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 $\mathrm{char}(K) = 0$ or $\mathrm{char}(K) = p > 0$ 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 ⇒ 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

1 Perfect Fields **10**
1.1 Rings and fields with prime characteristic 10
1.2 Finite fields 12
1.3 The Freshman's Dream in rings of non-zero characteristic . . 16
1.4 The Frobenius endomorphism 18
1.5 Inverting the Frobenius endomorphism on polynomials 21
1.6 Code generation 25
1.7 Perfect fields 28
1.8 Algebraically closed fields are perfect 30

2 The algebraic closure type **32**
2.1 Definition 33
2.2 The algebraic closure is algebraically closed 36
2.3 Converting between the base field and the closure 37
2.4 The algebraic closure is an algebraic extension 42
2.5 Alternative definition of perfect fields 45

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


instance bool :: prime_card
  by standard auto


theorem (in comm_semiring_1) binomial_ring:
  "(a + b :: 'a)^n = (∑k≤n. (of_nat (n choose k)) * a^k * b^(n-k))"
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  have decomp: "{0..n+1} = {0} ∪ {n + 1} ∪ {1..n}"
    by auto
  have decomp2: "{0..n} = {0} ∪ {1..n}"
    by auto
  have "(a + b)^(n+1) = (a + b) * (∑k≤n. of_nat (n choose k) * a^k *
b^(n - k))"
    using Suc.hyps by simp
  also have "... = a * (∑k≤n. of_nat (n choose k) * a^k * b^(n-k)) +
      b * (∑k≤n. of_nat (n choose k) * a^k * b^(n-k))"
    by (rule distrib_right)
  also have "... = (∑k≤n. of_nat (n choose k) * a^(k+1) * b^(n-k)) +
      (∑k≤n. of_nat (n choose k) * a^k * b^(n - k + 1))"
    by (auto simp add: sum_distrib_left ac_simps)
  also have "... = (∑k≤n. of_nat (n choose k) * a^k * b^(n + 1 - k))
+
      (∑k=1..n+1. of_nat (n choose (k - 1)) * a^k * b^(n + 1 - k))"
    by (simp add: atMost_atLeast0 sum.shift_bounds_cl_Suc_ivl Suc_diff_le
field_simps del: sum.cl_ivl_Suc)
  also have "... = b^(n + 1) +
      (∑k=1..n. of_nat (n choose k) * a^k * b^(n + 1 - k)) + (a^(n +
1) +
      (∑k=1..n. of_nat (n choose (k - 1)) * a^k * b^(n + 1 - k)))"
      using sum.nat_ivl_Suc' [of 1 n "λk. of_nat (n choose (k-1)) * a
^ k * b ^ (n + 1 - k)"]
    by (simp add: sum.atLeast_Suc_atMost atMost_atLeast0)
  also have "... = a^(n + 1) + b^(n + 1) +
      (∑k=1..n. of_nat (n + 1 choose k) * a^k * b^(n + 1 - k))"
    by (auto simp add: field_simps sum.distrib [symmetric] choose_reduce_nat)
  also have "... = (∑k≤n+1. of_nat (n + 1 choose k) * a^k * b^(n + 1
```

```
- k))"
    using decomp by (simp add: atMost_atLeast0 field_simps)
  finally show ?case
    by simp
qed


lemma prime_not_dvd_fact:
assumes kn: "k < n" and prime_n: "prime n"
shows "¬ n dvd fact k"
  using kn leD prime_dvd_fact_iff prime_n by auto


lemma dvd_choose_prime:
assumes kn: "k < n" and k: "k ≠ 0" and n: "n ≠ 0" and prime_n: "prime
n"
shows "n dvd (n choose k)"
proof -
  have "n dvd (fact n)" by (simp add: fact_num_eq_if n)
  moreover have "¬ n dvd (fact k * fact (n-k))"
  proof (rule ccontr, safe)
    assume "n dvd fact k * fact (n - k)"
    hence "n dvd fact k ∨ n dvd fact (n - k)" using prime_dvd_mult_eq_nat[OF
prime_n] by simp
    moreover have "¬ n dvd (fact k)" by (rule prime_not_dvd_fact[OF
kn prime_n])
    moreover have "¬ n dvd fact (n - k)" using  prime_not_dvd_fact[OF
_ prime_n] kn k by simp
    ultimately show False by simp
  qed
  moreover have "(fact n::nat) = fact k * fact (n-k) * (n choose k)"
    using binomial_fact_lemma kn by auto
  ultimately show ?thesis using prime_n
    by (auto simp add: prime_dvd_mult_iff)
qed

lemma CHAR_not_1 [simp]: "CHAR('a :: {semiring_1, zero_neq_one}) ≠ Suc
0"
  by (metis One_nat_def of_nat_1 of_nat_CHAR zero_neq_one)

lemma (in idom) CHAR_not_1' [simp]: "CHAR('a) ≠ Suc 0"
  using local.of_nat_CHAR by fastforce

lemma semiring_char_mod_ring [simp]:
  "CHAR('n :: nontriv mod_ring) = CARD('n)"
proof (rule CHAR_eq_posI)
  fix x assume "x > 0" "x < CARD('n)"
  thus "of_nat x ≠ (0 :: 'n mod_ring)"
    by transfer auto
```

```
qed auto

lemma of_nat_eq_iff_cong_CHAR:
  "of_nat x = (of_nat y :: 'a :: semiring_1_cancel) ⟷ [x = y] (mod CHAR('a))"
proof (induction x y rule: linorder_wlog)
  case (le x y)
  define z where "z = y - x"
  have [simp]: "y = x + z"
    using le by (auto simp: z_def)
  have "(CHAR('a) dvd z) = [x = x + z] (mod CHAR('a))"
    by (metis ‹y = x + z› cong_def le mod_eq_dvd_iff_nat z_def)
  thus ?case
    by (simp add: of_nat_eq_0_iff_char_dvd)
qed (simp add: eq_commute cong_sym_eq)

lemma (in ring_1) of_int_eq_0_iff_char_dvd:
  "(of_int n = (0 :: 'a)) = (int CHAR('a) dvd n)"
proof (cases "n ≥ 0")
  case True
  hence "(of_int n = (0 :: 'a)) ⟷ (of_nat (nat n)) = (0 :: 'a)"
    by auto
  also have "... ⟷ CHAR('a) dvd nat n"
    by (subst of_nat_eq_0_iff_char_dvd) auto
  also have "... ⟷ int CHAR('a) dvd n"
    using True by presburger
  finally show ?thesis .
next
  case False
  hence "(of_int n = (0 :: 'a)) ⟷ -(of_nat (nat (-n))) = (0 :: 'a)"
    by auto
  also have "... ⟷ CHAR('a) dvd nat (-n)"
    by (auto simp: of_nat_eq_0_iff_char_dvd)
  also have "... ⟷ int CHAR('a) dvd n"
    using False dvd_nat_abs_iff[of "CHAR('a)" n] by simp
  finally show ?thesis .
qed

lemma (in ring_1) of_int_eq_iff_cong_CHAR:
  "of_int x = (of_int y :: 'a) ⟷ [x = y] (mod int CHAR('a))"
proof -
  have "of_int x = (of_int y :: 'a) ⟷ of_int (x - y) = (0 :: 'a)"
    by auto
  also have "... ⟷ (int CHAR('a) dvd x - y)"
    by (rule of_int_eq_0_iff_char_dvd)
  also have "... ⟷ [x = y] (mod int CHAR('a))"
    by (simp add: cong_iff_dvd_diff)
  finally show ?thesis .
qed
```

