Perfect Fields

Manuel Eberl, Katharina Kreuzer

November 13, 2023

Abstract

This entry provides a type class for *perfect fields*. A perfect field K can be characterized by one of the following equivalent conditions [2]:

1. Any irreducible polynomial p is separable, i.e. $\gcd(p, p') = 1$, or, equivalently, $p' \neq 0$.
2. Either $\text{char}(K) = 0$ or $\text{char}(K) = p > 0$ and the Frobenius endomorphism $x \mapsto x^p$ is surjective (i.e. every element of K has a p-th root).

We define perfect fields using the second characterization and show the equivalence to the first characterization. The implication "2 $\Rightarrow$ 1" is relatively straightforward using the injectivity of the Frobenius homomorphism.

Examples for perfect fields are [2]:

- any field of characteristic 0 (e.g. $\mathbb{R}$ and $\mathbb{C}$)
- any finite field (i.e. $\mathbb{F}_q$ for $q = p^n$, $n > 0$ and p prime)
- any algebraically closed field (for example the formal Puiseux series over finite fields)

Contents

theory *Perfect_Field_Library*
imports
"HOL-Computational_Algebra.Computational_Algebra"
"Berlekamp_Zassenhaus.Finite_Field"
begin

instance *bool :: prime_card*
⟨*proof*⟩

theorem (**in** *comm_semiring_1*) *binomial_ring:*
*"(a + b :: 'a)^n = (∑k≤n. (of_nat (n choose k)) * a^k * b^(n-k))"*
⟨*proof*⟩

lemma *prime_not_dvd_fact:*
assumes *kn: "k < n"* **and** *prime_n: "prime n"*
shows *"¬ n dvd fact k"*
⟨*proof*⟩

lemma *dvd_choose_prime:*
assumes *kn: "k < n"* **and** *k: "k ≠ 0"* **and** *n: "n ≠ 0"* **and** *prime_n: "prime n"*
shows *"n dvd (n choose k)"*
⟨*proof*⟩

lemma *CHAR_not_1 [simp]: "CHAR('a :: {semiring_1, zero_neq_one}) ≠ Suc 0"*
⟨*proof*⟩

lemma (**in** *idom*) *CHAR_not_1' [simp]: "CHAR('a) ≠ Suc 0"*
⟨*proof*⟩

lemma *semiring_char_mod_ring [simp]:*
"CHAR('n :: nontriv mod_ring) = CARD('n)"
⟨*proof*⟩

lemma *of_nat_eq_iff_cong_CHAR:*
"of_nat x = (of_nat y :: 'a :: semiring_1_cancel) ⟷ [x = y] (mod CHAR('a))"
⟨*proof*⟩

lemma (**in** *ring_1*) *of_int_eq_0_iff_char_dvd:*
"(of_int n = (0 :: 'a)) = (int CHAR('a) dvd n)"
⟨*proof*⟩

lemma (**in** *ring_1*) *of_int_eq_iff_cong_CHAR:*
  "of_int x = (of_int y :: 'a) ⟷ [x = y] (mod int CHAR('a))"
⟨*proof*⟩

lemma *finite_imp_CHAR_pos:*
  assumes *"finite (UNIV :: 'a set)"*
  shows *"CHAR('a :: semiring_1_cancel) > 0"*
⟨*proof*⟩

lemma *CHAR_dvd_CARD: "CHAR('a :: ring_1) dvd CARD('a)"*
⟨*proof*⟩

lemma (**in** *idom*) *prime_CHAR_semidom:*
  assumes *"CHAR('a) > 0"*
  shows *"prime CHAR('a)"*
⟨*proof*⟩

Characteristics are preserved by typical functors (polynomials, power series, Laurent series):

lemma *semiring_char_poly [simp]: "CHAR('a :: comm_semiring_1 poly) = CHAR('a)"*
  ⟨*proof*⟩

lemma *semiring_char_fps [simp]: "CHAR('a :: comm_semiring_1 fps) = CHAR('a)"*
  ⟨*proof*⟩

lemma *fls_const_eq_0_iff [simp]: "fls_const c = 0 ⟷ c = 0"*
  ⟨*proof*⟩

lemma *semiring_char_fls [simp]: "CHAR('a :: comm_semiring_1 fls) = CHAR('a)"*
  ⟨*proof*⟩

lemma *irreducible_power_iff [simp]:*
  "irreducible (p ^ n) ⟷ irreducible p ∧ n = 1"
⟨*proof*⟩

lemma *pderiv_monom:*
  *"pderiv (Polynomial.monom c n) = of_nat n * Polynomial.monom c (n - 1)"*
⟨*proof*⟩

lemma *uminus_CHAR_2 [simp]:*
  assumes *"CHAR('a :: ring_1) = 2"*
  shows *"-(x :: 'a) = x"*
⟨*proof*⟩

lemma *minus_CHAR_2 [simp]:*
  assumes *"CHAR('a :: ring_1) = 2"*

shows *"(x - y :: 'a) = x + y"*
⟨*proof*⟩

lemma *minus_power_prime_CHAR:*
assumes *"p = CHAR('a :: {ring_1})" "prime p"*
shows *"(-x :: 'a) ^ p = -(x ^ p)"*
⟨*proof*⟩

end

1 Perfect Fields

theory *Perfect_Fields*
imports
"Berlekamp_Zassenhaus.Finite_Field"
Perfect_Field_Library
begin

