

Formal Puiseux Series

Manuel Eberl

February 6, 2026

Abstract

Formal Puiseux series are generalisations of formal power series and formal Laurent series that also allow for fractional exponents. They have the following general form:

$$\sum_{i=N}^{\infty} a_{i/d} X^{i/d}$$

where N is an integer and d is a positive integer.

This entry defines these series including their basic algebraic properties. Furthermore, it proves the Newton–Puiseux Theorem, namely that the Puiseux series over an algebraically closed field of characteristic 0 are also algebraically closed.

Contents

1	Auxiliary material	3
1.1	Facts about polynomials	3
1.2	A typeclass for algebraically closed fields	3
2	Hensel's lemma for formal power series	10
3	Formal Puiseux Series	19
3.1	Auxiliary facts and definitions	19
3.2	Definition	21
3.3	Basic algebraic typeclass instances	23
3.4	The substitution $X \mapsto X^r$	27
3.5	Multiplication and ring properties	32
3.6	Constant Puiseux series and the series X	36
3.7	More algebraic typeclass instances	37
3.8	Valuation	38
3.9	Powers of X and shifting	43
3.10	The n -th root of a Puiseux series	46
3.11	Algebraic closedness	48
3.12	Metric and topology	56

1 Auxiliary material

1.1 Facts about polynomials

theory *Puiseux-Polynomial-Library*

imports *HOL-Computational-Algebra.Computational-Algebra Polynomial-Interpolation.Ring-Hom-Poly*
begin

lemma *inj-idom-hom-compose* [intro]:

assumes *inj-idom-hom f inj-idom-hom g*

shows *inj-idom-hom (f ∘ g)*

proof –

interpret *f: inj-idom-hom f* **by** *fact*

interpret *g: inj-idom-hom g* **by** *fact*

show *?thesis*

by *unfold-locales (auto simp: f.hom-add g.hom-add f.hom-mult g.hom-mult)*

qed

lemma (in *inj-idom-hom*) *inj-idom-hom-map-poly* [intro]: *inj-idom-hom (map-poly hom)*

proof –

interpret *map-poly-inj-idom-hom hom* **by** *unfold-locales*

show *?thesis*

by (*simp add: inj-idom-hom-axioms*)

qed

lemma *inj-idom-hom-pcompose* [intro]:

assumes [*simp*]: *degree (p :: 'a :: idom poly) ≠ 0*

shows *inj-idom-hom (λq. pcompose q p)*

proof

show $\bigwedge x. x \circ_p p = 0 \implies x = 0$

using *pcompose-eq-0 assms* **by** *blast*

qed

1.2 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 ⟹ f n ≠ 0 ⟹ ∃ x. (∑ k ≤ n. f k * x ^ k) = 0*

We can then however easily show the equivalence to the proper definition:

lemma *alg-closed-imp-poly-has-root*:

assumes *degree (p :: 'a :: alg-closed-field poly) > 0*

shows $\exists x. \text{poly } p \ x = 0$

proof –

have $\exists x. (\sum k \leq \text{degree } p. \text{coeff } p \ k * x ^ k) = 0$

using *assms* **by** (*intro alg-closed*) *auto*

thus *?thesis*
by (*simp add: poly-altdef*)
qed

lemma *alg-closedI* [*Pure.intro*]:

assumes $\bigwedge p :: 'a \text{ poly. degree } p > 0 \implies \text{lead-coeff } p = 1 \implies \exists x. \text{poly } p \ x = 0$
shows *OFCLASS*('a :: field, alg-closed-field-class)

proof

fix $n :: \text{nat}$ **and** $f :: \text{nat} \Rightarrow 'a$
assume $n: n > 0 \wedge f \ n \neq 0$
define p **where** $p = \text{Abs-poly } (\lambda k. \text{if } k \leq n \text{ then } f \ k \text{ else } 0)$
have *coeff-p*: $\text{coeff } p \ k = (\text{if } k \leq n \text{ then } f \ k \text{ else } 0)$ **for** k
proof –
have *eventually* $(\lambda k. k > n)$ *cofinite*
by (*auto simp: MOST-nat*)
hence *eventually* $(\lambda k. (\text{if } k \leq n \text{ then } f \ k \text{ else } 0) = 0)$ *cofinite*
by *eventually-elim auto*
thus *?thesis*
unfolding *p-def* **by** (*subst Abs-poly-inverse*) *auto*
qed

from n **have** *degree* $p \geq n$
by (*intro le-degree*) (*auto simp: coeff-p*)
moreover **have** *degree* $p \leq n$
by (*intro degree-le*) (*auto simp: coeff-p*)
ultimately **have** *deg-p*: *degree* $p = n$
by *linarith*
from *deg-p* **and** n **have** [*simp*]: $p \neq 0$
by *auto*

define p' **where** $p' = \text{smult } (\text{inverse } (\text{lead-coeff } p)) \ p$
have *deg-p'*: *degree* $p' = \text{degree } p$
by (*auto simp: p'-def*)
have *lead-coeff-p'* [*simp*]: *lead-coeff* $p' = 1$
by (*auto simp: p'-def*)

from *deg-p* **and** *deg-p'* **and** n **have** *degree* $p' > 0$
by *simp*
from *assms*[*OF this*] **obtain** x **where** *poly* $p' \ x = 0$
by *auto*
hence *poly* $p \ x = 0$
by (*simp add: p'-def*)
also **have** *poly* $p \ x = (\sum_{k \leq n} f \ k * x \ ^k)$
unfolding *poly-altdef* **by** (*intro sum.cong*) (*auto simp: deg-p coeff-p*)
finally **show** $\exists x. (\sum_{k \leq n} f \ k * x \ ^k) = 0 \ ..$
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.premis 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]:  $q \neq 0$ 
  using assms  $q$  by auto

show ?thesis
proof (rule reducible-polyI)
  show  $p = [-z, 1:] * q$ 
    by fact
next
  have  $\text{degree } p = \text{degree } ([-z, 1:] * q)$ 
    by (simp only: q)
  also have  $\dots = \text{degree } q + 1$ 
    by (subst degree-mult-eq) auto
  finally show  $\text{degree } q > 0$ 
    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 $\bigwedge p :: 'a \text{ poly. degree } p > 1 \implies \text{lead-coeff } p = 1 \implies \text{coeff } p \ 0 \neq 0 \implies$
 $\quad \neg \text{irreducible } p$

shows $\text{OFCLASS}('a :: \text{field, alg-closed-field-class})$

proof

fix $p :: 'a \text{ poly}$ **assume** $p: \text{degree } p > 0 \text{ lead-coeff } p = 1$

show $\exists x. \text{poly } p \ x = 0$

proof (*cases coeff p 0 = 0*)

case *True*

hence $\text{poly } p \ 0 = 0$

by (*simp add: poly-0-coeff-0*)

thus *?thesis* **by** *blast*

next

case *False*

from p **and** *this* **show** *?thesis*

proof (*induction degree p arbitrary: p rule: less-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-splits elim!: degree-eq-zeroE*)

from *True* **have** [*simp*]: $b \neq 0$

by (*auto simp: p*)

have $\text{poly } p \ (-a/b) = 0$

by (*auto simp: p*)

thus *?thesis* **by** *blast*

next

```

case False
hence degree p > 1
  using less.premis by auto
from assms[OF <degree p > 1> <lead-coeff p = 1> <coeff p 0 ≠ 0>]
have  $\neg$ irreducible p by auto
then obtain r s where rs: degree r > 0 degree s > 0 p = r * s
  using less.premis by (auto simp: irreducible-def)
hence coeff r 0 ≠ 0
  using  $\langle$ coeff p 0 ≠ 0 $\rangle$  by (auto simp: coeff-mult-0)

define r' where r' = smult (inverse (lead-coeff r)) r
have [simp]: degree r' = degree r
  by (simp add: r'-def)
have lc: lead-coeff r' = 1
  using rs by (auto simp: r'-def)
have nz: coeff r' 0 ≠ 0
  using  $\langle$ coeff r 0 ≠ 0 $\rangle$  by (auto simp: r'-def)

have degree r < degree r + degree s
  using rs by linarith
also have  $\dots = \text{degree } (r * s)$ 
  using rs(3) less.premis by (subst degree-mult-eq) auto
also have  $r * s = p$ 
  using rs(3) by simp
finally have  $\exists x. \text{poly } r' x = 0$ 
  by (intro less) (use lc rs nz in auto)
thus ?thesis
  using rs(3) by (auto simp: r'-def)
qed
qed
qed
qed

```

Using a clever Tschirnhausen transformation mentioned e.g. in the article by Nowak [2], we can also assume w.l.o.g. that the coefficient a_{n-1} is zero.

lemma *alg-closedI-reducible-coeff-deg-minus-one-eq-0*:

assumes $\bigwedge p :: 'a \text{ poly. degree } p > 1 \implies \text{lead-coeff } p = 1 \implies \text{coeff } p (\text{degree } p - 1) = 0 \implies$

$\text{coeff } p 0 \neq 0 \implies \neg \text{irreducible } p$

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)
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 by (auto simp: irreducible-def)

have p = pcompose q s

```

by (simp add: q-def r-def s-def 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)
 ultimately show $\neg irreducible\ p$
 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 $poly\ p\ x = 0\ poly\ p\ y = 0\ x \neq y$

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]: $q \neq 0$

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 $poly\ q\ x \neq 0$

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 $\neg (degree\ d = 0)$

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 auto
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 poly-zero by blast
qed (use ⟨n > 0⟩ in ⟨simp-all add: p-eq degree-power-eq⟩)
qed

instance complex :: alg-closed-field
  by standard (use constant-degree fundamental-theorem-of-algebra neq0-conv in
blast)

end

```

2 Hensel’s lemma for formal power series

theory *FPS-Hensel*

imports *HOL-Computational-Algebra.Computational-Algebra Puiseux-Polynomial-Library*
begin

The following proof of Hensel’s lemma for formal power series follows the book “Algebraic Geometry for Scientists and Engineers” by Abhyankar [1, p. 90–92].

definition *fps-poly-swap1* :: 'a :: zero fps poly \Rightarrow 'a poly fps **where**
fps-poly-swap1 p = Abs-fps ($\lambda m. \text{Abs-poly } (\lambda n. \text{fps-nth } (\text{coeff } p \ n) \ m)$)

lemma *coeff-fps-nth-fps-poly-swap1* [*simp*]:

coeff (fps-nth (fps-poly-swap1 p) m) n = fps-nth (coeff p n) m

proof –

have $\forall \infty n. \text{poly.coeff } p \ n = 0$

using *MOST-coeff-eq-0* **by** blast

hence $\forall \infty n. \text{poly.coeff } p \ n \ \$ \ m = 0$

by *eventually-elim auto*

thus *?thesis*

by (*simp add: fps-poly-swap1-def poly.Abs-poly-inverse*)

qed

definition *fps-poly-swap2* :: 'a :: zero poly fps \Rightarrow 'a fps poly **where**

fps-poly-swap2 p = Abs-poly ($\lambda m. \text{Abs-fps } (\lambda n. \text{coeff } (\text{fps-nth } p \ n) \ m)$)

lemma *fps-nth-coeff-fps-poly-swap2*:
assumes $\bigwedge n. \text{degree } (\text{fps-nth } p \ n) \leq d$
shows $\text{fps-nth } (\text{coeff } (\text{fps-poly-swap2 } p) \ m) \ n = \text{coeff } (\text{fps-nth } p \ n) \ m$
proof –
have $\forall_{\infty} n. n > d$
using *MOST-nat* **by** *blast*
hence $\forall_{\infty} n. (\lambda m. \text{poly.coeff } (p \ \$ \ m) \ n) = (\lambda-. \ 0)$
by *eventually-elim* (*auto simp: fun-eq-iff intro!: coeff-eq-0 le-less-trans[OF assms(1)]*)
hence *ev*: $\forall_{\infty} n. \text{Abs-fps } (\lambda m. \text{poly.coeff } (p \ \$ \ m) \ n) = 0$
by *eventually-elim* (*simp add: fps-zero-def*)

have $\text{fps-nth } (\text{coeff } (\text{fps-poly-swap2 } p) \ m) \ n =$
 $\text{poly.coeff } (\text{Abs-poly } (\lambda m. \text{Abs-fps } (\lambda n. \text{poly.coeff } (p \ \$ \ n) \ m))) \ m \ \$ \ n$
by (*simp add: fps-poly-swap2-def*)
also have $\dots = \text{Abs-fps } (\lambda n. \text{poly.coeff } (p \ \$ \ n) \ m) \ \$ \ n$
using *ev* **by** (*subst poly.Abs-poly-inverse*) *auto*
finally show $\text{fps-nth } (\text{coeff } (\text{fps-poly-swap2 } p) \ m) \ n = \text{coeff } (\text{fps-nth } p \ n) \ m$
by *simp*
qed

lemma *degree-fps-poly-swap2-le*:
assumes $\bigwedge n. \text{degree } (\text{fps-nth } p \ n) \leq d$
shows $\text{degree } (\text{fps-poly-swap2 } p) \leq d$
proof (*safe intro!: degree-le*)
fix *n* **assume** $n > d$
show $\text{poly.coeff } (\text{fps-poly-swap2 } p) \ n = 0$
proof (*rule fps-ext*)
fix *m*
have $\text{poly.coeff } (\text{fps-poly-swap2 } p) \ n \ \$ \ m = \text{poly.coeff } (p \ \$ \ m) \ n$
by (*subst fps-nth-coeff-fps-poly-swap2[OF assms]*) *auto*
also have $\dots = 0$
by (*intro coeff-eq-0 le-less-trans[OF assms <n > d]*)
finally show $\text{poly.coeff } (\text{fps-poly-swap2 } p) \ n \ \$ \ m = 0 \ \$ \ m$
by *simp*
qed
qed

lemma *degree-fps-poly-swap2-eq*:
assumes $\bigwedge n. \text{degree } (\text{fps-nth } p \ n) \leq d$
assumes $d > 0 \vee \text{fps-nth } p \ n \neq 0$
assumes $\text{degree } (\text{fps-nth } p \ n) = d$
shows $\text{degree } (\text{fps-poly-swap2 } p) = d$
proof (*rule antisym*)
have $\text{fps-nth } (\text{coeff } (\text{fps-poly-swap2 } p) \ d) \ n = \text{poly.coeff } (\text{fps-nth } p \ n) \ d$
by (*subst fps-nth-coeff-fps-poly-swap2[OF assms(1)]*) *auto*
also have $\dots \neq 0$
using *assms(2,3)* **by** *force*

finally have $\text{coeff } (\text{fps-poly-swap2 } p) d \neq 0$
 by *force*
thus $\text{degree } (\text{fps-poly-swap2 } p) \geq d$
 using *le-degree* by *blast*
next
show $\text{degree } (\text{fps-poly-swap2 } p) \leq d$
 by (*intro degree-fps-poly-swap2-le*) *fact*
qed

definition $\text{reduce-fps-poly} :: 'a :: \text{zero fps poly} \Rightarrow 'a \text{ poly}$ **where**
 $\text{reduce-fps-poly } F = \text{fps-nth } (\text{fps-poly-swap1 } F) 0$

lemma

fixes $F :: 'a :: \text{field fps poly}$
assumes $\text{lead-coeff } F = 1$
shows $\text{degree-reduce-fps-poly-monic: degree } (\text{reduce-fps-poly } F) = \text{degree } F$
and $\text{reduce-fps-poly-monic: lead-coeff } (\text{reduce-fps-poly } F) = 1$
proof –

have $\text{eq1: coeff } (\text{reduce-fps-poly } F) (\text{degree } F) = 1$
unfolding *reduce-fps-poly-def* by (*simp add: asms*)
have $\text{eq2: coeff } (\text{reduce-fps-poly } F) n = 0$ **if** $n > \text{degree } F$ **for** n
unfolding *reduce-fps-poly-def* using *that* by (*simp add: coeff-eq-0*)

have $\text{degree } (\text{reduce-fps-poly } F) \leq \text{degree } F$
 by (*rule degree-le*) (*auto simp: eq2*)
moreover have $\text{degree } (\text{reduce-fps-poly } F) \geq \text{degree } F$
 by (*rule le-degree*) (*simp add: eq1*)
from eq1 eq2 **show** $\text{degree } (\text{reduce-fps-poly } F) = \text{degree } F$
 by (*intro antisym le-degree degree-le*) *auto*
with eq1 **show** $\text{lead-coeff } (\text{reduce-fps-poly } F) = 1$
 by *simp*

qed

locale *fps-hensel-aux* =

fixes $F :: 'a :: \text{field-gcd poly fps}$
fixes $g h :: 'a \text{ poly}$
assumes *coprime: coprime g h* **and** *deg-g: degree g > 0* **and** *deg-h: degree h > 0*
begin

context

fixes $g' h' :: 'a \text{ poly}$
defines $h' \equiv \text{fst } (\text{bezout-coefficients } g h)$ **and** $g' \equiv \text{snd } (\text{bezout-coefficients } g h)$
begin

fun *hensel-fpxs-aux* :: $\text{nat} \Rightarrow 'a \text{ poly} \times 'a \text{ poly}$ **where**

hensel-fpxs-aux $n = (\text{if } n = 0 \text{ then } (g, h) \text{ else}$

(*let*

$U = \text{fps-nth } F n -$

$(\sum (i,j) \mid i < n \wedge j < n \wedge i + j = n. \text{fst } (\text{hensel-fpxs-aux } i) * \text{snd}$

```

(hensel-fpxs-aux j))
  in (U * g' + g * ((U * h') div h), (U * h') mod h)))

lemmas [simp del] = hensel-fpxs-aux.simps

lemma hensel-fpxs-aux-0 [simp]: hensel-fpxs-aux 0 = (g, h)
  by (subst hensel-fpxs-aux.simps) auto

definition hensel-fpxs1 :: 'a poly fps
  where hensel-fpxs1 = Abs-fps (fst ∘ hensel-fpxs-aux)

definition hensel-fpxs2 :: 'a poly fps
  where hensel-fpxs2 = Abs-fps (snd ∘ hensel-fpxs-aux)

lemma hensel-fpxs1-0 [simp]: hensel-fpxs1 $ 0 = g
  by (simp add: hensel-fpxs1-def)

lemma hensel-fpxs2-0 [simp]: hensel-fpxs2 $ 0 = h
  by (simp add: hensel-fpxs2-def)

theorem fps-hensel-aux:
  defines f ≡ fps-nth F 0
  assumes f = g * h
  assumes ∀ n>0. degree (fps-nth F n) < degree f
  defines G ≡ hensel-fpxs1 and H ≡ hensel-fpxs2
  shows F = G * H fps-nth G 0 = g fps-nth H 0 = h
    ∀ n>0. degree (fps-nth G n) < degree g
    ∀ n>0. degree (fps-nth H n) < degree h
proof –
  show fps-nth G 0 = g fps-nth H 0 = h
    by (simp-all add: G-def H-def hensel-fpxs1-def hensel-fpxs2-def)

  have deg-f: degree f = degree g + degree h
    unfolding ⟨f = g * h⟩ using assms by (intro degree-mult-eq) auto

  have deg-H: degree (fps-nth H n) < degree h if ⟨n > 0⟩ for n
  proof (cases snd (hensel-fpxs-aux n) = 0)
    case False
      thus ?thesis
        using deg-h ⟨n > 0⟩
        by (auto simp: hensel-fpxs-aux.simps[of n] hensel-fpxs2-def H-def intro: de-
  gree-mod-less')
    qed (use assms deg-h in ⟨auto simp: hensel-fpxs2-def⟩)
  thus ∀ n>0. degree (fps-nth H n) < degree h
    by blast

  have *: fps-nth F n = fps-nth (G * H) n ∧ (n > 0 ⟶ degree (fps-nth G n) <
  degree g) for n
  proof (induction n rule: less-induct)

