

Formal Puiseux Series

Manuel Eberl

December 14, 2021

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	6
1.3	Facts about Laurent series	13
2	Hensel’s lemma for formal power series	17
3	Formal Puiseux Series	26
3.1	Auxiliary facts and definitions	26
3.2	Definition	28
3.3	Basic algebraic typeclass instances	30
3.4	The substitution $X \mapsto X^r$	34
3.5	Multiplication and ring properties	39
3.6	Constant Puiseux series and the series X	43
3.7	More algebraic typeclass instances	44
3.8	Valuation	45
3.9	Powers of X and shifting	50
3.10	The n -th root of a Puiseux series	53
3.11	Algebraic closedness	55
3.12	Metric and topology	63

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 *poly-sum-mset*: $\text{poly } (\sum x \in \#A. p x) y = (\sum x \in \#A. \text{poly } (p x) y)$
by (*induction A*) *auto*

lemma *poly-prod-mset*: $\text{poly } (\prod x \in \#A. p x) y = (\prod x \in \#A. \text{poly } (p x) y)$
by (*induction A*) *auto*

lemma *set-mset-subset-singletonD*:
assumes *set-mset* $A \subseteq \{x\}$
shows $A = \text{replicate-mset } (\text{size } A) x$
using *assms* **by** (*induction A*) *auto*

lemma *inj-idom-hom-compose* [*intro*]:
assumes *inj-idom-hom* f *inj-idom-hom* g
shows *inj-idom-hom* $(f \circ g)$

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 *coeff-pcompose-linear*:
 $\text{coeff } (pcompose p [:0, a :: 'a :: comm-semiring-1:]) i = a \wedge i * \text{coeff } p i$
by (*induction p arbitrary: i*) (*auto simp: pcompose-pCons coeff-pCons mult-ac split: nat.splits*)

lemma *degree-cong*:
assumes $\bigwedge i. \text{coeff } p i = 0 \longleftrightarrow \text{coeff } q i = 0$
shows $\text{degree } p = \text{degree } q$

proof –

have $(\lambda n. \forall i > n. \text{poly.coeff } p i = 0) = (\lambda n. \forall i > n. \text{poly.coeff } q i = 0)$

using *assms* **by** (*auto simp: fun-eq-iff*)

thus *?thesis*

by (*simp only: degree-def*)

qed

lemma *coeff-Abs-poly-If-le*:
 $\text{coeff } (\text{Abs-poly } (\lambda i. \text{if } i \leq n \text{ then } f i \text{ else } 0)) = (\lambda i. \text{if } i \leq n \text{ then } f i \text{ else } 0)$
proof (*rule Abs-poly-inverse, clarify*)
have *eventually* $(\lambda i. i > n)$ *cofinite*

by (auto simp: MOST-nat)
 thus eventually ($\lambda i. (if\ i \leq n\ then\ f\ i\ else\ 0) = 0$) cofinite
 by eventually-elim auto
 qed

lemma *coeff-Abs-poly*:
 assumes $\bigwedge i. i > n \implies f\ i = 0$
 shows $coeff\ (Abs\text{-}poly\ f) = f$
proof (rule *Abs-poly-inverse*, clarify)
 have eventually ($\lambda i. i > n$) cofinite
 by (auto simp: MOST-nat)
 thus eventually ($\lambda i. f\ i = 0$) cofinite
 by eventually-elim (use *assms* in *auto*)
 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*]: $degree\ (p :: 'a :: idom\ poly) \neq 0$
 shows *inj-idom-hom* ($\lambda q. pcompose\ q\ p$)
 by *unfold-locales* (*simp-all add: pcompose-eq-0*)

lemma *degree-sum-less*:
 assumes $\bigwedge x. x \in A \implies degree\ (f\ x) < n\ n > 0$
 shows $degree\ (sum\ f\ A) < n$
 using *assms* by (*induction rule: infinite-finite-induct*) (*auto intro!: degree-add-less*)

lemma *degree-lessI*:
 assumes $p \neq 0 \vee n > 0 \forall k \geq n. coeff\ p\ k = 0$
 shows $degree\ p < n$
proof (*cases p = 0*)
 case *False*
 show ?thesis
proof (*rule ccontr*)
 assume *: $\neg (degree\ p < n)$
 define *d* where $d = degree\ p$
 from $\langle p \neq 0 \rangle$ have $coeff\ p\ d \neq 0$
 by (*auto simp: d-def*)
 moreover have $coeff\ p\ d = 0$
 using *assms*(2) * by (*auto simp: not-less*)
 ultimately show *False* by *contradiction*
 qed
 qed (*use assms in auto*)

lemma *coeff-linear-poly-power*:
fixes $c :: 'a :: \text{semiring-1}$
assumes $i \leq n$
shows $\text{coeff } ([:a, b:] \wedge n) i = \text{of-nat } (n \text{ choose } i) * b \wedge i * a \wedge (n - i)$
proof –
have $[:a, b:] = \text{monom } b \ 1 + [:a:]$
by (*simp add: monom-altdef*)
also have $\text{coeff } (\dots \wedge n) i = (\sum_{k \leq n} a \wedge (n-k) * \text{of-nat } (n \text{ choose } k) * (\text{if } k = i \text{ then } b \wedge k \text{ else } 0))$
by (*subst binomial-ring*) (*simp add: coeff-sum of-nat-poly monom-power poly-const-pow mult-ac*)
also have $\dots = (\sum_{k \in \{i\}} a \wedge (n - i) * b \wedge i * \text{of-nat } (n \text{ choose } k))$
using *assms* **by** (*intro sum.mono-neutral-cong-right*) (*auto simp: mult-ac*)
finally show $*$: *?thesis* **by** (*simp add: mult-ac*)
qed

lemma *pcompose-altdef*: $p \text{ compose } q = \text{poly } (\text{map-poly } (\lambda x. [:x:])) p \ q$
by (*induction p*) *simp-all*

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

lemma *coeff-pcompose-0* [*simp*]:
 $\text{coeff } (p \text{ compose } q) \ 0 = \text{poly } p \ (\text{coeff } q \ 0)$
by (*induction p*) (*simp-all add: coeff-mult-0*)

lemma *reducible-polyI*:
fixes $p :: 'a :: \text{field poly}$
assumes $p = q * r \ \text{degree } q > 0 \ \text{degree } r > 0$
shows $\neg \text{irreducible } p$
using *assms* **by** (*auto simp: irreducible-def*)

lemma *root-imp-reducible-poly*:
fixes $x :: 'a :: \text{field}$
assumes $\text{poly } p \ x = 0$ **and** $\text{degree } p > 1$
shows $\neg \text{irreducible } p$
proof –
from *assms* **have** $p \neq 0$
by *auto*
define q **where** $q = [:-x, 1:]$
have $q \ \text{dvd } p$
using *assms* **by** (*simp add: poly-eq-0-iff-dvd q-def*)
then obtain r **where** $p\text{-eq}: p = q * r$
by (*elim dvdE*)
have [*simp*]: $q \neq 0 \ r \neq 0$
using $\langle p \neq 0 \rangle$ **by** (*auto simp: p-eq*)
have $\text{degree } p = \text{Suc } (\text{degree } r)$
unfolding $p\text{-eq}$ **by** (*subst degree-mult-eq*) (*auto simp: q-def*)

```

with assms(2) have degree  $r > 0$ 
  by auto
hence  $\neg$ is-unit  $r$ 
  by auto
moreover have  $\neg$ is-unit  $q$ 
  by (auto simp: q-def)
ultimately show ?thesis using p-eq
  by (auto simp: irreducible-def)
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:  $n > 0 \implies f\ n \neq 0 \implies \exists x. (\sum_{k \leq n}. f\ k * x^k) = 0$ 

```

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

```

lemma alg-closed-imp-poly-has-root:
  assumes degree ( $p :: 'a :: \text{alg-closed-field poly}$ )  $> 0$ 
  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  $\bigwedge p :: 'a \text{ poly}. \text{degree } p > 0 \implies \text{lead-coeff } p = 1 \implies \exists x. \text{poly } p\ x = 0$ 
  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  $p :: 'a :: \text{alg-closed-field}$  poly
  assumes  $p \neq 0$ 
  shows  $\exists A. \text{size } A = \text{degree } p \wedge p = \text{smult } (\text{lead-coeff } p) (\prod_{x \in \#A}. [:-x, 1:])$ 
  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  $x: \text{poly } p x = 0$ 
    using alg-closed-imp-poly-has-root by blast
  hence  $[:-x, 1:] \text{ dvd } p$ 
    using poly-eq-0-iff-dvd by blast
  then obtain  $q$  where  $p\text{-eq}: p = [:-x, 1:] * q$ 
    by (elim dvdE)
  have  $q \neq 0$ 
    using less.premis  $p\text{-eq}$  by auto
  moreover from this have deg: degree  $p = \text{Suc } (\text{degree } q)$ 