1.1 Rings and fields with prime characteristic

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

class *semiring_prime_char = semiring_1 +*
assumes *prime_char_aux: "∃n. prime n ∧ of_nat n = (0 :: 'a)"*
begin

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

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

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

end

lemma *semiring_prime_charI [intro?]:*
"prime CHAR('a :: semiring_1) ⟹ OFCLASS('a, semiring_prime_char_class)"
⟨*proof*⟩

lemma *idom_prime_charI [intro?]:*
assumes *"CHAR('a :: idom) > 0"*
shows *"OFCLASS('a, semiring_prime_char_class)"*
⟨*proof*⟩

class *comm_semiring_prime_char = comm_semiring_1 + semiring_prime_char*
class *comm_ring_prime_char = comm_ring_1 + semiring_prime_char*

begin
subclass *comm_semiring_prime_char* ⟨*proof*⟩
end
class *idom_prime_char = idom + semiring_prime_char*
begin
subclass *comm_ring_prime_char* ⟨*proof*⟩
end

class *field_prime_char = field +*
  assumes *pos_char_exists: "∃n>0. of_nat n = (0 :: 'a)"*
begin
subclass *idom_prime_char*
  ⟨*proof*⟩
end

lemma *field_prime_charI [intro?]:*
  "n > 0 ⟹ of_nat n = (0 :: 'a :: field) ⟹ OFCLASS('a, field_prime_char_class)"
  ⟨*proof*⟩

lemma *field_prime_charI' [intro?]:*
  "CHAR('a :: field) > 0 ⟹ OFCLASS('a, field_prime_char_class)"
  ⟨*proof*⟩

Typical functors like polynomials, formal power seires, and formal Laurent series preserve the characteristic of the coefficient ring.

instance *poly :: ("{semiring_prime_char,comm_semiring_1}") semiring_prime_char*
  ⟨*proof*⟩
instance *poly :: ("{comm_semiring_prime_char,comm_semiring_1}") comm_semiring_prime_char*
  ⟨*proof*⟩
instance *poly :: ("{comm_ring_prime_char,comm_semiring_1}") comm_ring_prime_char*
  ⟨*proof*⟩
instance *poly :: ("{idom_prime_char,comm_semiring_1}") idom_prime_char*
  ⟨*proof*⟩

instance *fps :: ("{semiring_prime_char,comm_semiring_1}") semiring_prime_char*
  ⟨*proof*⟩
instance *fps :: ("{comm_semiring_prime_char,comm_semiring_1}") comm_semiring_prime_char*
  ⟨*proof*⟩
instance *fps :: ("{comm_ring_prime_char,comm_semiring_1}") comm_ring_prime_char*
  ⟨*proof*⟩
instance *fps :: ("{idom_prime_char,comm_semiring_1}") idom_prime_char*
  ⟨*proof*⟩

instance *fls :: ("{semiring_prime_char,comm_semiring_1}") semiring_prime_char*
  ⟨*proof*⟩
instance *fls :: ("{comm_semiring_prime_char,comm_semiring_1}") comm_semiring_prime_char*
  ⟨*proof*⟩
instance *fls :: ("{comm_ring_prime_char,comm_semiring_1}") comm_ring_prime_char*

⟨*proof*⟩
instance `fls :: ("{idom_prime_char,comm_semiring_1}") idom_prime_char`
⟨*proof*⟩
instance `fls :: ("{field_prime_char,comm_semiring_1}") field_prime_char`
⟨*proof*⟩

1.2 Finite fields

class `finite_field = field_prime_char + finite`

lemma `finite_fieldI [intro?]:`
assumes `"finite (UNIV :: 'a :: field set)"`
shows `"OFCLASS('a, finite_field_class)"`
⟨*proof*⟩

class `enum_finite_field = finite_field +`
fixes `enum_finite_field :: "nat ⇒ 'a"`
assumes `enum_finite_field: "enum_finite_field ` {..<CARD('a)} = UNIV"`
begin

lemma `inj_on_enum_finite_field: "inj_on enum_finite_field {..<CARD('a)}"`
⟨*proof*⟩

end

instance `mod_ring :: (prime_card) finite_field`
⟨*proof*⟩

instantiation `mod_ring :: (prime_card) enum_finite_field`
begin

definition `enum_finite_field_mod_ring :: "nat ⇒ 'a mod_ring"` **where**
`"enum_finite_field_mod_ring n = of_int_mod_ring (int n)"`

instance ⟨*proof*⟩

end

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

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

lemma `finite_field_power_card_eq_same:`
fixes `x :: "'a :: finite_field"`
shows `"x ^ CARD('a) = x"`