```

```

case (less n)
have fin: finite {p. fst p < n ∧ snd p < n ∧ fst p + snd p = n}
  by (rule finite-subset[of - {..n} × {..n}]) auto
show ?case
proof (cases n = 0)
  case True
  thus ?thesis using assms
    by (auto simp: hensel-fpxs1-def hensel-fpxs2-def)
next
  case False
  define U where U = fps-nth F n -
    ( $\sum (i,j) \mid i < n \wedge j < n \wedge i + j = n. \text{fst} (\text{hensel-fpxs-aux } i) * \text{snd}$ 
    (hensel-fpxs-aux j))
  define g'' h'' where g'' = U * g' and h'' = U * h'

  have fps-nth (G * H) n =
    ( $\sum i=0..n. \text{fst} (\text{hensel-fpxs-aux } i) * \text{snd} (\text{hensel-fpxs-aux } (n - i))$ )
  using assms by (auto simp: hensel-fpxs1-def hensel-fpxs2-def fps-mult-nth)
  also have ... = ( $\sum (i,j) \mid i + j = n. \text{fst} (\text{hensel-fpxs-aux } i) * \text{snd} (\text{hensel-fpxs-aux}$ 
    (j))
    by (rule sum.reindex-bij-witness[of - fst  $\lambda i. (i, n - i)$ ]) auto
  also have {(i,j). i + j = n} = {(i,j). i < n ∧ j < n ∧ i + j = n} ∪ {(n,0),
    (0,n)}
    by auto
  also have ( $\sum (i,j) \in \dots \text{fst} (\text{hensel-fpxs-aux } i) * \text{snd} (\text{hensel-fpxs-aux } j) =$ 
    fps-nth F n - U + (fst (hensel-fpxs-aux n) * h + g * snd (hensel-fpxs-aux
    (n))
    using False fin by (subst sum.union-disjoint) (auto simp: case-prod-unfold
    U-def)
  also have eq: fst (hensel-fpxs-aux n) * h + g * snd (hensel-fpxs-aux n) = U
  proof -
  have fst (hensel-fpxs-aux n) * h + g * snd (hensel-fpxs-aux n) =
    (g'' + g * (h'' div h)) * h + g * (h'' mod h)
  using False by (simp add: hensel-fpxs-aux.simps[of n] U-def g''-def h''-def)
  also have h'' mod h = h'' - (h'' div h) * h
    by (simp add: minus-div-mult-eq-mod)
  also have (g'' + g * (h'' div h)) * h + g * (h'' - h'' div h * h) = g * h''
+ g'' * h
    by (simp add: algebra-simps)
  also have ... = U * (h' * g + g' * h)
    by (simp add: algebra-simps g''-def h''-def)
  also have h' * g + g' * h = gcd g h
    unfolding g'-def h'-def by (rule bezout-coefficients-fst-snd)
  also have gcd g h = 1
    using coprime by simp
  finally show ?thesis by simp
qed
finally have fps-nth F n = fps-nth (G * H) n by simp

```

```

have degree (G $ n) < degree g
proof (cases G $ n = 0)
  case False
  have degree (G $ n) + degree h = degree (G $ n * h)
  using False assms by (intro degree-mult-eq [symmetric]) auto
  also from eq have fps-nth G n * h = U - g * snd (hensel-fpxs-aux n)
  by (simp add: algebra-simps G-def hensel-fpxs1-def)
  hence degree (fps-nth G n * h) = degree (U - g * snd (hensel-fpxs-aux n))
  by (simp only: )
  also have ... < degree f
  proof (intro degree-diff-less)
  have degree (g * snd (local.hensel-fpxs-aux n)) ≤
    degree g + degree (snd (local.hensel-fpxs-aux n))
  by (intro degree-mult-le)
  also have degree (snd (local.hensel-fpxs-aux n)) < degree h
  using deg-H[of n] ⟨n ≠ 0⟩ by (auto simp: H-def hensel-fpxs2-def)
  also have degree g + degree h = degree f
  by (subst deg-f) auto
  finally show degree (g * snd (local.hensel-fpxs-aux n)) < degree f
  by simp
next
show degree U < degree f
  unfolding U-def
  proof (intro degree-diff-less degree-sum-less)
  show degree (F $ n) < degree f
  using ⟨n ≠ 0⟩ assms by auto
next
show degree f > 0
  unfolding deg-f using deg-g by simp
next
fix z assume z: z ∈ {(i, j). i < n ∧ j < n ∧ i + j = n}
have degree (case z of (i, j) ⇒ fst (hensel-fpxs-aux i) * snd (hensel-fpxs-aux
j)) =
  degree (fps-nth G (fst z) * fps-nth H (snd z)) (is ?lhs = -)
by (simp add: case-prod-unfold G-def H-def hensel-fpxs1-def hensel-fpxs2-def)
  also have ... ≤ degree (fps-nth G (fst z)) + degree (fps-nth H (snd z))
  by (intro degree-mult-le)
  also have ... < degree g + degree h
  using z less.IH[of fst z]
  by (intro add-strict-mono deg-H) (simp-all add: case-prod-unfold)
  finally show ?lhs < degree f
  by (simp add: deg-f)
qed
qed
finally show ?thesis
  by (simp add: deg-f)
qed (use deg-g in auto)

with ⟨fps-nth F n = fps-nth (G * H) n⟩ show ?thesis

```

```

      by blast
    qed
  qed

  from * show  $F = G * H$  and  $\forall n > 0. \text{degree} (\text{fps-nth } G \ n) < \text{degree } g$ 
    by (auto simp: fps-eq-iff)
  qed

end

end

locale fps-hensel =
  fixes  $F :: 'a :: \text{field-gcd fps poly}$  and  $f \ g \ h :: 'a \ \text{poly}$ 
  assumes monic:  $\text{lead-coeff } F = 1$ 
  defines  $f \equiv \text{reduce-fps-poly } F$ 
  assumes f-splits:  $f = g * h$ 
  assumes coprime:  $\text{coprime } g \ h$  and deg-g:  $\text{degree } g > 0$  and deg-h:  $\text{degree } h > 0$ 
begin

definition  $F'$  where  $F' = \text{fps-poly-swap1 } F$ 

sublocale fps-hensel-aux  $F' \ g \ h$ 
  by unfold-locales (fact deg-g deg-h coprime)+

definition  $G$  where
   $G = \text{fps-poly-swap2 hensel-fpxs1}$ 

definition  $H$  where
   $H = \text{fps-poly-swap2 hensel-fpxs2}$ 

lemma deg-f:  $\text{degree } f = \text{degree } F$ 
proof (intro antisym)
  have  $\text{coeff } f (\text{degree } F) \neq 0$ 
    using monic by (simp add: f-def reduce-fps-poly-def)
  thus  $\text{degree } f \geq (\text{degree } F)$ 
    by (rule le-degree)
next
  have  $\text{coeff } f \ n = 0$  if  $n > \text{degree } F$  for  $n$ 
    using that by (simp add: f-def reduce-fps-poly-def coeff-eq-0)
  thus  $\text{degree } f \leq \text{degree } F$ 
    using degree-le by blast
qed

lemma
   $F\text{-splits: } F = G * H$  and
   $\text{reduce-}G: \text{reduce-fps-poly } G = g$  and

```

reduce-H: *reduce-fps-poly* $H = h$ **and**
deg-G: *degree* $G = \text{degree } g$ **and**
deg-H: *degree* $H = \text{degree } h$ **and**
lead-coeff-G: *lead-coeff* $G = \text{fps-const } (\text{lead-coeff } g)$ **and**
lead-coeff-H: *lead-coeff* $H = \text{fps-const } (\text{lead-coeff } h)$

proof –

from *deg-g deg-h* **have** [*simp*]: $g \neq 0 \ h \neq 0$
by *auto*
define N **where** $N = \text{degree } F$

have *deg-f*: *degree* $f = N$
proof (*intro antisym*)
have *coeff f N* $\neq 0$
using *monic* **by** (*simp add: f-def reduce-fps-poly-def N-def*)
thus *degree f* $\geq N$
by (*rule le-degree*)

next

have *coeff f n* $= 0$ **if** $n > N$ **for** n
using *that* **by** (*simp add: f-def reduce-fps-poly-def N-def coeff-eq-0*)
thus *degree f* $\leq N$
using *degree-le* **by** *blast*

qed

have $F' \$ 0 = f$
unfolding *F'-def f-def reduce-fps-poly-def* ..
have *F'0*: $F' \$ 0 = g * h$
using *f-splits* **by** (*simp add: F'-def f-def reduce-fps-poly-def*)

have $\forall n > 0. \text{degree } (F' \$ n) < N$

proof (*subst F'-def, intro allI impI degree-lessI*)

fix $n :: \text{nat}$

assume $n: n > 0$

show *fps-poly-swap1* $F \$ n \neq 0 \vee 0 < N$

using n *deg-g deg-h f-splits deg-f* **by** (*auto simp: F'0 degree-mult-eq*)

fix k

assume $k: k \geq N$

have *coeff (F' \$ n) k* $= \text{coeff } F \ k \ \$ n$

unfolding *F'-def* **by** *simp*

also have $\dots = 0$

using *monic* $\langle n > 0 \rangle k$ **by** (*cases k > N*) (*auto simp: N-def coeff-eq-0*)

finally show *coeff (fps-poly-swap1 F \$ n) k* $= 0$

by (*simp add: F'-def*)

qed

hence *degs-less*: $\forall n > 0. \text{degree } (F' \$ n) < \text{degree } (F' \$ 0)$

by (*simp add: $\langle F' \$ 0 = f \rangle \text{deg-f}$*)

note *hensel* $= \text{fps-hensel-aux}[OF \ F'0 \ \text{degs-less}]$

have *deg-less1*: *degree* (*hensel-fpxs1* $\$ n$) $< \text{degree } g$ **if** $n > 0$ **for** n

using *hensel(4)* **that** **by** (*simp add: F'-def*)

have *deg-le1*: $\text{degree} (\text{hensel-fpxs1 } \$ n) \leq \text{degree } g$ **for** n
proof (*cases* $n = 0$)
 case *True*
 hence $\text{hensel-fpxs1 } \$ n = g$
 by (*simp add: hensel-fpxs1-def*)
 thus *?thesis* **by** *simp*
qed (*auto intro: less-imp-le deg-less1 simp: f-def*)

have *deg-less2*: $\text{degree} (\text{hensel-fpxs2 } \$ n) < \text{degree } h$ **if** $n > 0$ **for** n
 using *hensel(5)* **that** **by** (*simp add: F'-def*)
have *deg-le2*: $\text{degree} (\text{hensel-fpxs2 } \$ n) \leq \text{degree } h$ **for** n
proof (*cases* $n = 0$)
 case *True*
 hence $\text{hensel-fpxs2 } \$ n = h$
 by (*simp add: hensel-fpxs2-def*)
 thus *?thesis* **by** *simp*
qed (*auto intro: less-imp-le deg-less2 simp: f-def*)

show $F = G * H$
 unfolding *poly-eq-iff fps-eq-iff*
proof *safe*
 fix $n k$
 have $\text{poly.coeff } F n \$ k = \text{poly.coeff } (F' \$ k) n$
 unfolding *F'-def* **by** *simp*
 also have $F' = \text{hensel-fpxs1} * \text{hensel-fpxs2}$
 by (*rule hensel*)
 also have $\dots \$ k = (\sum_{i=0..k} \text{hensel-fpxs1 } \$ i * \text{hensel-fpxs2 } \$ (k - i))$
 unfolding *fps-mult-nth ..*
 also have $\text{poly.coeff } \dots n =$
 $(\sum_{i=0..k} \sum_{j \leq n} \text{coeff } (\text{hensel-fpxs1 } \$ i) j * \text{coeff } (\text{hensel-fpxs2 } \$$
 $(k - i)) (n - j))$
 by (*simp add: coeff-sum coeff-mult*)
 also have $(\lambda i j. \text{coeff } (\text{hensel-fpxs1 } \$ i) j) = (\lambda i j. \text{coeff } G j \$ i)$
 unfolding *G-def*
 by (*subst fps-nth-coeff-fps-poly-swap2[OF deg-le1]*) (*auto simp: F'-def*)
 also have $(\lambda i j. \text{coeff } (\text{hensel-fpxs2 } \$ i) j) = (\lambda i j. \text{coeff } H j \$ i)$
 unfolding *H-def*
 by (*subst fps-nth-coeff-fps-poly-swap2[OF deg-le2]*) (*auto simp: F'-def*)
 also have $(\sum_{i=0..k} \sum_{j \leq n} \text{poly.coeff } G j \$ i * \text{poly.coeff } H (n - j) \$ (k -$
 $i)) =$
 $(\sum_{j \leq n} \sum_{i=0..k} \text{poly.coeff } G j \$ i * \text{poly.coeff } H (n - j) \$ (k - i))$
 by (*rule sum.swap*)
 also have $\dots = \text{poly.coeff } (G * H) n \$ k$
 by (*simp add: coeff-mult fps-mult-nth fps-sum-nth*)
 finally show $\text{poly.coeff } F n \$ k = \text{poly.coeff } (G * H) n \$ k$.
qed

show $\text{reduce-fps-poly } G = g$ **unfolding** *G-def reduce-fps-poly-def poly-eq-iff*
 by (*auto simp: fps-nth-coeff-fps-poly-swap2[OF deg-le1]*)

```

show reduce-fps-poly H = h unfolding H-def reduce-fps-poly-def poly-eq-iff
  by (auto simp: fps-nth-coeff-fps-poly-swap2[OF deg-le2])
show degree G = degree g unfolding G-def
  by (rule degree-fps-poly-swap2-eq[where n = 0] deg-le1 disjI1 deg-g deg-le2)+
simp-all
show degree H = degree h unfolding H-def
  by (rule degree-fps-poly-swap2-eq[where n = 0] deg-le1 disjI1 deg-h deg-le2)+
simp-all

show lead-coeff G = fps-const (lead-coeff g)
proof (rule fps-ext)
  fix n :: nat
  have lead-coeff G $ n = coeff (hensel-fpxs1 $ n) (degree G)
    by (subst G-def, subst fps-nth-coeff-fps-poly-swap2[OF deg-le1]) auto
  also have ... = (if n = 0 then lead-coeff g else 0)
    by (auto simp: ‹degree G = degree g› intro: coeff-eq-0 deg-less1)
  finally show lead-coeff G $ n = fps-const (lead-coeff g) $ n
    by simp
qed

show lead-coeff H = fps-const (lead-coeff h)
proof (rule fps-ext)
  fix n :: nat
  have lead-coeff H $ n = coeff (hensel-fpxs2 $ n) (degree H)
    by (subst H-def, subst fps-nth-coeff-fps-poly-swap2[OF deg-le2]) auto
  also have ... = (if n = 0 then lead-coeff h else 0)
    by (auto simp: ‹degree H = degree h› intro: coeff-eq-0 deg-less2)
  finally show lead-coeff H $ n = fps-const (lead-coeff h) $ n
    by simp
qed
qed
end
end

```

3 Formal Puiseux Series

```

theory Formal-Puiseux-Series
  imports FPS-Hensel
begin

```

3.1 Auxiliary facts and definitions

```

lemma div-dvd-self:
  fixes a b :: 'a :: {semidom-divide}
  shows b dvd a  $\implies$  a div b dvd a
  by (elim dvdE; cases b = 0) simp-all

