Formal Puiseux Series

Manuel Eberl

March 17, 2025

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

1.1 Facts about polynomials

theory *Puiseux-Polynomial-Library*
imports *HOL−Computational-Algebra.Computational-Algebra Polynomial-Interpolation.Ring-Hom-Poly*
begin

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

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

lemma *inj-idom-hom-pcompose* [*intro*]:
assumes [*simp*]: *degree* $(p :: 'a :: idom\ poly) \neq 0$
shows *inj-idom-hom* $(\lambda q.\ pcompose\ q\ p)$
⟨*proof*⟩

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*: $n > 0 \Longrightarrow f\ n \neq 0 \Longrightarrow \exists x.\ (\sum k{\leq}n.\ f\ k * x\ \hat{}\ k) = 0$

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

lemma *alg-closed-imp-poly-has-root*:
assumes *degree* $(p :: 'a :: alg\text{-}closed\text{-}field\ poly) > 0$
shows $\exists x.\ poly\ p\ x = 0$
⟨*proof*⟩

lemma *alg-closedI* [*Pure.intro*]:
assumes $\bigwedge p :: 'a\ poly.\ degree\ p > 0 \Longrightarrow lead\text{-}coeff\ p = 1 \Longrightarrow \exists x.\ poly\ p\ x = 0$
shows *OFCLASS*$('a :: field,\ alg\text{-}closed\text{-}field\text{-}class)$
⟨*proof*⟩

We can now prove by induction that every polynomial of degree n splits into a product of n linear factors:

lemma *alg-closed-imp-factorization*:
fixes $p :: 'a :: alg\text{-}closed\text{-}field\ poly$
assumes $p \neq 0$
shows $\exists A.\ size\ A = degree\ p \wedge p = smult\ (lead\text{-}coeff\ p)\ (\prod x{\in}\#A.\ [:-x,\ 1:])$
⟨*proof*⟩

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

lemma *alg-closed-imp-reducible*:
assumes *degree* (*p* :: *'a* :: *alg-closed-field poly*) > *1*
shows ¬*irreducible p*
⟨*proof*⟩

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

lemma *alg-closedI-reducible*:
assumes ⋀*p* :: *'a poly*. *degree p* > *1* ⟹ *lead-coeff p* = *1* ⟹ *coeff p 0* ≠ *0* ⟹ ¬*irreducible p*
shows *OFCLASS*(*'a* :: *field*, *alg-closed-field-class*)
⟨*proof*⟩

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 ⋀*p* :: *'a poly*. *degree p* > *1* ⟹ *lead-coeff p* = *1* ⟹ *coeff p* (*degree p* − *1*) = *0* ⟹ *coeff p 0* ≠ *0* ⟹ ¬*irreducible p*
shows *OFCLASS*(*'a* :: *field-char-0*, *alg-closed-field-class*)
⟨*proof*⟩

As a consequence of the full factorisation lemma proven above, we can also show that any polynomial with at least two different roots splits into two non-constant coprime factors:

lemma *alg-closed-imp-poly-splits-coprime*:
assumes *degree* (*p* :: *'a* :: {*alg-closed-field*} *poly*) > *1*
assumes *poly p x* = *0 poly p y* = *0 x* ≠ *y*
obtains *r s* **where** *degree r* > *0 degree s* > *0 coprime r s p* = *r* ∗ *s*
⟨*proof*⟩

instance *complex* :: *alg-closed-field*
⟨*proof*⟩

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* (λ*m*. *Abs-poly* (λ*n*. *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*⟩

definition *fps-poly-swap2* :: *'a* :: *zero poly fps* ⇒ *'a fps poly* **where**
fps-poly-swap2 p = *Abs-poly* (λ*m*. *Abs-fps* (λ*n*. *coeff* (*fps-nth p n*) *m*))

lemma *fps-nth-coeff-fps-poly-swap2*:
assumes ⋀*n*. *degree* (*fps-nth p n*) ≤ *d*
shows *fps-nth* (*coeff* (*fps-poly-swap2 p*) *m*) *n* = *coeff* (*fps-nth p n*) *m*
⟨*proof*⟩