```
lemma finite_imp_CHAR_pos:
  assumes "finite (UNIV :: 'a set)"
  shows   "CHAR('a :: semiring_1_cancel) > 0"
proof -
  have "∃n∈UNIV. infinite {m ∈ UNIV. of_nat m = (of_nat n :: 'a)}"
  proof (rule pigeonhole_infinite)
    show "infinite (UNIV :: nat set)"
      by simp
    show "finite (range (of_nat :: nat ⇒ 'a))"
      by (rule finite_subset[OF _ assms]) auto
  qed
  then obtain n :: nat where "infinite {m ∈ UNIV. of_nat m = (of_nat
n :: 'a)}"
    by blast
  hence "¬({m ∈ UNIV. of_nat m = (of_nat n :: 'a)} ⊆ {n})"
    by (intro notI) (use finite_subset in blast)
  then obtain m where "m ≠ n" "of_nat m = (of_nat n :: 'a)"
    by blast
  hence "[m = n] (mod CHAR('a))"
    by (simp add: of_nat_eq_iff_cong_CHAR)
  hence "CHAR('a) ≠ 0"
    using ‹m ≠ n› by (intro notI) auto
  thus ?thesis
    by simp
qed

lemma CHAR_dvd_CARD: "CHAR('a :: ring_1) dvd CARD('a)"
proof (cases "CARD('a) = 0")
  case False
  hence [intro]: "CHAR('a) > 0"
    by (simp add: card_eq_0_iff finite_imp_CHAR_pos)
  define G where "G = ⦇ carrier = (UNIV :: 'a set), monoid.mult = (+),
one = (0 :: 'a) ⦈"
  define H where "H = (of_nat ` {..<CHAR('a)} :: 'a set)"
  interpret group G
  proof (rule groupI)
    fix x assume x: "x ∈ carrier G"
    show "∃y∈carrier G. y ⊗_G x = 1_G"
      by (intro bexI[of _ "-x"]) (auto simp: G_def)
  qed (auto simp: G_def add_ac)

  interpret subgroup H G
  proof
    show "1_G ∈ H"
      using False unfolding G_def H_def
      by (intro image_eqI[of _ _ 0]) auto
  next
    fix x y :: 'a
    assume "x ∈ H" "y ∈ H"
```

```
    then obtain x' y' where [simp]: "x = of_nat x'" "y = of_nat y'"
      by (auto simp: H_def)
    have "x + y = of_nat ((x' + y') mod CHAR('a))"
      by (auto simp flip: of_nat_add simp: of_nat_eq_iff_cong_CHAR)
    moreover have "(x' + y') mod CHAR('a) < CHAR('a)"
      using H_def ‹y ∈ H› by fastforce
    ultimately show "x ⊗G y ∈ H"
      by (auto simp: H_def G_def intro!: imageI)
  next
    fix x :: 'a
    assume x: "x ∈ H"
    then obtain x' where [simp]: "x = of_nat x'" and x': "x' < CHAR('a)"
      by (auto simp: H_def)
    have "CHAR('a) dvd x' + (CHAR('a) - x') mod CHAR('a)"
      by (metis x' dvd_eq_mod_eq_0 le_add_diff_inverse mod_add_right_eq
mod_self order_less_imp_le)
    hence "x + of_nat ((CHAR('a) - x') mod CHAR('a)) = 0"
      by (auto simp flip: of_nat_add simp: of_nat_eq_0_iff_char_dvd)
    moreover from this have "invG x = of_nat ((CHAR('a) - x') mod CHAR('a))"
      by (intro inv_equality) (auto simp: G_def add_ac)
    moreover have "of_nat ((CHAR('a) - x') mod CHAR('a)) ∈ H"
      unfolding H_def using ‹CHAR('a) > 0› by (intro imageI) auto
    ultimately show "invG x ∈ H" by force
  qed (auto simp: G_def H_def)

  have "card H dvd card (rcosetsG H) * card H"
    by simp
  also have "card (rcosetsG H) * card H = Coset.order G"
  proof (rule lagrange_finite)
    show "finite (carrier G)"
      using False card_ge_0_finite by (auto simp: G_def)
  qed (fact is_subgroup)
  finally have "card H dvd CARD('a)"
    by (simp add: Coset.order_def G_def)
  also have "card H = card {..<CHAR('a)}"
    unfolding H_def by (intro card_image inj_onI) (auto simp: of_nat_eq_iff_cong_CHAR
cong_def)
  finally show "CHAR('a) dvd CARD('a)"
    by simp
qed auto

lemma (in idom) prime_CHAR_semidom:
  assumes "CHAR('a) > 0"
  shows   "prime CHAR('a)"
proof -
  have False if ab: "a ≠ 1" "b ≠ 1" "CHAR('a) = a * b" for a b
  proof -
    from assms ab have "a > 0" "b > 0"
      by (auto intro!: Nat.gr0I)
```

```
    have "of_nat (a * b) = (0 :: 'a)"
      using ab by (metis of_nat_CHAR)
    also have "of_nat (a * b) = (of_nat a :: 'a) * of_nat b"
      by simp
    finally have "of_nat a * of_nat b = (0 :: 'a)" .
    moreover have "of_nat a * of_nat b ≠ (0 :: 'a)"
      using ab ‹a > 0› ‹b > 0›
      by (intro no_zero_divisors) (auto simp: of_nat_eq_0_iff_char_dvd)
    ultimately show False
      by contradiction
  qed
  moreover have "CHAR('a) > 1"
    using assms CHAR_not_1' by linarith
  ultimately have "prime_elem CHAR('a)"
    by (intro irreducible_imp_prime_elem) (auto simp: Factorial_Ring.irreducible_def)
  thus ?thesis
    by auto
qed
```

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

```
lemma semiring_char_poly [simp]: "CHAR('a :: comm_semiring_1 poly) =
CHAR('a)"
  by (rule CHAR_eqI) (auto simp: of_nat_poly of_nat_eq_0_iff_char_dvd)

lemma semiring_char_fps [simp]: "CHAR('a :: comm_semiring_1 fps) = CHAR('a)"
  by (rule CHAR_eqI) (auto simp flip: fps_of_nat simp: of_nat_eq_0_iff_char_dvd)


lemma fls_const_eq_0_iff [simp]: "fls_const c = 0 ⟷ c = 0"
  using fls_const_0 fls_const_nonzero by blast

lemma semiring_char_fls [simp]: "CHAR('a :: comm_semiring_1 fls) = CHAR('a)"
  by (rule CHAR_eqI) (auto simp: fls_of_nat of_nat_eq_0_iff_char_dvd fls_const_nonzero)

lemma irreducible_power_iff [simp]:
  "irreducible (p ^ n) ⟷ irreducible p ∧ n = 1"
proof
  assume *: "irreducible (p ^ n)"
  have [simp]: "¬p dvd 1"
  proof
    assume "p dvd 1"
    hence "p ^ n dvd 1"
      by (metis dvd_power_same power_one)
    with * show False
      by auto
  qed