```

lemma *quotient-of-int [simp]: quotient-of (of-int n) = (n, 1)*
using *Rat.of-int-def quotient-of-int by auto*

lemma *of-int-div-of-int-in-Ints-iff:*
(of-int n / of-int m :: 'a :: field-char-0) ∈ ℤ ↔ m = 0 ∨ m dvd n

proof

assume *: *(of-int n / of-int m :: 'a) ∈ ℤ*
{
assume *m ≠ 0*
from * **obtain** *k* **where** *k: (of-int n / of-int m :: 'a) = of-int k*
by *(auto elim!: Ints-cases)*
hence *of-int n = (of-int k * of-int m :: 'a)*
using *⟨m ≠ 0⟩ by (simp add: field-simps)*
also have *... = of-int (k * m)*
by *simp*
finally have *n = k * m*
by *(subst (asm) of-int-eq-iff)*
hence *m dvd n by auto*
}
thus *m = 0 ∨ m dvd n by blast*

qed *auto*

lemma *rat-eq-quotientD:*

assumes *r = rat-of-int a / rat-of-int b b ≠ 0*
shows *fst (quotient-of r) dvd a snd (quotient-of r) dvd b*

proof –

define *a' b'* **where** *a' = fst (quotient-of r) and b' = snd (quotient-of r)*
define *d* **where** *d = gcd a b*
have *b' > 0*
by *(auto simp: b'-def quotient-of-denom-pos')*

have *coprime a' b'*
by *(rule quotient-of-coprime[of r]) (simp add: a'-def b'-def)*

have *r: r = rat-of-int a' / rat-of-int b'*
by *(simp add: a'-def b'-def quotient-of-div)*
from *assms ⟨b' > 0⟩ have rat-of-int (a' * b) = rat-of-int (a * b')*
unfolding *of-int-mult by (simp add: field-simps r)*

hence *eq: a' * b = a * b'*
by *(subst (asm) of-int-eq-iff)*

have *a' dvd a * b'*
by *(simp flip: eq)*
hence *a' dvd a*
by *(subst (asm) coprime-dvd-mult-left-iff) fact*

moreover have *b' dvd a' * b*
by *(simp add: eq)*

hence *b' dvd b*
by *(subst (asm) coprime-dvd-mult-right-iff) (use ⟨coprime a' b'⟩ in ⟨simp add: coprime-commute⟩)*

ultimately show $\text{fst } (\text{quotient-of } r) \text{ dvd } a \text{ snd } (\text{quotient-of } r) \text{ dvd } b$
unfolding $a\text{'-def } b\text{'-def}$ **by** blast+
qed

lemma *quotient-of-denom-add-dvd*:

$\text{snd } (\text{quotient-of } (x + y)) \text{ dvd } \text{snd } (\text{quotient-of } x) * \text{snd } (\text{quotient-of } y)$

proof –

define $a \ b$ **where** $a = \text{fst } (\text{quotient-of } x)$ **and** $b = \text{snd } (\text{quotient-of } x)$

define $c \ d$ **where** $c = \text{fst } (\text{quotient-of } y)$ **and** $d = \text{snd } (\text{quotient-of } y)$

have $b > 0 \ d > 0$

by (*auto simp: b-def d-def quotient-of-denom-pos'*)

have $xy: x = \text{rat-of-int } a / \text{rat-of-int } b \ y = \text{rat-of-int } c / \text{rat-of-int } d$

unfolding $a\text{-def } b\text{-def } c\text{-def } d\text{-def}$ **by** (*simp-all add: quotient-of-div*)

show $\text{snd } (\text{quotient-of } (x + y)) \text{ dvd } b * d$

proof (*rule rat-eq-quotientD*)

show $x + y = \text{rat-of-int } (a * d + c * b) / \text{rat-of-int } (b * d)$

using $\langle b > 0 \rangle \langle d > 0 \rangle$ **by** (*simp add: field-simps xy*)

qed (*use* $\langle b > 0 \rangle \langle d > 0 \rangle$ **in** *auto*)

qed

lemma *quotient-of-denom-diff-dvd*:

$\text{snd } (\text{quotient-of } (x - y)) \text{ dvd } \text{snd } (\text{quotient-of } x) * \text{snd } (\text{quotient-of } y)$

using *quotient-of-denom-add-dvd*[*of* $x - y$]

by (*simp add: rat-uminus-code Let-def case-prod-unfold*)

definition $\text{supp} :: ('a \Rightarrow ('b :: \text{zero})) \Rightarrow 'a \text{ set}$ **where**

$\text{supp } f = f^{-1} \{-\{0\}\}$

lemma *supp-0* [*simp*]: $\text{supp } (\lambda-. 0) = \{\}$

and *supp-const*: $\text{supp } (\lambda-. c) = (\text{if } c = 0 \text{ then } \{\} \text{ else } \text{UNIV})$

and *supp-singleton* [*simp*]: $c \neq 0 \implies \text{supp } (\lambda x. \text{if } x = d \text{ then } c \text{ else } 0) = \{d\}$

by (*auto simp: supp-def*)

lemma *supp-uminus* [*simp*]: $\text{supp } (\lambda x. -f \ x :: 'a :: \text{group-add}) = \text{supp } f$

by (*auto simp: supp-def*)

3.2 Definition

Similarly to formal power series $R[[X]]$ and formal Laurent series $R((X))$, we define the ring of formal Puiseux series $R\{\{X\}\}$ as functions from the rationals into a ring such that

1. the support is bounded from below, and
2. the denominators of the numbers in the support have a common multiple other than 0

One can also think of a formal Puiseux series in the parameter X as a formal Laurent series in the parameter $X^{1/d}$ for some positive integer d . This is often written in the following suggestive notation:

$$R\{\{X\}\} = \bigcup_{d \geq 1} R((X^{1/d}))$$

Many operations will be defined in terms of this correspondence between Puiseux and Laurent series, and many of the simple properties proven that way.

definition *is-fpxs* :: (rat \Rightarrow 'a :: zero) \Rightarrow bool **where**
is-fpxs f \longleftrightarrow bdd-below (supp f) \wedge (LCM r \in supp f. snd (quotient-of r)) \neq 0

typedef (overloaded) 'a fpxs = {f::rat \Rightarrow 'a :: zero. *is-fpxs* f}
morphisms fpxs-nth Abs-fpxs
by (rule exI[of - λ -. 0]) (auto simp: *is-fpxs-def* *supp-def*)

setup-lifting *type-definition-fpxs*

lemma *fpxs-ext*: (\bigwedge r. fpxs-nth f r = fpxs-nth g r) \implies f = g
by transfer auto

lemma *fpxs-eq-iff*: f = g \longleftrightarrow (\forall r. fpxs-nth f r = fpxs-nth g r)
by transfer auto

lift-definition *fpxs-supp* :: 'a :: zero fpxs \Rightarrow rat set **is** supp .

lemma *fpxs-supp-altdef*: *fpxs-supp* f = {x. fpxs-nth f x \neq 0}
by transfer (auto simp: *supp-def*)

The following gives us the “root order” of f , i.e. the smallest positive integer d such that f is in $R((X^{1/p}))$.

lift-definition *fpxs-root-order* :: 'a :: zero fpxs \Rightarrow nat **is**
 λ f. nat (LCM r \in supp f. snd (quotient-of r)) .

lemma *fpxs-root-order-pos* [simp]: *fpxs-root-order* f > 0

proof transfer

fix f :: rat \Rightarrow 'a **assume** f: *is-fpxs* f

hence (LCM r \in supp f. snd (quotient-of r)) \neq 0

by (auto simp: *is-fpxs-def*)

moreover have (LCM r \in supp f. snd (quotient-of r)) \geq 0

by simp

ultimately show nat (LCM r \in supp f. snd (quotient-of r)) > 0

by linarith

qed

lemma *fpxs-root-order-nonzero* [simp]: *fpxs-root-order* f \neq 0
using *fpxs-root-order-pos*[of f] **by** linarith

Let d denote the root order of a Puiseux series f , i.e. the smallest number d such that all monomials with non-zero coefficients can be written in the form $X^{n/d}$ for some n . Then f can be written as a Laurent series in $X^{\mathbb{Z}/d}$. The following operation gives us this Laurent series.

lift-definition $fls\text{-of-fpxs} :: 'a :: zero\ fpxs \Rightarrow 'a\ fls\ is$
 $\lambda f\ n.\ f\ (of\text{-int}\ n\ /\ of\text{-int}\ (LCM\ r \in supp\ f.\ snd\ (quotient\text{-of}\ r)))$
proof –
fix $f :: rat \Rightarrow 'a$
assume $f: is\text{-fpxs}\ f$
hence $bdd\text{-below}\ (supp\ f)$
by $(auto\ simp: is\text{-fpxs}\text{-def})$
then obtain $r0$ **where** $\forall x \in supp\ f.\ r0 \leq x$
by $(auto\ simp: bdd\text{-below}\text{-def})$
hence $r0: f\ x = 0$ **if** $x < r0$ **for** x
using that by $(auto\ simp: supp\text{-def}\ vimage\text{-def})$
define $d :: int$ **where** $d = (LCM\ r \in supp\ f.\ snd\ (quotient\text{-of}\ r))$
have $d \geq 0$ **by** $(simp\ add: d\text{-def})$
moreover have $d \neq 0$
using f **by** $(auto\ simp: d\text{-def}\ is\text{-fpxs}\text{-def})$
ultimately have $d > 0$ **by** $linarith$

have $*$: $f\ (of\text{-int}\ n\ /\ of\text{-int}\ d) = 0$ **if** $n < \lfloor r0 * of\text{-int}\ d \rfloor$ **for** n
proof –
have $rat\text{-of-int}\ n < r0 * rat\text{-of-int}\ d$
using that by $linarith$
thus $?thesis$
using $\langle d > 0 \rangle$ **by** $(intro\ r0)\ (auto\ simp: field\text{-simps})$

qed
have $eventually\ (\lambda n.\ n > -\lfloor r0 * of\text{-int}\ d \rfloor)$ $at\text{-top}$
by $(rule\ eventually\text{-gt}\text{-at}\text{-top})$
hence $eventually\ (\lambda n.\ f\ (of\text{-int}\ (-n)\ /\ of\text{-int}\ d) = 0)$ $at\text{-top}$
by $(eventually\text{-elim})\ (rule\ *,\ auto)$
hence $eventually\ (\lambda n.\ f\ (of\text{-int}\ (-int\ n)\ /\ of\text{-int}\ d) = 0)$ $at\text{-top}$
by $(rule\ eventually\text{-compose}\text{-filterlim})\ (rule\ filterlim\text{-int}\text{-sequentially})$
thus $eventually\ (\lambda n.\ f\ (of\text{-int}\ (-int\ n)\ /\ of\text{-int}\ d) = 0)$ $cofinite$
by $(simp\ add: cofinite\text{-eq}\text{-sequentially})$

qed

lemma $fls\text{-nth}\text{-of-fpxs}$:

$fls\text{-nth}\ (fls\text{-of-fpxs}\ f)\ n = fpxs\text{-nth}\ f\ (of\text{-int}\ n\ /\ of\text{-nat}\ (fpxs\text{-root-order}\ f))$
by $transfer\ simp$

3.3 Basic algebraic typeclass instances

instantiation $fpxs :: (zero)\ zero$
begin

lift-definition $zero\text{-fpxs} :: 'a\ fpxs\ is\ \lambda r :: rat.\ 0 :: 'a$
by $(auto\ simp: is\text{-fpxs}\text{-def}\ supp\text{-def})$

```

instance ..

end

instantiation fpxs :: ({one, zero}) one
begin

lift-definition one-fpxs :: 'a fpxs is  $\lambda r::rat. \text{if } r = 0 \text{ then } 1 \text{ else } 0 :: 'a$ 
  by (cases (1 :: 'a) = 0) (auto simp: is-fpxs-def cong: if-cong)

instance ..

end

lemma fls-of-fpxs-0 [simp]: fls-of-fpxs 0 = 0
  by transfer auto

lemma fpxs-nth-0 [simp]: fpxs-nth 0 r = 0
  by transfer auto

lemma fpxs-nth-1: fpxs-nth 1 r = (if r = 0 then 1 else 0)
  by transfer auto

lemma fpxs-nth-1': fpxs-nth 1 0 = 1 r  $\neq$  0  $\implies$  fpxs-nth 1 r = 0
  by (auto simp: fpxs-nth-1)

instantiation fpxs :: (monoid-add) monoid-add
begin

lift-definition plus-fpxs :: 'a fpxs  $\Rightarrow$  'a fpxs  $\Rightarrow$  'a fpxs is
   $\lambda f g x. f x + g x$ 
proof -
  fix f g :: rat  $\Rightarrow$  'a
  assume fg: is-fpxs f is-fpxs g
  show is-fpxs ( $\lambda x. f x + g x$ )
    unfolding is-fpxs-def
  proof
    have supp: supp ( $\lambda x. f x + g x$ )  $\subseteq$  supp f  $\cup$  supp g
      by (auto simp: supp-def)
    show bdd-below (supp ( $\lambda x. f x + g x$ ))
      by (rule bdd-below-mono[OF - supp]) (use fg in  $\langle$ auto simp: is-fpxs-def $\rangle$ )
    have (LCM  $r \in$  supp ( $\lambda x. f x + g x$ ), snd (quotient-of r)) dvd
      (LCM  $r \in$  supp f  $\cup$  supp g, snd (quotient-of r))
      by (intro Lcm-subset image-mono supp)
    also have ... = lcm (LCM  $r \in$  supp f, snd (quotient-of r)) (LCM  $r \in$  supp g,
  snd (quotient-of r))
    unfolding image-Un Lcm-Un ..
    finally have (LCM  $r \in$  supp ( $\lambda x. f x + g x$ ), snd (quotient-of r)) dvd

```

$lcm (LCM r \in \text{supp } f. \text{snd } (\text{quotient-of } r)) (LCM r \in \text{supp } g. \text{snd } (\text{quotient-of } r))$.
moreover have $lcm (LCM r \in \text{supp } f. \text{snd } (\text{quotient-of } r)) (LCM r \in \text{supp } g. \text{snd } (\text{quotient-of } r)) \neq 0$
using fg **by** $(\text{auto simp: is-fpxs-def})$
ultimately show $(LCM r \in \text{supp } (\lambda x. f x + g x). \text{snd } (\text{quotient-of } r)) \neq 0$
by auto
qed
qed

instance

by $\text{standard } (\text{transfer; simp add: algebra-simps fun-eq-iff})+$

end

instance $fpxs :: (\text{comm-monoid-add}) \text{comm-monoid-add}$

proof

fix $f g :: 'a \text{ fpxs}$

show $f + g = g + f$

by $\text{transfer } (\text{auto simp: add-ac})$

qed simp-all

lemma $fpxs\text{-nth-add } [\text{simp}]: fpxs\text{-nth } (f + g) r = fpxs\text{-nth } f r + fpxs\text{-nth } g r$

by transfer auto

lift-definition $fpxs\text{-of-fls} :: 'a :: \text{zero fls} \Rightarrow 'a \text{ fpxs}$ **is**

$\lambda f r. \text{if } r \in \mathbb{Z} \text{ then } f \lfloor r \rfloor \text{ else } 0$

proof $-$

fix $f :: \text{int} \Rightarrow 'a$

assume $\text{eventually } (\lambda n. f (-\text{int } n) = 0) \text{ cofinite}$

hence $\text{eventually } (\lambda n. f (-\text{int } n) = 0) \text{ at-top}$

by $(\text{simp add: cofinite-eq-sequentially})$

then obtain N **where** $N: f (-\text{int } n) = 0$ **if** $n \geq N$ **for** n

by $(\text{auto simp: eventually-at-top-linorder})$

show $\text{is-fpxs } (\lambda r. \text{if } r \in \mathbb{Z} \text{ then } f \lfloor r \rfloor \text{ else } 0)$

unfolding is-fpxs-def

proof

have $\text{bdd-below } \{-\text{of-nat } N :: \text{rat}..\}$

by simp

moreover have $\text{supp } (\lambda r :: \text{rat}. \text{if } r \in \mathbb{Z} \text{ then } f \lfloor r \rfloor \text{ else } 0) \subseteq \{-\text{of-nat } N..\}$

proof

fix $r :: \text{rat}$ **assume** $r \in \text{supp } (\lambda r. \text{if } r \in \mathbb{Z} \text{ then } f \lfloor r \rfloor \text{ else } 0)$

then obtain m **where** $[\text{simp}]: r = \text{of-int } m / m \neq 0$

by $(\text{auto simp: supp-def elim!: Ints-cases split: if-splits})$

have $m \geq -\text{int } N$

using $N[\text{of nat } (-m)]$ **by** $(\text{cases } m \geq 0; \text{cases } -\text{int } N \leq m) (\text{auto simp: le-nat-iff})$

thus $r \in \{-\text{of-nat } N..\}$ **by** simp

qed
ultimately show *bdd-below* (*supp* ($\lambda r::\text{rat. if } r \in \mathbf{Z} \text{ then } f \lfloor r \rfloor \text{ else } 0$))
by (*rule bdd-below-mono*)
next
have (*LCM* $r \in \text{supp } (\lambda r. \text{if } r \in \mathbf{Z} \text{ then } f \lfloor r \rfloor \text{ else } 0)$). *snd* (*quotient-of* r)) *dvd* 1
by (*intro Lcm-least*) (*auto simp: supp-def elim!: Ints-cases split: if-splits*)
thus (*LCM* $r \in \text{supp } (\lambda r. \text{if } r \in \mathbf{Z} \text{ then } f \lfloor r \rfloor \text{ else } 0)$). *snd* (*quotient-of* r)) $\neq 0$
by (*intro notI*) *simp*
qed
qed

instantiation *fps* :: (*group-add*) *group-add*
begin

lift-definition *uminus-fps* :: 'a *fps* \Rightarrow 'a *fps* **is** $\lambda f x. -f x$
by (*auto simp: is-fps-def*)

definition *minus-fps* :: 'a *fps* \Rightarrow 'a *fps* \Rightarrow 'a *fps* **where**
minus-fps $f g = f + (-g)$