lemma *degree-fps-poly-swap2-le*:
assumes ⋀*n*. *degree* (*fps-nth p n*) ≤ *d*
shows *degree* (*fps-poly-swap2 p*) ≤ *d*
⟨*proof*⟩

lemma *degree-fps-poly-swap2-eq*:
assumes ⋀*n*. *degree* (*fps-nth p n*) ≤ *d*
assumes *d* > *0* ∨ *fps-nth p n* ≠ *0*
assumes *degree* (*fps-nth p n*) = *d*
shows *degree* (*fps-poly-swap2 p*) = *d*
⟨*proof*⟩

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*⟩

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)*
⟨*proof*⟩

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

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

lemma *hensel-fpxs1-0* [*simp*]: *hensel-fpxs1 $ 0 = g*
⟨*proof*⟩

lemma *hensel-fpxs2-0* [*simp*]: *hensel-fpxs2 $ 0 = h*
⟨*proof*⟩

theorem *fps-hensel-aux*:
defines *f ≡ fps-nth F 0*
assumes *f = g ∗ h*
assumes *∀ n>0. degree (fps-nth F n) < degree f*
defines *G ≡ hensel-fpxs1* **and** *H ≡ hensel-fpxs2*
shows *F = G ∗ H fps-nth G 0 = g fps-nth H 0 = h*
∀ n>0. degree (fps-nth G n) < degree g
∀ n>0. degree (fps-nth H n) < degree h
⟨*proof*⟩

end

end

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

definition *F′* **where** *F′ = fps-poly-swap1 F*

sublocale *fps-hensel-aux F' g h*
⟨*proof*⟩

definition *G* **where**
G = fps-poly-swap2 hensel-fpxs1

definition *H* **where**
H = fps-poly-swap2 hensel-fpxs2

lemma *deg-f*: *degree f = degree F*
⟨*proof*⟩

lemma
F-splits: *F = G * H* **and**
reduce-G: *reduce-fps-poly G = g* **and**
reduce-H: *reduce-fps-poly H = h* **and**
deg-G: *degree G = degree g* **and**
deg-H: *degree H = degree h* **and**
lead-coeff-G: *lead-coeff G = fps-const (lead-coeff g)* **and**
lead-coeff-H: *lead-coeff H = fps-const (lead-coeff h)*
⟨*proof*⟩

end

end

3 Formal Puiseux Series

theory *Formal-Puiseux-Series*
imports *FPS-Hensel*
begin

3.1 Auxiliary facts and definitions

lemma *div-dvd-self*:
fixes *a b* :: *'a* :: {*semidom-divide*}
shows *b dvd a* ⟹ *a div b dvd a*
⟨*proof*⟩

lemma *quotient-of-int* [*simp*]: *quotient-of* (*of-int n*) = (*n*, *1*)
⟨*proof*⟩

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

lemma *rat-eq-quotientD*:
assumes *r = rat-of-int a* / *rat-of-int b b* ≠ *0*

shows *fst (quotient-of r) dvd a snd (quotient-of r) dvd b*
⟨*proof*⟩

lemma *quotient-of-denom-add-dvd*:
*snd (quotient-of (x + y)) dvd snd (quotient-of x) * snd (quotient-of y)*
⟨*proof*⟩

lemma *quotient-of-denom-diff-dvd*:
*snd (quotient-of (x − y)) dvd snd (quotient-of x) * snd (quotient-of y)*
⟨*proof*⟩

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}*
⟨*proof*⟩

lemma *supp-uminus* [*simp*]: *supp (λx. −f x :: 'a :: group-add) = supp f*
⟨*proof*⟩

