivl-Suc del: prod.cl-ivl-Suc)
       have ?m1 n ∗ ?p b n =
         prod (λi. b − of-nat i) {0.. n − k − 1} ∗ pochhammer (b − of-nat n +
1) k
        using kn' m h unfolding th20 th21
        by (auto simp flip: prod.union-disjoint intro: prod.cong)
       then have th2: (?m1 n ∗ ?p b n)/pochhammer (b − of-nat n + 1) k =
        prod (λi. b − of-nat i) {0.. n − k − 1}
        using nz' by (simp add: field-simps)
      have (?m1 n ∗ ?p b n ∗ ?m1 k ∗ ?p (of-nat n) k) / (?f n ∗ pochhammer (b
− of-nat n + 1) k) =
        ((?m1 k ∗ ?p (of-nat n) k) / ?f n) ∗ ((?m1 n ∗ ?p b n)/pochhammer (b −
of-nat n + 1) k)
        using bnz0
        by (simp add: field-simps)
       also have ... = b gchoose (n − k)
        unfolding th1 th2
        using kn' m h
        by (auto simp: field-simps gbinomial-mult-fact intro: prod.cong)
       finally show ?thesis by simp
     qed
   qed
   then show ?gchoose and ?pochhammer
     using nz' by force+
 qed
 have ?r = ((a + b) gchoose n) ∗ (of-nat (fact n) / (?m1 n ∗ pochhammer (− b)
n))
   unfolding gbinomial-pochhammer
   using bn0 by (auto simp add: field-simps)
 also have ... = ?l
```

using *bn0*
unfolding *gbinomial-Vandermonde*[*symmetric*]
apply (*simp add*: *th00*)
by (*simp add*: *gbinomial-pochhammer sum-distrib-right sum-distrib-left field-simps*)
finally show *?thesis* **by** *simp*
qed

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 −
let *?a* = − *a*
let *?b* = *c* + *of-nat n* − *1*
have *h*: *?b* ≠ *of-nat j* **if** *j* < *n* **for** *j*
proof −
have *c* ≠ − *of-nat* (*n* − *j* − *1*)
using *c that* **by** (*auto simp*: *dest*!: *bspec* [**where** *x* = *n*−*j*−*1*])
with *that* **show** *?thesis*
by (*auto simp add*: *algebra-simps of-nat-diff*)
qed
have *th0*: *pochhammer* (− (*?a* + *?b*)) *n* = (− *1*)^*n* ∗ *pochhammer* (*c* − *a*) *n*
unfolding *pochhammer-minus*
by (*simp add*: *algebra-simps*)
have *th1*: *pochhammer* (− *?b*) *n* = (− *1*)^*n* ∗ *pochhammer c n*
unfolding *pochhammer-minus*
by *simp*
have *nz*: *pochhammer c n* ≠ *0* **using** *c*
by (*simp add*: *pochhammer-eq-0-iff*)
from *Vandermonde-pochhammer-lemma*[**where** *a* = *?a* **and** *b*=*?b* **and** *n*=*n*, *OF h*, *unfolded th0 th1*]
show *?thesis*
using *nz* **by** (*simp add*: *field-simps sum-distrib-left*)
qed

5.19.4 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*
by (*intro fps-ext*) (*auto simp*: *fps-sin-def elim*!: *oddE*)

lemma *fps-cos-0* [*simp*]: *fps-cos 0 = 1*
by (*intro fps-ext*) (*simp add*: *fps-cos-def*)

lemma *fps-sin-deriv*:
fps-deriv (*fps-sin c*) = *fps-const c* ∗ *fps-cos c*
(**is** *?lhs* = *?rhs*)
proof (*rule fps-ext*)
fix *n* :: *nat*
show *?lhs* $ *n* = *?rhs* $ *n*
proof (*cases even n*)
case *True*
have *?lhs*$*n* = *of-nat* (*n*+1) ∗ (*fps-sin c* $ (*n*+1)) **by** *simp*
also have ... = *of-nat* (*n*+1) ∗ ((− 1)^(*n div 2*) ∗ *c*^*Suc n* / *of-nat* (*fact* (*Suc n*)))
using *True* **by** (*simp add*: *fps-sin-def*)
also have ... = (− 1)^(*n div 2*) ∗ *c*^*Suc n* ∗ (*of-nat* (*n*+1) / (*of-nat* (*Suc n*) ∗ *of-nat* (*fact n*)))
unfolding *fact-Suc of-nat-mult*
by (*simp add*: *field-simps del*: *of-nat-add of-nat-Suc*)
also have ... = (− 1)^(*n div 2*) ∗ *c*^*Suc n* / *of-nat* (*fact n*)
by (*simp add*: *field-simps del*: *of-nat-add of-nat-Suc*)
finally show *?thesis*
using *True* **by** (*simp add*: *fps-cos-def field-simps*)
next
case *False*
then show *?thesis*
by (*simp-all add*: *fps-deriv-def fps-sin-def fps-cos-def*)
qed
qed

lemma *fps-cos-deriv*: *fps-deriv* (*fps-cos c*) = *fps-const* (− *c*)∗ (*fps-sin c*)
(**is** *?lhs* = *?rhs*)
proof (*rule fps-ext*)
have *th0*: − ((− 1::'*a*) ^ *n*) = (− 1)^*Suc n* **for** *n*
by *simp*
show *?lhs* $ *n* = *?rhs* $ *n* **for** *n*
proof (*cases even n*)
case *False*
then have *n0*: *n* ≠ *0* **by** *presburger*
from *False* **have** *th1*: *Suc* ((*n* − *1*) *div 2*) = *Suc n div 2*
by (*cases n*) *simp-all*
have *?lhs*$*n* = *of-nat* (*n*+1) ∗ (*fps-cos c* $ (*n*+1)) **by** *simp*
also have ... = *of-nat* (*n*+1) ∗ ((− 1)^((*n* + *1*) *div 2*) ∗ *c*^*Suc n* / *of-nat* (*fact* (*Suc n*)))
using *False* **by** (*simp add*: *fps-cos-def*)
also have ... = (− 1)^((*n* + *1*) *div 2*) ∗ *c*^*Suc n* ∗ (*of-nat* (*n*+1) / (*of-nat* (*Suc n*) ∗ *of-nat* (*fact n*)))
unfolding *fact-Suc of-nat-mult*
by (*simp add*: *field-simps del*: *of-nat-add of-nat-Suc*)

```
    also have ... = (- 1)^((n + 1) div 2) * c^Suc n / of-nat (fact n)
      by (simp add: field-simps del: of-nat-add of-nat-Suc)
    also have ... = (- ((- 1)^((n - 1) div 2))) * c^Suc n / of-nat (fact n)
      unfolding th0 unfolding th1 by simp
    finally show ?thesis
      using False by (simp add: fps-sin-def field-simps)
  next
    case True
    then show ?thesis
      by (simp-all add: fps-deriv-def fps-sin-def fps-cos-def)
  qed
qed

lemma fps-sin-cos-sum-of-squares: (fps-cos c)^2 + (fps-sin c)^2 = 1
  (is ?lhs = -)
proof -
  have fps-deriv ?lhs = 0
   by (simp add: fps-deriv-power fps-sin-deriv fps-cos-deriv field-simps flip: fps-const-neg)
  then have ?lhs = fps-const (?lhs $ 0)
    unfolding fps-deriv-eq-0-iff .
  also have ... = 1
    by (simp add: fps-eq-iff numeral-2-eq-2 fps-mult-nth fps-cos-def fps-sin-def)
  finally show ?thesis .
qed

lemma fps-sin-nth-0 [simp]: fps-sin c $ 0 = 0
  unfolding fps-sin-def by simp

lemma fps-sin-nth-1 [simp]: fps-sin c $ Suc 0 = c
  unfolding fps-sin-def by simp

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 (cases n)
  case (Suc n')
  then show ?thesis
    unfolding fps-sin-def by (simp add: field-simps)
qed (auto simp: fps-sin-def)


lemma fps-cos-nth-0 [simp]: fps-cos c $ 0 = 1
  unfolding fps-cos-def by simp

lemma fps-cos-nth-1 [simp]: fps-cos c $ Suc 0 = 0
  unfolding fps-cos-def by simp

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 (*cases n*)
  case (*Suc n'*)
  then show *?thesis*
    unfolding *fps-cos-def* **by** (*simp add*: *field-simps*)
qed (*auto simp*: *fps-cos-def*)

lemma *nat-add-1-add-1*: (*n*::*nat*) + *1* + *1* = *n* + *2*
  by *simp*

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 (*rule fps-ext*)
  fix *n*
  show *fps-sin c* $ *n* = *a* $ *n*
  proof (*induction n rule*: *nat-induct2*)
    case (*step n*)
    then have *of-nat* (*n* + *1*) ∗ (*of-nat* (*n* + *2*) ∗ *a* $ (*n* + *2*)) =
     − (*c* ∗ *c* ∗ *fps-sin c* $ *n*)
      using *a2*
    by (*metis fps-const-mult fps-deriv-nth fps-mult-left-const-nth fps-neg-nth nat-add-1-add-1*)
    with *step* **show** *?case*
     by (*metis* (*no-types*, *lifting*) *a0 add.commute add.inverse-inverse fps-sin-nth-0*
fps-sin-nth-add-2 mult-divide-mult-cancel-left-if mult-minus-right nonzero-mult-div-cancel-left
not-less-zero of-nat-eq-0-iff plus-1-eq-Suc zero-less-Suc)
  qed (*use assms* **in** *auto*)
qed

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 (*rule fps-ext*)
  fix *n*
  show *fps-cos c* $ *n* = *a* $ *n*
  proof (*induction n rule*: *nat-induct2*)
    case (*step n*)
    then have *of-nat* (*n* + *1*) ∗ (*of-nat* (*n* + *2*) ∗ *a* $ (*n* + *2*)) =
     − (*c* ∗ *c* ∗ *fps-cos c* $ *n*)
      using *a2*
    by (*metis fps-const-mult fps-deriv-nth fps-mult-left-const-nth fps-neg-nth nat-add-1-add-1*)
    with *step* **show** *?case*
     by (*metis* (*no-types*, *lifting*) *a0 add.commute add.inverse-inverse fps-cos-nth-0*
fps-cos-nth-add-2 mult-divide-mult-cancel-left-if mult-minus-right nonzero-mult-div-cancel-left
not-less-zero of-nat-eq-0-iff plus-1-eq-Suc zero-less-Suc)

qed (*use assms* **in** *auto*)
qed

lemma *fps-sin-add*: *fps-sin* (*a* + *b*) = *fps-sin a* ∗ *fps-cos b* + *fps-cos a* ∗ *fps-sin b*
proof −
have *fps-deriv* (*fps-deriv* (*fps-sin a* ∗ *fps-cos b* + *fps-cos a* ∗ *fps-sin b*)) =
− (*fps-const* (*a* + *b*) ∗ *fps-const* (*a* + *b*) ∗ (*fps-sin a* ∗ *fps-cos b* + *fps-cos a* ∗ *fps-sin b*))
by (*simp flip*: *fps-const-neg fps-const-add fps-const-mult*
add: *fps-sin-deriv fps-cos-deriv algebra-simps*)
then show *?thesis*
by (*auto intro*: *eq-fps-sin*)
qed

lemma *fps-cos-add*: *fps-cos* (*a* + *b*) = *fps-cos a* ∗ *fps-cos b* − *fps-sin a* ∗ *fps-sin b*
proof −
have *fps-deriv*
(*fps-deriv* (*fps-cos a* ∗ *fps-cos b* − *fps-sin a* ∗ *fps-sin b*)) =
− (*fps-const* (*a* + *b*) ∗ *fps-const* (*a* + *b*) ∗
(*fps-cos a* ∗ *fps-cos b* − *fps-sin a* ∗ *fps-sin b*))
by (*simp flip*: *fps-const-neg fps-const-add fps-const-mult*
add: *fps-sin-deriv fps-cos-deriv algebra-simps*)
then show *?thesis*
by (*auto intro*: *eq-fps-cos*)
qed

lemma *fps-sin-even*: *fps-sin* (− *c*) = − *fps-sin c*
by (*simp add*: *fps-eq-iff fps-sin-def*)

lemma *fps-cos-odd*: *fps-cos* (− *c*) = *fps-cos c*
by (*simp add*: *fps-eq-iff fps-cos-def*)

definition *fps-tan c* = *fps-sin c* / *fps-cos c*

lemma *fps-tan-0* [*simp*]: *fps-tan 0* = *0*
by (*simp add*: *fps-tan-def*)

lemma *fps-tan-deriv*: *fps-deriv* (*fps-tan c*) = *fps-const c* / (*fps-cos c*)^2
proof −
have *th0*: *fps-cos c* $ *0* ≠ *0* **by** (*simp add*: *fps-cos-def*)
from *this* **have** *fps-cos c* ≠ *0* **by** (*intro notI*) *simp*
hence *fps-deriv* (*fps-tan c*) =
fps-const c ∗ (*fps-cos c*^*2* + *fps-sin c*^*2*) / (*fps-cos c*^*2*)
by (*simp add*: *fps-tan-def fps-divide-deriv power2-eq-square algebra-simps*
fps-sin-deriv fps-cos-deriv fps-const-neg[*symmetric*] *div-mult-swap*
del: *fps-const-neg*)
also note *fps-sin-cos-sum-of-squares*
finally show *?thesis* **by** *simp*

qed

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 −
have *?l* $ *n* = *?r* $ *n* **for** *n*
proof (*cases even n*)
case *True*
then obtain *m* **where** *m*: *n* = *2* ∗ *m* **..**
show *?thesis*
by (*simp add*: *m fps-sin-def fps-cos-def power-mult-distrib power-mult power-minus*
[*of c* ^ *2*])
next
case *False*
then obtain *m* **where** *m*: *n* = *2* ∗ *m* + *1* **..**
show *?thesis*
by (*simp add*: *m fps-sin-def fps-cos-def power-mult-distrib*
power-mult power-minus [*of c* ^ *2*])
qed
then show *?thesis*
by (*simp add*: *fps-eq-iff*)
qed

lemma *fps-exp-minus-ii-sin-cos*: *fps-exp* (− (i ∗ *c*)) = *fps-cos c* − *fps-const* i ∗
fps-sin c
unfolding *minus-mult-right fps-exp-ii-sin-cos* **by** (*simp add*: *fps-sin-even fps-cos-odd*)

lemma *fps-cos-fps-exp-ii*: *fps-cos c* = (*fps-exp* (i ∗ *c*) + *fps-exp* (− i ∗ *c*)) /
fps-const 2
proof −
have *th*: *fps-cos c* + *fps-cos c* = *fps-cos c* ∗ *fps-const 2*
by (*simp add*: *numeral-fps-const*)
show *?thesis*
unfolding *fps-exp-ii-sin-cos minus-mult-commute*
by (*simp add*: *fps-sin-even fps-cos-odd numeral-fps-const fps-divide-unit fps-const-inverse*
th)
qed

lemma *fps-sin-fps-exp-ii*: *fps-sin c* = (*fps-exp* (i ∗ *c*) − *fps-exp* (− i ∗ *c*)) / *fps-const*
(*2*∗i)
proof −
have *th*: *fps-const* i ∗ *fps-sin c* + *fps-const* i ∗ *fps-sin c* = *fps-sin c* ∗ *fps-const*
(*2* ∗ i)
by (*simp add*: *fps-eq-iff numeral-fps-const*)
show *?thesis*
unfolding *fps-exp-ii-sin-cos minus-mult-commute*
by (*simp add*: *fps-sin-even fps-cos-odd fps-divide-unit fps-const-inverse th*)
qed

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*)))
unfolding *fps-tan-def fps-sin-fps-exp-ii fps-cos-fps-exp-ii*
by (*simp add*: *fps-divide-unit fps-inverse-mult fps-const-inverse*)

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*)
unfolding *fps-exp-ii-sin-cos*[*symmetric*] *fps-exp-power-mult*
by (*simp add*: *ac-simps*)

5.20 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*))
by (*simp add*: *fps-hypergeo-def*)

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*
by (*induct as arbitrary*: *x v*) (*auto simp add*: *algebra-simps*)

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*
by (*induct as arbitrary*: *v*) (*simp-all add*: *foldl-mult-start*)

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*))
by (*simp add*: *foldl-mult-start foldr-mult-foldl*)

lemma *fps-hypergeo-fps-exp*[*simp*]: *fps-hypergeo* [] [] *c* = *fps-exp c*
by (*simp add*: *fps-eq-iff*)

lemma *fps-hypergeo-1-0*[*simp*]: *fps-hypergeo* [*1*] [] *c* = *1*/(*1* − *fps-const c* ∗ *fps-X*)
proof −
let *?a* = (*Abs-fps* (λ*n*. *1*)) *oo* (*fps-const c* ∗ *fps-X*)
have *th0*: (*fps-const c* ∗ *fps-X*) $ *0* = *0* **by** *simp*
show *?thesis* **unfolding** *gp*[*OF th0*, *symmetric*]
by (*simp add*: *fps-eq-iff pochhammer-fact*[*symmetric*]
fps-compose-nth power-mult-distrib if-distrib cong del: *if-weak-cong*)

qed

lemma *fps-hypergeo-B*[*simp*]: *fps-hypergeo* [−*a*] [] (− *1*) = *fps-binomial a*
by (*simp add*: *fps-eq-iff gbinomial-pochhammer algebra-simps*)

lemma *fps-hypergeo-0*[*simp*]: *fps-hypergeo as bs c* $ *0* = *1*
proof −
have *foldl* (λ(*r*::′*a*) (*a*::′*a*). *r*) *1 as* = *1* **for** *as*
by (*induction as*) *auto*
then show *?thesis*
by *auto*
qed

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*
by (*induct as arbitrary*: *v w*) (*simp-all add*: *algebra-simps*)

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*
apply (*simp add*: *foldl-mult-start del*: *of-nat-Suc of-nat-add fact-Suc*)
unfolding *foldl-prod-prod*[*unfolded foldl-mult-start*] *pochhammer-Suc*
by (*simp add*: *algebra-simps*)

lemma *fps-XD-nth*[*simp*]: *fps-XD a* $ *n* = *of-nat n* ∗ *a*$*n*
by (*simp add*: *fps-XD-def*)

lemma *fps-XD-0th*[*simp*]: *fps-XD a* $ *0* = *0*
by *simp*
lemma *fps-XD-Suc*[*simp*]: *fps-XD a* $ *Suc n* = *of-nat* (*Suc n*) ∗ *a* $ *Suc n*
by *simp*

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*
by (*simp add*: *fps-XDp-def algebra-simps*)

lemma *fps-XDp-commute*: *fps-XDp b* ∘ *fps-XDp* (*c*::′*a*::*comm-ring-1*) = *fps-XDp c* ∘ *fps-XDp b*
by (*simp add*: *fps-XDp-def fun-eq-iff fps-eq-iff algebra-simps*)

lemma *fps-XDp0* [*simp*]: *fps-XDp 0* = *fps-XD*
by (*simp add*: *fun-eq-iff fps-eq-iff*)

lemma *fps-XDp-fps-integral* [*simp*]:

fixes $a :: 'a::\{division\text{-}ring,ring\text{-}char\text{-}0\}$ *fps*
shows *fps-XDp 0* (*fps-integral a c*) = *fps-X* ∗ *a*
using *fps-deriv-fps-integral*[*of a c*]
by (*simp add*: *fps-XD-def*)

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)
by (*simp-all add*: *pochhammer-eq-0-iff*)

lemma *pochhammer-rec-if*: *pochhammer a n* = (*if n* = *0 then 1 else a* ∗ *pochhammer* (*a* + *1*) (*n* − *1*))
by (*cases n*) (*simp-all add*: *pochhammer-rec*)

lemma *fps-XDp-foldr-nth* [*simp*]: *foldr* (λ*c r*. *fps-XDp c* ∘ *r*) *cs* (λ*c*. *fps-XDp c a*) *c0* $ *n* =
foldr (λ*c r*. (*c* + *of-nat n*) ∗ *r*) *cs* (*c0* + *of-nat n*) ∗ *a*$*n*
by (*induct cs arbitrary*: *c0*) (*simp-all add*: *algebra-simps*)

lemma *genric-fps-XDp-foldr-nth*:
assumes *f*: ∀ *n c a*. *f c a* $ *n* = (*of-nat n* + *k c*) ∗ *a*$*n*
shows *foldr* (λ*c r*. *f c* ∘ *r*) *cs* (λ*c*. *g c a*) *c0* $ *n* =
foldr (λ*c r*. (*k c* + *of-nat n*) ∗ *r*) *cs* (*g c0 a* $ *n*)
by (*induct cs arbitrary*: *c0*) (*simp-all add*: *algebra-simps f*)

lemma *dist-less-imp-nth-equal*:
assumes *dist f g* < *inverse* (*2* ^ *i*)
and*j* ≤ *i*
shows *f* $ *j* = *g* $ *j*
proof (*rule ccontr*)
assume *f* $ *j* ≠ *g* $ *j*
hence *f* ≠ *g* **by** *auto*
with *assms* **have** *i* < *subdegree* (*f* − *g*)
by (*simp add*: *if-split-asm dist-fps-def*)
also have ... ≤ *j*
using ‹*f* $ *j* ≠ *g* $ *j*› **by** (*intro subdegree-leI*) *simp-all*
finally show *False* **using** ‹*j* ≤ *i*› **by** *simp*
qed

lemma *nth-equal-imp-dist-less*:
assumes ⋀*j*. *j* ≤ *i* ⟹ *f* $ *j* = *g* $ *j*

```
  shows dist f g < inverse (2 ^ i)
proof (cases f = g)
  case True
  then show ?thesis by simp
next
  case False
  with assms have dist f g = inverse (2 ^ subdegree (f - g))
    by (simp add: if-split-asm dist-fps-def)
  moreover
  from assms and False have i < subdegree (f - g)
    by (intro subdegree-greaterI) simp-all
  ultimately show ?thesis by simp
qed

lemma dist-less-eq-nth-equal: dist f g < inverse (2 ^ i) ⟷ (∀j ≤ i. f $ j = g $
j)
  using dist-less-imp-nth-equal nth-equal-imp-dist-less by blast

instance fps :: (comm-ring-1) complete-space
proof
  fix fps-X :: nat ⇒ 'a fps
  assume Cauchy fps-X
  obtain M where M: ∀i. ∀m ≥ M i. ∀j ≤ i. fps-X (M i) $ j = fps-X m $ j
  proof -
    have ∃M. ∀m ≥ M. ∀j≤i. fps-X M $ j = fps-X m $ j for i
    proof -
      have 0 < inverse ((2::real)^i) by simp
      from metric-CauchyD[OF ‹Cauchy fps-X› this] dist-less-imp-nth-equal
      show ?thesis by blast
    qed
    then show ?thesis using that by metis
  qed

  show convergent fps-X
  proof (rule convergentI)
    show fps-X ⟶ Abs-fps (λi. fps-X (M i) $ i)
      unfolding tendsto-iff
    proof safe
      fix e::real assume e: 0 < e
    have (λn. inverse (2 ^ n) :: real) ⟶ 0 by (rule LIMSEQ-inverse-realpow-zero)
simp-all
      from this and e have eventually (λi. inverse (2 ^ i) < e) sequentially
        by (rule order-tendstoD)
      then obtain i where inverse (2 ^ i) < e
        by (auto simp: eventually-sequentially)
      have eventually (λx. M i ≤ x) sequentially
        by (auto simp: eventually-sequentially)
      then show eventually (λx. dist (fps-X x) (Abs-fps (λi. fps-X (M i) $ i)) <
e) sequentially
```

```
      proof eventually-elim
        fix x
        assume x: M i ≤ x
        have fps-X (M i) $ j = fps-X (M j) $ j if j ≤ i for j
          using M that by (metis nat-le-linear)
        with x have dist (fps-X x) (Abs-fps (λj. fps-X (M j) $ j)) < inverse (2 ^ i)
          using M by (force simp: dist-less-eq-nth-equal)
        also note ‹inverse (2 ^ i) < e›
        finally show dist (fps-X x) (Abs-fps (λj. fps-X (M j) $ j)) < e .
      qed
    qed
  qed
qed


bundle fps-syntax
begin
notation fps-nth (infixl ‹$› 75)
end

unbundle no fps-syntax

end
```

6 Converting polynomials to formal power series

```
theory Polynomial-FPS
  imports Polynomial Formal-Power-Series
begin

context
  includes fps-syntax
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
  by (simp add: fps-of-poly-def poly-eq-iff fps-eq-iff)

lemma fps-of-poly-nth [simp]: fps-of-poly p $ n = coeff p n
  by (simp add: fps-of-poly-def)

lemma fps-of-poly-const: fps-of-poly [:c:] = fps-const c
proof (subst fps-eq-iff, clarify)
  fix n :: nat show fps-of-poly [:c:] $ n = fps-const c $ n
    by (cases n) (auto simp: fps-of-poly-def)
qed
```

lemma *fps-of-poly-0* [*simp*]: *fps-of-poly 0 = 0*
by (*subst fps-const-0-eq-0* [*symmetric*], *subst fps-of-poly-const* [*symmetric*]) *simp*

lemma *fps-of-poly-1* [*simp*]: *fps-of-poly 1 = 1*
by (*simp add*: *fps-eq-iff*)

lemma *fps-of-poly-1'* [*simp*]: *fps-of-poly* [:*1*:] = *1*
by (*subst fps-const-1-eq-1* [*symmetric*], *subst fps-of-poly-const* [*symmetric*])
(*simp add*: *one-poly-def*)

lemma *fps-of-poly-numeral* [*simp*]: *fps-of-poly* (*numeral n*) = *numeral n*
by (*simp add*: *numeral-fps-const fps-of-poly-const* [*symmetric*] *numeral-poly*)

lemma *fps-of-poly-numeral'* [*simp*]: *fps-of-poly* [:*numeral n*:] = *numeral n*
by (*simp add*: *numeral-fps-const fps-of-poly-const* [*symmetric*] *numeral-poly*)

lemma *fps-of-poly-fps-X* [*simp*]: *fps-of-poly* [:*0*, *1*:] = *fps-X*
by (*auto simp add*: *fps-of-poly-def fps-eq-iff coeff-pCons split*: *nat.split*)

lemma *fps-of-poly-add*: *fps-of-poly* (*p* + *q*) = *fps-of-poly p* + *fps-of-poly q*
by (*simp add*: *fps-of-poly-def plus-poly.rep-eq fps-plus-def*)

lemma *fps-of-poly-diff*: *fps-of-poly* (*p* − *q*) = *fps-of-poly p* − *fps-of-poly q*
by (*simp add*: *fps-of-poly-def minus-poly.rep-eq fps-minus-def*)

lemma *fps-of-poly-uminus*: *fps-of-poly* (−*p*) = −*fps-of-poly p*
by (*simp add*: *fps-of-poly-def uminus-poly.rep-eq fps-uminus-def*)

lemma *fps-of-poly-mult*: *fps-of-poly* (*p* ∗ *q*) = *fps-of-poly p* ∗ *fps-of-poly q*
by (*simp add*: *fps-of-poly-def fps-times-def fps-eq-iff coeff-mult atLeast0AtMost*)

lemma *fps-of-poly-smult*:
fps-of-poly (*smult c p*) = *fps-const c* ∗ *fps-of-poly p*
using *fps-of-poly-mult*[*of* [:*c*:] *p*] **by** (*simp add*: *fps-of-poly-mult fps-of-poly-const*)

lemma *fps-of-poly-sum*: *fps-of-poly* (*sum f A*) = *sum* (λ*x*. *fps-of-poly* (*f x*)) *A*
by (*cases finite A*, *induction rule*: *finite-induct*) (*simp-all add*: *fps-of-poly-add*)

lemma *fps-of-poly-sum-list*: *fps-of-poly* (*sum-list xs*) = *sum-list* (*map fps-of-poly xs*)
by (*induction xs*) (*simp-all add*: *fps-of-poly-add*)

lemma *fps-of-poly-prod*: *fps-of-poly* (*prod f A*) = *prod* (λ*x*. *fps-of-poly* (*f x*)) *A*
by (*cases finite A*, *induction rule*: *finite-induct*) (*simp-all add*: *fps-of-poly-mult*)

lemma *fps-of-poly-prod-list*: *fps-of-poly* (*prod-list xs*) = *prod-list* (*map fps-of-poly xs*)
by (*induction xs*) (*simp-all add*: *fps-of-poly-mult*)

lemma *fps-of-poly-pCons*:
fps-of-poly (*pCons* (*c* :: *'a* :: *semiring-1*) *p*) = *fps-const c* + *fps-of-poly p* ∗ *fps-X*
by (*subst fps-mult-fps-X-commute* [*symmetric*], *intro fps-ext*)
(*auto simp*: *fps-of-poly-def coeff-pCons split*: *nat.split*)

lemma *fps-of-poly-pderiv*: *fps-of-poly* (*pderiv p*) = *fps-deriv* (*fps-of-poly p*)
by (*intro fps-ext*) (*simp add*: *fps-of-poly-nth coeff-pderiv*)

lemma *fps-of-poly-power*: *fps-of-poly* (*p* ^ *n*) = *fps-of-poly p* ^ *n*
by (*induction n*) (*simp-all add*: *fps-of-poly-mult*)

lemma *fps-of-poly-monom*: *fps-of-poly* (*monom* (*c* :: *'a* :: *comm-ring-1*) *n*) = *fps-const c* ∗ *fps-X* ^ *n*
by (*intro fps-ext*) *simp-all*

lemma *fps-of-poly-monom'*: *fps-of-poly* (*monom* (*1* :: *'a* :: *comm-ring-1*) *n*) = *fps-X* ^ *n*
by (*simp add*: *fps-of-poly-monom*)

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 (*cases q* = *0*)
case *False*
from *False fps-of-poly-eq-iff*[*of q 0*] **have** *nz*: *fps-of-poly q* ≠ *0* **by** *simp*
from *assms* **have** *p* = (*p div q*) ∗ *q* **by** *simp*
also have *fps-of-poly* ... = *fps-of-poly* (*p div q*) ∗ *fps-of-poly q*
by (*simp add*: *fps-of-poly-mult*)
also from *nz* **have** ... / *fps-of-poly q* = *fps-of-poly* (*p div q*)
by (*intro nonzero-mult-div-cancel-right*) (*auto simp*: *fps-of-poly-0*)
finally show *?thesis* **..**
qed *simp*

lemma *fps-of-poly-divide-numeral*:
fps-of-poly (*smult* (*inverse* (*numeral c* :: *'a* :: *field*)) *p*) = *fps-of-poly p* / *numeral c*
proof −
have *smult* (*inverse* (*numeral c*)) *p* = [:*inverse* (*numeral c*):] ∗ *p* **by** *simp*
also have *fps-of-poly* ... = *fps-of-poly p* / *numeral c*
by (*subst fps-of-poly-mult*) (*simp add*: *numeral-fps-const fps-of-poly-pCons*)
finally show *?thesis* **by** *simp*
qed

lemma *subdegree-fps-of-poly*:
assumes *p* ≠ *0*
defines *n* ≡ *Polynomial.order 0 p*
shows *subdegree* (*fps-of-poly p*) = *n*
proof (*rule subdegreeI*)

```
  from assms have monom 1 n dvd p by (simp add: monom-1-dvd-iff)
  thus zero: fps-of-poly p $ i = 0 if i < n for i
    using that by (simp add: monom-1-dvd-iff')

  from assms have ¬monom 1 (Suc n) dvd p
    by (auto simp: monom-1-dvd-iff simp del: power-Suc)
  then obtain k where k: k ≤ n fps-of-poly p $ k ≠ 0
    by (auto simp: monom-1-dvd-iff' less-Suc-eq-le)
  with zero[of k] have k = n by linarith
  with k show fps-of-poly p $ n ≠ 0 by simp
qed

lemma fps-of-poly-dvd:
  assumes p dvd q
  shows   fps-of-poly (p :: 'a :: field poly) dvd fps-of-poly q
proof (cases p = 0 ∨ q = 0)
  case False
  with assms fps-of-poly-eq-iff[of p 0] fps-of-poly-eq-iff[of q 0] show ?thesis
    by (auto simp: fps-dvd-iff subdegree-fps-of-poly dvd-imp-order-le)
qed (insert assms, auto)


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)
  using assms by (induction p rule: pCons-induct)
                 (auto simp: pcompose-pCons fps-of-poly-simps fps-of-poly-pCons
                             fps-compose-add-distrib fps-compose-mult-distrib)

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-X ≠ 0* that would otherwise not be solved automatically.

ML ‹

```
(* TODO: Support for division *)
```

```
signature POLY-FPS = sig

val reify-conv : conv
val eq-conv : conv
val eq-simproc : cterm -> thm option

end


structure Poly-Fps = struct

fun const-binop-conv s conv ct =
  case Thm.term-of ct of
    (Const (s', -) $ - $ -) =>
      if s = s' then
        Conv.binop-conv conv ct
      else
        raise CTERM (const-binop-conv, [ct])
  | - => raise CTERM (const-binop-conv, [ct])

fun reify-conv ct =
  let
    val rewr = Conv.rewrs-conv o map (fn thm => thm RS @{thm eq-reflection})
    val un = Conv.arg-conv reify-conv
    val bin = Conv.binop-conv reify-conv
  in
    case Thm.term-of ct of
      (Const (const-name ‹fps-of-poly›, -) $ -) => ct |> Conv.all-conv
    | (Const (const-name ‹Groups.plus›, -) $ - $ -) => ct |> (
        bin then-conv rewr @{thms fps-of-poly-add [symmetric]})
    | (Const (const-name ‹Groups.uminus›, -) $ -) => ct |> (
        un then-conv rewr @{thms fps-of-poly-uminus [symmetric]})
    | (Const (const-name ‹Groups.minus›, -) $ - $ -) => ct |> (
        bin then-conv rewr @{thms fps-of-poly-diff [symmetric]})
    | (Const (const-name ‹Groups.times›, -) $ - $ -) => ct |> (
        bin then-conv rewr @{thms fps-of-poly-mult [symmetric]})
   | (Const (const-name ‹Rings.divide›, -) $ - $ (Const (const-name ‹Num.numeral›,
-) $ -))
        => ct |> (Conv.fun-conv (Conv.arg-conv reify-conv)
             then-conv rewr @{thms fps-of-poly-divide-numeral [symmetric]})
    | (Const (const-name ‹Power.power›, -) $ Const (const-name ‹fps-X›,-) $ -)
=> ct |> (
        rewr @{thms fps-of-poly-monom' [symmetric]})
    | (Const (const-name ‹Power.power›, -) $ - $ -) => ct |> (
        Conv.fun-conv (Conv.arg-conv reify-conv)
        then-conv rewr @{thms fps-of-poly-power [symmetric]})
    | - => ct |> (
        rewr @{thms reify-fps-atom [symmetric]})
  end
```

```
fun eq-conv ct =
  case Thm.term-of ct of
    (Const (const-name ‹HOL.eq›, -) $ - $ -) => ct |> (
      Conv.binop-conv reify-conv
      then-conv Conv.rewr-conv @{thm fps-of-poly-eq-iff[THEN eq-reflection]})
  | - => raise CTERM (poly-fps-eq-conv, [ct])

val eq-simproc = try eq-conv

end
›

simproc-setup poly-fps-eq ((f :: 'a fps) = g) = ‹K (K Poly-Fps.eq-simproc)›

lemma fps-of-poly-linear: fps-of-poly [:a,1 :: 'a :: field:] = fps-X + fps-const a
  by simp

lemma fps-of-poly-linear': fps-of-poly [:1,a :: 'a :: field:] = 1 + fps-const a * fps-X
  by simp

lemma fps-of-poly-cutoff [simp]:
  fps-of-poly (poly-cutoff n p) = fps-cutoff n (fps-of-poly p)
  by (simp add: fps-eq-iff coeff-poly-cutoff)

lemma fps-of-poly-shift [simp]: fps-of-poly (poly-shift n p) = fps-shift n (fps-of-poly
p)
  by (simp add: fps-eq-iff coeff-poly-shift)


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
  unfolding poly-subdegree-def using nth-less-subdegree-zero[of k fps-of-poly p] by
simp


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
by (*induction xs arbitrary*: *acc*) (*simp-all add*: *prefix-length-def*)

lemma *prefix-length-code* [*code*]: *prefix-length P xs* = *prefix-length-aux P 0 xs*
by (*simp add*: *prefix-length-aux-correct*)

lemma *prefix-length-le-length*: *prefix-length P xs* ≤ *length xs*
by (*induction xs*) (*simp-all add*: *prefix-length-def*)

lemma *prefix-length-less-length*: (∃ *x*∈*set xs*. ¬*P x*) ⟹ *prefix-length P xs* < *length xs*
by (*induction xs*) (*simp-all add*: *prefix-length-def*)

lemma *nth-prefix-length*:
(∃ *x*∈*set xs*. ¬*P x*) ⟹ ¬*P* (*xs* ! *prefix-length P xs*)
by (*induction xs*) (*simp-all add*: *prefix-length-def*)

lemma *nth-less-prefix-length*:
n < *prefix-length P xs* ⟹ *P* (*xs* ! *n*)
by (*induction xs arbitrary*: *n*)
(*auto simp*: *prefix-length-def nth-Cons split*: *if-splits nat.splits*)

lemma *poly-subdegree-code* [*code*]: *poly-subdegree p* = *prefix-length* ((=) *0*) (*coeffs p*)
proof (*cases p* = *0*)
case *False*
note [*simp*] = *this*
define *n* **where** *n* = *prefix-length* ((=) *0*) (*coeffs p*)
from *False* **have** ∃ *k*. *coeff p k* ≠ *0* **by** (*auto simp*: *poly-eq-iff*)
hence *ex*: ∃ *x*∈*set* (*coeffs p*). *x* ≠ *0* **by** (*auto simp*: *coeffs-def*)
hence *n-less*: *n* < *length* (*coeffs p*) **and** *nonzero*: *coeffs p* ! *n* ≠ *0*
unfolding *n-def* **by** (*auto intro*!: *prefix-length-less-length nth-prefix-length*)
show *?thesis* **unfolding** *poly-subdegree-def*
proof (*intro subdegreeI*)
from *n-less* **have** *fps-of-poly p* $ *n* = *coeffs p* ! *n*
by (*subst coeffs-nth*) (*simp-all add*: *degree-eq-length-coeffs*)
with *nonzero* **show** *fps-of-poly p* $ *prefix-length* ((=) *0*) (*coeffs p*) ≠ *0*
unfolding *n-def* **by** *simp*
next
fix *k* **assume** *A*: *k* < *prefix-length* ((=) *0*) (*coeffs p*)
also have ... ≤ *length* (*coeffs p*) **by** (*rule prefix-length-le-length*)
finally show *fps-of-poly p* $ *k* = *0*
using *nth-less-prefix-length*[*OF A*]
by (*simp add*: *coeffs-nth degree-eq-length-coeffs*)
qed
qed (*simp-all add*: *poly-subdegree-def prefix-length-def*)

end

Truncation of formal power series: all monomials cx^k with $k \geq n$ are removed; the remainder is a polynomial of degree at most $n - 1$.

lift-definition *truncate-fps* :: *nat* ⇒ *'a fps* ⇒ *'a* :: *zero poly* **is**
  λn F k. if k ≥ n then 0 else fps-nth F k
proof *goal-cases*
  case (*1 n F*)
  have *eventually* (*λk. (if n ≤ k then 0 else fps-nth F k) = 0*) *at-top*
    using *eventually-ge-at-top*[*of n*] **by** *eventually-elim auto*
  thus *?case*
    by (*simp add*: *cofinite-eq-sequentially*)
qed

lemma *coeff-truncate-fps'* [*simp*]:
  k ≥ n ⟹ coeff (*truncate-fps n F*) *k = 0*
  k < n ⟹ coeff (*truncate-fps n F*) *k = fps-nth F k*
  by (*transfer*; *simp*; *fail*)+

lemma *coeff-truncate-fps*: *coeff* (*truncate-fps n F*) *k* = (*if k < n then fps-nth F k else 0*)
  by *auto*

lemma *truncate-0-fps* [*simp*]: *truncate-fps 0 F = 0*
  by (*rule poly-eqI*) *auto*

lemma *degree-truncate-fps*: *n > 0 ⟹ degree* (*truncate-fps n F*) *< n*
  by (*rule degree-lessI*) *auto*

lemma *truncate-fps-0* [*simp*]: *truncate-fps n 0 = 0*
  by (*rule poly-eqI*) (*auto simp*: *coeff-truncate-fps*)

lemma *truncate-fps-add*: *truncate-fps n* (*f + g*) = *truncate-fps n f + truncate-fps n g*
  by (*rule poly-eqI*) (*auto simp*: *coeff-truncate-fps*)

lemma *truncate-fps-diff*: *truncate-fps n* (*f − g*) = *truncate-fps n f − truncate-fps n g*
  by (*rule poly-eqI*) (*auto simp*: *coeff-truncate-fps*)

lemma *truncate-fps-uminus*: *truncate-fps n* (*−f*) = *−truncate-fps n f*
  by (*rule poly-eqI*) (*auto simp*: *coeff-truncate-fps*)

lemma *fps-of-poly-truncate* [*simp*]: *fps-of-poly* (*truncate-fps n f*) = *fps-cutoff n f*
  by (*rule fps-ext*) *auto*

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* = $\{f::int \Rightarrow 'a::zero.\ \forall_\infty\ n::nat.\ f\ (-\ int\ n) = 0\}$
  morphisms *fls-nth Abs-fls*
proof
  show $(\lambda x.\ 0) \in \{f::int \Rightarrow 'a::zero.\ \forall_\infty\ n::nat.\ f\ (-\ int\ n) = 0\}$
    by *simp*
qed

setup-lifting *type-definition-fls*

unbundle *fps-syntax*
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*: $f = g \longleftrightarrow (\forall\, n.\ f\ \$\$\ n = g\ \$\$\ n)$
  by (*simp add*: *fls-nth-inject*[*symmetric*] *fun-eq-iff*)

lemma *nth-Abs-fls* [*simp*]: $\forall_\infty n.\ f\ (-\ int\ n) = 0 \Longrightarrow$ *Abs-fls f* $\$\$\ n = f\ n$
 by (*simp add*: *Abs-fls-inverse*[*OF CollectI*])

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 $\exists\, N.\ \forall\, n{<}N.\ f\ n = 0$
  shows *Abs-fls f* $\$\$\ n = f\ n$
proof (*intro nth-Abs-fls-ex-nat-lower-bound*)
  from *assms* **obtain** *N::int* **where** $\forall\, n{<}N.\ f\ n = 0$ **by** *fast*
  hence $\forall\, n > (if\ N < 0\ then\ nat\ (-N)\ else\ 0).\ f\ (-int\ n) = 0$ **by** *auto*
  thus $\exists\, M.\ \forall\, n{>}M.\ f\ (-\ int\ n) = 0$ **by** *fast*
qed

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 $\forall n{>}N.\ f\$\$(-int\ n) = 0$
using *iffD1*[*OF MOST-nat MOST-fls-neg-nth-eq-0*]
by *blast*

lemma *fls-nth-vanishes-belowE*:
fixes $f :: 'a::zero\ fls$
obtains $N :: int$
where $\forall n{<}N.\ f\$\$n = 0$
proof−
obtain $K :: nat$ **where** K: $\forall n{>}K.\ f\$\$(-int\ n) = 0$ **by** (*elim fls-nth-vanishes-below-natE*)
have $\forall n < -int\ K.\ f\$\$n = 0$
proof *clarify*
fix n **assume** n: $n < -int\ K$
define m **where** $m \equiv nat\ (-n)$
with n **have** $m > K$ **by** *simp*
moreover from n *m-def* **have** $f\$\$n = f\ \$\$\ (-int\ m)$ **by** *simp*
ultimately show $f\ \$\$\ n = 0$ **using** K **by** *simp*
qed
thus $(\bigwedge N.\ \forall n{<}N.\ f\ \$\$\ n = 0 \Longrightarrow thesis) \Longrightarrow thesis$ **by** *fast*
qed

7.1.2 Definition of basic Laurent series

instantiation *fls* :: (*zero*) *zero*
begin
lift-definition *zero-fls* :: $'a\ fls$ **is** λ-. *0* **by** *simp*
instance ..
end

lemma *fls-zero-nth* [*simp*]: $0\ \$\$\ n = 0$
by (*simp add*: *zero-fls-def*)

lemma *fls-zero-eqI*: $(\bigwedge n.\ f\$\$n = 0) \Longrightarrow f = 0$
by (*fastforce intro*: *fls-eqI*)

lemma *fls-nonzeroI*: $f\$\$n \neq 0 \Longrightarrow f \neq 0$
by *auto*

lemma *fls-nonzero-nth*: $f \neq 0 \longleftrightarrow (\exists\ n.\ f\ \$\$\ n \neq 0)$
using *fls-zero-eqI* **by** *fastforce*

lemma *fls-trivial-delta-eq-zero* [*simp*]: $b = 0 \Longrightarrow Abs\text{-}fls\ (\lambda n.\ if\ n{=}a\ then\ b\ else\ 0) = 0$
by (*intro fls-zero-eqI*) *simp*

lemma *fls-delta-nth* [*simp*]:

Abs-fls (λ*n*. *if n=a then b else 0*) $$ *n* = (*if n=a then b else 0*)
using *nth-Abs-fls-lower-bound*[*of a* λ*n*. *if n=a then b else 0*] **by** *simp*

instantiation *fls* :: ({*zero*,*one*}) *one*
begin
lift-definition *one-fls* :: *'a fls* **is** λ*k*. *if k = 0 then 1 else 0*
by (*simp add*: *eventually-cofinite*)
instance ..
end

lemma *fls-one-nth* [*simp*]:
1 $$ *n* = (*if n = 0 then 1 else 0*)
by (*simp add*: *one-fls-def eventually-cofinite*)

instance *fls* :: (*zero-neq-one*) *zero-neq-one*
proof (*standard*, *standard*)
assume (*0*::*'a fls*) = (*1*::*'a fls*)
hence (*0*::*'a fls*) $$ *0* = (*1*::*'a fls*) $$ *0* **by** *simp*
thus *False* **by** *simp*
qed

definition *fls-const* :: *'a*::*zero* ⇒ *'a fls*
where *fls-const c* ≡ *Abs-fls* (λ*n*. *if n = 0 then c else 0*)

lemma *fls-const-nth* [*simp*]: *fls-const c* $$ *n* = (*if n = 0 then c else 0*)
by (*simp add*: *fls-const-def eventually-cofinite*)

lemma *fls-const-0* [*simp*]: *fls-const 0* = *0*
unfolding *fls-const-def* **using** *fls-trivial-delta-eq-zero* **by** *fast*

lemma *fls-const-nonzero*: *c* ≠ *0* ⟹ *fls-const c* ≠ *0*
using *fls-nonzeroI*[*of fls-const c 0*] **by** *simp*

lemma *fls-const-eq-0-iff* [*simp*]: *fls-const c* = *0* ⟷ *c* = *0*
by (*auto simp*: *fls-eq-iff*)

lemma *fls-const-1* [*simp*]: *fls-const 1* = *1*
unfolding *fls-const-def one-fls-def* **..**

lemma *fls-const-eq-1-iff* [*simp*]: *fls-const c* = *1* ⟷ *c* = *1*
by (*auto simp*: *fls-eq-iff*)

lift-definition *fls-X* :: *'a*::{*zero*,*one*} *fls*
is λ*n*. *if n = 1 then 1 else 0*
by *simp*

lemma *fls-X-nth* [*simp*]:
fls-X $$ *n* = (*if n = 1 then 1 else 0*)
by (*simp add*: *fls-X-def*)

lemma *fls-X-nonzero* [*simp*]: (*fls-X* :: *'a* :: *zero-neq-one fls*) ≠ *0*
by (*intro fls-nonzeroI*) *simp*

lift-definition *fls-X-inv* :: *'a*::{*zero*,*one*} *fls*
is λ*n*. *if n* = −*1 then 1 else 0*
by (*simp add*: *eventually-cofinite*)

lemma *fls-X-inv-nth* [*simp*]:
fls-X-inv $$ *n* = (*if n* = −*1 then 1 else 0*)
by (*simp add*: *fls-X-inv-def eventually-cofinite*)

lemma *fls-X-inv-nonzero* [*simp*]: (*fls-X-inv* :: *'a* :: *zero-neq-one fls*) ≠ *0*
by (*intro fls-nonzeroI*) *simp*

7.2 Subdegrees

lemma *unique-fls-subdegree*:
assumes *f* ≠ *0*
shows ∃!*n*. *f*$$*n* ≠ *0* ∧ (∀ *m*. *f*$$*m* ≠ *0* ⟶ *n* ≤ *m*)
proof−
obtain *N*::*nat* **where** *N*: ∀ *n*>*N*. *f*$$(−*int n*) = *0* **by** (*elim fls-nth-vanishes-below-natE*)
define *M* **where** *M* ≡ −*int N*
have *M*: ⋀*m*. *f*$$*m* ≠ *0* ⟹ *M* ≤ *m*
proof−
fix *m* **assume** *m*: *f*$$*m* ≠ *0*
show *M* ≤ *m*
proof (*cases m*<*0*)
case *True* **with** *m N M-def* **show** *?thesis*
using *allE*[*OF N*, *of nat* (−*m*) *False*] **by** *force*
qed (*simp add*: *M-def*)
qed
have ¬ (∀ *k*::*nat*. *f*$$(*M* + *int k*) = *0*)
proof
assume *above0*: ∀ *k*::*nat*. *f*$$(*M* + *int k*) = *0*
have *f*=*0*
proof (*rule fls-zero-eqI*)
fix *n* **show** *f*$$*n* = *0*
proof (*cases M* ≤ *n*)
case *True*
define *k* **where** *k* = *nat* (*n* − *M*)
from *True* **have** *n* = *M* + *int k* **by** (*simp add*: *k-def*)
with *above0* **show** *?thesis* **by** *simp*
next
case *False* **with** *M* **show** *?thesis* **by** *auto*
qed
qed
with *assms* **show** *False* **by** *fast*
qed

hence *ex-k*: $\exists\, k{::}nat.\ f\$\$(M + int\ k) \neq 0$ **by** *fast*
define k **where** $k \equiv (LEAST\ k{::}nat.\ f\$\$(M + int\ k) \neq 0)$
define n **where** $n \equiv M + int\ k$
from *k-def n-def* **have** *fn*: $f\$\$n \neq 0$ **using** *LeastI-ex*[*OF ex-k*] **by** *simp*
moreover have $\forall\, m.\ f\$\$m \neq 0 \longrightarrow n \leq m$
proof (*clarify*)
fix m **assume** m: $f\$\$m \neq 0$
with M **have** $M \leq m$ **by** *fast*
define l **where** $l = nat\ (m - M)$
from $\langle M \leq m \rangle$ **have** l: $m = M + int\ l$ **by** (*simp add*: *l-def*)
with *n-def m k-def l* **show** $n \leq m$
using *Least-le*[*of* $\lambda k.\ f\$\$(M + int\ k) \neq 0\ l$] **by** *auto*
qed
moreover have $\bigwedge n'.\ f\$\$n' \neq 0 \Longrightarrow (\forall\, m.\ f\$\$m \neq 0 \longrightarrow n' \leq m) \Longrightarrow n' = n$
proof−
fix n' :: *int*
assume n': $f\$\$n' \neq 0\ \forall\, m.\ f\$\$m \neq 0 \longrightarrow n' \leq m$
from $n'(1)\ M$ **have** $M \leq n'$ **by** *fast*
define l **where** $l = nat\ (n' - M)$
from $\langle M \leq n' \rangle$ **have** l: $n' = M + int\ l$ **by** (*simp add*: *l-def*)
with *n-def k-def n' fn* **show** $n' = n$
using *Least-le*[*of* $\lambda k.\ f\$\$(M + int\ k) \neq 0\ l$] **by** *force*
qed
ultimately show *?thesis*
using *ex1I*[*of* $\lambda n.\ f\$\$n \neq 0 \wedge (\forall\, m.\ f\$\$m \neq 0 \longrightarrow n \leq m)\ n$] **by** *blast*
qed

definition *fls-subdegree* :: $('a{::}zero)\ fls \Rightarrow int$
where $fls\text{-}subdegree\ f \equiv (if\ f = 0\ then\ 0\ else\ LEAST\ n{::}int.\ f\$\$n \neq 0)$

lemma *fls-zero-subdegree* [*simp*]: $fls\text{-}subdegree\ 0 = 0$
by (*simp add*: *fls-subdegree-def*)

lemma *nth-fls-subdegree-nonzero* [*simp*]: $f \neq 0 \Longrightarrow f\ \$\$\ fls\text{-}subdegree\ f \neq 0$
using *Least1I*[*OF unique-fls-subdegree*] **by** (*simp add*: *fls-subdegree-def*)

lemma *nth-fls-subdegree-zero-iff*: $(f\ \$\$\ fls\text{-}subdegree\ f = 0) \longleftrightarrow (f = 0)$
using *nth-fls-subdegree-nonzero* **by** *auto*

lemma *fls-subdegree-leI*: $f\ \$\$\ n \neq 0 \Longrightarrow fls\text{-}subdegree\ f \leq n$
using *Least1-le*[*OF unique-fls-subdegree*]
by (*auto simp*: *fls-subdegree-def*)

lemma *fls-subdegree-leI'*: $f\ \$\$\ n \neq 0 \Longrightarrow n \leq m \Longrightarrow fls\text{-}subdegree\ f \leq m$
using *fls-subdegree-leI* **by** *fastforce*

lemma *fls-eq0-below-subdegree* [*simp*]: $n < fls\text{-}subdegree\ f \Longrightarrow f\ \$\$\ n = 0$
using *fls-subdegree-leI* **by** *fastforce*

lemma *fls-subdegree-geI*: $f \neq 0 \Longrightarrow (\bigwedge k.\ k < n \Longrightarrow f\ \$\$\ k = 0) \Longrightarrow n \leq$ *fls-subdegree f*
using *nth-fls-subdegree-nonzero* **by** *force*

lemma *fls-subdegree-ge0I*: $(\bigwedge k.\ k < 0 \Longrightarrow f\ \$\$\ k = 0) \Longrightarrow 0 \leq$ *fls-subdegree f*
using *fls-subdegree-geI*[*of f 0*] **by** (*cases f=0*) *auto*

lemma *fls-subdegree-greaterI*:
assumes $f \neq 0$ $\bigwedge k.\ k \leq n \Longrightarrow f\ \$\$\ k = 0$
shows $n <$ *fls-subdegree f*
using *assms*(*1*) *assms*(*2*)[*of fls-subdegree f*] *nth-fls-subdegree-nonzero*[*of f*]
by *force*

lemma *fls-subdegree-eqI*: $f\ \$\$\ n \neq 0 \Longrightarrow (\bigwedge k.\ k < n \Longrightarrow f\ \$\$\ k = 0) \Longrightarrow$ *fls-subdegree f* $= n$
using *fls-subdegree-leI fls-subdegree-geI*[*of f*]
by *fastforce*

lemma *fls-delta-subdegree* [*simp*]:
$b \neq 0 \Longrightarrow$ *fls-subdegree* (*Abs-fls* ($\lambda n.$ *if n=a then b else 0*)) $= a$
by (*intro fls-subdegree-eqI*) *simp-all*

lemma *fls-delta0-subdegree*: *fls-subdegree* (*Abs-fls* ($\lambda n.$ *if n=0 then a else 0*)) $= 0$
by (*cases a=0*) *simp-all*

lemma *fls-one-subdegree* [*simp*]: *fls-subdegree 1* $= 0$
by (*auto intro*: *fls-delta0-subdegree simp*: *one-fls-def*)

lemma *fls-const-subdegree* [*simp*]: *fls-subdegree* (*fls-const c*) $= 0$
by (*cases c=0*) (*auto intro*: *fls-subdegree-eqI*)

lemma *fls-X-subdegree* [*simp*]: *fls-subdegree* (*fls-X*::*'a*::{*zero-neq-one*} *fls*) $= 1$
by (*intro fls-subdegree-eqI*) *simp-all*

lemma *fls-X-inv-subdegree* [*simp*]: *fls-subdegree* (*fls-X-inv*::*'a*::{*zero-neq-one*} *fls*) $= -1$
by (*intro fls-subdegree-eqI*) *simp-all*