instance proof
fix $f :: 'a \text{ fps}$
show $-f + f = 0$
by *transfer auto*
qed (*auto simp: minus-fps-def*)

end

lemma *fps-nth-uminus* [*simp*]: *fps-nth* $(-f) r = -\text{fps-nth } f r$
by *transfer auto*

lemma *fps-nth-minus* [*simp*]: *fps-nth* $(f - g) r = \text{fps-nth } f r - \text{fps-nth } g r$
unfolding *minus-fps-def fps-nth-add fps-nth-uminus* **by** *simp*

lemma *fps-of-fls-eq-iff* [*simp*]: *fps-of-fls* $f = \text{fps-of-fls } g \iff f = g$
by *transfer (force simp: fun-eq-iff Ints-def)*

lemma *fps-of-fls-0* [*simp*]: *fps-of-fls* 0 = 0
by *transfer auto*

lemma *fps-of-fls-1* [*simp*]: *fps-of-fls* 1 = 1
by *transfer (auto simp: fun-eq-iff elim!: Ints-cases)*

lemma *fps-of-fls-add* [*simp*]: *fps-of-fls* $(f + g) = \text{fps-of-fls } f + \text{fps-of-fls } g$
by *transfer (auto simp: fun-eq-iff elim!: Ints-cases)*

lemma *fps-to-fls-sum* [*simp*]: *fps-to-fls* $(\text{sum } f A) = (\sum_{x \in A}. \text{fps-to-fls } (f x))$
by (*induction A rule: infinite-finite-induct*) *auto*

lemma *fpxs-of-fls-sum* [*simp*]: $\text{fpxs-of-fls } (\text{sum } f \ A) = (\sum_{x \in A}. \text{fpxs-of-fls } (f \ x))$
by (*induction A rule: infinite-finite-induct*) *auto*

lemma *fpxs-nth-of-fls*:
 $\text{fpxs-nth } (\text{fpxs-of-fls } f) \ r = (\text{if } r \in \mathbb{Z} \text{ then } \text{fls-nth } f \ [r] \ \text{else } 0)$
by *transfer auto*

lemma *fpxs-of-fls-eq-0-iff* [*simp*]: $\text{fpxs-of-fls } f = 0 \iff f = 0$
using *fpxs-of-fls-eq-iff*[*of f 0*] **by** (*simp del: fpxs-of-fls-eq-iff*)

lemma *fpxs-of-fls-eq-1-iff* [*simp*]: $\text{fpxs-of-fls } f = 1 \iff f = 1$
using *fpxs-of-fls-eq-iff*[*of f 1*] **by** (*simp del: fpxs-of-fls-eq-iff*)

lemma *fpxs-root-order-of-fls* [*simp*]: $\text{fpxs-root-order } (\text{fpxs-of-fls } f) = 1$

proof (*transfer, goal-cases*)

case (*1 f*)

have $\text{supp } (\lambda r. \text{if } r \in \mathbb{Z} \text{ then } f \ [r] \ \text{else } 0) = \text{rat-of-int } \{n. f \ n \neq 0\}$
by (*force simp: supp-def Ints-def*)

also have $(\text{LCM } r \in \dots \ \text{snd } (\text{quotient-of } r)) = \text{nat } (\text{LCM } x \in \{n. f \ n \neq 0\}. \ 1)$
by (*simp add: image-image*)

also have $\dots = 1$

by *simp*

also have $\text{nat } 1 = 1$

by *simp*

finally show *?case .*

qed

3.4 The substitution $X \mapsto X^r$

This operation turns a formal Puiseux series $f(X)$ into $f(X^r)$, where r can be any positive rational number:

lift-definition *fpxs-compose-power* :: $'a :: \text{zero fpxs} \Rightarrow \text{rat} \Rightarrow 'a \ \text{fpxs}$ **is**
 $\lambda f \ r \ x. \text{if } r > 0 \text{ then } f \ (x / r) \ \text{else } 0$

proof –

fix $f :: \text{rat} \Rightarrow 'a$ **and** $r :: \text{rat}$

assume $f: \text{is-fpxs } f$

have $\text{is-fpxs } (\lambda x. f \ (x / r))$ **if** $r > 0$

unfolding *is-fpxs-def*

proof

define r' **where** $r' = \text{inverse } r$

have $r' > 0$

using $\langle r > 0 \rangle$ **by** (*auto simp: r'-def*)

have $(\lambda x. x / r') \text{ 'supp } f = \text{supp } (\lambda x. f \ (x * r'))$

using $\langle r' > 0 \rangle$ **by** (*auto simp: supp-def image-iff vimage-def field-simps*)

hence eq: $(\lambda x. x * r) \text{ 'supp } f = \text{supp } (\lambda x. f \ (x / r))$

using $\langle r > 0 \rangle$ **by** (*simp add: r'-def field-simps*)

from f **have** *bdd-below (supp f)*

by (*auto simp: is-fpxs-def*)

hence *bdd-below* $((\lambda x. x * r) \text{ 'supp } f)$
using $\langle r > 0 \rangle$ **by** (*intro bdd-below-image-mono*) (*auto simp: mono-def divide-right-mono*)
also note *eq*
finally show *bdd-below* (*supp* $(\lambda x. f (x / r))$) .

define *a b* **where** $a = \text{fst } (\text{quotient-of } r)$ **and** $b = \text{snd } (\text{quotient-of } r)$
have $b > 0$ **by** (*simp add: b-def quotient-of-denom-pos'*)
have [*simp*]: *quotient-of* $r = (a, b)$
by (*simp add: a-def b-def*)
have $r = \text{of-int } a / \text{of-int } b$
by (*simp add: quotient-of-div*)
with $\langle r > 0 \rangle$ **and** $\langle b > 0 \rangle$ **have** $\langle a > 0 \rangle$
by (*simp add: field-simps*)

have $(\text{LCM } r \in \text{supp } (\lambda x. f (x / r)). \text{snd } (\text{quotient-of } r)) =$
 $(\text{LCM } x \in \text{supp } f. \text{snd } (\text{quotient-of } (x * r)))$
by (*simp add: eq [symmetric] image-image*)
also have ... *dvd* $(\text{LCM } x \in \text{supp } f. \text{snd } (\text{quotient-of } x) * b)$
using $\langle a > 0 \rangle \langle b > 0 \rangle$
by (*intro Lcm-mono*)
(simp add: rat-times-code case-prod-unfold Let-def Rat.normalize-def quotient-of-denom-pos' div-dvd-self)
also have ... *dvd* *normalize* $(b * (\text{LCM } x \in \text{supp } f. \text{snd } (\text{quotient-of } x)))$
proof (*cases* *supp* $f = \{\}$)
case *False*
thus *?thesis* **using** *Lcm-mult*[*of* $(\lambda x. \text{snd } (\text{quotient-of } x)) \text{ 'supp } f$ *b*]
by (*simp add: mult-ac image-image*)
qed *auto*
hence $(\text{LCM } x \in \text{supp } f. \text{snd } (\text{quotient-of } x) * b) \text{ dvd}$
 $b * (\text{LCM } x \in \text{supp } f. \text{snd } (\text{quotient-of } x))$ **by** *simp*
finally show $(\text{LCM } r \in \text{supp } (\lambda x. f (x / r)). \text{snd } (\text{quotient-of } r)) \neq 0$
using $\langle b > 0 \rangle$ *f* **by** (*auto simp: is-fpxs-def*)

qed
thus *is-fpxs* $(\lambda x. \text{if } r > 0 \text{ then } f (x / r) \text{ else } 0)$
by (*cases* $r > 0$) (*auto simp: is-fpxs-def supp-def*)

qed

lemma *fpxs-as-fls*:

fpxs-compose-power (*fpxs-of-fls* (*fls-of-fpxs* f)) $(1 / \text{of-nat } (\text{fpxs-root-order } f)) =$
 f

proof (*transfer, goal-cases*)

case $(1 f)$

define *d* **where** $d = (\text{LCM } r \in \text{supp } f. \text{snd } (\text{quotient-of } r))$

have $d \geq 0$ **by** (*simp add: d-def*)

moreover have $d \neq 0$ **using** 1 **by** (*simp add: is-fpxs-def d-def*)

ultimately have $d > 0$ **by** *linarith*

have (*if* *rat-of-int* $d * x \in \mathbb{Z}$ *then* *f* $(\text{rat-of-int } \lfloor \text{rat-of-int } d * x \rfloor / \text{rat-of-int } d)$)

```

else 0) = f x for x
  proof (cases rat-of-int d * x ∈ ℤ)
    case True
      then obtain n where n: rat-of-int d * x = of-int n
        by (auto elim!: Ints-cases)
      have f (rat-of-int [rat-of-int d * x] / rat-of-int d) = f (rat-of-int n / rat-of-int
d)
        by (simp add: n)
      also have rat-of-int n / rat-of-int d = x
        using n ⟨d > 0⟩ by (simp add: field-simps)
      finally show ?thesis
        using True by simp
    next
      case False
      have x ∉ supp f
      proof
        assume x ∈ supp f
        hence snd (quotient-of x) dvd d
          by (simp add: d-def)
        hence rat-of-int (fst (quotient-of x) * d) / rat-of-int (snd (quotient-of x)) ∈
ℤ
          by (intro of-int-divide-in-Ints) auto
        also have rat-of-int (fst (quotient-of x) * d) / rat-of-int (snd (quotient-of x))
=
          rat-of-int d * (rat-of-int (fst (quotient-of x)) / rat-of-int (snd
(quotient-of x)))
          by (simp only: of-int-mult mult-ac times-divide-eq-right)
        also have ... = rat-of-int d * x
          by (metis Fract-of-int-quotient Rat-cases normalize-stable prod.sel(1) prod.sel(2)
quotient-of-Fract)
        finally have rat-of-int d * x ∈ ℤ .
          with False show False by contradiction
      qed
      thus ?thesis using False by (simp add: supp-def)
    qed
  thus ?case
    using ⟨d > 0⟩ by (simp add: is-fpxs-def d-def mult-ac fun-eq-iff cong: if-cong)
qed

lemma fpxs-compose-power-0 [simp]: fpxs-compose-power 0 r = 0
  by transfer simp

lemma fpxs-compose-power-1 [simp]: r > 0 ⇒ fpxs-compose-power 1 r = 1
  by transfer (auto simp: fun-eq-iff)

lemma fls-of-fpxs-eq-0-iff [simp]: fls-of-fpxs x = 0 ⟷ x = 0
  by (metis fls-of-fpxs-0 fpxs-as-fls fpxs-compose-power-0 fpxs-of-fls-0)

lemma fpxs-of-fls-compose-power [simp]:

```

$\text{fpxs-of-fls (fls-compose-power f d) = fpxs-compose-power (fpxs-of-fls f) (of-nat d)}$
proof (*transfer, goal-cases*)
case ($1\ f\ d$)
show *?case*
proof (*cases d = 0*)
case *False*
show *?thesis*
proof (*intro ext, goal-cases*)
case ($1\ r$)
show *?case*
proof (*cases r ∈ ℤ*)
case *True*
then obtain n **where** [*simp*]: $r = \text{of-int } n$
by (*cases r rule: Ints-cases*)
show *?thesis*
proof (*cases d dvd n*)
case *True*
thus *?thesis* **by** (*auto elim!: Ints-cases*)
next
case *False*
hence $\text{rat-of-int } n / \text{rat-of-int (int d)} \notin \mathbb{Z}$
using $\langle d \neq 0 \rangle$ **by** (*subst of-int-div-of-int-in-Ints-iff*) *auto*
thus *?thesis* **using** *False* **by** *auto*
qed
next
case *False*
hence $r / \text{rat-of-nat } d \notin \mathbb{Z}$
using $\langle d \neq 0 \rangle$ **by** (*auto elim!: Ints-cases simp: field-simps*)
thus *?thesis* **using** *False* **by** *auto*
qed
qed
qed *auto*
qed

lemma *fpxs-compose-power-add* [*simp*]:
 $\text{fpxs-compose-power (f + g) r = fpxs-compose-power f r + fpxs-compose-power g r}$
by *transfer (auto simp: fun-eq-iff)*

lemma *fpxs-compose-power-distrib*:
 $r1 > 0 \vee r2 > 0 \implies$
 $\text{fpxs-compose-power (fpxs-compose-power f r1) r2 = fpxs-compose-power f (r1 * r2)}$
by *transfer (auto simp: fun-eq-iff algebra-simps zero-less-mult-iff)*

lemma *fpxs-compose-power-divide-right*:
 $r1 > 0 \implies r2 > 0 \implies$
 $\text{fpxs-compose-power f (r1 / r2) = fpxs-compose-power (fpxs-compose-power f r1) (inverse r2)}$

by (simp add: fpxs-compose-power-distrib field-simps)

lemma *fpxs-compose-power-1-right* [simp]: *fpxs-compose-power* f 1 = f
by transfer auto

lemma *fpxs-compose-power-eq-iff* [simp]:
assumes $r > 0$
shows *fpxs-compose-power* f r = *fpxs-compose-power* g r \longleftrightarrow $f = g$
using *assms*
proof (transfer, goal-cases)
case (1 r f g)
have f x = g x **if** $\bigwedge x. f (x / r) = g (x / r)$ **for** x
using *that*[of $x * r$] $\langle r > 0 \rangle$ **by** auto
thus ?case **using** $\langle r > 0 \rangle$ **by** (auto simp: fun-eq-iff)
qed

lemma *fpxs-compose-power-eq-1-iff* [simp]:
assumes $l > 0$
shows *fpxs-compose-power* p l = 1 \longleftrightarrow $p = 1$
proof –
have *fpxs-compose-power* p l = 1 \longleftrightarrow *fpxs-compose-power* p l = *fpxs-compose-power* 1 l
by (subst *fpxs-compose-power-1*) (use *assms* **in** auto)
also have ... \longleftrightarrow $p = 1$
using *assms* **by** (subst *fpxs-compose-power-eq-iff*) auto
finally show ?thesis .
qed

lemma *fpxs-compose-power-eq-0-iff* [simp]:
assumes $r > 0$
shows *fpxs-compose-power* f r = 0 \longleftrightarrow $f = 0$
using *fpxs-compose-power-eq-iff*[of r f 0] *assms* **by** (simp del: *fpxs-compose-power-eq-iff*)

lemma *fls-of-fpxs-of-fls* [simp]: *fls-of-fpxs* (*fpxs-of-fls* f) = f
using *fpxs-as-fls*[of *fpxs-of-fls* f] **by** simp

lemma *fpxs-as-fls'*:
assumes *fpxs-root-order* f *dvd* d $d > 0$
obtains f' **where** $f = \textit{fpxs-compose-power}$ (*fpxs-of-fls* f') (1 / of-nat d)
proof –
define D **where** $D = \textit{fpxs-root-order}$ f
have $D > 0$
by (auto simp: *D-def*)
define f' **where** $f' = \textit{fls-of-fpxs}$ f
from *assms* **obtain** d' **where** $d': d = D * d'$
by (auto simp: *D-def*)
have $d' > 0$
using *assms* **by** (auto intro!: *Nat.gr0I* simp: d')
define f'' **where** $f'' = \textit{fls-compose-power}$ f' d'

```

have fpxs-compose-power (fpxs-of-fls f'') (1 / of-nat d) = f
  using ⟨D > 0⟩ ⟨d' > 0⟩
  by (simp add: d' D-def f''-def f'-def fpxs-as-fls fpxs-compose-power-distrib)
thus ?thesis using that[of f''] by blast
qed

```

3.5 Multiplication and ring properties

```

instantiation fpxs :: (comm-semiring-1) comm-semiring-1
begin

```

```

lift-definition times-fpxs :: 'a fpxs ⇒ 'a fpxs ⇒ 'a fpxs is

```

```

  λf g x. (∑ (y,z) | y ∈ supp f ∧ z ∈ supp g ∧ x = y + z. f y * g z)

```

```

proof -

```

```

  fix f g :: rat ⇒ 'a

```

```

  assume fg: is-fpxs f is-fpxs g

```

```

  show is-fpxs (λx. ∑ (y,z) | y ∈ supp f ∧ z ∈ supp g ∧ x = y + z. f y * g z)

```

```

    (is is-fpxs ?h) unfolding is-fpxs-def

```

```

  proof

```

```

    from fg obtain bnd1 bnd2 where bnds: ∀ x ∈ supp f. x ≥ bnd1 ∀ x ∈ supp g. x
    ≥ bnd2

```

```

    by (auto simp: is-fpxs-def bdd-below-def)

```

```

  have supp ?h ⊆ (λ(x,y). x + y) ` (supp f × supp g)

```

```

  proof

```

```

    fix x :: rat

```

```

    assume x ∈ supp ?h

```

```

    have {(y,z). y ∈ supp f ∧ z ∈ supp g ∧ x = y + z} ≠ {}

```

```

  proof

```

```

    assume eq: {(y,z). y ∈ supp f ∧ z ∈ supp g ∧ x = y + z} = {}

```

```

    hence ?h x = 0

```

```

    by (simp only:) auto

```

```

    with ⟨x ∈ supp ?h⟩ show False by (auto simp: supp-def)

```

```

  qed

```

```

  thus x ∈ (λ(x,y). x + y) ` (supp f × supp g)

```

```

    by auto

```

```

  qed

```

```

  also have ... ⊆ {bnd1 + bnd2..}

```

```

  using bnds by (auto intro: add-mono)

```

```

  finally show bdd-below (supp ?h)

```

```

    by auto

```

```

next

```

```

define d1 where d1 = (LCM r ∈ supp f. snd (quotient-of r))

```

```

define d2 where d2 = (LCM r ∈ supp g. snd (quotient-of r))

```

```

have (LCM r ∈ supp ?h. snd (quotient-of r)) dvd (d1 * d2)