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 paramter 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 ⇒ 'a :: zero) ⇒ bool* **where**
is-fpxs f ⟷ bdd-below (supp f) ∧ (LCM r∈supp f. snd (quotient-of r)) ≠ 0

typedef (**overloaded**) *'a fpxs* = {*f*::*rat* ⇒ *'a* :: *zero*. *is-fpxs f*}
morphisms *fpxs-nth Abs-fpxs*
⟨*proof*⟩

setup-lifting *type-definition-fpxs*

lemma *fpxs-ext*: (⋀*r*. *fpxs-nth f r* = *fpxs-nth g r*) ⟹ *f* = *g*
⟨*proof*⟩

lemma *fpxs-eq-iff*: *f* = *g* ⟷ (∀ *r*. *fpxs-nth f r* = *fpxs-nth g r*)
⟨*proof*⟩

lift-definition *fpxs-supp* :: *'a* :: *zero fpxs* ⇒ *rat set* **is** *supp* ⟨*proof*⟩

lemma *fpxs-supp-altdef*: *fpxs-supp f* = {*x*. *fpxs-nth f x* ≠ *0*}
⟨*proof*⟩

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

lift-definition *fpxs-root-order* :: *'a* :: *zero fpxs* ⇒ *nat* **is**
λf. *nat* (*LCM r*∈*supp f*. *snd* (*quotient-of r*)) ⟨*proof*⟩

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

lemma *fpxs-root-order-nonzero* [*simp*]: *fpxs-root-order f* ≠ *0*
⟨*proof*⟩

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 $X\hat{}\{1/d\}$. The following operation gives us this Laurent series.

lift-definition *fls-of-fpxs* :: *'a* :: *zero fpxs* ⇒ *'a fls* **is**
λf n. *f* (*of-int n* / *of-int* (*LCM r*∈*supp f*. *snd* (*quotient-of r*)))
⟨*proof*⟩

lemma *fls-nth-of-fpxs*:
fls-nth (*fls-of-fpxs f*) *n* = *fpxs-nth f* (*of-int n* / *of-nat* (*fpxs-root-order f*))
⟨*proof*⟩

3.3 Basic algebraic typeclass instances

instantiation *fpxs* :: (*zero*) *zero*
begin

lift-definition *zero-fpxs* :: *'a fpxs* **is** *λr*::*rat*. *0* :: *'a*
⟨*proof*⟩

instance ⟨*proof*⟩

end

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

lift-definition *one-fpxs* :: *'a fpxs* **is** *λr::rat. if r = 0 then 1 else 0* :: *'a*
⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *fls-of-fpxs-0* [*simp*]: *fls-of-fpxs 0 = 0*
⟨*proof*⟩

lemma *fpxs-nth-0* [*simp*]: *fpxs-nth 0 r = 0*
⟨*proof*⟩

lemma *fpxs-nth-1*: *fpxs-nth 1 r = (if r = 0 then 1 else 0)*
⟨*proof*⟩

lemma *fpxs-nth-1'*: *fpxs-nth 1 0 = 1 r ≠ 0 ⟹ fpxs-nth 1 r = 0*
⟨*proof*⟩

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

lift-definition *plus-fpxs* :: *'a fpxs ⇒ 'a fpxs ⇒ 'a fpxs* **is**
λf g x. f x + g x
⟨*proof*⟩

instance
⟨*proof*⟩

end

instance *fpxs* :: (*comm-monoid-add*) *comm-monoid-add*
⟨*proof*⟩

lemma *fpxs-nth-add* [*simp*]: *fpxs-nth (f + g) r = fpxs-nth f r + fpxs-nth g r*
⟨*proof*⟩

lift-definition *fpxs-of-fls* :: *'a :: zero fls ⇒ 'a fpxs* **is**
λf r. if r ∈ ℤ then f ⌊r⌋ else 0
⟨*proof*⟩

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

lift-definition *uminus-fpxs* :: *'a fpxs* ⇒ *'a fpxs* **is** λ*f x*. −*f x*
⟨*proof*⟩

definition *minus-fpxs* :: *'a fpxs* ⇒ *'a fpxs* ⇒ *'a fpxs* **where**
minus-fpxs f g = *f* + (−*g*)

instance ⟨*proof*⟩

end

lemma *fpxs-nth-uminus* [*simp*]: *fpxs-nth* (−*f*) *r* = −*fpxs-nth f r*
⟨*proof*⟩