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 (*rule fls-eqI*)
fix *n* **from** *assms* **show** $f\ \$\$\ n = g\ \$\$\ n$ **by** (*cases* $n < N$) *auto*
qed

7.3 Shifting

7.3.1 Shift definition

definition *fls-shift* :: *int* $\Rightarrow$ (*'a*::*zero*) *fls* $\Rightarrow$ *'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*)
unfolding *fls-shift-def*
proof (*rule nth-Abs-fls-ex-lower-bound*)
obtain *K*::*int* **where** *K*: ∀ *n*<*K*. *f*$$*n* = *0* **by** (*elim fls-nth-vanishes-belowE*)
hence ∀ *n*<*K*−*m*. *f*$$(*n*+*m*) = *0* **by** *auto*
thus ∃ *N*. ∀ *n*<*N*. *f* $$ (*n* + *m*) = *0* **by** *fast*
qed

lemma *fls-shift-eq-iff*: (*fls-shift m f* = *fls-shift m g*) ⟷ (*f* = *g*)
proof (*rule iffI*, *rule fls-eqI*)
fix *k*
assume *1*: *fls-shift m f* = *fls-shift m g*
have *f* $$ *k* = *fls-shift m g* $$ (*k* − *m*) **by** (*simp add*: *1*[*symmetric*])
thus *f* $$ *k* = *g* $$ *k* **by** *simp*
qed (*intro fls-eqI*, *simp*)

lemma *fls-shift-0* [*simp*]: *fls-shift 0 f* = *f*
by (*intro fls-eqI*) *simp*

lemma *fls-shift-subdegree* [*simp*]:
f ≠ *0* ⟹ *fls-subdegree* (*fls-shift n f*) = *fls-subdegree f* − *n*
by (*intro fls-subdegree-eqI*) *simp-all*

lemma *fls-shift-fls-shift* [*simp*]: *fls-shift m* (*fls-shift k f*) = *fls-shift* (*k*+*m*) *f*
by (*intro fls-eqI*) (*simp add*: *algebra-simps*)

lemma *fls-shift-fls-shift-reorder*:
fls-shift m (*fls-shift k f*) = *fls-shift k* (*fls-shift m f*)
using *fls-shift-fls-shift*[*of m k f*] *fls-shift-fls-shift*[*of k m f*] **by** (*simp add*: *add.commute*)

lemma *fls-shift-zero* [*simp*]: *fls-shift m 0* = *0*
by (*intro fls-zero-eqI*) *simp*

lemma *fls-shift-eq0-iff*: *fls-shift m f* = *0* ⟷ *f* = *0*
using *fls-shift-eq-iff*[*of m f 0*] **by** *simp*

lemma *fls-shift-eq-1-iff*: *fls-shift n f* = *1* ⟷ *f* = *fls-shift* (−*n*) *1*
by (*metis add-minus-cancel fls-shift-eq-iff fls-shift-fls-shift*)

lemma *fls-shift-nonneg-subdegree*: *m* ≤ *fls-subdegree f* ⟹ *fls-subdegree* (*fls-shift m f*) ≥ *0*
by (*cases f*=*0*) (*auto intro*: *fls-subdegree-geI*)

lemma *fls-shift-delta*:
fls-shift m (*Abs-fls* (λ*n*. *if n*=*a then b else 0*)) = *Abs-fls* (λ*n*. *if n*=*a*−*m then b*

else 0)
by (*intro fls-eqI*) *simp*

lemma *fls-shift-const*:
fls-shift m (*fls-const c*) = *Abs-fls* (λ*n*. *if n*=−*m then c else 0*)
by (*intro fls-eqI*) *simp*

lemma *fls-shift-const-nth*:
fls-shift m (*fls-const c*) $$ *n* = (*if n*=−*m then c else 0*)
by (*simp add*: *fls-shift-const*)

lemma *fls-X-conv-shift-1*: *fls-X* = *fls-shift* (−*1*) *1*
by (*intro fls-eqI*) *simp*

lemma *fls-X-shift-to-one* [*simp*]: *fls-shift 1 fls-X* = *1*
using *fls-shift-fls-shift*[*of* −*1 1 1*] **by** (*simp add*: *fls-X-conv-shift-1*)

lemma *fls-X-inv-conv-shift-1*: *fls-X-inv* = *fls-shift 1 1*
by (*intro fls-eqI*) *simp*

lemma *fls-X-inv-shift-to-one* [*simp*]: *fls-shift* (−*1*) *fls-X-inv* = *1*
using *fls-shift-fls-shift*[*of 1* −*1 1*] **by** (*simp add*: *fls-X-inv-conv-shift-1*)

lemma *fls-X-fls-X-inv-conv*:
fls-X = *fls-shift* (−*2*) *fls-X-inv fls-X-inv* = *fls-shift 2 fls-X*
by (*simp-all add*: *fls-X-conv-shift-1 fls-X-inv-conv-shift-1*)

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*)
by *simp*

lemma *fls-base-factor-nonzero* [*simp*]: *f* ≠ *0* ⟹ *fls-base-factor f* ≠ *0*
using *fls-nonzeroI*[*of fls-base-factor f 0*] **by** *simp*

lemma *fls-base-factor-subdegree* [*simp*]: *fls-subdegree* (*fls-base-factor f*) = *0*
by (*cases f*=*0*) *auto*

lemma *fls-base-factor-base* [*simp*]:
fls-base-factor f $$ *fls-subdegree* (*fls-base-factor f*) = *f* $$ *fls-subdegree f*

using *fls-base-factor-subdegree*[*of f*] **by** *simp*

lemma *fls-conv-base-factor-shift-subdegree*:
f = *fls-shift* (−*fls-subdegree f*) (*fls-base-factor f*)
by *simp*

lemma *fls-base-factor-idem*:
fls-base-factor (*fls-base-factor* (*f*::′*a*::*zero fls*)) = *fls-base-factor f*
using *fls-base-factor-subdegree*[*of f*] **by** *simp*

lemma *fls-base-factor-zero*: *fls-base-factor* (*0*::′*a*::*zero fls*) = *0*
by *simp*

lemma *fls-base-factor-zero-iff*: *fls-base-factor* (*f*::′*a*::*zero fls*) = *0* ⟷ *f* = *0*
proof
have *fls-shift* (−*fls-subdegree f*) (*fls-shift* (*fls-subdegree f*) *f*) = *f* **by** *simp*
thus *fls-base-factor f* = *0* ⟹ *f*=*0* **by** *simp*
qed *simp*

lemma *fls-base-factor-nth-0*: *f* ≠ *0* ⟹ *fls-base-factor f* $$ *0* ≠ *0*
by *simp*

lemma *fls-base-factor-one*: *fls-base-factor* (*1*::′*a*::{*zero*,*one*} *fls*) = *1*
by *simp*

lemma *fls-base-factor-const*: *fls-base-factor* (*fls-const c*) = *fls-const c*
by *simp*

lemma *fls-base-factor-delta*:
fls-base-factor (*Abs-fls* (λ*n*. *if n*=*a then c else 0*)) = *fls-const c*
by (*cases c*=*0*) (*auto intro*: *fls-eqI*)

lemma *fls-base-factor-X*: *fls-base-factor* (*fls-X*::′*a*::{*zero-neq-one*} *fls*) = *1*
by *simp*

lemma *fls-base-factor-X-inv*: *fls-base-factor* (*fls-X-inv*::′*a*::{*zero-neq-one*} *fls*) = *1*
by *simp*

lemma *fls-base-factor-shift* [*simp*]: *fls-base-factor* (*fls-shift n f*) = *fls-base-factor f*
by (*cases f*=*0*) *simp-all*

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 λ*f*. *Abs-fps* (λ*n*. *f* (*int n*))

.

lemma *fls-regpart-nth* [*simp*]: *fls-regpart f $ n = f $$ (int n)*
by (*simp add*: *fls-regpart-def*)

lemma *fls-regpart-zero* [*simp*]: *fls-regpart 0 = 0*
by (*intro fps-ext*) *simp*

lemma *fls-regpart-one* [*simp*]: *fls-regpart 1 = 1*
by (*intro fps-ext*) *simp*

lemma *fls-regpart-Abs-fls*:
$\forall_\infty n$. *F* (− *int n*) = *0* ⟹ *fls-regpart* (*Abs-fls F*) = *Abs-fps* (λ*n*. *F* (*int n*))
by (*intro fps-ext*) *auto*

lemma *fls-regpart-delta*:
fls-regpart (*Abs-fls* (λ*n*. *if n=a then b else 0*)) =
(*if a < 0 then 0 else Abs-fps* (λ*n*. *if n=nat a then b else 0*))
by (*rule fps-ext*, *auto*)

lemma *fls-regpart-const* [*simp*]: *fls-regpart* (*fls-const c*) = *fps-const c*
by (*intro fps-ext*) *simp*

lemma *fls-regpart-fls-X* [*simp*]: *fls-regpart fls-X = fps-X*
by (*intro fps-ext*) *simp*

lemma *fls-regpart-fls-X-inv* [*simp*]: *fls-regpart fls-X-inv = 0*
by (*intro fps-ext*) *simp*

lemma *fls-regpart-eq0-imp-nonpos-subdegree*:
assumes *fls-regpart f = 0*
shows *fls-subdegree f* ≤ *0*
proof (*cases f=0*)
case *False*
have *fls-subdegree f* ≥ *0* ⟹ *f $$ fls-subdegree f = 0*
proof−
assume *fls-subdegree f* ≥ *0*
hence *f $$* (*fls-subdegree f*) = (*fls-regpart f*) $ (*nat* (*fls-subdegree f*)) **by** *simp*
with *assms* **show** *f $$* (*fls-subdegree f*) = *0* **by** *simp*
qed
with *False* **show** *?thesis* **by** *fastforce*
qed *simp*

lemma *fls-subdegree-lt-fls-regpart-subdegree*:
fls-subdegree f ≤ *int* (*subdegree* (*fls-regpart f*))
using *fls-subdegree-leI nth-subdegree-nonzero*[*of fls-regpart f*]
by (*cases* (*fls-regpart f*) = *0*)
(*simp-all add*: *fls-regpart-eq0-imp-nonpos-subdegree*)

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 (*cases f=0*)
case *False* **with** *assms* **show** *?thesis* **by** (*intro subdegreeI*) *simp-all*
qed *simp*

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 (*rule fls-eq-above-subdegreeI*, *rule assms(1)*, *rule assms(2)*, *clarify*)
fix *k*::*int* **assume** *0 ≤ k*
with *assms(3)* **show** *f $$ k = g $$ k*
using *fls-regpart-nth*[*of f nat k*] *fls-regpart-nth*[*of g*] **by** *simp*
qed

lemma *fls-regpart-shift-conv-fps-shift*:
m ≥ 0 ⟹ fls-regpart (fls-shift m f) = fps-shift (nat m) (fls-regpart f)
by (*intro fps-ext*) *simp-all*

lemma *fps-shift-fls-regpart-conv-fls-shift*:
fps-shift m (fls-regpart f) = fls-regpart (fls-shift m f)
by (*intro fps-ext*) *simp-all*

lemma *fps-unit-factor-fls-regpart*:
fls-subdegree f ≥ 0 ⟹ unit-factor (fls-regpart f) = fls-regpart (fls-base-factor f)
by (*auto intro*: *fps-ext simp*: *fls-regpart-subdegree-conv*)

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 *λf. Abs-poly (λn. if n = 0 then 0 else f (− int n))*
.

lemma *fls-prpart-coeff* [*simp*]: *coeff (fls-prpart f) n = (if n = 0 then 0 else f $$ (− int n))*
proof−
have *{x. (if x = 0 then 0 else f $$ − int x) ≠ 0} ⊆ {x. f $$ − int x ≠ 0}*
by *auto*
hence *finite {x. (if x = 0 then 0 else f $$ − int x) ≠ 0}*
using *fls-finite-nonzero-neg-nth*[*of f*] **by** (*simp add*: *rev-finite-subset*)
hence *coeff (fls-prpart f) = (λn. if n = 0 then 0 else f $$ (− int n))*
using *Abs-poly-inverse*[*OF CollectI*, *OF iffD2*, *OF eventually-cofinite*]
by (*simp add*: *fls-prpart-def*)
thus *?thesis* **by** *simp*
qed

lemma *fls-prpart-eq0-iff*: (*fls-prpart f = 0*) ⟷ (*fls-subdegree f ≥ 0*)
proof
  assume *1*: *fls-prpart f = 0*
  show *fls-subdegree f ≥ 0*
  proof (*intro fls-subdegree-ge0I*)
    fix *k*::*int* **assume** *k < 0*
    with *1* **show** *f $$ k = 0* **using** *fls-prpart-coeff*[*of f nat* (−*k*)] **by** *simp*
  qed
qed (*intro poly-eqI*, *simp*)

lemma *fls-prpart0* [*simp*]: *fls-prpart 0 = 0*
  by (*simp add*: *fls-prpart-eq0-iff*)

lemma *fls-prpart-one* [*simp*]: *fls-prpart 1 = 0*
  by (*simp add*: *fls-prpart-eq0-iff*)

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*)
  by (*intro poly-eqI*) (*auto simp*: *nth-default-def nth-append*)

lemma *fls-prpart-const* [*simp*]: *fls-prpart* (*fls-const c*) = *0*
  by (*simp add*: *fls-prpart-eq0-iff*)

lemma *fls-prpart-X* [*simp*]: *fls-prpart fls-X = 0*
  by (*intro poly-eqI*) *simp*

lemma *fls-prpart-X-inv*: *fls-prpart fls-X-inv* = [:*0*,*1*:]
proof (*intro poly-eqI*)
  fix *n* **show** *coeff* (*fls-prpart fls-X-inv*) *n* = *coeff* [:*0*,*1*:] *n*
  proof (*cases n*)
    case (*Suc i*) **thus** *?thesis* **by** (*cases i*) *simp-all*
  qed *simp*
qed

lemma *degree-fls-prpart* [*simp*]:
  degree (*fls-prpart f*) = *nat* (−*fls-subdegree f*)
proof (*cases f=0*)
  case *False* **show** *?thesis* **unfolding** *degree-def*
  proof (*intro Least-equality*)
    fix *N* **assume** *N*: ∀ *i*>*N*. *coeff* (*fls-prpart f*) *i* = *0*
    have ∀ *i* < −*int N*. *f $$ i* = *0*
    proof *clarify*
      fix *i* **assume** *i*: *i* < −*int N*
      hence *nat* (−*i*) > *N* **by** *simp*
      with *N i* **show** *f $$ i* = *0* **using** *fls-prpart-coeff*[*of f nat* (−*i*)] **by** *auto*
    qed
    with *False* **have** *fls-subdegree f* ≥ −*int N* **using** *fls-subdegree-geI* **by** *auto*
    thus *nat* (− *fls-subdegree f*) ≤ *N* **by** *simp*

qed *auto*
qed *simp*

lemma *fls-prpart-shift*:
assumes $m \leq 0$
shows *fls-prpart* (*fls-shift m f*) = *pCons 0* (*poly-shift* (*Suc* (*nat* (−*m*))) (*fls-prpart f*))
proof (*intro poly-eqI*)
fix *n*
define *LHS RHS*
where *LHS* ≡ *fls-prpart* (*fls-shift m f*)
and *RHS* ≡ *pCons 0* (*poly-shift* (*Suc* (*nat* (−*m*))) (*fls-prpart f*))
show *coeff LHS n* = *coeff RHS n*
proof (*cases n*)
case (*Suc k*)
from *assms* **have** *1*: −*int* (*Suc k* + *nat* (−*m*)) = −*int* (*Suc k*) + *m* **by** *simp*
have *coeff RHS n* = *f* $$ (−*int* (*Suc k*) + *m*)
using *arg-cong*[*OF 1*, *of* ($$) *f*] **by** (*simp add*: *Suc RHS-def coeff-poly-shift*)
with *Suc* **show** *?thesis* **by** (*simp add*: *LHS-def*)
qed (*simp add*: *LHS-def RHS-def*)
qed

lemma *fls-prpart-base-factor*: *fls-prpart* (*fls-base-factor f*) = *0*
using *fls-base-factor-subdegree*[*of f*] **by** (*simp add*: *fls-prpart-eq0-iff*)

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*)
by (*simp add*: *assms fls-regpart-shift-conv-fps-shift*)

lemma *fls-base-factor-to-fps-nth*:
fls-base-factor-to-fps f $ *n* = *f* $$ (*fls-subdegree f* + *int n*)
by (*simp add*: *algebra-simps*)

lemma *fls-base-factor-to-fps-base*: *f* ≠ *0* ⟹ *fls-base-factor-to-fps f* $ *0* ≠ *0*
by *simp*

lemma *fls-base-factor-to-fps-nonzero*: *f* ≠ *0* ⟹ *fls-base-factor-to-fps f* ≠ *0*
using *fps-nonzeroI*[*of fls-base-factor-to-fps f 0*] *fls-base-factor-to-fps-base* **by** *simp*

lemma *fls-base-factor-to-fps-subdegree* [*simp*]: *subdegree* (*fls-base-factor-to-fps f*) = *0*
by (*cases f=0*) *auto*

lemma *fls-base-factor-to-fps-trivial*:

fls-subdegree f = 0 ⟹ fls-base-factor-to-fps f = fls-regpart f
by *simp*

lemma *fls-base-factor-to-fps-zero*: *fls-base-factor-to-fps 0 = 0*
by *simp*

lemma *fls-base-factor-to-fps-one*: *fls-base-factor-to-fps 1 = 1*
by *simp*

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
using *fls-base-factor-delta*[*of a c*] **by** *simp*

lemma *fls-base-factor-to-fps-const*:
fls-base-factor-to-fps (fls-const c) = fps-const c
by *simp*

lemma *fls-base-factor-to-fps-X*:
fls-base-factor-to-fps (fls-X::'a::{zero-neq-one} fls) = 1
by *simp*

lemma *fls-base-factor-to-fps-X-inv*:
fls-base-factor-to-fps (fls-X-inv::'a::{zero-neq-one} fls) = 1
by *simp*

lemma *fls-base-factor-to-fps-shift*:
fls-base-factor-to-fps (fls-shift m f) = fls-base-factor-to-fps f
using *fls-base-factor-shift*[*of m f*] **by** *simp*

lemma *fls-base-factor-to-fps-base-factor*:
fls-base-factor-to-fps (fls-base-factor f) = fls-base-factor-to-fps f
using *fls-base-factor-to-fps-shift* **by** *simp*

lemma *fps-unit-factor-fls-base-factor*:
unit-factor (fls-base-factor-to-fps f) = fls-base-factor-to-fps f
using *fls-base-factor-to-fps-subdegree*[*of f*] **by** *simp*

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* (λ*k*::*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*))
using *nth-Abs-fls-lower-bound*[*of 0* (λ*k*::*int*. *if k<0 then 0 else f* $ (*nat k*))]
unfolding *fps-to-fls-def*
by *simp*

lemma *fps-to-fls-eq-imp-fps-eq*:
assumes *fps-to-fls f = fps-to-fls g*
shows *f = g*
proof (*intro fps-ext*)
fix *n*
have *f $ n = fps-to-fls g $$ int n* **by** (*simp add*: *assms*[*symmetric*])
thus *f $ n = g $ n* **by** *simp*
qed

lemma *fps-to-fls-eq-iff* [*simp*]: *fps-to-fls f = fps-to-fls g ⟷ f = g*
using *fps-to-fls-eq-imp-fps-eq* **by** *blast*

lemma *fps-zero-to-fls* [*simp*]: *fps-to-fls 0 = 0*
by (*intro fls-zero-eqI*) *simp*

lemma *fps-to-fls-nonzeroI*: *f ≠ 0 ⟹ fps-to-fls f ≠ 0*
using *fps-to-fls-eq-imp-fps-eq*[*of f 0*] **by** *auto*

lemma *fps-one-to-fls* [*simp*]: *fps-to-fls 1 = 1*
by (*intro fls-eqI*) *simp*

lemma *fps-to-fls-Abs-fps*:
fps-to-fls (*Abs-fps F*) = *Abs-fls* (*λn. if n<0 then 0 else F* (*nat n*))
using *nth-Abs-fls-lower-bound*[*of 0* (*λn::int. if n<0 then 0 else F* (*nat n*))]
by (*intro fls-eqI*) *simp*

lemma *fps-delta-to-fls*:
fps-to-fls (*Abs-fps* (*λn. if n=a then b else 0*)) = *Abs-fls* (*λn. if n=int a then b else 0*)
using *fls-eqI*[*of - Abs-fls* (*λn. if n=int a then b else 0*)] **by** *force*

lemma *fps-const-to-fls* [*simp*]: *fps-to-fls* (*fps-const c*) = *fls-const c*
by (*intro fls-eqI*) *simp*

lemma *fps-X-to-fls* [*simp*]: *fps-to-fls fps-X = fls-X*
by (*fastforce intro*: *fls-eqI*)

lemma *fps-to-fls-eq-0-iff* [*simp*]: (*fps-to-fls f = 0*) ⟷ (*f=0*)
using *fps-to-fls-nonzeroI* **by** *auto*

lemma *fps-to-fls-eq-1-iff* [*simp*]: *fps-to-fls f = 1 ⟷ f = 1*
using *fps-to-fls-eq-iff* **by** *fastforce*

lemma *fls-subdegree-fls-to-fps-gt0*: *fls-subdegree* (*fps-to-fls f*) ≥ *0*
proof (*cases f=0*)
case *False* **show** *?thesis*
proof (*rule fls-subdegree-geI*, *rule fls-nonzeroI*)
from *False* **show** *fps-to-fls f $$ int* (*subdegree f*) ≠ *0*
by *simp*

```
  qed simp
qed simp

lemma fls-subdegree-fls-to-fps: fls-subdegree (fps-to-fls f) = int (subdegree f)
proof (cases f=0)
  case False
  have subdegree f = nat (fls-subdegree (fps-to-fls f))
  proof (rule subdegreeI)
    from False show f $ (nat (fls-subdegree (fps-to-fls f))) ≠ 0
      using fls-subdegree-fls-to-fps-gt0[of f] nth-fls-subdegree-nonzero[of fps-to-fls f]
          fps-to-fls-nonzeroI[of f]
      by    simp
  next
    fix k assume k: k < nat (fls-subdegree (fps-to-fls f))
    thus f $ k = 0
      using fls-eq0-below-subdegree[of int k fps-to-fls f] by simp
  qed
  thus ?thesis by (simp add: fls-subdegree-fls-to-fps-gt0)
qed simp

lemma fps-shift-to-fls [simp]:
  n ≤ subdegree f ⟹ fps-to-fls (fps-shift n f) = fls-shift (int n) (fps-to-fls f)
  by (auto intro: fls-eqI simp: nat-add-distrib nth-less-subdegree-zero)

lemma fls-base-factor-fps-to-fls: fls-base-factor (fps-to-fls f) = fps-to-fls (unit-factor
f)
  using nth-less-subdegree-zero[of - f]
  by    (auto intro: fls-eqI simp: fls-subdegree-fls-to-fps nat-add-distrib)

lemma fls-regpart-to-fls-trivial [simp]:
  fls-subdegree f ≥ 0 ⟹ fps-to-fls (fls-regpart f) = f
  by (intro fls-eqI) simp

lemma fls-regpart-fps-trivial [simp]: fls-regpart (fps-to-fls f) = f
  by (intro fps-ext) simp

lemma fps-to-fls-base-factor-to-fps:
  fps-to-fls (fls-base-factor-to-fps f) = fls-base-factor f
  by (intro fls-eqI) simp

lemma fls-conv-base-factor-to-fps-shift-subdegree:
  f = fls-shift (−fls-subdegree f) (fps-to-fls (fls-base-factor-to-fps f))
  using fps-to-fls-base-factor-to-fps[of f] fps-to-fls-base-factor-to-fps[of f] by simp

lemma fls-base-factor-to-fps-to-fls:
  fls-base-factor-to-fps (fps-to-fls f) = unit-factor f
  using fls-base-factor-fps-to-fls[of f] fls-regpart-fps-trivial[of unit-factor f]
  by    simp
```

lemma *fls-as-fps*:
fixes $f :: 'a :: zero\ fls$ **and** $n :: int$
assumes n: $n \geq -fls\text{-}subdegree\ f$
obtains f' **where** $f = fls\text{-}shift\ n\ (fps\text{-}to\text{-}fls\ f')$
proof −
have $fls\text{-}subdegree\ (fls\text{-}shift\ (-\ n)\ f) \geq 0$
by (*rule fls-shift-nonneg-subdegree*) (*use n* **in** *simp*)
hence $f = fls\text{-}shift\ n\ (fps\text{-}to\text{-}fls\ (fls\text{-}regpart\ (fls\text{-}shift\ (-n)\ f)))$
by (*subst fls-regpart-to-fls-trivial*) *simp-all*
thus *?thesis*
by (*rule that*)
qed

lemma *fls-as-fps'*:
fixes $f :: 'a :: zero\ fls$ **and** $n :: int$
assumes n: $n \geq -fls\text{-}subdegree\ f$
shows $\exists f'.\ f = fls\text{-}shift\ n\ (fps\text{-}to\text{-}fls\ f')$
using *fls-as-fps*[*OF assms*] **by** *metis*

abbreviation
$fls\text{-}regpart\text{-}as\text{-}fls\ f \equiv fps\text{-}to\text{-}fls\ (fls\text{-}regpart\ f)$
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*)
by *simp*

lemma *fls-regpart-idem*:
fls-regpart (*fls-regpart-as-fls f*) = *fls-regpart f*
by *simp*

lemma *fls-prpart-as-fls-nth*:
fls-prpart-as-fls f $$ n = (*if n* < *0 then f $$ n else 0*)
proof (*cases n* < *fls-subdegree f n* < *0 rule*: *case-split*[*case-product case-split*])
case *False-True*
hence *nat* (−*fls-subdegree f*) − *nat* (*n* − *fls-subdegree f*) = *nat* (−*n*) **by** *auto*
with *False-True* **show** *?thesis*
using *coeff-reflect-poly*[*of fls-prpart f nat* (*n* − *fls-subdegree f*)] **by** *auto*
next
case *False-False* **thus** *?thesis*
using *coeff-reflect-poly*[*of fls-prpart f nat* (*n* − *fls-subdegree f*)] **by** *auto*
qed *simp-all*

lemma *fls-prpart-idem* [*simp*]: *fls-prpart* (*fls-prpart-as-fls f*) = *fls-prpart f*
using *fls-prpart-as-fls-nth*[*of f*] **by** (*intro poly-eqI*) *simp*

lemma *fls-regpart-prpart*: *fls-regpart* (*fls-prpart-as-fls f*) = *0*

using *fls-prpart-as-fls-nth*[*of f*] **by** (*intro fps-ext*) *simp*

lemma *fls-prpart-regpart*: *fls-prpart* (*fls-regpart-as-fls f*) = *0*
by (*intro poly-eqI*) *simp*

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−
fix *f f'* :: *int* ⇒ *'a*
assume ∀$_\infty$*n*. *f* (− *int n*) = *0* ∀$_\infty$*n*. *f'* (− *int n*) = *0*
from *this* **obtain** *N N'* **where** ∀ *n*>*N*. *f* (−*int n*) = *0* ∀ *n*>*N'*. *f'* (−*int n*) = *0*
by (*auto simp*: *MOST-nat*)
hence ∀ *n* > *max N N'*. *f* (−*int n*) + *f'* (−*int n*) = *0* **by** *auto*
hence ∃ *K*. ∀ *n*>*K*. *f* (−*int n*) + *f'* (−*int n*) = *0* **by** *fast*
thus ∀$_\infty$*n*. *f* (− *int n*) + *f'* (−*int n*) = *0* **by** (*simp add*: *MOST-nat*)
qed
instance ..
end

lemma *fls-plus-nth* [*simp*]: (*f* + *g*) $$ *n* = *f* $$ *n* + *g* $$ *n*
by *transfer simp*

lemma *fls-plus-const*: *fls-const x* + *fls-const y* = *fls-const* (*x*+*y*)
by (*intro fls-eqI*) *simp*

lemma *fls-plus-subdegree*:
f + *g* ≠ *0* ⟹ *fls-subdegree* (*f* + *g*) ≥ *min* (*fls-subdegree f*) (*fls-subdegree g*)
by (*auto intro*: *fls-subdegree-geI*)

lemma *fls-shift-plus* [*simp*]:
fls-shift m (*f* + *g*) = (*fls-shift m f*) + (*fls-shift m g*)
by (*intro fls-eqI*) *simp*

lemma *fls-regpart-plus* [*simp*]: *fls-regpart* (*f* + *g*) = *fls-regpart f* + *fls-regpart g*
by (*intro fps-ext*) *simp*

lemma *fls-prpart-plus* [*simp*] : *fls-prpart* (*f* + *g*) = *fls-prpart f* + *fls-prpart g*
by (*intro poly-eqI*) *simp*

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 $f = R + P$
using *fls-prpart-as-fls-nth*[*of f*]
by (*auto intro*: *fls-eqI simp add*: *assms*)

lemma *fps-to-fls-plus* [*simp*]: *fps-to-fls* $(f + g)$ = *fps-to-fls* f + *fps-to-fls* g
by (*intro fls-eqI*) *simp*

instance *fls* :: (*monoid-add*) *monoid-add*
proof
fix $a\ b\ c$ :: $'a$ *fls*
show $a + b + c = a + (b + c)$ **by** *transfer* (*simp add*: *add.assoc*)
show $0 + a = a$ **by** *transfer simp*
show $a + 0 = a$ **by** *transfer simp*
qed

instance *fls* :: (*comm-monoid-add*) *comm-monoid-add*
by (*standard*, *transfer*, *auto simp*: *add.commute*)

lemma *fls-nth-sum*: *fls-nth* $(\sum x{\in}A.\ f\ x)\ n = (\sum x{\in}A.$ *fls-nth* $(f\ x)\ n)$
by (*induction A rule*: *infinite-finite-induct*) *auto*

7.5.2 Subtraction and negatives

instantiation *fls* :: (*group-add*) *minus*
begin
lift-definition *minus-fls* :: $'a$ *fls* $\Rightarrow$ $'a$ *fls* $\Rightarrow$ $'a$ *fls* **is** $\lambda f\ g\ n.\ f\ n - g\ n$
proof−
fix $f\ f'$ :: *int* $\Rightarrow$ $'a$
assume $\forall_\infty n.\ f\ (-\ int\ n) = 0\ \forall_\infty n.\ f'\ (-\ int\ n) = 0$
from *this* **obtain** $N\ N'$ **where** $\forall\, n{>}N.\ f\ (-int\ n) = 0\ \forall\, n{>}N'.\ f'\ (-int\ n) = 0$
by (*auto simp*: *MOST-nat*)
hence $\forall\, n > max\ N\ N'.\ f\ (-int\ n) - f'\ (-int\ n) = 0$ **by** *auto*
hence $\exists\, K.\ \forall\, n{>}K.\ f\ (-int\ n) - f'\ (-int\ n) = 0$ **by** *fast*
thus $\forall_\infty n.\ f\ (-\ int\ n) - f'\ (-int\ n) = 0$ **by** (*simp add*: *MOST-nat*)
qed
instance ..
end

lemma *fls-minus-nth* [*simp*]: $(f - g)$ \$\$ $n = f$ \$\$ $n - g$ \$\$ n
by *transfer simp*

lemma *fls-minus-const*: *fls-const* x − *fls-const* y = *fls-const* $(x{-}y)$
by (*intro fls-eqI*) *simp*

lemma *fls-subdegree-minus*:
$f - g \neq 0 \Longrightarrow$ *fls-subdegree* $(f - g) \geq min$ (*fls-subdegree* f) (*fls-subdegree* g)
by (*intro fls-subdegree-geI*) *simp-all*

lemma *fls-shift-minus* [*simp*]: *fls-shift m* (*f* − *g*) = (*fls-shift m f*) − (*fls-shift m g*)
by (*auto intro*: *fls-eqI*)

lemma *fls-regpart-minus* [*simp*]: *fls-regpart* (*f* − *g*) = *fls-regpart f* − *fls-regpart g*
by (*intro fps-ext*) *simp*

lemma *fls-prpart-minus* [*simp*] : *fls-prpart* (*f* − *g*) = *fls-prpart f* − *fls-prpart g*
by (*intro poly-eqI*) *simp*

lemma *fps-to-fls-minus* [*simp*]: *fps-to-fls* (*f* − *g*) = *fps-to-fls f* − *fps-to-fls g*
by (*intro fls-eqI*) *simp*

instantiation *fls* :: (*group-add*) *uminus*
begin
lift-definition *uminus-fls* :: *'a fls* ⇒ *'a fls* **is** λ*f n*. − *f n*
proof−
fix *f* :: *int* ⇒ *'a* **assume** ∀ ∞*n*. *f* (− *int n*) = *0*
from *this* **obtain** *N* **where** ∀ *n*>*N*. *f* (−*int n*) = *0*
by (*auto simp*: *MOST-nat*)
hence ∀ *n*>*N*. − *f* (−*int n*) = *0* **by** *auto*
hence ∃ *K*. ∀ *n*>*K*. − *f* (−*int n*) = *0* **by** *fast*
thus ∀ ∞*n*. − *f* (− *int n*) = *0* **by** (*simp add*: *MOST-nat*)
qed
instance ..
end

lemma *fls-uminus-nth* [*simp*]: (−*f*) $$ *n* = − (*f* $$ *n*)
by *transfer simp*

lemma *fls-const-uminus*[*simp*]: *fls-const* (−*x*) = −*fls-const x*
by (*intro fls-eqI*) *simp*

lemma *fls-shift-uminus* [*simp*]: *fls-shift m* (− *f*) = − (*fls-shift m f*)
by (*auto intro*: *fls-eqI*)

lemma *fls-regpart-uminus* [*simp*]: *fls-regpart* (− *f*) = − *fls-regpart f*
by (*intro fps-ext*) *simp*

lemma *fls-prpart-uminus* [*simp*] : *fls-prpart* (− *f*) = − *fls-prpart f*
by (*intro poly-eqI*) *simp*

lemma *fps-to-fls-uminus* [*simp*]: *fps-to-fls* (− *f*) = − *fps-to-fls f*
by (*intro fls-eqI*) *simp*

instance *fls* :: (*group-add*) *group-add*
proof
fix *a b* :: *'a fls*
show − *a* + *a* = *0* **by** *transfer simp*

show $a + - b = a - b$ **by** *transfer simp*
qed

instance *fls* :: (*ab-group-add*) *ab-group-add*
proof
fix $a\ b$ :: $'a$ *fls*
show $- a + a = 0$ **by** *transfer simp*
show $a - b = a + - b$ **by** *transfer simp*
qed

lemma *fls-uminus-subdegree* [*simp*]: *fls-subdegree* $(-f)$ = *fls-subdegree f*
by (*cases f=0*) (*auto intro*: *fls-subdegree-eqI*)

lemma *fls-subdegree-minus-sym*: *fls-subdegree* $(g - f)$ = *fls-subdegree* $(f - g)$
using *fls-uminus-subdegree*[*of g−f*] **by** (*simp add*: *algebra-simps*)

lemma *fls-regpart-sub-prpart*: *fls-regpart* $(f$ − *fls-prpart-as-fls f*$)$ = *fls-regpart f*
using *fls-decompose-reg-pr-parts*(*2*)[*of f*]
add-diff-cancel[*of fls-regpart-as-fls f fls-prpart-as-fls f*]
by *simp*

lemma *fls-prpart-sub-regpart*: *fls-prpart* $(f$ − *fls-regpart-as-fls f*$)$ = *fls-prpart f*
using *fls-decompose-reg-pr-parts*(*1*)[*of f*]
add-diff-cancel[*of fls-prpart-as-fls f fls-regpart-as-fls f*]
by *simp*

7.5.3 Multiplication

instantiation *fls* :: ({*comm-monoid-add*, *times*}) *times*
begin
definition *fls-times-def*:
$(*) = (\lambda f\ g.$
fls-shift
(− (*fls-subdegree f* + *fls-subdegree g*))
(*fps-to-fls* (*fls-base-factor-to-fps f* ∗ *fls-base-factor-to-fps g*))
)
instance ..
end

lemma *fls-times-nth-eq0*: $n <$ *fls-subdegree f* + *fls-subdegree g* $\Longrightarrow$ $(f * g)$ $$ $n = 0$
by (*simp add*: *fls-times-def*)

lemma *fls-times-nth*:
fixes *f df g dg*
defines *df* ≡ *fls-subdegree f* **and** *dg* ≡ *fls-subdegree g*
shows $(f * g)$ $$ $n = (\sum i{=}df + dg..n.\ f$ $$ $(i - dg) * g$ $$ $(dg + n - i))$
and $(f * g)$ $$ $n = (\sum i{=}df..n - dg.\ f$ $$ $i * g$ $$ $(n - i))$
and $(f * g)$ $$ $n = (\sum i{=}dg..n - df.\ f$ $$ $(df + i - dg) * g$ $$ $(dg + n -$

```
df − i))
  and     (f ∗ g) $$ n = (∑ i=0..n − (df + dg). f $$ (df + i) ∗ g $$ (n − df −
i))
proof−

  define dfg where dfg ≡ df + dg

  show 4: (f ∗ g) $$ n = (∑ i=0..n − dfg. f $$ (df + i) ∗ g $$ (n − df − i))
  proof (cases n < dfg)
    case False
    from False assms have
      (f ∗ g) $$ n =
        (∑ i = 0..nat (n − dfg). f $$ (df + int i) ∗ g $$ (dg + int (nat (n − dfg)
− i)))
      using fps-mult-nth[of fls-base-factor-to-fps f fls-base-factor-to-fps g]
            fls-base-factor-to-fps-nth[of f]
            fls-base-factor-to-fps-nth[of g]
      by    (simp add: dfg-def fls-times-def algebra-simps)
    moreover from False have index:
      ⋀i. i ∈ {0..nat (n − dfg)} ⟹ dg + int (nat (n − dfg) − i) = n − df − int
i
      by (auto simp: dfg-def)
    ultimately have
      (f ∗ g) $$ n = (∑ i=0..nat (n − dfg). f $$ (df + int i) ∗ g $$ (n − df − int
i))
      by (simp del: of-nat-diff)
    moreover have
      (∑ i=0..nat (n − dfg). f $$ (df + int i) ∗  g $$ (n − df − int i)) =
        (∑ i=0..n − dfg. f $$ (df + i) ∗  g $$ (n − df − i))
    proof (intro sum.reindex-cong)
      show inj-on nat {0..n − dfg} by standard auto
      show {0..nat (n − dfg)} = nat ‘ {0..n − dfg}
      proof
        show {0..nat (n − dfg)} ⊆ nat ‘ {0..n − dfg}
        proof
          fix i assume i ∈ {0..nat (n − dfg)}
          hence i: i ≥ 0 i ≤ nat (n − dfg) by auto
          with False have int i ≥ 0 int i ≤ n − dfg by auto
          hence int i ∈ {0..n − dfg} by simp
          moreover from i(1) have i = nat (int i) by simp
          ultimately show i ∈ nat ‘ {0..n − dfg} by fast
        qed
      qed (auto simp: False)
    qed (simp add: False)
    ultimately show (f ∗ g) $$ n = (∑ i=0..n − dfg. f $$ (df + i) ∗  g $$ (n −
df − i))
      by simp
  qed (simp add: fls-times-nth-eq0 assms dfg-def)
```

have
(∑ i=dfg..n. f $$ (i − dg) ∗ g $$ (dg + n − i)) =
(∑ i=0..n − dfg. f $$ (df + i) ∗ g $$ (n − df − i))
proof (*intro sum.reindex-cong*)
define T **where** T ≡ λi. i + dfg
show *inj-on T* {0..n − dfg} **by** *standard* (*simp add*: *T-def*)
qed (*simp-all add*: *dfg-def algebra-simps*)
with *4* **show** *1*: (f ∗ g) $$ n = (∑ i=dfg..n. f $$ (i − dg) ∗ g $$ (dg + n − i))
by *simp*

have
(∑ i=dfg..n. f $$ (i − dg) ∗ g $$ (dg + n − i)) = (∑ i=df..n − dg. f $$ i ∗ g $$ (n − i))
proof (*intro sum.reindex-cong*)
define T **where** T ≡ λi. i + dg
show *inj-on T* {df..n − dg} **by** *standard* (*simp add*: *T-def*)
qed (*auto simp*: *dfg-def*)
with *1* **show** (f ∗ g) $$ n = (∑ i=df..n − dg. f $$ i ∗ g $$ (n − i))
by *simp*

have
(∑ i=dfg..n. f $$ (i − dg) ∗ g $$ (dg + n − i)) =
(∑ i=dg..n − df. f $$ (df + i − dg) ∗ g $$ (dg + n − df − i))
proof (*intro sum.reindex-cong*)
define T **where** T ≡ λi. i + df
show *inj-on T* {dg..n − df} **by** *standard* (*simp add*: *T-def*)
qed (*simp-all add*: *dfg-def algebra-simps*)
with *1* **show** (f ∗ g) $$ n = (∑ i=dg..n − df. f $$ (df + i − dg) ∗ g $$ (dg + n − df − i))
by *simp*

qed

lemma *fls-times-base* [*simp*]:
(f ∗ g) $$ (*fls-subdegree* f + *fls-subdegree* g) =
(f $$ *fls-subdegree* f) ∗ (g $$ *fls-subdegree* g)
by (*simp add*: *fls-times-nth*(*1*))

instance *fls* :: ({*comm-monoid-add*, *mult-zero*}) *mult-zero*
proof
fix a :: 'a *fls*
have
(0::'a *fls*) ∗ a =
fls-shift (*fls-subdegree* a) (*fps-to-fls* ((0::'a *fps*)∗(*fls-base-factor-to-fps* a)))
by (*simp add*: *fls-times-def*)
moreover have
a ∗ (0::'a *fls*) =
fls-shift (*fls-subdegree* a) (*fps-to-fls* ((*fls-base-factor-to-fps* a)∗(0::'a *fps*)))

```
    by (simp add: fls-times-def)
  ultimately show 0 * a = (0::'a fls) a * 0 = (0::'a fls)
    by auto
qed

lemma fls-mult-one:
  fixes f :: 'a::{comm-monoid-add, mult-zero, monoid-mult} fls
  shows 1 * f = f
  and   f * 1 = f
  using fls-conv-base-factor-to-fps-shift-subdegree[of f]
  by    (simp-all add: fls-times-def fps-one-mult)

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-
  show (fls-const x * f) $$ n = x * f$$n
  proof (cases n<fls-subdegree f)
    case False
    hence {fls-subdegree f..n} = insert (fls-subdegree f) {fls-subdegree f+1..n} by
auto
    thus ?thesis by (simp add: fls-times-nth(1))
  qed (simp add: fls-times-nth-eq0)
  show (f * fls-const x ) $$ n = f$$n * x
  proof (cases n<fls-subdegree f)
    case False
    hence {fls-subdegree f..n} = insert n {fls-subdegree f..n-1} by auto
    thus ?thesis by (simp add: fls-times-nth(1))
  qed (simp add: fls-times-nth-eq0)
qed

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)
  by    (intro fls-eqI) simp

lemma fls-subdegree-add-eq1:
  assumes f ≠ 0 fls-subdegree f < fls-subdegree g
  shows   fls-subdegree (f + g) = fls-subdegree f
proof (intro antisym)
  from assms have *: fls-nth (f + g) (fls-subdegree f) ≠ 0
    by auto
  from * show fls-subdegree (f + g) ≤ fls-subdegree f
    by (rule fls-subdegree-leI)
  from * have f + g ≠ 0
    using fls-nonzeroI by blast
  thus fls-subdegree f ≤ fls-subdegree (f + g)
    using assms(2) fls-plus-subdegree by force
```

qed

lemma *fls-subdegree-add-eq2*:
assumes $g \neq 0$ *fls-subdegree* $g <$ *fls-subdegree* f
shows *fls-subdegree* $(f + g) =$ *fls-subdegree* g
proof (*intro antisym*)
from *assms* **have** $*$: *fls-nth* $(f + g)$ (*fls-subdegree* g) $\neq 0$
by *auto*
from $*$ **show** *fls-subdegree* $(f + g) \leq$ *fls-subdegree* g
by (*rule fls-subdegree-leI*)
from $*$ **have** $f + g \neq 0$
using *fls-nonzeroI* **by** *blast*
thus *fls-subdegree* $g \leq$ *fls-subdegree* $(f + g)$
using *assms*(2) *fls-plus-subdegree* **by** *force*
qed

lemma *fls-subdegree-diff-eq1*:
assumes $f \neq 0$ *fls-subdegree* $f <$ *fls-subdegree* g
shows *fls-subdegree* $(f - g) =$ *fls-subdegree* f
using *fls-subdegree-add-eq1*[*of* f $-g$] *assms* **by** *simp*

lemma *fls-subdegree-diff-eq2*:
assumes $g \neq 0$ *fls-subdegree* $g <$ *fls-subdegree* f
shows *fls-subdegree* $(f - g) =$ *fls-subdegree* g
using *fls-subdegree-add-eq2*[*of* $-g$ f] *assms* **by** *simp*

lemma *nat-minus-fls-subdegree-plus-const-eq*:
nat $(-$*fls-subdegree* $(F +$ *fls-const* $c))$ = *nat* $(-$*fls-subdegree* $F)$
proof (*cases fls-subdegree* $F < 0$)
case *True*
hence *fls-subdegree* $(F +$ *fls-const* $c) =$ *fls-subdegree* F
by (*intro fls-subdegree-add-eq1*) *auto*
thus *?thesis*
by *simp*
next
case *False*
thus *?thesis*
by (*auto simp*: *fls-subdegree-ge0I*)
qed

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
by (*auto intro*: *fls-subdegree-geI simp*: *assms fls-times-nth-eq0*)

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* $(f*g) \geq 0$
using *assms fls-mult-subdegree-ge*[*of f g*]
by *fastforce*

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−
from *assms* **have** *fls-subdegree* $(f*g) \geq$ *fls-subdegree f* + *fls-subdegree g*
using *fls-nonzeroI*[*of f*∗*g fls-subdegree f* + *fls-subdegree g*]
fls-mult-subdegree-ge[*of f g*]
by *simp*
moreover from *assms* **have** *fls-subdegree* $(f*g) \leq$ *fls-subdegree f* + *fls-subdegree g*
by (*intro fls-subdegree-leI*) *simp*
ultimately show *?thesis* **by** *simp*
qed

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*
using *assms*
by (*auto intro*: *fls-subdegree-eqI simp*: *fls-times-nth-eq0*)

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−

show f $*$ (*fls-shift n g*) = *fls-shift n* $(f*g)$
proof (*cases g=0*)
case *False*
hence
f $*$ (*fls-shift n g*) =
fls-shift (− (*fls-subdegree f* + (*fls-subdegree g* − *n*)))
(*fps-to-fls* (*fls-base-factor-to-fps f* ∗ *fls-base-factor-to-fps g*))
unfolding *fls-times-def* **by** (*simp add*: *fls-base-factor-to-fps-shift*)
thus f $*$ (*fls-shift n g*) = *fls-shift n* $(f*g)$
by (*simp add*: *algebra-simps fls-times-def*)
qed *auto*

show (*fls-shift n f*)$*g$ = *fls-shift n* $(f*g)$
proof (*cases f=0*)
case *False*
hence
(*fls-shift n f*)$*g$ =
fls-shift (− ((*fls-subdegree f* − *n*) + *fls-subdegree g*))

```
      (fps-to-fls (fls-base-factor-to-fps f * fls-base-factor-to-fps g))
    unfolding fls-times-def by (simp add: fls-base-factor-to-fps-shift)
  thus (fls-shift n f) * g = fls-shift n (f*g)
    by (simp add: algebra-simps fls-times-def)
 qed auto

qed

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
 using fls-shifted-times-simps(1)[of f n g] fls-shifted-times-simps(2)[of n f g]
 by   simp

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)
 by   (simp add: fls-shifted-times-simps)

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)
 using fls-base-factor-shift[of fls-subdegree g f*g]
       fls-base-factor-shift[of fls-subdegree f f*g]
 by   (simp-all add: fls-shifted-times-simps)

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)
 using fls-base-factor-mult-base-factor(1)[of fls-base-factor f g]
       fls-base-factor-mult-base-factor(2)[of f g]
 by   simp

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
 by   (cases f≠0 ∧ g≠0)
      (auto simp: fls-times-both-shifted-simp)

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)
 by (simp add: fls-times-both-shifted-simp)
```

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*)
by (*simp add*: *fls-times-both-shifted-simp*)

lemma *fls-times-conv-regpart*:
fixes *f g* :: *'a*::{*comm-monoid-add*,*mult-zero*} *fls*
assumes *fls-subdegree f* ≥ *0 fls-subdegree g* ≥ *0*
shows *fls-regpart* (*f* ∗ *g*) = *fls-regpart f* ∗ *fls-regpart g*
proof−
from *assms* **have** *1*:
f ∗ *g* =
fls-shift (− (*fls-subdegree f* + *fls-subdegree g*)) (
fps-to-fls (
fps-shift (*nat* (*fls-subdegree f*) + *nat* (*fls-subdegree g*)) (
fls-regpart f ∗ *fls-regpart g*
)
)
)
by (*simp add*:
fls-times-def fls-base-factor-to-fps-conv-fps-shift[*symmetric*]
fls-regpart-subdegree-conv fps-shift-mult-both[*symmetric*]
)
show *?thesis*
proof (*cases fls-regpart f* ∗ *fls-regpart g* = *0*)
case *False*
with *assms* **have**
subdegree (*fls-regpart f* ∗ *fls-regpart g*) ≥
nat (*fls-subdegree f*) + *nat* (*fls-subdegree g*)
by (*simp add*: *fps-mult-subdegree-ge fls-regpart-subdegree-conv*[*symmetric*])
with *1 assms* **show** *?thesis* **by** *simp*
qed (*simp add*: *1*)
qed

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*)
using *fls-base-factor-mult-both-base-factor*[*of f g*]
fps-unit-factor-fls-regpart[*of fls-base-factor f* ∗ *fls-base-factor g*]
fls-base-factor-subdegree[*of f*] *fls-base-factor-subdegree*[*of g*]
fls-mult-subdegree-ge-0[*of fls-base-factor f fls-base-factor g*]
fls-times-conv-regpart[*of fls-base-factor f fls-base-factor g*]
by *simp*

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*) ≠ *0*
shows *fls-base-factor-to-fps* (*f* ∗ *g*) = *fls-base-factor-to-fps f* ∗ *fls-base-factor-to-fps g*
using *assms fls-mult-nonzero-base-subdegree-eq*[*of f g*]
fls-times-base-factor-conv-shifted-times[*of f g*]
fls-times-conv-regpart[*of fls-base-factor f fls-base-factor g*]
fls-base-factor-subdegree[*of f*] *fls-base-factor-subdegree*[*of g*]
by *fastforce*