```

```

proof (intro Lcm-least, safe)

```

```

  fix r :: rat

```

```

  assume r ∈ supp ?h

```

```

  hence (∑ (y, z) | y ∈ supp f ∧ z ∈ supp g ∧ r = y + z. f y * g z) ≠ 0

```

```

    by (auto simp: supp-def)

```

hence $\{(y, z). y \in \text{supp } f \wedge z \in \text{supp } g \wedge r = y + z\} \neq \{\}$
by *(intro notI) simp-all*
then obtain $y z$ **where** $yz: y \in \text{supp } f \wedge z \in \text{supp } g \wedge r = y + z$
by *auto*
have $\text{snd}(\text{quotient-of } r) = \text{snd}(\text{quotient-of } y) * \text{snd}(\text{quotient-of } z) \text{ div}$
 $\text{gcd}(\text{fst}(\text{quotient-of } y) * \text{snd}(\text{quotient-of } z) +$
 $\text{fst}(\text{quotient-of } z) * \text{snd}(\text{quotient-of } y))$
 $(\text{snd}(\text{quotient-of } y) * \text{snd}(\text{quotient-of } z))$
by *(simp add: ⟨r = -⟩ rat-plus-code case-prod-unfold Let-def*
Rat.normalize-def quotient-of-denom-pos′)
also have $\dots \text{ dvd } \text{snd}(\text{quotient-of } y) * \text{snd}(\text{quotient-of } z)$
by *(metis dvd-def dvd-div-mult-self gcd-dvd2)*
also have $\dots \text{ dvd } d1 * d2$
using yz **by** *(auto simp: d1-def d2-def intro!: mult-dvd-mono)*
finally show $\text{snd}(\text{quotient-of } r) \text{ dvd } d1 * d2$
by *(simp add: d1-def d2-def)*
qed
moreover have $d1 * d2 \neq 0$
using fg **by** *(auto simp: d1-def d2-def is-fpxs-def)*
ultimately show $(\text{LCM } r \in \text{supp } ?h. \text{snd}(\text{quotient-of } r)) \neq 0$
by *auto*
qed
qed

lemma *fpxs-nth-mult*:
 $\text{fpxs-nth}(f * g) r =$
 $(\sum (y,z) \mid y \in \text{fpxs-supp } f \wedge z \in \text{fpxs-supp } g \wedge r = y + z. \text{fpxs-nth } f y * \text{fpxs-nth } g z)$
by *transfer simp*

lemma *fpxs-compose-power-mult [simp]*:
 $\text{fpxs-compose-power}(f * g) r = \text{fpxs-compose-power } f r * \text{fpxs-compose-power } g r$
proof *(transfer, rule ext, goal-cases)*
case $(1 f g r x)$
show *?case*
proof *(cases r > 0)*
case *True*
have $(\sum x \in \{(y, z). y \in \text{supp } f \wedge z \in \text{supp } g \wedge x / r = y + z\}.$
 $\text{case } x \text{ of } (y, z) \Rightarrow f y * g z) =$
 $(\sum x \in \{(y, z). y \in \text{supp } (\lambda x. f(x / r)) \wedge z \in \text{supp } (\lambda x. g(x / r)) \wedge x =$
 $y + z\}.$
 $\text{case } x \text{ of } (y, z) \Rightarrow f(y / r) * g(z / r))$
by *(rule sum.reindex-bij-witness[of - λ(x,y). (x/r,y/r) λ(x,y). (x*r,y*r)])*
(use ⟨r > 0⟩ in ⟨auto simp: supp-def field-simps⟩)
thus *?thesis*
by *(auto simp: fun-eq-iff)*
qed *auto*
qed

lemma *fpxs-supp-of-fls*: $fpxs\text{-supp} (fpxs\text{-of-fls } f) = of\text{-int } \text{'supp } (fls\text{-nth } f)$
by (*force simp: fpxs-supp-def fpxs-nth-of-fls supp-def elim!: Ints-cases*)

lemma *fpxs-of-fls-mult* [*simp*]: $fpxs\text{-of-fls } (f * g) = fpxs\text{-of-fls } f * fpxs\text{-of-fls } g$
proof (*rule fpxs-ext*)
fix $r :: rat$
show $fpxs\text{-nth} (fpxs\text{-of-fls } (f * g)) r = fpxs\text{-nth} (fpxs\text{-of-fls } f * fpxs\text{-of-fls } g) r$
proof (*cases* $r \in \mathbb{Z}$)
case *True*
define $h1$ **where** $h1 = (\lambda(x, y). (\lfloor x::rat \rfloor, \lfloor y::rat \rfloor))$
define $h2$ **where** $h2 = (\lambda(x, y). (of\text{-int } x :: rat, of\text{-int } y :: rat))$
define df dg **where** [*simp*]: $df = fls\text{-subdegree } f$ $dg = fls\text{-subdegree } g$
from *True* **obtain** n **where** [*simp*]: $r = of\text{-int } n$
by (*cases rule: Ints-cases*)
have $fpxs\text{-nth} (fpxs\text{-of-fls } f * fpxs\text{-of-fls } g) r =$
 $(\sum (y, z) \mid y \in fpxs\text{-supp} (fpxs\text{-of-fls } f) \wedge z \in fpxs\text{-supp} (fpxs\text{-of-fls } g) \wedge$
rat-of-int $n = y + z.$
 $(if\ y \in \mathbb{Z}\ then\ fls\text{-nth } f \lfloor y \rfloor\ else\ 0) * (if\ z \in \mathbb{Z}\ then\ fls\text{-nth } g \lfloor z \rfloor\ else\ 0))$
by (*auto simp: fpxs-nth-mult fpxs-nth-of-fls*)
also have $\dots = (\sum (y, z) \mid y \in \text{supp} (fls\text{-nth } f) \wedge z \in \text{supp} (fls\text{-nth } g) \wedge n =$
 $y + z.$
 $fls\text{-nth } f\ y * fls\text{-nth } g\ z)$
by (*rule sum.reindex-bij-witness[of - h2 h1]*) (*auto simp: h1-def h2-def fpxs-supp-of-fls*)
also have $\dots = (\sum y \mid y - fls\text{-subdegree } g \in \text{supp} (fls\text{-nth } f) \wedge fls\text{-subdegree } g$
 $+ n - y \in \text{supp} (fls\text{-nth } g).$
 $fls\text{-nth } f (y - fls\text{-subdegree } g) * fls\text{-nth } g (fls\text{-subdegree } g + n -$
 $y))$
by (*rule sum.reindex-bij-witness[of - $\lambda y. (y - fls\text{-subdegree } g, fls\text{-subdegree } g + n - y)$ $\lambda z. fst\ z + fls\text{-subdegree } g]$)*)
auto
also have $\dots = (\sum i = fls\text{-subdegree } f + fls\text{-subdegree } g..n.$
 $fls\text{-nth } f (i - fls\text{-subdegree } g) * fls\text{-nth } g (fls\text{-subdegree } g + n - i))$
using *fls-subdegree-leI*[*of* f] *fls-subdegree-leI* [*of* g]
by (*intro sum.mono-neutral-left; force simp: supp-def*)
also have $\dots = fpxs\text{-nth} (fpxs\text{-of-fls } (f * g)) r$
by (*auto simp: fls-times-nth fpxs-nth-of-fls*)
finally show *?thesis ..*

next
case *False*
have $fpxs\text{-nth} (fpxs\text{-of-fls } f * fpxs\text{-of-fls } g) r =$
 $(\sum (y, z) \mid y \in fpxs\text{-supp} (fpxs\text{-of-fls } f) \wedge z \in fpxs\text{-supp} (fpxs\text{-of-fls } g) \wedge$
 $r = y + z.$
 $(if\ y \in \mathbb{Z}\ then\ fls\text{-nth } f \lfloor y \rfloor\ else\ 0) * (if\ z \in \mathbb{Z}\ then\ fls\text{-nth } g \lfloor z \rfloor\ else\ 0))$
by (*simp add: fpxs-nth-mult fpxs-nth-of-fls*)
also have $\dots = 0$
using *False* **by** (*intro sum.neutral ballI*) *auto*
also have $0 = fpxs\text{-nth} (fpxs\text{-of-fls } (f * g)) r$
using *False* **by** (*simp add: fpxs-nth-of-fls*)
finally show *?thesis ..*

```

qed
qed

instance proof
  show 0 ≠ (1 :: 'a fpxs)
    by transfer (auto simp: fun-eq-iff)
next
  fix f :: 'a fpxs
  show 1 * f = f
  proof (transfer, goal-cases)
    case (1 f)
    have {(y, z). y ∈ supp (λr. if r = 0 then (1::'a) else 0) ∧ z ∈ supp f ∧ x = y
+ z} =
      (if x ∈ supp f then {(0, x)} else {}) for x
    by (auto simp: supp-def split: if-splits)
    thus ?case
      by (auto simp: fun-eq-iff supp-def)
  qed
next
  fix f :: 'a fpxs
  show 0 * f = 0
    by transfer (auto simp: fun-eq-iff supp-def)
  show f * 0 = 0
    by transfer (auto simp: fun-eq-iff supp-def)
next
  fix f g :: 'a fpxs
  show f * g = g * f
  proof (transfer, rule ext, goal-cases)
    case (1 f g x)
    show (∑ (y, z) ∈ {(y, z). y ∈ supp f ∧ z ∈ supp g ∧ x = y + z}. f y * g z) =
      (∑ (y, z) ∈ {(y, z). y ∈ supp g ∧ z ∈ supp f ∧ x = y + z}. g y * f z)
    by (rule sum.reindex-bij-witness[of - λ(x,y). (y,x) λ(x,y). (y,x)])
      (auto simp: mult-ac)
  qed
next
  fix f g h :: 'a fpxs
  define d where d = (LCM F ∈ {f,g,h}. fpxs-root-order F)
  have d > 0
    by (auto simp: d-def intro!: Nat.gr0I)
  obtain f' where f: f = fpxs-compose-power (fpxs-of-fls f') (1 / of-nat d)
    using fpxs-as-fls'[of f d] ⟨d > 0⟩ by (auto simp: d-def)
  obtain g' where g: g = fpxs-compose-power (fpxs-of-fls g') (1 / of-nat d)
    using fpxs-as-fls'[of g d] ⟨d > 0⟩ by (auto simp: d-def)
  obtain h' where h: h = fpxs-compose-power (fpxs-of-fls h') (1 / of-nat d)
    using fpxs-as-fls'[of h d] ⟨d > 0⟩ by (auto simp: d-def)
  show (f * g) * h = f * (g * h)
    by (simp add: f g h mult-ac
      flip: fpxs-compose-power-mult fpxs-compose-power-add fpxs-of-fls-mult)
  show (f + g) * h = f * h + g * h

```

```

    by (simp add: f g h ring-distrib
        flip: fpxs-compose-power-mult fpxs-compose-power-add fpxs-of-fls-mult
        fpxs-of-fls-add)
  qed

end

instance fpxs :: (comm-ring-1) comm-ring-1
  by intro-classes auto

instance fpxs :: ({comm-semiring-1, semiring-no-zero-divisors}) semiring-no-zero-divisors
proof
  fix f g :: 'a fpxs
  assume fg: f ≠ 0 g ≠ 0
  define d where d = lcm (fpxs-root-order f) (fpxs-root-order g)
  have d > 0
    by (auto simp: d-def intro!: lcm-pos-nat)
  obtain f' where f: f = fpxs-compose-power (fpxs-of-fls f') (1 / of-nat d)
    using fpxs-as-fls'[of f d] ⟨d > 0⟩ by (auto simp: d-def)
  obtain g' where g: g = fpxs-compose-power (fpxs-of-fls g') (1 / of-nat d)
    using fpxs-as-fls'[of g d] ⟨d > 0⟩ by (auto simp: d-def)
  show f * g ≠ 0
    using ⟨d > 0⟩ fg
    by (simp add: f g flip: fpxs-compose-power-mult fpxs-of-fls-mult)
  qed

lemma fpxs-of-fls-power [simp]: fpxs-of-fls (f ^ n) = fpxs-of-fls f ^ n
  by (induction n) auto

lemma fpxs-compose-power-power [simp]:
  r > 0 ⟹ fpxs-compose-power (f ^ n) r = fpxs-compose-power f r ^ n
  by (induction n) simp-all



### 3.6 Constant Puiseux series and the series $X$



lift-definition fpxs-const :: 'a :: zero ⟹ 'a fpxs is
  λc n. if n = 0 then c else 0
proof -
  fix c :: 'a
  have supp (λn::rat. if n = 0 then c else 0) = (if c = 0 then {} else {0})
    by auto
  thus is-fpxs (λn::rat. if n = 0 then c else 0)
    unfolding is-fpxs-def by auto
  qed

lemma fpxs-const-0 [simp]: fpxs-const 0 = 0
  by transfer auto

lemma fpxs-const-1 [simp]: fpxs-const 1 = 1

```

by transfer auto

lemma *fpxs-of-fls-const* [simp]: $fpxs\text{-of-fls } (fls\text{-const } c) = fpxs\text{-const } c$
by transfer (auto simp: fun-eq-iff Ints-def)

lemma *fls-of-fpxs-const* [simp]: $fls\text{-of-fpxs } (fpxs\text{-const } c) = fls\text{-const } c$
by (metis fls-of-fpxs-of-fls fpxs-of-fls-const)

lemma *fls-of-fpxs-1* [simp]: $fls\text{-of-fpxs } 1 = 1$
using *fls-of-fpxs-const*[of 1] by (simp del: fls-of-fpxs-const)

lift-definition *fpxs-X* :: 'a :: {one, zero} fpxs is
 $\lambda x. \text{if } x = 1 \text{ then } (1 :: 'a) \text{ else } 0$
by (cases $1 = (0 :: 'a)$) (auto simp: is-fpxs-def cong: if-cong)

lemma *fpxs-const-altdef*: $fpxs\text{-const } x = fpxs\text{-of-fls } (fls\text{-const } x)$
by transfer auto

lemma *fpxs-const-add* [simp]: $fpxs\text{-const } (x + y) = fpxs\text{-const } x + fpxs\text{-const } y$
by transfer auto

lemma *fpxs-const-mult* [simp]:
fixes $x y :: 'a :: \{comm\text{-semiring-1}\}$
shows $fpxs\text{-const } (x * y) = fpxs\text{-const } x * fpxs\text{-const } y$
unfolding *fpxs-const-altdef* *fls-const-mult-const*[symmetric] *fpxs-of-fls-mult* ..

lemma *fpxs-const-eq-iff* [simp]:
 $fpxs\text{-const } x = fpxs\text{-const } y \longleftrightarrow x = y$
by transfer (auto simp: fun-eq-iff)

lemma *of-nat-fpxs-eq*: $of\text{-nat } n = fpxs\text{-const } (of\text{-nat } n)$
by (induction n) auto

lemma *fpxs-const-uminus* [simp]: $fpxs\text{-const } (-x) = -fpxs\text{-const } x$
by transfer auto

lemma *fpxs-const-diff* [simp]: $fpxs\text{-const } (x - y) = fpxs\text{-const } x - fpxs\text{-const } y$
unfolding *minus-fpxs-def* by transfer auto

lemma *of-int-fpxs-eq*: $of\text{-int } n = fpxs\text{-const } (of\text{-int } n)$
by (induction n) (auto simp: of-nat-fpxs-eq)

3.7 More algebraic typeclass instances

instance *fpxs* :: ($\{comm\text{-semiring-1}, semiring\text{-char-0}\}$) *semiring-char-0*

proof

show *inj* ($of\text{-nat} :: nat \Rightarrow 'a \text{ fpxs}$)
by (intro *injI*) (auto simp: of-nat-fpxs-eq)

qed

```

instance fpxs :: ({comm-ring-1,ring-char-0}) ring-char-0 ..

instance fpxs :: (idom) idom ..

instantiation fpxs :: (field) field
begin

definition inverse-fpxs :: 'a fpxs ⇒ 'a fpxs where
  inverse-fpxs f =
    fpxs-compose-power (fpxs-of-fls (inverse (fls-of-fpxs f))) (1 / of-nat (fpxs-root-order
f))

definition divide-fpxs :: 'a fpxs ⇒ 'a fpxs ⇒ 'a fpxs where
  divide-fpxs f g = f * inverse g

instance proof
fix f :: 'a fpxs
assume f ≠ 0
define f' where f' = fls-of-fpxs f
define d where d = fpxs-root-order f
have d > 0 by (auto simp: d-def)
have f: f = fpxs-compose-power (fpxs-of-fls f') (1 / of-nat d)
  by (simp add: f'-def d-def fpxs-as-fls)

have inverse f * f = fpxs-compose-power (fpxs-of-fls (inverse f')) (1 / of-nat d)
* f
  by (simp add: inverse-fpxs-def f'-def d-def)
also have fpxs-compose-power (fpxs-of-fls (inverse f')) (1 / of-nat d) * f =
  fpxs-compose-power (fpxs-of-fls (inverse f' * f')) (1 / of-nat d)
  by (simp add: f)
also have inverse f' * f' = 1
  using ⟨f ≠ 0⟩ ⟨d > 0⟩ by (simp add: f field-simps)
finally show inverse f * f = 1
  using ⟨d > 0⟩ by simp
qed (auto simp: divide-fpxs-def inverse-fpxs-def)

end

instance fpxs :: (field-char-0) field-char-0 ..