lemma *fpxs-nth-minus* [*simp*]: *fpxs-nth* (*f* − *g*) *r* = *fpxs-nth f r* − *fpxs-nth g r*
⟨*proof*⟩

lemma *fpxs-of-fls-eq-iff* [*simp*]: *fpxs-of-fls f* = *fpxs-of-fls g* ⟷ *f* = *g*
⟨*proof*⟩

lemma *fpxs-of-fls-0* [*simp*]: *fpxs-of-fls 0* = *0*
⟨*proof*⟩

lemma *fpxs-of-fls-1* [*simp*]: *fpxs-of-fls 1* = *1*
⟨*proof*⟩

lemma *fpxs-of-fls-add* [*simp*]: *fpxs-of-fls* (*f* + *g*) = *fpxs-of-fls f* + *fpxs-of-fls g*
⟨*proof*⟩

lemma *fps-to-fls-sum* [*simp*]: *fps-to-fls* (*sum f A*) = (∑ *x*∈*A*. *fps-to-fls* (*f x*))
⟨*proof*⟩

lemma *fpxs-of-fls-sum* [*simp*]: *fpxs-of-fls* (*sum f A*) = (∑ *x*∈*A*. *fpxs-of-fls* (*f x*))
⟨*proof*⟩

lemma *fpxs-nth-of-fls*:
fpxs-nth (*fpxs-of-fls f*) *r* = (*if r* ∈ ℤ *then fls-nth f* ⌊*r*⌋ *else 0*)
⟨*proof*⟩

lemma *fpxs-of-fls-eq-0-iff* [*simp*]: *fpxs-of-fls f* = *0* ⟷ *f* = *0*
⟨*proof*⟩

lemma *fpxs-of-fls-eq-1-iff* [*simp*]: *fpxs-of-fls f* = *1* ⟷ *f* = *1*
⟨*proof*⟩

lemma *fpxs-root-order-of-fls* [*simp*]: *fpxs-root-order* (*fpxs-of-fls f*) = *1*
⟨*proof*⟩

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* ⇒ *rat* ⇒ *'a fpxs* **is**
λf r x. if r > 0 then f (x / r) else 0
⟨*proof*⟩

lemma *fpxs-as-fls*:
fpxs-compose-power (*fpxs-of-fls* (*fls-of-fpxs f*)) (*1 / of-nat* (*fpxs-root-order f*)) = *f*
⟨*proof*⟩

lemma *fpxs-compose-power-0* [*simp*]: *fpxs-compose-power 0 r = 0*
⟨*proof*⟩

lemma *fpxs-compose-power-1* [*simp*]: *r > 0 ⟹ fpxs-compose-power 1 r = 1*
⟨*proof*⟩

lemma *fls-of-fpxs-eq-0-iff* [*simp*]: *fls-of-fpxs x = 0 ⟷ x = 0*
⟨*proof*⟩

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*⟩

lemma *fpxs-compose-power-add* [*simp*]:
fpxs-compose-power (*f + g*) *r* = *fpxs-compose-power f r* + *fpxs-compose-power g r*
⟨*proof*⟩

lemma *fpxs-compose-power-distrib*:
r1 > 0 ∨ r2 > 0 ⟹
fpxs-compose-power (*fpxs-compose-power f r1*) *r2* = *fpxs-compose-power f* (*r1 * r2*)
⟨*proof*⟩

lemma *fpxs-compose-power-divide-right*:
r1 > 0 ⟹ r2 > 0 ⟹
fpxs-compose-power f (*r1 / r2*) = *fpxs-compose-power* (*fpxs-compose-power f r1*) (*inverse r2*)
⟨*proof*⟩

lemma *fpxs-compose-power-1-right* [*simp*]: *fpxs-compose-power f 1 = f*
⟨*proof*⟩

lemma *fpxs-compose-power-eq-iff* [*simp*]:
assumes *r > 0*
shows *fpxs-compose-power f r = fpxs-compose-power g r ⟷ f = g*