  consider "n = 0" | "n = 1" | "n > 1"
```

```
    by linarith
  thus "irreducible p ∧ n = 1"
  proof cases
    assume "n > 1"
    hence "p ^ n = p * p ^ (n - 1)"
      by (cases n) auto
    with * ‹¬ p dvd 1› have "p ^ (n - 1) dvd 1"
      using irreducible_multD by blast
    with ‹¬p dvd 1› and ‹n > 1› have False
      by (meson dvd_power dvd_trans zero_less_diff)
    thus ?thesis ..
  qed (use * in auto)
qed auto

lemma pderiv_monom:
  "pderiv (Polynomial.monom c n) = of_nat n * Polynomial.monom c (n -
1)"
proof (cases n)
  case (Suc n)
  show ?thesis
    unfolding monom_altdef Suc pderiv_smult pderiv_power_Suc pderiv_pCons
    by (simp add: of_nat_poly)
qed (auto simp: monom_altdef)

lemma uminus_CHAR_2 [simp]:
  assumes "CHAR('a :: ring_1) = 2"
  shows   "-(x :: 'a) = x"
proof -
  have "x + x = 2 * x"
    by (simp add: mult_2)
  also have "2 = (0 :: 'a)"
    using assms by (metis of_nat_CHAR of_nat_numeral)
  finally show ?thesis
    by (simp add: add_eq_0_iff2)
qed

lemma minus_CHAR_2 [simp]:
  assumes "CHAR('a :: ring_1) = 2"
  shows   "(x - y :: 'a) = x + y"
  using uminus_CHAR_2[of y] assms by simp

lemma minus_power_prime_CHAR:
  assumes "p = CHAR('a :: {ring_1})" "prime p"
  shows "(-x :: 'a) ^ p = -(x ^ p)"
proof (cases "p = 2")
  case False
  have "prime p"
    using assms by blast
  with False have "odd p"
```

```
    using primes_dvd_imp_eq two_is_prime_nat by blast
  thus ?thesis
    by simp
qed (use assms in auto)

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"
  using local.CHAR_pos_iff local.prime_char_aux prime_gt_0_nat by blast

lemma CHAR_nonzero [simp]: "CHAR('a) ≠ 0"
  using CHAR_pos by auto

lemma CHAR_prime [intro, simp]: "prime CHAR('a)"
  by (metis (mono_tags, lifting) gcd_nat.order_iff_strict local.of_nat_1
local.of_nat_eq_0_iff_char_dvd
        local.one_neq_zero local.prime_char_aux prime_nat_iff)

end

lemma semiring_prime_charI [intro?]:
  "prime CHAR('a :: semiring_1) ⟹ OFCLASS('a, semiring_prime_char_class)"
  by standard auto


lemma idom_prime_charI [intro?]:
  assumes "CHAR('a :: idom) > 0"
  shows   "OFCLASS('a, semiring_prime_char_class)"
proof
  show "prime CHAR('a)"
    using assms prime_CHAR_semidom by blast
qed
```

```
class comm_semiring_prime_char = comm_semiring_1 + semiring_prime_char
class comm_ring_prime_char = comm_ring_1 + semiring_prime_char
begin
subclass comm_semiring_prime_char ..
end
class idom_prime_char = idom + semiring_prime_char
begin
subclass comm_ring_prime_char ..
end

class field_prime_char = field +
  assumes pos_char_exists: "∃n>0. of_nat n = (0 :: 'a)"
begin
subclass idom_prime_char
  apply standard
  using pos_char_exists local.CHAR_pos_iff local.of_nat_CHAR local.prime_CHAR_semidom
by blast
end


lemma field_prime_charI [intro?]:
  "n > 0 ⟹ of_nat n = (0 :: 'a :: field) ⟹ OFCLASS('a, field_prime_char_class)"
  by standard auto

lemma field_prime_charI' [intro?]:
  "CHAR('a :: field) > 0 ⟹ OFCLASS('a, field_prime_char_class)"
  by standard auto
```

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
  by (rule semiring_prime_charI) auto
instance poly :: ("{comm_semiring_prime_char,comm_semiring_1}") comm_semiring_prime_char
  by standard
instance poly :: ("{comm_ring_prime_char,comm_semiring_1}") comm_ring_prime_char
  by standard
instance poly :: ("{idom_prime_char,comm_semiring_1}") idom_prime_char
  by standard

instance fps :: ("{semiring_prime_char,comm_semiring_1}") semiring_prime_char
  by (rule semiring_prime_charI) auto
instance fps :: ("{comm_semiring_prime_char,comm_semiring_1}") comm_semiring_prime_char
  by standard
instance fps :: ("{comm_ring_prime_char,comm_semiring_1}") comm_ring_prime_char
  by standard
instance fps :: ("{idom_prime_char,comm_semiring_1}") idom_prime_char
  by standard

instance fls :: ("{semiring_prime_char,comm_semiring_1}") semiring_prime_char
```

```
  by (rule semiring_prime_charI) auto
instance fls :: ("{comm_semiring_prime_char,comm_semiring_1}") comm_semiring_prime_char
  by standard
instance fls :: ("{comm_ring_prime_char,comm_semiring_1}") comm_ring_prime_char
  by standard
instance fls :: ("{idom_prime_char,comm_semiring_1}") idom_prime_char
  by standard
instance fls :: ("{field_prime_char,comm_semiring_1}") field_prime_char
  by (rule field_prime_charI') auto
```

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 standard
  show "∃n>0. of_nat n = (0 :: 'a)"
    using assms prime_CHAR_semidom[where ?'a = 'a] finite_imp_CHAR_pos[OF
assms]
    by (intro exI[of _ "CHAR('a)"]) auto
qed fact+

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)}"
  using enum_finite_field by (simp add: eq_card_imp_inj_on)

end

instance mod_ring :: (prime_card) finite_field
  by standard simp_all

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
  interpret type_definition "Rep_mod_ring :: 'a mod_ring ⇒ int" Abs_mod_ring
"{0..<CARD('a)}"
    by (rule type_definition_mod_ring)
  have "enum_finite_field ` {..<CARD('a mod_ring)} = of_int_mod_ring `
int ` {..<CARD('a mod_ring)}"
```

```
    unfolding enum_finite_field_mod_ring_def by (simp add: image_image
o_def)
  also have "int ` {..<CARD('a mod_ring)} = {0..<int CARD('a mod_ring)}"
    by (simp add: image_atLeastZeroLessThan_int)
  also have "of_int_mod_ring ` ... = (Abs_mod_ring ` ... :: 'a mod_ring
set)"
    by (intro image_cong refl) (auto simp: of_int_mod_ring_def)
  also have "... = (UNIV :: 'a mod_ring set)"
    using Abs_image by simp
  finally show "enum_finite_field ` {..<CARD('a mod_ring)} = (UNIV :: 'a
mod_ring set)" .
qed