⟨*proof*⟩

lemma *finite_field_power_card_power_eq_same:*
fixes *x :: "'a :: finite_field"*
assumes *"m = CARD('a) ^ n"*
shows *"x ^ m = x"*
⟨*proof*⟩

typedef (overloaded) *'a :: semiring_1 ring_char = "if CHAR('a) = 0 then UNIV else {0..<CHAR('a)}"*
⟨*proof*⟩

lemma *CARD_ring_char [simp]: "CARD ('a :: semiring_1 ring_char) = CHAR('a)"*
⟨*proof*⟩

instance *ring_char :: (semiring_prime_char) nontriv*
⟨*proof*⟩

instance *ring_char :: (semiring_prime_char) prime_card*
⟨*proof*⟩

lemma *to_int_mod_ring_add:*
"to_int_mod_ring (x + y :: 'a :: finite mod_ring) = (to_int_mod_ring x + to_int_mod_ring y) mod CARD('a)"
⟨*proof*⟩

lemma *to_int_mod_ring_mult:*
*"to_int_mod_ring (x * y :: 'a :: finite mod_ring) = (to_int_mod_ring x * to_int_mod_ring y) mod CARD('a)"*
⟨*proof*⟩

lemma *of_nat_mod_CHAR [simp]: "of_nat (x mod CHAR('a :: semiring_1)) = (of_nat x :: 'a)"*
⟨*proof*⟩

lemma *of_int_mod_CHAR [simp]: "of_int (x mod int CHAR('a :: ring_1)) = (of_int x :: 'a)"*
⟨*proof*⟩

lemma (in *vector_space)* *bij_betw_representation:*
assumes *[simp]: "independent B" "finite B"*
shows *"bij_betw (λv. ∑ b∈B. scale (v b) b) (B →$_E$ UNIV) (span B)"*
⟨*proof*⟩

lemma (in *vector_space)* *card_span:*
assumes *[simp]: "independent B" "finite B"*
shows *"card (span B) = CARD('a) ^ card B"*
⟨*proof*⟩

lemma (**in** *zero_neq_one*) *CARD_neq_1*: "CARD('a) ≠ Suc 0"
⟨*proof*⟩

theorem *CARD_finite_field_is_CHAR_power*: "∃n>0. CARD('a :: finite_field) = CHAR('a) ^ n"
⟨*proof*⟩

1.3 The Freshman's Dream in rings of non-zero characteristic

lemma (**in** *comm_semiring_1*) *freshmans_dream*:
fixes *x y* :: 'a **and** *n* :: *nat*
assumes "prime CHAR('a)"
assumes *n_def*: "n = CHAR('a)"
shows "(x + y) ^ n = x ^ n + y ^ n"
⟨*proof*⟩

lemma (**in** *comm_semiring_1*) *freshmans_dream'*:
assumes [*simp*]: "prime CHAR('a)" **and** "m = CHAR('a) ^ n"
shows "(x + y :: 'a) ^ m = x ^ m + y ^ m"
⟨*proof*⟩

lemma (**in** *comm_semiring_1*) *freshmans_dream_sum*:
fixes *f* :: "'b ⇒ 'a"
assumes "prime CHAR('a)" **and** "n = CHAR('a)"
shows "sum f A ^ n = sum (λi. f i ^ n) A"
⟨*proof*⟩

lemma (**in** *comm_semiring_1*) *freshmans_dream_sum'*:
fixes *f* :: "'b ⇒ 'a"
assumes "prime CHAR('a)" "m = CHAR('a) ^ n"
shows "sum f A ^ m = sum (λi. f i ^ m) A"
⟨*proof*⟩

1.4 The Frobenius endomorphism

definition (**in** *semiring_1*) *frob* :: "'a ⇒ 'a" **where**
"frob x = x ^ CHAR('a)"

definition (**in** *semiring_1*) *inv_frob* :: "'a ⇒ 'a" **where**
"inv_frob x = (if x ∈ {0, 1} then x else if x ∈ range frob then inv_into UNIV frob x else x)"

lemma (**in** *semiring_1*) *inv_frob_0* [*simp*]: "inv_frob 0 = 0"
and *inv_frob_1* [*simp*]: "inv_frob 1 = 1"
⟨*proof*⟩

lemma (**in** *semiring_prime_char*) *frob_0* [*simp*]: "frob (0 :: 'a) = 0"
⟨*proof*⟩

lemma (**in** *semiring_1*) *frob_1 [simp]*: "frob 1 = 1"
⟨*proof*⟩

lemma (**in** *comm_semiring_1*) *frob_mult*: "frob (x * y) = frob x * frob (y :: 'a)"
⟨*proof*⟩

lemma (**in** *comm_semiring_1*)
frob_add: "prime CHAR('a) ⟹ frob (x + y :: 'a) = frob x + frob (y :: 'a)"
⟨*proof*⟩

lemma (**in** *comm_ring_1*) *frob_uminus*: "prime CHAR('a) ⟹ frob (-x :: 'a) = -frob x"
⟨*proof*⟩