```

3.8 Valuation

```

definition fpxs-val :: 'a :: zero fpxs ⇒ rat where
  fpxs-val f =
    of-int (fls-subdegree (fls-of-fpxs f)) / rat-of-nat (fpxs-root-order f)

lemma fpxs-val-of-fls [simp]: fpxs-val (fpxs-of-fls f) = of-int (fls-subdegree f)
  by (simp add: fpxs-val-def)

```

lemma *fpxs-nth-compose-power* [*simp*]:
assumes $r > 0$
shows $fpxs\text{-}nth\ (fpxs\text{-}compose\text{-}power\ f\ r)\ n = fpxs\text{-}nth\ f\ (n / r)$
using *assms* **by** *transfer auto*

lemma *fls-of-fpxs-uminus* [*simp*]: $fls\text{-}of\text{-}fpxs\ (-f) = -fls\text{-}of\text{-}fpxs\ f$
by *transfer auto*

lemma *fpxs-root-order-uminus* [*simp*]: $fpxs\text{-}root\text{-}order\ (-f) = fpxs\text{-}root\text{-}order\ f$
by *transfer auto*

lemma *fpxs-val-uminus* [*simp*]: $fpxs\text{-}val\ (-f) = fpxs\text{-}val\ f$
unfolding *fpxs-val-def* **by** *simp*

lemma *fpxs-val-minus-commute*: $fpxs\text{-}val\ (f - g) = fpxs\text{-}val\ (g - f)$
by (*subst fpxs-val-uminus* [*symmetric*]) (*simp del: fpxs-val-uminus*)

lemma *fpxs-val-const* [*simp*]: $fpxs\text{-}val\ (fpxs\text{-}const\ c) = 0$
by (*simp add: fpxs-val-def*)

lemma *fpxs-val-1* [*simp*]: $fpxs\text{-}val\ 1 = 0$
by (*simp add: fpxs-val-def*)

lemma *of-int-fls-subdegree-of-fpxs*:
rat-of-int (*fls-subdegree* (*fls-of-fpxs* f)) = $fpxs\text{-}val\ f * of\text{-}nat\ (fpxs\text{-}root\text{-}order\ f)$
by (*simp add: fpxs-val-def*)

lemma *fpxs-nth-val-nonzero*:
assumes $f \neq 0$
shows $fpxs\text{-}nth\ f\ (fpxs\text{-}val\ f) \neq 0$
proof –
define N **where** $N = fpxs\text{-}root\text{-}order\ f$
define f' **where** $f' = fls\text{-}of\text{-}fpxs\ f$
define M **where** $M = fls\text{-}subdegree\ f'$
have $val: fpxs\text{-}val\ f = of\text{-}int\ M / of\text{-}nat\ N$
by (*simp add: M-def fpxs-val-def N-def f'-def*)
have $*$: $f = fpxs\text{-}compose\text{-}power\ (fpxs\text{-}of\text{-}fls\ f')\ (1 / rat\text{-}of\text{-}nat\ N)$
by (*simp add: fpxs-as-fls N-def f'-def*)
also have $fpxs\text{-}nth\ \dots\ (fpxs\text{-}val\ f) =$
 $fpxs\text{-}nth\ (fpxs\text{-}of\text{-}fls\ f')\ (fpxs\text{-}val\ f * rat\text{-}of\text{-}nat\ (fpxs\text{-}root\text{-}order\ f))$
by (*subst fpxs-nth-compose-power*) (*auto simp: N-def*)
also have $\dots = fls\text{-}nth\ f'\ M$
by (*subst fpxs-nth-of-fls*) (*auto simp: val N-def*)
also have $f' \neq 0$
using $*$ *assms* **by** *auto*
hence $fls\text{-}nth\ f'\ M \neq 0$
unfolding *M-def* **by** *simp*
finally show $fpxs\text{-}nth\ f\ (fpxs\text{-}val\ f) \neq 0$.

qed

lemma *fpxs-nth-below-val*:

assumes $n: n < \text{fpxs-val } f$

shows $\text{fpxs-nth } f \ n = 0$

proof (*cases* $f = 0$)

case *False*

define N **where** $N = \text{fpxs-root-order } f$

define f' **where** $f' = \text{fls-of-fpxs } f$

define M **where** $M = \text{fls-subdegree } f'$

have $\text{val}: \text{fpxs-val } f = \text{of-int } M / \text{of-nat } N$

by (*simp add: M-def fpxs-val-def N-def f'-def*)

have $*$: $f = \text{fpxs-compose-power } (\text{fpxs-of-fls } f') \ (1 / \text{rat-of-nat } N)$

by (*simp add: fpxs-as-fls N-def f'-def*)

have $\text{fpxs-nth } f \ n = \text{fpxs-nth } (\text{fpxs-of-fls } f') \ (n * \text{rat-of-nat } N)$

by (*subst *, subst fpxs-nth-compose-power*) (*auto simp: N-def*)

also have $\dots = 0$

proof (*cases* $\text{rat-of-nat } N * n \in \mathbb{Z}$)

case *True*

then obtain n' **where** $n': \text{of-int } n' = \text{rat-of-nat } N * n$

by (*elim Ints-cases*) *auto*

have $\text{of-int } n' < \text{rat-of-nat } N * \text{fpxs-val } f$

unfolding n' **using** n **by** (*intro mult-strict-left-mono*) (*auto simp: N-def*)

also have $\dots = \text{of-int } M$

by (*simp add: val N-def*)

finally have $n' < M$ **by** *linarith*

have $\text{fpxs-nth } (\text{fpxs-of-fls } f') \ (\text{rat-of-nat } N * n) = \text{fls-nth } f' \ n'$

unfolding n' [*symmetric*] **by** (*subst fpxs-nth-of-fls*) (*auto simp: N-def*)

also from $\langle n' < M \rangle$ **have** $\dots = 0$

unfolding M -*def* **by** *simp*

finally show *?thesis* **by** (*simp add: mult-ac*)

qed (*auto simp: fpxs-nth-of-fls mult-ac*)

finally show $\text{fpxs-nth } f \ n = 0$.

qed *auto*

lemma *fpxs-val-leI*: $\text{fpxs-nth } f \ r \neq 0 \implies \text{fpxs-val } f \leq r$

using *fpxs-nth-below-val*[*of* $r \ f$]

by (*cases* $f = 0$; *cases* $\text{fpxs-val } f \ r$ *rule: linorder-cases*) *auto*

lemma *fpxs-val-0* [*simp*]: $\text{fpxs-val } 0 = 0$

by (*simp add: fpxs-val-def*)

lemma *fpxs-val-geI*:

assumes $f \neq 0 \wedge r. r < r' \implies \text{fpxs-nth } f \ r = 0$

shows $\text{fpxs-val } f \geq r'$

using *fpxs-nth-val-nonzero*[*of* f] *assms* **by** *force*

lemma *fpxs-val-compose-power* [*simp*]:

assumes $r > 0$
shows $\text{fpxs-val} (\text{fpxs-compose-power } f \ r) = \text{fpxs-val } f * r$
proof (*cases* $f = 0$)
case [*simp*]: *False*
show *?thesis*
proof (*intro antisym*)
show $\text{fpxs-val} (\text{fpxs-compose-power } f \ r) \leq \text{fpxs-val } f * r$
using *assms* **by** (*intro fpxs-val-leI*) (*simp add: fpxs-nth-val-nonzero*)
next
show $\text{fpxs-val } f * r \leq \text{fpxs-val} (\text{fpxs-compose-power } f \ r)$
proof (*intro fpxs-val-geI*)
show $\text{fpxs-nth} (\text{fpxs-compose-power } f \ r) \ r' = 0$ **if** $r' < \text{fpxs-val } f * r$ **for** r'
unfolding *fpxs-nth-compose-power*[*OF assms*]
by (*rule fpxs-nth-below-val*) (*use that assms in <auto simp: field-simps>*)
qed (*use assms in auto*)
qed
qed *auto*

lemma *fpxs-val-add-ge*:
assumes $f + g \neq 0$
shows $\text{fpxs-val} (f + g) \geq \min (\text{fpxs-val } f) (\text{fpxs-val } g)$
proof (*rule ccontr*)
assume $\neg(\text{fpxs-val} (f + g) \geq \min (\text{fpxs-val } f) (\text{fpxs-val } g))$ (**is** $\neg(?n \geq -)$)
hence $?n < \text{fpxs-val } f$ $?n < \text{fpxs-val } g$
by *auto*
hence $\text{fpxs-nth } f \ ?n = 0$ $\text{fpxs-nth } g \ ?n = 0$
by (*intro fpxs-nth-below-val; simp; fail*)
hence $\text{fpxs-nth} (f + g) \ ?n = 0$
by *simp*
moreover **have** $\text{fpxs-nth} (f + g) \ ?n \neq 0$
by (*intro fpxs-nth-val-nonzero assms*)
ultimately show *False* **by** *contradiction*
qed

lemma *fpxs-val-diff-ge*:
assumes $f \neq g$
shows $\text{fpxs-val} (f - g) \geq \min (\text{fpxs-val } f) (\text{fpxs-val } g)$
using *fpxs-val-add-ge*[*of f -g*] *assms* **by** *simp*

lemma *fpxs-nth-mult-val*:
 $\text{fpxs-nth} (f * g) (\text{fpxs-val } f + \text{fpxs-val } g) = \text{fpxs-nth } f (\text{fpxs-val } f) * \text{fpxs-nth } g$
 $(\text{fpxs-val } g)$
proof (*cases* $f = 0 \vee g = 0$)
case *False*
have $\{(y, z). y \in \text{fpxs-supp } f \wedge z \in \text{fpxs-supp } g \wedge \text{fpxs-val } f + \text{fpxs-val } g = y + z\} \subseteq$
 $\{(\text{fpxs-val } f, \text{fpxs-val } g)\}$
using *False fpxs-val-leI*[*of f*] *fpxs-val-leI*[*of g*] **by** (*force simp: fpxs-supp-def supp-def*)

hence $\text{fpxs-nth } (f * g) (\text{fpxs-val } f + \text{fpxs-val } g) =$
 $(\sum (y, z) \in \{(\text{fpxs-val } f, \text{fpxs-val } g)\}. \text{fpxs-nth } f y * \text{fpxs-nth } g z)$
unfolding fpxs-nth-mult
 by $(\text{intro sum.mono-neutral-left}) (\text{auto simp: fpxs-supp-def supp-def})$
thus $?thesis$ **by** simp
qed auto

lemma $\text{fpxs-val-mult [simp]}$:
fixes $f g :: 'a :: \{\text{comm-semiring-1}, \text{semiring-no-zero-divisors}\} \text{fpxs}$
assumes $f \neq 0 \ g \neq 0$
shows $\text{fpxs-val } (f * g) = \text{fpxs-val } f + \text{fpxs-val } g$
proof $(\text{intro antisym fpxs-val-leI fpxs-val-geI})$
fix $r :: \text{rat}$
assume $r: r < \text{fpxs-val } f + \text{fpxs-val } g$
show $\text{fpxs-nth } (f * g) r = 0$
unfolding fpxs-nth-mult **using** $\text{assms fpxs-val-leI[of f] fpxs-val-leI[of g] r}$
by $(\text{intro sum.neutral; force})$
qed $(\text{use assms in } \langle \text{auto simp: fpxs-nth-mult-val fpxs-nth-val-nonzero} \rangle)$

lemma $\text{fpxs-val-power [simp]}$:
fixes $f :: 'a :: \{\text{comm-semiring-1}, \text{semiring-no-zero-divisors}\} \text{fpxs}$
assumes $f \neq 0 \ \vee \ n > 0$
shows $\text{fpxs-val } (f \wedge n) = \text{of-nat } n * \text{fpxs-val } f$
proof $(\text{cases } f = 0)$
case False
have $[\text{simp}]: f \wedge n \neq 0$ **for** n
using False **by** $(\text{induction } n) \text{ auto}$
thus $?thesis$ **using** False
by $(\text{induction } n) (\text{auto simp: algebra-simps})$
qed $(\text{use assms in } \langle \text{auto simp: power-0-left} \rangle)$

lemma $\text{fpxs-nth-power-val [simp]}$:
fixes $f :: 'a :: \{\text{comm-semiring-1}, \text{semiring-no-zero-divisors}\} \text{fpxs}$
shows $\text{fpxs-nth } (f \wedge r) (\text{rat-of-nat } r * \text{fpxs-val } f) = \text{fpxs-nth } f (\text{fpxs-val } f) \wedge r$
proof $(\text{cases } f \neq 0)$
case True
show $?thesis$
proof $(\text{induction } r)$
case $(\text{Suc } r)$
have $\text{fpxs-nth } (f \wedge \text{Suc } r) (\text{rat-of-nat } (\text{Suc } r) * \text{fpxs-val } f) =$
 $\text{fpxs-nth } (f * f \wedge r) (\text{fpxs-val } f + \text{fpxs-val } (f \wedge r))$
using True **by** $(\text{simp add: fpxs-nth-mult-val ring-distrib})$
also have $\dots = \text{fpxs-nth } f (\text{fpxs-val } f) \wedge \text{Suc } r$
using Suc True **by** $(\text{subst fpxs-nth-mult-val}) \text{ auto}$
finally show $?case$.
qed $(\text{auto simp: fpxs-nth-1'})$
next
case False
thus $?thesis$

by (cases r) (auto simp: fpxs-nth-1')
qed

3.9 Powers of X and shifting

lift-definition $fpxs\text{-}X\text{-power} :: \text{rat} \Rightarrow 'a :: \{\text{zero}, \text{one}\} \text{ fpxs is}$

$\lambda r n :: \text{rat. if } n = r \text{ then } 1 \text{ else } (0 :: 'a)$

proof –

fix r :: rat

have supp $(\lambda n. \text{if } n = r \text{ then } 1 \text{ else } (0 :: 'a)) = (\text{if } (1 :: 'a) = 0 \text{ then } \{\} \text{ else } \{r\})$

by (auto simp: supp-def)

thus is-fpxs $(\lambda n. \text{if } n = r \text{ then } 1 \text{ else } (0 :: 'a))$

using quotient-of-denom-pos'[of r] by (auto simp: is-fpxs-def)

qed

lemma $fpxs\text{-}X\text{-power-0}$ [simp]: $fpxs\text{-}X\text{-power } 0 = 1$

by transfer auto

lemma $fpxs\text{-}X\text{-power-add}$: $fpxs\text{-}X\text{-power } (a + b) = fpxs\text{-}X\text{-power } a * fpxs\text{-}X\text{-power } b$

proof (transfer, goal-cases)

case (1 a b)

have *: $\{(y, z). y \in \text{supp } (\lambda n. \text{if } n=a \text{ then } (1::'a) \text{ else } 0) \wedge z \in \text{supp } (\lambda n. \text{if } n=b \text{ then } (1::'a) \text{ else } 0) \wedge x=y+z\} =$
 $(\text{if } x = a + b \text{ then } \{(a, b)\} \text{ else } \{\})$ for x

by (auto simp: supp-def fun-eq-iff)

show ?case

unfolding * by (auto simp: fun-eq-iff case-prod-unfold)

qed

lemma $fpxs\text{-}X\text{-power-mult}$: $fpxs\text{-}X\text{-power } (\text{rat-of-nat } n * m) = fpxs\text{-}X\text{-power } m \wedge^n$

by (induction n) (auto simp: ring-distrib fpxs-X-power-add)

lemma $fpxs\text{-of-fls-X-power}$ [simp]: $fpxs\text{-of-fls } (\text{fls-shift } n 1) = fpxs\text{-}X\text{-power } (-\text{rat-of-int } n)$

by transfer (auto simp: fun-eq-iff Ints-def simp flip: of-int-minus)

lemma $fpxs\text{-}X\text{-power-neq-0}$ [simp]: $fpxs\text{-}X\text{-power } r \neq (0 :: 'a :: \text{zero-neq-one fpxs})$

by transfer (auto simp: fun-eq-iff)

lemma $fpxs\text{-}X\text{-power-eq-1-iff}$ [simp]: $fpxs\text{-}X\text{-power } r = (1 :: 'a :: \text{zero-neq-one fpxs}) \iff r = 0$

by transfer (auto simp: fun-eq-iff)

lift-definition $fpxs\text{-shift} :: \text{rat} \Rightarrow 'a :: \text{zero fpxs} \Rightarrow 'a \text{ fpxs is}$

$\lambda r f n. f (n + r)$

```

proof –
  fix  $r :: \text{rat}$  and  $f :: \text{rat} \Rightarrow 'a$ 
  assume  $f: \text{is-fpxs } f$ 
  have  $\text{subset}: \text{supp } (\lambda n. f (n + r)) \subseteq (\lambda n. n + r) - ' \text{supp } f$ 
    by (auto simp: supp-def)
  have  $\text{eq}: (\lambda n. n + r) - ' \text{supp } f = (\lambda n. n - r) - ' \text{supp } f$ 
    by (auto simp: image-iff algebra-simps)

  show  $\text{is-fpxs } (\lambda n. f (n + r))$ 
    unfolding is-fpxs-def
  proof
    have  $\text{bdd-below } ((\lambda n. n + r) - ' \text{supp } f)$ 
      unfolding  $\text{eq}$  by (rule bdd-below-image-mono) (use f in ‹auto simp: is-fpxs-def mono-def›)
    thus  $\text{bdd-below } (\text{supp } (\lambda n. f (n + r)))$ 
      by (rule bdd-below-mono[OF - subset])
    next
      have  $(\text{LCM } r \in \text{supp } (\lambda n. f (n + r)). \text{snd } (\text{quotient-of } r)) \text{ dvd } (\text{LCM } r \in (\lambda n. n + r) - ' \text{supp } f. \text{snd } (\text{quotient-of } r))$ 
        by (intro Lcm-subset image-mono subset)
      also have  $\dots = (\text{LCM } x \in \text{supp } f. \text{snd } (\text{quotient-of } (x - r)))$ 
        by (simp only: eq image-image o-def)
      also have  $\dots \text{ dvd } (\text{LCM } x \in \text{supp } f. \text{snd } (\text{quotient-of } r) * \text{snd } (\text{quotient-of } x))$ 
        by (subst mult.commute, intro Lcm-mono quotient-of-denom-diff-dvd)
      also have  $\dots = \text{Lcm } ((\lambda x. \text{snd } (\text{quotient-of } r) * x) - ' (\lambda x. \text{snd } (\text{quotient-of } x)) - ' \text{supp } f)$ 
        by (simp add: image-image o-def)
      also have  $\dots \text{ dvd } \text{normalize } (\text{snd } (\text{quotient-of } r) * (\text{LCM } x \in \text{supp } f. \text{snd } (\text{quotient-of } x)))$ 
        proof (cases sup f = {})
          case False
            thus ?thesis by (subst Lcm-mult) auto
          qed auto
        finally show  $(\text{LCM } r \in \text{supp } (\lambda n. f (n + r)). \text{snd } (\text{quotient-of } r)) \neq 0$ 
          using quotient-of-denom-pos'[of r] f by (auto simp: is-fpxs-def)
    qed
  qed

lemma fpxs-nth-shift [simp]:  $\text{fpxs-nth } (\text{fpxs-shift } r f) n = \text{fpxs-nth } f (n + r)$ 
  by transfer simp-all

lemma fpxs-shift-0-left [simp]:  $\text{fpxs-shift } 0 f = f$ 
  by transfer auto

lemma fpxs-shift-add-left:  $\text{fpxs-shift } (m + n) f = \text{fpxs-shift } m (\text{fpxs-shift } n f)$ 
  by transfer (simp-all add: add-ac)

lemma fpxs-shift-diff-left:  $\text{fpxs-shift } (m - n) f = \text{fpxs-shift } m (\text{fpxs-shift } (-n) f)$ 
  by (subst fpxs-shift-add-left [symmetric]) auto