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 (cases "x = 0")
  case False
  let ?R = "⦇carrier = (UNIV :: 'a set), monoid.mult = (*), one = 1, zero
= 0, add = (+)⦈"
  interpret field "?R" rewrites "([^]?R) = (^)"
  proof -
    show "field ?R"
      by unfold_locales (auto simp: Units_def add_eq_0_iff ring_distribs
                              intro!: exI[of _ "inverse x" for x] left_inverse
right_inverse)
    have "x [^]?R n = x ^ n" for x n
      by (induction n) auto
    thus "([^]?R) = (^)"
      by blast
  qed

  note fin [intro] = finite_class.finite_UNIV[where ?'a = 'a]
  have "x ^ (CARD('a) - 1) * x = x ^ CARD('a)"
    using finite_UNIV_card_ge_0 power_minus_mult by blast
  also have "x ^ (CARD('a) - 1) = 1"
    using units_power_order_eq_one[of x] fin False
    by (simp add: field_Units)
  finally show ?thesis
```

```
    by simp
qed (use finite_class.finite_UNIV[where ?'a = 'a] in ‹auto simp: card_gt_0_iff›)

lemma finite_field_power_card_power_eq_same:
  fixes x :: "'a :: finite_field"
  assumes "m = CARD('a) ^ n"
  shows   "x ^ m = x"
  unfolding assms
  by (induction n) (simp_all add: finite_field_power_card_eq_same power_mult)


typedef (overloaded) 'a :: semiring_1 ring_char = "if CHAR('a) = 0 then
UNIV else {0..<CHAR('a)}"
  by auto

lemma CARD_ring_char [simp]: "CARD ('a :: semiring_1 ring_char) = CHAR('a)"
proof -
  let ?A = "if CHAR('a) = 0 then UNIV else {0..<CHAR('a)}"
  interpret type_definition "Rep_ring_char :: 'a ring_char ⇒ nat" Abs_ring_char
?A
    by (rule type_definition_ring_char)
  from card show ?thesis
    by auto
qed

instance ring_char :: (semiring_prime_char) nontriv
proof
  show "CARD('a ring_char) > 1"
    using prime_nat_iff by auto
qed

instance ring_char :: (semiring_prime_char) prime_card
proof
  from CARD_ring_char show "prime CARD('a ring_char)"
    by auto
qed

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)"
  by transfer auto

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)"
  by transfer auto

lemma of_nat_mod_CHAR [simp]: "of_nat (x mod CHAR('a :: semiring_1))
= (of_nat x :: 'a)"
```

```
  by (metis (no_types, opaque_lifting) comm_monoid_add_class.add_0 div_mod_decomp
        mult_zero_right of_nat_CHAR of_nat_add of_nat_mult)

lemma of_int_mod_CHAR [simp]: "of_int (x mod int CHAR('a :: ring_1))
= (of_int x :: 'a)"
  by (simp add: of_int_eq_iff_cong_CHAR)

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 (rule bij_betwI)
  show "(λv. ∑b∈B. v b *s b) ∈ (B →E UNIV) → local.span B"
    (is "?f ∈ _")
    by (auto intro: span_sum span_scale span_base)
  show "(λx. restrict (representation B x) B) ∈ local.span B → B →E
UNIV"
    (is "?g ∈ _") by auto
  show "?g (?f v) = v" if "v ∈ B →E UNIV" for v
  proof
    fix b :: 'b
    show "?g (?f v) b = v b"
    proof (cases "b ∈ B")
      case b: True
      have "?g (?f v) b = (∑i∈B. local.representation B (v i *s i) b)"
        using b by (subst representation_sum) (auto intro: span_scale
span_base)
      also have "... = (∑i∈B. v i * local.representation B i b)"
        by (intro sum.cong) (auto simp: representation_scale span_base)
      also have "... = (∑i∈{b}. v i * local.representation B i b)"
        by (intro sum.mono_neutral_right) (auto simp: representation_basis
b)
      also have "... = v b"
        by (simp add: representation_basis b)
      finally show "?g (?f v) b = v b" .
    qed (use that in auto)
  qed
  show "?f (?g v) = v" if "v ∈ span B" for v
    using that by (simp add: sum_representation_eq)
qed

lemma (in vector_space) card_span:
  assumes [simp]: "independent B" "finite B"
  shows   "card (span B) = CARD('a) ^ card B"
proof -
  have "card (B →E (UNIV :: 'a set)) = card (span B)"
    by (rule bij_betw_same_card, rule bij_betw_representation) fact+
  thus ?thesis
    by (simp add: card_PiE dim_span_eq_card_independent)
qed
```

```
lemma (in zero_neq_one) CARD_neq_1: "CARD('a) ≠ Suc 0"
proof
  assume "CARD('a) = Suc 0"
  have "{0, 1} ⊆ (UNIV :: 'a set)"
    by simp
  also have "is_singleton (UNIV :: 'a set)"
    by (simp add: is_singleton_altdef ‹CARD('a) = _›)
  then obtain x :: 'a where "UNIV = {x}"
    by (elim is_singletonE)
  finally have "0 = (1 :: 'a)"
    by blast
  thus False
    using zero_neq_one by contradiction
qed

theorem CARD_finite_field_is_CHAR_power: "∃n>0. CARD('a :: finite_field)
= CHAR('a) ^ n"
proof -
  define s :: "'a ring_char mod_ring ⇒ 'a ⇒ 'a" where
    "s = (λx y. of_int (to_int_mod_ring x) * y)"
  interpret vector_space s
    by unfold_locales (auto simp: s_def algebra_simps to_int_mod_ring_add
to_int_mod_ring_mult)
  obtain B where B: "independent B" "span B = UNIV"
    by (rule basis_exists[of UNIV]) auto
  have [simp]: "finite B"
    by simp
  have "card (span B) = CHAR('a) ^ card B"
    using B by (subst card_span) auto
  hence *: "CARD('a) = CHAR('a) ^ card B"
    using B by simp
  from * have "card B ≠ 0"
    by (auto simp: B(2) CARD_neq_1)
  with * show ?thesis
    by blast
qed
```

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 -
  interpret comm_semiring_prime_char
    by standard (auto intro!: exI[of _ "CHAR('a)"] assms)
  have "n > 0"
```

```
    unfolding n_def by simp
  have "(x + y) ^ n = (∑k≤n. of_nat (n choose k) * x ^ k * y ^ (n -
k))"
    by (rule binomial_ring)
  also have "... = (∑k∈{0,n}. of_nat (n choose k) * x ^ k * y ^ (n -
k))"
  proof (intro sum.mono_neutral_right ballI)
    fix k assume "k ∈ {..n} - {0, n}"
    hence k: "k > 0" "k < n"
      by auto
    have "CHAR('a) dvd (n choose k)"
      unfolding n_def
      by (rule dvd_choose_prime) (use k in ‹auto simp: n_def›)
    hence "of_nat (n choose k) = (0 :: 'a)"
      using of_nat_eq_0_iff_char_dvd by blast
    thus "of_nat (n choose k) * x ^ k * y ^ (n - k) = 0"
      by simp
  qed auto
  finally show ?thesis
    using ‹n > 0› by (simp add: add_ac)
qed

lemma (in comm_semiring_1) freshmans_dream':
  assumes [simp]: "prime CHAR('a)" and "m = CHAR('a) ^ n"
  shows "(x + y :: 'a) ^ m = x ^ m + y ^ m"
  unfolding assms(2)
proof (induction n)
  case (Suc n)
  have "(x + y) ^ (CHAR('a) ^ n * CHAR('a)) = ((x + y) ^ (CHAR('a) ^ n))
^ CHAR('a)"
    by (rule power_mult)
  thus ?case
    by (simp add: Suc.IH freshmans_dream Groups.mult_ac flip: power_mult)
qed auto

lemma (in comm_semiring_1) freshmans_dream_sum:
  fixes f :: "'b ⇒ 'a"
  assumes "prime CHAR('a)" and "n = CHAR('a)"
  shows "sum f A ^ n = sum (λi. f i ^ n) A"
  using assms
  by (induct A rule: infinite_finite_induct)
     (auto simp add: power_0_left freshmans_dream)

lemma (in comm_semiring_1) freshmans_dream_sum':
  fixes f :: "'b ⇒ 'a"
  assumes "prime CHAR('a)" "m = CHAR('a) ^ n"
  shows   "sum f A ^ m = sum (λi. f i ^ m) A"
  using assms
  by (induction A rule: infinite_finite_induct)
```

```
(auto simp: freshmans_dream' power_0_left)
```