⟨*proof*⟩

lemma *fpxs-compose-power-eq-1-iff* [*simp*]:
assumes $l > 0$
shows *fpxs-compose-power* $p\ l = 1 \longleftrightarrow p = 1$
⟨*proof*⟩

lemma *fpxs-compose-power-eq-0-iff* [*simp*]:
assumes $r > 0$
shows *fpxs-compose-power* $f\ r = 0 \longleftrightarrow f = 0$
⟨*proof*⟩

lemma *fls-of-fpxs-of-fls* [*simp*]: *fls-of-fpxs* (*fpxs-of-fls* f) $= f$
⟨*proof*⟩

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*⟩

3.5 Mutiplication and ring properties

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

lift-definition *times-fpxs* :: $'a$ *fpxs* $\Rightarrow$ $'a$ *fpxs* $\Rightarrow$ $'a$ *fpxs* **is**
$\lambda f\ g\ x.\ (\sum (y,z) \mid y \in supp\ f \wedge z \in supp\ g \wedge x = y + z.\ f\ y * g\ z)$
⟨*proof*⟩

lemma *fpxs-nth-mult*:
fpxs-nth $(f * g)\ r =$
$(\sum (y,z) \mid y \in$ *fpxs-supp* $f \wedge z \in$ *fpxs-supp* $g \wedge r = y + z.$ *fpxs-nth* $f\ y *$ *fpxs-nth* $g\ z)$
⟨*proof*⟩

lemma *fpxs-compose-power-mult* [*simp*]:
fpxs-compose-power $(f * g)\ r =$ *fpxs-compose-power* $f\ r *$ *fpxs-compose-power* $g\ r$
⟨*proof*⟩

lemma *fpxs-supp-of-fls*: *fpxs-supp* (*fpxs-of-fls* f) $=$ *of-int* ‘ *supp* (*fls-nth* f)
⟨*proof*⟩

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

instance ⟨*proof*⟩

end

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

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

lemma *fpxs-of-fls-power* [*simp*]: *fpxs-of-fls* (*f* ^ *n*) = *fpxs-of-fls f* ^ *n*
⟨*proof*⟩

lemma *fpxs-compose-power-power* [*simp*]:
r > *0* ⟹ *fpxs-compose-power* (*f* ^ *n*) *r* = *fpxs-compose-power f r* ^ *n*
⟨*proof*⟩

3.6 Constant Puiseux series and the series X

lift-definition *fpxs-const* :: *'a* :: *zero* ⇒ *'a fpxs* **is**
λ*c n*. *if n* = *0 then c else 0*
⟨*proof*⟩

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

lemma *fpxs-const-1* [*simp*]: *fpxs-const 1* = *1*
⟨*proof*⟩

lemma *fpxs-of-fls-const* [*simp*]: *fpxs-of-fls* (*fls-const c*) = *fpxs-const c*
⟨*proof*⟩

lemma *fls-of-fpxs-const* [*simp*]: *fls-of-fpxs* (*fpxs-const c*) = *fls-const c*
⟨*proof*⟩

lemma *fls-of-fpxs-1* [*simp*]: *fls-of-fpxs 1* = *1*
⟨*proof*⟩

lift-definition *fpxs-X* :: *'a* :: {*one*, *zero*} *fpxs* **is**
λ*x*. *if x* = *1 then* (*1*::*'a*) *else 0*
⟨*proof*⟩

lemma *fpxs-const-altdef*: *fpxs-const x* = *fpxs-of-fls* (*fls-const x*)
⟨*proof*⟩

lemma *fpxs-const-add* [*simp*]: *fpxs-const* (*x* + *y*) = *fpxs-const x* + *fpxs-const y*
⟨*proof*⟩

lemma *fpxs-const-mult* [*simp*]:
fixes *x y* :: *'a*::{*comm-semiring-1*}
shows *fpxs-const* (*x* ∗ *y*) = *fpxs-const x* ∗ *fpxs-const y*
⟨*proof*⟩

lemma *fpxs-const-eq-iff* [*simp*]:
fpxs-const x = *fpxs-const y* ⟷ *x* = *y*
⟨*proof*⟩

lemma *of-nat-fpxs-eq*: *of-nat n* = *fpxs-const* (*of-nat n*)
⟨*proof*⟩

lemma *fpxs-const-uminus* [*simp*]: *fpxs-const* (−*x*) = −*fpxs-const x*
⟨*proof*⟩