```

lemma *fpxs-shift-0* [*simp*]: $\text{fpxs-shift } r \ 0 = 0$
by *transfer simp-all*

lemma *fpxs-shift-add* [*simp*]: $\text{fpxs-shift } r \ (f + g) = \text{fpxs-shift } r \ f + \text{fpxs-shift } r \ g$
by *transfer auto*

lemma *fpxs-shift-uminus* [*simp*]: $\text{fpxs-shift } r \ (-f) = -\text{fpxs-shift } r \ f$
by *transfer auto*

lemma *fpxs-shift-shift-uminus* [*simp*]: $\text{fpxs-shift } r \ (\text{fpxs-shift } (-r) \ f) = f$
by (*simp flip: fpxs-shift-add-left*)

lemma *fpxs-shift-shift-uminus'* [*simp*]: $\text{fpxs-shift } (-r) \ (\text{fpxs-shift } r \ f) = f$
by (*simp flip: fpxs-shift-add-left*)

lemma *fpxs-shift-diff* [*simp*]: $\text{fpxs-shift } r \ (f - g) = \text{fpxs-shift } r \ f - \text{fpxs-shift } r \ g$
unfolding *minus-fpxs-def* **by** (*subst fpxs-shift-add*) *auto*

lemma *fpxs-shift-compose-power* [*simp*]:
 $\text{fpxs-shift } r \ (\text{fpxs-compose-power } f \ s) = \text{fpxs-compose-power } (\text{fpxs-shift } (r / s) \ f)$
 s
by *transfer (simp-all add: add-divide-distrib add-ac cong: if-cong)*

lemma *rat-of-int-div-dvd*: $d \ \text{dvd} \ n \implies \text{rat-of-int } (n \ \text{div} \ d) = \text{rat-of-int } n / \text{rat-of-int } d$
by *auto*

lemma *fpxs-of-fls-shift* [*simp*]:
 $\text{fpxs-of-fls } (\text{fls-shift } n \ f) = \text{fpxs-shift } (\text{of-int } n) \ (\text{fpxs-of-fls } f)$
proof (*transfer, goal-cases*)
case ($1 \ n \ f$)
show *?case*
proof
fix $r :: \text{rat}$
have $\text{eq}: r + \text{rat-of-int } n \in \mathbb{Z} \iff r \in \mathbb{Z}$
by (*metis Ints-add Ints-diff Ints-of-int add-diff-cancel-right'*)
show (*if* $r \in \mathbb{Z}$ *then* $f \ (\lfloor r \rfloor + n)$ *else* 0) =
(*if* $r + \text{rat-of-int } n \in \mathbb{Z}$ *then* $f \ \lfloor r + \text{rat-of-int } n \rfloor$ *else* 0)
unfolding *eq* **by** *auto*
qed
qed

lemma *fpxs-shift-mult*: $f * \text{fpxs-shift } r \ g = \text{fpxs-shift } r \ (f * g)$
 $\text{fpxs-shift } r \ f * g = \text{fpxs-shift } r \ (f * g)$
proof –
obtain $a \ b$ **where** *ab*: $r = \text{of-int } a / \text{of-nat } b$ **and** $b > 0$
by (*metis Fract-of-int-quotient of-int-of-nat-eq quotient-of-unique zero-less-imp-eq-int*)

```

define  $s$  where  $s = \text{lcm } b (\text{lcm } (\text{fps-root-order } f) (\text{fps-root-order } g))$ 
have  $s > 0$  using  $\langle b > 0 \rangle$ 
  by (auto simp: s-def intro!: Nat.gr0I)
obtain  $f'$  where  $f: f = \text{fps-compose-power } (\text{fps-of-fls } f') (1 / \text{rat-of-nat } s)$ 
  using  $\text{fps-as-fls}'[\text{of } f \ s] \ \langle s > 0 \rangle$  by (auto simp: s-def)
obtain  $g'$  where  $g: g = \text{fps-compose-power } (\text{fps-of-fls } g') (1 / \text{rat-of-nat } s)$ 
  using  $\text{fps-as-fls}'[\text{of } g \ s] \ \langle s > 0 \rangle$  by (auto simp: s-def)

```

```

define  $n$  where  $n = (a * s) \text{ div } b$ 
have  $b \text{ dvd } s$ 
  by (auto simp: s-def)
have  $\text{sr-eq}: r * \text{rat-of-nat } s = \text{rat-of-int } n$ 
  using  $\langle b > 0 \rangle \ \langle b \text{ dvd } s \rangle$ 
  by (simp add: ab field-simps of-rat-divide of-rat-mult n-def rat-of-int-div-dvd)

```

```

show  $f * \text{fps-shift } r \ g = \text{fps-shift } r (f * g) \ \text{fps-shift } r \ f * g = \text{fps-shift } r (f * g)$ 
unfolding  $f \ g$  using  $\langle s > 0 \rangle$ 
by (simp-all flip: fps-compose-power-mult fps-of-fls-mult fps-of-fls-shift add: sr-eq fls-shifted-times-simps mult-ac)

```

qed

```

lemma  $\text{fps-shift-1}: \text{fps-shift } r \ 1 = \text{fps-X-power } (-r)$ 
by transfer (auto simp: fun-eq-iff)

```

```

lemma  $\text{fps-X-power-conv-shift}: \text{fps-X-power } r = \text{fps-shift } (-r) \ 1$ 
by (simp add: fps-shift-1)

```

```

lemma  $\text{fps-shift-power}$  [simp]:  $\text{fps-shift } n \ x^{\wedge} m = \text{fps-shift } (\text{of-nat } m * n) (x^{\wedge} m)$ 
by (induction m (simp-all add: algebra-simps fps-shift-mult flip: fps-shift-add-left))

```

```

lemma  $\text{fps-compose-power-X-power}$  [simp]:
 $s > 0 \implies \text{fps-compose-power } (\text{fps-X-power } r) \ s = \text{fps-X-power } (r * s)$ 
by transfer (simp add: field-simps)

```

3.10 The n -th root of a Puiseux series

In this section, we define the formal root of a Puiseux series. This is done using the same concept for formal power series. There is still one interesting theorem that is missing here, e.g. the uniqueness (which could probably be lifted over from FPSs) somehow.

```

definition  $\text{fps-radical} :: (\text{nat} \Rightarrow 'a :: \text{field-char-0} \Rightarrow 'a) \Rightarrow \text{nat} \Rightarrow 'a \ \text{fps} \Rightarrow 'a \ \text{fps}$ 
where
 $\text{fps-radical } r \ f = (\text{if } f = 0 \text{ then } 0 \text{ else}$ 
  ( $\text{let } f' = \text{fls-base-factor-to-fps } (\text{fls-of-fps } f);$ 
    $f'' = \text{fps-of-fls } (\text{fps-to-fls } (\text{fps-radical } r \ f'))$ 
    $\text{in } \text{fps-shift } (-\text{fps-val } f / \text{rat-of-nat } r)$ )

```

(*fpxs-compose-power* f'' (1 / *rat-of-nat* (*fpxs-root-order* f))))

lemma *fpxs-radical-0* [*simp*]: *fpxs-radical* rt r 0 = 0
by (*simp add: fpxs-radical-def*)

lemma

fixes $r :: nat$

assumes $r: r > 0$

shows *fpxs-power-radical*:

rt r (*fpxs-nth* f (*fpxs-val* f)) $\wedge^r =$ *fpxs-nth* f (*fpxs-val* f) \implies *fpxs-radical* rt r $f \wedge^r = f$

and *fpxs-radical-lead-coeff*:

$f \neq 0 \implies$ *fpxs-nth* (*fpxs-radical* rt r f) (*fpxs-val* f / *rat-of-nat* r) =
 rt r (*fpxs-nth* f (*fpxs-val* f))

proof –

define q **where** $q =$ *fpxs-root-order* f

define f' **where** $f' =$ *fls-base-factor-to-fps* (*fls-of-fpxs* f)

have [*simp*]: *fps-nth* f' 0 = *fpxs-nth* f (*fpxs-val* f)

by (*simp add: f'-def fls-nth-of-fpxs of-int-fls-subdegree-of-fpxs*)

define f'' **where** $f'' =$ *fpxs-of-fls* (*fps-to-fls* (*fps-radical* rt r f'))

have *eq1*: *fls-of-fpxs* $f =$ *fls-shift* ($-$ *fls-subdegree* (*fls-of-fpxs* f)) (*fps-to-fls* f')

by (*subst fls-conv-base-factor-to-fps-shift-subdegree*) (*simp add: f'-def*)

have *eq2*: *fpxs-compose-power* (*fpxs-of-fls* (*fls-of-fpxs* f)) (1 / *of-nat* q) = f

unfolding *q-def* **by** (*rule fpxs-as-fls*)

also note *eq1*

also have *fpxs-of-fls* (*fls-shift* ($-$ *fls-subdegree* (*fls-of-fpxs* f)) (*fps-to-fls* f')) =
fpxs-shift ($-$ (*fpxs-val* f * *rat-of-nat* q)) (*fpxs-of-fls* (*fps-to-fls* f'))

by (*simp add: of-int-fls-subdegree-of-fpxs q-def*)

finally have *eq3*: *fpxs-compose-power* (*fpxs-shift* ($-$ (*fpxs-val* f * *rat-of-nat* q))
(*fpxs-of-fls* (*fps-to-fls* f'))) (1 / *rat-of-nat* q) = f .

{

assume rt : rt r (*fpxs-nth* f (*fpxs-val* f)) $\wedge^r =$ *fpxs-nth* f (*fpxs-val* f)

show *fpxs-radical* rt r $f \wedge^r = f$

proof (*cases* $f = 0$)

case [*simp*]: *False*

have $f'' \wedge^r =$ *fpxs-of-fls* (*fps-to-fls* (*fps-radical* rt r $f' \wedge^r$))

by (*simp add: fps-to-fls-power f''-def*)

also have *fps-radical* rt r $f' \wedge^r = f'$

using *power-radical*[*of* f' rt $r - 1$] r rt **by** (*simp add: fpxs-nth-val-nonzero*)

finally have $f'' \wedge^r =$ *fpxs-of-fls* (*fps-to-fls* f') .

have *fpxs-shift* ($-$ *fpxs-val* f / *rat-of-nat* r) (*fpxs-compose-power* f'' (1 / *of-nat* q)) $\wedge^r =$

fpxs-shift ($-$ *fpxs-val* f) (*fpxs-compose-power* ($f'' \wedge^r$) (1 / *of-nat* q))

unfolding *q-def* **using** r

by (*subst fpxs-shift-power, subst fpxs-compose-power-power* [*symmetric*])

simp-all

also have $f'' \wedge^r =$ *fpxs-of-fls* (*fps-to-fls* f')

```

    by fact
  also have fpxs-shift (-fpxs-val f) (fpxs-compose-power
    (fpxs-of-fls (fpxs-to-fls f')) (1 / of-nat q)) = f
    using r eq3 by simp
  finally show fpxs-radical rt r f ^ r = f
    by (simp add: fpxs-radical-def f'-def f''-def q-def)
  qed (use r in auto)
}

assume [simp]: f ≠ 0
have fpxs-nth (fpxs-shift (-fpxs-val f / of-nat r) (fpxs-compose-power f'' (1 /
of-nat q)))
  (fpxs-val f / of-nat r) = fpxs-nth f'' 0
  using r by (simp add: q-def)
also have fpxs-shift (-fpxs-val f / of-nat r) (fpxs-compose-power f'' (1 / of-nat
q)) =
  fpxs-radical rt r f
  by (simp add: fpxs-radical-def q-def f'-def f''-def)
also have fpxs-nth f'' 0 = rt r (fpxs-nth f (fpxs-val f))
  using r by (simp add: f''-def fpxs-nth-of-fls)
finally show fpxs-nth (fpxs-radical rt r f) (fpxs-val f / rat-of-nat r) =
  rt r (fpxs-nth f (fpxs-val f)) .
qed

lemma fls-base-factor-power:
  fixes f :: 'a::{semiring-1, semiring-no-zero-divisors} fls
  shows fls-base-factor (f ^ n) = fls-base-factor f ^ n
proof (cases f = 0)
  case False
  have [simp]: f ^ n ≠ 0 for n
    by (induction n) (use False in auto)
  show ?thesis using False
  by (induction n) (auto simp: fls-base-factor-mult simp flip: fls-times-both-shifted-simp)
qed (cases n; simp)

```

hide-const (open) *supp*

3.11 Algebraic closedness

We will now show that the field of formal Puiseux series over an algebraically closed field of characteristic 0 is again algebraically closed.

The typeclass constraint *field-gcd* is a technical constraint that mandates that the field has a (trivial) GCD operation defined on it. It comes from some peculiarities of Isabelle's typeclass system and can be considered unimportant, since any concrete type of class *field* can easily be made an instance of *field-gcd*.

It would be possible to get rid of this constraint entirely here, but it is not worth the effort.

The proof is a fairly standard one that uses Hensel's lemma. Some preliminary tricks are required to be able to use it, however, namely a number of non-obvious changes of variables to turn the polynomial with Puiseux coefficients into one with formal power series coefficients. The overall approach was taken from an article by Nowak [2].

Basically, what we need to show is this: Let

$$p(X, Z) = a_n(Z)X^n + a_{n-1}(Z)X^{n-1} + \dots + a_0(Z)$$

be a polynomial in X of degree at least 2 with coefficients that are formal Puiseux series in Z . Then p is reducible, i.e. it splits into two non-constant factors.

Due to work we have already done elsewhere, we may assume here that $a_n = 1$, $a_{n-1} = 0$, and $a_0 \neq 0$, all of which will come in very useful.

instance *fpxs* :: (*alg-closed-field*, *field-char-0*, *field-gcd*) *alg-closed-field*

proof (*rule alg-closedI-reducible-coeff-deg-minus-one-eq-0*)

fix *p* :: 'a *fpxs poly*

assume *deg-p*: *degree p > 1* **and** *lc-p*: *lead-coeff p = 1*

assume *coeff-deg-minus-1*: *coeff p (degree p - 1) = 0*

assume *coeff p 0 ≠ 0*

define *N* **where** *N = degree p*

Let a_0, \dots, a_n be the coefficients of p with $a_n = 1$. Now let r be the maximum of $-\frac{\text{val}(a_i)}{n-i}$ ranging over all $i < n$ such that $a_i \neq 0$.

define *r* :: *rat*

where *r* = (*MAX i ∈ {i ∈ {..<N}. coeff p i ≠ 0}*).