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"
  by (simp_all add: inv_frob_def)

lemma (in semiring_prime_char) frob_0 [simp]: "frob (0 :: 'a) = 0"
  by (simp add: frob_def power_0_left)

lemma (in semiring_1) frob_1 [simp]: "frob 1 = 1"
  by (simp add: frob_def)

lemma (in comm_semiring_1) frob_mult: "frob (x * y) = frob x * frob (y
:: 'a)"
  by (simp add: frob_def power_mult_distrib)

lemma (in comm_semiring_1)
  frob_add: "prime CHAR('a) ⟹ frob (x + y :: 'a) = frob x + frob (y
:: 'a)"
  by (simp add: frob_def freshmans_dream)

lemma (in comm_ring_1) frob_uminus: "prime CHAR('a) ⟹ frob (-x :: 'a)
= -frob x"
proof -
  assume "prime CHAR('a)"
  hence "frob (-x) + frob x = 0"
    by (subst frob_add [symmetric]) (auto simp: frob_def power_0_left)
  thus ?thesis
    by (simp add: add_eq_0_iff)
qed

lemma (in comm_ring_prime_char) frob_diff:
  "prime CHAR('a) ⟹ frob (x - y :: 'a) = frob x - frob (y :: 'a)"
  using frob_add[of x "-y"] by (simp add: frob_uminus)

interpretation frob_sr: semiring_hom "frob :: 'a :: {comm_semiring_prime_char}
⇒ 'a"
  by standard (auto simp: frob_add frob_mult)

interpretation frob: ring_hom "frob :: 'a :: {comm_ring_prime_char} ⇒
```

```
'a"
  by standard auto

interpretation frob: field_hom "frob :: 'a :: {field_prime_char} ⇒ 'a"
  by standard auto

lemma frob_mod_ring' [simp]: "(x :: 'a :: prime_card mod_ring) ^ CARD('a)
= x"
  by (metis CARD_mod_ring finite_field_power_card_eq_same)

lemma frob_mod_ring [simp]: "frob (x :: 'a :: prime_card mod_ring) =
x"
  by (simp add: frob_def)

context semiring_1_no_zero_divisors
begin

lemma frob_eq_0D:
  "frob (x :: 'a) = 0 ⟹ x = 0"
  by (auto simp: frob_def)

lemma frob_eq_0_iff [simp]:
  "frob (x :: 'a) = 0 ⟷ x = 0 ∧ CHAR('a) > 0"
  by (auto simp: frob_def)

end


context idom_prime_char
begin

lemma inj_frob: "inj (frob :: 'a ⇒ 'a)"
proof
  fix x y :: 'a
  assume "frob x = frob y"
  hence "frob (x - y) = 0"
    by (simp add: frob_diff del: frob_eq_0_iff)
  thus "x = y"
    by simp
qed

lemma frob_eq_frob_iff [simp]:
  "frob (x :: 'a) = frob y ⟷ x = y"
  using inj_frob by (auto simp: inj_def)

lemma frob_eq_1_iff [simp]: "frob (x :: 'a) = 1 ⟷ x = 1"
  using frob_eq_frob_iff by fastforce

lemma inv_frob_frob [simp]: "inv_frob (frob (x :: 'a)) = x"
```

```
  by (simp add: inj_frob inv_frob_def)

lemma frob_inv_frob [simp]:
  assumes "x ∈ range frob"
  shows   "frob (inv_frob x) = (x :: 'a)"
  using assms by (auto simp: inj_frob inv_frob_def)

lemma inv_frob_eqI: "frob y = x ⟹ inv_frob x = y"
  using inv_frob_frob local.frob_def by force

lemma inv_frob_eq_0_iff [simp]: "inv_frob (x :: 'a) = 0 ⟷ x = 0"
  using inj_frob by (auto simp: inv_frob_def split: if_splits)

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"
  using surj_frob by blast

lemma inv_frob_eq_iff [simp]: "inv_frob (x :: 'a) = y ⟷ frob y = x"
  using frob_inv_frob inv_frob_frob by blast