```

```

    unfolding p-eq by (subst degree-mult-eq) auto
    ultimately obtain A where A: size A = degree q q = smult (lead-coeff q)
  (∏ x∈#A. [-x, 1:])
    using less.hyps[of q] by auto
    have smult (lead-coeff p) (∏ y∈#add-mset x A. [-y, 1:]) =
      [-x, 1:] * smult (lead-coeff q) (∏ y∈#A. [-y, 1:])
    unfolding p-eq lead-coeff-mult by simp
    also note A(2) [symmetric]
    also note p-eq [symmetric]
    finally show ?thesis using A(1)
      by (intro exI[of - add-mset x A]) (auto simp: deg)
  qed
qed

```

As an alternative characterisation of algebraic closure, one can also say that any polynomial of degree at least 2 splits into non-constant factors:

lemma *alg-closed-imp-reducible*:

```

  assumes degree (p :: 'a :: alg-closed-field poly) > 1
  shows ¬irreducible p

```

proof –

```

  have degree p > 0
    using assms by auto
  then obtain z where z: poly p z = 0
    using alg-closed-imp-poly-has-root[of p] by blast
  then have dvd: [-z, 1:] dvd p
    by (subst dvd-iff-poly-eq-0) auto
  then obtain q where q: p = [-z, 1:] * q
    by (erule dvdE)
  have [simp]: q ≠ 0
    using assms q by auto

```

show ?thesis

proof (rule reducible-polyI)

```

  show p = [-z, 1:] * q
    by fact

```

next

```

  have degree p = degree ([-z, 1:] * q)
    by (simp only: q)
  also have ... = degree q + 1
    by (subst degree-mult-eq) auto
  finally show degree q > 0
    using assms by linarith

```

qed auto

qed

When proving algebraic closure through reducibility, we can assume w.l.o.g. that the polynomial is monic and has a non-zero constant coefficient:

lemma *alg-closedI-reducible*:

```

  assumes ∧p :: 'a poly. degree p > 1 ⇒ lead-coeff p = 1 ⇒ coeff p 0 ≠ 0 ⇒

```



```

       $\neg$ irreducible  $p$ 
shows OFCLASS('a :: field, alg-closed-field-class)
proof
  fix  $p :: 'a$  poly assume  $p$ : degree  $p > 0$  lead-coeff  $p = 1$ 
  show  $\exists x. \text{poly } p \ x = 0$ 
  proof (cases coeff  $p$  0 = 0)
    case True
      hence 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 poly  $p$   $(-a/b) = 0$ 
              by (auto simp:  $p$ )
              thus ?thesis by blast
            next
              case False
              hence degree  $p > 1$ 
              using less.prem by auto
              from assms[OF  $\langle$ degree  $p > 1$  $\rangle$   $\langle$ lead-coeff  $p = 1$  $\rangle$   $\langle$ coeff  $p$  0  $\neq$  0 $\rangle$ ]
              have  $\neg$ irreducible  $p$  by auto
              then obtain  $r$   $s$  where  $rs$ : degree  $r > 0$  degree  $s > 0$   $p = r * s$ 
              using less.prem by (auto simp: irreducible-def)
              hence coeff  $r$  0  $\neq$  0
              using  $\langle$ coeff  $p$  0  $\neq$  0 $\rangle$  by (auto simp: coeff-mult-0)

              define  $r'$  where  $r' = \text{smult } (\text{inverse } (\text{lead-coeff } r)) \ r$ 
              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  $\neq$  0
              using  $\langle$ coeff  $r$  0  $\neq$  0 $\rangle$  by (auto simp:  $r'$ -def)

              have degree  $r <$  degree  $r +$  degree  $s$ 
              using  $rs$  by linarith
              also have ... = degree  $(r * s)$ 
              using  $rs(3)$  less.prem by (subst degree-mult-eq) auto
              also have  $r * s = p$ 

```

```

    using rs(β) by simp
  finally have ∃ x. poly r' x = 0
    by (intro less) (use lc rs nz in auto)
  thus ?thesis
    using rs(β) 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 {})).

```

$\text{coeff } p \ i * (\text{of-nat } (i \text{ choose } k) * (-a / \text{of-nat } n) ^{\wedge} (i - k))$
 by (rule sum.mono-neutral-right) auto
 also have $\dots = a - \text{of-nat } (n \text{ choose } (n - 1)) * a / \text{of-nat } n$
 using 1 by (simp add: a-def)
 also have $n \text{ choose } (n - 1) = n$
 using $\langle n > 0 \rangle$ by (subst binomial-symmetric) auto
 also have $a - \text{of-nat } n * a / \text{of-nat } n = 0$
 using $\langle n > 0 \rangle$ by simp
 finally have $\text{coeff } q \ (n - 1) = 0$.

show ?case
 proof (cases coeff q 0 = 0)
 case True
 hence $\text{poly } p \ (- (a / \text{of-nat } (\text{degree } p))) = 0$
 by (auto simp: q-def r-def)
 thus ?thesis
 by (rule root-imp-reducible-poly) (use 1 in auto)
 next
 case False
 hence $\neg \text{irreducible } q$
 using assms[of q] and lc-q and 1 and $\langle \text{coeff } q \ (n - 1) = 0 \rangle$
 by (auto simp: deg-q)
 then obtain $u \ v$ where uv : $\text{degree } u > 0 \ \text{degree } v > 0 \ q = u * v$
 using $\langle q \neq 0 \rangle$ 1 deg-q by (auto simp: irreducible-def)

 have $p = \text{pcompose } q \ s$
 by (simp add: q-def r-def s-def flip: pcompose-assoc)
 also have $q = u * v$
 by fact
 finally have $p = \text{pcompose } u \ s * \text{pcompose } v \ s$
 by (simp add: pcompose-mult)
 moreover have $\text{degree } (\text{pcompose } u \ s) > 0 \ \text{degree } (\text{pcompose } v \ s) > 0$
 using uv by (simp-all add: s-def)
 ultimately show $\neg \text{irreducible } p$
 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 $\text{degree } (p :: 'a :: \{\text{alg-closed-field}\} \ \text{poly}) > 1$
 assumes $\text{poly } p \ x = 0 \ \text{poly } p \ y = 0 \ x \neq y$
 obtains $r \ s$ where $\text{degree } r > 0 \ \text{degree } s > 0 \ \text{coprime } r \ s \ p = r * s$
proof –
 define n where $n = \text{order } x \ p$
 have $n > 0$
 using assms by (metis degree-0 grOI n-def not-one-less-zero order-root)

```

have [:-x, 1:] ^ n dvd p
  unfolding n-def by (simp add: order-1)
then obtain q where p-eq: p = [:-x, 1:] ^ n * q
  by (elim dvdE)
from assms have [simp]: q ≠ 0
  by (auto simp: p-eq)
have order x p = n + Polynomial.order x q
  unfolding p-eq by (subst order-mult) (auto simp: order-power-n-n)
hence Polynomial.order x q = 0
  by (simp add: n-def)
hence poly q x ≠ 0
  by (simp add: order-root)

show ?thesis
proof (rule that)
  show coprime ([:-x, 1:] ^ n) q
  proof (rule coprimeI)
    fix d
    assume d: d dvd [:-x, 1:] ^ n d dvd q
    have degree d = 0
    proof (rule ccontr)
      assume ¬(degree d = 0)
      then obtain z where z: poly d z = 0
        using alg-closed-imp-poly-has-root by blast
      moreover from this and d(1) have poly ([:-x, 1:] ^ n) z = 0
        using dvd-trans poly-eq-0-iff-dvd by blast
      ultimately have poly d x = 0
        by auto
      with d(2) have poly q x = 0
        using dvd-trans poly-eq-0-iff-dvd by blast
      with ⟨poly q x ≠ 0⟩ show False by contradiction
    qed
    thus is-unit d using d
      by 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

```

1.3 Facts about Laurent series

```
theory Puiseux-Laurent-Library
  imports HOL-Computational-Algebra.Computational-Algebra
begin
```

```
lemma filterlim-at-top-div-const-nat:
  assumes  $c > 0$ 
  shows filterlim ( $\lambda x::nat. x \text{ div } c$ ) at-top at-top
  unfolding filterlim-at-top
proof
  fix  $C :: nat$ 
  have  $*: n \text{ div } c \geq C$  if  $n \geq C * c$  for  $n$ 
    using assms that by (metis div-le-mono div-mult-self-is-m)
  have eventually ( $\lambda n. n \geq C * c$ ) at-top
    by (rule eventually-ge-at-top)
  thus eventually ( $\lambda n. n \text{ div } c \geq C$ ) at-top
    by eventually-elim (use * in auto)
qed
```

```
lemma fls-eq-iff:  $f = g \iff (\forall n. \text{fls-nth } f \ n = \text{fls-nth } g \ n)$ 
  by transfer auto
```

```
lemma fls-shift-eq-1-iff:  $\text{fls-shift } n \ f = 1 \iff f = \text{fls-shift } (-n) \ 1$ 
proof -
  have  $\text{fls-shift } n \ f = 1 \iff \text{fls-shift } n \ f = \text{fls-shift } n \ (\text{fls-shift } (-n) \ 1)$ 
    by (simp del: fls-shift-eq-iff)
  also have  $\dots \iff f = \text{fls-shift } (-n) \ 1$ 
    by (subst fls-shift-eq-iff) auto
  finally show ?thesis .
qed
```

```
lemma fps-to-fls-eq-iff [simp]:  $\text{fps-to-fls } f = \text{fps-to-fls } g \iff f = g$ 
proof safe
  assume  $\text{fps-to-fls } f = \text{fps-to-fls } g$ 
  hence  $*: n < 0 \vee f \ \$ \ nat \ n = g \ \$ \ nat \ n$  for  $n$ 
    by (force simp: fls-eq-iff split: if-splits)
  have  $f \ \$ \ n = g \ \$ \ n$  for  $n$ 
    using *[of int n] by auto
  thus  $f = g$ 
    by (auto simp: fps-eq-iff)
qed
```