-fpxs-val (poly.coeff p i) / (rat-of-nat N - rat-of-nat i))

We write $r = a/b$ such that all the a_i can be written as Laurent series in $X^{1/b}$, i.e. the root orders of all the a_i divide b :

obtain *a b* **where** *ab*: *b > 0 r = of-int a / of-nat b ∀ i ≤ N. fpxs-root-order (coeff p i) dvd b*

proof –

define *b* **where** *b = lcm (nat (snd (quotient-of r))) (LCM i ∈ {..N}. fpxs-root-order (coeff p i))*

define *x* **where** *x = b div nat (snd (quotient-of r))*

define *a* **where** *a = fst (quotient-of r) * int x*

show *?thesis*

proof (*rule that*)

show *b > 0*

using *quotient-of-denom-pos'[of r]* **by** (*auto simp: b-def intro!: Nat.gr0I*)

have *b-eq*: *b = nat (snd (quotient-of r)) * x*

by (*simp add: x-def b-def*)

```

have  $x > 0$ 
  using  $b\text{-eq } \langle b > 0 \rangle$  by (auto intro!: Nat.gr0I)
have  $r = \text{rat-of-int (fst (quotient-of r)) / rat-of-int (int (nat (snd (quotient-of r))))$ 
  using  $\text{quotient-of-denom-pos}[of r] \text{quotient-of-div}[of r]$  by simp
also have  $\dots = \text{rat-of-int } a / \text{rat-of-nat } b$ 
  using  $\langle x > 0 \rangle$  by (simp add: a-def b-eq)
finally show  $r = \text{rat-of-int } a / \text{rat-of-nat } b .$ 
show  $\forall i \leq N. \text{fpxs-root-order (poly.coeff } p \ i) \text{ dvd } b$ 
  by (auto simp: b-def)
qed
qed

```

We write all the coefficients of p as Laurent series in $X^{1/b}$:

```

have  $\exists c. \text{coeff } p \ i = \text{fpxs-compose-power (fpxs-of-fls } c) (1 / \text{rat-of-nat } b)$  if  $i: i \leq N$  for  $i$ 
proof -
  have  $\text{fpxs-root-order (coeff } p \ i) \text{ dvd } b$ 
    using  $ab(3) \ i$  by auto
  from  $\text{fpxs-as-fls}[OF \text{ this } \langle b > 0 \rangle]$  show ?thesis by metis
qed
then obtain  $c\text{-aux}$  where  $c\text{-aux}$ :
   $\text{coeff } p \ i = \text{fpxs-compose-power (fpxs-of-fls (c-aux } i)) (1 / \text{rat-of-nat } b)$  if  $i \leq N$  for  $i$ 
  by metis
define  $c$  where  $c = (\lambda i. \text{if } i \leq N \text{ then } c\text{-aux } i \text{ else } 0)$ 
have  $c: \text{coeff } p \ i = \text{fpxs-compose-power (fpxs-of-fls (c } i)) (1 / \text{rat-of-nat } b)$  for  $i$ 
  using  $c\text{-aux}[of \ i]$  by (auto simp: c-def N-def coeff-eq-0)
have  $c\text{-eq-0}$  [simp]:  $c \ i = 0$  if  $i > N$  for  $i$ 
  using that by (auto simp: c-def)
have  $c\text{-eq}: \text{fpxs-of-fls (c } i) = \text{fpxs-compose-power (coeff } p \ i) (\text{rat-of-nat } b)$  for  $i$ 
  using  $c[of \ i] \ \langle b > 0 \rangle$  by (simp add: fpxs-compose-power-distrib)

```

We perform another change of variables and multiply with a suitable power of X to turn our Laurent coefficients into FPS coefficients:

```

define  $c'$  where  $c' = (\lambda i. \text{fls-}X\text{-intpow ((int } N - \text{int } i) * a) * c \ i)$ 
have  $c' \ N = 1$ 
  using  $c[of \ N] \ \langle \text{lead-coeff } p = 1 \rangle \ \langle b > 0 \rangle$  by (simp add:  $c'\text{-def } N\text{-def}$ )

have  $\text{subdegree-}c: \text{of-int (fls-subdegree (c } i)) = \text{fpxs-val (coeff } p \ i) * \text{rat-of-nat } b$ 
if  $i: i \leq N$  for  $i$ 
proof -
  have  $\text{rat-of-int (fls-subdegree (c } i)) = \text{fpxs-val (fpxs-of-fls (c } i))$ 
    by simp
  also have  $\text{fpxs-of-fls (c } i) = \text{fpxs-compose-power (poly.coeff } p \ i) (\text{rat-of-nat } b)$ 
    by (subst  $c\text{-eq}$ ) auto
  also have  $\text{fpxs-val } \dots = \text{fpxs-val (coeff } p \ i) * \text{rat-of-nat } b$ 
    using  $\langle b > 0 \rangle$  by simp
  finally show ?thesis .

```

qed

We now write all the coefficients as FPSs:

```

have  $\exists c''$ .  $c' i = \text{fps-to-fls } c''$  if  $i \leq N$  for  $i$ 
proof (cases  $i = N$ )
  case True
    hence  $c' i = \text{fps-to-fls } 1$ 
    using  $\langle c' N = 1 \rangle$  by simp
    thus ?thesis by metis
  next
    case  $i$ : False
    show ?thesis
    proof (cases  $c i = 0$ )
      case True
        hence  $c' i = 0$  by (auto simp: c'-def)
        thus ?thesis
        by (metis fps-zero-to-fls)
      next
        case False
        hence  $\text{coeff } p i \neq 0$ 
        using c-eq[of i] by auto
        hence  $r\text{-ge}$ :  $r \geq -\text{fps-val } (\text{poly.coeff } p i) / (\text{rat-of-nat } N - \text{rat-of-nat } i)$ 
        unfolding r-def using  $i$  that False by (intro Max.coboundedI) auto

    have  $\text{fls-subdegree } (c' i) = \text{fls-subdegree } (c i) + (\text{int } N - \text{int } i) * a$ 
    using  $i$  that False by (simp add: c'-def fls-X-intpow-times-conv-shift subdegree-c)
    also have  $\text{rat-of-int } \dots =$ 
       $\text{fps-val } (\text{poly.coeff } p i) * \text{of-nat } b + (\text{of-nat } N - \text{of-nat } i) * \text{of-int } a$ 
    using  $i$  that False by (simp add: subdegree-c)
    also have  $\dots = \text{of-nat } b * (\text{of-nat } N - \text{of-nat } i) *$ 
       $(\text{fps-val } (\text{poly.coeff } p i) / (\text{of-nat } N - \text{of-nat } i) + r)$ 
    using  $\langle b > 0 \rangle$   $i$  by (auto simp: field-simps ab(2))
    also have  $\dots \geq 0$ 
    using  $r\text{-ge}$  that by (intro mult-nonneg-nonneg) auto
    finally have  $\text{fls-subdegree } (c' i) \geq 0$  by simp
    hence  $\exists c''$ .  $c' i = \text{fls-shift } 0$  (fps-to-fls c'')
    by (intro fls-as-fps') (auto simp: algebra-simps)
    thus ?thesis by simp
  qed
qed
then obtain  $c''\text{-aux}$  where  $c''\text{-aux}$ :  $c' i = \text{fps-to-fls } (c''\text{-aux } i)$  if  $i \leq N$  for  $i$ 
by metis
define  $c''$  where  $c'' = (\lambda i$ . if  $i \leq N$  then  $c''\text{-aux } i$  else  $0$ )
have  $c'$ :  $c' i = \text{fps-to-fls } (c'' i)$  for  $i$ 
proof (cases  $i \leq N$ )
  case False
    thus ?thesis by (auto simp: c'-def c''-def)
qed (auto simp: c''-def c''-aux)

```

```

have  $c''$ -eq: fps-to-fls ( $c''$   $i$ ) =  $c'$   $i$  for  $i$ 
  using  $c'$ [of  $i$ ] by simp

define  $p'$  where  $p' = \text{Abs-poly } c''$ 
have coeff-p': coeff  $p' = c''$ 
  unfolding  $p'$ -def
proof (rule coeff-Abs-poly)
  fix  $i$  assume  $i > N$ 
  hence coeff  $p$   $i = 0$ 
  by (simp add: N-def coeff-eq-0)
  thus  $c''$   $i = 0$  using  $c'$ [of  $i$ ]  $c$ [of  $i$ ]  $\langle b > 0 \rangle \langle N < i \rangle$   $c''$ -def by auto
qed

```

We set up some homomorphisms to convert between the two polynomials:

```

interpret comppow: map-poly-inj-idom-hom ( $\lambda x::'a \text{ fpxs. fpxs-compose-power } x$ 
  ( $1/\text{rat-of-nat } b$ ))
  by unfold-locales (use  $\langle b > 0 \rangle$  in simp-all)
define lift-poly ::  $'a \text{ fpxs poly} \Rightarrow 'a \text{ fpxs poly}$  where
  lift-poly = ( $\lambda p. p \text{compose } p$  [ $:0$ , fpxs-X-power  $r$ :])  $\circ$ 
    (map-poly (( $\lambda x. \text{fpxs-compose-power } x$  ( $1/\text{rat-of-nat } b$ ))  $\circ$  fpxs-of-fls
   $\circ$  fps-to-fls))
have [simp]: degree (lift-poly  $q$ ) = degree  $q$  for  $q$ 
  unfolding lift-poly-def by (simp add: degree-map-poly)

interpret fps-to-fls: map-poly-inj-idom-hom fps-to-fls
  by unfold-locales (simp-all add: fls-times-fps-to-fls)
interpret fpxs-of-fls: map-poly-inj-idom-hom fpxs-of-fls
  by unfold-locales simp-all
interpret lift-poly: inj-idom-hom lift-poly
  unfolding lift-poly-def
  by (intro inj-idom-hom-compose inj-idom-hom-pcompose inj-idom-hom.inj-idom-hom-map-poly
    fps-to-fls.base.inj-idom-hom-axioms fpxs-of-fls.base.inj-idom-hom-axioms
    comppow.base.inj-idom-hom-axioms) simp-all
interpret lift-poly: map-poly-inj-idom-hom lift-poly
  by unfold-locales

define  $C$  ::  $'a \text{ fpxs}$  where  $C = \text{fpxs-X-power } (- (\text{rat-of-nat } N * r))$ 
have [simp]:  $C \neq 0$ 
  by (auto simp: C-def)

```

Now, finally: the original polynomial and the new polynomial are related through the *lift-poly* homomorphism:

```

have  $p$ -eq:  $p = \text{smult } C$  (lift-poly  $p'$ )
  using  $\langle b > 0 \rangle$ 
  by (intro poly-eqI)
  (simp-all add: coeff-map-poly coeff-pcompose-linear coeff-p' c c''-eq c'-def
  C-def
  ring-distrib fpxs-X-power-conv-shift fpxs-shift-mult lift-poly-def
  ab(2))

```

flip: fpxs-X-power-add fpxs-X-power-mult fpxs-shift-add-left

have [*simp*]: *degree p' = N*
unfolding *N-def* **using** $\langle b > 0 \rangle$ **by** (*simp add: p-eq*)
have *lc-p'*: *lead-coeff p' = 1*
using *c''-eq[of N]* **by** (*simp add: coeff-p' $\langle c' N = 1 \rangle$*)
have *coeff p' (N - 1) = 0*
using *coeff-deg-minus-1 $\langle b > 0 \rangle$* **unfolding** *N-def [symmetric]*
by (*simp add: p-eq lift-poly-def coeff-map-poly coeff-pcompose-linear*)

We reduce $p'(X, Z)$ to $p'(X, 0)$:

define *p'-proj* **where** *p'-proj = reduce-fps-poly p'*
have [*simp*]: *degree p'-proj = N*
unfolding *p'-proj-def* **using** *lc-p'* **by** (*subst degree-reduce-fps-poly-monic simp-all*)
have *lc-p'-proj*: *lead-coeff p'-proj = 1*
unfolding *p'-proj-def* **using** *lc-p'* **by** (*subst reduce-fps-poly-monic simp-all*)
hence [*simp*]: *p'-proj $\neq 0$*
by *auto*
have *coeff p'-proj (N - 1) = 0*
using $\langle \text{coeff } p' (N - 1) = 0 \rangle$ **by** (*simp add: p'-proj-def reduce-fps-poly-def*)

We now show that p' -proj splits into non-trivial coprime factors. To do this, we have to show that it has two distinct roots, i.e. that it is not of the form $(X - c)^n$.

obtain *g h* **where** *gh: degree g > 0 degree h > 0 coprime g h p'-proj = g * h*
proof –
have *degree p'-proj > 1*
using *deg-p* **by** (*auto simp: N-def*)

Let x be an arbitrary root of p' -proj:

then obtain *x* **where** *x: poly p'-proj x = 0*
using *alg-closed-imp-poly-has-root[of p'-proj]* **by** *force*

Assume for the sake of contradiction that p' -proj were equal to $(1 - x)^n$:

have *not-only-one-root: p'-proj \neq $[: -x, 1:] \wedge N$*
proof *safe*
assume $*$: *p'-proj = $[: -x, 1:] \wedge N$*

If x were non-zero, all the coefficients of p' -proj would also be non-zero by the Binomial Theorem. Since we know that the coefficient of $n - 1$ is zero, this means that x must be zero:

have *coeff p'-proj (N - 1) = 0* **by** *fact*
hence *x = 0*
by (*subst (asm) *, subst (asm) coeff-linear-poly-power*) *auto*

However, by our choice of r , we know that there is an index i such that $c' i$ has is non-zero and has valuation (i.e. subdegree) 0, which means that the i -th coefficient of p' -proj must also be non-zero.

```

have 0 < N ∧ coeff p 0 ≠ 0
  using deg-p ⟨coeff p 0 ≠ 0⟩ by (auto simp: N-def)
hence {i∈{..

```

We can thus obtain our second root y from the factorisation:

```

have ∃ y. x ≠ y ∧ poly p'-proj y = 0
proof (rule ccontr)
  assume *: ¬(∃ y. x ≠ y ∧ poly p'-proj y = 0)
  have p'-proj ≠ 0 by simp
  then obtain A where A: size A = degree p'-proj
    p'-proj = smult (lead-coeff p'-proj) (∏ x∈#A. [-x, 1:])
    using alg-closed-imp-factorization[of p'-proj] by blast
  have set-mset A = {x. poly p'-proj x = 0}

```

```

    using lc-p'-proj by (subst A) (auto simp: poly-prod-mset)
  also have ... = {x}
    using x * by auto
  finally have A = replicate-mset N x
    using set-mset-subset-singletonD[of A x] A(1) by simp
  with A(2) have p'-proj = [: - x, 1:] ^ N
    using lc-p'-proj by simp
  with not-only-one-root show False
    by contradiction
qed
then obtain y where x ≠ y poly p'-proj y = 0
  by blast

```

It now follows easily that p' -proj splits into non-trivial and coprime factors:

```

show ?thesis
proof (rule alg-closed-imp-poly-splits-coprime)
  show degree p'-proj > 1
    using deg-p by (simp add: N-def)
  show x ≠ y poly p'-proj x = 0 poly p'-proj y = 0
    by fact+
qed (use that in metis)
qed

```

By Hensel's lemma, these factors give rise to corresponding factors of p' :

```

interpret hensel: fps-hensel p' p'-proj g h
proof unfold-locales
  show lead-coeff p' = 1
    using lc-p' by simp
qed (use gh ⟨coprime g h⟩ in ⟨simp-all add: p'-proj-def⟩)

```

All that remains now is to undo the variable substitutions we did above:

```

have p = [:C:] * lift-poly hensel.G * lift-poly hensel.H
  unfolding p-eq by (subst hensel.F-splits) (simp add: hom-distrib)
thus -irreducible p
  by (rule reducible-polyI) (use hensel.deg-G hensel.deg-H gh in simp-all)
qed

```

We do not actually show that this is the algebraic closure since this cannot be stated idiomatically in the typeclass setting and is probably not very useful either, but it can be motivated like this:

Suppose we have an algebraically closed extension L of the field of Laurent series. Clearly, $X^{a/b} \in L$ for any integer a and any positive integer b since $(X^{a/b})^b - X^a = 0$. But any Puiseux series $F(X)$ with root order b can be written as

$$F(X) = \sum_{k=0}^{b-1} X^{k/b} F_k(X)$$

where the Laurent series $F_k(X)$ are defined as follows:

$$F_k(X) := \sum_{n=n_{0,k}}^{\infty} [X^{n+k/b}]F(X)X^n$$

Thus, $F(X)$ can be written as a finite sum of products of elements in L and must therefore also be in L . Thus, the Puiseux series are all contained in L .

3.12 Metric and topology

Formal Puiseux series form a metric space with the usual metric for formal series: Two series are “close” to one another if they have many initial coefficients in common.

instantiation *fpxs* :: (zero) norm
begin

definition *norm-fpxs* :: 'a fpxs \Rightarrow real **where**
norm $f = (if\ f = 0\ then\ 0\ else\ 2\ powr\ (-of-rat\ (fpxs-val\ f)))$

instance ..

end

instantiation *fpxs* :: (group-add) dist
begin

definition *dist-fpxs* :: 'a fpxs \Rightarrow 'a fpxs \Rightarrow real **where**
dist $f\ g = (if\ f = g\ then\ 0\ else\ 2\ powr\ (-of-rat\ (fpxs-val\ (f - g))))$

instance ..

end

instantiation *fpxs* :: (group-add) metric-space
begin

definition *uniformity-fpxs-def* [code del]:
(uniformity :: ('a fpxs \times 'a fpxs) filter) = (INF $e \in \{0 < ..\}$. principal $\{(x, y). dist\ x\ y < e\}$)

definition *open-fpxs-def* [code del]:
open ($U :: 'a\ fpxs\ set$) $\longleftrightarrow (\forall x \in U. eventually\ (\lambda(x', y). x' = x \longrightarrow y \in U))$
uniformity)

instance proof

```

fix f g h :: 'a fpxs
show dist f g ≤ dist f h + dist g h
proof (cases f ≠ g ∧ f ≠ h ∧ g ≠ h)
  case True
  have dist f g ≤ 2 powr -real-of-rat (min (fpxs-val (f - h)) (fpxs-val (g - h)))
    using fpxs-val-add-ge[of f - h h - g] True
  by (auto simp: algebra-simps fpxs-val-minus-commute dist-fpxs-def of-rat-less-eq)
  also have ... ≤ dist f h + dist g h
    using True by (simp add: dist-fpxs-def min-def)
  finally show ?thesis .
qed (auto simp: dist-fpxs-def fpxs-val-minus-commute)
qed (simp-all add: uniformity-fpxs-def open-fpxs-def dist-fpxs-def)

end

```

```

instance fpxs :: (group-add) dist-norm
  by standard (auto simp: dist-fpxs-def norm-fpxs-def)

```

```

lemma fpxs-const-eq-0-iff [simp]: fpxs-const x = 0 ↔ x = 0
  by (metis fpxs-const-0 fpxs-const-eq-iff)

```

```

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

```

```

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

```

end

References

- [1] S. S. Abhyankar. *Algebraic Geometry for Scientists and Engineers*. Mathematical surveys and monographs. American Mathematical Society, 1990.
- [2] K. J. Nowak. Some elementary proofs of Puiseuxs theorems. *Univ. Iagel. Acta Math*, 38:279–282, 2000.