lemma *fpxs-const-diff* [*simp*]: *fpxs-const* (*x* − *y*) = *fpxs-const x* − *fpxs-const y*
⟨*proof*⟩

lemma *of-int-fpxs-eq*: *of-int n* = *fpxs-const* (*of-int n*)
⟨*proof*⟩

3.7 More algebraic typeclass instances

instance *fpxs* :: ({*comm-semiring-1*,*semiring-char-0*}) *semiring-char-0*
⟨*proof*⟩

instance *fpxs* :: ({*comm-ring-1*,*ring-char-0*}) *ring-char-0* ⟨*proof*⟩

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

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*⟩

end

instance *fpxs* :: (*field-char-0*) *field-char-0* ⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *fpxs-nth-compose-power* [*simp*]:
assumes $r > 0$
shows *fpxs-nth* (*fpxs-compose-power f r*) n = *fpxs-nth f* (n / r)
⟨*proof*⟩

lemma *fls-of-fpxs-uminus* [*simp*]: *fls-of-fpxs* ($-f$) = $-$*fls-of-fpxs f*
⟨*proof*⟩

lemma *fpxs-root-order-uminus* [*simp*]: *fpxs-root-order* ($-f$) = *fpxs-root-order f*
⟨*proof*⟩

lemma *fpxs-val-uminus* [*simp*]: *fpxs-val* ($-f$) = *fpxs-val f*
⟨*proof*⟩

lemma *fpxs-val-minus-commute*: *fpxs-val* ($f - g$) = *fpxs-val* ($g - f$)
⟨*proof*⟩

lemma *fpxs-val-const* [*simp*]: *fpxs-val* (*fpxs-const c*) = 0
⟨*proof*⟩

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

lemma *of-int-fls-subdegree-of-fpxs*:
rat-of-int (*fls-subdegree* (*fls-of-fpxs f*)) = *fpxs-val f* $*$ *of-nat* (*fpxs-root-order f*)
⟨*proof*⟩

lemma *fpxs-nth-val-nonzero*:
assumes $f \neq 0$
shows *fpxs-nth f* (*fpxs-val f*) $\neq$ 0
⟨*proof*⟩

lemma *fpxs-nth-below-val*:
assumes n: $n <$ *fpxs-val f*
shows *fpxs-nth f n* = 0
⟨*proof*⟩

lemma *fpxs-val-leI*: *fpxs-nth f r* $\neq$ 0 $\Longrightarrow$ *fpxs-val f* $\leq r$
⟨*proof*⟩

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

lemma *fpxs-val-geI*:
assumes $f \neq 0$ $\bigwedge r.\ r < r' \Longrightarrow$ *fpxs-nth f r* = 0
shows *fpxs-val f* $\geq r'$
⟨*proof*⟩

lemma *fpxs-val-compose-power* [*simp*]:
assumes $r > 0$
shows *fpxs-val* (*fpxs-compose-power f r*) = *fpxs-val f* ∗ *r*
⟨*proof*⟩

lemma *fpxs-val-add-ge*:
assumes $f + g \neq 0$
shows *fpxs-val* $(f + g) \geq$ *min* (*fpxs-val f*) (*fpxs-val g*)
⟨*proof*⟩

lemma *fpxs-val-diff-ge*:
assumes $f \neq g$
shows *fpxs-val* $(f - g) \geq$ *min* (*fpxs-val f*) (*fpxs-val g*)
⟨*proof*⟩

lemma *fpxs-nth-mult-val*:
fpxs-nth (*f* ∗ *g*) (*fpxs-val f* + *fpxs-val g*) = *fpxs-nth f* (*fpxs-val f*) ∗ *fpxs-nth g* (*fpxs-val g*)
⟨*proof*⟩

lemma *fpxs-val-mult* [*simp*]:
fixes *f g* :: *'a* :: {*comm-semiring-1*, *semiring-no-zero-divisors*} *fpxs*
assumes $f \neq 0$ $g \neq 0$
shows *fpxs-val* (*f* ∗ *g*) = *fpxs-val f* + *fpxs-val g*
⟨*proof*⟩

lemma *fpxs-val-power* [*simp*]:
fixes *f* :: *'a* :: {*comm-semiring-1*, *semiring-no-zero-divisors*} *fpxs*
assumes $f \neq 0 \vee n > 0$
shows *fpxs-val* (*f* ^ *n*) = *of-nat n* ∗ *fpxs-val f*
⟨*proof*⟩