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*
using *fls-base-factor-to-fps-mult'*[*of f g*]
by (*cases f*=*0* ∨ *g*=*0*) *auto*

lemma *fls-times-conv-fps-times*:
fixes *f g* :: *'a*::{*comm-monoid-add*,*mult-zero*} *fls*
assumes *fls-subdegree f* ≥ *0 fls-subdegree g* ≥ *0*
shows *f* ∗ *g* = *fps-to-fls* (*fls-regpart f* ∗ *fls-regpart g*)
using *assms fls-mult-subdegree-ge*[*of f g*]
by (*cases f* ∗ *g* = *0*) (*simp all add*: *fls-times-conv-regpart*[*symmetric*])

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*)
using *fls-subdegree-fls-to-fps-gt0 fls-times-conv-regpart*[*symmetric*]
by *fastforce*

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 (*intro fls-eq-conv-fps-eqI*, *rule fls-subdegree-fls-to-fps-gt0*)
show *fls-subdegree* (*fps-to-fls f* ∗ *fps-to-fls g*) ≥ *0*
proof (*cases fps-to-fls f* ∗ *fps-to-fls g* = *0*)
case *False* **thus** *?thesis*
using *fls-mult-subdegree-ge fls-subdegree-fls-to-fps-gt0*[*of f*]
fls-subdegree-fls-to-fps-gt0[*of g*]
by *fastforce*
qed *simp*
qed (*simp add*: *fps-times-conv-fls-times*)

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*
by (*simp all add*: *fls-X-conv-shift-1 fls-mult-one fls-shifted-times-simps*)

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*
by (*auto simp*: *fls-X-times-conv-shift assms*)

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*
by (*auto simp*: *fls-X-times-conv-shift fls-shift-eq0-iff assms*)

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*
using *fls-base-factor-shift*[*of* −*1 f*]
by (*auto simp*: *fls-X-times-conv-shift*)

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*
by (*simp-all add*: *fls-X-inv-conv-shift-1 fls-mult-one fls-shifted-times-simps*)

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*
by (*auto simp*: *fls-X-inv-times-conv-shift assms*)

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*
by (*auto simp*: *fls-X-inv-times-conv-shift fls-shift-eq0-iff assms*)

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*
using *fls-base-factor-shift*[*of 1 f*]

by (*auto simp*: *fls-X-inv-times-conv-shift*)

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*)
using *assms*
by (*simp add*: *fls-times-conv-fps-times fls-subdegree-fls-to-fps-gt0 mult.assoc*)

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*)
by (*simp add*: *fls-mult-assoc-subdegree-ge-0 del*: *fls-base-factor-def*)

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−
have *fls-subdegree* (*f*+*g*) ≥ *0*
proof (*cases f*+*g* = *0*)
case *False*
with *assms*(*1*,*2*) **show** *?thesis*
using *fls-plus-subdegree* **by** *fastforce*
qed *simp*
with *assms* **show** (*f* + *g*) ∗ *h* = *f* ∗ *h* + *g* ∗ *h h* ∗ (*f* + *g*) = *h* ∗ *f* + *h* ∗ *g*
using *distrib-right*[*of fls-regpart f*] *distrib-left*[*of fls-regpart h*]
by (*simp-all add*: *fls-times-conv-fps-times*)
qed

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*
by (*simp add*: *fls-mult-distrib-subdegree-ge-0 del*: *fls-base-factor-def*)

instance *fls* :: (*semiring-0*) *semiring-0*
proof

fix *a b c* :: *'a fls*
have
a ∗ *b* ∗ *c* =
fls-shift (− (*fls-subdegree a* + *fls-subdegree b* + *fls-subdegree c*))
(*fls-base-factor a* ∗ *fls-base-factor b* ∗ *fls-base-factor c*)
by (*simp add*: *fls-times-both-shifted-simp*)

```
moreover have
  a * (b * c) =
    fls-shift (- (fls-subdegree a + fls-subdegree b + fls-subdegree c))
      (fls-base-factor a * fls-base-factor b * fls-base-factor c)
 using fls-mult-assoc-base-factor[of a b c] by (simp add: fls-times-both-shifted-simp)
ultimately show a * b * c = a * (b * c) by simp

have ab:
  fls-subdegree (fls-shift (min (fls-subdegree a) (fls-subdegree b)) a) ≥ 0
  fls-subdegree (fls-shift (min (fls-subdegree a) (fls-subdegree b)) b) ≥ 0
  by (simp-all add: fls-shift-nonneg-subdegree)
have
  (a + b) * c =
    fls-shift (- (min (fls-subdegree a) (fls-subdegree b) + fls-subdegree c)) (
      (
        fls-shift (min (fls-subdegree a) (fls-subdegree b)) a +
        fls-shift (min (fls-subdegree a) (fls-subdegree b)) b
      ) * fls-base-factor c)
  using fls-times-both-shifted-simp[of
          -min (fls-subdegree a) (fls-subdegree b)
          fls-shift (min (fls-subdegree a) (fls-subdegree b)) a +
          fls-shift (min (fls-subdegree a) (fls-subdegree b)) b
          -fls-subdegree c fls-base-factor c
        ]
  by    simp
also have
  ... =
    fls-shift (-(min (fls-subdegree a) (fls-subdegree b) + fls-subdegree c))
      (fls-shift (min (fls-subdegree a) (fls-subdegree b)) a * fls-base-factor c)
    +
    fls-shift (-(min (fls-subdegree a) (fls-subdegree b) + fls-subdegree c))
      (fls-shift (min (fls-subdegree a) (fls-subdegree b)) b * fls-base-factor c)
  using ab
  by    (simp add: fls-mult-distrib-subdegree-ge-0(1) del: fls-base-factor-def)
finally show (a + b) * c = a * c + b * c by (simp add: fls-times-both-shifted-simp)

have bc:
  fls-subdegree (fls-shift (min (fls-subdegree b) (fls-subdegree c)) b) ≥ 0
  fls-subdegree (fls-shift (min (fls-subdegree b) (fls-subdegree c)) c) ≥ 0
  by (simp-all add: fls-shift-nonneg-subdegree)
have
  a * (b + c) =
    fls-shift (- (fls-subdegree a + min (fls-subdegree b) (fls-subdegree c))) (
      fls-base-factor a * (
        fls-shift (min (fls-subdegree b) (fls-subdegree c)) b +
        fls-shift (min (fls-subdegree b) (fls-subdegree c)) c
      )
    )
```

```
    using fls-times-both-shifted-simp[of
          −fls-subdegree a fls-base-factor a
          −min (fls-subdegree b) (fls-subdegree c)
          fls-shift (min (fls-subdegree b) (fls-subdegree c)) b +
          fls-shift (min (fls-subdegree b) (fls-subdegree c)) c
        ]
    by    simp
  also have
    ... =
      fls-shift (−(fls-subdegree a + min (fls-subdegree b) (fls-subdegree c)))
        (fls-base-factor a * fls-shift (min (fls-subdegree b) (fls-subdegree c)) b)
      +
      fls-shift (−(fls-subdegree a + min (fls-subdegree b) (fls-subdegree c)))
        (fls-base-factor a * fls-shift (min (fls-subdegree b) (fls-subdegree c)) c)

    using bc
    by    (simp add: fls-mult-distrib-subdegree-ge-0(2) del: fls-base-factor-def)
  finally show a * (b + c) = a * b + a * c by (simp add: fls-times-both-shifted-simp)

qed

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
  using   assms
  by      (simp add: fls-times-conv-fps-times mult.commute)

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
  by    (simp add: fls-mult-commute-subdegree-ge-0 del: fls-base-factor-def)

instance fls :: (comm-semiring-0) comm-semiring-0
proof
  fix a b c :: 'a fls
  show a * b = b * a
    using fls-times-conv-base-factor-times[of a b] fls-times-conv-base-factor-times[of
b a]
          fls-mult-commute-base-factor[of a b]
    by    (simp add: add.commute)
qed (simp add: distrib-right)

instance fls :: (semiring-1) semiring-1
  by (standard, simp-all add: fls-mult-one)

lemma fls-of-nat: (of-nat n :: 'a::semiring-1 fls) = fls-const (of-nat n)
  by (induct n) (auto intro: fls-eqI)
```

lemma *fls-of-nat-nth*: *of-nat n $$ k = (if k=0 then of-nat n else 0)*
by (*simp add*: *fls-of-nat*)

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*
by (*simp-all add*: *fls-of-nat*)

lemma *fls-subdegree-of-nat* [*simp*]: *fls-subdegree* (*of-nat n*) = *0*
by (*simp add*: *fls-of-nat*)

lemma *fls-shift-of-nat-nth*:
fls-shift k (*of-nat a*) $$ *n* = (*if n=−k then of-nat a else 0*)
by (*simp add*: *fls-of-nat fls-shift-const-nth*)

lemma *fls-base-factor-of-nat* [*simp*]:
fls-base-factor (*of-nat n* :: *'a*::*semiring-1 fls*) = (*of-nat n* :: *'a fls*)
by (*simp add*: *fls-of-nat*)

lemma *fls-regpart-of-nat* [*simp*]: *fls-regpart* (*of-nat n*) = (*of-nat n* :: *'a*::*semiring-1 fps*)
by (*simp add*: *fls-of-nat fps-of-nat*)

lemma *fls-prpart-of-nat* [*simp*]: *fls-prpart* (*of-nat n*) = *0*
by (*simp add*: *fls-prpart-eq0-iff*)

lemma *fls-base-factor-to-fps-of-nat*:
fls-base-factor-to-fps (*of-nat n*) = (*of-nat n* :: *'a*::*semiring-1 fps*)
by *simp*

lemma *fps-to-fls-of-nat*:
fps-to-fls (*of-nat n*) = (*of-nat n* :: *'a*::*semiring-1 fls*)
proof −
have *fps-to-fls* (*of-nat n*) = *fps-to-fls* (*fps-const* (*of-nat n*))
by (*simp add*: *fps-of-nat*)
thus *?thesis* **by** (*simp add*: *fls-of-nat*)
qed

lemma *fps-to-fls-numeral* [*simp*]: *fps-to-fls* (*numeral n*) = *numeral n*
by (*metis fps-to-fls-of-nat of-nat-numeral*)

lemma *fls-const-power*: *fls-const* (*a ^ b*) = *fls-const a ^ b*
by (*induction b*) (*auto simp flip*: *fls-const-mult-const*)

lemma *fls-const-numeral* [*simp*]: *fls-const* (*numeral n*) = *numeral n*
by (*metis fls-of-nat of-nat-numeral*)

lemma *fls-mult-of-numeral-nth* [*simp*]:
shows (*numeral k * f*) $$ *n = numeral k * f* $$ *n*

and $(f * numeral\ k)\ \$\$\ n = f\ \$\$\ n * numeral\ k$
by (*metis fls-const-numeral fls-mult-const-nth*)+

lemma *fls-nth-numeral'* [*simp*]:
$numeral\ n\ \$\$\ 0 = numeral\ n\ k \neq 0 \Longrightarrow numeral\ n\ \$\$\ k = 0$
by (*metis fls-const-nth fls-const-numeral*)+

instance *fls* :: (*comm-semiring-1*) *comm-semiring-1*
by *standard simp*

instance *fls* :: (*ring*) *ring* **..**

instance *fls* :: (*comm-ring*) *comm-ring* **..**

instance *fls* :: (*ring-1*) *ring-1* **..**

lemma *fls-of-int-nonneg*: (*of-int* (*int n*) :: *'a*::*ring-1 fls*) = *fls-const* (*of-int* (*int n*))
by (*induct n*) (*auto intro*: *fls-eqI*)

lemma *fls-of-int*: (*of-int i* :: *'a*::*ring-1 fls*) = *fls-const* (*of-int i*)
proof (*induct i*)
case (*neg i*)
have *of-int* (*int* (*Suc i*)) = *fls-const* (*of-int* (*int* (*Suc i*)) :: *'a*)
using *fls-of-int-nonneg*[*of Suc i*] **by** *simp*
hence $-$ *of-int* (*int* (*Suc i*)) = $-$ *fls-const* (*of-int* (*int* (*Suc i*)) :: *'a*)
by *simp*
thus *?case* **by** (*simp add*: *fls-const-uminus*[*symmetric*])
qed (*rule fls-of-int-nonneg*)

lemma *fls-of-int-nth*: *of-int n* $\$\$\ k$ = (*if k=0 then of-int n else 0*)
by (*simp add*: *fls-of-int*)

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*
by (*simp-all add*: *fls-of-int*)

lemma *fls-subdegree-of-int* [*simp*]: *fls-subdegree* (*of-int i*) = *0*
by (*simp add*: *fls-of-int*)

lemma *fls-shift-of-int-nth*:
fls-shift k (*of-int i*) $\$\$\ n$ = (*if n=−k then of-int i else 0*)
by (*simp add*: *fls-of-int-nth*)

lemma *fls-base-factor-of-int* [*simp*]:
fls-base-factor (*of-int i* :: *'a*::*ring-1 fls*) = (*of-int i* :: *'a fls*)
by (*simp add*: *fls-of-int*)

lemma *fls-regpart-of-int* [*simp*]:
fls-regpart (*of-int i*) = (*of-int i* :: *'a*::*ring-1 fps*)
by (*simp add*: *fls-of-int fps-of-int*)

lemma *fls-prpart-of-int* [*simp*]: *fls-prpart* (*of-int n*) = *0*
by (*simp add*: *fls-prpart-eq0-iff*)

lemma *fls-base-factor-to-fps-of-int*:
fls-base-factor-to-fps (*of-int i*) = (*of-int i* :: *'a*::*ring-1 fps*)
by *simp*

lemma *fps-to-fls-of-int*:
fps-to-fls (*of-int i*) = (*of-int i* :: *'a*::*ring-1 fls*)
proof −
have *fps-to-fls* (*of-int i*) = *fps-to-fls* (*fps-const* (*of-int i*))
by (*simp add*: *fps-of-int*)
thus *?thesis* **by** (*simp add*: *fls-of-int*)
qed

instance *fls* :: (*comm-ring-1*) *comm-ring-1* **..**

instance *fls* :: (*semiring-no-zero-divisors*) *semiring-no-zero-divisors*
proof
fix *a b* :: *'a fls*
assume *a* ≠ *0* **and** *b* ≠ *0*
hence (*a* ∗ *b*) $$ (*fls-subdegree a* + *fls-subdegree b*) ≠ *0* **by** *simp*
thus *a* ∗ *b* ≠ *0* **using** *fls-nonzeroI* **by** *fast*
qed

instance *fls* :: (*semiring-1-no-zero-divisors*) *semiring-1-no-zero-divisors* **..**

instance *fls* :: (*ring-no-zero-divisors*) *ring-no-zero-divisors* **..**

instance *fls* :: (*ring-1-no-zero-divisors*) *ring-1-no-zero-divisors* **..**

instance *fls* :: (*idom*) *idom* **..**

lemma *semiring-char-fls* [*simp*]: *CHAR*(*'a* :: *comm-semiring-1 fls*) = *CHAR*(*'a*)
by (*rule CHAR-eqI*) (*auto simp*: *fls-of-nat of-nat-eq-0-iff-char-dvd fls-const-nonzero*)

instance *fls* :: ({*semiring-prime-char*,*comm-semiring-1*}) *semiring-prime-char*
by (*rule semiring-prime-charI*) *auto*
instance *fls* :: ({*comm-semiring-prime-char*,*comm-semiring-1*}) *comm-semiring-prime-char*
by *standard*
instance *fls* :: ({*comm-ring-prime-char*,*comm-semiring-1*}) *comm-ring-prime-char*
by *standard*
instance *fls* :: ({*idom-prime-char*,*comm-semiring-1*}) *idom-prime-char*
by *standard*

lemma *fls-subdegree-numeral* [*simp*]: *fls-subdegree* (*numeral n*) = *0*
by (*metis fls-subdegree-of-nat of-nat-numeral*)

lemma *fls-regpart-numeral* [*simp*]: *fls-regpart* (*numeral n*) = *numeral n*
by (*metis fls-regpart-of-nat of-nat-numeral*)

7.5.4 Powers

lemma *fls-subdegree-prod*:
fixes *F* :: *'a* ⇒ *'b* :: *field-char-0 fls*
assumes ⋀*x*. *x* ∈ *I* ⟹ *F x* ≠ *0*
shows *fls-subdegree* (∏ *x*∈*I*. *F x*) = (∑ *x*∈*I*. *fls-subdegree* (*F x*))
using *assms* **by** (*induction I rule*: *infinite-finite-induct*) *auto*

lemma *fls-subdegree-prod'*:
fixes *F* :: *'a* ⇒ *'b* :: *field-char-0 fls*
assumes ⋀*x*. *x* ∈ *I* ⟹ *fls-subdegree* (*F x*) ≠ *0*
shows *fls-subdegree* (∏ *x*∈*I*. *F x*) = (∑ *x*∈*I*. *fls-subdegree* (*F x*))
proof (*intro fls-subdegree-prod*)
show *F x* ≠ *0* **if** *x* ∈ *I* **for** *x*
using *assms*[*OF that*] **by** *auto*
qed

lemma *fls-pow-subdegree-ge*:
f^*n* ≠ *0* ⟹ *fls-subdegree* (*f*^*n*) ≥ *n* ∗ *fls-subdegree f*
proof (*induct n*)
case (*Suc n*) **thus** *?case*
using *fls-mult-subdegree-ge*[*of f f*^*n*] **by** (*fastforce simp*: *algebra-simps*)
qed *simp*

lemma *fls-pow-nth-below-subdegree*:
k < *n* ∗ *fls-subdegree f* ⟹ (*f*^*n*) $$ *k* = *0*
using *fls-pow-subdegree-ge*[*of f n*] **by** (*cases f*^*n* = *0*) *auto*

lemma *fls-pow-base* [*simp*]:
(*f* ^ *n*) $$ (*n* ∗ *fls-subdegree f*) = (*f* $$ *fls-subdegree f*) ^ *n*
proof (*induct n*)
case (*Suc n*)
show *?case*
proof (*cases Suc n* ∗ *fls-subdegree f* < *fls-subdegree f* + *fls-subdegree* (*f*^*n*))
case *True* **with** *Suc* **show** *?thesis*
by (*simp-all add*: *fls-times-nth-eq0 distrib-right*)
next
case *False*
from *False* **have**
{*0*..*int n* ∗ *fls-subdegree f* − *fls-subdegree* (*f* ^ *n*)} =
insert 0 {*1*..*int n* ∗ *fls-subdegree f* − *fls-subdegree* (*f* ^ *n*)}
by (*auto simp*: *algebra-simps*)

with *False Suc* **show** *?thesis*
by (*simp add*: *algebra-simps fls-times-nth*(*4*) *fls-pow-nth-below-subdegree*)
qed
qed *simp*

lemma *fls-pow-subdegree-eqI*:
(*f* $$ *fls-subdegree f*) ^ *n* ≠ *0* ⟹ *fls-subdegree* (*f*^*n*) = *n* * *fls-subdegree f*
using *fls-pow-nth-below-subdegree* **by** (*fastforce intro*: *fls-subdegree-eqI*)

lemma *fls-unit-base-subdegree-power*:
x * *f* $$ *fls-subdegree f* = *1* ⟹ *fls-subdegree* (*f* ^ *n*) = *n* * *fls-subdegree f*
f $$ *fls-subdegree f* * *y* = *1* ⟹ *fls-subdegree* (*f* ^ *n*) = *n* * *fls-subdegree f*
proof−
show *x* * *f* $$ *fls-subdegree f* = *1* ⟹ *fls-subdegree* (*f* ^ *n*) = *n* * *fls-subdegree f*
using *left-right-inverse-power*[*of x f* $$ *fls-subdegree f n*]
by (*auto intro*: *fls-pow-subdegree-eqI*)
show *f* $$ *fls-subdegree f* * *y* = *1* ⟹ *fls-subdegree* (*f* ^ *n*) = *n* * *fls-subdegree f*
using *left-right-inverse-power*[*of f* $$ *fls-subdegree f y n*]
by (*auto intro*: *fls-pow-subdegree-eqI*)
qed

lemma *fls-base-dvd1-subdegree-power*:
f $$ *fls-subdegree f dvd 1* ⟹ *fls-subdegree* (*f* ^ *n*) = *n* * *fls-subdegree f*
using *fls-unit-base-subdegree-power* **unfolding** *dvd-def* **by** *auto*

lemma *fls-pow-subdegree-ge0*:
assumes *fls-subdegree f* ≥ *0*
shows *fls-subdegree* (*f*^*n*) ≥ *0*
proof (*cases f*^*n* = *0*)
case *False*
moreover from *assms* **have** *int n* * *fls-subdegree f* ≥ *0* **by** *simp*
ultimately show *?thesis* **using** *fls-pow-subdegree-ge* **by** *fastforce*
qed *simp*

lemma *fls-subdegree-pow*:
fixes *f* :: *'a*::*semiring-1-no-zero-divisors fls*
shows *fls-subdegree* (*f* ^ *n*) = *n* * *fls-subdegree f*
proof (*cases f*=*0*)
case *False* **thus** *?thesis* **by** (*induct n*) (*simp-all add*: *algebra-simps*)
qed (*cases n*=*0*, *auto simp*: *zero-power*)

lemma *fls-shifted-pow*:
(*fls-shift m f*) ^ *n* = *fls-shift* (*n*∗*m*) (*f* ^ *n*)
by (*induct n*) (*simp-all add*: *fls-times-both-shifted-simp algebra-simps*)

lemma *fls-pow-conv-fps-pow*:
assumes *fls-subdegree f* ≥ *0*
shows *f* ^ *n* = *fps-to-fls* ((*fls-regpart f*) ^ *n*)
proof (*induct n*)

case (*Suc n*) **with** *assms* **show** *?case*
using *fls-pow-subdegree-ge0*[*of f n*]
by (*simp add*: *fls-times-conv-fps-times*)
qed *simp*

lemma *fps-to-fls-power*: *fps-to-fls* (*f* ^ *n*) = *fps-to-fls f* ^ *n*
by (*simp add*: *fls-pow-conv-fps-pow fls-subdegree-fls-to-fps-gt0*)

lemma *fls-pow-conv-regpart*:
fls-subdegree f ≥ *0* ⟹ *fls-regpart* (*f* ^ *n*) = (*fls-regpart f*) ^ *n*
by (*simp add*: *fls-pow-conv-fps-pow*)

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*
by (*simp add*: *fls-X-conv-shift-1 fls-shifted-pow*)

lemma *fls-X-inv-power-conv-shift-1*: *fls-X-inv* ^ *n* = *fls-shift n 1*
by (*simp add*: *fls-X-inv-conv-shift-1 fls-shifted-pow*)

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*
by (*simp add*: *fls-shift-eq0-iff*)

lemma *fls-X-intpow-power*: (*fls-X-intpow i*) ^ *n* = *fls-X-intpow* (*n* ∗ *i*)
by (*simp add*: *fls-shifted-pow*)

lemma *fls-X-power-nth* [*simp*]: *fls-X* ^ *n* $$ *k* = (*if k*=*n then 1 else 0*)
by (*simp add*: *fls-X-power-conv-shift-1*)

lemma *fls-X-inv-power-nth* [*simp*]: *fls-X-inv* ^ *n* $$ *k* = (*if k*=−*n then 1 else 0*)
by (*simp add*: *fls-X-inv-power-conv-shift-1*)

lemma *fls-X-pow-nonzero*[*simp*]: (*fls-X* ^ *n* :: *'a* :: *semiring-1 fls*) ≠ *0*
proof
assume (*fls-X* ^ *n* :: *'a fls*) = *0*
hence (*fls-X* ^ *n* :: *'a fls*) $$ *n* = *0* **by** *simp*
thus *False* **by** *simp*
qed

lemma *fls-X-inv-pow-nonzero*[*simp*]: (*fls-X-inv* ^ *n* :: *'a* :: *semiring-1 fls*) ≠ *0*
proof
assume (*fls-X-inv* ^ *n* :: *'a fls*) = *0*
hence (*fls-X-inv* ^ *n* :: *'a fls*) $$ −*n* = *0* **by** *simp*
thus *False* **by** *simp*

qed

lemma *fls-subdegree-fls-X-pow* [*simp*]: *fls-subdegree* (*fls-X* ^ *n*) = *n*
by (*intro fls-subdegree-eqI*) (*simp-all add*: *fls-X-power-conv-shift-1*)

lemma *fls-subdegree-fls-X-inv-pow* [*simp*]: *fls-subdegree* (*fls-X-inv* ^ *n*) = −*n*
by (*intro fls-subdegree-eqI*) (*simp-all add*: *fls-X-inv-power-conv-shift-1*)

lemma *fls-subdegree-fls-X-intpow* [*simp*]:
fls-subdegree ((*fls-X-intpow i*) :: *'a*::*zero-neq-one fls*) = *i*
by *simp*

lemma *fls-X-pow-conv-fps-X-pow*: *fls-regpart* (*fls-X* ^ *n*) = *fps-X* ^ *n*
by (*simp add*: *fls-pow-conv-regpart*)

lemma *fls-X-inv-pow-regpart*: *n* > *0* ⟹ *fls-regpart* (*fls-X-inv* ^ *n*) = *0*
by (*auto intro*: *fps-ext simp*: *fls-X-inv-power-conv-shift-1*)

lemma *fls-X-intpow-regpart*:
fls-regpart (*fls-X-intpow i*) = (*if i*≥*0 then fps-X* ^ *nat i else 0*)
using *fls-X-pow-conv-fps-X-pow*[*of nat i*]
fls-regpart-shift-conv-fps-shift[*of* −*i 1*]
by (*auto simp*: *fls-X-power-conv-shift-1 fps-shift-one*)

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*
using *fls-times-both-shifted-simp*[*of* −*int n 1 0 f*]
fls-times-both-shifted-simp[*of 0 f* −*int n 1*]
by (*simp-all add*: *fls-X-power-conv-shift-1*)

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*
using *fls-times-both-shifted-simp*[*of int n 1 0 f*]
fls-times-both-shifted-simp[*of 0 f int n 1*]
by (*simp-all add*: *fls-X-inv-power-conv-shift-1*)

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*
by (*simp-all add*: *fls-shifted-times-simps*)

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*
by (*simp add*: *fls-X-intpow-times-conv-shift*)

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*)
by (*simp add*: *fls-times-both-shifted-simp*)

lemma *fls-X-intpow-diff-conv-times*:
fls-X-intpow (*i*−*j*) = (*fls-X-intpow i* :: *'a*::*semiring-1 fls*) ∗ *fls-X-intpow* (−*j*)
using *fls-X-intpow-times-fls-X-intpow*[*of i* −*j*,*symmetric*] **by** *simp*

lemma *fls-mult-fls-X-power-nonzero*:
assumes *f* ≠ *0*
shows *fls-X* ^ *n* ∗ *f* ≠ *0 f* ∗ *fls-X* ^ *n* ≠ *0*
by (*auto simp*: *fls-X-power-times-conv-shift fls-shift-eq0-iff assms*)

lemma *fls-mult-fls-X-inv-power-nonzero*:
assumes *f* ≠ *0*
shows *fls-X-inv* ^ *n* ∗ *f* ≠ *0 f* ∗ *fls-X-inv* ^ *n* ≠ *0*
by (*auto simp*: *fls-X-inv-power-times-conv-shift fls-shift-eq0-iff assms*)

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*
by (*auto simp*: *fls-X-intpow-times-conv-shift fls-shift-eq0-iff assms*)

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*
by (*auto simp*: *fls-X-power-times-conv-shift assms*)

lemma *fls-subdegree-mult-fls-X-inv-power*:
assumes *f* ≠ *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*
by (*auto simp*: *fls-X-inv-power-times-conv-shift assms*)

lemma *fls-subdegree-mult-fls-X-intpow*:
fixes *f* :: *'a*::*semiring-1 fls*
assumes *f* ≠ *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*
by (*auto simp*: *fls-X-intpow-times-conv-shift assms*)

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*
using *fls-X-power-conv-shift-1*[*of Suc n*, *symmetric*]
by (*simp-all add*: *fls-X-conv-shift-1 fls-X-inv-power-conv-shift-1*)

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*
using *fls-X-inv-power-conv-shift-1*[*of Suc n*, *symmetric*]
by (*simp-all add*: *fls-X-inv-conv-shift-1 fls-X-power-conv-shift-1*)

lemma *fls-X-power-base-factor*: *fls-base-factor* (*fls-X* ^ *n*) = *1*
by (*simp add*: *fls-X-power-conv-shift-1*)

lemma *fls-X-inv-power-base-factor*: *fls-base-factor* (*fls-X-inv* ^ *n*) = *1*
by (*simp add*: *fls-X-inv-power-conv-shift-1*)

lemma *fls-X-intpow-base-factor*: *fls-base-factor* (*fls-X-intpow i*) = *1*
using *fls-base-factor-shift*[*of* −*i 1*] **by** *simp*

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*
using *fls-base-factor-shift*[*of* −*int n f*]
by (*auto simp*: *fls-X-power-times-conv-shift*)

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*
using *fls-base-factor-shift*[*of int n f*]
by (*auto simp*: *fls-X-inv-power-times-conv-shift*)

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*
using *fls-base-factor-shift*[*of* −*i f*]
by (*auto simp*: *fls-X-intpow-times-conv-shift*)

lemma *fls-X-power-base-factor-to-fps*: *fls-base-factor-to-fps* (*fls-X* ^ *n*) = *1*
proof−
define *X* **where** *X* ≡ *fls-X* :: *'a*::*semiring-1 fls*
hence *fls-base-factor* (*X* ^ *n*) = *1* **using** *fls-X-power-base-factor* **by** *simp*
thus *fls-base-factor-to-fps* (*X*^*n*) = *1* **by** *simp*
qed

lemma *fls-X-inv-power-base-factor-to-fps*: *fls-base-factor-to-fps* (*fls-X-inv* ^ *n*) = *1*
proof−
define *iX* **where** *iX* ≡ *fls-X-inv* :: *'a*::*semiring-1 fls*
hence *fls-base-factor* (*iX* ^ *n*) = *1* **using** *fls-X-inv-power-base-factor* **by** *simp*
thus *fls-base-factor-to-fps* (*iX*^*n*) = *1* **by** *simp*
qed

lemma *fls-X-intpow-base-factor-to-fps*: *fls-base-factor-to-fps* (*fls-X-intpow i*) = *1*
proof−
define *f* :: *'a fls* **where** *f* ≡ *fls-X-intpow i*
moreover have *fls-base-factor* (*fls-X-intpow i*) = *1* **by** (*rule fls-X-intpow-base-factor*)
ultimately have *fls-base-factor f* = *1* **by** *simp*
thus *fls-base-factor-to-fps f* = *1* **by** *simp*
qed

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*
by (*simp-all add*: *fls-times-both-shifted-simp*)

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*))
using *fls-mult-subdegree-ge*[*of f g*]
fls-times-base-factor-conv-shifted-times[*of f g*]
by (*simp-all add*: *assms fls-X-power-conv-shift-1 algebra-simps*)

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*)))
using *fls-times-conv-regpart*[*of fls-base-factor f fls-base-factor g*]
fls-base-factor-subdegree[*of f*] *fls-base-factor-subdegree*[*of g*]
fls-normalized-product-of-inverses(*1*)[*OF assms*]
by (*force simp*: *fls-X-pow-conv-fps-X-pow*)

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 ..
end

lemma *fls-inverse-def'*:
inverse f = *fls-right-inverse f* (*inverse* (*f* $$ *fls-subdegree f*))
by (*simp add*: *fls-inverse-def fps-inverse-def*)

lemma *fls-lr-inverse-base*:
fls-left-inverse f x $$ (−*fls-subdegree f*) = *x*
fls-right-inverse f y $$ (−*fls-subdegree f*) = *y*
by *auto*

lemma *fls-inverse-base*:
f ≠ *0* ⟹ *inverse f* $$ (−*fls-subdegree f*) = *inverse* (*f* $$ *fls-subdegree f*)
by (*simp add*: *fls-inverse-def'*)

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*
by (*simp-all add*: *fps-lr-inverse-starting0*)

lemma *fls-lr-inverse-eq0-imp-starting0*:
fls-left-inverse f x = *0* ⟹ *x* = *0*
fls-right-inverse f x = *0* ⟹ *x* = *0*
by (*metis fls-lr-inverse-base fls-nonzeroI*)+

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*
using *fls-lr-inverse-starting0 fls-lr-inverse-eq0-imp-starting0*
by *auto*

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*)

using *fls-lr-inverse-eq-0-iff*(*2*)[*of f inverse* (*f* $$ *fls-subdegree f*)]
by (*simp add*: *fls-inverse-def'*)

lemma *fls-inverse-eq-0-iff*[*simp*]:
inverse f = (*0*:: ('*a*::*division-ring*) *fls*) ⟷ *f* $$ *fls-subdegree f* = *0*
using *fls-inverse-eq-0-iff'*[*of f*] **by** (*cases f*=*0*) *auto*

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*
by (*simp-all add*: *fps-const-lr-inverse*)

lemma *fls-inverse-const*:
fixes *a* :: '*a*::{*comm-monoid-add*,*inverse*,*mult-zero*,*uminus*}
shows *inverse* (*fls-const a*) = *fls-const* (*inverse a*)
using *fls-lr-inverse-const*(*2*)
by (*auto simp*: *fls-inverse-def'*)

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*
using *fls-lr-inverse-const*
by (*auto simp*: *fls-of-nat*)

lemma *fls-inverse-of-nat*:
inverse (*of-nat n* :: '*a* :: {*semiring-1*,*inverse*,*uminus*} *fls*) = *fls-const* (*inverse* (*of-nat n*))
by (*simp add*: *fls-inverse-const fls-of-nat*)

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*
using *fls-lr-inverse-const*
by (*auto simp*: *fls-of-int*)

lemma *fls-inverse-of-int*:
inverse (*of-int n* :: '*a* :: {*ring-1*,*inverse*,*uminus*} *fls*) = *fls-const* (*inverse* (*of-int n*))
by (*simp add*: *fls-inverse-const fls-of-int*)

lemma *fls-lr-inverse-zero*:

fixes $x :: 'a::\{ab\text{-}group\text{-}add,mult\text{-}zero\}$
and $y :: 'b::\{comm\text{-}monoid\text{-}add,mult\text{-}zero,uminus\}$
shows *fls-left-inverse 0 x = fls-const x*
and *fls-right-inverse 0 y = fls-const y*
using *fls-lr-inverse-const*[*of 0*]
by *auto*

lemma *fls-inverse-zero-conv-fls-const*:
inverse (*0*::*'a*::{*comm-monoid-add*,*mult-zero*,*uminus*,*inverse*} *fls*) = *fls-const* (*inverse 0*)
using *fls-lr-inverse-zero*(*2*)[*of inverse* (*0*::*'a*)] **by** (*simp add*: *fls-inverse-def'*)

lemma *fls-inverse-zero'*:
assumes *inverse* (*0*::*'a*::{*comm-monoid-add*,*inverse*,*mult-zero*,*uminus*}) = *0*
shows *inverse* (*0*::*'a fls*) = *0*
by (*simp add*: *fls-inverse-zero-conv-fls-const assms*)

lemma *fls-inverse-zero* [*simp*]: *inverse* (*0*::*'a*::*division-ring fls*) = *0*
by (*rule fls-inverse-zero'*[*OF inverse-zero*])

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*)
by (*cases f=0*) (*simp-all add*: *fls-inverse-zero-conv-fls-const fls-inverse-def'*)

lemma *fls-lr-inverse-one*:
fixes $x :: 'a::\{ab\text{-}group\text{-}add,mult\text{-}zero,one\}$
and $y :: 'b::\{comm\text{-}monoid\text{-}add,mult\text{-}zero,uminus,one\}$
shows *fls-left-inverse 1 x = fls-const x*
and *fls-right-inverse 1 y = fls-const y*
using *fls-lr-inverse-const*[*of 1*]
by *auto*

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*)
using *fls-lr-inverse-one*[*of 1*] **by** *auto*

lemma *fls-inverse-one*:
assumes *inverse* (*1*::*'a*::{*comm-monoid-add*,*inverse*,*mult-zero*,*uminus*,*one*}) = *1*
shows *inverse* (*1*::*'a fls*) = *1*
using *assms fls-lr-inverse-one-one*(*2*)
by (*simp add*: *fls-inverse-def'*)

lemma *fls-left-inverse-delta*:
fixes $b :: 'a::\{ab\text{-}group\text{-}add,mult\text{-}zero\}$
assumes $b \neq 0$

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 (*intro fls-eqI*)
fix *n* **from** *assms* **show**
fls-left-inverse (*Abs-fls* (λ*n*. *if n=a then b else 0*)) *x* $$ *n*
= *Abs-fls* (λ*n*. *if n = − a then x else 0*) $$ *n*
using *fls-base-factor-to-fps-delta*[*of a b*]
fls-lr-inverse-const(*1*)[*of b*]
fls-shift-const
by *simp*
qed

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 (*intro fls-eqI*)
fix *n* **from** *assms* **show**
fls-right-inverse (*Abs-fls* (λ*n*. *if n=a then b else 0*)) *x* $$ *n*
= *Abs-fls* (λ*n*. *if n = − a then x else 0*) $$ *n*
using *fls-base-factor-to-fps-delta*[*of a b*]
fls-lr-inverse-const(*2*)[*of b*]
fls-shift-const
by *simp*
qed

lemma *fls-inverse-delta-nonzero*:
fixes *b* :: *'a*::{*comm-monoid-add*,*inverse*,*mult-zero*,*uminus*}
assumes *b* ≠ *0*
shows *inverse* (*Abs-fls* (λ*n*. *if n=a then b else 0*)) =
Abs-fls (λ*n*. *if n=−a then inverse b else 0*)
using *assms fls-nonzeroI*[*of Abs-fls* (λ*n*. *if n=a then b else 0*) *a*]
by (*simp add*: *fls-inverse-def' fls-right-inverse-delta*[*symmetric*])

lemma *fls-inverse-delta*:
fixes *b* :: *'a*::*division-ring*
shows *inverse* (*Abs-fls* (λ*n*. *if n=a then b else 0*)) =
Abs-fls (λ*n*. *if n=−a then inverse b else 0*)
by (*cases b=0*) (*simp-all add*: *fls-inverse-delta-nonzero*)

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*)
using *fls-lr-inverse-one*(*1*)[*of x*] *fls-lr-inverse-one*(*2*)[*of y*]
by *auto*

lemma *fls-lr-inverse-X'*:
fixes $x :: 'a::\{ab\text{-}group\text{-}add,mult\text{-}zero,zero\text{-}neq\text{-}one,monoid\text{-}mult\}$
and $y :: 'b::\{comm\text{-}monoid\text{-}add,uminus,mult\text{-}zero,zero\text{-}neq\text{-}one,monoid\text{-}mult\}$
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*
using *fls-lr-inverse-X(1)[of x] fls-lr-inverse-X(2)[of y]*
by (*simp-all add*: *fls-X-inv-times-conv-shift(2)*)

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*
using *assms fls-lr-inverse-X(2)[of 1::'a]*
by (*simp add*: *fls-inverse-def' fls-X-inv-conv-shift-1*)

lemma *fls-inverse-X*: *inverse (fls-X::'a::division-ring fls) = fls-X-inv*
by (*simp add*: *fls-inverse-X'*)

lemma *fls-lr-inverse-X-inv*:
fixes $x :: 'a::\{ab\text{-}group\text{-}add,mult\text{-}zero,zero\text{-}neq\text{-}one\}$
and $y :: 'b::\{comm\text{-}monoid\text{-}add,uminus,mult\text{-}zero,zero\text{-}neq\text{-}one\}$
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)*
using *fls-lr-inverse-one(1)[of x] fls-lr-inverse-one(2)[of y]*
by *auto*

lemma *fls-lr-inverse-X-inv'*:
fixes $x :: 'a::\{ab\text{-}group\text{-}add,mult\text{-}zero,zero\text{-}neq\text{-}one,monoid\text{-}mult\}$
and $y :: 'b::\{comm\text{-}monoid\text{-}add,uminus,mult\text{-}zero,zero\text{-}neq\text{-}one,monoid\text{-}mult\}$
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*
using *fls-lr-inverse-X-inv(1)[of x] fls-lr-inverse-X-inv(2)[of y]*
by (*simp-all add*: *fls-X-times-conv-shift(2)*)

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*
using *assms fls-lr-inverse-X-inv(2)[of 1::'a]*
by (*simp add*: *fls-inverse-def' fls-X-conv-shift-1*)

lemma *fls-inverse-X-inv*: *inverse (fls-X-inv::'a::division-ring fls) = fls-X*
by (*simp add*: *fls-inverse-X-inv'*)

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*
by (*auto intro*: *fls-subdegree-eqI simp*: *assms*)

lemma *fls-inverse-subdegree'*:

inverse (*f* $$ *fls-subdegree f*) ≠ *0* ⟹ *fls-subdegree* (*inverse f*) = − *fls-subdegree f*
using *fls-lr-inverse-subdegree*(*2*)[*of inverse* (*f* $$ *fls-subdegree f*)]
by (*simp add*: *fls-inverse-def'*)

lemma *fls-inverse-subdegree* [*simp*]:
fixes *f* :: '*a*::*division-ring fls*
shows *fls-subdegree* (*inverse f*) = − *fls-subdegree f*
by (*cases f*=*0*)
(*auto intro*: *fls-inverse-subdegree'* *simp*: *nonzero-imp-inverse-nonzero*)

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*)
using *assms fls-inverse-subdegree'*[*of f*] *fls-inverse-base*[*of f*]
by *simp*

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*)
using *fls-inverse-eq-0-iff'*[*of f*] *fls-inverse-subdegree-base-nonzero*[*of f*]
by (*cases f*=*0* ∨ *inverse* (*f* $$ *fls-subdegree f*) = *0*)
(*auto simp*: *fls-inverse-zero-conv-fls-const*)

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*
using *fls-subdegree-ge0I*[*of fls-left-inverse f x*]
fls-subdegree-ge0I[*of fls-right-inverse f x*]
by (*auto simp*: *assms*)

lemma *fls-inverse-subdegree-0*:
fls-subdegree f = *0* ⟹ *fls-subdegree* (*inverse f*) ≥ *0*
using *fls-lr-inverse-subdegree-0*(*2*)[*of f*] **by** (*simp add*: *fls-inverse-def'*)

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*)
using *assms fls-base-factor-to-fps-shift*[*of m f*] *fls-shift-subdegree*
by *auto*

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*)
using *assms fls-lr-inverse-shift-nonzero*(*2*)[*of f m inverse* (*f* $$ *fls-subdegree f*)]

by (*simp add*: *fls-inverse-def'*)

lemma *fls-inverse-shift*:
fixes *f* :: *'a*::*division-ring fls*
shows *inverse* (*fls-shift m f*) = *fls-shift* (−*m*) (*inverse f*)
using *fls-inverse-shift-nonzero*
by (*cases f*=*0*) *simp-all*

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*)
using *assms fls-lr-inverse-zero*(*1*)[*of x*] *fls-lr-inverse-subdegree*(*1*)[*of x*]
by (*cases f*=*0*) *auto*

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*)
using *assms fls-lr-inverse-zero*(*2*)[*of y*] *fls-lr-inverse-subdegree*(*2*)[*of y*]
by (*cases f*=*0*) *auto*

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*)
by (*cases f*=*0*)
(*simp-all add*:
assms fls-inverse-shift-nonzero fls-inverse-subdegree'
fls-inverse-zero-conv-fls-const
)

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*)
using *fls-base-factor-base*[*of f*] *fls-inverse-eq-0-iff'*[*of f*]
fls-inverse-eq-0-iff'[*of fls-base-factor f*] *fls-inverse-base-factor'*[*of f*]
by (*cases inverse* (*f* $$ *fls-subdegree f*) = *0*) *simp-all*

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*
using *assms*
by *auto*

lemma *fls-inverse-regpart*:
assumes *fls-subdegree f* = *0*

shows *fls-regpart* (*inverse f*) = *inverse* (*fls-regpart f*)
by (*simp add*: *assms fls-inverse-def*)

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*
using *fls-left-inverse-base-factor*[*of x f*] *fls-base-factor-subdegree*[*of f*]
by (*cases x=0*) (*simp-all add*: *fls-lr-inverse-starting0*(*1*) *fps-lr-inverse-starting0*(*1*))

lemma *fls-base-factor-to-fps-right-inverse-nonzero*:
fixes *y* :: *'a*::{*comm-monoid-add*,*mult-zero*,*uminus*}
assumes *y* ≠ *0*
shows *fls-base-factor-to-fps* (*fls-right-inverse f y*) =
fps-right-inverse (*fls-base-factor-to-fps f*) *y*
using *assms fls-right-inverse-base-factor*[*of y f*]
fls-base-factor-subdegree[*of f*]
by *simp*

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*
using *fls-base-factor-to-fps-right-inverse-nonzero*[*of y f*]
by (*cases y=0*) (*simp-all add*: *fls-lr-inverse-starting0*(*2*) *fps-lr-inverse-starting0*(*2*))

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*)
using *assms fls-base-factor-to-fps-right-inverse-nonzero*
by (*simp add*: *fls-inverse-def' fps-inverse-def*)

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*)
using *fls-base-factor-to-fps-right-inverse*
by (*simp add*: *fls-inverse-def' fps-inverse-def*)

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*)
using *assms fls-base-factor-to-fps-to-fls*[*of f*]
by (*simp-all add*: *fls-subdegree-fls-to-fps*)

lemma *fls-inverse-fps-to-fls*:
subdegree f = *0* ⟹ *inverse* (*fps-to-fls f*) = *fps-to-fls* (*inverse f*)
using *nth-subdegree-nonzero*[*of f*]

by (*cases f=0*)
(*auto simp add*:
fps-to-fls-nonzeroI fls-inverse-def' fls-subdegree-fls-to-fps fps-inverse-def
fls-lr-inverse-fps-to-fls(*2*)
)

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*)
using *fls-lr-inverse-one*(*1*)[*of x*] *fls-lr-inverse-one*(*2*)[*of y*]
by (*simp-all add*: *fls-X-power-conv-shift-1*)

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*
using *fls-lr-inverse-X-power*(*1*)[*of n x*] *fls-lr-inverse-X-power*(*2*)[*of n y*]
by (*simp-all add*: *fls-X-inv-power-times-conv-shift*(*2*))

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*
using *fls-lr-inverse-X-power'*(*2*)[*of n 1*]
by (*simp add*: *fls-inverse-def' assms*)

lemma *fls-inverse-X-power*:
inverse ((*fls-X*::*'a*::*division-ring fls*) ^ *n*) = *fls-X-inv* ^ *n*
by (*simp add*: *fls-inverse-X-power'*)

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*)
using *fls-lr-inverse-one*(*1*)[*of x*] *fls-lr-inverse-one*(*2*)[*of y*]
by (*simp-all add*: *fls-X-inv-power-conv-shift-1*)

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*
using *fls-lr-inverse-X-inv-power*(*1*)[*of n x*] *fls-lr-inverse-X-inv-power*(*2*)[*of n y*]
by (*simp-all add*: *fls-X-power-times-conv-shift*(*2*))

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*
using *fls-lr-inverse-X-inv-power'(2)[of n 1]*
by *(simp add: fls-inverse-def' assms)*

lemma *fls-inverse-X-inv-power:*
inverse ((fls-X-inv::'a::division-ring fls) ^ n) = fls-X ^ n
by *(simp add: fls-inverse-X-inv-power')*

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)*
using *fls-lr-inverse-one(1)[of x] fls-lr-inverse-one(2)[of y]*
by *auto*

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)*
using *fls-lr-inverse-X-intpow(1)[of i x] fls-lr-inverse-X-intpow(2)[of i y]*
by *(simp-all add: fls-shifted-times-simps(1))*

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)*
using *fls-lr-inverse-X-intpow'(2)[of i 1]*
by *(simp add: fls-inverse-def' assms)*

lemma *fls-inverse-X-intpow:*
inverse (fls-X-intpow i :: 'a::division-ring fls) = fls-X-intpow (-i)
by *(simp add: fls-inverse-X-intpow')*

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−
from *assms* **have** *x ≠ 0 x * (fls-base-factor-to-fps f$0) = 1* **by** *auto*
thus *?thesis*
using *fls-base-factor-to-fps-left-inverse[of f x]*
fls-lr-inverse-subdegree(1)[of x] fps-left-inverse
by *(fastforce simp: fls-times-def)*
qed

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−
from *assms* **have** *y ≠ 0* (*fls-base-factor-to-fps f$0*) * *y = 1* **by** *auto*
thus *?thesis*
using *fls-base-factor-to-fps-right-inverse*[*of f y*]
fls-lr-inverse-subdegree(*2*)[*of y*] *fps-right-inverse*
by (*fastforce simp*: *fls-times-def*)
qed

— 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 *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*
using *assms*
by (*simp add*: *fps-left-inverse-eq-fps-right-inverse*)

lemma *fls-left-inverse-eq-inverse*:
fixes *f* :: *'a*::*division-ring fls*
shows *fls-left-inverse f* (*inverse* (*f $$ fls-subdegree f*)) = *inverse f*
proof (*cases f=0*)
case *True*
hence *fls-left-inverse f* (*inverse* (*f $$ fls-subdegree f*)) = *fls-const* (*0*::*'a*)
by (*simp add*: *fls-lr-inverse-zero*(*1*)[*symmetric*])
with *True* **show** *?thesis* **by** *simp*
next
case *False* **thus** *?thesis*
using *fls-left-inverse-eq-fls-right-inverse*[*of inverse* (*f $$ fls-subdegree f*)]
by (*auto simp add*: *fls-inverse-def'*)
qed

lemma *fls-right-inverse-eq-inverse*:
fixes *f* :: *'a*::*division-ring fls*
shows *fls-right-inverse f* (*inverse* (*f $$ fls-subdegree f*)) = *inverse f*
proof (*cases f=0*)
case *True*
hence *fls-right-inverse f* (*inverse* (*f $$ fls-subdegree f*)) = *fls-const* (*0*::*'a*)
by (*simp add*: *fls-lr-inverse-zero*(*2*)[*symmetric*])
with *True* **show** *?thesis* **by** *simp*
qed (*simp add*: *fls-inverse-def'*)

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*
using *assms fls-left-inverse-eq-fls-right-inverse*[*of x f x*]

by (*simp add*: *mult.commute*)

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*
using *assms fls-left-inverse-eq-fls-right-inverse*[*of x f y*] *fls-left-inverse*[*of x f*]
by *simp*

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*
using *assms fls-left-inverse-eq-fls-right-inverse*[*of x f y*] *fls-right-inverse*[*of f y*]
by *simp*

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*)
using *assms fls-base-factor-to-fps-base-factor*[*of f*] *fls-base-factor-subdegree*[*of f*]
fls-shifted-times-simps(*2*)[*of* −*fls-subdegree f fls-left-inverse f x f*]
fls-left-inverse[*of x f*]
by *simp*

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*)
using *assms fls-base-factor-to-fps-base-factor*[*of f*] *fls-base-factor-subdegree*[*of f*]
fls-shifted-times-simps(*1*)[*of f* −*fls-subdegree f fls-right-inverse f y*]
fls-right-inverse[*of f y*]
by *simp*

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*)
using *fls-mult-right-inverse-base-factor*[*of f inverse* (*f* $$ *fls-subdegree f*)]
fls-base-factor-base[*of f*]
by (*simp add*: *assms fls-right-inverse-eq-inverse*[*symmetric*])

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−
from *assms*(*1*) **have**
fls-left-inverse (*fls-left-inverse f x*) *y* ∗ *fls-left-inverse f x* ∗ *f* =
fls-left-inverse (*fls-left-inverse f x*) *y*
using *fls-left-inverse*[*of x f*]
by (*simp add*: *mult.assoc*)
moreover have
fls-left-inverse (*fls-left-inverse f x*) *y* ∗ *fls-left-inverse f x* = *1*
using *assms fls-lr-inverse-subdegree*(*1*)[*of x f*] *fls-lr-inverse-base*(*1*)[*of f x*]
by (*fastforce intro*: *fls-left-inverse*)
ultimately show *?thesis* **by** *simp*
qed

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*
using *assms fls-left-inverse-idempotent-ring1*[*of x f f* $$ *fls-subdegree f*]
by (*simp add*: *mult.commute*)

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−
from *assms*(*1*) **have**
f ∗ (*fls-right-inverse f x* ∗ *fls-right-inverse* (*fls-right-inverse f x*) *y*) =
fls-right-inverse (*fls-right-inverse f x*) *y*
using *fls-right-inverse* [*of f*]
by (*simp add*: *mult.assoc*[*symmetric*])
moreover have
fls-right-inverse f x ∗ *fls-right-inverse* (*fls-right-inverse f x*) *y* = *1*
using *assms fls-lr-inverse-subdegree*(*2*)[*of x f*] *fls-lr-inverse-base*(*2*)[*of f x*]
by (*fastforce intro*: *fls-right-inverse*)
ultimately show *?thesis* **by** *simp*
qed

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*
using *assms fls-right-inverse-idempotent-ring1*[*of f x f* $$ *fls-subdegree f*]
by (*simp add*: *mult.commute*)