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

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

```
lemma fps-to-fls-power:  $\text{fps-to-fls } (f \wedge n) = \text{fps-to-fls } f \wedge n$ 
```

by (induction n) (auto simp: fls-times-fps-to-fls)

lemma *fls-as-fps*:

fixes $f :: 'a :: \text{zero fls}$ and $n :: \text{int}$

assumes $n: n \geq -\text{fls-subdegree } f$

obtains f' where $f = \text{fls-shift } n (\text{fps-to-fls } f')$

proof –

have $\text{fls-subdegree } (\text{fls-shift } (-n) f) \geq 0$

by (rule *fls-shift-nonneg-subdegree*) (use n in *simp*)

hence $f = \text{fls-shift } n (\text{fps-to-fls } (\text{fls-regpart } (\text{fls-shift } (-n) f)))$

by (*subst fls-regpart-to-fls-trivial*) *simp-all*

thus *?thesis*

by (*rule that*)

qed

lemma *fls-as-fps'*:

fixes $f :: 'a :: \text{zero fls}$ and $n :: \text{int}$

assumes $n: n \geq -\text{fls-subdegree } f$

shows $\exists f'. f = \text{fls-shift } n (\text{fps-to-fls } f')$

using *fls-as-fps*[*OF assms*] by *metis*

definition *fls-compose-fps* :: $'a :: \text{field fls} \Rightarrow 'a \text{ fps} \Rightarrow 'a \text{ fls}$ where

fls-compose-fps f $g =$

(if $f = 0$ then 0

else if $\text{fls-subdegree } f \geq 0$ then $\text{fps-to-fls } (\text{fps-compose } (\text{fls-regpart } f) g)$

else $\text{fps-to-fls } (\text{fps-compose } (\text{fls-base-factor-to-fps } f) g) /$

$\text{fps-to-fls } g \wedge \text{nat } (-\text{fls-subdegree } f))$

lemma *fls-compose-fps-fps* [*simp*]:

fls-compose-fps ($\text{fps-to-fls } f$) $g = \text{fps-to-fls } (\text{fps-compose } f g)$

by (*auto simp: fls-compose-fps-def fls-subdegree-fls-to-fps*)

lemma *fls-const-transfer* [*transfer-rule*]:

rel-fun (=) (*pcr-fls* (=))

($\lambda c n. \text{if } n = 0 \text{ then } c \text{ else } 0$) *fls-const*

by (*auto simp: fls-const-def rel-fun-def pcr-fls-def OO-def cr-fls-def*)

lemma *fls-shift-transfer* [*transfer-rule*]:

rel-fun (=) (*rel-fun* (*pcr-fls* (=)) (*pcr-fls* (=)))

($\lambda n f k. f (k+n)$) *fls-shift*

by (*auto simp: fls-const-def rel-fun-def pcr-fls-def OO-def cr-fls-def*)

lift-definition *fls-compose-power* :: $'a :: \text{zero fls} \Rightarrow \text{nat} \Rightarrow 'a \text{ fls}$ is

$\lambda f d n. \text{if } d > 0 \wedge \text{int } d \text{ dvd } n \text{ then } f (n \text{ div int } d) \text{ else } 0$

proof –

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

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

show *eventually* ($\lambda n. (\text{if } d > 0 \wedge \text{int } d \text{ dvd } -\text{int } n \text{ then } f (-\text{int } n \text{ div int } d) \text{ else } 0$)

$0) = 0)$ *cofinite*
proof (*cases* $d = 0$)
 case *False*
 from * **have** *eventually* $(\lambda n. f (-int\ n) = 0)$ *at-top*
 by (*simp add: cofinite-eq-sequentially*)
 hence *eventually* $(\lambda n. f (-int\ (n\ div\ d)) = 0)$ *at-top*
 by (*rule eventually-compose-filterlim[OF - filterlim-at-top-div-const-nat]*) (*use False in auto*)
 hence *eventually* $(\lambda n. (if\ d > 0 \wedge int\ d\ dvd\ -int\ n\ then\ f\ (-int\ n\ div\ int\ d)\ else\ 0) = 0)$ *at-top*
 by *eventually-elim (auto simp: zdiv-int dvd-neg-div)*
 thus *?thesis*
 by (*simp add: cofinite-eq-sequentially*)
qed *auto*
qed

lemma *fls-nth-compose-power:*
 assumes $d > 0$
 shows $fls\ nth\ (fls\ compose\ power\ f\ d)\ n = (if\ int\ d\ dvd\ n\ then\ fls\ nth\ f\ (n\ div\ int\ d)\ else\ 0)$
 using *assms* **by** *transfer auto*

lemma *fls-compose-power-0-left [simp]:* $fls\ compose\ power\ 0\ d = 0$
by *transfer auto*

lemma *fls-compose-power-1-left [simp]:* $d > 0 \implies fls\ compose\ power\ 1\ d = 1$
by *transfer (auto simp: fun-eq-iff)*

lemma *fls-compose-power-const-left [simp]:*
 $d > 0 \implies fls\ compose\ power\ (fls\ const\ c)\ d = fls\ const\ c$
by *transfer (auto simp: fun-eq-iff)*

lemma *fls-compose-power-shift [simp]:*
 $d > 0 \implies fls\ compose\ power\ (fls\ shift\ n\ f)\ d = fls\ shift\ (d * n)\ (fls\ compose\ power\ f\ d)$
by *transfer (auto simp: fun-eq-iff add-ac mult-ac)*

lemma *fls-compose-power-X-intpow [simp]:*
 $d > 0 \implies fls\ compose\ power\ (fls\ X\ intpow\ n)\ d = fls\ X\ intpow\ (int\ d * n)$
by *simp*

lemma *fls-compose-power-X [simp]:*
 $d > 0 \implies fls\ compose\ power\ fls\ X\ d = fls\ X\ intpow\ (int\ d)$
by *transfer (auto simp: fun-eq-iff)*

lemma *fls-compose-power-X-inv [simp]:*
 $d > 0 \implies fls\ compose\ power\ fls\ X\ inv\ d = fls\ X\ intpow\ (-int\ d)$
by (*simp add: fls-X-inv-conv-shift-1*)

lemma *fls-compose-power-0-right* [*simp*]: *fls-compose-power* f $0 = 0$
by *transfer auto*

lemma *fls-compose-power-add* [*simp*]:
fls-compose-power $(f + g)$ $d = fls-compose-power$ f $d + fls-compose-power$ g d
by *transfer auto*

lemma *fls-compose-power-diff* [*simp*]:
fls-compose-power $(f - g)$ $d = fls-compose-power$ f $d - fls-compose-power$ g d
by *transfer auto*

lemma *fls-compose-power-uminus* [*simp*]:
fls-compose-power $(-f)$ $d = -fls-compose-power$ f d
by *transfer auto*

lemma *fps-nth-compose-X-power*:
fps-nth $(f$ *oo* $(fps-X \wedge d))$ $n = (if$ d *dvd* n *then* *fps-nth* f $(n$ *div* $d)$ *else* $0)$
proof –
have *fps-nth* $(f$ *oo* $(fps-X \wedge d))$ $n = (\sum$ $i = 0..n.$ f $\$$ $i * (fps-X \wedge (d * i))$ $\$$ $n)$
unfolding *fps-compose-def* **by** (*simp add: power-mult*)
also have $\dots = (\sum$ $i \in (if$ d *dvd* n *then* $\{n$ *div* $d\}$ *else* $\{\}$) $. f$ $\$$ $i * (fps-X \wedge (d * i))$ $\$$ $n)$
by (*intro sum.mono-neutral-right*) *auto*
also have $\dots = (if$ d *dvd* n *then* *fps-nth* f $(n$ *div* $d)$ *else* $0)$
by *auto*
finally show *?thesis* .
qed

lemma *fls-compose-power-fps-to-fls*:
assumes $d > 0$
shows *fls-compose-power* $(fps-to-fls$ $f)$ $d = fps-to-fls$ $(fps-compose$ f $(fps-X \wedge d))$
using *assms*
by (*intro fls-eqI*) (*auto simp: fls-nth-compose-power fps-nth-compose-X-power pos-imp-zdiv-neg-iff div-neg-pos-less0 nat-div-distrib simp flip: int-dvd-int-iff*)

lemma *fls-compose-power-mult* [*simp*]:
fls-compose-power $(f * g :: 'a :: idom$ *fls*) $d = fls-compose-power$ f $d * fls-compose-power$ g d
proof (*cases* $d > 0$)
case *True*
define n **where** $n = nat$ $(max$ 0 $(max$ $(- fls-subdegree$ $f)$ $(- fls-subdegree$ $g)))$
have $n-ge$: $-fls-subdegree$ $f \leq int$ $n - fls-subdegree$ $g \leq int$ n
unfolding *n-def* **by** *auto*
obtain f' **where** $f' : f = fls-shift$ n $(fps-to-fls$ $f')$
using *fls-as-fps[OF n-ge(1)]* **by** (*auto simp: n-def*)
obtain g' **where** $g' : g = fls-shift$ n $(fps-to-fls$ $g')$


```

    using fls-as-fps[OF n-ge(2)] by (auto simp: n-def)
  show ?thesis using ‹d > 0›
    by (simp add: f' g' fls-shifted-times-simps mult-ac fls-compose-power-fps-to-fls
        fps-compose-mult-distrib flip: fls-times-fps-to-fls)
qed auto