lemma (**in** *comm_ring_prime_char*) *frob_diff*:
"prime CHAR('a) ⟹ frob (x - y :: 'a) = frob x - frob (y :: 'a)"
⟨*proof*⟩

interpretation *frob_sr*: semiring_hom "frob :: 'a :: {comm_semiring_prime_char} ⇒ 'a"
⟨*proof*⟩

interpretation *frob*: ring_hom "frob :: 'a :: {comm_ring_prime_char} ⇒ 'a"
⟨*proof*⟩

interpretation *frob*: field_hom "frob :: 'a :: {field_prime_char} ⇒ 'a"
⟨*proof*⟩

lemma *frob_mod_ring' [simp]*: "(x :: 'a :: prime_card mod_ring) ^ CARD('a) = x"
⟨*proof*⟩

lemma *frob_mod_ring [simp]*: "frob (x :: 'a :: prime_card mod_ring) = x"
⟨*proof*⟩

context *semiring_1_no_zero_divisors*
begin

lemma *frob_eq_0*:
"frob (x :: 'a) = 0 ⟹ x = 0"
⟨*proof*⟩

lemma *frob_eq_0_iff [simp]*:
"frob (x :: 'a) = 0 ⟷ x = 0 ∧ CHAR('a) > 0"
⟨*proof*⟩

end

context *idom_prime_char*
begin

lemma *inj_frob: "inj (frob :: 'a ⇒ 'a)"*
⟨*proof*⟩

lemma *frob_eq_frob_iff [simp]:*
"frob (x :: 'a) = frob y ⟷ x = y"
⟨*proof*⟩

lemma *frob_eq_1_iff [simp]: "frob (x :: 'a) = 1 ⟷ x = 1"*
⟨*proof*⟩

lemma *inv_frob_frob [simp]: "inv_frob (frob (x :: 'a)) = x"*
⟨*proof*⟩

lemma *frob_inv_frob [simp]:*
assumes *"x ∈ range frob"*
shows *"frob (inv_frob x) = (x :: 'a)"*
⟨*proof*⟩

lemma *inv_frob_eqI: "frob y = x ⟹ inv_frob x = y"*
⟨*proof*⟩

lemma *inv_frob_eq_0_iff [simp]: "inv_frob (x :: 'a) = 0 ⟷ x = 0"*
⟨*proof*⟩

end

class *surj_frob = field_prime_char +*
assumes *surj_frob [simp]: "surj (frob :: 'a ⇒ 'a)"*
begin

lemma *in_range_frob [simp, intro]: "(x :: 'a) ∈ range frob"*
⟨*proof*⟩

lemma *inv_frob_eq_iff [simp]: "inv_frob (x :: 'a) = y ⟷ frob y = x"*
⟨*proof*⟩

end

The following type class describes a field with a surjective Frobenius endo-

morphism that is effectively computable. This includes all finite fields.

class *inv_frob = surj_frob +*
  fixes *inv_frob_code :: "'a ⇒ 'a"*
  assumes *inv_frob_code: "inv_frob x = inv_frob_code x"*

lemmas *[code] = inv_frob_code*

context *finite_field*
begin

subclass *surj_frob*
⟨*proof*⟩

end

lemma *inv_frob_mod_ring [simp]: "inv_frob (x :: 'a :: prime_card mod_ring) = x"*
  ⟨*proof*⟩

instantiation *mod_ring :: (prime_card) inv_frob*
begin

definition *inv_frob_code_mod_ring :: "'a mod_ring ⇒ 'a mod_ring"* **where**
  "inv_frob_code_mod_ring x = x"

instance
  ⟨*proof*⟩

end

1.5 Inverting the Frobenius endomorphism on polynomials

If `K` is a field of prime characteristic `p` with a surjective Frobenius endomorphism, every polynomial `P` with `P' = 0` has a `p`-th root.
To see that, let $\phi(a) = a^p$ denote the Frobenius endomorphism of `K` and its extension to `K[X]`.
If `P' = 0` for some `P ∈ K[X]`, then `P` must be of the form

$$P = a_0 + a_p x^p + a_{2p} x^{2p} + \ldots + a_{kp} x^{kp} \ .$$

If we now set

$$Q := \phi^{-1}(a_0) + \phi^{-1}(a_p) x + \phi^{-1}(a_{2p}) x^2 + \ldots + \phi^{-1}(a_{kp}) x^k$$

we get $\phi(Q) = P$, i.e. Q is the p-th root of $P(x)$.