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−

have *f* $$ *fls-subdegree f* ∗ *g* $$ *fls-subdegree g* ≠ *0*
proof
assume *f* $$ *fls-subdegree f* ∗ *g* $$ *fls-subdegree g* = *0*
hence *f* $$ *fls-subdegree f* ∗ (*g* $$ *fls-subdegree g* ∗ *f* $$ *fls-subdegree f*) = *0*
by (*simp add*: *mult.assoc*[*symmetric*])
with *fg*(*2*) **show** *False* **by** *simp*
qed
with *fg*(*1*) **have** *subdeg-sum*: *fls-subdegree f* + *fls-subdegree g* = *0*
using *fls-mult-nonzero-base-subdegree-eq*[*of f g*] **by** *simp*
hence *subdeg-sum′*:
fls-subdegree f = −*fls-subdegree g fls-subdegree g* = −*fls-subdegree f*
by *auto*

from *fg*(*1*) **have** *f-ne-0*: *f*≠*0* **by** *auto*
moreover have
fps-left-inverse (*fls-base-factor-to-fps g*) (*fls-regpart* (*fls-shift* (−*fls-subdegree g*) *f*)$*0*)
= *fls-regpart* (*fls-shift* (−*fls-subdegree g*) *f*)
proof (*intro fps-lr-inverse-unique-ring1*(*1*))
from *fg*(*1*) **show**
fls-regpart (*fls-shift* (−*fls-subdegree g*) *f*) ∗ *fls-base-factor-to-fps g* = *1*
using *f-ne-0 fls-times-conv-regpart*[*of fls-shift* (−*fls-subdegree g*) *f fls-base-factor g*]
fls-base-factor-subdegree[*of g*]
by (*simp add*: *fls-times-both-shifted-simp subdeg-sum*)
from *fg*(*2*) **show**
fls-base-factor-to-fps g $ *0* ∗ *fls-regpart* (*fls-shift* (−*fls-subdegree g*) *f*) $ *0* = *1*
by (*simp add*: *subdeg-sum′*(*2*))
qed
ultimately show *fls-left-inverse g* (*f* $$ *fls-subdegree f*) = *f*
by (*simp add*: *subdeg-sum′*(*2*))

from *fg*(*1*) **have** *g-ne-0*: *g*≠*0* **by** *auto*
moreover have
fps-right-inverse (*fls-base-factor-to-fps f*) (*fls-regpart* (*fls-shift* (−*fls-subdegree f*) *g*)$*0*)
= *fls-regpart* (*fls-shift* (−*fls-subdegree f*) *g*)
proof (*intro fps-lr-inverse-unique-ring1*(*2*))
from *fg*(*1*) **show**
fls-base-factor-to-fps f ∗ *fls-regpart* (*fls-shift* (−*fls-subdegree f*) *g*) = *1*
using *g-ne-0 fls-times-conv-regpart*[*of fls-base-factor f fls-shift* (−*fls-subdegree f*) *g*]
fls-base-factor-subdegree[*of f*]
by (*simp add*: *fls-times-both-shifted-simp subdeg-sum add.commute*)

from *fg*(*2*) **show**
fls-regpart (*fls-shift* (−*fls-subdegree f*) *g*) $ *0* ∗ *fls-base-factor-to-fps f* $ *0* = *1*
by (*simp add*: *subdeg-sum′*(*1*))
qed
ultimately show *fls-right-inverse f* (*g* $$ *fls-subdegree g*) = *g*
by (*simp add*: *subdeg-sum′*(*2*))

qed

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−
from *fg* **have** *f* ≠*0 g* ≠ *0* **by** *auto*
with *fg* **have** *fls-subdegree f* + *fls-subdegree g* = *0* **using** *fls-subdegree-mult* **by** *force*
with *fg* **have** *f* $$ *fls-subdegree f* ∗ *g* $$ *fls-subdegree g* = *1*
using *fls-times-base*[*of f g*] **by** *simp*
hence *g* $$ *fls-subdegree g* ∗ *f* $$ *fls-subdegree f* = *1*
using *inverse-unique*[*of f* $$ *fls-subdegree f*] *left-inverse*[*of f* $$ *fls-subdegree f*]
by *force*
thus
fls-left-inverse g (*f* $$ *fls-subdegree f*) = *f*
fls-right-inverse f (*g* $$ *fls-subdegree g*) = *g*
using *fg fls-lr-inverse-unique-ring1*
by *auto*
qed

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*
by (*simp-all add*: *fps-lr-inverse-minus*)

lemma *fls-inverse-minus* [*simp*]: *inverse* (−*f*) = −*inverse* (*f* :: *′a* :: *division-ring fls*)
using *fls-lr-inverse-minus*(*2*)[*of f*] **by** (*simp add*: *fls-inverse-def′*)

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−
from *x*(*1*) *y*(*2*) **have** *x* ∗ (*f* $$ *fls-subdegree f* ∗ *g* $$ *fls-subdegree g*) ∗ *y* = *1*

```
  by (simp add: mult.assoc)
hence base-prod: f $$ fls-subdegree f * g $$ fls-subdegree g ≠ 0 by auto
hence subdegrees: fls-subdegree (f*g) = fls-subdegree f + fls-subdegree g
  using fls-mult-nonzero-base-subdegree-eq[of f g] by simp

have norm:
  fls-base-factor-to-fps (f * g) = fls-base-factor-to-fps f * fls-base-factor-to-fps g
  using base-prod fls-base-factor-to-fps-mult'[of f g] by simp

have
  fls-left-inverse (f * g) (y*x) =
    fls-shift (fls-subdegree (f * g)) (
      fps-to-fls (
        fps-left-inverse (fls-base-factor-to-fps f * fls-base-factor-to-fps g) (y*x)
      )
    )

  using norm
  by    simp
thus fls-left-inverse (f * g) (y*x) = fls-left-inverse g y * fls-left-inverse f x
  using x y
        fps-lr-inverse-mult-ring1(1)[of
          x fls-base-factor-to-fps f y fls-base-factor-to-fps g
        ]
  by    (simp add:
          fls-times-both-shifted-simp fls-times-fps-to-fls subdegrees algebra-simps
        )

have
  fls-right-inverse (f * g) (y*x) =
    fls-shift (fls-subdegree (f * g)) (
      fps-to-fls (
        fps-right-inverse (fls-base-factor-to-fps f * fls-base-factor-to-fps g) (y*x)
      )
    )

  using norm
  by    simp
thus fls-right-inverse (f * g) (y*x) = fls-right-inverse g y * fls-right-inverse f x
  using x y
        fps-lr-inverse-mult-ring1(2)[of
          x fls-base-factor-to-fps f y fls-base-factor-to-fps g
        ]
  by    (simp add:
          fls-times-both-shifted-simp fls-times-fps-to-fls subdegrees algebra-simps
        )

qed
```

```
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 ^ n) (x ^ n) = (fls-left-inverse f x) ^ n
          fls-right-inverse (f ^ n) (x ^ n) = (fls-right-inverse f x) ^ n
proof-

  show fls-left-inverse (f ^ n) (x ^ n) = (fls-left-inverse f x) ^ n
  proof (induct n)
    case 0 show ?case using fls-lr-inverse-one(1)[of 1] by simp
  next
    case (Suc n) with assms show ?case
      using fls-lr-inverse-mult-ring1(1)[of x f x^n f^n]
      by    (simp add:
          power-Suc2[symmetric] fls-unit-base-subdegree-power(1) left-right-inverse-power
            )
  qed

  show fls-right-inverse (f ^ n) (x ^ n) = (fls-right-inverse f x) ^ n
  proof (induct n)
    case 0 show ?case using fls-lr-inverse-one(2)[of 1] by simp
  next
    case (Suc n) with assms show ?case
      using fls-lr-inverse-mult-ring1(2)[of x f x^n f^n]
      by    (simp add:
          power-Suc2[symmetric] fls-unit-base-subdegree-power(1) left-right-inverse-power
            )
  qed

qed

lemma fls-divide-convert-times-inverse:
  fixes   f g :: 'a::{comm-monoid-add,inverse,mult-zero,uminus} fls
  shows   f / g = f * inverse g
  using fls-base-factor-to-fps-subdegree[of g] fps-to-fls-base-factor-to-fps[of f]
        fls-times-both-shifted-simp[of -fls-subdegree f fls-base-factor f]
  by    (simp add:
          fls-divide-def fps-divide-unit' fls-times-fps-to-fls
          fls-conv-base-factor-shift-subdegree fls-inverse-def
        )

instance fls :: (division-ring) division-ring
proof
  fix a b :: 'a fls
  show a ≠ 0 ⟹ inverse a * a = 1
    using fls-left-inverse'[of inverse (a $$ fls-subdegree a) a]
    by    (simp add: fls-inverse-def')
  show a ≠ 0 ⟹ a * inverse a = 1
    using fls-right-inverse[of a]
```

```
    by    (simp add: fls-inverse-def')
  show a / b = a * inverse b using fls-divide-convert-times-inverse by fast
  show inverse (0::'a fls) = 0 by simp
qed

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 -
  show
    fls-left-inverse (f*g) (inverse ((f*g)$$(df+dg))) =
      fls-left-inverse g (inverse (g$$dg)) * fls-left-inverse f (inverse (f$$df))
  proof (cases f=0 ∨ g=0)
    case True thus ?thesis
      using fls-lr-inverse-zero(1)[of inverse (0::'a)] by (auto simp add: assms)
  next
    case False thus ?thesis
      using fls-left-inverse-eq-inverse[of f*g] nonzero-inverse-mult-distrib[of f g]
            fls-left-inverse-eq-inverse[of g] fls-left-inverse-eq-inverse[of f]
      by    (simp add: assms)
  qed
  show
    fls-right-inverse (f*g) (inverse ((f*g)$$(df+dg))) =
      fls-right-inverse g (inverse (g$$dg)) * fls-right-inverse f (inverse (f$$df))
  proof (cases f=0 ∨ g=0)
    case True thus ?thesis
      using fls-lr-inverse-zero(2)[of inverse (0::'a)] by (auto simp add: assms)
  next
    case False thus ?thesis
      using fls-inverse-def'[of f*g] nonzero-inverse-mult-distrib[of f g]
            fls-inverse-def'[of g] fls-inverse-def'[of f]
      by    (simp add: assms)
  qed
qed

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 -
  note fls-left-inverse-eq-inverse [of f] fls-right-inverse-eq-inverse[of f]
```

```
  moreover have
    fls-right-inverse (f ^ n) ((inverse (f $$ fls-subdegree f)) ^ n) =
      inverse f ^ n
    using fls-right-inverse-eq-inverse [of f ^ n]
    by (simp add: fls-subdegree-pow power-inverse)
  moreover have
    fls-left-inverse (f ^ n) ((inverse (f $$ fls-subdegree f)) ^ n) =
      inverse f ^ n
    using fls-left-inverse-eq-inverse [of f ^ n]
    by (simp add: fls-subdegree-pow power-inverse)
  ultimately show ?P and ?Q
    by simp-all
qed

lemma one-plus-fls-X-powi-eq:
  (1 + fls-X) powi n = fps-to-fls (fps-binomial (of-int n :: 'a :: field-char-0))
proof (cases n ≥ 0)
  case True
  thus ?thesis
    using fps-binomial-of-nat[of nat n, where ?'a = 'a]
    by (simp add: power-int-def fps-to-fls-power)
next
  case False
  thus ?thesis
    using fps-binomial-minus-of-nat[of nat (−n), where ?'a = 'a]
   by (simp add: power-int-def fps-to-fls-power fps-inverse-power flip: fls-inverse-fps-to-fls)
qed

instance fls :: (field) field
  by (standard, simp-all add: field-simps)

instance fls :: ({field-prime-char,comm-semiring-1}) field-prime-char
  by (rule field-prime-charI') auto

instance fls :: (semiring-char-0) semiring-char-0
proof
  show inj (of-nat :: nat ⇒ 'a fls)
    by (metis fls-regpart-of-nat injI of-nat-eq-iff)
qed

instance fls :: (field-char-0) field-char-0 ..


lemma fls-subdegree-power-int [simp]:
  fixes   F :: 'a :: field fls
  shows fls-subdegree (F powi n) = n * fls-subdegree F
  by (auto simp: power-int-def fls-subdegree-pow)
```

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*
by (*simp add*: *fls-divide-def*)

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*
using *fps-divide-nth-0'*[*of fls-base-factor-to-fps g fls-base-factor-to-fps f*]
fls-base-factor-to-fps-subdegree[*of g*]
by (*simp add*: *fls-divide-def*)

lemma *fls-div-zero* [*simp*]:
0 div (*g* :: *'a* :: {*comm-monoid-add*,*inverse*,*mult-zero*,*uminus*} *fls*) = *0*
by (*simp add*: *fls-divide-def*)

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*
by (*simp add*: *fls-divide-def assms fps-div-by-zero'*)

lemma *fls-divide-times*:
fixes *f g* :: *'a*::{*semiring-0*,*inverse*,*uminus*} *fls*
shows (*f* ∗ *g*) / *h* = *f* ∗ (*g* / *h*)
by (*simp add*: *fls-divide-convert-times-inverse mult.assoc*)

lemma *fls-divide-times2*:
fixes *f g* :: *'a*::{*comm-semiring-0*,*inverse*,*uminus*} *fls*
shows (*f* ∗ *g*) / *h* = (*f* / *h*) ∗ *g*
using *fls-divide-times*[*of g f h*]
by (*simp add*: *mult.commute*)

lemma *fls-divide-subdegree-ge*:
fixes *f g* :: *'a*::{*comm-monoid-add*,*uminus*,*times*,*inverse*} *fls*
assumes *f* / *g* ≠ *0*
shows *fls-subdegree* (*f* / *g*) ≥ *fls-subdegree f* − *fls-subdegree g*
using *assms fls-divide-nth-below*
by (*intro fls-subdegree-geI*) *simp*

lemma *fls-divide-subdegree*:
fixes *f g* :: *'a*::*division-ring fls*
assumes *f* ≠ *0 g* ≠ *0*
shows *fls-subdegree* (*f* / *g*) = *fls-subdegree f* − *fls-subdegree g*
proof (*intro antisym*)
from *assms* **have** *f* $$ *fls-subdegree f* / *g* $$ *fls-subdegree g* ≠ *0* **by** (*simp add*:

field-simps)
  thus *fls-subdegree* $(f/g) \leq$ *fls-subdegree* $f -$ *fls-subdegree* g
    using *fls-divide-nth-base*[*of f g*] **by** (*intro fls-subdegree-leI*) *simp*
  from *assms* **have** f / $g \neq 0$ **by** (*simp add*: *field-simps*)
  thus *fls-subdegree* $(f/g) \geq$ *fls-subdegree* $f -$ *fls-subdegree* g
    using *fls-divide-subdegree-ge* **by** *fast*
qed

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* $m\ (f/g)$
  using *assms fls-base-factor-to-fps-shift*[*of m f*]
  by (*simp add*: *fls-divide-def algebra-simps*)

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* $m\ (f/g)$
  using *fls-divide-shift-numer-nonzero*
  by (*cases f=0*) *auto*

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* $(-m)\ (f/g)$
  using *assms fls-base-factor-to-fps-shift*[*of m g*]
  by (*simp add*: *fls-divide-def algebra-simps*)

lemma *fls-divide-shift-denom*:
  fixes $f\ g :: 'a ::$ *division-ring fls*
  shows f / *fls-shift* $m\ g =$ *fls-shift* $(-m)\ (f/g)$
  using *fls-divide-shift-denom-nonzero*
  by (*cases g=0*) *auto*

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* $(n-m)\ (f/g)$
  by (*simp add*: *assms fls-divide-shift-numer-nonzero fls-divide-shift-denom-nonzero*)

lemma *fls-divide-shift-both* [*simp*]:
  fixes $f\ g :: 'a ::$ *division-ring fls*
  shows *fls-shift* $n\ f$ / *fls-shift* $m\ g =$ *fls-shift* $(n-m)\ (f/g)$
  using *fls-divide-shift-both-nonzero*
  by (*cases f=0* $\vee$ *g=0*) *auto*

lemma *fls-divide-base-factor-numer*:
  fls-base-factor f / $g =$ *fls-shift* (*fls-subdegree* f) (f/g)
  using *fls-base-factor-to-fps-base-factor*[*of f*]

fls-base-factor-subdegree[*of f*]
by (*simp add*: *fls-divide-def algebra-simps*)

lemma *fls-divide-base-factor-denom*:
f / *fls-base-factor g* = *fls-shift* (−*fls-subdegree g*) (*f*/*g*)
using *fls-base-factor-to-fps-base-factor*[*of g*]
fls-base-factor-subdegree[*of g*]
by (*simp add*: *fls-divide-def*)

lemma *fls-divide-base-factor'*:
fls-base-factor f / *fls-base-factor g* = *fls-shift* (*fls-subdegree f* − *fls-subdegree g*)
(*f*/*g*)
using *fls-divide-base-factor-numer*[*of f fls-base-factor g*]
fls-divide-base-factor-denom[*of f g*]
by *simp*

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*)
using *fls-divide-subdegree*[*of f g*] *fls-divide-base-factor'*
by *fastforce*

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 −
have *deg0*:
⋀*g*. *fls-subdegree g* = *0* ⟹
fls-regpart (*f* / *g*) = *fls-regpart f* / *fls-regpart g*
by (*simp add*:
assms(*1*) *fls-divide-convert-times-inverse fls-inverse-subdegree-0*
fls-times-conv-regpart fls-inverse-regpart fls-regpart-subdegree-conv fps-divide-unit'
)
show *?thesis*
proof (*cases fls-subdegree g* = *0*)
case *False*
hence *fls-base-factor g* ≠ *0* **using** *fls-base-factor-nonzero*[*of g*] **by** *force*
with *assms*(*2*) **show** *?thesis*
using *fls-divide-shift-denom-nonzero*[*of fls-base-factor g f* −*fls-subdegree g*]
fps-shift-fls-regpart-conv-fls-shift[*of*
nat (*fls-subdegree g*) *f* / *fls-base-factor g*
]
fls-base-factor-subdegree[*of g*] *deg0*
fls-regpart-subdegree-conv[*of g*] *fps-unit-factor-fls-regpart*[*of g*]
by (*simp add*:
fls-conv-base-factor-shift-subdegree fls-regpart-subdegree-conv fps-divide-def
)
qed (*rule deg0*)

qed

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*))
using *fls-base-factor-subdegree*[*of f*] *fls-base-factor-subdegree*[*of g*]
fls-divide-regpart[*of fls-base-factor f fls-base-factor g*]
fls-divide-base-factor'[*of f g*]
by *simp*

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*)
using *fls-divide-fls-base-factor-to-fps' fls-divide-subdegree*[*of f g*]
by *fastforce*

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−
have *1*:
fps-to-fls f / fps-to-fls g =
fls-shift (*int* (*subdegree g*)) (*fps-to-fls* (*f* ∗ *inverse* (*unit-factor g*)))
using *fls-base-factor-to-fps-to-fls*[*of f*] *fls-base-factor-to-fps-to-fls*[*of g*]
fls-subdegree-fls-to-fps[*of f*] *fls-subdegree-fls-to-fps*[*of g*]
fps-divide-def[*of unit-factor f unit-factor g*]
fls-times-fps-to-fls[*of unit-factor f inverse* (*unit-factor g*)]
fls-shifted-times-simps(*2*)[*of* −*int* (*subdegree f*) *fps-to-fls* (*unit-factor f*)]
fls-times-fps-to-fls[*of f inverse* (*unit-factor g*)]
by (*simp add*: *fls-divide-def*)
with *assms* **show** *?thesis*
using *fps-mult-subdegree-ge*[*of f inverse* (*unit-factor g*)]
fps-shift-to-fls[*of subdegree g f* ∗ *inverse* (*unit-factor g*)]
by (*cases f* ∗ *inverse* (*unit-factor g*) = *0*) (*simp-all add*: *fps-divide-def*)
qed

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*
using *assms fls-conv-base-factor-to-fps-shift-subdegree*[*of f*]
by (*simp add*: *fls-divide-def fps-divide-1'*)

lemma *fls-divide-1* [*simp*]: *a / 1* = (*a*::*'a*::*division-ring fls*)

by (*rule fls-divide-1'*[*OF inverse-1*])

lemma *fls-const-divide-const*:
fixes *x y* :: *'a*::*division-ring*
shows *fls-const x / fls-const y* = *fls-const* (*x*/*y*)
by (*simp add*: *fls-divide-def fls-base-factor-to-fps-const fps-const-divide*)

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−
from *assms* **have**
f / fls-X =
fls-shift 1 (*fls-shift* (−*fls-subdegree f*) (*fps-to-fls* (*fls-base-factor-to-fps f*)))
by (*simp add*: *fls-divide-def fps-divide-1'*)
also have ... = *fls-shift 1 f*
using *fls-conv-base-factor-to-fps-shift-subdegree*[*of f*]
by *simp*
finally show *?thesis* **by** *simp*
qed

lemma *fls-divide-X* [*simp*]:
fixes *f* :: *'a*::*division-ring fls*
shows *f / fls-X* = *fls-shift 1 f*
by (*rule fls-divide-X'*[*OF inverse-1*])

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−
have *fls-base-factor-to-fps* ((*fls-X*::*'a fls*) ^ *n*) = *1* **by** (*rule fls-X-power-base-factor-to-fps*)
with *assms* **have**
f / (fls-X ^ *n)* =
fls-shift n (*fls-shift* (−*fls-subdegree f*) (*fps-to-fls* (*fls-base-factor-to-fps f*)))
by (*simp add*: *fls-divide-def fps-divide-1'*)
also have ... = *fls-shift n f*
using *fls-conv-base-factor-to-fps-shift-subdegree*[*of f*] **by** *simp*
finally show *?thesis* **by** *simp*
qed

lemma *fls-divide-X-power* [*simp*]:
fixes *f* :: *'a*::*division-ring fls*
shows *f / (fls-X* ^ *n)* = *fls-shift n f*
by (*rule fls-divide-X-power'*[*OF inverse-1*])

lemma *fls-divide-X-inv'*:

fixes $f :: 'a::\{comm\text{-}monoid\text{-}add,inverse,mult\text{-}zero,uminus,zero\text{-}neq\text{-}one,monoid\text{-}mult\}$ *fls*
assumes *inverse* $(1::'a) = 1$
shows $f \mathbin{/} fls\text{-}X\text{-}inv = fls\text{-}shift\ (-1)\ f$
proof−
from *assms* **have**
$f \mathbin{/} fls\text{-}X\text{-}inv =$
$fls\text{-}shift\ (-1)\ (fls\text{-}shift\ (-fls\text{-}subdegree\ f)\ (fps\text{-}to\text{-}fls\ (fls\text{-}base\text{-}factor\text{-}to\text{-}fps\ f)))$
by (*simp add*: *fls-divide-def fps-divide-1'* *algebra-simps*)
also have $\ldots = fls\text{-}shift\ (-1)\ f$
using *fls-conv-base-factor-to-fps-shift-subdegree*[*of f*]
by *simp*
finally show *?thesis* **by** *simp*
qed

lemma *fls-divide-X-inv* [*simp*]:
fixes $f :: 'a::division\text{-}ring\ fls$
shows $f \mathbin{/} fls\text{-}X\text{-}inv = fls\text{-}shift\ (-1)\ f$
by (*rule fls-divide-X-inv'*[*OF inverse-1*])

lemma *fls-divide-X-inv-power'*:
fixes $f :: 'a::\{semiring\text{-}1,inverse,uminus\}\ fls$
assumes *inverse* $(1::'a) = 1$
shows $f \mathbin{/} (fls\text{-}X\text{-}inv \mathbin{\hat{}} n) = fls\text{-}shift\ (-int\ n)\ f$
proof−
have *fls-base-factor-to-fps* $((fls\text{-}X\text{-}inv::'a\ fls) \mathbin{\hat{}} n) = 1$
by (*rule fls-X-inv-power-base-factor-to-fps*)
with *assms* **have**
$f \mathbin{/} (fls\text{-}X\text{-}inv \mathbin{\hat{}} n) =$
$fls\text{-}shift\ (-int\ n + -fls\text{-}subdegree\ f)\ (fps\text{-}to\text{-}fls\ (fls\text{-}base\text{-}factor\text{-}to\text{-}fps\ f))$
by (*simp add*: *fls-divide-def fps-divide-1'*)
also have
$\ldots = fls\text{-}shift\ (-int\ n)\ (fls\text{-}shift\ (-fls\text{-}subdegree\ f)\ (fps\text{-}to\text{-}fls\ (fls\text{-}base\text{-}factor\text{-}to\text{-}fps\ f)))$
by (*simp add*: *add.commute*)
also have $\ldots = fls\text{-}shift\ (-int\ n)\ f$
using *fls-conv-base-factor-to-fps-shift-subdegree*[*of f*] **by** *simp*
finally show *?thesis* **by** *simp*
qed

lemma *fls-divide-X-inv-power* [*simp*]:
fixes $f :: 'a::division\text{-}ring\ fls$
shows $f \mathbin{/} (fls\text{-}X\text{-}inv \mathbin{\hat{}} n) = fls\text{-}shift\ (-int\ n)\ f$
by (*rule fls-divide-X-inv-power'*[*OF inverse-1*])

lemma *fls-divide-X-intpow'*:
fixes $f :: 'a::\{semiring\text{-}1,inverse,uminus\}\ fls$
assumes *inverse* $(1::'a) = 1$
shows $f \mathbin{/} (fls\text{-}X\text{-}intpow\ i) = fls\text{-}shift\ i\ f$

using *assms*
by (*simp add*: *fls-divide-shift-denom-nonzero fls-divide-1'*)

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*)
using *assms fls-X-intpow-times-conv-shift*(*2*)[*of f* −*i*]
by (*simp add*: *fls-divide-X-intpow'*)

lemma *fls-divide-X-intpow*:
fixes *f* :: *'a*::*division-ring fls*
shows *f* / (*fls-X-intpow i*) = *fls-shift i f*
by (*rule fls-divide-X-intpow'*[*OF inverse-1*])

lemma *fls-divide-X-intpow-conv-times*:
fixes *f* :: *'a*::*division-ring fls*
shows *f* / (*fls-X-intpow i*) = *f* ∗ *fls-X-intpow* (−*i*)
by (*rule fls-divide-X-intpow-conv-times'*[*OF inverse-1*])

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*)
by (*simp add*: *assms fls-divide-shift-both-nonzero fls-divide-1'*)

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*)
by (*rule fls-X-intpow-div-fls-X-intpow-semiring1*[*OF inverse-1*])

lemma *fls-divide-add*:
fixes *f g h* :: *'a*::{*semiring-0*,*inverse*,*uminus*} *fls*
shows (*f* + *g*) / *h* = *f* / *h* + *g* / *h*
by (*simp add*: *fls-divide-convert-times-inverse algebra-simps*)

lemma *fls-divide-diff*:
fixes *f g h* :: *'a*::{*ring*,*inverse*} *fls*
shows (*f* − *g*) / *h* = *f* / *h* − *g* / *h*
by (*simp add*: *fls-divide-convert-times-inverse algebra-simps*)

lemma *fls-divide-uminus*:
fixes *f g h* :: *'a*::{*ring*,*inverse*} *fls*
shows (− *f*) / *g* = − (*f* / *g*)
by (*simp add*: *fls-divide-convert-times-inverse*)

lemma *fls-divide-uminus'*:
fixes *f g h* :: *'a*::*division-ring fls*
shows *f* / (− *g*) = − (*f* / *g*)
by (*simp add*: *fls-divide-convert-times-inverse*)

7.5.7 Units

lemma *fls-is-left-unit-iff-base-is-left-unit*:
fixes $f :: 'a :: ring\text{-}1\text{-}no\text{-}zero\text{-}divisors\ fls$
shows $(\exists g.\ 1 = f * g) \longleftrightarrow (\exists k.\ 1 = f\ \$\$\ fls\text{-}subdegree\ f * k)$
proof
assume $\exists g.\ 1 = f * g$
then obtain g **where** $1 = f * g$ **by** *fast*
hence $1 = (f\ \$\$\ fls\text{-}subdegree\ f) * (g\ \$\$\ fls\text{-}subdegree\ g)$
using *fls-subdegree-mult*[*of f g*] *fls-times-base*[*of f g*] **by** *fastforce*
thus $\exists k.\ 1 = f\ \$\$\ fls\text{-}subdegree\ f * k$ **by** *fast*
next
assume $\exists k.\ 1 = f\ \$\$\ fls\text{-}subdegree\ f * k$
then obtain k **where** $1 = f\ \$\$\ fls\text{-}subdegree\ f * k$ **by** *fast*
hence $1 = f * fls\text{-}right\text{-}inverse\ f\ k$
using *fls-right-inverse* **by** *simp*
thus $\exists g.\ 1 = f * g$ **by** *fast*
qed

lemma *fls-is-right-unit-iff-base-is-right-unit*:
fixes $f :: 'a :: ring\text{-}1\text{-}no\text{-}zero\text{-}divisors\ fls$
shows $(\exists g.\ 1 = g * f) \longleftrightarrow (\exists k.\ 1 = k * f\ \$\$\ fls\text{-}subdegree\ f)$
proof
assume $\exists g.\ 1 = g * f$
then obtain g **where** $1 = g * f$ **by** *fast*
hence $1 = (g\ \$\$\ fls\text{-}subdegree\ g) * (f\ \$\$\ fls\text{-}subdegree\ f)$
using *fls-subdegree-mult*[*of g f*] *fls-times-base*[*of g f*] **by** *fastforce*
thus $\exists k.\ 1 = k * f\ \$\$\ fls\text{-}subdegree\ f$ **by** *fast*
next
assume $\exists k.\ 1 = k * f\ \$\$\ fls\text{-}subdegree\ f$
then obtain k **where** $1 = k * f\ \$\$\ fls\text{-}subdegree\ f$ **by** *fast*
hence $1 = fls\text{-}left\text{-}inverse\ f\ k * f$
using *fls-left-inverse* **by** *simp*
thus $\exists g.\ 1 = g * f$ **by** *fast*
qed

7.6 Composition

definition $fls\text{-}compose\text{-}fps :: 'a :: field\ fls \Rightarrow 'a\ fps \Rightarrow 'a\ fls$ **where**
$fls\text{-}compose\text{-}fps\ F\ G =$
$fps\text{-}to\text{-}fls\ (fps\text{-}compose\ (fls\text{-}base\text{-}factor\text{-}to\text{-}fps\ F)\ G) * fps\text{-}to\text{-}fls\ G\ powi\ fls\text{-}subdegree\ F$

lemma *fps-compose-of-nat* [*simp*]: $fps\text{-}compose\ (of\text{-}nat\ n :: 'a :: comm\text{-}ring\text{-}1\ fps)\ H = of\text{-}nat\ n$
and *fps-compose-of-int* [*simp*]: $fps\text{-}compose\ (of\text{-}int\ i)\ H = of\text{-}int\ i$
unfolding *fps-of-nat* [*symmetric*] *fps-of-int* [*symmetric*] *numeral-fps-const*
by (*rule fps-const-compose*)+

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*
by (*simp-all add*: *fls-compose-fps-def*)

lemma *fls-compose-fps-0-right*:
fls-compose-fps F 0 = (*if 0* ≤ *fls-subdegree F then fls-const* (*F $$ 0*) *else 0*)
by (*cases fls-subdegree F* = *0*) (*simp-all add*: *fls-compose-fps-def*)

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 (*cases F* = *0*)
case *False*
thus *?thesis*
using *assms* **by** (*simp add*: *fls-compose-fps-def power-int-diff power-int-minus field-simps*)
qed *auto*

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 (*cases F* = *0*)
case *False*
define *n* **where** *n* = *subdegree F*
define *F′* **where** *F′* = *fps-shift n F*
have [*simp*]: *F′* ≠ *0 subdegree F′* = *0*
using *False* **by** (*auto simp*: *F′-def n-def*)
have *F-eq*: *F* = *F′* ∗ *fps-X* ^ *n*
unfolding *F′-def n-def* **using** *subdegree-decompose* **by** *blast*
have *fls-compose-fps* (*fps-to-fls F*) *G* =
fps-to-fls (*fps-shift n* (*fls-regpart* (*fps-to-fls F′* ∗ *fls-X-intpow* (*int n*))) *oo G*) ∗ *fps-to-fls* (*G* ^ *n*)
unfolding *F-eq fls-compose-fps-def*
by (*simp add*: *fls-times-fps-to-fls fls-X-power-conv-shift-1 power-int-add fls-subdegree-fls-to-fps fps-to-fls-power fls-regpart-shift-conv-fps-shift flip*: *fls-times-both-shifted-simp*)
also have *fps-to-fls F′* ∗ *fls-X-intpow* (*int n*) = *fps-to-fls F*
by (*simp add*: *F-eq fls-times-fps-to-fls fps-to-fls-power fls-X-power-conv-shift-1*)
also have *fps-to-fls* (*fps-shift n* (*fls-regpart* (*fps-to-fls F*)) *oo G*) ∗ *fps-to-fls* (*G* ^ *n*) =
fps-to-fls ((*fps-shift n* (*fls-regpart* (*fps-to-fls F*)) ∗ *fps-X* ^ *n*) *oo G*)
by (*simp add*: *fls-times-fps-to-fls flip*: *fps-compose-power add*: *fps-compose-mult-distrib*)
also have *fps-shift n* (*fls-regpart* (*fps-to-fls F*)) ∗ *fps-X* ^ *n* = *F*

by (*simp add*: *F-eq*)
finally show *?thesis* **.**
qed (*auto simp*: *fls-compose-fps-def*)

lemma *fls-compose-fps-mult*:
assumes [*simp*]: $H \neq 0$ *fps-nth* $H\ 0 = 0$
shows *fls-compose-fps* $(F * G)\ H =$ *fls-compose-fps* $F\ H *$ *fls-compose-fps* $G\ H$
using *assms*
proof (*cases* $F * G = 0$)
case *False*
hence [*simp*]: $F \neq 0\ G \neq 0$
by *auto*
define $n\ m$ **where** $n =$ *fls-subdegree* $F\ m =$ *fls-subdegree* G
define F' **where** $F' =$ *fls-regpart* (*fls-shift* $n\ F$)
define G' **where** $G' =$ *fls-regpart* (*fls-shift* $m\ G$)
have *F-eq*: $F =$ *fls-shift* $(-n)$ (*fps-to-fls* F') **and** *G-eq*: $G =$ *fls-shift* $(-m)$ (*fps-to-fls* G')
by (*simp-all add*: F'*-def* G'*-def n-m-def*)
have *fls-compose-fps* $(F * G)\ H =$ *fls-compose-fps* (*fls-shift* $(-(n + m))$ (*fps-to-fls* $(F' * G')$)) H
by (*simp add*: *fls-times-fps-to-fls F-eq G-eq fls-shifted-times-simps*)
also have $\ldots =$ *fps-to-fls* $((F'$ *oo* $H) * (G'$ *oo* $H)) *$ *fps-to-fls* H *powi* $(m + n)$
by (*simp add*: *fls-compose-fps-shift fps-compose-mult-distrib*)
also have $\ldots =$ *fls-compose-fps* $F\ H *$ *fls-compose-fps* $G\ H$
by (*simp add*: *F-eq G-eq fls-compose-fps-shift fls-times-fps-to-fls power-int-add*)
finally show *?thesis* **.**
qed *auto*

lemma *fls-compose-fps-power*:
assumes [*simp*]: $G \neq 0$ *fps-nth* $G\ 0 = 0$
shows *fls-compose-fps* $(F \hat{}\ n)\ G =$ *fls-compose-fps* $F\ G \hat{}\ n$
by (*induction* n) (*auto simp*: *fls-compose-fps-mult*)

lemma *fls-compose-fps-add*:
assumes [*simp*]: $H \neq 0$ *fps-nth* $H\ 0 = 0$
shows *fls-compose-fps* $(F + G)\ H =$ *fls-compose-fps* $F\ H +$ *fls-compose-fps* $G\ H$
proof (*cases* $F = 0 \vee G = 0$)
case *False*
hence [*simp*]: $F \neq 0\ G \neq 0$
by *auto*
define n **where** $n =$ *min* (*fls-subdegree* F) (*fls-subdegree* G)
define F' **where** $F' =$ *fls-regpart* (*fls-shift* $n\ F$)
define G' **where** $G' =$ *fls-regpart* (*fls-shift* $n\ G$)
have *F-eq*: $F =$ *fls-shift* $(-n)$ (*fps-to-fls* F') **and** *G-eq*: $G =$ *fls-shift* $(-n)$ (*fps-to-fls* G')
unfolding *n-def* **by** (*simp-all add*: F'*-def* G'*-def n-def*)
have $F + G =$ *fls-shift* $(-n)$ (*fps-to-fls* $(F' + G')$)

by (*simp add*: *F-eq G-eq*)
also have *fls-compose-fps* ... *H* = *fls-compose-fps* (*fps-to-fls* (*F'* + *G'*)) *H* ∗ *fps-to-fls H powi n*
by (*subst fls-compose-fps-shift*) *auto*
also have ... = *fps-to-fls* (*fps-compose* (*F'* + *G'*) *H*) ∗ *fps-to-fls H powi n*
by (*subst fls-compose-fps-to-fls*) *auto*
also have ... = *fls-compose-fps F H* + *fls-compose-fps G H*
by (*simp add*: *F-eq G-eq fls-compose-fps-shift fps-compose-add-distrib algebra-simps*)
finally show *?thesis* **.**
qed *auto*

lemma *fls-compose-fps-uminus* [*simp*]: *fls-compose-fps* (−*F*) *H* = −*fls-compose-fps F H*
by (*simp add*: *fls-compose-fps-def fps-compose-uminus*)

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*
using *fls-compose-fps-add*[*of H F* −*G*] **by** *simp*

lemma *fls-compose-fps-eq-0-iff*:
assumes *H* ≠ *0 fps-nth H 0* = *0*
shows *fls-compose-fps F H* = *0* ⟷ *F* = *0*
using *assms fls-base-factor-to-fps-nonzero*[*of F*]
by (*cases F* = *0*) (*auto simp*: *fls-compose-fps-def fps-compose-eq-0-iff*)

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 (*cases F* = *0*)
case *False*
have *fls-compose-fps* (*inverse F*) *H* ∗ *fls-compose-fps F H* =
fls-compose-fps (*inverse F* ∗ *F*) *H*
by (*subst fls-compose-fps-mult*) *auto*
also have *inverse F* ∗ *F* = *1*
using *False* **by** *simp*
finally show *?thesis*
using *False* **by** (*simp add*: *field-simps fls-compose-fps-eq-0-iff*)
qed *auto*

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*
using *fls-compose-fps-mult*[*of H F inverse G*] *fls-compose-fps-inverse*[*of H G*]
by (*simp add*: *field-simps*)

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*
by (*simp add*: *power-int-def fls-compose-fps-power fls-compose-fps-inverse*)

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 (*cases F* = *0*)
case [*simp*]: *False*
define *n* **where** *n* = *fls-subdegree F*
define *F'* **where** *F'* = *fls-regpart* (*fls-shift n F*)
have *F-eq*: *F* = *fls-shift* (−*n*) (*fps-to-fls F'*)
by (*simp add*: *F'-def n-def*)
show *?thesis*
by (*simp add*: *F-eq fls-compose-fps-shift fls-compose-fps-mult fls-compose-fps-powi fps-compose-eq-0-iff fps-compose-assoc*)
qed *auto*

lemma *subdegree-pos-iff*: *subdegree F* > *0* ⟷ *F* ≠ *0* ∧ *fps-nth F 0* = *0*
using *subdegree-eq-0-iff*[*of F*] **by** *auto*

lemma *fls-X-power-int* [*simp*]: *fls-X powi n* = (*fls-X-intpow n* :: *'a* :: *division-ring fls*)
by (*auto simp*: *power-int-def fls-X-power-conv-shift-1 fls-inverse-X fls-inverse-shift simp flip*: *fls-inverse-X-power*)

lemma *fls-const-power-int*: *fls-const* (*c powi n*) = *fls-const* (*c* :: *'a* :: *division-ring*) *powi n*
by (*auto simp*: *power-int-def fls-const-power fls-inverse-const*)

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 −
{
assume ∗: *n* ≥ *fls-subdegree F*
hence *c* ^ *nat* (*n* − *fls-subdegree F*) = *c powi int* (*nat* (*n* − *fls-subdegree F*))
by (*simp add*: *power-int-def*)
also have ... ∗ *c powi fls-subdegree F* = *c powi* (*int* (*nat* (*n* − *fls-subdegree F*)) + *fls-subdegree F*)
using ∗ **by** (*subst power-int-add*) *auto*
also have ... = *c powi n*
using ∗ **by** *simp*
finally have *c* ^ *nat* (*n* − *fls-subdegree F*) ∗ *c powi fls-subdegree F* = *c powi n*
.
}

thus *?thesis*
by (*simp add*: *fls-compose-fps-def fps-compose-linear fls-times-fps-to-fls power-int-mult-distrib fls-shifted-times-simps*
flip: *fls-const-power-int*)
qed

lemma *fls-const-transfer* [*transfer-rule*]:
rel-fun (=) (*pcr-fls* (=))
(λ*c n*. *if n* = *0 then c else 0*) *fls-const*
by (*auto simp*: *fls-const-def rel-fun-def pcr-fls-def OO-def cr-fls-def*)

lemma *fls-shift-transfer* [*transfer-rule*]:
rel-fun (=) (*rel-fun* (*pcr-fls* (=)) (*pcr-fls* (=)))
(λ*n f k*. *f* (*k*+*n*)) *fls-shift*
by (*auto simp*: *fls-const-def rel-fun-def pcr-fls-def OO-def cr-fls-def*)

lift-definition *fls-compose-power* :: *'a* :: *zero fls* ⇒ *nat* ⇒ *'a fls* **is**
λ*f d n*. *if d* > *0* ∧ *int d dvd n then f* (*n div int d*) *else 0*
proof −
fix *f* :: *int* ⇒ *'a* **and** *d* :: *nat*
assume ∗: *eventually* (λ*n*. *f* (−*int n*) = *0*) *cofinite*
show *eventually* (λ*n*. (*if d* > *0* ∧ *int d dvd* −*int n then f* (−*int n div int d*) *else 0*) = *0*) *cofinite*
proof (*cases d* = *0*)
case *False*
from ∗ **have** *eventually* (λ*n*. *f* (−*int n*) = *0*) *at-top*
by (*simp add*: *cofinite-eq-sequentially*)
hence *eventually* (λ*n*. *f* (−*int* (*n div d*)) = *0*) *at-top*
by (*rule eventually-compose-filterlim*[*OF - filterlim-at-top-div-const-nat*]) (*use False* **in** *auto*)
hence *eventually* (λ*n*. (*if d* > *0* ∧ *int d dvd* −*int n then f* (−*int n div int d*) *else 0*) = *0*) *at-top*
by *eventually-elim* (*auto simp*: *zdiv-int dvd-neg-div*)
thus *?thesis*
by (*simp add*: *cofinite-eq-sequentially*)
qed *auto*
qed

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*)
by (*simp add*: *assms fls-compose-power.rep-eq*)

lemma *fls-compose-power-0-left* [*simp*]: *fls-compose-power 0 d* = *0*
by *transfer auto*

lemma *fls-compose-power-1-left* [*simp*]: *d* > *0* ⟹ *fls-compose-power 1 d* = *1*

by *transfer* (*auto simp*: *fun-eq-iff*)

lemma *fls-compose-power-const-left* [*simp*]:
d > *0* ⟹ *fls-compose-power* (*fls-const c*) *d* = *fls-const c*
by *transfer* (*auto simp*: *fun-eq-iff*)

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*)
by *transfer* (*auto simp*: *fun-eq-iff add-ac mult-ac*)

lemma *fls-compose-power-X-intpow* [*simp*]:
d > *0* ⟹ *fls-compose-power* (*fls-X-intpow n*) *d* = *fls-X-intpow* (*int d* ∗ *n*)
by *simp*

lemma *fls-compose-power-X* [*simp*]:
d > *0* ⟹ *fls-compose-power fls-X d* = *fls-X-intpow* (*int d*)
by *transfer* (*auto simp*: *fun-eq-iff*)

lemma *fls-compose-power-X-inv* [*simp*]:
d > *0* ⟹ *fls-compose-power fls-X-inv d* = *fls-X-intpow* (−*int d*)
by (*simp add*: *fls-X-inv-conv-shift-1*)

lemma *fls-compose-power-0-right* [*simp*]: *fls-compose-power f 0* = *0*
by *transfer auto*

lemma *fls-compose-power-add* [*simp*]:
fls-compose-power (*f* + *g*) *d* = *fls-compose-power f d* + *fls-compose-power g d*
by *transfer auto*

lemma *fls-compose-power-diff* [*simp*]:
fls-compose-power (*f* − *g*) *d* = *fls-compose-power f d* − *fls-compose-power g d*
by *transfer auto*

lemma *fls-compose-power-uminus* [*simp*]:
fls-compose-power (−*f*) *d* = −*fls-compose-power f d*
by *transfer auto*

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 −
have *fps-nth* (*f oo* (*fps-X* ^ *d*)) *n* = (∑ *i* = *0*..*n*. *f* $ *i* ∗ (*fps-X* ^ (*d* ∗ *i*)) $ *n*)
unfolding *fps-compose-def* **by** (*simp add*: *power-mult*)
also have ... = (∑ *i*∈(*if d dvd n then* {*n div d*} *else* {}). *f* $ *i* ∗ (*fps-X* ^ (*d* ∗ *i*)) $ *n*)
by (*intro sum.mono-neutral-right*) *auto*
also have ... = (*if d dvd n then fps-nth f* (*n div d*) *else 0*)
by *auto*
finally show *?thesis* **.**

qed

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))
using *assms*
by (*intro fls-eqI*) (*auto simp*: *fls-nth-compose-power fps-nth-compose-X-power pos-imp-zdiv-neg-iff div-neg-pos-less0 nat-div-distrib simp flip*: *int-dvd-int-iff*)

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 (*cases* $d > 0$)
case *True*
define *n* **where** n = *nat* (*max 0* (*max* (− *fls-subdegree f*) (− *fls-subdegree g*)))
have *n-ge*: −*fls-subdegree f* ≤ *int n* −*fls-subdegree g* ≤ *int n*
unfolding *n-def* **by** *auto*
obtain *f'* **where** *f'*: *f* = *fls-shift n* (*fps-to-fls f'*)
using *fls-as-fps*[*OF n-ge*(*1*)] **by** (*auto simp*: *n-def*)
obtain *g'* **where** *g'*: *g* = *fls-shift n* (*fps-to-fls g'*)
using *fls-as-fps*[*OF n-ge*(*2*)] **by** (*auto simp*: *n-def*)
show *?thesis* **using** ‹$d > 0$›
by (*simp add*: *f' g' fls-shifted-times-simps mult-ac fls-compose-power-fps-to-fls fps-compose-mult-distrib flip*: *fls-times-fps-to-fls*)
qed *auto*

lemma *fls-compose-power-power* [*simp*]:
assumes $d > 0 \vee n > 0$
shows *fls-compose-power* (*f* ^ *n* :: *'a* :: *idom fls*) d = *fls-compose-power f d* ^ *n*
proof (*cases* $d > 0$)
case *True*
thus *?thesis* **by** (*induction n*) *auto*
qed (*use assms* **in** *auto*)

lemma *fls-nth-compose-power'* [*simp*]:
$d = 0 \vee \neg d$ *dvd* $n \Longrightarrow$ *fls-compose-power f d* $$ *int n* = *0*
d dvd n $\Longrightarrow$ $d > 0 \Longrightarrow$ *fls-compose-power f d* $$ *int n* = *f* $$ *int* (*n div d*)
by (*transfer*; *force*; *fail*)+

lemma *subdegree-fls-compose-fps* [*simp*]:
fixes *G* :: *'a* :: *field fps*
assumes [*simp*]: *fps-nth G 0* = *0*
shows *fls-subdegree* (*fls-compose-fps F G*) = *fls-subdegree F* ∗ *subdegree G*
proof (*cases F* = *0*; *cases G* = *0*)
assume [*simp*]: *G* ≠ *0 F* ≠ *0*
have *nz1*: *fls-base-factor-to-fps F* ≠ *0*
using ‹*F* ≠ *0*› *fls-base-factor-to-fps-nonzero* **by** *blast*

show *?thesis*
unfolding *fls-compose-fps-def* **using** *nz1*
by (*subst fls-subdegree-mult*) (*simp-all add*: *fps-compose-eq-0-iff fls-subdegree-fls-to-fps*)
qed (*auto simp*: *fls-compose-fps-0-right*)

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−
obtain *N* **where** ∀ *n*<*N*. *f*$$*n* = 0 **by** (*elim fls-nth-vanishes-belowE*)
hence ∀ *n*<*N*−1. *of-int* (*n*+1) ∗ *f*$$(*n*+1) = 0 **by** *auto*
thus *?thesis* **using** *nth-Abs-fls-lower-bound* **unfolding** *fls-deriv-def* **by** *simp*
qed

lemma *fls-deriv-residue*: *fls-deriv f* $$ −1 = 0
by *simp*

lemma *fls-deriv-const*[*simp*]: *fls-deriv* (*fls-const x*) = 0
proof (*intro fls-eqI*)
fix *n* **show** *fls-deriv* (*fls-const x*) $$ *n* = 0$$*n*
by (*cases n*+1=0) *auto*
qed

lemma *fls-deriv-of-nat*[*simp*]: *fls-deriv* (*of-nat n*) = 0
by (*simp add*: *fls-of-nat*)

lemma *fls-deriv-of-int*[*simp*]: *fls-deriv* (*of-int i*) = 0
by (*simp add*: *fls-of-int*)

lemma *fls-deriv-zero*[*simp*]: *fls-deriv* 0 = 0
using *fls-deriv-const*[*of* 0] **by** *simp*

lemma *fls-deriv-one*[*simp*]: *fls-deriv* 1 = 0
using *fls-deriv-const*[*of* 1] **by** *simp*

lemma *fls-deriv-numeral* [*simp*]: *fls-deriv* (*numeral n*) = 0
by (*metis fls-deriv-of-int of-int-numeral*)

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
by (*auto intro*: *fls-subdegree-eqI simp*: *assms*)

lemma *fls-deriv-subdegree0*:
assumes *fls-subdegree f* = 0
shows *fls-subdegree* (*fls-deriv f*) ≥ 0

proof (*cases fls-deriv f = 0*)
case *False*
show *?thesis*
proof (*intro fls-subdegree-geI*, *rule False*)
fix *k* :: *int* **assume** *k < 0*
with *assms* **show** *fls-deriv f $$ k = 0* **by** (*cases k=−1*) *auto*
qed
qed *simp*

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*
using *assms nth-fls-subdegree-zero-iff*[*of f*]
by (*auto intro*: *fls-deriv-subdegree'*)

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*
by (*auto intro*: *fls-subdegree-deriv' simp*: *assms*)

lemma *fps-deriv-fls-regpart*: *fps-deriv* (*fls-regpart F*) = *fls-regpart* (*fls-deriv F*)
by (*intro fps-ext*) (*auto simp*: *add-ac*)

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*)
by (*intro fls-eqI*) (*simp flip*: *fls-shift-fls-shift add*: *algebra-simps*)

lemma *fls-deriv-X* [*simp*]: *fls-deriv fls-X = 1*
by (*intro fls-eqI*) *simp*

lemma *fls-deriv-X-inv* [*simp*]: *fls-deriv fls-X-inv = − (fls-X-inv*^{2})
proof−
have *fls-deriv fls-X-inv = − (fls-shift 2 1)*
by (*simp add*: *fls-X-inv-conv-shift-1 fls-deriv-shift*)
thus *?thesis* **by** (*simp add*: *fls-X-inv-power-conv-shift-1*)
qed

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−
have
fls-deriv (*Abs-fls* (*λn. if n=m then c else 0*)) = *fls-shift* (*1−m*) (*fls-const* (*of-int m ∗ c*))
using *fls-deriv-shift*[*of −m fls-const c*]

```
    by    (simp
           add: fls-shift-const fls-of-int fls-shifted-times-simps(1)[symmetric]
           fls-const-mult-const[symmetric]
           del: fls-const-mult-const
          )
  thus ?thesis by (simp add: fls-shift-const)
qed

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)
  by (simp add: fls-deriv-shift)

lemma fls-regpart-deriv: fls-regpart (fls-deriv f) = fps-deriv (fls-regpart f)
proof (intro fps-ext)
  fix n
  have  1: (of-nat n :: 'a) + 1 = of-nat (n+1)
  and   2: int n + 1 = int (n + 1)
    by  auto
  show fls-regpart (fls-deriv f) $ n = fps-deriv (fls-regpart f) $ n by (simp add: 1
2)
qed

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 (intro poly-eqI)
  fix n
  show
    coeff (fls-prpart (fls-deriv f)) n =
      coeff (− pCons 0 (pCons 0 (pderiv (fls-prpart f)))) n
  proof (cases n)
    case (Suc m)
    hence n: n = Suc m by fast
    show ?thesis
    proof (cases m)
      case (Suc k)
      with n have
        coeff (− pCons 0 (pCons 0 (pderiv (fls-prpart f)))) n =
          − coeff (pderiv (fls-prpart f)) k
        by (simp flip: coeff-minus)
      with Suc n show ?thesis by (simp add: coeff-pderiv algebra-simps)
    qed (simp add: n)
  qed simp
qed

lemma pderiv-fls-prpart:
```

pderiv (*fls-prpart f*) = − *poly-shift 2* (*fls-prpart* (*fls-deriv f*))
by (*intro poly-eqI*) (*simp add*: *coeff-pderiv coeff-poly-shift algebra-simps*)

lemma *fls-deriv-fps-to-fls*: *fls-deriv* (*fps-to-fls f*) = *fps-to-fls* (*fps-deriv f*)
proof (*intro fls-eqI*)
fix *n*
show *fls-deriv* (*fps-to-fls f*) $$ *n* = *fps-to-fls* (*fps-deriv f*) $$ *n*
proof (*cases n≥0*)
case *True*
from *True* **have** *1*: *nat* (*n* + *1*) = *nat n* + *1* **by** *simp*
from *True* **have** *2*: (*of-int* (*n* + *1*) :: *'a*) = *of-nat* (*nat* (*n+1*)) **by** *simp*
from *True* **show** *?thesis* **using** *arg-cong*[*OF 2*, *of* λ*x*. *x* ∗ *f* $ (*nat n+1*)] **by**
(*simp add*: *1*)
next
case *False* **thus** *?thesis* **by** (*cases n=−1*) *auto*
qed
qed