```

```

lemma fls-compose-power-power [simp]:
  assumes d > 0 ∨ n > 0
  shows fls-compose-power (f ^ n :: 'a :: idom fls) d = fls-compose-power f d ^ n
proof (cases d > 0)
  case True
  thus ?thesis by (induction n) auto
qed (use assms in auto)

```

```

lemma fls-nth-compose-power' [simp]:
  d = 0 ∨ ¬d dvd n ⇒ fls-nth (fls-compose-power f d) n = 0
  d dvd n ⇒ d > 0 ⇒ fls-nth (fls-compose-power f d) n = fls-nth f (n div d)
  by (transfer; force; fail)+

```

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 ⇒ 'a poly fps where
  fps-poly-swap1 p = Abs-fps (λm. Abs-poly (λn. fps-nth (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 ∀ ∞ n. poly.coeff p n = 0
    using MOST-coeff-eq-0 by blast
  hence ∀ ∞ n. 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 ⇒ 'a fps poly where
  fps-poly-swap2 p = Abs-poly (λm. Abs-fps (λn. coeff (fps-nth p n) m))

```

```

lemma fps-nth-coeff-fps-poly-swap2:

```

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

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

lemma *degree-fps-poly-swap2-le*:

assumes $\bigwedge n. \text{degree } (fps\text{-nth } p \ n) \leq d$
shows $\text{degree } (fps\text{-poly}\text{-swap2 } p) \leq d$
proof (*safe intro!: degree-le*)
fix *n* **assume** $n > d$
show $poly.coeff \ (fps\text{-poly}\text{-swap2 } p) \ n = 0$
proof (*rule fps-ext*)
fix *m*
have $poly.coeff \ (fps\text{-poly}\text{-swap2 } p) \ n \ \$ \ m = poly.coeff \ (p \ \$ \ m) \ n$
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 $poly.coeff \ (fps\text{-poly}\text{-swap2 } p) \ n \ \$ \ m = 0 \ \$ \ m$
by *simp*
qed
qed

lemma *degree-fps-poly-swap2-eq*:

assumes $\bigwedge n. \text{degree } (fps\text{-nth } p \ n) \leq d$
assumes $d > 0 \vee fps\text{-nth } p \ n \neq 0$
assumes $\text{degree } (fps\text{-nth } p \ n) = d$
shows $\text{degree } (fps\text{-poly}\text{-swap2 } p) = d$
proof (*rule antisym*)
have $fps\text{-nth } (coeff \ (fps\text{-poly}\text{-swap2 } p) \ d) \ n = poly.coeff \ (fps\text{-nth } p \ n) \ d$
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 $coeff \ (fps\text{-poly}\text{-swap2 } p) \ d \neq 0$
by *force*

```

thus degree (fps-poly-swap2 p) ≥ d
  using le-degree by blast
next
  show degree (fps-poly-swap2 p) ≤ d
    by (intro degree-fps-poly-swap2-le) fact
qed

definition reduce-fps-poly :: 'a :: zero fps poly ⇒ 'a poly where
  reduce-fps-poly F = fps-nth (fps-poly-swap1 F) 0

lemma
  fixes F :: 'a :: field fps poly
  assumes lead-coeff F = 1
  shows degree-reduce-fps-poly-monic: degree (reduce-fps-poly F) = degree F
    and reduce-fps-poly-monic: lead-coeff (reduce-fps-poly F) = 1
proof -
  have eq1: coeff (reduce-fps-poly F) (degree F) = 1
    unfolding reduce-fps-poly-def by (simp add: assms)
  have eq2: coeff (reduce-fps-poly F) n = 0 if n > degree F for n
    unfolding reduce-fps-poly-def using that by (simp add: coeff-eq-0)

  have degree (reduce-fps-poly F) ≤ degree F
    by (rule degree-le) (auto simp: eq2)
  moreover have degree (reduce-fps-poly F) ≥ degree F
    by (rule le-degree) (simp add: eq1)
  from eq1 eq2 show degree (reduce-fps-poly F) = degree F
    by (intro antisym le-degree degree-le) auto
  with eq1 show lead-coeff (reduce-fps-poly F) = 1
    by simp
qed

locale fps-hensel-aux =
  fixes F :: 'a :: field-gcd poly fps
  fixes g h :: 'a poly
  assumes coprime: coprime g h and deg-g: degree g > 0 and deg-h: degree h > 0
begin

context
  fixes g' h' :: 'a poly
  defines h' ≡ fst (bezout-coefficients g h) and g' ≡ snd (bezout-coefficients g h)
begin

fun hensel-fpxs-aux :: nat ⇒ 'a poly × 'a poly where
  hensel-fpxs-aux n = (if n = 0 then (g, h) else
    (let
      U = fps-nth F n -
        (∑ (i,j) | i < n ∧ j < n ∧ i + j = n. fst (hensel-fpxs-aux i) * 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 $hensel-fpxs1 = Abs-fps\ (fst\ \circ\ hensel-fpxs-aux)$

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

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 $f \equiv fps-nth\ F\ 0$
assumes $f = g * h$
assumes $\forall n > 0. degree\ (fps-nth\ F\ n) < degree\ f$
defines $G \equiv hensel-fpxs1$ **and** $H \equiv hensel-fpxs2$
shows $F = G * H\ fps-nth\ G\ 0 = g\ fps-nth\ H\ 0 = h$
 $\forall n > 0. degree\ (fps-nth\ G\ n) < degree\ g$
 $\forall n > 0. degree\ (fps-nth\ H\ n) < degree\ h$

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 $\langle f = g * h \rangle$ **using** $assms$ **by** $(intro\ degree-mult-eq)\ auto$

have $deg-H: degree\ (fps-nth\ H\ n) < degree\ h$ **if** $\langle n > 0 \rangle$ **for** n
proof $(cases\ snd\ (hensel-fpxs-aux\ n) = 0)$
case $False$
thus $?thesis$
using $deg-h\ \langle n > 0 \rangle$
by $(auto\ simp:\ hensel-fpxs-aux.simps[of\ n]\ hensel-fpxs2-def\ H-def\ intro:\ de-
gree-mod-less')$

qed $(use\ assms\ deg-h\ in\ \langle auto\ simp:\ hensel-fpxs2-def \rangle)$
thus $\forall n > 0. degree\ (fps-nth\ H\ n) < degree\ h$
by $blast$

have $*$: $fps-nth\ F\ n = fps-nth\ (G * H)\ n \wedge (n > 0 \longrightarrow degree\ (fps-nth\ G\ n) < degree\ g)$ **for** n
proof $(induction\ n\ rule:\ less-induct)$
case $(less\ n)$
have $fin: finite\ \{p. fst\ p < n \wedge snd\ p < n \wedge fst\ p + snd\ p = n\}$

```

    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 -
    (∑ (i,j) | i < n ∧ j < n ∧ i + j = n. fst (hensel-fpxs-aux i) * snd
(hensel-fpxs-aux j))
  define g'' h'' where g'' = U * g' and h'' = U * h'

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

with  $\langle \text{fps-nth } F \ n = \text{fps-nth } (G * H) \ n \rangle$  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  $F :: 'a :: \text{field-gcd } \text{fps } \text{poly}$  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-locale (fact deg-g deg-h coprime)+

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

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

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