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*⟩

3.9 Powers of X and shifting

lift-definition *fpxs-X-power* :: *rat* ⇒ *'a* :: {*zero*, *one*} *fpxs* **is**
λr n :: *rat*. *if n* = *r then 1 else* (*0* :: *'a*)
⟨*proof*⟩

lemma *fpxs-X-power-0* [*simp*]: *fpxs-X-power 0* = *1*
⟨*proof*⟩

lemma *fpxs-X-power-add*: *fpxs-X-power* (*a* + *b*) = *fpxs-X-power a* ∗ *fpxs-X-power b*
⟨*proof*⟩

lemma *fpxs-X-power-mult*: *fpxs-X-power* (*rat-of-nat n* ∗ *m*) = *fpxs-X-power m* ^ *n*
⟨*proof*⟩

lemma *fpxs-of-fls-X-power* [*simp*]: *fpxs-of-fls* (*fls-shift n 1*) = *fpxs-X-power* (−*rat-of-int n*)
⟨*proof*⟩

lemma *fpxs-X-power-neq-0* [*simp*]: *fpxs-X-power r* ≠ (*0* :: *'a* :: *zero-neq-one fpxs*)
⟨*proof*⟩

lemma *fpxs-X-power-eq-1-iff* [*simp*]: *fpxs-X-power r* = (*1* :: *'a* :: *zero-neq-one fpxs*) ⟷ *r* = *0*
⟨*proof*⟩

lift-definition *fpxs-shift* :: *rat* ⇒ *'a* :: *zero fpxs* ⇒ *'a fpxs* **is**
λ*r f n*. *f* (*n* + *r*)
⟨*proof*⟩

lemma *fpxs-nth-shift* [*simp*]: *fpxs-nth* (*fpxs-shift r f*) *n* = *fpxs-nth f* (*n* + *r*)
⟨*proof*⟩

lemma *fpxs-shift-0-left* [*simp*]: *fpxs-shift 0 f* = *f*
⟨*proof*⟩

lemma *fpxs-shift-add-left*: *fpxs-shift* (*m* + *n*) *f* = *fpxs-shift m* (*fpxs-shift n f*)
⟨*proof*⟩

lemma *fpxs-shift-diff-left*: *fpxs-shift* (*m* − *n*) *f* = *fpxs-shift m* (*fpxs-shift* (−*n*) *f*)
⟨*proof*⟩

lemma *fpxs-shift-0* [*simp*]: *fpxs-shift r 0* = *0*
⟨*proof*⟩

lemma *fpxs-shift-add* [*simp*]: *fpxs-shift r* (*f* + *g*) = *fpxs-shift r f* + *fpxs-shift r g*
⟨*proof*⟩

lemma *fpxs-shift-uminus* [*simp*]: *fpxs-shift r* (−*f*) = −*fpxs-shift r f*
⟨*proof*⟩

lemma *fpxs-shift-shift-uminus* [*simp*]: *fpxs-shift r* (*fpxs-shift* (−*r*) *f*) = *f*
⟨*proof*⟩

lemma *fpxs-shift-shift-uminus'* [*simp*]: *fpxs-shift* (−*r*) (*fpxs-shift r f*) = *f*
⟨*proof*⟩

lemma *fpxs-shift-diff* [*simp*]: *fpxs-shift r* (*f* − *g*) = *fpxs-shift r f* − *fpxs-shift r g*