7.7.2 Algebraic rules of the derivative

lemma *fls-deriv-add* [*simp*]: *fls-deriv* (*f+g*) = *fls-deriv f* + *fls-deriv g*
by (*auto intro*: *fls-eqI simp*: *algebra-simps*)

lemma *fls-deriv-sub* [*simp*]: *fls-deriv* (*f−g*) = *fls-deriv f* − *fls-deriv g*
by (*auto intro*: *fls-eqI simp*: *algebra-simps*)

lemma *fls-deriv-neg* [*simp*]: *fls-deriv* (−*f*) = − *fls-deriv f*
using *fls-deriv-sub*[*of 0 f*] **by** *simp*

lemma *fls-deriv-mult* [*simp*]:
fls-deriv (*f*∗*g*) = *f* ∗ *fls-deriv g* + *fls-deriv f* ∗ *g*
proof−
define *df dg* :: *int*
where *df* ≡ *fls-subdegree f*
and *dg* ≡ *fls-subdegree g*
define *uf ug* :: *'a fls*
where *uf* ≡ *fls-base-factor f*
and *ug* ≡ *fls-base-factor g*
have
f ∗ *fls-deriv g* =
of-int dg ∗ *fls-shift* (*1* − *dg*) (*f* ∗ *ug*) + *fls-shift* (−*dg*) (*f* ∗ *fls-deriv ug*)
fls-deriv f ∗ *g* =
of-int df ∗ *fls-shift* (*1* − *df*) (*uf* ∗ *g*) + *fls-shift* (−*df*) (*fls-deriv uf* ∗ *g*)
using *fls-deriv-shift*[*of* −*df uf*] *fls-deriv-shift*[*of* −*dg ug*]
mult-of-int-commute[*of dg f*]
mult.assoc[*of of-int dg f*]
fls-shifted-times-simps(*1*)[*of f 1* − *dg ug*]
fls-shifted-times-simps(*1*)[*of f* −*dg fls-deriv ug*]
fls-shifted-times-simps(*2*)[*of 1* − *df uf g*]

```
      fls-shifted-times-simps(2)[of −df fls-deriv uf g]
  by (auto simp add: algebra-simps df-def dg-def uf-def ug-def)
 moreover have
  fls-deriv (f∗g) =
   ( of-int dg ∗ fls-shift (1 − dg) (f ∗ ug) + fls-shift (−dg) (f ∗ fls-deriv ug) ) +
   ( of-int df ∗ fls-shift (1 − df) (uf ∗ g) + fls-shift (−df) (fls-deriv uf ∗ g) )

  using fls-deriv-shift[of
      − (df + dg) fps-to-fls (fls-base-factor-to-fps f ∗ fls-base-factor-to-fps g)
    ]
    fls-deriv-fps-to-fls[of fls-base-factor-to-fps f ∗ fls-base-factor-to-fps g]
    fps-deriv-mult[of fls-base-factor-to-fps f fls-base-factor-to-fps g]
    distrib-right[of
      of-int df of-int dg
      fls-shift (1 − (df + dg)) (
        fps-to-fls (fls-base-factor-to-fps f ∗ fls-base-factor-to-fps g)
      )
    ]
    fls-times-conv-fps-times[of uf ug]
    fls-base-factor-subdegree[of f] fls-base-factor-subdegree[of g]
    fls-regpart-deriv[of ug]
    fls-times-conv-fps-times[of uf fls-deriv ug]
    fls-deriv-subdegree0[of ug]
    fls-regpart-deriv[of uf]
    fls-times-conv-fps-times[of fls-deriv uf ug]
    fls-deriv-subdegree0[of uf]
    fls-shifted-times-simps(1)[of uf −dg ug]
    fls-shifted-times-simps(1)[of fls-deriv uf −dg ug]
    fls-shifted-times-simps(2)[of −df uf ug]
    fls-shifted-times-simps(2)[of −df uf fls-deriv ug]
  by (simp add: fls-times-def algebra-simps df-def dg-def uf-def ug-def)
 ultimately show ?thesis by simp
qed

lemma fls-deriv-mult-const-left:
 fls-deriv (fls-const c ∗ f) = fls-const c ∗ fls-deriv f
 by simp

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
 by simp

lemma fls-deriv-mult-const-right:
 fls-deriv (f ∗ fls-const c) = fls-deriv f ∗ fls-const c
 by simp

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
  by simp

lemma fls-deriv-sum:
  fls-deriv (sum f S) = sum (λi. fls-deriv (f i)) S
proof (cases finite S)
  case True show ?thesis
    by (induct rule: finite-induct [OF True]) simp-all
qed simp

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 (cases n)
  case (Suc m)
  have fls-deriv (f^Suc m) = of-nat (Suc m) * f^m * fls-deriv f
    by (induct m) (simp-all add: algebra-simps)
  with Suc show ?thesis by simp
qed simp

lemma fls-deriv-X-power:
  fls-deriv (fls-X ^ n) = of-nat n * fls-X ^ (n-1)
proof (cases n)
  case (Suc m)
  have fls-deriv (fls-X^Suc m) = of-nat (Suc m) * fls-X^m
    by (induct m) (simp-all add: mult-of-nat-commute algebra-simps)
  with Suc show ?thesis by simp
qed simp

lemma fls-deriv-X-inv-power:
  fls-deriv (fls-X-inv ^ n) = - of-nat n * fls-X-inv ^ (Suc n)
proof (cases n)
  case (Suc m)
  define iX :: 'a fls where iX ≡ fls-X-inv
  have fls-deriv (iX ^ Suc m) = - of-nat (Suc m) * iX ^ (Suc (Suc m))
  proof (induct m)
    case (Suc m)
    have - of-nat (Suc m + 1) * iX ^ Suc (Suc (Suc m)) =
          iX * (-of-nat (Suc m) * iX ^ Suc (Suc m)) +
              - (iX ^ 2 * iX ^ Suc m)
      using distrib-right[of -of-nat (Suc m) -(1::'a fls) fls-X-inv ^ Suc (Suc (Suc
m))]
    by (simp add: algebra-simps mult-of-nat-commute power2-eq-square Suc iX-def)
    thus ?case using Suc by (simp add: iX-def)
  qed (simp add: numeral-2-eq-2 iX-def)
  with Suc show ?thesis by (simp add: iX-def)
qed simp

lemma fls-deriv-X-intpow:
```

fls-deriv (*fls-X-intpow i*) = *of-int i* ∗ *fls-X-intpow* (*i*−*1*)
by (*simp add*: *fls-deriv-shift*)

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−

define *L* **where** *L* ≡ *fls-left-inverse f x*
hence *fls-deriv* (*L* ∗ *f*) = *0* **using** *fls-left-inverse*[*OF assms*(*1*)] **by** *simp*
with *assms* **show** *fls-deriv L* = − *L* ∗ *fls-deriv f* ∗ *L*
using *fls-right-inverse*′[*OF assms*]
by (*simp add*: *minus-unique mult.assoc L-def*)

define *R* **where** *R* ≡ *fls-right-inverse f y*
hence *fls-deriv* (*f* ∗ *R*) = *0* **using** *fls-right-inverse*[*OF assms*(*2*)] **by** *simp*
hence *1*: *f* ∗ *fls-deriv R* + *fls-deriv f* ∗ *R* = *0* **by** *simp*
have *R* ∗ *f* ∗ *fls-deriv R* = − *R* ∗ *fls-deriv f* ∗ *R*
using *iffD2*[*OF eq-neg-iff-add-eq-0*, *OF 1*] **by** (*simp add*: *mult.assoc*)
thus *fls-deriv R* = − *R* ∗ *fls-deriv f* ∗ *R*
using *fls-left-inverse*′[*OF assms*] **by** (*simp add*: *R-def*)

qed

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
using *assms fls-deriv-lr-inverse*[*of x f x*]
by (*simp-all add*: *mult.commute power2-eq-square*)

lemma *fls-inverse-deriv-divring*:
fixes *a* :: ′*a*::*division-ring fls*
shows *fls-deriv* (*inverse a*) = − *inverse a* ∗ *fls-deriv a* ∗ *inverse a*
proof (*cases a*=*0*)
case *False* **thus** *?thesis*
using *fls-deriv-lr-inverse*(*2*)[*of*
inverse (*a* $$ *fls-subdegree a*) *a inverse* (*a* $$ *fls-subdegree a*)
]
by (*auto simp add*: *fls-inverse-def*′)
qed *simp*

lemma *fls-inverse-deriv*:
fixes *a* :: ′*a*::*field fls*

shows *fls-deriv* (*inverse a*) = − *fls-deriv a* ∗ (*inverse a*)^2
by (*simp add*: *fls-inverse-deriv-divring power2-eq-square*)

lemma *fls-inverse-deriv′*:
fixes *a* :: *′a*::*field fls*
shows *fls-deriv* (*inverse a*) = − *fls-deriv a* / *a*^{2}
using *fls-inverse-deriv*[*of a*]
by (*simp add*: *field-simps*)

lemma *fls-deriv-divide*:
fixes *f g* :: *′a* :: *field fls*
shows *fls-deriv* (*f* / *g*) = (*g* ∗ *fls-deriv f* − *f* ∗ *fls-deriv g*) / *g* ^ *2*
proof −
have *fls-deriv* (*f* / *g*) = *fls-deriv* (*f* ∗ *inverse g*)
by (*simp add*: *field-simps*)
also have ... = (*g* ∗ *fls-deriv f* − *f* ∗ *fls-deriv g*) / *g* ^ *2*
by (*subst fls-deriv-mult*, *subst fls-inverse-deriv′*)
(*simp add*: *divide-simps power2-eq-square*)
finally show *?thesis* **.**
qed

7.7.3 Equality of derivatives

lemma *fls-deriv-eq-0-iff*:
fls-deriv f = *0* ⟷ *f* = *fls-const* (*f*$$*0* :: *′a*::{*ring-1-no-zero-divisors*,*ring-char-0*})
proof
assume *f*: *fls-deriv f* = *0*
show *f* = *fls-const* (*f*$$*0*)
proof (*intro fls-eqI*)
fix *n*
from *f* **have** *of-int n* ∗ *f*$$ *n* = *0* **using** *fls-deriv-nth*[*of f n−1*] **by** *simp*
thus *f*$$*n* = *fls-const* (*f*$$*0*) $$ *n* **by** (*cases n=0*) *auto*
qed
next
show *f* = *fls-const* (*f*$$*0*) ⟹ *fls-deriv f* = *0* **using** *fls-deriv-const*[*of f*$$*0*] **by** *simp*
qed

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 −
have *fls-deriv f* = *fls-deriv g* ⟷ *fls-deriv* (*f* − *g*) = *0*
by *simp*
also have ... ⟷ *f* − *g* = *fls-const* ((*f* − *g*) $$ *0*)
unfolding *fls-deriv-eq-0-iff* **..**
finally show *?thesis*
by (*simp add*: *field-simps*)
qed

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*)
by (*auto simp*: *fls-deriv-eq-iff*)

lemma *fls-logderiv-unique*:
fixes *F1 F2 G* :: *'a* :: *field-char-0 fls*
assumes *fls-deriv F1* = *H* ∗ *F1 fls-deriv F2* = *H* ∗ *F2 F1* ≠ *0*
obtains *c* **where** *F2* = *fls-const c* ∗ *F1*
proof −
define *Q* **where** *Q* = *F2* / *F1*
have *fls-deriv Q* = *0*
by (*simp add*: *Q-def fls-deriv-divide assms*)
then obtain *c* **where** *Q* = *fls-const c*
using *fls-deriv-eq-0-iff* **by** *auto*
thus *?thesis*
by (*intro that*[*of c*]) (*use assms* **in** ‹*auto simp*: *Q-def field-simps*›)
qed

lemma *fps-logderiv-unique*:
fixes *F1 F2 G* :: *'a* :: *field-char-0 fps*
assumes *fps-deriv F1* = *H* ∗ *F1 fps-deriv F2* = *H* ∗ *F2 F1* ≠ *0*
obtains *c* **where** *F2* = *fps-const c* ∗ *F1*
proof −
obtain *c* **where** *fps-to-fls F2* = *fls-const c* ∗ *fps-to-fls F1*
by (*rule fls-logderiv-unique*[*of fps-to-fls F1 fps-to-fls H fps-to-fls F2*])
(*use assms* **in** ‹*auto simp*: *fls-deriv-fps-to-fls fls-times-fps-to-fls*›)
also have ... = *fps-to-fls* (*fps-const c* ∗ *F1*)
by (*simp add*: *fls-times-fps-to-fls*)
finally show *?thesis*
by (*intro that*[*of c*]) *simp*
qed

7.7.4 Residues

definition *fls-residue-def*[*simp*]: *fls-residue f* ≡ *f* $$ −*1*

lemma *fls-residue-deriv*: *fls-residue* (*fls-deriv f*) = *0*
by *simp*

lemma *fls-residue-add*: *fls-residue* (*f*+*g*) = *fls-residue f* + *fls-residue g*
by *simp*

lemma *fls-residue-times-deriv*:
fls-residue (*fls-deriv f* ∗ *g*) = − *fls-residue* (*f* ∗ *fls-deriv g*)
using *fls-residue-deriv*[*of f*∗*g*] *minus-unique*[*of fls-residue* (*f* ∗ *fls-deriv g*)]
by *simp*

lemma *fls-residue-power-series*: *fls-subdegree f* $\geq$ *0* $\Longrightarrow$ *fls-residue f* = *0*
by *simp*

lemma *fls-residue-fls-X-intpow*:
fls-residue (*fls-X-intpow i*) = (*if i*=−*1 then 1 else 0*)
by *simp*

lemma *fls-residue-shift-nth*:
fixes *f* :: *'a*::*semiring-1 fls*
shows *f*$$*n* = *fls-residue* (*fls-X-intpow* (−*n*−*1*) ∗ *f*)
by (*simp add*: *fls-shifted-times-transfer*)

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*
by *simp-all*

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*
by (*simp-all add*: *fls-residue-fls-const-times fls-of-int*)

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−
define *df* :: *int* **where** *df* ≡ *fls-subdegree f*
define *B X* :: *'a fls*
where *B* ≡ *fls-base-factor f*
and *X* ≡ (*fls-X-intpow df* :: *'a fls*)
define *D L R* :: *'a fls*
where *D* ≡ *fls-deriv B*
and *L* ≡ *fls-left-inverse B y*
and *R* ≡ *fls-right-inverse B y*
have *intpow-diff*: *fls-X-intpow* (*df* − *1*) = *X* ∗ *fls-X-inv*
using *fls-X-intpow-diff-conv-times*[*of df 1*] **by** (*simp add*: *X-def fls-X-inv-conv-shift-1*)

show *fls-residue* (*fls-deriv f* ∗ *fls-right-inverse f y*) = *of-int df*
proof−
have *subdegree-DR*: *fls-subdegree* (*D* ∗ *R*) $\geq$ *0*
using *fls-base-factor-subdegree*[*of f*] *fls-base-factor-subdegree*[*of fls-right-inverse f y*]

```
        assms(1) fls-right-inverse-base-factor[of y f] fls-mult-subdegree-ge-0[of D
R]
    by  (force simp: fls-deriv-subdegree0 D-def R-def B-def)
  have decomp: f = X * B
   unfolding X-def B-def df-def by (rule fls-base-factor-X-power-decompose(2)[of
f])
  hence fls-deriv f = X * D + of-int df * X * fls-X-inv * B
    using intpow-diff fls-deriv-mult[of X B]
    by  (simp add: fls-deriv-X-intpow X-def B-def D-def mult.assoc)
  moreover from assms have fls-right-inverse (X * B) y = R * fls-right-inverse
X 1
    using fls-base-factor-base[of f] fls-lr-inverse-mult-ring1(2)[of 1 X]
    by  (simp add: X-def B-def R-def)
  ultimately have
    fls-deriv f * fls-right-inverse f y =
      (D + of-int df * fls-X-inv * B) * R * (X * fls-right-inverse X 1)
    by (simp add: decomp algebra-simps X-def fls-X-intpow-times-comm)
  also have ... = D * R + of-int df * fls-X-inv
    using fls-right-inverse[of X 1]
        assms fls-base-factor-base[of f] fls-right-inverse[of B y]
    by  (simp add: X-def distrib-right mult.assoc B-def R-def)
  finally show ?thesis using subdegree-DR by simp
 qed

 with assms show fls-residue (fls-deriv f * fls-left-inverse f y) = of-int df
  using fls-left-inverse-eq-fls-right-inverse[of y f] by simp

 show fls-residue (fls-left-inverse f y * fls-deriv f) = of-int df
 proof−
  have subdegree-LD: fls-subdegree (L * D) ≥ 0
   using fls-base-factor-subdegree[of f] fls-base-factor-subdegree[of fls-left-inverse
f y]
        assms(1) fls-left-inverse-base-factor[of y f] fls-mult-subdegree-ge-0[of L
D]
    by  (force simp: fls-deriv-subdegree0 D-def L-def B-def)
  have decomp: f = B * X
   unfolding X-def B-def df-def by (rule fls-base-factor-X-power-decompose(1)[of
f])
  hence fls-deriv f = D * X + B * of-int df * X * fls-X-inv
    using intpow-diff fls-deriv-mult[of B X]
    by  (simp add: fls-deriv-X-intpow X-def D-def B-def mult.assoc)
  moreover from assms have fls-left-inverse (B * X) y = fls-left-inverse X 1 *
L
    using fls-base-factor-base[of f] fls-lr-inverse-mult-ring1(1)[of - - 1 X]
    by  (simp add: X-def B-def L-def)
  ultimately have
    fls-left-inverse f y * fls-deriv f =
      fls-left-inverse X 1 * X * L * (D + B * (of-int df * fls-X-inv))
    by (simp add: decomp algebra-simps X-def fls-X-intpow-times-comm)
```

also have ... = $L * D$ + *of-int df* * *fls-X-inv*
using *assms fls-left-inverse*[*of 1 X*] *fls-base-factor-base*[*of f*] *fls-left-inverse*[*of y B*]
by (*simp add*: *X-def distrib-left mult.assoc*[*symmetric*] *L-def B-def*)
finally show *?thesis* **using** *subdegree-LD* **by** *simp*
qed

with *assms* **show** *fls-residue* (*fls-right-inverse f y* * *fls-deriv f*) = *of-int df*
using *fls-left-inverse-eq-fls-right-inverse*[*of y f*] **by** *simp*

qed

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−
show *fls-residue* (*fls-deriv f* * *inverse f*) = *of-int* (*fls-subdegree f*)
using *fls-residue-deriv-times-lr-inverse-eq-subdegree*(*1*)[*of - f*]
by (*cases f=0*) (*auto simp*: *fls-inverse-def'*)
show *fls-residue* (*inverse f* * *fls-deriv f*) = *of-int* (*fls-subdegree f*)
using *fls-residue-deriv-times-lr-inverse-eq-subdegree*(*4*)[*of - f*]
by (*cases f=0*) (*auto simp*: *fls-inverse-def'*)
qed

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−
define *F* **where** *F* ≡ (λ*n*. *if n=0 then 0 else inverse* (*of-int n*) * *a*$$(*n* − *1*))
obtain *N* **where** ∀ *n*<*N*. *a*$$*n* = *0* **by** (*elim fls-nth-vanishes-belowE*)
hence ∀ *n*<*N*. *F n* = *0* **by** (*auto simp add*: *F-def*)
thus *?thesis* **using** *nth-Abs-fls-lower-bound*[*of N F*] **unfolding** *fls-integral-def F-def* **by** *simp*
qed

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 (*rule fls-eqI*)
fix *n*
show *fls-integral a* $$ *n* = *fps-to-fls* (*fps-integral0* (*fls-regpart a*)) $$ *n*
proof (*cases n>0*)
case *False* **with** *assms* **show** *?thesis* **by** *simp*

```
  next
    case True
    hence int ((nat n) - 1) = n - 1 by simp
    with True show ?thesis by (simp add: fps-integral-def)
  qed
qed

lemma fls-integral-zero [simp]: fls-integral 0 = 0
  by (intro fls-eqI) simp

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
  by      (intro fls-eqI) (simp add: assms)

lemma fls-integral-const:
  fixes x :: 'a::division-ring
  shows fls-integral (fls-const x) = fls-const x * fls-X
  by    (rule fls-integral-const'[OF inverse-1])

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
  by      (simp add: assms fls-integral-const' fls-of-nat)

lemma fls-integral-of-nat:
  fls-integral (of-nat n :: 'a::division-ring fls) = of-nat n * fls-X
  by (rule fls-integral-of-nat'[OF inverse-1])

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
  by      (simp add: assms fls-integral-const' fls-of-int)

lemma fls-integral-of-int:
  fls-integral (of-int i :: 'a::division-ring fls) = of-int i * fls-X
  by (rule fls-integral-of-int'[OF inverse-1])

lemma fls-integral-one':
  assumes inverse (1::'a::{ring-1,inverse}) = 1
  shows   fls-integral (1::'a fls) = fls-X
  using   fls-integral-const'[of 1]
  by      (force simp: assms)

lemma fls-integral-one: fls-integral (1::'a::division-ring fls) = fls-X
  by (rule fls-integral-one'[OF inverse-1])

lemma fls-subdegree-integral-ge:
```

fls-integral f $\neq$ *0* $\Longrightarrow$ *fls-subdegree* (*fls-integral f*) $\geq$ *fls-subdegree f* + *1*
by (*intro fls-subdegree-geI*) *simp-all*

lemma *fls-subdegree-integral*:
fixes *f* :: *'a*::{*division-ring*,*ring-char-0*} *fls*
assumes *f* $\neq$ *0 fls-subdegree f* $\neq$ -1
shows *fls-subdegree* (*fls-integral f*) = *fls-subdegree f* + *1*
using *assms of-int-0-eq-iff*[*of fls-subdegree f* + *1*] *fls-subdegree-integral-ge*
by (*intro fls-subdegree-eqI*) *simp-all*

lemma *fls-integral-X* [*simp*]:
fls-integral (*fls-X*::*'a*::{*ring-1*,*inverse*} *fls*) =
fls-const (*inverse* (*of-int 2*)) $*$ *fls-X*^{2}
proof (*intro fls-eqI*)
fix *n*
show *fls-integral* (*fls-X*::*'a fls*) $$ *n* = (*fls-const* (*inverse* (*of-int 2*)) $*$ *fls-X*^{2}) $$ *n*
using *arg-cong*[*OF fls-X-power-nth*, *of* λx. *inverse* (*of-int 2*) $*$ *x*, *of 2 n*, *symmetric*]
by (*auto simp add*:)
qed

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 (*intro fls-eqI*)
fix *k*
have (*fls-X* :: *'a fls*) ^ *Suc n* $$ *k* = (*if k*=*Suc n then 1 else 0*)
by (*rule fls-X-power-nth*)
thus
fls-integral ((*fls-X*::*'a fls*) ^ *n*) $$ *k* =
(*fls-const* (*inverse* (*of-nat* (*Suc n*))) $*$ (*fls-X*::*'a fls*) ^ *Suc n*) $$ *k*
by *simp*
qed

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 $-$
have (*of-nat* (*Suc n*) :: *'a*) $\neq$ *0* **by** (*rule of-nat-neq-0*)
hence *fls-const* (*inverse* (*of-nat* (*Suc n*) :: *'a*)) = *inverse* (*fls-const* (*of-nat* (*Suc n*)))
by (*simp add*: *fls-inverse-const*)
moreover have
fls-integral ((*fls-X*::*'a fls*) ^ *n*) = *fls-const* (*inverse* (*of-nat* (*Suc n*))) $*$ *fls-X* ^ *Suc n*
by (*rule fls-integral-X-power*)
ultimately show *?thesis* **by** (*simp add*: *fls-of-nat*)
qed

lemma *fls-integral-X-inv* [*simp*]: *fls-integral* (*fls-X-inv*::*'a*::{*ring-1*,*inverse*} *fls*) = *0*
by (*intro fls-eqI*) *simp*

lemma *fls-integral-X-inv-power*:
assumes $n \geq 2$
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 (*rule fls-eqI*)
fix *k* **show**
fls-integral (*fls-X-inv* ^ *n* :: *'a fls*) $$ *k*=
(*fls-const* (*inverse* (*of-int* (*1* − *int n*))) ∗ *fls-X-inv* ^ (*n*−*1*)) $$ *k*
proof (*cases k*=*0*)
case *True* **with** *assms* **show** *?thesis* **by** *simp*
next
case *False*
from *assms* **have** *int* (*n*−*1*) = *int n* − *1* **by** *simp*
hence
(*fls-const* (*inverse* (*of-int* (*1* − *int n*))) ∗ (*fls-X-inv*:: *'a fls*) ^ (*n*−*1*)) $$ *k* =
(*if k* = *1* − *int n then inverse* (*of-int k*) *else 0*)
by (*simp add*: *fls-X-inv-power-times-conv-shift*(*2*))
with *False* **show** *?thesis* **by** (*simp add*: *algebra-simps*)
qed
qed

lemma *fls-integral-X-inv-power-char0*:
assumes $n \geq 2$
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−
from *assms* **have** (*of-int* (*1* − *int n*) :: *'a*) ≠ *0* **by** *simp*
hence
fls-const (*inverse* (*of-int* (*1* − *int n*) :: *'a*)) = *inverse* (*fls-const* (*of-int* (*1* − *int n*)))
by (*simp add*: *fls-inverse-const*)
moreover have
fls-integral (*fls-X-inv* ^ *n* :: *'a fls*) =
fls-const (*inverse* (*of-int* (*1* − *int n*))) ∗ *fls-X-inv* ^ (*n*−*1*)
using *assms* **by** (*rule fls-integral-X-inv-power*)
ultimately show *?thesis* **by** (*simp add*: *fls-of-int*)
qed

lemma *fls-integral-X-inv-power'*:
assumes $n \geq 1$
shows
fls-integral (*fls-X-inv* ^ *n* :: *'a* :: *division-ring fls*) =

```
    − fls-const (inverse (of-nat (n−1))) ∗ fls-X-inv ^ (n−1)
proof (cases n = 1)
  case False
  with assms have n: n ≥ 2 by simp
  hence
    fls-integral (fls-X-inv ^ n :: 'a fls) =
      fls-const (inverse (− of-nat (nat (int n − 1)))) ∗ fls-X-inv ^ (n−1)
    by (simp add: fls-integral-X-inv-power)
  moreover from n have nat (int n − 1) = n − 1 by simp
  ultimately show ?thesis
    using inverse-minus-eq[of of-nat (n−1) :: 'a] by simp
qed simp

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 (cases n=1)
  case False with assms show ?thesis
    by (simp add: fls-integral-X-inv-power' fls-inverse-const fls-of-nat)
qed simp

lemma fls-integral-delta:
  assumes m ≠ −1
  shows
    fls-integral (Abs-fls (λn. if n=m then c else 0)) =
      Abs-fls (λn. if n=m+1 then inverse (of-int (m+1)) ∗ c else 0)
  using   assms
  by      (intro fls-eqI) auto

lemma fls-regpart-integral:
  fls-regpart (fls-integral f) = fps-integral0 (fls-regpart f)
proof (rule fps-ext)
  fix n
  show fls-regpart (fls-integral f) $ n = fps-integral0 (fls-regpart f) $ n
    by (cases n) (simp-all add: fps-integral-def)
qed

lemma fls-integral-fps-to-fls:
  fls-integral (fps-to-fls f) = fps-to-fls (fps-integral0 f)
proof (intro fls-eqI)
  fix n :: int
  show fls-integral (fps-to-fls f) $$ n = fps-to-fls (fps-integral0 f) $$ n
  proof (cases n<1)
    case True thus ?thesis by simp
  next
    case False
    hence nat (n−1) = nat n − 1 by simp
```

with *False* **show** *?thesis* **by** (*cases nat n*) *auto*
qed
qed

7.7.6 Algebraic rules of the integral

lemma *fls-integral-add* [*simp*]: *fls-integral* (*f*+*g*) = *fls-integral f* + *fls-integral g*
by (*intro fls-eqI*) (*simp add*: *algebra-simps*)

lemma *fls-integral-sub* [*simp*]: *fls-integral* (*f*−*g*) = *fls-integral f* − *fls-integral g*
by (*intro fls-eqI*) (*simp add*: *algebra-simps*)

lemma *fls-integral-neg* [*simp*]: *fls-integral* (−*f*) = − *fls-integral f*
using *fls-integral-sub*[*of 0 f*] **by** *simp*

lemma *fls-integral-mult-const-left*:
fls-integral (*fls-const c* ∗ *f*) = *fls-const c* ∗ *fls-integral* (*f* :: *'a*::*division-ring fls*)
by (*intro fls-eqI*) (*simp add*: *mult.assoc mult-inverse-of-int-commute*)

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*
by (*intro fls-eqI*) (*simp add*: *mult.assoc mult.commute*)

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*
by (*simp add*: *fls-integral-mult-const-left*)

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*
by (*simp add*: *fls-integral-mult-const-left-comm*)

lemma *fls-integral-mult-const-right*:
fls-integral (*f* ∗ *fls-const c*) = *fls-integral f* ∗ *fls-const c*
by (*intro fls-eqI*) (*simp add*: *mult.assoc*)

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*
by (*simp add*: *fls-integral-mult-const-right*)

lemma *fls-integral-sum*:
fls-integral (*sum f S*) = *sum* (λ*i*. *fls-integral* (*f i*)) *S*

proof (*cases finite S*)
  case *True* **show** *?thesis*
    by (*induct rule*: *finite-induct* [*OF True*]) *simp-all*
qed *simp*

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*
  by (*intro fls-eqI*) (*simp add*: *mult.assoc*[*symmetric*])

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 (*intro fls-eqI*)
  fix *n* :: *int*
  show *fls-deriv* (*fls-integral a*) $$ *n* = *a* $$ *n*
  proof (*cases n*=−*1*)
    case *True* **with** *assms* **show** *?thesis* **by** *simp*
  next
    case *False*
    hence (*of-int* (*n*+*1*) :: *'a*) ≠ *0* **using** *of-int-eq-0-iff*[*of n*+*1*] **by** *simp*
    hence (*of-int* (*n*+*1*) :: *'a*) ∗ *inverse* (*of-int* (*n*+*1*) :: *'a*) = (*1*::*'a*)
      using *of-int-eq-0-iff*[*of n*+*1*] **by** *simp*
    moreover have
      fls-deriv (*fls-integral a*) $$ *n* =
        (*if n*=−*1 then 0 else of-int* (*n*+*1*) ∗ *inverse* (*of-int* (*n*+*1*)) ∗ *a$$n*)
      by (*simp add*: *mult.assoc*)
    ultimately show *?thesis*
      by (*simp add*: *False*)
  qed
qed

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*
  using *assms fls-deriv-fls-integral*
  by *auto*

lemma *fls-ex-antiderivative-residue-nonzero*:
  assumes ∃ *F*. *fls-deriv F* = *f*
  shows *fls-residue f* = *0*
  using *assms fls-residue-deriv*
  by *auto*

lemma *fls-residue-nonzero-ex-anitderivative-iff*:
fixes f :: *'a*::{*division-ring*,*ring-char-0*} *fls*
shows *fls-residue* $f = 0 \longleftrightarrow (\exists F.$ *fls-deriv* $F = f)$
using *fls-residue-nonzero-ex-antiderivative fls-ex-antiderivative-residue-nonzero*
by *fast*

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 $(a-b) \geq 0$
then inverse $(2 \hat{} nat$ (*fls-subdegree* $(a-b)))$
else $2 \hat{} nat$ $(-$*fls-subdegree* $(a-b))$
)

lemma *dist-fls-ge0*: *dist* (a :: *'a fls*) $b \geq 0$
by (*simp add*: *dist-fls-def*)

definition *uniformity-fls-def*:
(*uniformity* :: (*'a fls* × *'a fls*) *filter*) = (*INF* $e \in \{0 <..\}$. *principal* $\{(x, y).$ *dist* $x\ y < e\}$)

definition *open-fls-def'*:
open (U :: *'a fls set*) $\longleftrightarrow$ ($\forall x \in U.$ *eventually* $(\lambda(x', y).\ x' = x \longrightarrow y \in U)$ *uniformity*)

lemma *dist-fls-sym*: *dist* (a :: *'a fls*) b = *dist* $b\ a$
by (*cases* $a \neq b$, *cases fls-subdegree* $(a-b) \geq 0$)
(*simp-all add*: *fls-subdegree-minus-sym dist-fls-def*)

context
begin

private lemma *instance-helper*:
fixes $a\ b\ c$ :: *'a fls*
assumes *neq*: $a \neq b\ a \neq c$
and *dist-ineq*: *dist* $a\ b >$ *dist* $a\ c$
shows *fls-subdegree* $(a - b) <$ *fls-subdegree* $(a - c)$
proof (
cases fls-subdegree $(a-b) \geq 0$ *fls-subdegree* $(a-c) \geq 0$
rule: *case-split*[*case-product case-split*]
)
case *True-True* **with** *neq dist-ineq* **show** *?thesis* **by** (*simp add*: *dist-fls-def*)

next
case *False-True* **with** *dist-ineq* **show** *?thesis* **by** (*simp add*: *dist-fls-def*)
next
case *False-False* **with** *neq dist-ineq* **show** *?thesis* **by** (*simp add*: *dist-fls-def*)
next
case *True-False*
with *neq*
have (*1*::*real*) > *2* ^ (*nat* (*fls-subdegree* (*a*−*b*)) + *nat* (−*fls-subdegree* (*a*−*c*)))
and *nat* (*fls-subdegree* (*a*−*b*)) + *nat* (−*fls-subdegree* (*a*−*c*)) =
nat (*fls-subdegree* (*a*−*b*) − *fls-subdegree* (*a*−*c*))
using *dist-ineq*
by (*simp-all add*: *dist-fls-def field-simps power-add*)
hence ¬ (*1*::*real*) < *2* ^ (*nat* (*fls-subdegree* (*a*−*b*) − *fls-subdegree* (*a*−*c*))) **by** *simp*
hence ¬ (*0* < *nat* (*fls-subdegree* (*a* − *b*) − *fls-subdegree* (*a* − *c*))) **by** *auto*
hence *fls-subdegree* (*a* − *b*) ≤ *fls-subdegree* (*a* − *c*) **by** *simp*
with *True-False* **show** *?thesis* **by** *simp*
qed

instance
proof
show *th*: *dist a b* = *0* ⟷ *a* = *b* **for** *a b* :: *'a fls*
by (*simp add*: *dist-fls-def split*: *if-split-asm*)
then have *th'*[*simp*]: *dist a a* = *0* **for** *a* :: *'a fls* **by** *simp*

fix *a b c* :: *'a fls*
consider *a* = *b* | *c* = *a* ∨ *c* = *b* | *a* ≠ *b a* ≠ *c b* ≠ *c* **by** *blast*
then show *dist a b* ≤ *dist a c* + *dist b c*
proof *cases*
case *1*
then show *?thesis* **by** (*simp add*: *dist-fls-def*)
next
case *2*
then show *?thesis*
by (*cases c* = *a*) (*simp-all add*: *th dist-fls-sym*)
next
case *neq*: *3*
have *False* **if** *dist a b* > *dist a c* + *dist b c*
proof −
from *neq* **have** *dist a b* > *0 dist b c* > *0 dist a c* > *0* **by** (*simp-all add*: *dist-fls-def*)
with *that* **have** *dist-ineq*: *dist a b* > *dist a c dist a b* > *dist b c* **by** *simp-all*
have *fls-subdegree* (*a* − *b*) < *fls-subdegree* (*a* − *c*)
and *fls-subdegree* (*a* − *b*) < *fls-subdegree* (*b* − *c*)
using *instance-helper*[*of a b c*] *instance-helper*[*of b a c*] *neq dist-ineq*
by (*simp-all add*: *dist-fls-sym fls-subdegree-minus-sym*)
hence (*a* − *c*) $$ *fls-subdegree* (*a* − *b*) = *0* **and** (*b* − *c*) $$ *fls-subdegree* (*a* − *b*) = *0*
by (*simp-all only*: *fls-eq0-below-subdegree*)

hence $(a - b)$ $\$\$$ *fls-subdegree* $(a - b) = 0$ **by** *simp*
moreover from *neq* **have** $(a - b)$ $\$\$$ *fls-subdegree* $(a - b) \neq 0$
by (*intro nth-fls-subdegree-nonzero*) *simp*
ultimately show *False* **by** *contradiction*
qed
thus *?thesis* **by** (*auto simp*: *not-le*[*symmetric*])
qed
qed (*rule open-fls-def' uniformity-fls-def*)+

end
end

lemma *open-fls-def*:
open $(S :: 'a::group\text{-}add\ fls\ set) = (\forall a \in S.\ \exists r.\ r > 0 \wedge \{y.\ dist\ y\ a < r\} \subseteq S)$
unfolding *open-dist subset-eq* **by** *simp*

7.9 Notation

bundle *fps-syntax*
begin
notation *fls-nth* (**infixl** ‹$$› *75*)
end

unbundle *no fps-syntax*

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 $x\ y \longleftrightarrow snd\ x \neq 0 \wedge snd\ y \neq 0 \wedge fst\ x * snd\ y = fst\ y * snd\ x$
by (*simp add*: *fractrel-def*)

lemma *symp-fractrel*: *symp fractrel*
by (*simp add*: *symp-def*)

lemma *transp-fractrel*: *transp fractrel*
proof (*rule transpI*, *unfold split-paired-all*)
  fix *a b a' b' a'' b''* :: *'a*
  assume *A*: *fractrel* (*a*, *b*) (*a'*, *b'*)
  assume *B*: *fractrel* (*a'*, *b'*) (*a''*, *b''*)
  have $b' * (a * b'') = b'' * (a * b')$ **by** (*simp add*: *ac-simps*)
  also from *A* **have** $a * b' = a' * b$ **by** *auto*
  also have $b'' * (a' * b) = b * (a' * b'')$ **by** (*simp add*: *ac-simps*)
  also from *B* **have** $a' * b'' = a'' * b'$ **by** *auto*
  also have $b * (a'' * b') = b' * (a'' * b)$ **by** (*simp add*: *ac-simps*)
  finally have $b' * (a * b'') = b' * (a'' * b)$ **.**
  moreover from *B* **have** $b' \neq 0$ **by** *auto*
  ultimately have $a * b'' = a'' * b$ **by** *simp*
  with *A B* **show** *fractrel* (*a*, *b*) (*a''*, *b''*) **by** *auto*
qed

lemma *part-equivp-fractrel*: *part-equivp fractrel*
using - *symp-fractrel transp-fractrel*
by(*rule part-equivpI*)(*rule exI*[**where** *x*=(*0*, *1*)]; *simp*)

end

quotient-type (**overloaded**) *'a fract* = *'a* :: *idom* × *'a* / *partial*: *fractrel*
by(*rule part-equivp-fractrel*)

8.1.2 Representation and basic operations

lift-definition *Fract* :: *'a* :: *idom* ⇒ *'a* ⇒ *'a fract*
  is *λa b. if b = 0 then* (*0*, *1*) *else* (*a*, *b*)
  by *simp*

lemma *Fract-cases* [*cases type*: *fract*]:
  obtains (*Fract*) *a b* **where** *q* = *Fract a b b* ≠ *0*
by *transfer simp*

lemma *Fract-induct* [*case-names Fract*, *induct type*: *fract*]:
  (⋀*a b. b* ≠ *0* ⟹ *P* (*Fract a b*)) ⟹ *P q*
  by (*cases q*) *simp*

lemma *eq-fract*:
  shows ⋀*a b c d. b* ≠ *0* ⟹ *d* ≠ *0* ⟹ *Fract a b* = *Fract c d* ⟷ *a* * *d* = *c* * *b*
    and ⋀*a. Fract a 0* = *Fract 0 1*
    and ⋀*a c. Fract 0 a* = *Fract 0 c*
by(*transfer*; *simp*)+

instantiation *fract* :: (*idom*) *comm-ring-1*
begin

lift-definition *zero-fract* :: *'a fract* **is** (*0*, *1*) **by** *simp*

lemma *Zero-fract-def*: *0 = Fract 0 1*
by *transfer simp*

lift-definition *one-fract* :: *'a fract* **is** *(1, 1)* **by** *simp*

lemma *One-fract-def*: *1 = Fract 1 1*
by *transfer simp*

lift-definition *plus-fract* :: *'a fract ⇒ 'a fract ⇒ 'a fract*
is *λq r. (fst q * snd r + fst r * snd q, snd q * snd r)*
by(*auto simp add*: *algebra-simps*)

lemma *add-fract* [*simp*]:
*⟦ b ≠ 0; d ≠ 0 ⟧ ⟹ Fract a b + Fract c d = Fract (a * d + c * b) (b * d)*
by *transfer simp*

lift-definition *uminus-fract* :: *'a fract ⇒ 'a fract*
is *λx. (− fst x, snd x)*
by *simp*

lemma *minus-fract* [*simp*]:
fixes *a b* :: *'a*::*idom*
shows *− Fract a b = Fract (− a) b*
by *transfer simp*

lemma *minus-fract-cancel* [*simp*]: *Fract (− a) (− b) = Fract a b*
by *(cases b = 0) (simp-all add*: *eq-fract)*

definition *diff-fract-def*: *q − r = q + − (r*::*'a fract)*

lemma *diff-fract* [*simp*]:
*⟦ b ≠ 0; d ≠ 0 ⟧ ⟹ Fract a b − Fract c d = Fract (a * d − c * b) (b * d)*
by *(simp add*: *diff-fract-def)*

lift-definition *times-fract* :: *'a fract ⇒ 'a fract ⇒ 'a fract*
is *λq r. (fst q * fst r, snd q * snd r)*
by(*simp add*: *algebra-simps*)

lemma *mult-fract* [*simp*]: *Fract (a*::*'a*::*idom) b * Fract c d = Fract (a * c) (b * d)*
by *transfer simp*

lemma *mult-fract-cancel*:
*c ≠ 0 ⟹ Fract (c * a) (c * b) = Fract a b*
by *transfer simp*

instance
proof
fix *q r s* :: *'a fract*

```
  show (q * r) * s = q * (r * s)
    by (cases q, cases r, cases s) (simp add: eq-fract algebra-simps)
  show q * r = r * q
    by (cases q, cases r) (simp add: eq-fract algebra-simps)
  show 1 * q = q
    by (cases q) (simp add: One-fract-def eq-fract)
  show (q + r) + s = q + (r + s)
    by (cases q, cases r, cases s) (simp add: eq-fract algebra-simps)
  show q + r = r + q
    by (cases q, cases r) (simp add: eq-fract algebra-simps)
  show 0 + q = q
    by (cases q) (simp add: Zero-fract-def eq-fract)
  show - q + q = 0
    by (cases q) (simp add: Zero-fract-def eq-fract)
  show q - r = q + - r
    by (cases q, cases r) (simp add: eq-fract)
  show (q + r) * s = q * s + r * s
    by (cases q, cases r, cases s) (simp add: eq-fract algebra-simps)
  show (0::'a fract) ≠ 1
    by (simp add: Zero-fract-def One-fract-def eq-fract)
qed

end

lemma of-nat-fract: of-nat k = Fract (of-nat k) 1
  by (induct k) (simp-all add: Zero-fract-def One-fract-def)

lemma Fract-of-nat-eq: Fract (of-nat k) 1 = of-nat k
  by (rule of-nat-fract [symmetric])

lemma fract-collapse:
  Fract 0 k = 0
  Fract 1 1 = 1
  Fract k 0 = 0
by(transfer; simp)+

lemma fract-expand:
  0 = Fract 0 1
  1 = Fract 1 1
  by (simp-all add: fract-collapse)

lemma Fract-cases-nonzero:
  obtains (Fract) a b where q = Fract a b and b ≠ 0 and a ≠ 0
    | (0) q = 0
proof (cases q = 0)
  case True
  then show thesis using 0 by auto
next
  case False
```

then obtain *a b* **where** *q = Fract a b* **and** *b ≠ 0* **by** (*cases q*) *auto*
with *False* **have** *0 ≠ Fract a b* **by** *simp*
with ‹*b ≠ 0*› **have** *a ≠ 0* **by** (*simp add*: *Zero-fract-def eq-fract*)
with *Fract* ‹*q = Fract a b*› ‹*b ≠ 0*› **show** *thesis* **by** *auto*
qed

8.1.3 The field of rational numbers

context *idom*
begin

subclass *ring-no-zero-divisors* **..**

end

instantiation *fract* :: (*idom*) *field*
begin

lift-definition *inverse-fract* :: *'a fract ⇒ 'a fract*
is *λx. if fst x = 0 then* (*0*, *1*) *else* (*snd x*, *fst x*)
by(*auto simp add*: *algebra-simps*)

lemma *inverse-fract* [*simp*]: *inverse* (*Fract a b*) = *Fract* (*b*::*'a*::*idom*) *a*
by *transfer simp*

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*)
by (*simp add*: *divide-fract-def*)

instance
proof
fix *q* :: *'a fract*
assume *q ≠ 0*
then show *inverse q * q = 1*
by (*cases q rule*: *Fract-cases-nonzero*)
(*simp-all add*: *fract-expand eq-fract mult.commute*)
next
fix *q r* :: *'a fract*
show *q div r = q * inverse r* **by** (*simp add*: *divide-fract-def*)
next
show *inverse 0 =* (*0*:: *'a fract*)
by (*simp add*: *fract-expand*) (*simp add*: *fract-collapse*)
qed

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 −
let *?le* $= \lambda a\ b\ c\ d.\ ((a * d) * (b * d) \leq (c * b) * (b * d))$
{
fix $a\ b\ c\ d\ x :: 'a$
assume *x*: $x \neq 0$
have *?le* $a\ b\ c\ d =$ *?le* $(a * x)\ (b * x)\ c\ d$
proof −
from *x* **have** $0 < x * x$
by (*auto simp add*: *zero-less-mult-iff*)
then have *?le* $a\ b\ c\ d =$
$((a * d) * (b * d) * (x * x) \leq (c * b) * (b * d) * (x * x))$
by (*simp add*: *mult-le-cancel-right*)
also have ... $=$ *?le* $(a * x)\ (b * x)\ c\ d$
by (*simp add*: *ac-simps*)
finally show *?thesis* **.**
qed
} note *le-factor* $=$ *this*

let $?D = b * d$ **and** $?D' = b' * d'$
from *neq* **have** *D*: $?D \neq 0$ **by** *simp*
from *neq* **have** $?D' \neq 0$ **by** *simp*
then have *?le* $a\ b\ c\ d =$ *?le* $(a * ?D')\ (b * ?D')\ c\ d$
by (*rule le-factor*)
also have ... $= ((a * b') * ?D * ?D' * d * d' \leq (c * d') * ?D * ?D' * b * b')$
by (*simp add*: *ac-simps*)
also have ... $= ((a' * b) * ?D * ?D' * d * d' \leq (c' * d) * ?D * ?D' * b * b')$
by (*simp only*: *eq1 eq2*)
also have ... $=$ *?le* $(a' * ?D)\ (b' * ?D)\ c'\ d'$
by (*simp add*: *ac-simps*)
also from *D* **have** ... $=$ *?le* $a'\ b'\ c'\ d'$
by (*rule le-factor* [*symmetric*])
finally show *?le* $a\ b\ c\ d =$ *?le* $a'\ b'\ c'\ d'$ **.**
qed

lift-definition *less-eq-fract* :: $'a\ fract \Rightarrow 'a\ fract \Rightarrow bool$
is $\lambda q\ r.\ (fst\ q * snd\ r) * (snd\ q * snd\ r) \leq (fst\ r * snd\ q) * (snd\ q * snd\ r)$
by (*clarsimp simp add*: *less-eq-fract-respect*)

definition *less-fract-def*: $z < (w::'a\ fract) \longleftrightarrow z \leq w \wedge \neg\ w \leq z$

lemma *le-fract* [*simp*]:
  ⟦ $b \neq 0$; $d \neq 0$ ⟧ ⟹ *Fract a b* ≤ *Fract c d* ⟷ $(a * d) * (b * d) \leq (c * b) * (b * d)$
  by *transfer simp*

lemma *less-fract* [*simp*]:
  ⟦ $b \neq 0$; $d \neq 0$ ⟧ ⟹ *Fract a b* < *Fract c d* ⟷ $(a * d) * (b * d) < (c * b) * (b * d)$
  by (*simp add*: *less-fract-def less-le-not-le ac-simps*)

instance
proof
  fix *q r s* :: *'a fract*
  assume $q \leq r$ **and** $r \leq s$
  then show $q \leq s$
  proof (*induct q*, *induct r*, *induct s*)
    fix *a b c d e f* :: *'a*
    assume *neq*: $b \neq 0$ $d \neq 0$ $f \neq 0$
    assume *1*: *Fract a b* ≤ *Fract c d*
    assume *2*: *Fract c d* ≤ *Fract e f*
    show *Fract a b* ≤ *Fract e f*
    proof −
      from *neq* **obtain** *bb*: $0 < b * b$ **and** *dd*: $0 < d * d$ **and** *ff*: $0 < f * f$
        by (*auto simp add*: *zero-less-mult-iff linorder-neq-iff*)
      have $(a * d) * (b * d) * (f * f) \leq (c * b) * (b * d) * (f * f)$
      proof −
        from *neq 1* **have** $(a * d) * (b * d) \leq (c * b) * (b * d)$
          by *simp*
        with *ff* **show** *?thesis* **by** (*simp add*: *mult-le-cancel-right*)
      qed
      also have ... = $(c * f) * (d * f) * (b * b)$
        by (*simp only*: *ac-simps*)
      also have ... ≤ $(e * d) * (d * f) * (b * b)$
      proof −
        from *neq 2* **have** $(c * f) * (d * f) \leq (e * d) * (d * f)$
          by *simp*
        with *bb* **show** *?thesis* **by** (*simp add*: *mult-le-cancel-right*)
      qed
      finally have $(a * f) * (b * f) * (d * d) \leq e * b * (b * f) * (d * d)$
        by (*simp only*: *ac-simps*)
      with *dd* **have** $(a * f) * (b * f) \leq (e * b) * (b * f)$
        by (*simp add*: *mult-le-cancel-right*)
      with *neq* **show** *?thesis* **by** *simp*
    qed
  qed
next
  fix *q r* :: *'a fract*
  assume $q \leq r$ **and** $r \leq q$
  then show $q = r$

```
  proof (induct q, induct r)
    fix a b c d :: 'a
    assume neq: b ≠ 0 d ≠ 0
    assume 1: Fract a b ≤ Fract c d
    assume 2: Fract c d ≤ Fract a b
    show Fract a b = Fract c d
    proof −
      from neq 1 have (a * d) * (b * d) ≤ (c * b) * (b * d)
        by simp
      also have ... ≤ (a * d) * (b * d)
      proof −
        from neq 2 have (c * b) * (d * b) ≤ (a * d) * (d * b)
          by simp
        then show ?thesis by (simp only: ac-simps)
      qed
      finally have (a * d) * (b * d) = (c * b) * (b * d) .
      moreover from neq have b * d ≠ 0 by simp
      ultimately have a * d = c * b by simp
      with neq show ?thesis by (simp add: eq-fract)
    qed
  qed
next
  fix q r :: 'a fract
  show q ≤ q
    by (induct q) simp
  show (q < r) = (q ≤ r ∧ ¬ r ≤ q)
    by (simp only: less-fract-def)
  show q ≤ r ∨ r ≤ q
    by (induct q, induct r)
       (simp add: mult.commute, rule linorder-linear)
qed

end

instantiation fract :: (linordered-idom) linordered-field
begin

definition abs-fract-def2:
  |q| = (if q < 0 then −q else (q::'a fract))

definition sgn-fract-def:
  sgn (q::'a fract) = (if q = 0 then 0 else if 0 < q then 1 else − 1)

theorem abs-fract [simp]: |Fract a b| = Fract |a| |b|
  unfolding abs-fract-def2 not-le [symmetric]
  by transfer (auto simp add: zero-less-mult-iff le-less)

instance proof
  fix q r s :: 'a fract
```

```
  assume q ≤ r
  then show s + q ≤ s + r
  proof (induct q, induct r, induct s)
    fix a b c d e f :: 'a
    assume neq: b ≠ 0 d ≠ 0 f ≠ 0
    assume le: Fract a b ≤ Fract c d
    show Fract e f + Fract a b ≤ Fract e f + Fract c d
    proof −
      let ?F = f * f from neq have F: 0 < ?F
        by (auto simp add: zero-less-mult-iff)
      from neq le have (a * d) * (b * d) ≤ (c * b) * (b * d)
        by simp
      with F have (a * d) * (b * d) * ?F * ?F ≤ (c * b) * (b * d) * ?F * ?F
        by (simp add: mult-le-cancel-right)
      with neq show ?thesis by (simp add: field-simps)
    qed
  qed
next
  fix q r s :: 'a fract
  assume q < r and 0 < s
  then show s * q < s * r
  proof (induct q, induct r, induct s)
    fix a b c d e f :: 'a
    assume neq: b ≠ 0 d ≠ 0 f ≠ 0
    assume le: Fract a b < Fract c d
    assume gt: 0 < Fract e f
    show Fract e f * Fract a b < Fract e f * Fract c d
    proof −
      let ?E = e * f and ?F = f * f
      from neq gt have 0 < ?E
        by (auto simp add: Zero-fract-def order-less-le eq-fract)
      moreover from neq have 0 < ?F
        by (auto simp add: zero-less-mult-iff)
      moreover from neq le have (a * d) * (b * d) < (c * b) * (b * d)
        by simp
      ultimately have (a * d) * (b * d) * ?E * ?F < (c * b) * (b * d) * ?E * ?F
        by (simp add: mult-less-cancel-right)
      with neq show ?thesis
        by (simp add: ac-simps)
    qed
  qed
qed (fact sgn-fract-def abs-fract-def2)+

end

instantiation fract :: (linordered-idom) distrib-lattice
begin

definition inf-fract-def:
```

```
  (inf :: 'a fract ⇒ 'a fract ⇒ 'a fract) = min

definition sup-fract-def:
  (sup :: 'a fract ⇒ 'a fract ⇒ 'a fract) = max

instance
  by standard (simp-all add: inf-fract-def sup-fract-def max-min-distrib2)

end

lemma fract-induct-pos [case-names Fract]:
  fixes P :: 'a::linordered-idom fract ⇒ bool
  assumes step: ⋀a b. 0 < b ⟹ P (Fract a b)
  shows P q
proof (cases q)
  case (Fract a b)
  {
    fix a b :: 'a
    assume b: b < 0
    have P (Fract a b)
    proof −
      from b have 0 < − b by simp
      then have P (Fract (− a) (− b))
        by (rule step)
      then show P (Fract a b)
        by (simp add: order-less-imp-not-eq [OF b])
    qed
  }
  with Fract show P q
    by (auto simp add: linorder-neq-iff step)
qed

lemma zero-less-Fract-iff: 0 < b ⟹ 0 < Fract a b ⟷ 0 < a
  by (auto simp add: Zero-fract-def zero-less-mult-iff)

lemma Fract-less-zero-iff: 0 < b ⟹ Fract a b < 0 ⟷ a < 0
  by (auto simp add: Zero-fract-def mult-less-0-iff)

lemma zero-le-Fract-iff: 0 < b ⟹ 0 ≤ Fract a b ⟷ 0 ≤ a
  by (auto simp add: Zero-fract-def zero-le-mult-iff)

lemma Fract-le-zero-iff: 0 < b ⟹ Fract a b ≤ 0 ⟷ a ≤ 0
  by (auto simp add: Zero-fract-def mult-le-0-iff)

lemma one-less-Fract-iff: 0 < b ⟹ 1 < Fract a b ⟷ b < a
  by (auto simp add: One-fract-def mult-less-cancel-right-disj)

lemma Fract-less-one-iff: 0 < b ⟹ Fract a b < 1 ⟷ a < b
  by (auto simp add: One-fract-def mult-less-cancel-right-disj)
```

lemma *one-le-Fract-iff*: $0 < b \Longrightarrow 1 \leq Fract\ a\ b \longleftrightarrow b \leq a$
by (*auto simp add*: *One-fract-def mult-le-cancel-right*)

lemma *Fract-le-one-iff*: $0 < b \Longrightarrow Fract\ a\ b \leq 1 \longleftrightarrow a \leq b$
by (*auto simp add*: *One-fract-def mult-le-cancel-right*)