```

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

proof –

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

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

next

have $\text{coeff } f \ n = 0$ **if** $n > N$ **for** n
using *that* **by** (*simp add: f-def reduce-fps-poly-def N-def coeff-eq-0*)
thus $\text{degree } f \leq N$
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 $n \text{ deg-g deg-h f-splits deg-f}$ **by** (*auto simp: F'0 degree-mult-eq*)
fix k
assume $k: k \geq N$
have $\text{coeff } (F' \$ n) \ k = \text{coeff } F \ k \$ n$
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 $\text{coeff } (\text{fps-poly-swap1 } F \$ n) \ k = 0$
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* = *fps-hensel-aux*[*OF F'0 degs-less*]

have *deg-less1*: $\text{degree } (\text{hensel-fpxs1 } \$ n) < \text{degree } g$ **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 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: degree (hensel-fpxs2 $ n) < degree h if n > 0 for n
  using hensel(5) that by (simp add: F'-def)
have deg-le2: degree (hensel-fpxs2 $ n) ≤ degree h for n
proof (cases n = 0)
  case True
  hence 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 poly.coeff F n $ k = poly.coeff (F' $ k) n
    unfolding F'-def by simp
  also have F' = hensel-fpxs1 * hensel-fpxs2
    by (rule hensel)
  also have ... $ k = (∑ i=0..k. hensel-fpxs1 $ i * hensel-fpxs2 $ (k - i))
    unfolding fps-mult-nth ..
  also have poly.coeff ... n =
    (∑ i=0..k. ∑ j≤n. coeff (hensel-fpxs1 $ i) j * coeff (hensel-fpxs2 $
(k - i)) (n - j))
    by (simp add: coeff-sum coeff-mult)
  also have (λ i j. coeff (hensel-fpxs1 $ i) j) = (λ i j. coeff G j $ i)
    unfolding G-def
    by (subst fps-nth-coeff-fps-poly-swap2[OF deg-le1]) (auto simp: F'-def)
  also have (λ i j. coeff (hensel-fpxs2 $ i) j) = (λ i j. coeff H j $ i)
    unfolding H-def
    by (subst fps-nth-coeff-fps-poly-swap2[OF deg-le2]) (auto simp: F'-def)
  also have (∑ i=0..k. ∑ j≤n. poly.coeff G j $ i * poly.coeff H (n - j) $ (k -
i)) =
    (∑ j≤n. ∑ i=0..k. poly.coeff G j $ i * poly.coeff H (n - j) $ (k - i))
    by (rule sum.swap)
  also have ... = poly.coeff (G * H) n $ k
    by (simp add: coeff-mult fps-mult-nth fps-sum-nth)
  finally show poly.coeff F n $ k = poly.coeff (G * H) n $ k .
qed

show 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 Puiseux-Laurent-Library 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*:
 $(\text{of-int } n / \text{of-int } m :: 'a :: \text{field-char-0}) \in \mathbb{Z} \longleftrightarrow m = 0 \vee m \text{ dvd } n$
proof
assume *: $(\text{of-int } n / \text{of-int } m :: 'a) \in \mathbb{Z}$
{
assume $m \neq 0$
from * **obtain** k **where** $k: (\text{of-int } n / \text{of-int } m :: 'a) = \text{of-int } k$
by (*auto elim!: Ints-cases*)
hence $\text{of-int } n = (\text{of-int } k * \text{of-int } m :: 'a)$
using $\langle m \neq 0 \rangle$ **by** (*simp add: field-simps*)
also have $\dots = \text{of-int } (k * m)$
by *simp*
finally have $n = k * m$
by (*subst (asm) of-int-eq-iff*)
hence $m \text{ dvd } n$ **by** *auto*
}
thus $m = 0 \vee m \text{ dvd } n$ **by** *blast*
qed *auto*

lemma *rat-eq-quotientD*:
assumes $r = \text{rat-of-int } a / \text{rat-of-int } b \text{ } b \neq 0$
shows $\text{fst } (\text{quotient-of } r) \text{ dvd } a \text{ snd } (\text{quotient-of } r) \text{ dvd } b$
proof –
define $a' \ b'$ **where** $a' = \text{fst } (\text{quotient-of } r)$ **and** $b' = \text{snd } (\text{quotient-of } r)$
define d **where** $d = \text{gcd } a \ b$
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 $r: r = \text{rat-of-int } a' / \text{rat-of-int } b'$
by (*simp add: a'-def b'-def quotient-of-div*)
from *assms* $\langle b' > 0 \rangle$ **have** $\text{rat-of-int } (a' * b) = \text{rat-of-int } (a * b')$
unfolding *of-int-mult* **by** (*simp add: field-simps r*)
hence *eq*: $a' * b = a * b'$
by (*subst (asm) of-int-eq-iff*)

have $a' \text{ dvd } a * b'$
by (*simp flip: eq*)
hence $a' \text{ dvd } a$
by (*subst (asm) coprime-dvd-mult-left-iff*) *fact*
moreover have $b' \text{ dvd } a' * b$
by (*simp add: eq*)
hence $b' \text{ dvd } b$
by (*subst (asm) coprime-dvd-mult-right-iff*) (*use* $\langle \text{coprime } a' \ b' \rangle$ **in** $\langle \text{simp add: coprime-commute} \rangle$)
ultimately show $\text{fst } (\text{quotient-of } r) \text{ dvd } a \text{ snd } (\text{quotient-of } r) \text{ dvd } b$
unfolding *a'-def b'-def* **by** *blast+*

qed

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

*snd (quotient-of (x + y)) dvd snd (quotient-of x) * snd (quotient-of y)*

proof –

define *a b* **where** *a = fst (quotient-of x)* **and** *b = snd (quotient-of x)*

define *c d* **where** *c = fst (quotient-of y)* **and** *d = snd (quotient-of y)*

have *b > 0 d > 0*

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

have *xy: x = rat-of-int a / rat-of-int b y = rat-of-int c / rat-of-int d*

unfolding *a-def b-def c-def d-def* **by** (*simp-all add: quotient-of-div*)

show *snd (quotient-of (x + y)) dvd b * d*

proof (*rule rat-eq-quotientD*)

show *x + y = rat-of-int (a * d + c * b) / rat-of-int (b * d)*

using * 0> <d > 0>* **by** (*simp add: field-simps xy*)

qed (*use 0> <d > 0> in auto*)

qed

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

*snd (quotient-of (x - y)) dvd snd (quotient-of x) * snd (quotient-of y)*

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

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

definition *supp* :: (*'a* ⇒ (*'b* :: zero)) ⇒ *'a* set **where**

supp f = f -' (-{0})

lemma *supp-0* [*simp*]: *supp (λ-. 0) = {}*

and *supp-const: supp (λ-. c) = (if c = 0 then {} else UNIV)*

and *supp-singleton* [*simp*]: *c ≠ 0 ⇒ supp (λx. if x = d then c else 0) = {d}*

by (*auto simp: supp-def*)

lemma *supp-uminus* [*simp*]: *supp (λx. -f x :: 'a :: group-add) = 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*: (\wedge 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 $X^{\wedge}\{1/d\}$. The following operation gives us this Laurent series.

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

have $*$: f (of-int n / of-int d) = 0 **if** $n < \lfloor r0 * \text{of-int } d \rfloor$ **for** n

proof –

have *rat-of-int* $n < r0 * \text{rat-of-int } d$
using that by *linarith*
thus *?thesis*
using $\langle d > 0 \rangle$ **by** (*intro r0*) (*auto simp: field-simps*)

qed

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

qed

lemma *fls-nth-of-fpxs*:

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

3.3 Basic algebraic typeclass instances

instantiation *fpxs* :: (zero) zero
begin

lift-definition *zero-fpxs* :: 'a *fpxs* **is** $\lambda r :: \text{rat}. 0 :: 'a$
by (*auto simp: is-fpxs-def supp-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 ≠ 0 ⇒ fpxs-nth 1 r = 0
  by (auto simp: fpxs-nth-1)

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

lift-definition plus-fpxs :: 'a fpxs ⇒ 'a fpxs ⇒ 'a fpxs is
   $\lambda f g x. f x + g x$ 
proof –
  fix f g :: rat ⇒ '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 ( $\lambda x. f x + g x$ ) ⊆ supp f ∪ 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 <auto simp: is-fpxs-def>)
    have (LCM r ∈ supp ( $\lambda x. f x + g x$ )). snd (quotient-of r) dvd
      (LCM r ∈ supp f ∪ supp g. snd (quotient-of r))
      by (intro Lcm-subset image-mono supp)
    also have ... = lcm (LCM r ∈ supp f. snd (quotient-of r)) (LCM r ∈ supp g.
snd (quotient-of r))
      unfolding image-Un Lcm-Un ..
    finally have (LCM r ∈ supp ( $\lambda x. f x + g x$ )). snd (quotient-of r) dvd
      lcm (LCM r ∈ supp f. snd (quotient-of r)) (LCM r ∈ supp g. snd
(quotient-of r)) .
  qed

```

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

instance
by $\text{standard} (\text{transfer}; \text{simp add: algebra-simps fun-eq-iff})+$
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 $\text{eventually} (\lambda n. f (-\text{int } n) = 0)$ cofinite
hence $\text{eventually} (\lambda n. f (-\text{int } n) = 0)$ 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 $\text{bdd-below} (\text{supp} (\lambda r :: \text{rat}. \text{if } r \in \mathbb{Z} \text{ then } f \lfloor r \rfloor \text{ else } 0))$

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

instantiation $\text{fpxs} :: (\text{group-add}) \text{group-add}$
begin

lift-definition $\text{uminus-fpxs} :: 'a \text{fpxs} \Rightarrow 'a \text{fpxs}$ is $\lambda f x. -f x$
 by (auto simp: is-fpxs-def)

definition $\text{minus-fpxs} :: 'a \text{fpxs} \Rightarrow 'a \text{fpxs} \Rightarrow 'a \text{fpxs}$ where
 $\text{minus-fpxs } f g = f + (-g)$

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

end

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

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

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

lemma fpxs-of-fls-0 [simp]: $\text{fpxs-of-fls } 0 = 0$
 by transfer auto

lemma fpxs-of-fls-1 [simp]: $\text{fpxs-of-fls } 1 = 1$
 by transfer (auto simp: fun-eq-iff elim!: Ints-cases)

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

lemma fpxs-to-fls-sum [simp]: $\text{fpxs-to-fls } (\text{sum } f A) = (\sum x \in A. \text{fpxs-to-fls } (f x))$
 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*:

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

lemma *fpxs-of-fls-eq-0-iff* [*simp*]: *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*]: *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*]: *fpxs-root-order* (*fpxs-of-fls* f) = 1

proof (*transfer, goal-cases*)

case (*1 f*)

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

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

also have $\dots = 1$

by *simp*

also have *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 ’ :: *zero fpxs* \Rightarrow *rat* \Rightarrow ‘ a ’ *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 *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 ($\lambda x. x / r'$) ‘*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*: ($\lambda x. x * r$) ‘*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* ($(\lambda x. x * r)$ ‘*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]: $\text{quotient-of } r = (a, b)$

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 $\text{supp } f = \{\}$)

case False

thus ?thesis **using** Lcm-mult[of ($\lambda x. \text{snd} (\text{quotient-of } x)$) 'supp f b]

by (simp add: mult-ac image-image)

qed auto

hence ($\text{LCM } x \in \text{supp } f. \text{snd} (\text{quotient-of } x) * b$) 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:

$\text{fpxs-compose-power} (\text{fpxs-of-fls} (\text{fls-of-fpxs } f)) (1 / \text{of-nat} (\text{fpxs-root-order } f)) =$
 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 $\text{rat-of-int } d * x \in \mathbb{Z}$ 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]:
  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 = 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 rat-of-int n / rat-of-int (int d) ∉ ℤ
            using ⟨d ≠ 0⟩ by (subst of-int-div-of-int-in-Ints-iff) auto
          thus ?thesis using False by auto
        qed
      next
        case False
        hence r / rat-of-nat d ∉ ℤ
          using ⟨d ≠ 0⟩ by (auto elim!: Ints-cases simp: field-simps)
        thus ?thesis using False by auto
      qed
    qed
  qed auto
qed