lift_definition *inv_frob_poly :: "'a :: field poly ⇒ 'a poly"* **is**

```
  "λp i. if CHAR('a) = 0 then p i else inv_frob (p (i * CHAR('a)) :: 'a)"
```
⟨*proof*⟩

lemma *coeff_inv_frob_poly [simp]:*
  fixes *p :: "'a :: field poly"*
  assumes *"CHAR('a) > 0"*
  shows *"poly.coeff (inv_frob_poly p) i = inv_frob (poly.coeff p (i * CHAR('a)))"*
  ⟨*proof*⟩

lemma *inv_frob_poly_0 [simp]: "inv_frob_poly 0 = 0"*
  ⟨*proof*⟩

lemma *inv_frob_poly_1 [simp]: "inv_frob_poly 1 = 1"*
  ⟨*proof*⟩

lemma *degree_inv_frob_poly_le:*
  fixes *p :: "'a :: field poly"*
  assumes *"CHAR('a) > 0"*
  shows *"Polynomial.degree (inv_frob_poly p) ≤ Polynomial.degree p div CHAR('a)"*
⟨*proof*⟩

context
  assumes *"SORT_CONSTRAINT('a :: comm_ring_1)"*
  assumes *prime_char: "prime CHAR('a)"*
begin

lemma *poly_power_prime_char_as_sum_of_monoms:*
  fixes *h :: "'a poly"*
  shows *"h ^ CHAR('a) = (∑i≤Polynomial.degree h. Polynomial.monom (Polynomial.coeff h i ^ CHAR('a)) (CHAR('a)*i))"*
⟨*proof*⟩

lemma *coeff_of_prime_char_power [simp]:*
  fixes *y :: "'a poly"*
  shows *"poly.coeff (y ^ CHAR('a)) (i * CHAR('a)) = poly.coeff y i ^ CHAR('a)"*
  ⟨*proof*⟩

lemma *coeff_of_prime_char_power':*
  fixes *y :: "'a poly"*
  shows *"poly.coeff (y ^ CHAR('a)) i =*
          (if CHAR('a) dvd i then poly.coeff y (i div CHAR('a)) ^ CHAR('a) else 0)"
⟨*proof*⟩

end

context
  assumes *"SORT_CONSTRAINT('a :: field)"*
  assumes *pos_char: "CHAR('a) > 0"*
begin

interpretation *field_prime_char "(/)" inverse "(*)" "1 :: 'a" "(+)" 0 "(-)" uminus*
  rewrites *"semiring_1.frob 1 (*) (+) (0 :: 'a) = frob"* **and**
    "semiring_1.inv_frob 1 (*) (+) (0 :: 'a) = inv_frob" **and**
    "semiring_1.semiring_char 1 (+) 0 TYPE('a) = CHAR('a)"
⟨*proof*⟩

lemma *inv_frob_poly_power': "inv_frob_poly (p ^ CHAR('a) :: 'a poly) = p"*
  ⟨*proof*⟩

lemma *inv_frob_poly_power:*
  fixes *p :: "'a poly"*
  assumes *"is_nth_power CHAR('a) p"* **and** *"n = CHAR('a)"*
  shows *"inv_frob_poly p ^ CHAR('a) = p"*
⟨*proof*⟩

theorem *pderiv_eq_0_imp_nth_power:*
  assumes *"pderiv (p :: 'a poly) = 0"*
  assumes *[simp]: "surj (frob :: 'a ⇒ 'a)"*
  shows *"is_nth_power CHAR('a) p"*
⟨*proof*⟩

end

1.6 Code generation

We now also make this notion of "taking the *p*-th root of a polynomial" executable. For this, we need an auxiliary function that takes a list $[x_0, \ldots, x_m]$ and returns the list of every *n*-th element, i.e. it throws away all elements except those x_i where i is a multiple of n.

fun *take_every :: "nat ⇒ 'a list ⇒ 'a list"* **where**
  "take_every _ [] = []"
| "take_every n (x # xs) = x # take_every n (drop (n - 1) xs)"

lemma *take_every_0 [simp]: "take_every 0 xs = xs"*
  ⟨*proof*⟩

lemma *take_every_1 [simp]: "take_every (Suc 0) xs = xs"*
  ⟨*proof*⟩

lemma *int_length_take_every: "n > 0 ⟹ int (length (take_every n xs)) = ceiling (length xs / n)"*

⟨*proof*⟩

lemma *length_take_every:*
  "n > 0 ⟹ length (take_every n xs) = nat (ceiling (length xs / n))"
  ⟨*proof*⟩

lemma *take_every_nth [simp]:*
  *"n > 0 ⟹ i < length (take_every n xs) ⟹ take_every n xs ! i = xs ! (n * i)"*
⟨*proof*⟩

lemma *coeffs_eq_strip_whileI:*
  assumes *"⋀i. i < length xs ⟹ Polynomial.coeff p i = xs ! i"*
  assumes *"p ≠ 0 ⟹ length xs > Polynomial.degree p"*
  shows *"Polynomial.coeffs p = strip_while ((=) 0) xs"*
⟨*proof*⟩

This implements the code equation for *inv_frob_poly*.

lemma *inv_frob_poly_code [code]:*
  "Polynomial.coeffs (inv_frob_poly (p :: 'a :: field_prime_char poly)) =
     (if CHAR('a) = 0 then Polynomial.coeffs p else
        map inv_frob (strip_while ((=) 0) (take_every CHAR('a) (Polynomial.coeffs p))))"
     (is *"_ = If _ _ ?rhs")*
⟨*proof*⟩

1.7 Perfect fields

We now introduce perfect fields. The textbook definition of a perfect field is that every irreducible polynomial is separable, i.e. if a polynomial P has no non-trivial divisors then $\gcd(P, P') = 0$.