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*: $cmod\ w \leq cmod\ (w + z) + norm\ z$
by (*metis add-diff-cancel norm-triangle-ineq4*)

9.2 Basic lemmas about polynomials

lemma *poly-bound-exists*:
fixes $p :: 'a::\{comm\text{-}semiring\text{-}0,real\text{-}normed\text{-}div\text{-}algebra\}\ poly$
shows $\exists m.\ m > 0 \wedge (\forall z.\ norm\ z \leq r \longrightarrow norm\ (poly\ p\ z) \leq m)$
proof (*induct p*)
case *0*
then show *?case* **by** (*rule exI*[**where** *x=1*]) *simp*
next
case (*pCons c cs*)
from *pCons.hyps* **obtain** *m* **where** *m*: $\forall z.\ norm\ z \leq r \longrightarrow norm\ (poly\ cs\ z) \leq m$
by *blast*
let *?k* = $1 + norm\ c + |r * m|$
have *kp*: $?k > 0$
using *abs-ge-zero*[*of r*∗*m*] *norm-ge-zero*[*of c*] **by** *arith*
have *norm* (*poly* (*pCons c cs*) *z*) ≤ *?k* **if** *H*: *norm z* ≤ *r* **for** *z*
proof −
from *m H* **have** *th*: *norm* (*poly cs z*) ≤ *m*
by *blast*
from *H* **have** *rp*: $r \geq 0$
using *norm-ge-zero*[*of z*] **by** *arith*
have *norm* (*poly* (*pCons c cs*) *z*) ≤ *norm c* + *norm* (*z* ∗ *poly cs z*)
using *norm-triangle-ineq*[*of c z*∗ *poly cs z*] **by** *simp*
also have ... ≤ *?k*
using *mult-mono*[*OF H th rp norm-ge-zero*[*of poly cs z*]]
by (*simp add*: *norm-mult*)

finally show *?thesis* **.**
qed
with *kp* **show** *?case* **by** *blast*
qed

Offsetting the variable in a polynomial gives another of same degree

definition *offset-poly* :: *'a::comm-semiring-0 poly* ⇒ *'a* ⇒ *'a poly*
where *offset-poly p h* = *fold-coeffs* (λ*a q*. *smult h q* + *pCons a q*) *p 0*

lemma *offset-poly-0*: *offset-poly 0 h* = *0*
by (*simp add*: *offset-poly-def*)

lemma *offset-poly-pCons*:
offset-poly (*pCons a p*) *h* =
smult h (*offset-poly p h*) + *pCons a* (*offset-poly p h*)
by (*cases p* = *0* ∧ *a* = *0*) (*auto simp add*: *offset-poly-def*)

lemma *offset-poly-single* [*simp*]: *offset-poly* [:*a*:] *h* = [:*a*:]
by (*simp add*: *offset-poly-pCons offset-poly-0*)

lemma *poly-offset-poly*: *poly* (*offset-poly p h*) *x* = *poly p* (*h* + *x*)
by (*induct p*) (*auto simp add*: *offset-poly-0 offset-poly-pCons algebra-simps*)

lemma *offset-poly-eq-0-lemma*: *smult c p* + *pCons a p* = *0* ⟹ *p* = *0*
by (*induct p arbitrary*: *a*) (*simp*, *force*)

lemma *offset-poly-eq-0-iff* [*simp*]: *offset-poly p h* = *0* ⟷ *p* = *0*
proof
show *offset-poly p h* = *0* ⟹ *p* = *0*
proof(*induction p*)
case *0*
then show *?case* **by** *blast*
next
case (*pCons a p*)
then show *?case*
by (*metis offset-poly-eq-0-lemma offset-poly-pCons offset-poly-single*)
qed
qed (*simp add*: *offset-poly-0*)

lemma *degree-offset-poly* [*simp*]: *degree* (*offset-poly p h*) = *degree p*
proof(*induction p*)
case *0*
then show *?case*
by (*simp add*: *offset-poly-0*)
next
case (*pCons a p*)
have *p* ≠ *0* ⟹ *degree* (*offset-poly* (*pCons a p*) *h*) = *Suc* (*degree p*)
by (*metis degree-add-eq-right degree-pCons-eq degree-smult-le le-imp-less-Suc offset-poly-eq-0-iff offset-poly-pCons pCons.IH*)

then show *?case*
by *simp*
qed

definition *psize p* = (*if p* = *0 then 0 else Suc* (*degree p*))

lemma *psize-eq-0-iff* [*simp*]: *psize p* = *0* ⟷ *p* = *0*
unfolding *psize-def* **by** *simp*

lemma *poly-offset*:
fixes *p* :: *'a*::*comm-ring-1 poly*
shows ∃ *q*. *psize q* = *psize p* ∧ (∀ *x*. *poly q x* = *poly p* (*a* + *x*))
by (*metis degree-offset-poly offset-poly-eq-0-iff poly-offset-poly psize-def*)

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
from *bz* **have** *bdd-above* (*Collect P*)
by (*force intro*: *less-imp-le*)
then show ∀ *y*. (∃ *x*. *P x* ∧ *y* < *x*) ⟷ *y* < *Sup* (*Collect P*)
using *ex bz* **by** (*subst less-cSup-iff*) *auto*
qed

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 −
obtain *x y* **where** *z*: *z* = *Complex x y*
by (*cases z*) *auto*
from *md z* **have** *xy*: $x^2 + y^2 = 1$
by (*simp add*: *cmod-def*)
have *False* **if** *cmod* (*z* + *1*) ≥ *1 cmod* (*z* − *1*) ≥ *1 cmod* (*z* + i) ≥ *1 cmod* (*z* − i) ≥ *1*
proof −
from *that z xy* **have** ∗: *2* ∗ *x* ≤ *1 2* ∗ *x* ≥ −*1 2* ∗ *y* ≤ *1 2* ∗ *y* ≥ −*1*
by (*simp-all add*: *cmod-def power2-eq-square algebra-simps*)
then have |*2* ∗ *x*| ≤ *1* |*2* ∗ *y*| ≤ *1*
by *simp-all*
then have $|2 * x|^2 \le 1^2$ $|2 * y|^2 \le 1^2$
by (*metis abs-square-le-1 one-power2 power2-abs*)+
with *xy* ∗ **show** *?thesis*
by (*smt* (*verit*, *best*) *four-x-squared square-le-1*)
qed

```
  then show ?thesis
    by force
qed
```

Hence we can always reduce modulus of $1 + b\ z\hat{}n$ if nonzero

```
lemma reduce-poly-simple:
  assumes b: b ≠ 0
    and n: n ≠ 0
  shows ∃z. cmod (1 + b * z^n) < 1
  using n
proof (induct n rule: nat-less-induct)
  fix n
  assume IH: ∀m<n. m ≠ 0 ⟶ (∃z. cmod (1 + b * z ^ m) < 1)
  assume n: n ≠ 0
  let ?P = λz n. cmod (1 + b * z ^ n) < 1
  show ∃z. ?P z n
  proof cases
    assume even n
    then obtain m where m: n = 2 * m and m ≠ 0 m < n
      using n by auto
    with IH obtain z where z: ?P z m
      by blast
    from z have ?P (csqrt z) n
      by (simp add: m power-mult)
    then show ?thesis ..
  next
    assume odd n
    then have ∃m. n = Suc (2 * m)
      by presburger+
    then obtain m where m: n = Suc (2 * m)
      by blast
    have 0: cmod (complex-of-real (cmod b) / b) = 1
      using b by (simp add: norm-divide)
    have ∃v. cmod (complex-of-real (cmod b) / b + v^n) < 1
    proof (cases cmod (complex-of-real (cmod b) / b + 1) < 1)
      case True
      then show ?thesis
        by (metis power-one)
    next
      case F1: False
      show ?thesis
      proof (cases cmod (complex-of-real (cmod b) / b - 1) < 1)
        case True
        with ‹odd n› show ?thesis
          by (metis add-uminus-conv-diff neg-one-odd-power)
      next
        case F2: False
        show ?thesis
        proof (cases cmod (complex-of-real (cmod b) / b + i) < 1)
```

```
        case T1: True
        show ?thesis
        proof (cases even m)
          case True
          with T1 show ?thesis
            by (rule-tac x=i in exI) (simp add: m power-mult)
        next
          case False
          with T1 show ?thesis
            by (rule-tac x=- i in exI) (simp add: m power-mult)
        qed
      next
        case False
        then have lt1: cmod (of-real (cmod b) / b - i) < 1
          using 0 F1 F2 unimodular-reduce-norm by blast
        show ?thesis
        proof (cases even m)
          case True
          with m lt1 show ?thesis
            by (rule-tac x=- i in exI) (simp add: power-mult)
        next
          case False
          with m lt1 show ?thesis
            by (rule-tac x=i in exI) (simp add: power-mult)
        qed
      qed
    qed
  qed
  then obtain v where v: cmod (complex-of-real (cmod b) / b + v^n) < 1
    by blast
  let ?w = v / complex-of-real (root n (cmod b))
  from odd-real-root-pow[OF ‹odd n›, of cmod b]
  have 1: ?w ^ n = v^n / complex-of-real (cmod b)
    by (simp add: power-divide of-real-power[symmetric])
  have 2:cmod (complex-of-real (cmod b) / b) = 1
    using b by (simp add: norm-divide)
  then have 3: cmod (complex-of-real (cmod b) / b) ≥ 0
    by simp
  have 4: cmod (complex-of-real (cmod b) / b) *
      cmod (1 + b * (v ^ n / complex-of-real (cmod b))) <
      cmod (complex-of-real (cmod b) / b) * 1
    apply (simp only: norm-mult[symmetric] distrib-left)
    using b v
    apply (simp add: 2)
    done
  show ?thesis
    by (metis 1 mult-left-less-imp-less[OF 4 3])
 qed
qed
```

Bolzano-Weierstrass type property for closed disc in complex plane.

lemma *metric-bound-lemma*: *cmod* $(x - y) \leq |Re\ x - Re\ y| + |Im\ x - Im\ y|$
using *real-sqrt-sum-squares-triangle-ineq*[*of Re x − Re y 0 0 Im x − Im y*]
unfolding *cmod-def* **by** *simp*

lemma *Bolzano-Weierstrass-complex-disc*:
assumes r: $\forall\, n.$ *cmod* $(s\ n) \leq r$
shows $\exists\, f\ z.$ *strict-mono* $(f :: nat \Rightarrow nat) \wedge (\forall\, e > 0.\ \exists\, N.\ \forall\, n \geq N.$ *cmod* $(s\ (f\ n) - z) < e)$
proof −
from *seq-monosub*[*of Re* ∘ *s*]
obtain *f* **where** *f*: *strict-mono f monoseq* $(\lambda n.$ *Re* $(s\ (f\ n)))$
unfolding *o-def* **by** *blast*
from *seq-monosub*[*of Im* ∘ *s* ∘ *f*]
obtain *g* **where** *g*: *strict-mono g monoseq* $(\lambda n.$ *Im* $(s\ (f\ (g\ n))))$
unfolding *o-def* **by** *blast*
let *?h* = *f* ∘ *g*
have $r \geq 0$
by (*meson norm-ge-zero order-trans r*)
have $\forall\, n.\ r + 1 \geq |Re\ (s\ n)|$
by (*smt* (*verit*, *ccfv-threshold*) *abs-Re-le-cmod r*)
then have *conv1*: *convergent* $(\lambda n.$ *Re* $(s\ (f\ n)))$
by (*metis Bseq-monoseq-convergent f*(*2*) *BseqI' real-norm-def*)
have $\forall\, n.\ r + 1 \geq |Im\ (s\ n)|$
by (*smt* (*verit*) *abs-Im-le-cmod r*)
then have *conv2*: *convergent* $(\lambda n.$ *Im* $(s\ (f\ (g\ n))))$
by (*metis Bseq-monoseq-convergent g*(*2*) *BseqI' real-norm-def*)

obtain *x* **where** x: $\forall\, r{>}0.\ \exists\, n0.\ \forall\, n{\geq}n0.\ |Re\ (s\ (f\ n)) - x| < r$
using *conv1*[*unfolded convergent-def*] *LIMSEQ-iff real-norm-def* **by** *metis*
obtain *y* **where** y: $\forall\, r{>}0.\ \exists\, n0.\ \forall\, n{\geq}n0.\ |Im\ (s\ (f\ (g\ n))) - y| < r$
using *conv2*[*unfolded convergent-def*] *LIMSEQ-iff real-norm-def* **by** *metis*
let *?w* = *Complex x y*
from *f*(*1*) *g*(*1*) **have** *hs*: *strict-mono ?h*
unfolding *strict-mono-def* **by** *auto*
have $\exists\, N.\ \forall\, n{\geq}N.$ *cmod* $(s\ (?h\ n) - ?w) < e$ **if** $e > 0$ **for** *e*
proof −
from *that* **have** *e2*: $e/2 > 0$
by *simp*
from *x y e2*
obtain *N1 N2* **where** *N1*: $\forall\, n{\geq}N1.\ |Re\ (s\ (f\ n)) - x| < e\ /\ 2$
and *N2*: $\forall\, n{\geq}N2.\ |Im\ (s\ (f\ (g\ n))) - y| < e\ /\ 2$
by *blast*
have *cmod* $(s\ (?h\ n) - ?w) < e$ **if** $n \geq N1 + N2$ **for** *n*
proof −
from *that* **have** *nN1*: $g\ n \geq N1$ **and** *nN2*: $n \geq N2$
using *seq-suble*[*OF g*(*1*), *of n*] **by** *arith*+
show *?thesis*
using *metric-bound-lemma*[*of s* (*f* (*g n*)) *?w*] *N1 N2 nN1 nN2* **by** *fastforce*

```
    qed
    then show ?thesis by blast
  qed
  with hs show ?thesis by blast
qed
```

Polynomial is continuous.

```
lemma poly-cont:
  fixes p :: 'a::{comm-semiring-0,real-normed-div-algebra} poly
  assumes ep: e > 0
  shows ∃ d >0. ∀ w. 0 < norm (w − z) ∧ norm (w − z) < d ⟶ norm (poly p
w − poly p z) < e
proof −
  obtain q where degree q = degree p and q: ⋀w. poly p w = poly q (w − z)
    by (metis add.commute degree-offset-poly diff-add-cancel poly-offset-poly)
  show ?thesis unfolding q
  proof (induct q)
    case 0
    then show ?case
      using ep by auto
  next
    case (pCons c cs)
    obtain m where m: m > 0 norm z ≤ 1 ⟹ norm (poly cs z) ≤ m for z
      using poly-bound-exists[of 1 cs] by blast
    with ep have em0: e/m > 0
      by (simp add: field-simps)
    obtain d where d: d > 0 d < 1 d < e / m
      by (meson em0 field-lbound-gt-zero zero-less-one)
    then have ⋀w. norm (w − z) < d ⟹ norm (w − z) ∗ norm (poly cs (w −
z)) < e
      by (smt (verit, del-insts) m mult-left-mono norm-ge-zero pos-less-divide-eq)
    with d show ?case
      by (force simp add: norm-mult)
  qed
qed
```

Hence a polynomial attains minimum on a closed disc in the complex plane.

```
lemma poly-minimum-modulus-disc: ∃ z. ∀ w. cmod w ≤ r ⟶ cmod (poly p z) ≤
cmod (poly p w)
proof −
  show ?thesis
  proof (cases r ≥ 0)
    case False
    then show ?thesis
      by (metis norm-ge-zero order.trans)
  next
    case True
    then have mth1: ∃ x z. cmod z ≤ r ∧ cmod (poly p z) = − x
      by (metis add.inverse-inverse norm-zero)
```

obtain s **where** s: $\forall y.\ (\exists x.\ (\exists z.\ cmod\ z \leq r \wedge cmod\ (poly\ p\ z) = -\ x) \wedge y < x) \longleftrightarrow y < s$
by (*smt* (*verit*, *del-insts*) *real-sup-exists*[*OF mth1*] *norm-zero zero-less-norm-iff*)

let *?m* = − *s*
have *s1*: $(\exists z.\ cmod\ z \leq r \wedge -\ (-\ cmod\ (poly\ p\ z)) < y) \longleftrightarrow ?m < y$ **for** *y*
by (*metis add.inverse-inverse minus-less-iff s*)
then have *s1m*: $\bigwedge z.\ cmod\ z \leq r \Longrightarrow cmod\ (poly\ p\ z) \geq ?m$
by *force*
have $\exists z.\ cmod\ z \leq r \wedge cmod\ (poly\ p\ z) < -\ s + 1\ /\ real\ (Suc\ n)$ **for** *n*
using *s1*[*of ?m + 1/real* (*Suc n*)] **by** *simp*
then obtain *g* **where** *g*: $\forall n.\ cmod\ (g\ n) \leq r\ \forall n.\ cmod\ (poly\ p\ (g\ n)) <$*?m + 1 /real*(*Suc n*)
by *metis*
from *Bolzano-Weierstrass-complex-disc*[*OF g*(*1*)]
obtain *f*::*nat* ⇒ *nat* **and** *z* **where** *fz*: *strict-mono f* $\forall e{>}0.\ \exists N.\ \forall n{\geq}N.\ cmod\ (g\ (f\ n) - z) < e$
by *blast*
{
fix *w*
assume *wr*: *cmod w* ≤ *r*
let *?e* = |*cmod* (*poly p z*) − *?m*|
{
assume *e*: *?e* > *0*
then have *e2*: *?e/2* > *0*
by *simp*
with *poly-cont* **obtain** *d*
where *d* > *0* **and** *d*: $\bigwedge w.\ 0{<}cmod\ (w - z) \wedge cmod(w - z) < d \longrightarrow$ *cmod*(*poly p w* − *poly p z*) < *?e/2*
by *blast*
have *1*: *cmod*(*poly p w* − *poly p z*) < *?e / 2* **if** *w*: *cmod* (*w* − *z*) < *d* **for** *w*
using *d*[*of w*] *w e* **by** (*cases w* = *z*) *simp-all*
from *fz*(*2*) ‹*d* > *0*› **obtain** *N1* **where** *N1*: $\forall n{\geq}N1.\ cmod\ (g\ (f\ n) - z) <$ *d*
by *blast*
from *reals-Archimedean2* **obtain** *N2* :: *nat* **where** *N2*: *2/?e* < *real N2*
by *blast*
have *2*: *cmod* (*poly p* (*g* (*f* (*N1* + *N2*))) − *poly p z*) < *?e/2*
using *N1 1* **by** *auto*
have *0*: *a* < *e2* ⟹ |*b* − *m*| < *e2* ⟹ *2* ∗ *e2* ≤ |*b* − *m*| + *a* ⟹ *False*
for *a b e2 m* :: *real*
by *arith*
from *seq-suble*[*OF fz*(*1*), *of N1* + *N2*]
have *00*: *?m* + *1* / *real* (*Suc* (*f* (*N1* + *N2*))) ≤ *?m* + *1* / *real* (*Suc* (*N1* + *N2*))
by (*simp add*: *frac-le*)
from *N2 e2 less-imp-inverse-less*[*of 2/?e real* (*Suc* (*N1* + *N2*))]
have *?e/2* > *1/ real* (*Suc* (*N1* + *N2*))
by (*simp add*: *inverse-eq-divide*)

```
      with  order-less-le-trans[OF - 00]
      have 1: |cmod (poly p (g (f (N1 + N2)))) − ?m| < ?e/2
        using g s1 by (smt (verit))
      with 0[OF 2] have False
        by (smt (verit) field-sum-of-halves norm-triangle-ineq3)
    }
    then have ?e = 0
      by auto
    with s1m[OF wr] have cmod (poly p z) ≤ cmod (poly p w)
      by simp
  }
  then show ?thesis by blast
 qed
qed
```

Nonzero polynomial in z goes to infinity as z does.

```
lemma poly-infinity:
  fixes p:: 'a::{comm-semiring-0,real-normed-div-algebra} poly
  assumes ex: p ≠ 0
  shows ∃ r. ∀ z. r ≤ norm z ⟶ d ≤ norm (poly (pCons a p) z)
  using ex
proof (induct p arbitrary: a d)
  case 0
  then show ?case by simp
next
  case (pCons c cs a d)
  show ?case
  proof (cases cs = 0)
    case False
   with pCons.hyps obtain r where r: ∀ z. r ≤ norm z ⟶ d + norm a ≤ norm
(poly (pCons c cs) z)
      by blast
    let ?r = 1 + |r|
    have d ≤ norm (poly (pCons a (pCons c cs)) z) if 1 + |r| ≤ norm z for z
    proof −
      have d ≤ norm(z * poly (pCons c cs) z) − norm a
        by (smt (verit, best) norm-ge-zero mult-less-cancel-right2 norm-mult r that)
      with norm-diff-ineq add.commute
      show ?thesis
        by (metis order.trans poly-pCons)
    qed
    then show ?thesis by blast
  next
    case True
    have d ≤ norm (poly (pCons a (pCons c cs)) z)
      if (|d| + norm a) / norm c ≤ norm z for z :: 'a
    proof −
      have |d| + norm a ≤ norm (z * c)
      by (metis that True norm-mult pCons.hyps(1) pos-divide-le-eq zero-less-norm-iff)
```

```
    also have ... ≤ norm (a + z * c) + norm a
      by (simp add: add.commute norm-add-leD)
    finally show ?thesis
      using True by auto
  qed
  then show ?thesis by blast
 qed
qed
```

Hence polynomial's modulus attains its minimum somewhere.

```
lemma poly-minimum-modulus: ∃ z.∀ w. cmod (poly p z) ≤ cmod (poly p w)
proof (induct p)
 case 0
 then show ?case by simp
next
 case (pCons c cs)
 show ?case
 proof (cases cs = 0)
  case False
  from poly-infinity[OF False, of cmod (poly (pCons c cs) 0) c]
  obtain r where r: cmod (poly (pCons c cs) 0) ≤ cmod (poly (pCons c cs) z)
    if r ≤ cmod z for z
    by blast
  from poly-minimum-modulus-disc[of |r| pCons c cs] show ?thesis
    by (smt (verit, del-insts) order.trans linorder-linear r)
 qed (use pCons.hyps in auto)
qed
```

Constant function (non-syntactic characterization).

```
definition constant f ⟷ (∀ x y. f x = f y)

lemma nonconstant-length: ¬ constant (poly p) ⟹ psize p ≥ 2
 by (induct p) (auto simp: constant-def psize-def)

lemma poly-replicate-append: poly (monom 1 n * p) (x::'a::comm-ring-1) = x^n
* poly p x
 by (simp add: poly-monom)
```

Decomposition of polynomial, skipping zero coefficients after the first.

```
lemma poly-decompose-lemma:
 assumes nz: ¬ (∀ z. z ≠ 0 ⟶ poly p z = (0::'a::idom))
 shows ∃ k a q. a ≠ 0 ∧ Suc (psize q + k) = psize p ∧ (∀ z. poly p z = z^k * poly
(pCons a q) z)
 unfolding psize-def
 using nz
proof (induct p)
 case 0
 then show ?case by simp
next
```

```
  case (pCons c cs)
  show ?case
  proof (cases c = 0)
    case True
    from pCons.hyps pCons.prems True show ?thesis
      apply auto
      apply (rule-tac x=k+1 in exI)
      apply (rule-tac x=a in exI)
      apply clarsimp
      apply (rule-tac x=q in exI)
      apply auto
      done
  qed force
qed

lemma poly-decompose:
  fixes p :: 'a::idom poly
  assumes nc: ¬ constant (poly p)
  shows ∃ k a q. a ≠ 0 ∧ k ≠ 0 ∧
               psize q + k + 1 = psize p ∧
              (∀ z. poly p z = poly p 0 + z^k * poly (pCons a q) z)
  using nc
proof (induct p)
  case 0
  then show ?case
    by (simp add: constant-def)
next
  case (pCons c cs)
  have ¬ (∀ z. z ≠ 0 ⟶ poly cs z = 0)
    by (smt (verit) constant-def mult-eq-0-iff pCons.prems poly-pCons)
  from poly-decompose-lemma[OF this]
  obtain k a q where *: a ≠ 0 ∧
     Suc (psize q + k) = psize cs ∧ (∀ z. poly cs z = z ^ k * poly (pCons a q) z)
    by blast
  then have psize q + k + 2 = psize (pCons c cs)
    by (auto simp add: psize-def split: if-splits)
  then show ?case
    using * by force
qed
```

Fundamental theorem of algebra

```
theorem fundamental-theorem-of-algebra:
  assumes nc: ¬ constant (poly p)
  shows ∃ z::complex. poly p z = 0
  using nc
proof (induct psize p arbitrary: p rule: less-induct)
  case less
  let ?p = poly p
  let ?ths = ∃ z. ?p z = 0
```

```
  from nonconstant-length[OF less(2)] have n2: psize p ≥ 2 .
  from poly-minimum-modulus obtain c where c: ∀ w. cmod (?p c) ≤ cmod (?p
w)
    by blast

  show ?ths
  proof (cases ?p c = 0)
    case True
    then show ?thesis by blast
  next
    case False
    obtain q where q: psize q = psize p ∀ x. poly q x = ?p (c + x)
      using poly-offset[of p c] by blast
    then have qnc: ¬ constant (poly q)
      by (metis (no-types, opaque-lifting) add.commute constant-def diff-add-cancel
less.prems)
    from q(2) have pqc0: ?p c = poly q 0
      by simp
    from c pqc0 have cq0: ∀ w. cmod (poly q 0) ≤ cmod (?p w)
      by simp
    let ?a0 = poly q 0
    from False pqc0 have a00: ?a0 ≠ 0
      by simp
    from a00 have qr: ∀ z. poly q z = poly (smult (inverse ?a0) q) z * ?a0
      by simp
    let ?r = smult (inverse ?a0) q
    have lgqr: psize q = psize ?r
      by (simp add: a00 psize-def)
    have rnc: ¬ constant (poly ?r)
      using constant-def qnc qr by fastforce
    have r01: poly ?r 0 = 1
      by (simp add: a00)
    have mrmq-eq: cmod (poly ?r w) < 1 ⟷ cmod (poly q w) < cmod ?a0 for w
    by (smt (verit, del-insts) a00 mult-less-cancel-right2 norm-mult qr zero-less-norm-iff)
    from poly-decompose[OF rnc] obtain k a s where
      kas: a ≠ 0 k ≠ 0 psize s + k + 1 = psize ?r
        ∀ z. poly ?r z = poly ?r 0 + z^k* poly (pCons a s) z by blast
    have ∃ w. cmod (poly ?r w) < 1
    proof (cases psize p = k + 1)
      case True
      with kas q have s0: s = 0
        by (simp add: lgqr)
      with reduce-poly-simple kas show ?thesis
        by (metis mult.commute mult.right-neutral poly-1 poly-smult r01 smult-one)
    next
      case False note kn = this
      from kn kas(3) q(1) lgqr have k1n: k + 1 < psize p
        by simp
```

have *01*: ¬ *constant* (*poly* (*pCons 1* (*monom a* (*k* − *1*))))
unfolding *constant-def poly-pCons poly-monom*
by (*metis add-cancel-left-right kas*(*1*) *mult.commute mult-cancel-right2 power-one*)
have *02*: *k* + *1* = *psize* (*pCons 1* (*monom a* (*k* − *1*)))
using *kas* **by** (*simp add*: *psize-def degree-monom-eq*)
from *less*(*1*) [*OF - 01*] *k1n 02*
obtain *w* **where** *w*: *1* + *w*^*k* ∗ *a* = *0*
by (*metis kas*(*2*) *mult.commute mult.left-commute poly-monom poly-pCons power-eq-if*)
from *poly-bound-exists*[*of cmod w s*] **obtain** *m* **where**
m: *m* > *0* ∀ *z*. *cmod z* ≤ *cmod w* ⟶ *cmod* (*poly s z*) ≤ *m* **by** *blast*
have *w* ≠ *0*
using *kas*(*2*) *w* **by** (*auto simp add*: *power-0-left*)
from *w* **have** *wm1*: *w*^*k* ∗ *a* = − *1*
by (*simp add*: *add-eq-0-iff*)
have *inv0*: *0* < *inverse* (*cmod w* ^ (*k* + *1*) ∗ *m*)
by (*simp add*: ‹*w* ≠ *0*› *m*(*1*))
with *field-lbound-gt-zero*[*OF zero-less-one*] **obtain** *t* **where**
t: *t* > *0 t* < *1 t* < *inverse* (*cmod w* ^ (*k* + *1*) ∗ *m*) **by** *blast*
let *?ct* = *complex-of-real t*
let *?w* = *?ct* ∗ *w*
have *1* + *?w*^*k* ∗ (*a* + *?w* ∗ *poly s ?w*) = *1* + *?ct*^*k* ∗ (*w*^*k* ∗ *a*) + *?w*^*k* ∗ *?w* ∗ *poly s ?w*
using *kas*(*1*) **by** (*simp add*: *algebra-simps power-mult-distrib*)
also have ... = *complex-of-real* (*1* − *t*^*k*) + *?w*^*k* ∗ *?w* ∗ *poly s ?w*
unfolding *wm1* **by** *simp*
finally have *cmod* (*1* + *?w*^*k* ∗ (*a* + *?w* ∗ *poly s ?w*)) =
cmod (*complex-of-real* (*1* − *t*^*k*) + *?w*^*k* ∗ *?w* ∗ *poly s ?w*)
by *metis*
with *norm-triangle-ineq*[*of complex-of-real* (*1* − *t*^*k*) *?w*^*k* ∗ *?w* ∗ *poly s ?w*]
have *11*: *cmod* (*1* + *?w*^*k* ∗ (*a* + *?w* ∗ *poly s ?w*)) ≤ |*1* − *t*^*k*| + *cmod* (*?w*^*k* ∗ *?w* ∗ *poly s ?w*)
unfolding *norm-of-real* **by** *simp*
have *ath*: ⋀*x t*::*real*. *0* ≤ *x* ⟹ *x* < *t* ⟹ *t* ≤ *1* ⟹ |*1* − *t*| + *x* < *1*
by *arith*
have *tw*: *cmod ?w* ≤ *cmod w*
by (*smt* (*verit*) *mult-le-cancel-right2 norm-ge-zero norm-mult norm-of-real t*)
have *t* ∗ (*cmod w* ^ (*k* + *1*) ∗ *m*) < *1*
by (*smt* (*verit*, *best*) *inv0 inverse-positive-iff-positive left-inverse mult-strict-right-mono t*(*3*))
with *zero-less-power*[*OF t*(*1*), *of k*] **have** *30*: *t*^*k* ∗ (*t*∗ (*cmod w* ^ (*k* + *1*) ∗ *m*)) < *t*^*k*
by *simp*
have *cmod* (*?w*^*k* ∗ *?w* ∗ *poly s ?w*) = *t*^*k* ∗ (*t*∗ (*cmod w* ^ (*k* + *1*) ∗ *cmod* (*poly s ?w*)))
using ‹*w* ≠ *0*› *t*(*1*) **by** (*simp add*: *algebra-simps norm-power norm-mult*)
with *30* **have** *120*: *cmod* (*?w*^*k* ∗ *?w* ∗ *poly s ?w*) < *t*^*k*

by (*smt* (*verit*, *ccfv-SIG*) *m*(*2*) *mult-left-mono norm-ge-zero t*(*1*) *tw zero-le-power*)
from *power-strict-mono*[*OF t*(*2*), *of k*] *t*(*1*) *kas*(*2*) **have** *121*: *t*^*k* ≤ *1*
by *auto*
from *ath*[*OF norm-ge-zero*[*of ?w*^*k* ∗ *?w* ∗ *poly s ?w*] *120 121*]
show *?thesis*
by (*smt* (*verit*) *11 kas*(*4*) *poly-pCons r01*)
qed
with *cq0 q*(*2*) **show** *?thesis*
by (*smt* (*verit*) *mrmq-eq*)
qed
qed

Alternative version with a syntactic notion of constant polynomial.

lemma *fundamental-theorem-of-algebra-alt*:
assumes *nc*: ¬ (∃ *a l*. *a* ≠ *0* ∧ *l* = *0* ∧ *p* = *pCons a l*)
shows ∃ *z*. *poly p z* = (*0*::*complex*)
proof (*rule ccontr*)
assume *N*: ∄ *z*. *poly p z* = *0*
then have ¬ *constant* (*poly p*)
unfolding *constant-def*
by (*metis* (*no-types*, *opaque-lifting*) *nc poly-pcompose pcompose-0*′ *pcompose-const poly-0-coeff-0*
poly-all-0-iff-0 poly-diff right-minus-eq)
then show *False*
using *N fundamental-theorem-of-algebra* **by** *blast*
qed

9.4 Nullstellensatz, degrees and divisibility of polynomials

lemma *nullstellensatz-lemma*:
fixes *p* :: *complex poly*
assumes ∀ *x*. *poly p x* = *0* ⟶ *poly q x* = *0*
and *degree p* = *n*
and *n* ≠ *0*
shows *p dvd* (*q* ^ *n*)
using *assms*
proof (*induct n arbitrary*: *p q rule*: *nat-less-induct*)
fix *n* :: *nat*
fix *p q* :: *complex poly*
assume *IH*: ∀ *m*<*n*. ∀ *p q*.
(∀ *x*. *poly p x* = (*0*::*complex*) ⟶ *poly q x* = *0*) ⟶
degree p = *m* ⟶ *m* ≠ *0* ⟶ *p dvd* (*q* ^ *m*)
and *pq0*: ∀ *x*. *poly p x* = *0* ⟶ *poly q x* = *0*
and *dpn*: *degree p* = *n*
and *n0*: *n* ≠ *0*
from *dpn n0* **have** *pne*: *p* ≠ *0* **by** *auto*
show *p dvd* (*q* ^ *n*)
proof (*cases* ∃ *a*. *poly p a* = *0*)

```
    case True
    then obtain a where a: poly p a = 0 ..
    have ?thesis if oa: order a p ≠ 0
    proof −
      let ?op = order a p
      from pne have ap: ([:− a, 1:] ^ ?op) dvd p ¬ [:− a, 1:] ^ (Suc ?op) dvd p
        using order by blast+
      note oop = order-degree[OF pne, unfolded dpn]
      show ?thesis
      proof (cases q = 0)
        case True
        with n0 show ?thesis by (simp add: power-0-left)
      next
        case False
        from pq0[rule-format, OF a, unfolded poly-eq-0-iff-dvd]
        obtain r where r: q = [:− a, 1:] * r by (rule dvdE)
        from ap(1) obtain s where s: p = [:− a, 1:] ^ ?op * s
          by (rule dvdE)
        have sne: s ≠ 0
          using s pne by auto
        show ?thesis
        proof (cases degree s = 0)
          case True
          then obtain k where kpn: s = [:k:]
            by (cases s) (auto split: if-splits)
          from sne kpn have k: k ≠ 0 by simp
          let ?w = ([:1/k:] * ([:−a,1:] ^ (n − ?op))) * (r ^ n)
          have q^n = [:− a, 1:] ^ n * r ^ n
            using power-mult-distrib r by blast
          also have ... = [:− a, 1:] ^ order a p * [:k:] * ([:1 / k:] * [:− a, 1:] ^ (n
− order a p) * r ^ n)
            using k oop [of a] by (simp flip: power-add)
          also have ... = p * ?w
            by (metis s kpn)
          finally show ?thesis
            unfolding dvd-def by blast
        next
          case False
          with sne dpn s oa have dsn: degree s < n
          by (metis add-diff-cancel-right' degree-0 degree-linear-power degree-mult-eq
gr0I zero-less-diff)
          have poly r x = 0 if h: poly s x = 0 for x
          proof −
            have x ≠ a
                by (metis ap(2) dvd-refl mult-dvd-mono poly-eq-0-iff-dvd power-Suc
power-commutes s that)
            moreover have poly p x = 0
              by (metis (no-types) mult-eq-0-iff poly-mult s that)
            ultimately show ?thesis
```

```
        using pq0 r by auto
      qed
      with False IH dsn obtain u where u: r ^ (degree s) = s * u
        by blast
      then have u': ⋀x. poly s x * poly u x = poly r x ^ degree s
        by (simp only: poly-mult[symmetric] poly-power[symmetric])
      have q^n = [:- a, 1:] ^ n * r ^ n
        using power-mult-distrib r by blast
      also have ... = [:- a, 1:] ^ order a p * (s * u * ([:- a, 1:] ^ (n - order
a p) * r ^ (n - degree s)))
        by (smt (verit, del-insts) s u mult-ac power-add add-diff-cancel-right'
degree-linear-power degree-mult-eq dpn mult-zero-left)
      also have ... = p * (u * ([:-a,1:] ^ (n - ?op))) * (r ^ (n - degree s))
        using s by force
      finally show ?thesis
        unfolding dvd-def by auto
    qed
   qed
  qed
  then show ?thesis
   using a order-root pne by blast
 next
  case False
  then show ?thesis
   using dpn n0 fundamental-theorem-of-algebra-alt[of p]
   by fastforce
 qed
qed

lemma nullstellensatz-univariate:
 (∀x. poly p x = (0::complex) ⟶ poly q x = 0) ⟷
  p dvd (q ^ (degree p)) ∨ (p = 0 ∧ q = 0)
proof -
 consider p = 0 | p ≠ 0 degree p = 0 | n where p ≠ 0 degree p = Suc n
  by (cases degree p) auto
 then show ?thesis
 proof cases
  case p: 1
  then have (∀x. poly p x = (0::complex) ⟶ poly q x = 0) ⟷ q = 0
   by (auto simp add: poly-all-0-iff-0)
  with p show ?thesis
   by force
 next
  case dp: 2
  then show ?thesis
   by (meson dvd-trans is-unit-iff-degree poly-eq-0-iff-dvd unit-imp-dvd)
 next
  case dp: 3
  have False if p dvd (q ^ (Suc n)) poly p x = 0 poly q x ≠ 0 for x
```

by (*metis dvd-trans poly-eq-0-iff-dvd poly-power power-eq-0-iff that*)
with *dp nullstellensatz-lemma*[*of p q degree p*] **show** *?thesis*
by *auto*
qed
qed

Useful lemma

lemma *constant-degree*:
fixes *p* :: *'a*::{*idom*,*ring-char-0*} *poly*
shows *constant* (*poly p*) ⟷ *degree p* = *0* (**is** *?lhs* = *?rhs*)
proof
show *?rhs* **if** *?lhs*
proof −
from *that*[*unfolded constant-def*, *rule-format*, *of - 0*]
have *poly p* = *poly* [:*poly p 0*:]
by *auto*
then show *?thesis*
by (*metis degree-pCons-0 poly-eq-poly-eq-iff*)
qed
show *?lhs* **if** *?rhs*
unfolding *constant-def*
by (*metis degree-eq-zeroE pcompose-const poly-0 poly-pcompose that*)
qed

lemma *complex-poly-decompose*:
smult (*lead-coeff p*) (∏ *z*|*poly p z* = *0*. [:−*z*, *1*:] ^ *order z p*) = (*p* :: *complex poly*)
proof (*induction p rule*: *poly-root-order-induct*)
case (*no-roots p*)
show *?case*
proof (*cases degree p* = *0*)
case *False*
hence ¬*constant* (*poly p*) **by** (*subst constant-degree*)
with *fundamental-theorem-of-algebra* **and** *no-roots* **show** *?thesis* **by** *blast*
qed (*auto elim*!: *degree-eq-zeroE*)
next
case (*root p x n*)
from *root* **have** ∗: {*z*. *poly* ([:− *x*, *1*:] ^ *n* ∗ *p*) *z* = *0*} = *insert x* {*z*. *poly p z* = *0*}
by *auto*
have *smult* (*lead-coeff* ([:−*x*, *1*:] ^ *n* ∗ *p*))
(∏ *z*|*poly* ([:−*x*,*1*:] ^ *n* ∗ *p*) *z* = *0*. [:−*z*, *1*:] ^ *order z* ([:− *x*, *1*:] ^ *n* ∗ *p*)) =
[:− *x*, *1*:] ^ *order x* ([:− *x*, *1*:] ^ *n* ∗ *p*) ∗
smult (*lead-coeff p*) (∏ *z*∈{*z*. *poly p z* = *0*}. [:− *z*, *1*:] ^ *order z* ([:− *x*, *1*:] ^ *n* ∗ *p*))
by (*subst* ∗, *subst prod.insert*)
(*insert root*, *auto intro*: *poly-roots-finite simp*: *mult-ac lead-coeff-mult lead-coeff-power*)
also have *order x* ([:− *x*, *1*:] ^ *n* ∗ *p*) = *n*
using *root* **by** (*subst order-mult*) (*auto simp*: *order-power-n-n order-0I*)

```
  also have (∏ z∈{z. poly p z = 0}. [:− z, 1:] ^ order z ([:− x, 1:] ^ n * p)) =
              (∏ z∈{z. poly p z = 0}. [:− z, 1:] ^ order z p)
  proof (intro prod.cong refl, goal-cases)
    case (1 y)
    with root have order y ([:−x,1:] ^ n) = 0 by (intro order-0I) auto
    thus ?case using root by (subst order-mult) auto
  qed
  also note root.IH
  finally show ?case .
qed simp-all

instance complex :: alg-closed-field
  by standard (use fundamental-theorem-of-algebra constant-degree neq0-conv in
blast)

lemma size-proots-complex: size (proots (p :: complex poly)) = degree p
proof (cases p = 0)
  case [simp]: False
  show size (proots p) = degree p
    by (subst (1 2) complex-poly-decompose [symmetric])
       (simp add: proots-prod proots-power degree-prod-sum-eq degree-power-eq)
qed auto

lemma complex-poly-decompose-multiset:
  smult (lead-coeff p) (∏ x∈#proots p. [:−x, 1:]) = (p :: complex poly)
proof (cases p = 0)
  case False
  hence (∏ x∈#proots p. [:−x, 1:]) = (∏ x | poly p x = 0. [:−x, 1:] ^ order x p)
    by (subst image-prod-mset-multiplicity) simp-all
  also have smult (lead-coeff p) ... = p
    by (rule complex-poly-decompose)
  finally show ?thesis .
qed auto

lemma complex-poly-decompose':
  obtains root where smult (lead-coeff p) (∏ i<degree p. [:−root i, 1:]) = (p ::
complex poly)
proof −
  obtain roots where roots: mset roots = proots p
    using ex-mset by blast

  have p = smult (lead-coeff p) (∏ x∈#proots p. [:−x, 1:])
    by (rule complex-poly-decompose-multiset [symmetric])
  also have (∏ x∈#proots p. [:−x, 1:]) = (∏ x←roots. [:−x, 1:])
    by (subst prod-mset-prod-list [symmetric]) (simp add: roots)
  also have ... = (∏ i<length roots. [:−roots ! i, 1:])
    by (subst prod.list-conv-set-nth) (auto simp: atLeast0LessThan)
  finally have eq: p = smult (lead-coeff p) (∏ i<length roots. [:−roots ! i, 1:]) .
  also have [simp]: degree p = length roots
```

using *roots* **by** (*subst eq*) (*auto simp*: *degree-prod-sum-eq*)
finally show *?thesis* **by** (*intro that*[*of* $\lambda i.$ *roots* ! *i*]) *auto*
qed

lemma *complex-poly-decompose-rsquarefree*:
assumes *rsquarefree p*
shows *smult* (*lead-coeff p*) ($\prod$ *z*|*poly p z* = *0*. [:−*z*, *1*:]) = (*p* :: *complex poly*)
proof (*cases p* = *0*)
case *False*
have ($\prod$ *z*|*poly p z* = *0*. [:−*z*, *1*:]) = ($\prod$ *z*|*poly p z* = *0*. [:−*z*, *1*:] ^ *order z p*)
using *assms False* **by** (*intro prod.cong*) (*auto simp*: *rsquarefree-root-order*)
also have *smult* (*lead-coeff p*) ... = *p*
by (*rule complex-poly-decompose*)
finally show *?thesis* **.**
qed *auto*

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*
by *simp-all*

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*
by *simp-all*

lemma *mpoly-sub-conv*:
fixes *x* :: *'a*::*comm-ring-1*
shows *poly p x* − *poly q x* = *poly p x* + −*1* ∗ *poly q x*
by *simp*

lemma *poly-pad-rule*: *poly p x* = *0* ⟹ *poly* (*pCons 0 p*) *x* = *0*
by *simp*

lemma *poly-cancel-eq-conv*:
fixes *x* :: *'a*::*field*
shows *x* = *0* ⟹ *a* ≠ *0* ⟹ *y* = *0* ⟷ *a* ∗ *y* − *b* ∗ *x* = *0*
by *auto*

lemma *poly-divides-pad-rule*:
fixes *p*:: (*'a*::*comm-ring-1*) *poly*
assumes *pq*: *p dvd q*
shows *p dvd* (*pCons 0 q*)
by (*metis add-0 dvd-def mult-pCons-right pq smult-0-left*)

lemma *poly-divides-conv0*:
fixes *p*:: *'a*::*field poly*
assumes *lgpq*: *degree q* < *degree p* **and** *lq*: *p* ≠ *0*

shows *p dvd q* $\longleftrightarrow$ *q = 0*
using *lgpq mod-poly-less* **by** *fastforce*

lemma *poly-divides-conv1*:
fixes *p* :: *'a*::*field poly*
assumes *a0*: *a* $\neq$ *0*
and *pp'*: *p dvd p'*
and *qrp'*: *smult a q − p' = r*
shows *p dvd q* $\longleftrightarrow$ *p dvd r*
by (*metis a0 diff-add-cancel dvd-add-left-iff dvd-smult-iff pp' qrp'*)

lemma *basic-cqe-conv1*:
(∃ *x*. *poly p x = 0* ∧ *poly 0 x* $\neq$ *0*) $\longleftrightarrow$ *False*
(∃ *x*. *poly 0 x* $\neq$ *0*) $\longleftrightarrow$ *False*
(∃ *x*. *poly* [:*c*:] *x* $\neq$ *0*) $\longleftrightarrow$ *c* $\neq$ *0*
(∃ *x*. *poly 0 x = 0*) $\longleftrightarrow$ *True*
(∃ *x*. *poly* [:*c*:] *x = 0*) $\longleftrightarrow$ *c = 0*
by *simp-all*

lemma *basic-cqe-conv2*:
assumes *l*: *p* $\neq$ *0*
shows ∃ *x*. *poly* (*pCons a* (*pCons b p*)) *x* = (*0*::*complex*)
by (*meson fundamental-theorem-of-algebra-alt l pCons-eq-0-iff pCons-eq-iff*)

lemma *basic-cqe-conv-2b*: (∃ *x*. *poly p x* $\neq$ (*0*::*complex*)) $\longleftrightarrow$ *p* $\neq$ *0*
by (*metis poly-all-0-iff-0*)

lemma *basic-cqe-conv3*:
fixes *p q* :: *complex poly*
assumes *l*: *p* $\neq$ *0*
shows (∃ *x*. *poly* (*pCons a p*) *x = 0* ∧ *poly q x* $\neq$ *0*) $\longleftrightarrow$ ¬ (*pCons a p*) *dvd* (*q* ^ *psize p*)
by (*metis degree-pCons-eq-if l nullstellensatz-univariate pCons-eq-0-iff psize-def*)

lemma *basic-cqe-conv4*:
fixes *p q* :: *complex poly*
assumes *h*: ⋀*x*. *poly* (*q* ^ *n*) *x = poly r x*
shows *p dvd* (*q* ^ *n*) $\longleftrightarrow$ *p dvd r*
by (*metis* (*no-types*) *basic-cqe-conv-2b h poly-diff right-minus-eq*)

lemma *poly-const-conv*:
fixes *x* :: *'a*::*comm-ring-1*
shows *poly* [:*c*:] *x = y* $\longleftrightarrow$ *c = y*
by *simp*