```

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

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

lemma *fps-compose-power-divide-right*:
 $r1 > 0 \implies r2 > 0 \implies$
 $\text{fps-compose-power } f (r1 / r2) = \text{fps-compose-power } (\text{fps-compose-power } f r1) (\text{inverse } r2)$
by (simp add: fps-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 = *fpxs-compose-power* (*fpxs-of-fls* f') (1 / *of-nat* d)
proof –
define D **where** D = *fpxs-root-order* f
have $D > 0$
by (*auto simp: D-def*)
define f' **where** f' = *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'' = *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 ∈ supp f ∧ z ∈ supp g ∧ r = y + z} ≠ {}

```

```

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

lemma fpxs-nth-mult:
  fpxs-nth (f * g) r =
    (∑ (y,z) | y ∈ fpxs-supp f ∧ z ∈ fpxs-supp g ∧ r = y + z. fpxs-nth f y *
     fpxs-nth g z)
  by transfer simp

lemma fpxs-compose-power-mult [simp]:
  fpxs-compose-power (f * g) r = fpxs-compose-power f r * 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 (∑ x ∈ {(y, z). y ∈ supp f ∧ z ∈ supp g ∧ x / r = y + z}.
          case x of (y, z) ⇒ f y * g z) =
      (∑ x ∈ {(y, z). y ∈ supp (λx. f (x / r)) ∧ z ∈ supp (λx. g (x / r)) ∧ x =
y + z}.
          case x of (y, z) ⇒ 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-supp (fpxs-of-fls f) = of-int ‘ supp (fls-nth f)

```


by (force simp: fpxs-supp-def fpxs-nth-of-fls supp-def elim!: Ints-cases)

lemma fpxs-of-fls-mult [simp]: fpxs-of-fls (f * g) = fpxs-of-fls f * fpxs-of-fls g

proof (rule fpxs-ext)

fix r :: rat

show fpxs-nth (fpxs-of-fls (f * g)) r = fpxs-nth (fpxs-of-fls f * fpxs-of-fls g) r

proof (cases r ∈ ℤ)

case True

define h1 where h1 = (λ(x, y). ([x::rat], [y::rat]))

define h2 where h2 = (λ(x, y). (of-int x :: rat, of-int y :: rat))

define df dg where [simp]: df = fls-subdegree f dg = fls-subdegree g

from True obtain n where [simp]: r = of-int n

by (cases rule: Ints-cases)

have fpxs-nth (fpxs-of-fls f * fpxs-of-fls g) r =
 $(\sum (y,z) \mid y \in \text{fpxs-supp } (fpxs\text{-of-fls } f) \wedge z \in \text{fpxs-supp } (fpxs\text{-of-fls } g) \wedge$
rat-of-int n = y + z.
 (if y ∈ ℤ then fls-nth f [y] else 0) * (if z ∈ ℤ then fls-nth g [z] else 0))

by (auto simp: fpxs-nth-mult fpxs-nth-of-fls)

also have ... = $(\sum (y,z) \mid y \in \text{supp } (fls\text{-nth } f) \wedge z \in \text{supp } (fls\text{-nth } g) \wedge n =$
 y + z.
 fls-nth f y * fls-nth g z)

by (rule sum.reindex-bij-witness[of - h2 h1]) (auto simp: h1-def h2-def fpxs-supp-of-fls)

also have ... = $(\sum y \mid y - \text{fls-subdegree } g \in \text{supp } (fls\text{-nth } f) \wedge \text{fls-subdegree } g$
 + n - y ∈ supp (fls-nth g).
 fls-nth f (y - fls-subdegree g) * fls-nth g (fls-subdegree g + n - y))

by (rule sum.reindex-bij-witness[of - λy. (y - fls-subdegree g, fls-subdegree g + n - y) λz. fst z + fls-subdegree g])

auto

also have ... = $(\sum i = \text{fls-subdegree } f + \text{fls-subdegree } g..n.$
 fls-nth f (i - fls-subdegree g) * fls-nth g (fls-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 ... = fpxs-nth (fpxs-of-fls (f * g)) r

by (auto simp: fls-times-nth fpxs-nth-of-fls)

finally show ?thesis ..

next

case False

have fpxs-nth (fpxs-of-fls f * fpxs-of-fls g) r =
 $(\sum (y,z) \mid y \in \text{fpxs-supp } (fpxs\text{-of-fls } f) \wedge z \in \text{fpxs-supp } (fpxs\text{-of-fls } g) \wedge$
 r = y + z.
 (if y ∈ ℤ then fls-nth f [y] else 0) * (if z ∈ ℤ then fls-nth g [z] else 0))

by (simp add: fpxs-nth-mult fpxs-nth-of-fls)

also have ... = 0

using False by (intro sum.neutral ballI) auto

also have 0 = fpxs-nth (fpxs-of-fls (f * g)) r

using False by (simp add: fpxs-nth-of-fls)

finally show ?thesis ..

qed

qed

instance proof

show $0 \neq (1 :: 'a \text{ fpxs})$
by transfer (auto simp: fun-eq-iff)

next

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

show $1 * f = f$

proof (transfer, goal-cases)

case (1 f)

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

$(\text{if } x \in \text{supp } f \text{ then } \{(0, x)\} \text{ else } \{\}) \text{ for } x$

by (auto simp: supp-def split: if-splits)

thus ?case

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

qed

next

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

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

show $f * g = g * f$

proof (transfer, rule ext, goal-cases)

case (1 f g x)

show $(\sum (y, z) \in \{(y, z). y \in \text{supp } f \wedge z \in \text{supp } g \wedge x = y + z\}. f y * g z) =$
 $(\sum (y, z) \in \{(y, z). y \in \text{supp } g \wedge z \in \text{supp } f \wedge x = y + z\}. g y * f z)$

by (rule sum.reindex-bij-witness[of - $\lambda(x,y). (y,x)$ $\lambda(x,y). (y,x)$])
(auto simp: mult-ac)

qed

next

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

define d where $d = (\text{LCM } F \in \{f, g, h\}. \text{fpxs-root-order } F)$

have $d > 0$

by (auto simp: d-def intro!: Nat.gr0I)

obtain f' where $f: f = \text{fpxs-compose-power } (\text{fpxs-of-fls } f') (1 / \text{of-nat } d)$

using $\text{fpxs-as-fls}[of f d] \langle d > 0 \rangle$ by (auto simp: d-def)

obtain g' where $g: g = \text{fpxs-compose-power } (\text{fpxs-of-fls } g') (1 / \text{of-nat } d)$

using $\text{fpxs-as-fls}[of g d] \langle d > 0 \rangle$ by (auto simp: d-def)

obtain h' where $h: h = \text{fpxs-compose-power } (\text{fpxs-of-fls } h') (1 / \text{of-nat } d)$

using $\text{fpxs-as-fls}[of h d] \langle d > 0 \rangle$ by (auto simp: d-def)

show $(f * g) * h = f * (g * h)$

by (simp add: f g h mult-ac

flip: $\text{fpxs-compose-power-mult fpxs-compose-power-add fpxs-of-fls-mult}$)

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*: $f \neq 0 \ g \neq 0$

define *d* where $d = \text{lcm} (\text{fpxs-root-order } f) (\text{fpxs-root-order } g)$

have $d > 0$

by (*auto simp: d-def intro!: lcm-pos-nat*)

obtain *f'* where $f: f = \text{fpxs-compose-power} (\text{fpxs-of-fls } f') (1 / \text{of-nat } d)$

using *fpxs-as-fls*'[*of f d*] $\langle d > 0 \rangle$ by (*auto simp: d-def*)

obtain *g'* where $g: g = \text{fpxs-compose-power} (\text{fpxs-of-fls } g') (1 / \text{of-nat } d)$

using *fpxs-as-fls*'[*of g d*] $\langle d > 0 \rangle$ by (*auto simp: d-def*)

show $f * g \neq 0$

using $\langle d > 0 \rangle$ *fg*

by (*simp add: f g flip: fpxs-compose-power-mult fpxs-of-fls-mult*)

qed

lemma *fpxs-of-fls-power* [*simp*]: $\text{fpxs-of-fls} (f \wedge n) = \text{fpxs-of-fls } f \wedge n$
by (*induction n auto*)

lemma *fpxs-compose-power-power* [*simp*]:
 $r > 0 \implies \text{fpxs-compose-power} (f \wedge n) r = \text{fpxs-compose-power } f r \wedge n$
by (*induction n simp-all*)

3.6 Constant Puiseux series and the series *X*

lift-definition *fpxs-const* :: 'a :: zero \implies 'a *fpxs* is

$\lambda c n. \text{if } n = 0 \text{ then } c \text{ else } 0$

proof –

fix *c* :: 'a

have *supp* ($\lambda n::\text{rat}. \text{if } n = 0 \text{ then } c \text{ else } 0$) = (*if c = 0 then {} else {0}*)

by *auto*

thus *is-fpxs* ($\lambda n::\text{rat}. \text{if } n = 0 \text{ then } c \text{ else } 0$)

unfolding *is-fpxs-def* by *auto*

qed

lemma *fpxs-const-0* [*simp*]: $\text{fpxs-const } 0 = 0$
by *transfer auto*

lemma *fpxs-const-1* [*simp*]: $\text{fpxs-const } 1 = 1$
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**
 $\lambda x.$ *if* $x = 1$ *then* $(1 :: 'a)$ *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-nat* :: $nat \Rightarrow 'a\ fpxs$)
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 $fpxs\text{-nth} (fpxs\text{-compose-power } f \ r) \ n = fpxs\text{-nth } f \ (n / r)$
using *assms* **by** *transfer auto*

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

lemma *fpxs-root-order-uminus* [simp]: $fpxs\text{-root-order} \ (-f) = fpxs\text{-root-order} \ f$
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*:
 $rat\text{-of-int} \ (fls\text{-subdegree} \ (fls\text{-of-fpxs} \ f)) = fpxs\text{-val} \ f * of\text{-nat} \ (fpxs\text{-root-order} \ f)$
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** $N = fpxs\text{-root-order} \ f$
define f' **where** $f' = fls\text{-of-fpxs} \ f$
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 $*$: $f = fpxs\text{-compose-power} \ (fpxs\text{-of-fls} \ f') \ (1 / rat\text{-of-nat} \ N)$
by (*simp add: fpxs-as-fls N-def f'-def*)
also have $fpxs\text{-nth} \ \dots \ (fpxs\text{-val} \ f) =$
 $fpxs\text{-nth} \ (fpxs\text{-of-fls} \ f') \ (fpxs\text{-val} \ f * rat\text{-of-nat} \ (fpxs\text{-root-order} \ f))$
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 $M\text{-def}$ **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 $\text{fpxs-nth } (f * g) (\text{fpxs-val } f + \text{fpxs-val } g) =$