For technical reasons, this is somewhat difficult to express in Isabelle/HOL's typeclass system. We therefore use the following much simpler equivalent definition (and prove equivalence later): a field is perfect if it either has characteristic 0 or its Frobenius endomorphism is surjective.

class *perfect_field = field +*
  assumes *perfect_field: "CHAR('a) = 0 ∨ surj (frob :: 'a ⇒ 'a)"*

context *field_char_0*
begin
subclass *perfect_field*
  ⟨*proof*⟩
end

context *surj_frob*
begin
subclass *perfect_field*

⟨*proof*⟩
end

theorem *irreducible_imp_pderiv_nonzero:*
assumes *"irreducible (p :: 'a :: perfect_field poly)"*
shows *"pderiv p ≠ 0"*
⟨*proof*⟩

corollary *irreducible_imp_separable:*
assumes *"irreducible (p :: 'a :: perfect_field poly)"*
shows *"coprime p (pderiv p)"*
⟨*proof*⟩

end

1.8 Algebraically closed fields are perfect

theory *Perfect_Field_Algebraically_Closed*
imports *Perfect_Fields "Formal_Puiseux_Series.Formal_Puiseux_Series"*
begin

lemma (**in** *alg_closed_field*) *nth_root_exists:*
assumes *"n > 0"*
shows *"∃y. y ^ n = (x :: 'a)"*
⟨*proof*⟩

context *alg_closed_field*
begin

lemma *alg_closed_surj_frob:*
assumes *"CHAR('a) > 0"*
shows *"surj (frob :: 'a ⇒ 'a)"*
⟨*proof*⟩

sublocale *perfect_field*
⟨*proof*⟩

end

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

2 The algebraic closure type

theory *Algebraic_Closure_Type*
imports
"HOL-Algebra.Algebra"
"Formal_Puiseux_Series.Formal_Puiseux_Series"
"HOL-Computational_Algebra.Field_as_Ring"
begin

definition (**in** *ring_1*) *ring_of_type_algebra :: "'a ring"*
where *"ring_of_type_algebra = ⦇*
*carrier = UNIV, monoid.mult = (λx y. x * y),*
one = 1,
ring.zero = 0,
add = (λ x y. x + y) ⦈"

lemma (**in** *comm_ring_1*) *ring_from_type_algebra [intro]:*
"ring (ring_of_type_algebra :: 'a ring)"
⟨*proof*⟩

lemma (**in** *comm_ring_1*) *cring_from_type_algebra [intro]:*
"cring (ring_of_type_algebra :: 'a ring)"
⟨*proof*⟩

lemma (**in** *Fields.field*) *field_from_type_algebra [intro]:*
"field (ring_of_type_algebra :: 'a ring)"
⟨*proof*⟩

2.1 Definition

typedef (**overloaded**) *'a :: field alg_closure =*
"carrier (field.alg_closure (ring_of_type_algebra :: 'a :: field ring))"
⟨*proof*⟩

setup_lifting *type_definition_alg_closure*

instantiation *alg_closure :: (field) field*
begin

context
  fixes *L K*
  defines *"K ≡ (ring_of_type_algebra :: 'a :: field ring)"*
  defines *"L ≡ field.alg_closure K"*
begin

interpretation *K: field K*
  ⟨*proof*⟩

interpretation *algebraic_closure L "range K.indexed_const"*
⟨*proof*⟩

lift_definition *zero_alg_closure :: "'a alg_closure"* **is** *"ring.zero L"*
  ⟨*proof*⟩

lift_definition *one_alg_closure :: "'a alg_closure"* **is** *"monoid.one L"*
  ⟨*proof*⟩

lift_definition *plus_alg_closure :: "'a alg_closure ⇒ 'a alg_closure ⇒ 'a alg_closure"*
  is *"ring.add L"*
  ⟨*proof*⟩

lift_definition *minus_alg_closure :: "'a alg_closure ⇒ 'a alg_closure ⇒ 'a alg_closure"*
  is *"a_minus L"*
  ⟨*proof*⟩

lift_definition *times_alg_closure :: "'a alg_closure ⇒ 'a alg_closure ⇒ 'a alg_closure"*
  is *"monoid.mult L"*
  ⟨*proof*⟩

lift_definition *uminus_alg_closure :: "'a alg_closure ⇒ 'a alg_closure"*
  is *"a_inv L"*
  ⟨*proof*⟩

lift_definition *inverse_alg_closure :: "'a alg_closure ⇒ 'a alg_closure"*
  is *"λx. if x = ring.zero L then ring.zero L else m_inv L x"*
  ⟨*proof*⟩

lift_definition *divide_alg_closure :: "'a alg_closure ⇒ 'a alg_closure ⇒ 'a alg_closure"*

is *"λx y. if y = ring.zero L then ring.zero L else monoid.mult L x (m_inv L y)"*
⟨*proof*⟩