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*
using *group-closure.base* [*of 0 S*] **by** *simp*

lemma *group-closure-minus-iff* [*simp*]:
− *s* ∈ *group-closure S* ⟷ *s* ∈ *group-closure S*
using *group-closure.diff* [*of 0 S s*] *group-closure.diff* [*of 0 S* − *s*] **by** *auto*

lemma *group-closure-add*:
s + *t* ∈ *group-closure S* **if** *s* ∈ *group-closure S* **and** *t* ∈ *group-closure S*
using *that group-closure.diff* [*of s S* − *t*] **by** *auto*

lemma *group-closure-empty* [*simp*]:
group-closure {} = {*0*}
by (*rule ccontr*) (*auto elim*: *group-closure.induct*)

lemma *group-closure-insert-zero* [*simp*]:
group-closure (*insert 0 S*) = *group-closure S*
by (*auto elim*: *group-closure.induct intro*: *group-closure.intros*)

end

context *comm-ring-1*
begin

lemma *group-closure-scalar-mult-left*:
of-nat n ∗ *s* ∈ *group-closure S* **if** *s* ∈ *group-closure S*
using *that* **by** (*induction n*) (*auto simp add*: *algebra-simps intro*: *group-closure-add*)

lemma *group-closure-scalar-mult-right*:
s ∗ *of-nat n* ∈ *group-closure S* **if** *s* ∈ *group-closure S*
using *that group-closure-scalar-mult-left* [*of s S n*] **by** (*simp add*: *ac-simps*)

end

lemma *group-closure-abs-iff* [*simp*]:
|*s*| ∈ *group-closure S* ⟷ *s* ∈ *group-closure S* **for** *s* :: *int*
by (*simp add*: *abs-if*)

lemma *group-closure-mult-left*:
  $s * t \in$ *group-closure* S **if** $s \in$ *group-closure* S **for** $s\ t ::$ *int*
proof −
  from *that group-closure-scalar-mult-right* [*of s S nat* $|t|$]
    have $s *$ *int* (*nat* $|t|) \in$ *group-closure* S
    by (*simp only*:)
  then show *?thesis*
    by (*cases* $t \geq 0$) *simp-all*
qed

lemma *group-closure-mult-right*:
  $s * t \in$ *group-closure* S **if** $t \in$ *group-closure* S **for** $s\ t ::$ *int*
  using *that group-closure-mult-left* [*of t S s*] **by** (*simp add*: *ac-simps*)

context *idom*
begin

lemma *group-closure-mult-all-eq*:
  group-closure (*times k ‘ S*) = *times k ‘ group-closure S*
proof (*rule*; *rule*)
  fix *s*
  have ∗: $k * a + k * b = k * (a + b)$
    $k * a - k * b = k * (a - b)$ **for** $a\ b$
    by (*simp-all add*: *algebra-simps*)
  assume $s \in$ *group-closure* (*times k ‘ S*)
  then show $s \in$ *times k ‘ group-closure S*
   by *induction* (*auto simp add*: ∗ *image-iff intro*: *group-closure.base group-closure.diff bexI* [*of - 0*])
next
  fix *s*
  assume $s \in$ *times k ‘ group-closure S*
  then obtain *r* **where** *r*: $r \in$ *group-closure* S **and** *s*: $s = k * r$
    by *auto*
  from *r* **have** $k * r \in$ *group-closure* (*times k ‘ S*)
   by (*induction arbitrary*: *s*) (*auto simp add*: *algebra-simps intro*: *group-closure.intros*)
  with *s* **show** $s \in$ *group-closure* (*times k ‘ S*)
    by *simp*
qed

end

lemma *Gcd-group-closure-eq-Gcd*:
  Gcd (*group-closure S*) = *Gcd S* **for** $S ::$ *int set*
proof (*rule associated-eqI*)
  have *Gcd S dvd s* **if** $s \in$ *group-closure* S **for** *s*
    using *that* **by** *induction auto*
  then show *Gcd S dvd Gcd* (*group-closure S*)
    by *auto*

```
  have Gcd (group-closure S) dvd s if s ∈ S for s
  proof −
    from that have s ∈ group-closure S
      by (simp add: group-closure.base)
    then show ?thesis
      by (rule Gcd-dvd)
  qed
  then show Gcd (group-closure S) dvd Gcd S
    by auto
qed simp-all

lemma group-closure-sum:
  fixes S :: int set
  assumes X: finite X X ≠ {} X ⊆ S
  shows (∑ x∈X. a x * x) ∈ group-closure S
  using X by (induction X rule: finite-ne-induct)
    (auto intro: group-closure-mult-right group-closure.base group-closure-add)

lemma Gcd-group-closure-in-group-closure:
  Gcd (group-closure S) ∈ group-closure S for S :: int set
proof (cases S ⊆ {0})
  case True
  then have S = {} ∨ S = {0}
    by auto
  then show ?thesis
    by auto
next
  case False
  then obtain s where s: s ≠ 0 s ∈ S
    by auto
  then have s': |s| ≠ 0 |s| ∈ group-closure S
    by (auto intro: group-closure.base)
  define m where m = (LEAST n. n > 0 ∧ int n ∈ group-closure S)
  have m > 0 ∧ int m ∈ group-closure S
    unfolding m-def
    apply (rule LeastI [of - nat |s|])
    using s'
    by simp
  then have m: int m ∈ group-closure S and 0 < m
    by auto

  have Gcd (group-closure S) = int m
  proof (rule associated-eqI)
    from m show Gcd (group-closure S) dvd int m
      by (rule Gcd-dvd)
    show int m dvd Gcd (group-closure S)
    proof (rule Gcd-greatest)
      fix s
      assume s: s ∈ group-closure S
```

```
      show int m dvd s
      proof (rule ccontr)
        assume ¬ int m dvd s
        then have ∗: 0 < s mod int m
          using ‹0 < m› le-less by fastforce
        have m ≤ nat (s mod int m)
        proof (subst m-def, rule Least-le, rule)
          from ∗ show 0 < nat (s mod int m)
            by simp
          from minus-div-mult-eq-mod [symmetric, of s int m]
          have s mod int m = s − s div int m ∗ int m
            by auto
          also have s − s div int m ∗ int m ∈ group-closure S
            by (auto intro: group-closure.diff s group-closure-mult-right m)
          finally  show int (nat (s mod int m)) ∈ group-closure S
            by simp
        qed
        with ∗ have int m ≤ s mod int m
          by simp
        moreover have s mod int m < int m
          using ‹0 < m› by simp
        ultimately show False
          by auto
      qed
    qed
  qed simp-all
  with m show ?thesis
    by simp
qed

lemma Gcd-in-group-closure:
  Gcd S ∈ group-closure S for S :: int set
  using Gcd-group-closure-in-group-closure [of S]
  by (simp add: Gcd-group-closure-eq-Gcd)

lemma group-closure-eq:
  group-closure S = range (times (Gcd S)) for S :: int set
proof (auto intro: Gcd-in-group-closure group-closure-mult-left)
  fix s
  assume s ∈ group-closure S
  then show s ∈ range (times (Gcd S))
  proof induction
    case (base s)
    then have Gcd S dvd s
      by (auto intro: Gcd-dvd)
    then obtain t where s = Gcd S ∗ t ..
    then show ?case
      by auto
  next
```

case (*diff s t*)
moreover have *Gcd S * a - Gcd S * b = Gcd S * (a - b)* **for** *a b*
by (*simp add*: *algebra-simps*)
ultimately show *?case*
by *auto*
qed
qed

end

theory *Normalized-Fraction*
imports
Main
Euclidean-Algorithm
Fraction-Field
begin

lemma *unit-factor-1-imp-normalized*: *unit-factor x = 1 ⟹ normalize x = x*
using *unit-factor-mult-normalize* [*of x*] **by** *simp*

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*)
by (*simp add*: *normalize-quot-def*)

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*
using *that* **by** (*simp-all add*: *normalize-quot-def Let-def mult.commute* [*of -unit-factor b*] *dvd-div-mult2-eq mult-unit-dvd-iff'*)

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*
by (*auto simp*: *normalized-fracts-def*)

lemma *unit-factor-snd-normalize-quot* [*simp*]:
unit-factor (*snd* (*normalize-quot x*)) = *1*
by (*simp add*: *normalize-quot-def case-prod-unfold Let-def dvd-unit-factor-div*

mult-unit-dvd-iff unit-factor-mult unit-factor-gcd)

lemma *snd-normalize-quot-nonzero* [*simp*]: *snd* (*normalize-quot x*) ≠ *0*
using *unit-factor-snd-normalize-quot*[*of x*]
by (*auto simp del*: *unit-factor-snd-normalize-quot*)

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 −
from *assms* **show** *d dvd a d dvd b*
by (*simp-all add*: *d-def mult-unit-dvd-iff*)
thus *a* = *fst* (*normalize-quot* (*a*,*b*)) ∗ *d b* = *snd* (*normalize-quot* (*a*,*b*)) ∗ *d d* ≠ *0*
by (*auto simp*: *normalize-quot-def Let-def d-def* ‹*b* ≠ *0*›)
qed

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*
using *that*[*OF normalize-quot-aux*[*OF assms*]] **.**

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 −
from *normalize-quotE*[*OF assms*, *of fst x*] **obtain** *d* **where**
fst x = *fst* (*normalize-quot* (*fst x*, *snd x*)) ∗ *d*
snd x = *snd* (*normalize-quot* (*fst x*, *snd x*)) ∗ *d*
d dvd fst x
d dvd snd x
d ≠ *0* **.**
then show *?thesis* **unfolding** *prod.collapse* **by** (*intro that*[*of d*])
qed

lemma *coprime-normalize-quot*:
coprime (*fst* (*normalize-quot x*)) (*snd* (*normalize-quot x*))
by (*simp add*: *normalize-quot-def case-prod-unfold div-mult-unit2*)
(*metis coprime-mult-self-right-iff div-gcd-coprime unit-div-mult-self unit-factor-is-unit*)

lemma *normalize-quot-in-normalized-fracts* [*simp*]: *normalize-quot x* ∈ *normalized-fracts*

by (*simp add*: *normalized-fracts-def coprime-normalize-quot case-prod-unfold*)

lemma *normalize-quot-eq-iff*:
assumes $b \neq 0$ $d \neq 0$
shows *normalize-quot* $(a,b) =$ *normalize-quot* $(c,d) \longleftrightarrow a * d = b * c$
proof −
define x y **where** $x =$ *normalize-quot* (a,b) **and** $y =$ *normalize-quot* (c,d)
from *normalize-quotE*[*OF assms*(*1*), *of a*] *normalize-quotE*[*OF assms*(*2*), *of c*]
obtain *d1 d2*
where $a =$ *fst* $x * d1$ $b =$ *snd* $x * d1$ $c =$ *fst* $y * d2$ $d =$ *snd* $y * d2$ $d1 \neq 0$ $d2 \neq 0$
unfolding *x-def y-def* **by** *metis*
hence $a * d = b * c \longleftrightarrow$ *fst* $x *$ *snd* $y =$ *snd* $x *$ *fst* y **by** *simp*
also have ... $\longleftrightarrow$ *fst* $x =$ *fst* $y \wedge$ *snd* $x =$ *snd* y
by (*intro coprime-crossproduct'*) (*simp-all add*: *x-def y-def coprime-normalize-quot*)
also have ... $\longleftrightarrow x = y$ **using** *prod-eqI* **by** *blast*
finally show $x = y \longleftrightarrow a * d = b * c$ **..**
qed

lemma *normalize-quot-eq-iff'*:
assumes *snd* $x \neq 0$ *snd* $y \neq 0$
shows *normalize-quot* $x =$ *normalize-quot* $y \longleftrightarrow$ *fst* $x *$ *snd* $y =$ *snd* $x *$ *fst* y
using *assms* **by** (*cases x*, *cases y*, *hypsubst*) (*subst normalize-quot-eq-iff*, *simp-all*)

lemma *normalize-quot-id*: $x \in$ *normalized-fracts* $\Longrightarrow$ *normalize-quot* $x = x$
by (*auto simp*: *normalized-fracts-def normalize-quot-def case-prod-unfold*)

lemma *normalize-quot-idem* [*simp*]: *normalize-quot* (*normalize-quot x*) = *normalize-quot x*
by (*rule normalize-quot-id*) *simp-all*

lemma *fractrel-iff-normalize-quot-eq*:
fractrel x $y \longleftrightarrow$ *normalize-quot* $x =$ *normalize-quot* $y \wedge$ *snd* $x \neq 0 \wedge$ *snd* $y \neq 0$
by (*cases x*, *cases y*) (*auto simp*: *fractrel-def normalize-quot-eq-iff*)

lemma *fractrel-normalize-quot-left*:
assumes *snd* $x \neq 0$
shows *fractrel* (*normalize-quot x*) $y \longleftrightarrow$ *fractrel* x y
using *assms* **by** (*subst* (*1 2*) *fractrel-iff-normalize-quot-eq*) *auto*

lemma *fractrel-normalize-quot-right*:
assumes *snd* $x \neq 0$
shows *fractrel* y (*normalize-quot x*) $\longleftrightarrow$ *fractrel* y x
using *assms* **by** (*subst* (*1 2*) *fractrel-iff-normalize-quot-eq*) *auto*

lift-definition *quot-of-fract* ::
'*a* :: {*ring-gcd*,*idom-divide*,*semiring-gcd-mult-normalize*} *fract* ⇒ '*a* × '*a*
is *normalize-quot*

by (*subst* (*asm*) *fractrel-iff-normalize-quot-eq*) *simp-all*

lemma *quot-to-fract-quot-of-fract* [*simp*]: *quot-to-fract* (*quot-of-fract x*) = *x*
unfolding *quot-to-fract-def*
proof *transfer*
fix *x* :: '*a* × '*a* **assume** *rel*: *fractrel x x*
define *x'* **where** *x'* = *normalize-quot x*
obtain *a b* **where** [*simp*]: *x* = (*a*, *b*) **by** (*cases x*)
from *rel* **have** *b* ≠ *0* **by** *simp*
from *normalize-quotE*[*OF this*, *of a*] **obtain** *d*
where
a = *fst* (*normalize-quot* (*a*, *b*)) ∗ *d*
b = *snd* (*normalize-quot* (*a*, *b*)) ∗ *d*
d dvd a
d dvd b
d ≠ *0* **.**
hence *a* = *fst x'* ∗ *d b* = *snd x'* ∗ *d d* ≠ *0 snd x'* ≠ *0* **by** (*simp-all add*: *x'-def*)
thus *fractrel* (*case x' of* (*a*, *b*) ⇒ *if b* = *0 then* (*0*, *1*) *else* (*a*, *b*)) *x*
by (*auto simp add*: *case-prod-unfold*)
qed

lemma *quot-of-fract-quot-to-fract*: *quot-of-fract* (*quot-to-fract x*) = *normalize-quot x*
proof (*cases snd x* = *0*)
case *True*
thus *?thesis* **unfolding** *quot-to-fract-def*
by *transfer* (*simp add*: *case-prod-unfold normalize-quot-def*)
next
case *False*
thus *?thesis* **unfolding** *quot-to-fract-def* **by** *transfer* (*simp add*: *case-prod-unfold*)
qed

lemma *quot-of-fract-quot-to-fract'*:
x ∈ *normalized-fracts* ⟹ *quot-of-fract* (*quot-to-fract x*) = *x*
unfolding *quot-to-fract-def* **by** *transfer* (*auto simp*: *normalize-quot-id*)

lemma *quot-of-fract-in-normalized-fracts* [*simp*]: *quot-of-fract x* ∈ *normalized-fracts*
by *transfer simp*

lemma *normalize-quotI*:
assumes *a* ∗ *d* = *b* ∗ *c b* ≠ *0* (*c*, *d*) ∈ *normalized-fracts*
shows *normalize-quot* (*a*, *b*) = (*c*, *d*)
proof −
from *assms* **have** *normalize-quot* (*a*, *b*) = *normalize-quot* (*c*, *d*)
by (*subst normalize-quot-eq-iff*) *auto*
also have ... = (*c*, *d*) **by** (*intro normalize-quot-id*) *fact*
finally show *?thesis* **.**
qed

lemma *td-normalized-fract*:
type-definition quot-of-fract quot-to-fract normalized-fracts
by *standard* (*simp-all add*: *quot-of-fract-quot-to-fract'*)

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 −
from *normalize-quotE'*[*OF assms*(*1*)] **obtain** *d*
where *d*:
fst x = *fst* (*normalize-quot x*) ∗ *d*
snd x = *snd* (*normalize-quot x*) ∗ *d*
d dvd fst x
d dvd snd x
d ≠ *0* **.**
from *normalize-quotE'*[*OF assms*(*2*)] **obtain** *e*
where *e*:
fst y = *fst* (*normalize-quot y*) ∗ *e*
snd y = *snd* (*normalize-quot y*) ∗ *e*
e dvd fst y
e dvd snd y
e ≠ *0* **.**
show *?thesis* **by** (*simp-all add*: *d e algebra-simps*)
qed

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*))
by *transfer* (*insert quot-of-fract-add-aux*,
simp-all add: *Let-def case-prod-unfold normalize-quot-eq-iff*)

lemma *quot-of-fract-uminus*:
quot-of-fract (−*x*) = (*let* (*a*,*b*) = *quot-of-fract x in* (−*a*, *b*))
by *transfer* (*auto simp*: *case-prod-unfold Let-def normalize-quot-def dvd-neg-div mult-unit-dvd-iff*)

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 -
  have x - y = x + -y by simp
  also have quot-of-fract ... = ?rhs
    by (simp only: quot-of-fract-add quot-of-fract-uminus Let-def case-prod-unfold)
simp-all
  finally show ?thesis .
qed

lemma normalize-quot-mult-coprime:
  assumes coprime a b coprime c d unit-factor b = 1 unit-factor d = 1
  defines e ≡ fst (normalize-quot (a, d)) and f ≡ snd (normalize-quot (a, d))
     and  g ≡ fst (normalize-quot (c, b)) and h ≡ snd (normalize-quot (c, b))
  shows   normalize-quot (a * c, b * d) = (e * g, f * h)
proof (rule normalize-quotI)
  from assms have gcd a b = 1 gcd c d = 1
    by simp-all
  from assms have b ≠ 0 d ≠ 0 by auto
  with assms have normalize b = b normalize d = d
    by (auto intro: normalize-unit-factor-eqI)
  from normalize-quotE [OF ‹b ≠ 0›, of c] obtain k
    where
      c = fst (normalize-quot (c, b)) * k
      b = snd (normalize-quot (c, b)) * k
      k dvd c k dvd b k ≠ 0 .
  note k = this [folded ‹gcd a b = 1› ‹gcd c d = 1› assms(3) assms(4)]
  from normalize-quotE [OF ‹d ≠ 0›, of a] obtain l
    where a = fst (normalize-quot (a, d)) * l
      d = snd (normalize-quot (a, d)) * l
      l dvd a l dvd d l ≠ 0 .
  note l = this [folded ‹gcd a b = 1› ‹gcd c d = 1› assms(3) assms(4)]
  from k l show a * c * (f * h) = b * d * (e * g)
    by (metis e-def f-def g-def h-def mult.commute mult.left-commute)
  from assms have [simp]: unit-factor f = 1 unit-factor h = 1
    by simp-all
 from assms have coprime e f coprime g h by (simp-all add: coprime-normalize-quot)
  with k l assms(1,2) ‹b ≠ 0› ‹d ≠ 0› ‹unit-factor b = 1› ‹unit-factor d = 1›
    ‹normalize b = b› ‹normalize d = d›
  show (e * g, f * h) ∈ normalized-fracts
    by (simp add: normalized-fracts-def unit-factor-mult e-def f-def g-def h-def
      coprime-normalize-quot dvd-unit-factor-div unit-factor-gcd)
      (metis coprime-mult-left-iff coprime-mult-right-iff)
qed (insert assms(3,4), auto)

lemma normalize-quot-mult:
  assumes snd x ≠ 0 snd y ≠ 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 −
from *normalize-quotE'*[*OF assms*(*1*)] **obtain** *d* **where** *d*:
fst x = *fst* (*normalize-quot x*) ∗ *d*
snd x = *snd* (*normalize-quot x*) ∗ *d*
d dvd fst x
d dvd snd x
d ≠ *0* **.**
from *normalize-quotE'*[*OF assms*(*2*)] **obtain** *e* **where** *e*:
fst y = *fst* (*normalize-quot y*) ∗ *e*
snd y = *snd* (*normalize-quot y*) ∗ *e*
e dvd fst y
e dvd snd y
e ≠ *0* **.**
show *?thesis* **by** (*simp-all add*: *d e algebra-simps normalize-quot-eq-iff*)
qed

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*))
by *transfer*
(*simp add*: *split-def Let-def coprime-normalize-quot normalize-quot-mult normalize-quot-mult-coprime*)

lemma *normalize-quot-0* [*simp*]:
normalize-quot (*0*, *x*) = (*0*, *1*) *normalize-quot* (*x*, *0*) = (*0*, *1*)
by (*simp-all add*: *normalize-quot-def*)

lemma *normalize-quot-eq-0-iff* [*simp*]: *fst* (*normalize-quot x*) = *0* ⟷ *fst x* = *0* ∨ *snd x* = *0*
by (*auto simp*: *normalize-quot-def case-prod-unfold Let-def div-mult-unit2 dvd-div-eq-0-iff*)

lemma *fst-quot-of-fract-0-imp*: *fst* (*quot-of-fract x*) = *0* ⟹ *snd* (*quot-of-fract x*) = *1*
by *transfer auto*

lemma *normalize-quot-swap*:
assumes *a* ≠ *0* *b* ≠ *0*
defines *a'* ≡ *fst* (*normalize-quot* (*a*, *b*)) **and** *b'* ≡ *snd* (*normalize-quot* (*a*, *b*))
shows *normalize-quot* (*b*, *a*) = (*b' div unit-factor a'*, *a' div unit-factor a'*)
proof (*rule normalize-quotI*)
from *normalize-quotE*[*OF assms*(*2*), *of a*] **obtain** *d* **where**
a = *fst* (*normalize-quot* (*a*, *b*)) ∗ *d*
b = *snd* (*normalize-quot* (*a*, *b*)) ∗ *d*
d dvd a d dvd b d ≠ *0* **.**
note *d* = *this* [*folded assms*(*3*,*4*)]
show *b* ∗ (*a' div unit-factor a'*) = *a* ∗ (*b' div unit-factor a'*)

```
    using assms(1,2) d
     by (simp add: div-unit-factor [symmetric] unit-div-mult-swap mult-ac del:
div-unit-factor)
  have coprime a' b' by (simp add: a'-def b'-def coprime-normalize-quot)
  thus (b' div unit-factor a', a' div unit-factor a') ∈ normalized-fracts
    using assms(1,2) d
     by (auto simp add: normalized-fracts-def ac-simps dvd-div-unit-iff elim: co-
prime-imp-coprime)
qed fact+

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 (transfer, goal-cases)
  case (1 x)
  from normalize-quot-swap[of fst x snd x] show ?case
    by (auto simp: Let-def case-prod-unfold)
qed

lemma normalize-quot-div-unit-left:
  fixes x y u
  assumes is-unit u
  defines x' ≡ fst (normalize-quot (x, y)) and y' ≡ snd (normalize-quot (x, y))
  shows normalize-quot (x div u, y) = (x' div u, y')
proof (cases y = 0)
  case False
  define v where v = 1 div u
  with ‹is-unit u› have is-unit v and u: ⋀a. a div u = a * v
    by simp-all
  from ‹is-unit v› have coprime v = top
    by (simp add: fun-eq-iff is-unit-left-imp-coprime)
  from normalize-quotE[OF False, of x] obtain d where
    x = fst (normalize-quot (x, y)) * d
    y = snd (normalize-quot (x, y)) * d
    d dvd x d dvd y d ≠ 0 .
  note d = this[folded assms(2,3)]
  from assms have coprime x' y' unit-factor y' = 1
    by (simp-all add: coprime-normalize-quot)
  with d ‹coprime v = top› have normalize-quot (x * v, y) = (x' * v, y')
    by (auto simp: normalized-fracts-def intro: normalize-quotI)
  then show ?thesis
    by (simp add: u)
qed (simp-all add: assms)

lemma normalize-quot-div-unit-right:
  fixes x y u
  assumes is-unit u
  defines x' ≡ fst (normalize-quot (x, y)) and y' ≡ snd (normalize-quot (x, y))
```

shows *normalize-quot* $(x,\ y\ div\ u) = (x' * u,\ y')$
proof (*cases* $y = 0$)
case *False*
from *normalize-quotE*[*OF this*, *of x*]
obtain *d* **where** *d*:
$x = fst\ (normalize\text{-}quot\ (x,\ y)) * d$
$y = snd\ (normalize\text{-}quot\ (x,\ y)) * d$
$d\ dvd\ x\ d\ dvd\ y\ d \neq 0$ **.**
note $d = this$[*folded assms*(*2*,*3*)]
from *assms* **have** *coprime* $x'\ y'$ *unit-factor* $y' = 1$ **by** (*simp-all add*: *coprime-normalize-quot*)
with d ‹*is-unit u*› **show** *?thesis*
by (*auto simp add*: *normalized-fracts-def is-unit-left-imp-coprime unit-div-eq-0-iff intro*: *normalize-quotI*)
qed (*simp-all add*: *assms*)

lemma *normalize-quot-normalize-left*:
fixes *x y u*
defines $x' \equiv fst\ (normalize\text{-}quot\ (x,\ y))$ **and** $y' \equiv snd\ (normalize\text{-}quot\ (x,\ y))$
shows *normalize-quot* $(normalize\ x,\ y) = (x'\ div\ unit\text{-}factor\ x,\ y')$
using *normalize-quot-div-unit-left*[*of unit-factor x x y*]
by (*cases* $x = 0$) (*simp-all add*: *assms*)

lemma *normalize-quot-normalize-right*:
fixes *x y u*
defines $x' \equiv fst\ (normalize\text{-}quot\ (x,\ y))$ **and** $y' \equiv snd\ (normalize\text{-}quot\ (x,\ y))$
shows *normalize-quot* $(x,\ normalize\ y) = (x' * unit\text{-}factor\ y,\ y')$
using *normalize-quot-div-unit-right*[*of unit-factor y x y*]
by (*cases* $y = 0$) (*simp-all add*: *assms*)

lemma *quot-of-fract-0* [*simp*]: *quot-of-fract* $0 = (0,\ 1)$
by *transfer auto*

lemma *quot-of-fract-1* [*simp*]: *quot-of-fract* $1 = (1,\ 1)$
by *transfer* (*rule normalize-quotI*, *simp-all add*: *normalized-fracts-def*)

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 (*cases* $y = 0$)
case *False*
hence *A*: *fst* (*quot-of-fract y*) $\neq 0$ **by** *transfer auto*
have $x\ /\ y = x * inverse\ y$ **by** (*simp add*: *divide-inverse*)
also from *False A* **have** *quot-of-fract* $\ldots$ = *?rhs*
by (*simp only*: *quot-of-fract-mult quot-of-fract-inverse*)
(*simp-all add*: *Let-def case-prod-unfold fst-quot-of-fract-0-imp*
normalize-quot-div-unit-left normalize-quot-div-unit-right
normalize-quot-normalize-right normalize-quot-normalize-left)

finally show *?thesis* **.**
qed *simp-all*

lemma *snd-quot-of-fract-nonzero* [*simp*]: *snd* (*quot-of-fract x*) ≠ *0*
by *transfer simp*

lemma *Fract-quot-of-fract* [*simp*]: *Fract* (*fst* (*quot-of-fract x*)) (*snd* (*quot-of-fract x*)) = *x*
by *transfer* (*simp del*: *fractrel-iff*, *subst fractrel-normalize-quot-left*, *simp*)

lemma *snd-quot-of-fract-Fract-whole*:
assumes *y dvd x*
shows *snd* (*quot-of-fract* (*Fract x y*)) = *1*
using *assms* **by** *transfer* (*auto simp*: *normalize-quot-def Let-def gcd-proj2-if-dvd*)

lemma *fst-quot-of-fract-eq-0-iff* [*simp*]: *fst* (*quot-of-fract x*) = *0* ⟷ *x* = *0*
by *transfer simp*

lemma *coprime-quot-of-fract*:
coprime (*fst* (*quot-of-fract x*)) (*snd* (*quot-of-fract x*))
by *transfer* (*simp add*: *coprime-normalize-quot*)

lemma *unit-factor-snd-quot-of-fract*: *unit-factor* (*snd* (*quot-of-fract x*)) = *1*
using *quot-of-fract-in-normalized-fracts*[*of x*]
by (*simp add*: *normalized-fracts-def case-prod-unfold*)

lemma *normalize-snd-quot-of-fract*: *normalize* (*snd* (*quot-of-fract x*)) = *snd* (*quot-of-fract x*)
by (*intro unit-factor-1-imp-normalized unit-factor-snd-quot-of-fract*)

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*)
by (*auto simp add*: *is-nth-power-def*)

lemma *is-nth-powerI* [*intro?*]: *x* = *y* ^ *n* ⟹ *is-nth-power n x*
by (*auto simp*: *is-nth-power-def*)

lemma *is-nth-powerE*: *is-nth-power n x* $\Longrightarrow$ $(\bigwedge y.\ x = y\ \widehat{}\ n \Longrightarrow P) \Longrightarrow P$
by (*auto simp*: *is-nth-power-def*)

abbreviation *is-square* **where** *is-square* $\equiv$ *is-nth-power 2*

lemma *is-zeroth-power* [*simp*]: *is-nth-power 0 x* $\longleftrightarrow$ *x = 1*
by (*simp add*: *is-nth-power-def*)

lemma *is-first-power* [*simp*]: *is-nth-power 1 x*
by (*simp add*: *is-nth-power-def*)

lemma *is-first-power'* [*simp*]: *is-nth-power* (*Suc 0*) *x*
by (*simp add*: *is-nth-power-def*)

lemma *is-nth-power-0* [*simp*]: *n > 0* $\Longrightarrow$ *is-nth-power n* (*0* :: *'a* :: *semiring-1*)
by (*auto simp*: *is-nth-power-def power-0-left intro*!: *exI*[*of - 0*])

lemma *is-nth-power-0-iff* [*simp*]: *is-nth-power n* (*0* :: *'a* :: *semiring-1*) $\longleftrightarrow$ *n > 0*
by (*cases n*) *auto*

lemma *is-nth-power-1* [*simp*]: *is-nth-power n 1*
by (*auto simp*: *is-nth-power-def intro*!: *exI*[*of - 1*])

lemma *is-nth-power-Suc-0* [*simp*]: *is-nth-power n* (*Suc 0*)
by (*metis One-nat-def is-nth-power-1*)

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 (*cases x = 0*)
case *False*
show *?thesis*
proof (*safe intro*!: *is-nth-powerI elim*!: *is-nth-powerE*)
fix *y p* :: *'a* **assume** $*$: *normalize x = y* $\widehat{}$ *n prime p*
with *assms* **and** *False* **have** [*simp*]: *y* $\neq$ *0* **by** (*auto simp*: *power-0-left*)
have *multiplicity p x = multiplicity p* (*y* $\widehat{}$ *n*)
by (*metis* $*$(*1*) *multiplicity-normalize-right*)
with *False* **and** $*$ **and** *assms* **show** *n dvd multiplicity p x*
by (*auto simp*: *prime-elem-multiplicity-power-distrib*)
next
assume $*$: $\forall p.\ prime\ p \longrightarrow n\ dvd\ multiplicity\ p\ x$
have *multiplicity p* $((\prod p \in prime\text{-}factors\ x.\ p\ \widehat{}\ (multiplicity\ p\ x\ div\ n))\ \widehat{}\ n) =$
multiplicity p x **if** *prime p* **for** *p*
proof −
from *that* **and** $*$ **have** *n dvd multiplicity p x* **by** *blast*
have *multiplicity p x = 0* **if** *p* $\notin$ *prime-factors x*

```
      using that and ‹prime p› by (simp add: prime-factors-multiplicity)
      with that and * and assms show ?thesis unfolding prod-power-distrib
power-mult [symmetric]
     by (subst multiplicity-prod-prime-powers) (auto simp: in-prime-factors-imp-prime)
    qed
    with assms False
     have normalize x = normalize ((∏ p∈prime-factors x. p ^ (multiplicity p x
div n)) ^ n)
     by (intro multiplicity-eq-imp-eq) (auto simp: multiplicity-prod-prime-powers)
    thus normalize x = normalize (∏ p∈prime-factors x. p ^ (multiplicity p x div
n)) ^ n
     by (simp add: normalize-power)
  qed
qed (insert assms, auto)

lemma is-nth-power-conv-multiplicity-nat:
  assumes n > 0
  shows   is-nth-power n (x :: nat) ⟷ (∀ p. prime p ⟶ n dvd multiplicity p x)
  using is-nth-power-conv-multiplicity[OF assms, of x] by simp

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)
  by (metis assms is-nth-power-def power-mult-distrib)

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 −
  have A: is-nth-power n a if coprime a b is-nth-power n (a * b) a ≠ 0 b ≠ 0 n >
0
    for a b :: nat unfolding is-nth-power-conv-multiplicity-nat[OF ‹n > 0›]
  proof safe
    fix p :: nat assume p: prime p
    from ‹coprime a b› have ¬(p dvd a ∧ p dvd b)
      using coprime-common-divisor-nat[of a b p] p by auto
    moreover from that and p
      have n dvd multiplicity p a + multiplicity p b
     by (auto simp: is-nth-power-conv-multiplicity-nat prime-elem-multiplicity-mult-distrib)
    ultimately show n dvd multiplicity p a
      by (auto simp: not-dvd-imp-multiplicity-0)
  qed
  from A [of a b] assms show is-nth-power n a
    by (cases n = 0) simp-all
  from A [of b a] assms show is-nth-power n b
    by (cases n = 0) (simp-all add: ac-simps)
qed
```

lemma *is-nth-power-mult-coprime-nat-iff*:
fixes *a b* :: *nat*
assumes *coprime a b*
shows *is-nth-power n* (*a* ∗ *b*) ⟷ *is-nth-power n a* ∧*is-nth-power n b*
using *assms*
by (*cases a* = *0*; *cases b* = *0*)
(*auto intro*: *is-nth-power-mult dest*: *is-nth-power-mult-coprime-natD*[*of a b n*]
simp del: *One-nat-def*)

lemma *is-nth-power-prime-power-nat-iff*:
fixes *p* :: *nat* **assumes** *prime p*
shows *is-nth-power n* (*p* ^ *k*) ⟷ *n dvd k*
using *assms*
by (*cases n* > *0*)
(*auto simp*: *is-nth-power-conv-multiplicity-nat prime-elem-multiplicity-power-distrib*)

lemma *is-nth-power-nth-power*′:
assumes *n dvd n*′
shows *is-nth-power n* (*m* ^ *n*′)
by (*metis assms dvd-div-mult-self is-nth-power-def power-mult*)

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 (∃ *k*∈{*1*..*m*}. *k* ^ *n* = *m*))
by (*auto simp*: *is-nth-power-nat-def is-nth-power-def power-eq-iff-eq-base self-le-power*)

lemma *is-nth-power-mult-cancel-left*:
fixes *a b* :: ′*a* :: *semiring-gcd*
assumes *is-nth-power n a a* ≠ *0*
shows *is-nth-power n* (*a* ∗ *b*) ⟷ *is-nth-power n b*
proof (*cases n* > *0*)
case *True*
show *?thesis*
proof
assume *is-nth-power n* (*a* ∗ *b*)
then obtain *x* **where** *x*: *a* ∗ *b* = *x* ^ *n*
by (*elim is-nth-powerE*)
obtain *y* **where** *y*: *a* = *y* ^ *n*
using *assms* **by** (*elim is-nth-powerE*)
have *y* ^ *n dvd x* ^ *n*
by (*simp flip*: *x y*)
hence *y dvd x*
using ‹*n* > *0*› **by** *simp*

then obtain z **where** z: $x = y * z$
by (*elim dvdE*)
with ‹$a \neq 0$› **show** *is-nth-power n b*
by (*metis is-nth-powerI mult-left-cancel power-mult-distrib x y*)
qed (*use assms* **in** ‹*auto intro*: *is-nth-power-mult*›)
qed (*use assms* **in** *auto*)

lemma *is-nth-power-mult-cancel-right*:
fixes *a b* :: *'a* :: *semiring-gcd*
assumes *is-nth-power n b b* $\neq$ *0*
shows *is-nth-power n* (*a* $*$ *b*) $\longleftrightarrow$ *is-nth-power n a*
by (*metis assms is-nth-power-mult-cancel-left mult.commute*)

10.2 The n-root of a natural number

definition *nth-root-nat* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat* **where**
nth-root-nat k n = (*if k* = *0 then 0 else Max* {*m*. *m* $\hat{}$ *k* $\leq$ *n*})

lemma *zeroth-root-nat* [*simp*]: *nth-root-nat 0 n* = *0*
by (*simp add*: *nth-root-nat-def*)

lemma *nth-root-nat-aux1*:
assumes *k* > *0*
shows {*m*::*nat*. *m* $\hat{}$ *k* $\leq$ *n*} $\subseteq$ {..*n*}
proof *safe*
fix *m* **assume** *m* $\hat{}$ *k* $\leq$ *n*
show *m* $\leq$ *n*
proof (*cases m* = *0*)
case *False*
with *assms* **have** *m* $\hat{}$ *1* $\leq$ *m* $\hat{}$ *k* **by** (*intro power-increasing*) *simp-all*
also note ‹*m* $\hat{}$ *k* $\leq$ *n*›
finally show *?thesis* **by** *simp*
qed *simp-all*
qed

lemma *nth-root-nat-aux2*:
assumes *k* > *0*
shows *finite* {*m*::*nat*. *m* $\hat{}$ *k* $\leq$ *n*} {*m*::*nat*. *m* $\hat{}$ *k* $\leq$ *n*} $\neq$ {}
proof −
from *assms* **have** {*m*. *m* $\hat{}$ *k* $\leq$ *n*} $\subseteq$ {..*n*} **by** (*rule nth-root-nat-aux1*)
moreover have *finite* {..*n*} **by** *simp*
ultimately show *finite* {*m*::*nat*. *m* $\hat{}$ *k* $\leq$ *n*} **by** (*rule finite-subset*)
next
from *assms* **show** {*m*::*nat*. *m* $\hat{}$ *k* $\leq$ *n*} $\neq$ {} **by** (*auto intro*!: *exI*[*of - 0*] *simp*: *power-0-left*)
qed

lemma
assumes *k* > *0*

shows *nth-root-nat-power-le*: *nth-root-nat k n ^ k ≤ n*
and *nth-root-nat-ge*: *x ^ k ≤ n ⟹ x ≤ nth-root-nat k n*
using *Max-in*[*OF nth-root-nat-aux2*[*OF assms*], *of n*]
Max-ge[*OF nth-root-nat-aux2*(*1*)[*OF assms*], *of x n*] *assms*
by (*auto simp*: *nth-root-nat-def*)

lemma *nth-root-nat-less*:
assumes *k > 0 x ^ k > n*
shows *nth-root-nat k n < x*
by (*meson assms nth-root-nat-power-le order.strict-trans1 power-less-imp-less-base zero-le*)

lemma *nth-root-nat-unique*:
assumes *m ^ k ≤ n* (*m + 1*) *^ k > n*
shows *nth-root-nat k n = m*
proof (*cases k > 0*)
case *True*
from *nth-root-nat-less*[*OF* ‹*k > 0*› *assms*(*2*)]
have *nth-root-nat k n ≤ m* **by** *simp*
moreover from ‹*k > 0*› **and** *assms*(*1*) **have** *nth-root-nat k n ≥ m*
by (*intro nth-root-nat-ge*)
ultimately show *?thesis* **by** (*rule antisym*)
qed (*insert assms*, *auto*)

lemma *nth-root-nat-0* [*simp*]: *nth-root-nat k 0 = 0*
by (*simp add*: *nth-root-nat-def*)

lemma *nth-root-nat-1* [*simp*]: *k > 0 ⟹ nth-root-nat k 1 = 1*
by (*rule nth-root-nat-unique*) (*auto simp del*: *One-nat-def*)

lemma *nth-root-nat-Suc-0* [*simp*]: *k > 0 ⟹ nth-root-nat k* (*Suc 0*) *= Suc 0*
using *One-nat-def is-nth-power-nat-def nth-root-nat-1*
by *presburger*

lemma *first-root-nat* [*simp*]: *nth-root-nat 1 n = n*
by (*intro nth-root-nat-unique*) *auto*

lemma *first-root-nat′* [*simp*]: *nth-root-nat* (*Suc 0*) *n = n*
by (*intro nth-root-nat-unique*) *auto*

lemma *nth-root-nat-code-naive′*:
nth-root-nat k n = (*if k = 0 then 0 else Max* (*Set.filter* (*λm. m ^ k ≤ n*) {*..n*}))
proof (*cases k > 0*)
case *True*
then have {*m. m ^ k ≤ n*} ⊆ {*..n*} **by** (*rule nth-root-nat-aux1*)
then have *Set.filter* (*λm. m ^ k ≤ n*) {*..n*} = {*m. m ^ k ≤ n*}
by (*auto simp*:)
with *True* **show** *?thesis*
by (*simp add*: *nth-root-nat-def*)

qed *simp*

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 ≥ *n* ∨ *acc′* > *n then k else nth-root-nat-aux m* (*k*+*1*) *acc′ n*)
by *auto*
termination by (*relation measure* (λ(-,*k*,-,*n*). *n* − *k*), *goal-cases*) *auto*

lemma *nth-root-nat-aux-le*:
assumes *k* ^ *m* ≤ *n m* > *0*
shows *nth-root-nat-aux m k* (*k* ^ *m*) *n* ^ *m* ≤ *n*
using *assms*
by (*induction m k k* ^ *m n rule*: *nth-root-nat-aux.induct*) (*auto simp*: *Let-def*)

lemma *nth-root-nat-aux-gt*:
assumes *m* > *0*
shows (*nth-root-nat-aux m k* (*k* ^ *m*) *n* + *1*) ^ *m* > *n*
using *assms*
proof (*induction m k k* ^ *m n rule*: *nth-root-nat-aux.induct*)
case (*1 m k n*)
have *n* < *Suc k* ^ *m* **if** *n* ≤ *k*
proof −
note *that*
also have *k* < *Suc k* ^ *1* **by** *simp*
also from ‹*m* > *0*› **have** ... ≤ *Suc k* ^ *m*
by (*intro power-increasing*) *simp-all*
finally show *?thesis* **.**
qed
with *1* **show** *?case* **by** (*auto simp*: *Let-def*)
qed

lemma *nth-root-nat-aux-correct*:
assumes *k* ^ *m* ≤ *n m* > *0*
shows *nth-root-nat-aux m k* (*k* ^ *m*) *n* = *nth-root-nat m n*
by (*metis assms nth-root-nat-aux-gt nth-root-nat-aux-le nth-root-nat-unique*)

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)
using *nth-root-nat-aux-correct*[*of 1 m n*] **by** *auto*

lemma *nth-root-nat-nth-power* [*simp*]: *k* > *0* ⟹ *nth-root-nat k* (*n* ^ *k*) = *n*
by (*intro nth-root-nat-unique order.refl power-strict-mono*) *simp-all*

lemma *nth-root-nat-nth-power′*:
assumes *k* > *0 k dvd m*
shows *nth-root-nat k* (*n* ^ *m*) = *n* ^ (*m div k*)

by (*metis assms dvd-div-mult-self nth-root-nat-nth-power power-mult*)

lemma *nth-root-nat-mono*:
assumes $m \leq n$
shows *nth-root-nat k m* $\leq$ *nth-root-nat k n*
proof (*cases k = 0*)
case *False*
with *assms* **show** *?thesis* **unfolding** *nth-root-nat-def*
using *nth-root-nat-aux2*[*of k m*] *nth-root-nat-aux2*[*of k n*]
by (*auto intro*!: *Max-mono*)
qed *auto*

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* $\Rightarrow$ *'a fract*
where *to-fract x* = *Fract x 1*
— FIXME: more idiomatic name, abbreviation

lemma *to-fract-0* [*simp*]: *to-fract 0* = *0*
by (*simp add*: *to-fract-def eq-fract Zero-fract-def*)

lemma *to-fract-1* [*simp*]: *to-fract 1* = *1*
by (*simp add*: *to-fract-def eq-fract One-fract-def*)

lemma *to-fract-add* [*simp*]: *to-fract* $(x + y)$ = *to-fract x* + *to-fract y*
by (*simp add*: *to-fract-def*)

lemma *to-fract-diff* [*simp*]: *to-fract* $(x - y)$ = *to-fract x* $-$ *to-fract y*
by (*simp add*: *to-fract-def*)

lemma *to-fract-uminus* [*simp*]: *to-fract* $(-x) = -$*to-fract x*
by (*simp add*: *to-fract-def*)

lemma *to-fract-mult* [*simp*]: *to-fract* $(x * y)$ = *to-fract x* $*$ *to-fract y*
by (*simp add*: *to-fract-def*)

lemma *to-fract-eq-iff* [*simp*]: *to-fract x* = *to-fract y* $\longleftrightarrow$ $x = y$
by (*simp add*: *to-fract-def eq-fract*)

lemma *to-fract-eq-0-iff* [*simp*]: *to-fract x = 0 ⟷ x = 0*
by (*simp add*: *to-fract-def Zero-fract-def eq-fract*)

lemma *to-fract-quot-of-fract*:
assumes *snd* (*quot-of-fract x*) = *1*
shows *to-fract* (*fst* (*quot-of-fract x*)) = *x*
proof −
have *x* = *Fract* (*fst* (*quot-of-fract x*)) (*snd* (*quot-of-fract x*)) **by** *simp*
also note *assms*
finally show *?thesis* **by** (*simp add*: *to-fract-def*)
qed

lemma *Fract-conv-to-fract*: *Fract a b* = *to-fract a* / *to-fract b*
by (*simp add*: *to-fract-def*)

lemma *quot-of-fract-to-fract* [*simp*]: *quot-of-fract* (*to-fract x*) = (*x*, *1*)
unfolding *to-fract-def* **by** *transfer* (*simp add*: *normalize-quot-def*)

lemma *snd-quot-of-fract-to-fract* [*simp*]: *snd* (*quot-of-fract* (*to-fract x*)) = *1*
unfolding *to-fract-def* **by** (*rule snd-quot-of-fract-Fract-whole*) *simp-all*

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)
by (*simp add*: *smult-conv-map-poly map-poly-map-poly o-def*)

lemma *fract-poly-0* [*simp*]: *fract-poly 0* = *0*
by (*simp add*: *poly-eqI coeff-map-poly*)

lemma *fract-poly-1* [*simp*]: *fract-poly 1* = *1*
by (*simp add*: *map-poly-pCons*)

lemma *fract-poly-add* [*simp*]:
fract-poly (*p* + *q*) = *fract-poly p* + *fract-poly q*
by (*intro poly-eqI*) (*simp-all add*: *coeff-map-poly*)

lemma *fract-poly-diff* [*simp*]:
fract-poly (*p* − *q*) = *fract-poly p* − *fract-poly q*
by (*intro poly-eqI*) (*simp-all add*: *coeff-map-poly*)

lemma *to-fract-sum* [*simp*]: *to-fract* (*sum f A*) = *sum* (λ*x*. *to-fract* (*f x*)) *A*
by (*cases finite A*, *induction A rule*: *finite-induct*) *simp-all*

lemma *fract-poly-mult* [*simp*]:
fract-poly (*p* ∗ *q*) = *fract-poly p* ∗ *fract-poly q*
by (*intro poly-eqI*) (*simp-all add*: *coeff-map-poly coeff-mult*)

lemma *fract-poly-eq-iff* [*simp*]: *fract-poly p* = *fract-poly q* ⟷ *p* = *q*
by (*auto simp*: *poly-eq-iff coeff-map-poly*)

lemma *fract-poly-eq-0-iff* [*simp*]: *fract-poly p* = *0* ⟷ *p* = *0*
using *fract-poly-eq-iff*[*of p 0*] **by** (*simp del*: *fract-poly-eq-iff*)

lemma *fract-poly-dvd*: *p dvd q* ⟹ *fract-poly p dvd fract-poly q*
by *auto*

lemma *prod-mset-fract-poly*:
(∏ *x*∈#*A*. *map-poly to-fract* (*f x*)) = *fract-poly* (*prod-mset* (*image-mset f A*))
by (*induct A*) (*simp-all add*: *ac-simps*)

lemma *is-unit-fract-poly-iff*:
p dvd 1 ⟷ *fract-poly p dvd 1* ∧ *Polynomial.content p* = *1*
proof *safe*
assume *A*: *p dvd 1*
with *fract-poly-dvd* [*of p 1*] **show** *is-unit* (*fract-poly p*)
by *simp*
from *A* **show** *Polynomial.content p* = *1*
by (*auto simp*: *is-unit-poly-iff normalize-1-iff*)
next
assume *A*: *fract-poly p dvd 1* **and** *B*: *Polynomial.content p* = *1*
from *A* **obtain** *c* **where** *c*: *fract-poly p* = [:*c*:] **by** (*auto simp*: *is-unit-poly-iff*)
{
fix *n* :: *nat* **assume** *n* > *0*
have *to-fract* (*coeff p n*) = *coeff* (*fract-poly p*) *n* **by** (*simp add*: *coeff-map-poly*)
also note *c*
also from ‹*n* > *0*› **have** *coeff* [:*c*:] *n* = *0* **by** (*simp add*: *coeff-pCons split*: *nat.splits*)
finally have *coeff p n* = *0* **by** *simp*
}
hence *degree p* ≤ *0* **by** (*intro degree-le*) *simp-all*
with *B* **show** *p dvd 1* **by** (*auto simp*: *is-unit-poly-iff normalize-1-iff elim*!: *degree-eq-zeroE*)
qed

lemma *fract-poly-is-unit*: *p dvd 1* ⟹ *fract-poly p dvd 1*
using *fract-poly-dvd*[*of p 1*] **by** *simp*

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 −
define *a b* **where** *a = fst (quot-of-fract c)* **and** *b = snd (quot-of-fract c)*
have *smult (to-fract a) (fract-poly q) = smult (to-fract b) (fract-poly p)*
by (*subst smult-eq-iff*) (*simp-all add*: *a-def b-def Fract-conv-to-fract* [*symmetric*] *assms*)
hence *fract-poly (smult a q) = fract-poly (smult b p)* **by** (*simp del*: *fract-poly-eq-iff*)
hence *smult b p = smult a q* **by** (*simp only*: *fract-poly-eq-iff*)
moreover have *c = to-fract a / to-fract b coprime b a normalize b = b*
by (*simp-all add*: *a-def b-def coprime-quot-of-fract* [*of c*] *ac-simps normalize-snd-quot-of-fract Fract-conv-to-fract* [*symmetric*])
ultimately show *?thesis* **by** (*intro that*[*of a b*])
qed

11.3 Fractional Polynomial.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 (Polynomial.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*
by (*simp add*: *primitive-part-fract-def*)

lemma *fract-content-eq-0-iff* [*simp*]:
fract-content p = 0 ⟷ p = 0
unfolding *fract-content-def Let-def Zero-fract-def*
by (*subst eq-fract*) (*auto simp*: *Lcm-0-iff map-poly-eq-0-iff*)

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 ⟹ Polynomial.content (primitive-part-fract p) = 1*

unfolding *primitive-part-fract-def*
by (*rule content-primitive-part*)
(*auto simp*: *primitive-part-fract-def map-poly-eq-0-iff Lcm-0-iff*)

lemma *content-times-primitive-part-fract*:
smult (*fract-content p*) (*fract-poly* (*primitive-part-fract p*)) = *p*
proof −
define *p'* **where** *p'* = *unfract-poly* (*smult* (*to-fract* (*Lcm-coeff-denoms p*)) *p*)
have *fract-poly p'* =
map-poly (*to-fract* ∘ *fst* ∘ *quot-of-fract*) (*smult* (*to-fract* (*Lcm-coeff-denoms p*)) *p*)
unfolding *primitive-part-fract-def p'-def*
by (*subst map-poly-map-poly*) (*simp-all add*: *o-assoc*)
also have ... = *smult* (*to-fract* (*Lcm-coeff-denoms p*)) *p*
proof (*intro map-poly-idI*, *unfold o-apply*)
fix *c* **assume** *c* ∈ *set* (*coeffs* (*smult* (*to-fract* (*Lcm-coeff-denoms p*)) *p*))
then obtain *c'* **where** *c*: *c'* ∈ *set* (*coeffs p*) *c* = *to-fract* (*Lcm-coeff-denoms p*) ∗ *c'*
by (*auto simp add*: *Lcm-0-iff coeffs-smult split*: *if-splits*)
note *c*(*2*)
also have *c'* = *Fract* (*fst* (*quot-of-fract c'*)) (*snd* (*quot-of-fract c'*))
by *simp*
also have *to-fract* (*Lcm-coeff-denoms p*) ∗ ... =
Fract (*Lcm-coeff-denoms p* ∗ *fst* (*quot-of-fract c'*)) (*snd* (*quot-of-fract c'*))
unfolding *to-fract-def* **by** (*subst mult-fract*) *simp-all*
also have *snd* (*quot-of-fract* ...) = *1*
by (*intro snd-quot-of-fract-Fract-whole dvd-mult2 dvd-Lcm*) (*insert c*(*1*), *auto*)
finally show *to-fract* (*fst* (*quot-of-fract c*)) = *c*
by (*rule to-fract-quot-of-fract*)
qed
also have *p'* = *smult* (*Polynomial.content p'*) (*primitive-part p'*)
by (*rule content-times-primitive-part* [*symmetric*])
also have *primitive-part p'* = *primitive-part-fract p*
by (*simp add*: *primitive-part-fract-def p'-def*)
also have *fract-poly* (*smult* (*Polynomial.content p'*) (*primitive-part-fract p*)) =
smult (*to-fract* (*Polynomial.content p'*)) (*fract-poly* (*primitive-part-fract p*)) **by** *simp*
finally have *smult* (*to-fract* (*Polynomial.content p'*)) (*fract-poly* (*primitive-part-fract p*)) =
smult (*to-fract* (*Lcm-coeff-denoms p*)) *p* .
thus *?thesis*
by (*subst* (*asm*) *smult-eq-iff*)
(*auto simp add*: *Let-def p'-def Fract-conv-to-fract field-simps Lcm-0-iff fract-content-def*)
qed

lemma *fract-content-fract-poly* [*simp*]: *fract-content* (*fract-poly p*) = *to-fract* (*Polynomial.content p*)

proof −
have *Lcm-coeff-denoms* (*fract-poly p*) = *1*
by (*auto simp*: *set-coeffs-map-poly*)
hence *fract-content* (*fract-poly p*) =
to-fract (*Polynomial.content* (*map-poly* (*fst* ∘ *quot-of-fract* ∘ *to-fract*) *p*))
by (*simp add*: *fract-content-def to-fract-def fract-collapse map-poly-map-poly del*: *Lcm-1-iff*)
also have *map-poly* (*fst* ∘ *quot-of-fract* ∘ *to-fract*) *p* = *p*
by (*intro map-poly-idI*) *simp-all*
finally show *?thesis* **.**
qed

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'*) *Polynomial.content p'* = *1*
proof (*cases p* = *0*)
case *True*
hence *p* = *smult 0* (*map-poly to-fract 1*) *Polynomial.content 1* = *1* **by** *simp-all*
thus *?thesis* **..**
next
case *False*
thus *?thesis*
by (*rule that*[*OF content-times-primitive-part-fract* [*symmetric*] *content-primitive-part-fract*])
qed

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 Polynomial.content p* = *1*
shows *p dvd q*
proof −
from *assms*(*1*) **obtain** *r* **where** *r*: *fract-poly q* = *fract-poly p* ∗ *r* **by** (*erule dvdE*)
from *content-decompose-fract*[*of r*]
obtain *c r'* **where** *r'*: *r* = *smult c* (*map-poly to-fract r'*) *Polynomial.content r'* = *1* **.**
from *r r'* **have** *eq*: *fract-poly q* = *smult c* (*fract-poly* (*p* ∗ *r'*)) **by** *simp*
from *fract-poly-smult-eqE*[*OF this*] **obtain** *a b*
where *ab*:
c = *to-fract b* / *to-fract a*
smult a q = *smult b* (*p* ∗ *r'*)
coprime a b
normalize a = *a* **.**
have *Polynomial.content* (*smult a q*) = *Polynomial.content* (*smult b* (*p* ∗ *r'*)) **by** (*simp only*: *ab*(*2*))
hence *eq'*: *normalize b* = *a* ∗ *Polynomial.content q* **by** (*simp add*: *assms content-mult r' ab*(*4*))

have *1 = gcd a* (*normalize b*) **by** (*simp add*: *ab*)
also note *eq′*
also have *gcd a* (*a ∗ Polynomial.content q*) = *a* **by** (*simp add*: *gcd-proj1-if-dvd ab*(*4*))
finally have [*simp*]: *a = 1* **by** *simp*
from *eq ab* **have** *q = p ∗* ([:*b*:] *∗ r′*) **by** *simp*
thus *?thesis* **by** (*rule dvdI*)
qed