end
```

The following type class describes a field with a surjective Frobenius endomorphism 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
  show "surj (frob :: 'a ⇒ 'a)"
    using inj_frob finite_UNIV by (simp add: finite_UNIV_inj_surj)
qed

end
```

```
lemma inv_frob_mod_ring [simp]: "inv_frob (x :: 'a :: prime_card mod_ring)
= x"
  by (auto simp: frob_def)

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
  by standard (auto simp: inv_frob_code_mod_ring_def)

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 goal_cases
  case (1 f)
  show ?case
  proof (cases "CHAR('a) > 0")
    case True
    from 1 obtain N where N: "f i = 0" if "i ≥ N" for i
      using cofinite_eq_sequentially eventually_sequentially by auto
    have "inv_frob (f (i * CHAR('a))) = 0" if "i ≥ N" for i
    proof -
      have "f (i * CHAR('a)) = 0"
      proof (rule N)
        show "N ≤ i * CHAR('a)"
          using that True
          by (metis One_nat_def Suc_leI le_trans mult.right_neutral mult_le_mono2)
```

```
      qed
      thus "inv_frob (f (i * CHAR('a))) = 0"
        by (auto simp: power_0_left)
    qed
    thus ?thesis using True
      unfolding cofinite_eq_sequentially eventually_sequentially by auto
  qed (use 1 in auto)
qed

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)))"
  using assms by transfer auto

lemma inv_frob_poly_0 [simp]: "inv_frob_poly 0 = 0"
  by transfer (auto simp: fun_eq_iff power_0_left)

lemma inv_frob_poly_1 [simp]: "inv_frob_poly 1 = 1"
  by transfer (auto simp: fun_eq_iff power_0_left)

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 (intro degree_le allI impI)
  fix i assume "Polynomial.degree p div CHAR('a) < i"
  hence "i * CHAR('a) > Polynomial.degree p"
    using assms div_less_iff_less_mult by blast
  thus "Polynomial.coeff (inv_frob_poly p) i = 0"
    by (simp add: coeff_eq_0 power_0_left assms)
qed

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 -
  have "h ^ CHAR('a) = (∑i≤Polynomial.degree h. Polynomial.monom (Polynomial.coeff
h i) i) ^ CHAR('a)"
    by (simp add: poly_as_sum_of_monoms)
  also have "... = (∑i≤Polynomial.degree h. (Polynomial.monom (Polynomial.coeff
```

```
h i) i) ^ CHAR('a))"
    by (simp add: freshmans_dream_sum prime_char)
  also have "... = (∑ i≤Polynomial.degree h. Polynomial.monom (Polynomial.coeff
h i ^ CHAR('a)) (CHAR('a)*i))"
  proof (rule sum.cong, rule)
    fix x assume x: "x ∈ {..Polynomial.degree h}"
    show "Polynomial.monom (Polynomial.coeff h x) x ^ CHAR('a) = Polynomial.monom
(Polynomial.coeff h x ^ CHAR('a)) (CHAR('a) * x)"
      by (unfold poly_eq_iff, auto simp add: monom_power)
  qed
  finally show ?thesis .
qed

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)"
  using prime_char
  by (subst poly_power_prime_char_as_sum_of_monoms, subst Polynomial.coeff_sum)
     (auto intro: le_degree simp: power_0_left)

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 -
  have "poly.coeff (y ^ CHAR('a)) i =
         (∑ j≤Polynomial.degree y. Polynomial.coeff (Polynomial.monom
(Polynomial.coeff y j ^ CHAR('a)) (CHAR('a) * j)) i)"
    by (subst poly_power_prime_char_as_sum_of_monoms, subst Polynomial.coeff_sum)
auto
  also have "... = (∑ j∈(if CHAR('a) dvd i ∧ i div CHAR('a) ≤ Polynomial.degree
y then {i div CHAR('a)} else {}).
                    Polynomial.coeff (Polynomial.monom (Polynomial.coeff
y j ^ CHAR('a)) (CHAR('a) * j)) i)"
    by (intro sum.mono_neutral_right) (use prime_char in auto)
  also have "... = (if CHAR('a) dvd i then poly.coeff y (i div CHAR('a))
^ CHAR('a) else 0)"
  proof (cases "CHAR('a) dvd i ∧ i div CHAR('a) > Polynomial.degree y")
    case True
    hence "Polynomial.coeff y (i div CHAR('a)) ^ CHAR('a) = 0"
      using prime_char by (simp add: coeff_eq_0 zero_power power_0_left)
    thus ?thesis
      by auto
  qed auto
  finally show ?thesis .
qed

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 unfold_locales
  have *: "class.semiring_1 (1 :: 'a) (*) (+) 0" ..
  have [simp]: "semiring_1.of_nat (1 :: 'a) (+) 0 = of_nat"
    by (auto simp: of_nat_def semiring_1.of_nat_def[OF *])
  thus "∃n>0. semiring_1.of_nat (1 :: 'a) (+) 0 n = 0"
    by (intro exI[of _ "CHAR('a)"]) (use pos_char in auto)
  show "semiring_1.semiring_char 1 (+) 0 TYPE('a) = CHAR('a)"
    by (simp add: fun_eq_iff semiring_char_def semiring_1.semiring_char_def[OF
*])
  show [simp]: "semiring_1.frob (1 :: 'a) (*) (+) 0 = frob"
    by (simp add: frob_def semiring_1.frob_def[OF *] fun_eq_iff
                  power.power_def power_def semiring_char_def semiring_1.semiring_char_def[
*])
  show "semiring_1.inv_frob (1 :: 'a) (*) (+) 0 = inv_frob"
    by (simp add: inv_frob_def semiring_1.inv_frob_def[OF *] fun_eq_iff)
qed

lemma inv_frob_poly_power': "inv_frob_poly (p ^ CHAR('a) :: 'a poly)
= p"
  using prime_CHAR_semidom[OF pos_char] pos_char
  by (auto simp: poly_eq_iff simp flip: frob_def)

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 -
  from assms(1) obtain q where q: "p = q ^ CHAR('a)"
    by (elim is_nth_powerE)
  thus ?thesis using assms
    by (simp add: q inv_frob_poly_power')
qed

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 -
  have *: "poly.coeff p n = 0" if n: "¬CHAR('a) dvd n" for n
  proof (cases "n = 0")
    case False
    have "poly.coeff (pderiv p) (n - 1) = of_nat n * poly.coeff p n"
      using False by (auto simp: coeff_pderiv)
    with assms and n show "poly.coeff p n = 0"
      by (auto simp: of_nat_eq_0_iff_char_dvd)
  qed (use that in auto)

  have **: "inv_frob_poly p ^ CHAR('a) = p"
  proof (rule poly_eqI)
    fix n :: nat
    show "poly.coeff (inv_frob_poly p ^ CHAR('a)) n = poly.coeff p n"
      using * CHAR_dvd_CARD[where ?'a = 'a]
      by (subst coeff_of_prime_char_power')
         (auto simp: poly_eq_iff frob_def [symmetric]
                     coeff_of_prime_char_power'[where ?'a = 'a] simp
flip: power_mult)
  qed

  show ?thesis
    by (subst **[symmetric]) auto
qed

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"
  by (induction xs) auto

lemma take_every_1 [simp]: "take_every (Suc 0) xs = xs"
  by (induction xs) auto

lemma int_length_take_every: "n > 0 ⟹ int (length (take_every n xs))
= ceiling (length xs / n)"
proof (induction n xs rule: take_every.induct)
  case (2 n x xs)
  show ?case
```

```
  proof (cases "Suc (length xs) ≥ n")
    case True
    thus ?thesis using 2
      by (auto simp: dvd_imp_le of_nat_diff diff_divide_distrib split:
if_splits)
  next
    case False
    hence "⌈(1 + real (length xs)) / real n⌉ = 1"
      by (intro ceiling_unique) auto
    thus ?thesis using False
      by auto
  qed
qed auto

lemma length_take_every:
  "n > 0 ⟹ length (take_every n xs) = nat (ceiling (length xs / n))"
  using int_length_take_every[of n xs] by simp

lemma take_every_nth [simp]:
  "n > 0 ⟹ i < length (take_every n xs) ⟹ take_every n xs ! i = xs
! (n * i)"
proof (induction n xs arbitrary: i rule: take_every.induct)
  case (2 n x xs i)
  show ?case
  proof (cases i)
    case (Suc j)
    have "n - Suc 0 ≤ length xs"
      using Suc "2.prems" nat_le_linear by force
    hence "drop (n - Suc 0) xs ! (n * j) = xs ! (n - 1 + n * j)"
      using Suc by (subst nth_drop) auto
    also have "n - 1 + n * j = n + n * j - 1"
      using ‹n > 0› by linarith
    finally show ?thesis
      using "2.IH"[of j] "2.prems" Suc by simp
  qed auto
qed auto

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 (rule coeffs_eqI)
  fix n :: nat
  show "Polynomial.coeff p n = nth_default 0 (strip_while ((=) 0) xs)
n"
    using assms
    by (metis coeff_0 coeff_Poly_eq coeffs_Poly le_degree nth_default_coeffs_eq

      nth_default_eq_dflt_iff nth_default_nth order_le_less_trans)
```

```
qed auto
```