end

instance ⟨*proof*⟩

end

2.2 The algebraic closure is algebraically closed

instance *alg_closure :: (field) alg_closed_field*
⟨*proof*⟩

2.3 Converting between the base field and the closure

context
fixes *L K*
defines *"K ≡ (ring_of_type_algebra :: 'a :: field ring)"*
defines *"L ≡ field.alg_closure K"*
begin

interpretation *K: field K*
⟨*proof*⟩

interpretation *algebraic_closure L "range K.indexed_const"*
⟨*proof*⟩

lemma *alg_closure_hom: "K.indexed_const ∈ Ring.ring_hom K L"*
⟨*proof*⟩

lift_definition *to_ac :: "'a :: field ⇒ 'a alg_closure"*
is *"ring.indexed_const K"*
⟨*proof*⟩

lemma *to_ac_0 [simp]: "to_ac (0 :: 'a) = 0"*
⟨*proof*⟩

lemma *to_ac_1 [simp]: "to_ac (1 :: 'a) = 1"*
⟨*proof*⟩

lemma *to_ac_add [simp]: "to_ac (x + y :: 'a) = to_ac x + to_ac y"*
⟨*proof*⟩

lemma *to_ac_minus [simp]: "to_ac (-x :: 'a) = -to_ac x"*
⟨*proof*⟩

lemma *to_ac_diff [simp]: "to_ac (x - y :: 'a) = to_ac x - to_ac y"*
⟨*proof*⟩

lemma *to_ac_mult [simp]: "to_ac (x * y :: 'a) = to_ac x * to_ac y"*
⟨*proof*⟩

lemma *to_ac_inverse [simp]: "to_ac (inverse x :: 'a) = inverse (to_ac x)"*
⟨*proof*⟩

lemma *to_ac_divide [simp]: "to_ac (x / y :: 'a) = to_ac x / to_ac y"*
⟨*proof*⟩

lemma *to_ac_power [simp]: "to_ac (x ^ n) = to_ac x ^ n"*
⟨*proof*⟩

lemma *to_ac_of_nat [simp]: "to_ac (of_nat n) = of_nat n"*
⟨*proof*⟩

lemma *to_ac_of_int [simp]: "to_ac (of_int n) = of_int n"*
⟨*proof*⟩

lemma *to_ac_numeral [simp]: "to_ac (numeral n) = numeral n"*
⟨*proof*⟩

lemma *to_ac_sum: "to_ac (∑x∈A. f x) = (∑x∈A. to_ac (f x))"*
⟨*proof*⟩

lemma *to_ac_prod: "to_ac (∏x∈A. f x) = (∏x∈A. to_ac (f x))"*
⟨*proof*⟩

lemma *to_ac_sum_list: "to_ac (sum_list xs) = (∑x←xs. to_ac x)"*
⟨*proof*⟩

lemma *to_ac_prod_list: "to_ac (prod_list xs) = (∏x←xs. to_ac x)"*
⟨*proof*⟩

lemma *to_ac_sum_mset: "to_ac (sum_mset xs) = (∑x∈#xs. to_ac x)"*
⟨*proof*⟩

lemma *to_ac_prod_mset: "to_ac (prod_mset xs) = (∏x∈#xs. to_ac x)"*
⟨*proof*⟩

end

lemma (**in** *ring*) *indexed_const_eq_iff [simp]:*
"indexed_const x = (indexed_const y :: 'c multiset ⇒ 'a) ⟷ x = y"
⟨*proof*⟩

lemma *inj_to_ac: "inj to_ac"*
⟨*proof*⟩

lemma *to_ac_eq_iff [simp]: "to_ac x = to_ac y ⟷ x = y"*
⟨*proof*⟩

lemma *to_ac_eq_0_iff [simp]: "to_ac x = 0 ⟷ x = 0"*
and *to_ac_eq_0_iff' [simp]: "0 = to_ac x ⟷ x = 0"*
and *to_ac_eq_1_iff [simp]: "to_ac x = 1 ⟷ x = 1"*
and *to_ac_eq_1_iff' [simp]: "1 = to_ac x ⟷ x = 1"*
⟨*proof*⟩

definition *of_ac :: "'a :: field alg_closure ⇒ 'a"* **where**
"of_ac x = (if x ∈ range to_ac then inv_into UNIV to_ac x else 0)"

lemma *of_ac_eqI: "to_ac x = y ⟹ of_ac y = x"*
⟨*proof*⟩

lemma *of_ac_0 [simp]: "of_ac 0 = 0"*
and *of_ac_1 [simp]: "of_ac 1 = 1"*
⟨*proof*⟩

lemma *of_ac_to_ac [simp]: "of_ac (to_ac x) = x"*
⟨*proof*⟩

lemma *to_ac_of_ac: "x ∈ range to_ac ⟹ to_ac (of_ac x) = x"*
⟨*proof*⟩

lemma *CHAR_alg_closure [simp]:*
"CHAR('a :: field alg_closure) = CHAR('a)"
⟨*proof*⟩

instance *alg_closure :: (field_char_0) field_char_0*
⟨*proof*⟩

bundle *alg_closure_syntax*
begin
notation *to_ac ("_↑" [1000] 999)*
notation *of_ac ("_↓" [1000] 999)*
end

bundle *alg_closure_syntax'*
begin
notation (**output**) *to_ac ("_")*
notation (**output**) *of_ac ("_")*
end