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 = λp. 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 −
show *dvd.dvd* (*times* :: *′a poly ⇒* -) = *Rings.dvd*
by (*simp add*: *dvd-dict*)
show *comm-monoid-mult.prod-mset times 1 = prod-mset*
by (*simp add*: *prod-mset-dict*)
show *comm-semiring-1.irreducible times 1 0 = irreducible*
by (*simp add*: *irreducible-dict*)
show *comm-semiring-1.prime-elem times 1 0 = prime-elem*
by (*simp add*: *prime-elem-dict*)
show *class.normalization-euclidean-semiring-multiplicative divide plus minus* (*0* :: *′a poly*) *times 1*
modulo (*λp. if p = 0 then 0 else 2 ^ degree p*)
(*λp.* [:*lead-coeff p*:]) (*λp. smult* (*inverse* (*lead-coeff p*)) *p*)
proof (*standard*, *fold dvd-dict*)
fix *p* :: *′a poly*
show [:*lead-coeff p*:] *∗ smult* (*inverse* (*lead-coeff p*)) *p = p*
by (*cases p = 0*) *simp-all*
next
fix *p* :: *′a poly* **assume** *is-unit p*
then show [:*lead-coeff p*:] = *p*
by (*elim is-unit-polyE*) (*auto simp*: *monom-0 one-poly-def field-simps*)
next
fix *p* :: *′a poly* **assume** *p ≠ 0*

then show *is-unit* [:*lead-coeff p*:]
by (*simp add*: *is-unit-pCons-iff*)
next
fix *a b* :: *'a poly* **assume** *is-unit a*
thus [:*lead-coeff* (*a* ∗ *b*):] = *a* ∗ [:*lead-coeff b*:]
by (*auto elim*!: *is-unit-polyE*)
qed (*auto simp*: *lead-coeff-mult Rings.div-mult-mod-eq intro*!: *degree-mod-less'* *degree-mult-right-le*)
qed

lemma *field-poly-irreducible-imp-prime*:
prime-elem p **if** *irreducible p* **for** *p* :: *'a* :: *field poly*
using *that* **by** (*fact field-poly.irreducible-imp-prime-elem*)

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*
using *that* **by** (*fact field-poly.prod-mset-prime-factorization*)

lemma *field-poly-in-prime-factorization-imp-prime*:
prime-elem p **if** *p* ∈# *field-poly.prime-factorization x*
for *p* :: *'a* :: *field poly*
by (*rule field-poly.prime-imp-prime-elem*, *rule field-poly.in-prime-factors-imp-prime*) (*fact that*)

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*) ∧ *Polynomial.content p* = *1*
proof *safe*
assume *p*: *irreducible p*

from *content-decompose*[*of p*] **obtain** *p'* **where** *p'*: *p* = *smult* (*Polynomial.content p*) *p' Polynomial.content p'* = *1* .
hence *p* = [:*Polynomial.content p*:] ∗ *p'* **by** *simp*
from *p this* **have** [:*Polynomial.content p*:] *dvd 1* ∨ *p' dvd 1* **by** (*rule irreducibleD*)
moreover have ¬*p' dvd 1*
proof
assume *p' dvd 1*
hence *degree p* = *0* **by** (*subst p'*) (*auto simp*: *is-unit-poly-iff*)
with *assms* **show** *False* **by** *contradiction*
qed
ultimately show [*simp*]: *Polynomial.content p* = *1* **by** (*simp add*: *is-unit-const-poly-iff*)

show *irreducible* (*map-poly to-fract p*)
proof (*rule irreducibleI*)

```
    have fract-poly p = 0 ⟷ p = 0 by (intro map-poly-eq-0-iff) auto
    with assms show map-poly to-fract p ≠ 0 by auto
  next
    show ¬is-unit (fract-poly p)
    proof
      assume is-unit (map-poly to-fract p)
      hence degree (map-poly to-fract p) = 0
        by (auto simp: is-unit-poly-iff)
      hence degree p = 0 by (simp add: degree-map-poly)
      with assms show False by contradiction
    qed
  next
    fix q r assume qr: fract-poly p = q * r
    from content-decompose-fract[of q]
    obtain cg q' where q: q = smult cg (map-poly to-fract q') Polynomial.content
q' = 1 .
    from content-decompose-fract[of r]
    obtain cr r' where r: r = smult cr (map-poly to-fract r') Polynomial.content
r' = 1 .
    from qr q r p have nz: cg ≠ 0 cr ≠ 0 by auto
    from qr have eq: fract-poly p = smult (cr * cg) (fract-poly (q' * r'))
      by (simp add: q r)
    from fract-poly-smult-eqE[OF this] obtain a b
      where ab: cr * cg = to-fract b / to-fract a
        smult a p = smult b (q' * r') coprime a b normalize a = a .
    hence Polynomial.content (smult a p) = Polynomial.content (smult b (q' * r'))
by (simp only:)
    with ab(4) have a: a = normalize b by (simp add: content-mult q r)
    then have normalize b = gcd a b
      by simp
    with ‹coprime a b› have normalize b = 1
      by simp
    then have a = 1 is-unit b
      by (simp-all add: a normalize-1-iff)

    note eq
    also from ab(1) ‹a = 1› have cr * cg = to-fract b by simp
    also have smult ... (fract-poly (q' * r')) = fract-poly (smult b (q' * r')) by
simp
    finally have p = ([:b:] * q') * r' by (simp del: fract-poly-smult)
    from p and this have ([:b:] * q') dvd 1 ∨ r' dvd 1 by (rule irreducibleD)
    hence q' dvd 1 ∨ r' dvd 1 by (auto dest: dvd-mult-right simp del: mult-pCons-left)
    hence fract-poly q' dvd 1 ∨ fract-poly r' dvd 1 by (auto simp: fract-poly-is-unit)
    with q r show is-unit q ∨ is-unit r
      by (auto simp add: is-unit-smult-iff dvd-field-iff nz)
  qed

next
```

assume *irred*: *irreducible* (*fract-poly p*) **and** *primitive*: *Polynomial.content p* = *1*
show *irreducible p*
proof (*rule irreducibleI*)
from *irred* **show** *p* ≠ *0* **by** *auto*
next
from *irred* **show** ¬*p dvd 1*
by (*auto simp*: *irreducible-def dest*: *fract-poly-is-unit*)
next
fix *q r* **assume** *qr*: *p* = *q* ∗ *r*
hence *fract-poly p* = *fract-poly q* ∗ *fract-poly r* **by** *simp*
from *irred* **and** *this* **have** *fract-poly q dvd 1* ∨ *fract-poly r dvd 1*
by (*rule irreducibleD*)
with *primitive qr* **show** *q dvd 1* ∨ *r dvd 1*
by (*auto simp*: *content-prod-eq-1-iff is-unit-fract-poly-iff*)
qed
qed

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 (*cases degree p* = *0*)
case *True*
with *assms* **show** *?thesis*
by (*auto simp*: *prime-elem-const-poly-iff irreducible-const-poly-iff*
intro!: *irreducible-imp-prime-elem elim*!: *degree-eq-zeroE*)
next
case *False*
from *assms False* **have** *irred*: *irreducible* (*fract-poly p*) **and** *primitive*: *Polynomial.content p* = *1*
by (*simp-all add*: *nonconst-poly-irreducible-iff*)
from *irred* **have** *prime*: *prime-elem* (*fract-poly p*) **by** (*rule field-poly-irreducible-imp-prime*)
show *?thesis*
proof (*rule prime-elemI*)
fix *q r* **assume** *p dvd q* ∗ *r*
hence *fract-poly p dvd fract-poly* (*q* ∗ *r*) **by** (*rule fract-poly-dvd*)
hence *fract-poly p dvd fract-poly q* ∗ *fract-poly r* **by** *simp*
from *prime* **and** *this* **have** *fract-poly p dvd fract-poly q* ∨ *fract-poly p dvd fract-poly r*
by (*rule prime-elem-dvd-multD*)
with *primitive* **show** *p dvd q* ∨ *p dvd r* **by** (*auto dest*: *fract-poly-dvdD*)
qed (*insert assms*, *auto simp*: *irreducible-def*)
qed

lemma *degree-primitive-part-fract* [*simp*]:
degree (*primitive-part-fract p*) = *degree p*
proof −

have $p = smult\ (fract\text{-}content\ p)\ (fract\text{-}poly\ (primitive\text{-}part\text{-}fract\ p))$
by (*simp add*: *content-times-primitive-part-fract*)
also have $degree\ \ldots = degree\ (primitive\text{-}part\text{-}fract\ p)$
by (*auto simp*: *degree-map-poly*)
finally show *?thesis* **..**
qed

lemma *irreducible-primitive-part-fract*:
fixes $p :: 'a :: \{idom\text{-}divide, ring\text{-}gcd, factorial\text{-}semiring, semiring\text{-}Gcd, semiring\text{-}gcd\text{-}mult\text{-}normalize\}$ *fract poly*
assumes *irreducible p*
shows *irreducible* (*primitive-part-fract p*)
proof −
from *assms* **have** *deg*: $degree\ (primitive\text{-}part\text{-}fract\ p) \neq 0$
by (*intro notI*)
(*auto elim*!: *degree-eq-zeroE simp*: *irreducible-def is-unit-poly-iff dvd-field-iff*)
hence [*simp*]: $p \neq 0$ **by** *auto*

note ‹*irreducible p*›
also have $p = [:fract\text{-}content\ p:] * fract\text{-}poly\ (primitive\text{-}part\text{-}fract\ p)$
by (*simp add*: *content-times-primitive-part-fract*)
also have $irreducible\ \ldots \longleftrightarrow irreducible\ (fract\text{-}poly\ (primitive\text{-}part\text{-}fract\ p))$
by (*intro irreducible-mult-unit-left*) (*simp-all add*: *is-unit-poly-iff dvd-field-iff*)
finally show *?thesis* **using** *deg*
by (*simp add*: *nonconst-poly-irreducible-iff*)
qed

lemma *prime-elem-primitive-part-fract*:
fixes $p :: 'a :: \{idom\text{-}divide, ring\text{-}gcd, factorial\text{-}semiring, semiring\text{-}Gcd, semiring\text{-}gcd\text{-}mult\text{-}normalize\}$ *fract poly*
shows $irreducible\ p \Longrightarrow prime\text{-}elem\ (primitive\text{-}part\text{-}fract\ p)$
by (*intro irreducible-imp-prime-poly irreducible-primitive-part-fract*)

lemma *irreducible-linear-field-poly*:
fixes *a b* :: *'a*::*field*
assumes $b \neq 0$
shows *irreducible* [:*a*,*b*:]
proof (*rule irreducibleI*)
fix *p q* **assume** *pq*: $[:a,b:] = p * q$
also from *pq assms* **have** $degree\ \ldots = degree\ p + degree\ q$
by (*intro degree-mult-eq*) *auto*
finally have $degree\ p = 0 \vee degree\ q = 0$ **using** *assms* **by** *auto*
with *assms pq* **show** *is-unit p* ∨ *is-unit q*
by (*auto simp*: *is-unit-const-poly-iff dvd-field-iff elim*!: *degree-eq-zeroE*)
qed (*insert assms*, *auto simp*: *is-unit-poly-iff*)

lemma *prime-elem-linear-field-poly*:
$(b :: 'a :: field) \neq 0 \Longrightarrow prime\text{-}elem\ [:a,b:]$
by (*rule field-poly-irreducible-imp-prime*, *rule irreducible-linear-field-poly*)

lemma *irreducible-linear-poly*:
fixes *a b* :: *'a*::{*idom-divide*,*ring-gcd*,*factorial-semiring*,*semiring-Gcd*,*semiring-gcd-mult-normalize*}
shows $b \neq 0 \Longrightarrow$ *coprime a b* $\Longrightarrow$ *irreducible* [:*a*,*b*:]
by (*auto intro*!: *irreducible-linear-field-poly*
simp: *nonconst-poly-irreducible-iff content-def map-poly-pCons*)

lemma *prime-elem-linear-poly*:
fixes *a b* :: *'a*::{*idom-divide*,*ring-gcd*,*factorial-semiring*,*semiring-Gcd*,*semiring-gcd-mult-normalize*}
shows $b \neq 0 \Longrightarrow$ *coprime a b* $\Longrightarrow$ *prime-elem* [:*a*,*b*:]
by (*rule irreducible-imp-prime-poly*, *rule irreducible-linear-poly*)

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 $p \neq 0$ *Polynomial.content p* = 1
shows $\exists A.\ (\forall p.\ p \in\# A \longrightarrow$ *prime-elem p*$) \wedge$ *prod-mset A* = *normalize p*
proof −
let *?P* = *field-poly.prime-factorization* (*fract-poly p*)
define *c* **where** *c* = *prod-mset* (*image-mset fract-content ?P*)
define *c'* **where** *c'* = *c* ∗ *to-fract* (*lead-coeff p*)
define *e* **where** *e* = *prod-mset* (*image-mset primitive-part-fract ?P*)
define *A* **where** *A* = *image-mset* (*normalize* ∘ *primitive-part-fract*) *?P*
have *Polynomial.content e* = ($\prod x \in\#$ *field-poly.prime-factorization* (*map-poly to-fract p*).
Polynomial.content (*primitive-part-fract x*))
by (*simp add*: *e-def content-prod-mset multiset.map-comp o-def*)
also have *image-mset* (λ*x*. *Polynomial.content* (*primitive-part-fract x*)) *?P* = *image-mset* (λ*-*. *1*) *?P*
by (*intro image-mset-cong content-primitive-part-fract*) *auto*
finally have *content-e*: *Polynomial.content e* = *1*
by *simp*

from ‹*p* ≠ *0*› **have** *fract-poly p* = [:*lead-coeff* (*fract-poly p*):] ∗
smult (*inverse* (*lead-coeff* (*fract-poly p*))) (*fract-poly p*)
by *simp*
also have [:*lead-coeff* (*fract-poly p*):] = [:*to-fract* (*lead-coeff p*):]
by (*simp add*: *monom-0 degree-map-poly coeff-map-poly*)
also from *assms* **have** *smult* (*inverse* (*lead-coeff* (*fract-poly p*))) (*fract-poly p*) = *prod-mset ?P*
by (*subst field-poly-prod-mset-prime-factorization*) *simp-all*
also have ... = *prod-mset* (*image-mset id ?P*) **by** *simp*
also have *image-mset id ?P* =
image-mset (λ*x*. [:*fract-content x*:] ∗ *fract-poly* (*primitive-part-fract x*)) *?P*
by (*intro image-mset-cong*) (*auto simp*: *content-times-primitive-part-fract*)
also have *prod-mset* ... = *smult c* (*fract-poly e*)

by (*subst prod-mset.distrib*) (*simp-all add*: *prod-mset-fract-poly prod-mset-const-poly c-def e-def*)
also have [:*to-fract* (*lead-coeff p*):] ∗ ... = *smult c′* (*fract-poly e*)
by (*simp add*: *c′-def*)
finally have *eq*: *fract-poly p* = *smult c′* (*fract-poly e*) **.**
also obtain *b* **where** *b*: *c′* = *to-fract b is-unit b*
proof −
from *fract-poly-smult-eqE*[*OF eq*]
obtain *a b* **where** *ab*:
c′ = *to-fract b* / *to-fract a*
smult a p = *smult b e*
coprime a b
normalize a = *a* **.**
from *ab*(*2*) **have** *Polynomial.content* (*smult a p*) = *Polynomial.content* (*smult b e*) **by** (*simp only*:)
with *assms content-e* **have** *a* = *normalize b* **by** (*simp add*: *ab*(*4*))
with *ab* **have** *ab′*: *a* = *1 is-unit b*
by (*simp-all add*: *normalize-1-iff*)
with *ab ab′* **have** *c′* = *to-fract b* **by** *auto*
from *this* **and** ‹*is-unit b*› **show** *?thesis* **by** (*rule that*)
qed
hence *smult c′* (*fract-poly e*) = *fract-poly* (*smult b e*) **by** *simp*
finally have *p* = *smult b e* **by** (*simp only*: *fract-poly-eq-iff*)
hence *p* = [:*b*:] ∗ *e* **by** *simp*
with *b* **have** *normalize p* = *normalize e*
by (*simp only*: *normalize-mult*) (*simp add*: *is-unit-normalize is-unit-poly-iff*)
also have *normalize e* = *prod-mset A*
by (*simp add*: *multiset.map-comp e-def A-def normalize-prod-mset*)
finally have *prod-mset A* = *normalize p* **..**

have *prime-elem p* **if** *p* ∈# *A* **for** *p*
using *that* **by** (*auto simp*: *A-def prime-elem-primitive-part-fract prime-elem-imp-irreducible*

dest!: *field-poly-in-prime-factorization-imp-prime*)
from *this* **and** ‹*prod-mset A* = *normalize p*› **show** *?thesis*
by (*intro exI*[*of* - *A*]) *blast*
qed

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 −
define *B* **where** *B* = *image-mset* (λ*x*. [:*x*:]) (*prime-factorization* (*Polynomial.content p*))
have ∃ *A*. (∀ *p*. *p* ∈# *A* ⟶ *prime-elem p*) ∧ *prod-mset A* = *normalize* (*primitive-part p*)

by (*rule poly-prime-factorization-exists-content-1*) (*insert assms*, *simp-all*)
then obtain *A* **where** *A*: ∀ *p*. *p* ∈# *A* ⟶ *prime-elem p* ∏# *A* = *normalize* (*primitive-part p*)
by *blast*
have *normalize* (*prod-mset* (*A* + *B*)) = *normalize* (*prod-mset A* ∗ *normalize* (*prod-mset B*))
by *simp*
also from *assms* **have** *normalize* (*prod-mset B*) = *normalize* [:*Polynomial.content p*:]
by (*simp add*: *prod-mset-const-poly normalize-const-poly prod-mset-prime-factorization-weak B-def*)
also have *prod-mset A* = *normalize* (*primitive-part p*)
using *A* **by** *simp*
finally have *normalize* (*prod-mset* (*A* + *B*)) = *normalize* (*primitive-part p* ∗ [:*Polynomial.content p*:])
by *simp*
moreover have ∀ *p*. *p* ∈# *B* ⟶ *prime-elem p*
by (*auto simp*: *B-def intro*!: *lift-prime-elem-poly dest*: *in-prime-factors-imp-prime*)
ultimately show *?thesis* **using** *A* **by** (*intro exI*[*of* - *A* + *B*]) (*auto*)
qed

end

11.7 Typeclass instances

instance *poly* :: ({*factorial-ring-gcd*,*semiring-gcd-mult-normalize*}) *factorial-semiring*
by *standard* (*rule poly-prime-factorization-exists*)

instantiation *poly* :: ({*factorial-ring-gcd*, *semiring-gcd-mult-normalize*}) *factorial-ring-gcd*
begin

definition *gcd-poly* :: *'a poly* ⇒ *'a poly* ⇒ *'a poly* **where**
gcd-poly = *gcd-factorial*

definition *lcm-poly* :: *'a poly* ⇒ *'a poly* ⇒ *'a poly* **where**
lcm-poly = *lcm-factorial*

definition *Gcd-poly* :: *'a poly set* ⇒ *'a poly* **where**
Gcd-poly = *Gcd-factorial*

definition *Lcm-poly* :: *'a poly set* ⇒ *'a poly* **where**
Lcm-poly = *Lcm-factorial*

instance by *standard* (*simp-all add*: *gcd-poly-def lcm-poly-def Gcd-poly-def Lcm-poly-def*)

end

instance *poly* :: ({*factorial-ring-gcd*, *semiring-gcd-mult-normalize*}) *semiring-gcd-mult-normalize*
..

instance *poly* :: ({*field*,*factorial-ring-gcd*,*semiring-gcd-mult-normalize*})
normalization-euclidean-semiring **..**

instance *poly* :: ({*field*, *normalization-euclidean-semiring*, *factorial-ring-gcd*,
semiring-gcd-mult-normalize}) *euclidean-ring-gcd*
by (*rule euclidean-ring-gcd-class.intro*, *rule factorial-euclidean-semiring-gcdI*) *standard*

instance *poly* :: ({*field*, *normalization-euclidean-semiring*, *factorial-ring-gcd*,
semiring-gcd-mult-normalize}) *factorial-semiring-multiplicative* **..**

11.8 Polynomial GCD

lemma *gcd-degree-le-min*:
assumes *p≠0 q≠0*
shows *degree* (*gcd p q*) ≤ *min* (*degree p*) (*degree q*)
by (*simp add*: *assms dvd-imp-degree-le*)

lemma *lead-coeff-gcd-field*:
fixes *p q*::*'a*::*field-gcd poly*
assumes *p≠0* ∨ *q≠0*
shows *lead-coeff* (*gcd p q*) = *1*
using *assms* **by** (*metis gcd.normalize-idem gcd-eq-0-iff lead-coeff-normalize-field*)

lemma *poly-gcd-0-iff*:
poly (*gcd p q*) *x* = *0* ⟷ *poly p x* = *0* ∧ *poly q x* = *0*
by (*simp add:poly-eq-0-iff-dvd*)

lemma *gcd-poly-decompose*:
fixes *p q* :: *'a* :: {*factorial-ring-gcd*,*semiring-gcd-mult-normalize*} *poly*
shows *gcd p q* =
smult (*gcd* (*Polynomial.content p*) (*Polynomial.content q*)) (*gcd* (*primitive-part p*) (*primitive-part q*))
proof (*rule sym*, *rule gcdI*)
have [:*gcd* (*Polynomial.content p*) (*Polynomial.content q*):] ∗ *gcd* (*primitive-part p*) (*primitive-part q*) *dvd*
[:*Polynomial.content p*:] ∗ *primitive-part p* **by** (*intro mult-dvd-mono*) *simp-all*
thus *smult* (*gcd* (*Polynomial.content p*) (*Polynomial.content q*)) (*gcd* (*primitive-part p*) (*primitive-part q*)) *dvd p*
by *simp*
next
have [:*gcd* (*Polynomial.content p*) (*Polynomial.content q*):] ∗ *gcd* (*primitive-part p*) (*primitive-part q*) *dvd*
[:*Polynomial.content q*:] ∗ *primitive-part q* **by** (*intro mult-dvd-mono*) *simp-all*
thus *smult* (*gcd* (*Polynomial.content p*) (*Polynomial.content q*)) (*gcd* (*primitive-part p*) (*primitive-part q*)) *dvd q*

```
    by simp
next
  fix d assume d dvd p d dvd q
  hence [:Polynomial.content d:] * primitive-part d dvd
        [:gcd (Polynomial.content p) (Polynomial.content q):] * gcd (primitive-part
p) (primitive-part q)
    by (intro mult-dvd-mono) auto
   thus d dvd smult (gcd (Polynomial.content p) (Polynomial.content q)) (gcd
(primitive-part p) (primitive-part q))
    by simp
qed (auto simp: normalize-smult)


lemma gcd-poly-pseudo-mod:
  fixes p q :: 'a :: {factorial-ring-gcd,semiring-gcd-mult-normalize} poly
  assumes nz: q ≠ 0 and prim: Polynomial.content p = 1 Polynomial.content q
= 1
  shows   gcd p q = gcd q (primitive-part (pseudo-mod p q))
proof −
  define r s where r = fst (pseudo-divmod p q) and s = snd (pseudo-divmod p
q)
  define a where a = [:coeff q (degree q) ^(Suc (degree p) − degree q):]
  have [simp]: primitive-part a = unit-factor a
    by (simp add: a-def unit-factor-poly-def unit-factor-power monom-0)
  from nz have [simp]: a ≠ 0 by (auto simp: a-def)

  have rs: pseudo-divmod p q = (r, s) by (simp add: r-def s-def)
  have gcd (q * r + s) q = gcd q s
    using gcd-add-mult[of q r s] by (simp add: gcd.commute add-ac mult-ac)
  with pseudo-divmod(1)[OF nz rs]
    have gcd (p * a) q = gcd q s by (simp add: a-def)
  also from prim have gcd (p * a) q = gcd p q
    by (subst gcd-poly-decompose)
       (auto simp: primitive-part-mult gcd-mult-unit1 primitive-part-prim
            simp del: mult-pCons-right )
  also from prim have gcd q s = gcd q (primitive-part s)
    by (subst gcd-poly-decompose) (simp-all add: primitive-part-prim)
  also have s = pseudo-mod p q by (simp add: s-def pseudo-mod-def)
  finally show ?thesis .
qed

lemma degree-pseudo-mod-less:
  assumes q ≠ 0 pseudo-mod p q ≠ 0
  shows   degree (pseudo-mod p q) < degree q
  using pseudo-mod(2)[of q p] assms by auto

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)))
by auto
termination
  by (relation measure ((λp. if p = 0 then 0 else Suc (degree p)) ∘ snd))
    (auto simp: degree-pseudo-mod-less)

declare gcd-poly-code-aux.simps [simp del]

lemma gcd-poly-code-aux-correct:
  assumes Polynomial.content p = 1 q = 0 ∨ Polynomial.content q = 1
  shows   gcd-poly-code-aux p q = gcd p q
  using assms
proof (induction p q rule: gcd-poly-code-aux.induct)
  case (1 p q)
  show ?case
  proof (cases q = 0)
    case True
    thus ?thesis by (subst gcd-poly-code-aux.simps) auto
  next
    case False
   hence gcd-poly-code-aux p q = gcd-poly-code-aux q (primitive-part (pseudo-mod
p q))
      by (subst gcd-poly-code-aux.simps) simp-all
    also from 1.prems False
      have primitive-part (pseudo-mod p q) = 0 ∨
            Polynomial.content (primitive-part (pseudo-mod p q)) = 1
      by (cases pseudo-mod p q = 0) auto
    with 1.prems False
      have gcd-poly-code-aux q (primitive-part (pseudo-mod p q)) =
            gcd q (primitive-part (pseudo-mod p q))
      by (intro 1) simp-all
    also from 1.prems False
      have ... = gcd p q by (intro gcd-poly-pseudo-mod [symmetric]) auto
    finally show ?thesis .
  qed
qed

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 (Polynomial.content p) (Polynomial.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
 by (simp add: gcd-poly-code-def gcd-poly-code-aux-correct gcd-poly-decompose [symmetric])

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*)
by (*fact lcm-gcd*)

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*]

11.9 The isomorphism *int fract* → *rat*

Isabelle's *rat* and the field of fractions *int fract* of the integers are both built as (*int* × *int*) quotients with the same *Fract* constructor and identical arithmetic.

definition *fract-to-rat* :: *int fract* ⇒ *rat* **where**
fract-to-rat x ≡ *Rat.Fract* (*fst* (*quot-of-fract x*)) (*snd* (*quot-of-fract x*))

The key bridge: on the common *Fract* representation the map is the identity.

lemma *fract-to-rat-Fract*:
assumes *b* ≠ *0*
shows *fract-to-rat* (*Fraction-Field.Fract a b*) = *Rat.Fract a b*
proof −
obtain *a' b'* **where** *q*: *quot-of-fract* (*Fraction-Field.Fract a b*) = (*a'*, *b'*) **by** *fastforce*
then have *ff*: *fract-to-rat* (*Fraction-Field.Fract a b*) = *Rat.Fract a' b'*
by (*simp add*: *fract-to-rat-def*)
have *b'*: *b'* ≠ *0* **using** *q snd-quot-of-fract-nonzero*[*of Fraction-Field.Fract a b*] **by** *simp*
have *Fraction-Field.Fract a' b'* = *Fraction-Field.Fract a b*
using *Fract-quot-of-fract*[*of Fraction-Field.Fract a b*] *q* **by** *simp*
then show *?thesis*
by (*simp add*: *assms b' eq-fract*(*1*) *eq-rat*(*1*) *ff*)
qed

lemma *fract-to-rat-0*: *fract-to-rat 0* = *0*
by (*metis Zero-fract-def fract-to-rat-Fract Rat.Zero-rat-def zero-neq-one*)

lemma *fract-to-rat-1*: *fract-to-rat 1* = *1*
by (*metis One-fract-def fract-to-rat-Fract Rat.One-rat-def zero-neq-one*)

lemma *fract-to-rat-add*: *fract-to-rat* (*x* + *y*) = *fract-to-rat x* + *fract-to-rat y*
proof −
obtain *a b* **where** *x*: *x* = *Fraction-Field.Fract a b b* ≠ *0* **by** (*metis Fract-cases*)
obtain *c d* **where** *y*: *y* = *Fraction-Field.Fract c d d* ≠ *0* **by** (*metis Fract-cases*)
have *x* + *y* = *Fraction-Field.Fract* (*a*∗*d* + *c*∗*b*) (*b*∗*d*) **using** *x y* **by** *simp*
then show *?thesis*

by (*simp add*: *fract-to-rat-Fract x y*)
qed

lemma *fract-to-rat-mult*: *fract-to-rat* (*x* ∗ *y*) = *fract-to-rat x* ∗ *fract-to-rat y*
proof −
obtain *a b* **where** *x*: *x* = *Fraction-Field.Fract a b b* ≠ *0* **by** (*metis Fract-cases*)
obtain *c d* **where** *y*: *y* = *Fraction-Field.Fract c d d* ≠ *0* **by** (*metis Fract-cases*)
have *x* ∗ *y* = *Fraction-Field.Fract* (*a*∗*c*) (*b*∗*d*) **using** *x y* **by** *simp*
then show *?thesis*
by (*simp add*: *fract-to-rat-Fract x y*)
qed

lemma *fract-to-rat-diff*: *fract-to-rat* (*x* − *y*) = *fract-to-rat x* − *fract-to-rat y*
by (*metis eq-diff-eq fract-to-rat-add*)

lemma *fract-to-rat-surj*: *surj fract-to-rat*
proof (*rule surjI*)
fix *y*
show *fract-to-rat* (*Fraction-Field.Fract* (*fst* (*quotient-of y*)) (*snd* (*quotient-of y*)))
= *y*
by (*metis Fract-quotient-of fract-to-rat-Fract less-irrefl quotient-of-denom-pos′*)
qed

lemma *fract-to-rat-eq-0*: *fract-to-rat z* = *0* ⟹ *z* = *0*
proof (*rule ccontr*)
assume *z0*: *fract-to-rat z* = *0* **and** *zne*: *z* ≠ *0*
have *fract-to-rat z* ∗ *fract-to-rat* (*inverse z*) = *fract-to-rat* (*z* ∗ *inverse z*)
by (*simp add*: *fract-to-rat-mult*)
then show *False*
by (*simp add*: *fract-to-rat-1 z0 zne*)
qed

lemma *fract-to-rat-inj*: *inj fract-to-rat*
proof (*rule injI*)
fix *x y* **assume** *fract-to-rat x* = *fract-to-rat y*
then show *x* = *y*
by (*metis fract-to-rat-diff fract-to-rat-eq-0 right-minus-eq*)
qed

lemma *fract-to-rat-bij*: *bij fract-to-rat*
using *fract-to-rat-inj fract-to-rat-surj* **by** (*rule bijI*)

lemma *fract-to-rat-to-fract*: *fract-to-rat* (*to-fract n*) = *of-int n*
by (*simp add*: *fract-to-rat-Fract of-int-rat to-fract-def*)

11.10 The induced isomorphism on polynomials

lemma *fract-to-rat-sum*: *fract-to-rat* (*sum f A*) = (∑ *x*∈*A*. *fract-to-rat* (*f x*))
by (*cases finite A*, *induction A rule*: *finite-induct*)

(*simp-all add*: *fract-to-rat-0 fract-to-rat-add*)

lemma *mfr-0* [*simp*]: *map-poly fract-to-rat 0 = 0*

by (*simp add*: *fract-to-rat-0*)
lemma *mfr-1* [*simp*]: *map-poly fract-to-rat 1 = 1*

by (*simp add*: *map-poly-1′ fract-to-rat-1*)

lemma *mfr-mult*:
map-poly fract-to-rat (*p* ∗ *q*) = *map-poly fract-to-rat p* ∗ *map-poly fract-to-rat q*
by (*intro poly-eqI*)
(*simp-all add*: *coeff-map-poly fract-to-rat-0 fract-to-rat-sum fract-to-rat-mult coeff-mult*)

lemma *inv-fract-to-rat-0*: *inv fract-to-rat 0 = 0*
using *fract-to-rat-0 fract-to-rat-inj* **by** (*metis inv-f-f*)

lemma *mfr-bij*: *bij* (*map-poly fract-to-rat*)
proof (*rule bijI*)
let *?g* = *inv fract-to-rat*
have *gf*: *?g* ∘ *fract-to-rat* = *id* **using** *fract-to-rat-inj* **by** (*simp add*: *inv-o-cancel*)
have *fg*: *fract-to-rat* ∘ *?g* = *id* **using** *fract-to-rat-surj* **by** (*simp add*: *surj-iff*)
have *map-poly ?g* (*map-poly fract-to-rat p*) = *p* **for** *p*
by (*simp add*: *fract-to-rat-0 gf inv-fract-to-rat-0 map-poly-map-poly*)
then show *inj* (*map-poly fract-to-rat*) **by** (*metis injI*)
have *map-poly fract-to-rat* (*map-poly ?g p*) = *p* **for** *p*
by (*simp add*: *fg fract-to-rat-0 inv-fract-to-rat-0 map-poly-map-poly*)
then show *surj* (*map-poly fract-to-rat*) **by** (*metis surjI*)
qed

11.11 Transfer of irreducibility from *int poly* to *rat poly*

An irreducible primitive integer polynomial of positive degree stays irreducible over the rationals. (Gauss bridge to *int fract*, then transport along the isomorphism.)

theorem *irreducible-int-imp-rat*:
fixes *p* :: *int poly*
assumes *irr*: *irreducible p* **and** *deg*: *degree p* ≠ *0* **and** *cont*: *content p = 1*
shows *irreducible* (*map-poly of-int p* :: *rat poly*)
proof −
have *irr-ff*: *irreducible* (*map-poly to-fract p* :: *int fract poly*)
using *irr deg cont* **by** (*subst* (*asm*) *nonconst-poly-irreducible-iff*) *auto*
have *irreducible* (*map-poly fract-to-rat* (*map-poly to-fract p*))
using *ring-iso-irreducible*[*OF mfr-mult mfr-1 mfr-0 mfr-bij*] *irr-ff* **by** *blast*
also have *map-poly fract-to-rat* (*map-poly to-fract p*) = *map-poly of-int p*
by (*simp add*: *coeff-map-poly fract-to-rat-0 fract-to-rat-to-fract poly-eqI*)
finally show *?thesis* **.**
qed

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*
by (*auto simp*: *squarefree-def*)

lemma *squarefreeD*: *squarefree n* ⟹ *x* ^ 2 *dvd n* ⟹ *x dvd 1*
by (*auto simp*: *squarefree-def*)

lemma *not-squarefreeI*: *x* ^ 2 *dvd n* ⟹ ¬*x dvd 1* ⟹ ¬*squarefree n*
by (*auto simp*: *squarefree-def*)

lemma *not-squarefreeE* [*case-names square-dvd*]:
¬*squarefree n* ⟹ (⋀*x*. *x* ^ 2 *dvd n* ⟹ ¬*x dvd 1* ⟹ *P*) ⟹ *P*
by (*auto simp*: *squarefree-def*)

lemma *not-squarefree-0* [*simp*]: ¬*squarefree* (*0* :: *'a* :: *comm-semiring-1*)
by (*rule not-squarefreeI*[*of 0*]) *auto*

lemma *squarefree-factorial-semiring*:
assumes *n* ≠ *0*
shows *squarefree* (*n* :: *'a* :: *factorial-semiring*) ⟷ (∀ *p*. *prime p* ⟶ ¬*p* ^ 2 *dvd n*)
unfolding *squarefree-def*
proof *safe*
assume ∗: ∀ *p*. *prime p* ⟶ ¬*p* ^ 2 *dvd n*
fix *x* :: *'a* **assume** *x*: *x* ^ 2 *dvd n*
{
assume ¬*is-unit x*
moreover from *assms* **and** *x* **have** *x* ≠ *0* **by** *auto*
ultimately obtain *p* **where** *p dvd x prime p*
using *prime-divisor-exists* **by** *blast*
with ∗ **have** ¬*p* ^ 2 *dvd n* **by** *blast*
moreover from ‹*p dvd x*› **have** *p* ^ 2 *dvd x* ^ 2 **by** (*rule dvd-power-same*)
ultimately have ¬*x* ^ 2 *dvd n* **by** (*blast dest*: *dvd-trans*)
with *x* **have** *False* **by** *contradiction*
}
thus *is-unit x* **by** *blast*

qed *auto*

lemma *squarefree-factorial-semiring'*:
assumes $n \neq 0$
shows *squarefree* (n :: $'a$ :: *factorial-semiring*) $\longleftrightarrow$
($\forall$ p∈*prime-factors* n. *multiplicity* p n = 1)
proof (*subst squarefree-factorial-semiring* [*OF assms*], *safe*)
fix p **assume** $\forall$ p∈#*prime-factorization* n. *multiplicity* p n = 1 *prime* p p^2 *dvd* n
with *assms* **show** *False*
by (*cases* p *dvd* n)
(*auto simp*: *prime-factors-dvd power-dvd-iff-le-multiplicity not-dvd-imp-multiplicity-0*)
qed (*auto intro*!: *multiplicity-eqI simp*: *power2-eq-square* [*symmetric*])

lemma *squarefree-factorial-semiring''*:
assumes $n \neq 0$
shows *squarefree* (n :: $'a$ :: *factorial-semiring*) $\longleftrightarrow$
($\forall$ p. *prime* p $\longrightarrow$ *multiplicity* p $n \leq 1$)
by (*subst squarefree-factorial-semiring'*[*OF assms*]) (*auto simp*: *prime-factors-multiplicity*)

lemma *squarefree-unit* [*simp*]: *is-unit* n $\Longrightarrow$ *squarefree* n
proof (*rule squarefreeI*)
fix x **assume** x^2 *dvd* n n *dvd* 1
hence *is-unit* (x^2) **by** (*rule dvd-unit-imp-unit*)
thus *is-unit* x **by** (*simp add*: *is-unit-power-iff*)
qed

lemma *squarefree-1* [*simp*]: *squarefree* (1 :: $'a$ :: *algebraic-semidom*)
by *simp*

lemma *squarefree-minus* [*simp*]: *squarefree* ($-n$ :: $'a$ :: *comm-ring-1*) $\longleftrightarrow$ *squarefree* n
by (*simp add*: *squarefree-def*)

lemma *squarefree-mono*: a *dvd* b $\Longrightarrow$ *squarefree* b $\Longrightarrow$ *squarefree* a
by (*auto simp*: *squarefree-def intro*: *dvd-trans*)

lemma *squarefree-multD*:
assumes *squarefree* (a * b)
shows *squarefree* a *squarefree* b
by (*rule squarefree-mono*[*OF - assms*], *simp*)+

lemma *squarefree-prime-elem*:
assumes *prime-elem* (p :: $'a$ :: *factorial-semiring*)
shows *squarefree* p
proof −
from *assms* **have** $p \neq 0$ **by** *auto*
show *?thesis*
proof (*subst squarefree-factorial-semiring* [*OF* ‹$p \neq 0$›]; *safe*)

```
    fix q assume *: prime q q^2 dvd p
    with assms have multiplicity q p ≥ 2 by (intro multiplicity-geI) auto
    thus False using assms ‹prime q› prime-multiplicity-other[of q normalize p]
      by (cases q = normalize p) simp-all
  qed
qed

lemma squarefree-prime:
  assumes prime (p :: 'a :: factorial-semiring)
  shows   squarefree p
  using assms by (intro squarefree-prime-elem) auto

lemma squarefree-mult-coprime:
  fixes a b :: 'a :: factorial-semiring-gcd
  assumes coprime a b squarefree a squarefree b
  shows   squarefree (a * b)
proof -
  from assms have nz: a * b ≠ 0 by auto
  show ?thesis unfolding squarefree-factorial-semiring'[OF nz]
  proof
    fix p assume p: p ∈ prime-factors (a * b)
    with nz have prime p
      by (simp add: prime-factors-dvd)
    have ¬ (p dvd a ∧ p dvd b)
    proof
      assume p dvd a ∧ p dvd b
      with ‹coprime a b› have is-unit p
        by (auto intro: coprime-common-divisor)
      with ‹prime p› show False
        by simp
    qed
    moreover from p have p dvd a ∨ p dvd b using nz
      by (auto simp: prime-factors-dvd prime-dvd-mult-iff)
    ultimately show multiplicity p (a * b) = 1 using nz p assms(2,3)
      by (auto simp: prime-elem-multiplicity-mult-distrib prime-factors-multiplicity
            not-dvd-imp-multiplicity-0 squarefree-factorial-semiring')
  qed
qed

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)
  using assms
  by (induction A rule: infinite-finite-induct)
     (auto intro!: squarefree-mult-coprime prod-coprime-right)

lemma squarefree-powerD: m > 0 ⟹ squarefree (n ^ m) ⟹ squarefree n
```

by (*cases m*) (*auto dest*: *squarefree-multD*)

lemma *squarefree-power-iff*:
squarefree (*n* ^ *m*) ⟷ *m* = *0* ∨ *is-unit n* ∨ (*squarefree n* ∧ *m* = *1*)
proof *safe*
assume *squarefree* (*n* ^ *m*) *m* > *0* ¬*is-unit n*
show *m* = *1*
proof (*rule ccontr*)
assume *m* ≠ *1*
with ‹*m* > *0*› **have** *n* ^ *2 dvd n* ^ *m* **by** (*intro le-imp-power-dvd*) *auto*
from *this* **and** ‹¬*is-unit n*› **have** ¬*squarefree* (*n* ^ *m*) **by** (*rule not-squarefreeI*)
with ‹*squarefree* (*n* ^ *m*)› **show** *False* **by** *contradiction*
qed
qed (*auto simp*: *is-unit-power-iff dest*: *squarefree-powerD*)

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 *safe*
assume ∗: ∀ *k*∈{*2*..*n*}. ¬ *k*^{2} *dvd n* **and** *n*: *n* > *0*
show *squarefree-nat n* **unfolding** *squarefree-nat-def*
proof (*rule squarefreeI*)
fix *k* **assume** *k*: *k* ^ *2 dvd n*
have *k dvd n* **by** (*rule dvd-trans*[*OF - k*]) *auto*
with *n* **have** *k* ≤ *n* **by** (*intro dvd-imp-le*)
with *bspec*[*OF* ∗, *of k*] *k* **have** ¬*k* > *1* **by** (*intro notI*) *auto*
moreover from *k* **and** *n* **have** *k* ≠ *0* **by** (*intro notI*) *auto*
ultimately have *k* = *1* **by** *presburger*
thus *is-unit k* **by** *simp*
qed
qed (*auto simp*: *squarefree-nat-def squarefree-def intro*!: *Nat.gr0I*)

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*))
by (*simp add*: *square-part-def*)

lemma *square-part-0* [*simp*]: *square-part 0* = *0*
by (*simp add*: *square-part-def*)

lemma *square-part-unit* [*simp*]: *is-unit x* ⟹ *square-part x* = *1*

by (*auto simp*: *square-part-def prime-factorization-unit*)

lemma *square-part-1* [*simp*]: *square-part 1 = 1*
by *simp*

lemma *square-part-0-iff* [*simp*]: *square-part n = 0 ⟷ n = 0*
by (*simp add*: *square-part-def*)

lemma *normalize-uminus* [*simp*]:
normalize (−*x* :: *'a* :: {*normalization-semidom*, *comm-ring-1*}) = *normalize x*
by (*rule associatedI*) *auto*

lemma *multiplicity-uminus-right* [*simp*]:
multiplicity (*x* :: *'a* :: {*factorial-semiring*, *comm-ring-1*}) (−*y*) = *multiplicity x y*
proof −
have *multiplicity x* (−*y*) = *multiplicity x* (*normalize* (−*y*))
by (*rule multiplicity-normalize-right* [*symmetric*])
also have ... = *multiplicity x y* **by** *simp*
finally show *?thesis* **.**
qed

lemma *multiplicity-uminus-left* [*simp*]:
multiplicity (−*x* :: *'a* :: {*factorial-semiring*, *comm-ring-1*}) *y* = *multiplicity x y*
proof −
have *multiplicity* (−*x*) *y* = *multiplicity* (*normalize* (−*x*)) *y*
by (*rule multiplicity-normalize-left* [*symmetric*])
also have ... = *multiplicity x y* **by** *simp*
finally show *?thesis* **.**
qed

lemma *prime-factorization-uminus* [*simp*]:
prime-factorization (−*x* :: *'a* :: {*factorial-semiring*, *comm-ring-1*}) = *prime-factorization x*
by (*rule prime-factorization-cong*) *simp-all*

lemma *square-part-uminus* [*simp*]:
square-part (−*x* :: *'a* :: {*factorial-semiring*, *comm-ring-1*}) = *square-part x*
by (*simp add*: *square-part-def*)

lemma *prime-multiplicity-square-part*:
assumes *prime p*
shows *multiplicity p* (*square-part n*) = *multiplicity p n div 2*
proof (*cases n = 0*)
case *False*
thus *?thesis* **unfolding** *square-part-nonzero*[*OF False*] *multiplicity-normalize-right*
using *finite-prime-divisors*[*of n*] *assms*
by (*subst multiplicity-prod-prime-powers*)
(*auto simp*: *not-dvd-imp-multiplicity-0 prime-factors-dvd multiplicity-prod-prime-powers*)
qed *auto*

lemma *square-part-square-dvd* [*simp*, *intro*]: *square-part n ^ 2 dvd n*
proof (*cases n = 0*)
  case *False*
  thus *?thesis*
    by (*intro multiplicity-le-imp-dvd*)
     (*auto simp*: *prime-multiplicity-square-part prime-elem-multiplicity-power-distrib*)
qed *auto*

lemma *prime-multiplicity-le-imp-dvd*:
  assumes *x ≠ 0 y ≠ 0*
  shows *x dvd y ⟷ (∀ p. prime p ⟶ multiplicity p x ≤ multiplicity p y)*
  using *assms* **by** (*auto intro*: *multiplicity-le-imp-dvd dvd-imp-multiplicity-le*)

lemma *dvd-square-part-iff*: *x dvd square-part n ⟷ x ^ 2 dvd n*
proof (*cases x = 0*; *cases n = 0*)
  assume *nz*: *x ≠ 0 n ≠ 0*
  thus *?thesis*
    by (*subst* (*1 2*) *prime-multiplicity-le-imp-dvd*)
     (*auto simp*: *prime-multiplicity-square-part prime-elem-multiplicity-power-distrib*)
qed *auto*

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*
  by (*simp add*: *squarefree-part-def*)

lemma *squarefree-part-unit* [*simp*]: *is-unit n ⟹ squarefree-part n = n*
  by (*auto simp add*: *squarefree-part-def*)

lemma *squarefree-part-1* [*simp*]: *squarefree-part 1 = 1*
  by *simp*

lemma *squarefree-decompose*: *n = squarefree-part n * square-part n ^ 2*
  by (*simp add*: *squarefree-part-def*)

lemma *squarefree-part-uminus* [*simp*]:
  assumes *x ≠ 0*
 shows *squarefree-part* (−*x* :: *'a* :: {*factorial-semiring*, *comm-ring-1*}) = −*squarefree-part x*
proof −
  have −(*squarefree-part x * square-part x ^ 2*) = −*x*
    by (*subst squarefree-decompose* [*symmetric*]) *auto*
  also have ... = *squarefree-part* (−*x*) * *square-part* (−*x*) ^ *2* **by** (*rule squarefree-decompose*)
  finally have (− *squarefree-part x*) * *square-part x ^ 2* =
                squarefree-part (−*x*) * *square-part x ^ 2* **by** *simp*

thus *?thesis* **using** *assms* **by** (*subst* (*asm*) *mult-right-cancel*) *auto*
qed

lemma *squarefree-part-nonzero* [*simp*]: *squarefree-part n* ≠ *0*
using *squarefree-decompose*[*of n*] **by** (*cases n* ≠ *0*) *auto*

lemma *prime-multiplicity-squarefree-part*:
assumes *prime p*
shows *multiplicity p* (*squarefree-part n*) = *multiplicity p n mod 2*
proof (*cases n* = *0*)
case *False*
hence *n*: *n* ≠ *0* **by** *auto*
have *multiplicity p n mod 2* + *2* ∗ (*multiplicity p n div 2*) = *multiplicity p n* **by** *simp*
also have ... = *multiplicity p* (*squarefree-part n* ∗ *square-part n* ^ *2*)
by (*subst squarefree-decompose*[*of n*]) *simp*
also from *assms n* **have** ... = *multiplicity p* (*squarefree-part n*) + *2* ∗ (*multiplicity p n div 2*)
by (*subst prime-elem-multiplicity-mult-distrib*)
(*auto simp*: *prime-elem-multiplicity-power-distrib prime-multiplicity-square-part*)
finally show *?thesis* **by** (*subst* (*asm*) *add-right-cancel*) *simp*
qed *auto*

lemma *prime-multiplicity-squarefree-part-le-Suc-0* [*intro*]:
assumes *prime p*
shows *multiplicity p* (*squarefree-part n*) ≤ *Suc 0*
by (*simp add*: *assms prime-multiplicity-squarefree-part*)

lemma *squarefree-squarefree-part* [*simp*, *intro*]: *squarefree* (*squarefree-part n*)
by (*subst squarefree-factorial-semiring''*)
(*auto simp*: *prime-multiplicity-squarefree-part-le-Suc-0*)

lemma *squarefree-decomposition-unique*:
assumes *square-part m* = *square-part n*
assumes *squarefree-part m* = *squarefree-part n*
shows *m* = *n*
by (*subst* (*1 2*) *squarefree-decompose*) (*simp-all add*: *assms*)

lemma *normalize-square-part* [*simp*]: *normalize* (*square-part x*) = *square-part x*
by (*simp add*: *square-part-def*)

lemma *square-part-even-power'*: *square-part* (*x* ^ (*2* ∗ *n*)) = *normalize* (*x* ^ *n*)
proof (*cases x* = *0*)
case *False*
have *normalize* (*square-part* (*x* ^ (*2* ∗ *n*))) = *normalize* (*x* ^ *n*) **using** *False*
by (*intro multiplicity-eq-imp-eq*)
(*auto simp*: *prime-multiplicity-square-part prime-elem-multiplicity-power-distrib*)
thus *?thesis* **by** *simp*
qed (*auto simp*: *power-0-left*)

lemma *square-part-even-power*: *even n* $\Longrightarrow$ *square-part* (*x* ^ *n*) = *normalize* (*x* ^ (*n div 2*))
by (*subst square-part-even-power'* [*symmetric*]) *auto*

lemma *square-part-odd-power'*: *square-part* (*x* ^ (*Suc* (*2* * *n*))) = *normalize* (*x* ^ *n* * *square-part x*)
proof (*cases x* = *0*)
case *False*
have *normalize* (*square-part* (*x* ^ (*Suc* (*2* * *n*)))) = *normalize* (*square-part x* * *x* ^ *n*)
proof (*rule multiplicity-eq-imp-eq*, *goal-cases*)
case (*3 p*)
hence *multiplicity p* (*square-part* (*x* ^ *Suc* (*2* * *n*))) =
(*2* * (*n* * *multiplicity p x*) + *multiplicity p x*) *div 2*
by (*subst prime-multiplicity-square-part*)
(*auto simp*: *False prime-elem-multiplicity-power-distrib algebra-simps simp del*: *power-Suc*)
also from *3 False* **have** ... = *multiplicity p* (*square-part x* * *x* ^ *n*)
by (*subst div-mult-self4*) (*auto simp*: *prime-multiplicity-square-part prime-elem-multiplicity-mult-distrib prime-elem-multiplicity-power-distrib*)
finally show *?case* **.**
qed (*insert False*, *auto*)
thus *?thesis* **by** (*simp add*: *mult-ac*)
qed *auto*

lemma *square-part-odd-power*:
odd n $\Longrightarrow$ *square-part* (*x* ^ *n*) = *normalize* (*x* ^ (*n div 2*) * *square-part x*)
by (*subst square-part-odd-power'* [*symmetric*]) *auto*

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* **..**

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
by *standard*
(*simp-all add*: *dvd-field-iff field-split-simps split*: *if-splits*)

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 by *standard* (*simp-all add*: *gcd-real-def lcm-real-def Gcd-real-def Lcm-real-def*)

end

instance *real* :: *field-gcd* **..**

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
by *standard*
(*simp-all add*: *dvd-field-iff field-split-simps split*: *if-splits*)

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 by *standard* (*simp-all add*: *gcd-rat-def lcm-rat-def Gcd-rat-def Lcm-rat-def*)

end

instance *rat* :: *field-gcd* **..**

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
by *standard*
(*simp-all add*: *dvd-field-iff field-split-simps split*: *if-splits*)

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 by *standard* (*simp-all add*: *gcd-complex-def lcm-complex-def Gcd-complex-def Lcm-complex-def*)

end

instance *complex* :: *field-gcd* **..**

end

14 Computation checks

theory *Computation-Checks*
imports *Primes Polynomial-Factorial HOL−Library.Discrete-Functions HOL−Library.Code-Target-Numeral*
begin

floor-sqrt 16476148165462159 = 128359449

prime 97

prime 97

prime 9973

prime 9973

Gcd {[:*1*, *2*, *3*:], [:*2*, *3*, *4*:]} = *1*

Lcm {[:*1*, *2*, *3*:], [:*2*, *3*, *4*:]} = [:[:*2*:], [:*7*:], [:*16*:], [:*17*:], [:*12*:]:]

end

References

[1] K. J. Nowak. Some elementary proofs of Puiseuxs theorems. *Univ. Iagel. Acta Math*, 38:279–282, 2000.