⟨*proof*⟩

lemma *fpxs-shift-compose-power* [*simp*]:
fpxs-shift r (*fpxs-compose-power f s*) = *fpxs-compose-power* (*fpxs-shift* (*r* / *s*) *f*) *s*
⟨*proof*⟩

lemma *rat-of-int-div-dvd*: *d dvd n* ⟹ *rat-of-int* (*n div d*) = *rat-of-int n* / *rat-of-int d*
⟨*proof*⟩

lemma *fpxs-of-fls-shift* [*simp*]:
fpxs-of-fls (*fls-shift n f*) = *fpxs-shift* (*of-int n*) (*fpxs-of-fls f*)
⟨*proof*⟩

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*⟩

lemma *fpxs-shift-1*: *fpxs-shift r 1* = *fpxs-X-power* (−*r*)
⟨*proof*⟩

lemma *fpxs-X-power-conv-shift*: *fpxs-X-power r* = *fpxs-shift* (−*r*) *1*
⟨*proof*⟩

lemma *fpxs-shift-power* [*simp*]: *fpxs-shift n x* ^ *m* = *fpxs-shift* (*of-nat m* ∗ *n*) (*x* ^ *m*)
⟨*proof*⟩

lemma *fpxs-compose-power-X-power* [*simp*]:
s > *0* ⟹ *fpxs-compose-power* (*fpxs-X-power r*) *s* = *fpxs-X-power* (*r* ∗ *s*)
⟨*proof*⟩

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 theorems that is missing here, e.g. the uniqueness (which could probably be lifted over from FPSs) somehow.

definition *fpxs-radical* :: (*nat* ⇒ *'a* :: *field-char-0* ⇒ *'a*) ⇒ *nat* ⇒ *'a fpxs* ⇒ *'a fpxs* **where**
fpxs-radical rt r f = (*if f* = *0 then 0 else*
(*let f'* = *fls-base-factor-to-fps* (*fls-of-fpxs f*);
f'' = *fpxs-of-fls* (*fps-to-fls* (*fps-radical rt r f'*))
in fpxs-shift (−*fpxs-val f* / *rat-of-nat r*)
(*fpxs-compose-power f''* (*1* / *rat-of-nat* (*fpxs-root-order f*)))))

lemma *fpxs-radical-0* [*simp*]: *fpxs-radical rt r 0* = *0*

⟨*proof*⟩

lemma
fixes *r* :: *nat*
assumes *r*: *r* > *0*
shows *fpxs-power-radical*:
rt r (*fpxs-nth f* (*fpxs-val f*)) ^ *r* = *fpxs-nth f* (*fpxs-val f*) ⟹ *fpxs-radical rt r f* ^ *r* = *f*
and *fpxs-radical-lead-coeff*:
f ≠ *0* ⟹ *fpxs-nth* (*fpxs-radical rt r f*) (*fpxs-val f* / *rat-of-nat r*) = *rt r* (*fpxs-nth f* (*fpxs-val f*))
⟨*proof*⟩

lemma *fls-base-factor-power*:
fixes *f* :: *'a*::{*semiring-1*, *semiring-no-zero-divisors*} *fls*
shows *fls-base-factor* (*f* ^ *n*) = *fls-base-factor f* ^ *n*
⟨*proof*⟩

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

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

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*⟩

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

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

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

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

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

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

3.12 Metric and topology

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

instantiation *fpxs* :: (*zero*) *norm*
begin

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

instance ⟨*proof*⟩

end

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

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

instance ⟨*proof*⟩

end

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

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

definition *open-fpxs-def* [*code del*]:
open (*U* :: '*a fpxs set*) ⟷ (∀ *x*∈*U*. *eventually* (λ(*x'*, *y*). *x'* = *x* ⟶ *y* ∈ *U*) *uniformity*)

instance ⟨*proof*⟩

end

instance *fpxs* :: (*group-add*) *dist-norm*
⟨*proof*⟩

lemma *fpxs-const-eq-0-iff* [*simp*]: *fpxs-const* *x* = *0* ⟷ *x* = *0*
⟨*proof*⟩

lemma *semiring-char-fpxs* [*simp*]: *CHAR*('*a* :: *comm-semiring-1 fpxs*) = *CHAR*('*a*)
⟨*proof*⟩

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

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.