2.4 The algebraic closure is an algebraic extension

The algebraic closure is an algebraic extension, i.e. every element in it is a root of some non-zero polynomial in the base field.

theorem *alg_closure_algebraic:*
fixes *x :: "'a :: field alg_closure"*
obtains *p :: "'a poly"* **where** *"p ≠ 0" "poly (map_poly to_ac p) x = 0"*
⟨*proof*⟩

instantiation *alg_closure :: (field)*
"{unique_euclidean_ring, normalization_euclidean_semiring, normalization_semidom_multipli
begin

definition *[simp]: "normalize_alg_closure = (normalize_field :: 'a alg_closure ⇒ _)"*
definition *[simp]: "unit_factor_alg_closure = (unit_factor_field :: 'a alg_closure ⇒ _)"*
definition *[simp]: "modulo_alg_closure = (mod_field :: 'a alg_closure ⇒ _)"*
definition *[simp]: "euclidean_size_alg_closure = (euclidean_size_field :: 'a alg_closure ⇒ _)"*
definition *[simp]: "division_segment (x :: 'a alg_closure) = 1"*

instance
⟨*proof*⟩

end

instantiation *alg_closure :: (field) euclidean_ring_gcd*
begin

definition *gcd_alg_closure :: "'a alg_closure ⇒ 'a alg_closure ⇒ 'a alg_closure"*
where
"gcd_alg_closure = Euclidean_Algorithm.gcd"
definition *lcm_alg_closure :: "'a alg_closure ⇒ 'a alg_closure ⇒ 'a alg_closure"*
where
"lcm_alg_closure = Euclidean_Algorithm.lcm"
definition *Gcd_alg_closure :: "'a alg_closure set ⇒ 'a alg_closure"* **where**
"Gcd_alg_closure = Euclidean_Algorithm.Gcd"
definition *Lcm_alg_closure :: "'a alg_closure set ⇒ 'a alg_closure"* **where**
"Lcm_alg_closure = Euclidean_Algorithm.Lcm"

instance ⟨*proof*⟩

end

instance *alg_closure :: (field) semiring_gcd_mult_normalize*
⟨*proof*⟩

end

2.5 Alternative definition of perfect fields

theory *Perfect_Field_Altdef*
imports
Algebraic_Closure_Type
Perfect_Fields
Perfect_Field_Algebraically_Closed
"HOL-Computational_Algebra.Field_as_Ring"
begin

instance *poly :: ("{field, normalization_euclidean_semiring, factorial_ring_gcd, semiring_gcd_mult_normalize}") factorial_semiring_multiplicative*
⟨*proof*⟩

In the following, we will show that our definition of perfect fields is equivalent to the usual textbook one (for example [1]). That is: a field in which every irreducible polynomial is separable (or, equivalently, has non-zero derivative) either has characteristic 0 or a surjective Frobenius endomorphism.

The proof works like this:

Let's call our field `K` with prime characteristic `p`. Suppose there were some `c` $\in$ `K` that is not a `p`-th root. The polynomial $P := X^p - c$ in $K[X]$ clearly has a zero derivative and is therefore not separable. By our assumption, it must then have a monic non-trivial factor $Q \in K[X]$.

Let `L` be some field extension of `K` where `c` does have a `p`-th root α (in our case, we choose `L` to be the algebraic closure of `K`).

Clearly, `Q` is also a non-trivial factor of `P` in `L`. However, we also have `P = X^p - c = X^p - α^p = (X - α)^p`, so we must have $Q = (X - \alpha)^m$ for some `0` $\leq$ `m < p` since `X - α` is prime.

However, the coefficient of X^{m-1} in $(X - \alpha)^m$ is `-mα`, and since `Q` $\in$ `K[X]` we must have `-mα` $\in$ `K` and therefore $\alpha \in$ `K`.

theorem *perfect_field_alt:*
assumes *"⋀p :: 'a :: field_gcd poly. Factorial_Ring.irreducible p ⟹ pderiv p ≠ 0"*
shows *"CHAR('a) = 0 ∨ surj (frob :: 'a ⇒ 'a)"*
⟨*proof*⟩

corollary *perfect_field_alt':*
assumes *"⋀p :: 'a :: field_gcd poly. Factorial_Ring.irreducible p ⟹ Rings.coprime p (pderiv p)"*
shows *"CHAR('a) = 0 ∨ surj (frob :: 'a ⇒ 'a)"*
⟨*proof*⟩

end

References

[1] K. Conrad. Perfect fields. Online at https://kconrad.math.uconn.edu/blurbs/galoistheory/perfect.pdf, 2021. Course notes, University of Connecticut.

[2] Wikipedia contributors. Perfect field — Wikipedia, the free encyclopedia, 2023. [Online; accessed 3-November-2023].