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 (cases "CHAR('a) = 0 ∨ p = 0")
  case False
  from False have "p ≠ 0"
    by auto
  have "Polynomial.coeffs (inv_frob_poly p) =
          strip_while ((=) 0) (map inv_frob (take_every CHAR('a) (Polynomial.coeffs
p)))"
  proof (rule coeffs_eq_strip_whileI)
    fix i assume i: "i < length (map inv_frob (take_every CHAR('a) (Polynomial.coeffs
p)))"
    show "Polynomial.coeff (inv_frob_poly p) i = map inv_frob (take_every
CHAR('a) (Polynomial.coeffs p)) ! i"
    proof -
      have "i < length (take_every CHAR('a) (Polynomial.coeffs p))"
        using i by simp
      also have "length (take_every CHAR('a) (Polynomial.coeffs p)) =
                 nat ⌈(Polynomial.degree p + 1) / real CHAR('a)⌉"
        using False CHAR_pos[where ?'a = 'a]
        by (simp add: length_take_every length_coeffs)
      finally have "i < real (Polynomial.degree p + 1) / real CHAR('a)"
        by linarith
      hence "real i * real CHAR('a) < real (Polynomial.degree p + 1)"
        using False CHAR_pos[where ?'a = 'a] by (simp add: field_simps)
      hence "i * CHAR('a) ≤ Polynomial.degree p"
        unfolding of_nat_mult [symmetric] by linarith
      hence "Polynomial.coeffs p ! (i * CHAR('a)) = Polynomial.coeff p
(i * CHAR('a))"
        using False by (intro coeffs_nth) (auto simp: length_take_every)
      thus ?thesis using False i CHAR_pos[where ?'a = 'a]
        by (auto simp: nth_default_def mult.commute)
    qed
  next
    assume nz: "inv_frob_poly p ≠ 0"
    have "Polynomial.degree (inv_frob_poly p) ≤ Polynomial.degree p div
CHAR('a)"
      by (rule degree_inv_frob_poly_le) (fact CHAR_pos)
    also have "... < nat ⌈(real (Polynomial.degree p) + 1) / real CHAR('a)⌉"
      using CHAR_pos[where ?'a = 'a]
      by (metis div_less_iff_less_mult linorder_not_le nat_le_real_less
```

```
of_nat_0_less_iff
                of_nat_ceiling of_nat_mult pos_less_divide_eq)
    also have "... = length (take_every CHAR('a) (Polynomial.coeffs p))"
      using CHAR_pos[where ?'a = 'a] ‹p ≠ 0› by (simp add: length_take_every
length_coeffs add_ac)
    finally show "length (map inv_frob (take_every CHAR('a) (Polynomial.coeffs
p))) > Polynomial.degree (inv_frob_poly p)"
      by simp_all
  qed
  also have "strip_while ((=) 0) (map inv_frob (take_every CHAR('a) (Polynomial.coeffs
p))) =
             map inv_frob (strip_while ((=) 0 ∘ inv_frob) (take_every
CHAR('a) (Polynomial.coeffs p)))"
    by (rule strip_while_map)
  also have "(=) 0 ∘ inv_frob = (=) (0 :: 'a)"
    by (auto simp: fun_eq_iff)
  finally show ?thesis
    using False by metis
qed auto
```

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
  by standard auto
end

context surj_frob
begin
subclass perfect_field
  by standard auto
end

theorem irreducible_imp_pderiv_nonzero:
  assumes "irreducible (p :: 'a :: perfect_field poly)"
  shows   "pderiv p ≠ 0"
proof (cases "CHAR('a) = 0")
```

```
  case True
  interpret A: semiring_1 "1 :: 'a" "(*)" "(+)" "0 :: 'a" ..
  have *: "class.semiring_1 (1 :: 'a) (*) (+) 0" ..
  interpret A: field_char_0 "(/)" inverse "(*)" "1 :: 'a" "(+)" 0 "(-)"
uminus
  proof
    have "inj (of_nat :: nat ⇒ 'a)"
      by (auto simp: inj_on_def of_nat_eq_iff_cong_CHAR True)
    also have "of_nat = semiring_1.of_nat (1 :: 'a) (+) 0"
      by (simp add: of_nat_def [abs_def] semiring_1.of_nat_def [OF *,
abs_def])
    finally show "inj ..." .
  qed

  show ?thesis
  proof
    assume "pderiv p = 0"
    hence **: "poly.coeff p (Suc n) = 0" for n
      by (auto simp: poly_eq_iff coeff_pderiv of_nat_eq_0_iff_char_dvd
True simp del: of_nat_Suc)
    have "poly.coeff p n = 0" if "n > 0" for n
      using **[of "n - 1"] that by (cases n) auto
    hence "Polynomial.degree p = 0"
      by force
    thus False
      using assms by force
  qed

next
  case False
  hence [simp]: "surj (frob :: 'a ⇒ 'a)"
    by (meson perfect_field)

  interpret A: field_prime_char "(/)" inverse "(*)" "1 :: 'a" "(+)" 0 "(-)"
uminus
  proof
    have *: "class.semiring_1 1 (*) (+) (0 :: 'a)" ..
    have "semiring_1.of_nat 1 (+) (0 :: 'a) = of_nat"
      by (simp add: fun_eq_iff of_nat_def semiring_1.of_nat_def[OF *])
    thus "∃n>0. semiring_1.of_nat 1 (+) 0 n = (0 :: 'a)"
      by (intro exI[of _ "CHAR('a)"]) (use False in auto)
  qed

  show ?thesis
  proof
    assume "pderiv p = 0"
    hence "is_nth_power CHAR('a) p"
      using pderiv_eq_0_imp_nth_power[of p] surj_frob False by simp
    then obtain q where "p = q ^ CHAR('a)"
```

```
      by (elim is_nth_powerE)
    with assms show False
      by auto
  qed
qed

corollary irreducible_imp_separable:
  assumes "irreducible (p :: 'a :: perfect_field poly)"
  shows   "coprime p (pderiv p)"
proof (rule coprimeI)
  fix q assume q: "q dvd p" "q dvd pderiv p"
  have "¬p dvd q"
  proof
    assume "p dvd q"
    hence "p dvd pderiv p"
      using q dvd_trans by blast
    hence "Polynomial.degree p ≤ Polynomial.degree (pderiv p)"
      by (rule dvd_imp_degree_le) (use assms irreducible_imp_pderiv_nonzero
in auto)
    also have "... ≤ Polynomial.degree p - 1"
      using degree_pderiv_le by auto
    finally have "Polynomial.degree p = 0"
      by simp
    with assms show False
      using irreducible_imp_pderiv_nonzero is_unit_iff_degree by blast
  qed
  with ‹q dvd p› show "is_unit q"
    using assms comm_semiring_1_class.irreducibleD' by blast
qed

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 -
  define f where "f = (λi. if i = 0 then -x else if i = n then 1 else
0)"
  have "∃x. (∑k≤n. f k * x ^ k) = 0"
    by (rule alg_closed) (use assms in ‹auto simp: f_def›)
```

```
  also have "(λx. ∑k≤n. f k * x ^ k) = (λx. ∑k∈{0,n}. f k * x ^ k)"
    by (intro ext sum.mono_neutral_right) (auto simp: f_def)
  finally show "∃y. y ^ n = x"
    using assms by (simp add: f_def)
qed


context alg_closed_field
begin

lemma alg_closed_surj_frob:
  assumes "CHAR('a) > 0"
  shows   "surj (frob :: 'a ⇒ 'a)"
proof -
  show "surj (frob :: 'a ⇒ 'a)"
  proof safe
    fix x :: 'a
    obtain y where "y ^ CHAR('a) = x"
      using nth_root_exists CHAR_pos assms by blast
    hence "frob y = x"
      using CHAR_pos by (simp add: frob_def)
    thus "x ∈ range frob"
      by (metis rangeI)
  qed auto
qed

sublocale perfect_field
  by standard (use alg_closed_surj_frob in auto)

end



lemma fpxs_const_eq_0_iff [simp]: "fpxs_const x = 0 ⟷ x = 0"
  by (metis fpxs_const_0 fpxs_const_eq_iff)

lemma semiring_char_fpxs [simp]: "CHAR('a :: comm_semiring_1 fpxs) =
CHAR('a)"
  by (rule CHAR_eqI; unfold of_nat_fpxs_eq) (auto simp: of_nat_eq_0_iff_char_dvd)

instance fpxs :: ("{semiring_prime_char,comm_semiring_1}") semiring_prime_char
  by (rule semiring_prime_charI) auto
instance fpxs :: ("{comm_semiring_prime_char,comm_semiring_1}") comm_semiring_prime_char
  by standard
instance fpxs :: ("{comm_ring_prime_char,comm_semiring_1}") comm_ring_prime_char
  by standard
instance fpxs :: ("{idom_prime_char,comm_semiring_1}") idom_prime_char
  by standard
instance fpxs :: ("field_prime_char") field_prime_char
```

```
  by standard auto

end
```