```

      (∑ (y, z) ∈ {(fpxs-val f, fpxs-val g)}. fpxs-nth f y * fpxs-nth g z)
    unfolding fpxs-nth-mult
    by (intro sum.mono-neutral-left) (auto simp: fpxs-supp-def supp-def)
  thus ?thesis by simp
qed auto

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

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

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

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

proof –

fix $r :: \text{rat}$

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

by (*auto simp: supp-def*)

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

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

qed

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

by *transfer auto*

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

proof (*transfer, goal-cases*)

case ($1\ a\ b$)

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

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

show *?case*

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

qed

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

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

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

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

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

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

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

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

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

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

proof –

```

fix r :: rat and f :: rat ⇒ 'a
assume f: is-fpxs f
have subset: supp (λn. f (n + r)) ⊆ (λn. n + r) -' supp f
  by (auto simp: supp-def)
have eq: (λn. n + r) -' supp f = (λn. n - r) ' supp f
  by (auto simp: image-iff algebra-simps)

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

lemma fpxs-nth-shift [simp]: fpxs-nth (fpxs-shift r f) n = fpxs-nth f (n + r)
  by transfer simp-all

lemma fpxs-shift-0-left [simp]: fpxs-shift 0 f = f
  by transfer auto

lemma fpxs-shift-add-left: fpxs-shift (m + n) f = fpxs-shift m (fpxs-shift n f)
  by transfer (simp-all add: add-ac)

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

```

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

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

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

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

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

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

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

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

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

lemma *fpxs-shift-mult*: $f * fpxs-shift\ r\ g = fpxs-shift\ r\ (f * g)$
 $fpxs-shift\ r\ f * g = fpxs-shift\ r\ (f * g)$
proof –
obtain $a\ b$ **where** $ab: r = of-int\ a / of-nat\ b$ **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** $s = lcm\ b\ (lcm\ (fpxs-root-order\ f)\ (fpxs-root-order\ g))$

```

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  $sr\text{-eq}: r * \text{rat-of-nat } s = \text{rat-of-int } n$ 
  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  $\text{fps-shift-power [simp]: fps-shift } n \ x \ ^m = \text{fps-shift } (\text{of-nat } m * n) \ (x \ ^m)$ 
  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
   $\text{fps-radical } rt \ r \ f = (\text{if } f = 0 \text{ then } 0 \text{ else}$ 
    (let  $f' = \text{fls-base-factor-to-fps } (\text{fls-of-fps } f);$ 
       $f'' = \text{fps-of-fls } (\text{fps-to-fls } (\text{fps-radical } rt \ r \ f'))$ 
      in  $\text{fps-shift } (-\text{fps-val } f / \text{rat-of-nat } r)$ 
       $(\text{fps-compose-power } f'' (1 / \text{rat-of-nat } (\text{fps-root-order } f))))))$ 

```

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*:

$rt\ r\ (fpxs\text{-}nth\ f\ (fpxs\text{-}val\ f))\ \hat{\ }^r = fpxs\text{-}nth\ f\ (fpxs\text{-}val\ f) \implies fpxs\text{-}radical\ rt\ r\ f\ \hat{\ }^r = f$

and *fpxs-radical-lead-coeff*:

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

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* ($-fls\text{-}subdegree\ (fls\text{-}of\text{-}fpxs\ f)$) (*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* ($-fls\text{-}subdegree\ (fls\text{-}of\text{-}fpxs\ f)$) (*fps-to-fls* *f'*)) = *fpxs-shift* ($-(fpxs\text{-}val\ f * \text{rat-of-nat}\ q)$) (*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* ($-(fpxs\text{-}val\ f * \text{rat-of-nat}\ q)$) (*fpxs-of-fls* (*fps-to-fls* *f'*))) (1 / *rat-of-nat* *q*) = *f* .

{

assume *rt*: *rt* *r* (*fpxs-nth* *f* (*fpxs-val* *f*)) $\hat{\ }^r = fpxs\text{-}nth\ f\ (fpxs\text{-}val\ f)$

show *fpxs-radical* *rt* *r* *f* $\hat{\ }^r = f$

proof (*cases* *f* = 0)

case [simp]: *False*

have $f''\ \hat{\ }^r = fpxs\text{-}of\text{-}fls\ (fps\text{-}to\text{-}fls\ (fps\text{-}radical\ rt\ r\ f'\ \hat{\ }^r))$

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

also have *fps-radical* *rt* *r* $f'\ \hat{\ }^r = f'$

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

finally have $f''\ \hat{\ }^r = fpxs\text{-}of\text{-}fls\ (fps\text{-}to\text{-}fls\ f')$.

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

$fpxs\text{-}shift\ (-fpxs\text{-}val\ f)\ (fpxs\text{-}compose\text{-}power\ (f''\ \hat{\ }^r)\ (1 / \text{of-nat}\ q))$

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

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

simp-all

also have $f''\ \hat{\ }^r = fpxs\text{-}of\text{-}fls\ (fps\text{-}to\text{-}fls\ f')$

by *fact*

```

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

assume [simp]:  $f \neq 0$ 
have fpxs-nth (fpxs-shift ( $-fpxs\text{-val } f / \text{of-nat } r$ ) (fpxs-compose-power  $f''$  ( $1 /$ 
of-nat } q)))
  (fpxs-val } f / \text{of-nat } r) = fpxs-nth  $f''$   $0$ 
using  $r$  by (simp add: q-def)
also have fpxs-shift ( $-fpxs\text{-val } f / \text{of-nat } r$ ) (fpxs-compose-power  $f''$  ( $1 / \text{of-nat}$ 
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  $f / \text{rat-of-nat } r$ ) =
   $rt$   $r$  (fpxs-nth  $f$  (fpxs-val  $f$ )) .

qed

lemma fls-base-factor-power:
  fixes  $f :: 'a::\{\text{semiring-1}, \text{semiring-no-zero-divisors}\}$  fls
  shows fls-base-factor ( $f \wedge n$ ) = fls-base-factor  $f \wedge n$ 
proof (cases } f = 0)
  case False
  have [simp]:  $f \wedge n \neq 0$  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* $\neq 0$

define N **where** $N = \text{degree } p$

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 $r = (\text{MAX } i \in \{i \in \{..<N\}. \text{coeff } p \ i \neq 0\}.$

$-\text{fpxs-val } (\text{poly.coeff } p \ i) / (\text{rat-of-nat } N - \text{rat-of-nat } i))$

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 : $b > 0$ $r = \text{of-int } a / \text{of-nat } b \ \forall i \leq N. \text{fpxs-root-order } (\text{coeff } p \ i) \ \text{dvd } b$

proof –

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

define x **where** $x = b \ \text{div } \text{nat } (\text{snd } (\text{quotient-of } r))$

define a **where** $a = \text{fst } (\text{quotient-of } r) * \text{int } x$

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*: $b = \text{nat } (\text{snd } (\text{quotient-of } r)) * x$

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 quotient-of-denom-pos'[of r] 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 fpxs-as-fls'[OF this  $\langle b > 0 \rangle$ ] show ?thesis by metis
qed
then obtain  $c\text{-aux}$  where  $c\text{-aux}$ :
   $\text{coeff } p \ i = \text{fpxs-compose-power (fpxs-of-fls (} c\text{-aux } i)) (1 / \text{rat-of-nat } b)$  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  $c\text{-eq: fpxs-of-fls (} c \ i) = \text{fpxs-compose-power (coeff } p \ i) (\text{rat-of-nat } b)$  for  $i$ 
  using  $c[\text{of } i] \langle b > 0 \rangle$  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  $c[\text{of } N] \langle \text{lead-coeff } p = 1 \rangle \langle b > 0 \rangle$  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 auto
      next
        case False
        hence  $\text{coeff } p i \neq 0$ 
        using  $c\text{-eq}[of i]$  by auto
        hence  $r\text{-ge}$ :  $r \geq -\text{fps-}\text{val } (\text{poly.coeff } p i) / (\text{rat-of-nat } N - \text{rat-of-nat } i)$ 
        unfolding  $r\text{-def}$  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 subde-
gree-c)
        also have  $\text{rat-of-int } \dots =$ 
           $\text{fps-}\text{val } (\text{poly.coeff } p i) * \text{of-nat } b + (\text{of-nat } N - \text{of-nat } i) * \text{of-int } a$ 
        using  $i$  that False by (simp add: subdegree-c)
        also have  $\dots = \text{of-nat } b * (\text{of-nat } N - \text{of-nat } i) *$ 
           $(\text{fps-}\text{val } (\text{poly.coeff } p i) / (\text{of-nat } N - \text{of-nat } i) + r)$ 
        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  $\exists c''$ .  $c' i = \text{fls-shift } 0 (\text{fps-to-fls } c'')$ 
        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  $c''\text{-eq}$ :  $\text{fps-to-fls } (c'' i) = c' i$  for  $i$ 
  using  $c'[of i]$  by simp

```

```

define  $p'$  where  $p' = \text{Abs-poly } c''$ 
have  $\text{coeff-}p'$ :  $\text{coeff } p' = c''$ 
  unfolding  $p'$ -def
proof (rule  $\text{coeff-Abs-poly}$ )
  fix  $i$  assume  $i > N$ 
  hence  $\text{coeff } p \ i = 0$ 
    by (simp add:  $N$ -def  $\text{coeff-eq-0}$ )
  thus  $c'' \ i = 0$  using  $c''[\text{of } i] \ c[\text{of } i] \ \langle b > 0 \rangle$ 
    by (auto simp:  $c'$ -def  $\text{fls-shift-eq0-iff}$ )
qed

```

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

```

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

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

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

```

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

```

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

```

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

We now show that $p'\text{-proj}$ 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* $g\ h$ $p'\text{-proj} = g * h$
proof –
have *degree* $p'\text{-proj} > 1$
using $deg\text{-}p$ **by** (*auto simp: N-def*)

Let x be an arbitrary root of $p'\text{-proj}$:

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

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

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

If x were non-zero, all the coefficients of $p'\text{-proj}$ 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 $coeff\ p'\text{-proj} (N - 1) = 0$ **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 $p'\text{-proj}$ must also be non-zero.

have $0 < N \wedge coeff\ p\ 0 \neq 0$
using $deg\text{-}p$ $\langle coeff\ p\ 0 \neq 0 \rangle$ **by** (*auto simp: N-def*)

hence $\{i \in \{..<N\}. \text{coeff } p \ i \neq 0\} \neq \{\}$
by *blast*
hence $r \in (\lambda i. -\text{fps-val } (\text{poly.coeff } p \ i) / (\text{rat-of-nat } N - \text{rat-of-nat } i)) \text{ `}$
 $\{i \in \{..<N\}. \text{coeff } p \ i \neq 0\}$
unfolding *r-def* **using** *deg-p* **by** (*intro Max-in*) (*auto simp: N-def*)
then obtain *i* **where** $i: i < N \text{ coeff } p \ i \neq 0$
 $-\text{fps-val } (\text{coeff } p \ i) / (\text{rat-of-nat } N - \text{rat-of-nat } i) = r$
by *blast*
hence [*simp*]: $c' \ i \neq 0$
using $i \ c[\text{of } i]$ **by** (*auto simp: c'-def*)
have $\text{fps-val } (\text{poly.coeff } p \ i) = \text{rat-of-int } (\text{fls-subdegree } (c \ i)) / \text{rat-of-nat } b$
using $\text{subdegree-c}[\text{of } i] \ i \ \langle b > 0 \rangle$ **by** (*simp add: field-simps*)
also have $\text{fps-val } (\text{coeff } p \ i) = -r * (\text{rat-of-nat } N - \text{rat-of-nat } i)$
using *i* **by** (*simp add: field-simps*)
finally have $\text{rat-of-int } (\text{fls-subdegree } (c \ i)) = -r * (\text{of-nat } N - \text{of-nat } i) *$
of-nat b
using $\langle b > 0 \rangle$ **by** (*simp add: field-simps*)
also have $c \ i = \text{fls-shift } ((\text{int } N - \text{int } i) * a) (c' \ i)$
using *i* **by** (*simp add: c'-def ring-distrib fls-X-intpow-times-conv-shift*
flip: fls-shifted-times-simps(2))
also have $\text{fls-subdegree } \dots = \text{fls-subdegree } (c' \ i) - (\text{int } N - \text{int } i) * a$
by (*subst fls-shift-subdegree*) *auto*
finally have $\text{fls-subdegree } (c' \ i) = 0$
using $\langle b > 0 \rangle$ **by** (*simp add: ab(2)*)
hence $\text{subdegree } (\text{coeff } p' \ i) = 0$
by (*simp flip: c''-eq add: fls-subdegree-fls-to-fps coeff-p'*)
moreover have $\text{coeff } p' \ i \neq 0$
using $\langle c' \ i \neq 0 \rangle \ c' \ \text{coeff-p}'$ **by** *auto*
ultimately have $\text{coeff } p' \ i \ \$ \ 0 \neq 0$
using *subdegree-eq-0-iff* **by** *blast*

also have $\text{coeff } p' \ i \ \$ \ 0 = \text{coeff } p' \text{-proj } i$
by (*simp add: p'-proj-def reduce-fps-poly-def*)
also have $\dots = 0$
by (*subst *, subst coeff-linear-poly-power*) (*use i \ \langle x = 0 \rangle in auto*)
finally show *False* **by** *simp*

qed

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

have $\exists y. x \neq y \wedge \text{poly } p' \text{-proj } y = 0$
proof (*rule ccontr*)
assume $*$: $\neg(\exists y. x \neq y \wedge \text{poly } p' \text{-proj } y = 0)$
have $p' \text{-proj } \neq 0$ **by** *simp*
then obtain *A* **where** $A: \text{size } A = \text{degree } p' \text{-proj}$
 $p' \text{-proj} = \text{smult } (\text{lead-coeff } p' \text{-proj}) (\prod_{x \in \#A. [-x, 1:])$
using *alg-closed-imp-factorization[of p'-proj]* **by** *blast*
have $\text{set-mset } A = \{x. \text{poly } p' \text{-proj } x = 0\}$
using *lc-p'-proj* **by** (*subst A*) (*auto simp: poly-prod-mset*)
also have $\dots = \{x\}$

```

    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 *fps* :: (zero) norm
begin

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

instance ..

end

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

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

instance ..

end

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

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

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

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.