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 -
  have "∃y. x + y = 0" for x :: 'a
    using add.right_inverse by blast
  thus ?thesis
    unfolding ring_of_type_algebra_def using add.right_inverse
    by unfold_locales (auto simp:algebra_simps Units_def)
qed

lemma (in comm_ring_1) cring_from_type_algebra [intro]:
  "cring (ring_of_type_algebra :: 'a ring)"
proof -
  have "∃y. x + y = 0" for x :: 'a
    using add.right_inverse by blast
  thus ?thesis
    unfolding ring_of_type_algebra_def using add.right_inverse
    by unfold_locales (auto simp:algebra_simps Units_def)
qed

lemma (in Fields.field) field_from_type_algebra [intro]:
  "field (ring_of_type_algebra :: 'a ring)"
proof -
  have "∃y. x + y = 0" for x :: 'a
    using add.right_inverse by blast

  moreover have "x ≠ 0 ⟹ ∃y. x * y = 1" for x :: 'a
    by (rule exI[of _ "inverse x"]) auto
```

```
  ultimately show ?thesis
    unfolding ring_of_type_algebra_def using add.right_inverse
    by unfold_locales (auto simp:algebra_simps Units_def)
qed
```

2.1 Definition

```
typedef (overloaded) 'a :: field alg_closure =
  "carrier (field.alg_closure (ring_of_type_algebra :: 'a :: field ring))"
proof -
  define K where "K ≡ (ring_of_type_algebra :: 'a ring)"
  define L where "L ≡ field.alg_closure K"

  interpret K: field K
    unfolding K_def by rule

  interpret algebraic_closure L "range K.indexed_const"
  proof -
    have *: "carrier K = UNIV"
      by (auto simp: K_def ring_of_type_algebra_def)
    show "algebraic_closure L (range K.indexed_const)"
      unfolding * [symmetric] L_def by (rule K.alg_closureE)
  qed

  show "∃x. x ∈ carrier L"
    using zero_closed by blast
qed

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
  unfolding K_def by rule

interpretation algebraic_closure L "range K.indexed_const"
proof -
  have *: "carrier K = UNIV"
    by (auto simp: K_def ring_of_type_algebra_def)
  show "algebraic_closure L (range K.indexed_const)"
    unfolding * [symmetric] L_def by (rule K.alg_closureE)
qed
```

lift_definition *zero_alg_closure :: "'a alg_closure"* **is** *"ring.zero L"*
  by *(fold K_def, fold L_def) (rule ring_simprules)*

lift_definition *one_alg_closure :: "'a alg_closure"* **is** *"monoid.one L"*
  by *(fold K_def, fold L_def) (rule ring_simprules)*

lift_definition *plus_alg_closure :: "'a alg_closure ⇒ 'a alg_closure ⇒ 'a alg_closure"*
  is *"ring.add L"*
  by *(fold K_def, fold L_def) (rule ring_simprules)*

lift_definition *minus_alg_closure :: "'a alg_closure ⇒ 'a alg_closure ⇒ 'a alg_closure"*
  is *"a_minus L"*
  by *(fold K_def, fold L_def) (rule ring_simprules)*

lift_definition *times_alg_closure :: "'a alg_closure ⇒ 'a alg_closure ⇒ 'a alg_closure"*
  is *"monoid.mult L"*
  by *(fold K_def, fold L_def) (rule ring_simprules)*

lift_definition *uminus_alg_closure :: "'a alg_closure ⇒ 'a alg_closure"*
  is *"a_inv L"*
  by *(fold K_def, fold L_def) (rule ring_simprules)*

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"*
  by *(fold K_def, fold L_def) (auto simp: field_Units)*

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)"*
  by *(fold K_def, fold L_def) (auto simp: field_Units)*

end

instance proof -
  define *K* **where** *"K ≡ (ring_of_type_algebra :: 'a ring)"*
  define *L* **where** *"L ≡ field.alg_closure K"*

  interpret *K: field K*
    unfolding *K_def* **by** *rule*

  interpret *algebraic_closure L "range K.indexed_const"*
  proof -
    have **: "carrier K = UNIV"*
      by *(auto simp: K_def ring_of_type_algebra_def)*

```
  show "algebraic_closure L (range K.indexed_const)"
    unfolding * [symmetric] L_def by (rule K.alg_closureE)
qed

show "OFCLASS('a alg_closure, field_class)"
proof (standard, goal_cases)
  case 1
  show ?case
    by (transfer, fold K_def, fold L_def) (rule m_assoc)
next
  case 2
  show ?case
    by (transfer, fold K_def, fold L_def) (rule m_comm)
next
  case 3
  show ?case
    by (transfer, fold K_def, fold L_def) (rule l_one)
next
  case 4
  show ?case
    by (transfer, fold K_def, fold L_def) (rule a_assoc)
next
  case 5
  show ?case
    by (transfer, fold K_def, fold L_def) (rule a_comm)
next
  case 6
  show ?case
    by (transfer, fold K_def, fold L_def) (rule l_zero)
next
  case 7
  show ?case
    by (transfer, fold K_def, fold L_def) (rule ring_simprules)
next
  case 8
  show ?case
    by (transfer, fold K_def, fold L_def) (rule ring_simprules)
next
  case 9
  show ?case
    by (transfer, fold K_def, fold L_def) (rule ring_simprules)
next
  case 10
  show ?case
    by (transfer, fold K_def, fold L_def) (rule zero_not_one)
next
  case 11
  thus ?case
    by (transfer, fold K_def, fold L_def) (auto simp: field_Units)
```

```
  next
    case 12
    thus ?case
      by (transfer, fold K_def, fold L_def) auto
  next
    case 13
    thus ?case
      by transfer auto
  qed
qed

end
```

2.2 The algebraic closure is algebraically closed

```
instance alg_closure :: (field) alg_closed_field
proof
  define K where "K ≡ (ring_of_type_algebra :: 'a ring)"
  define L where "L ≡ field.alg_closure K"

  interpret K: field K
    unfolding K_def by rule

  interpret algebraic_closure L "range K.indexed_const"
  proof -
    have *: "carrier K = UNIV"
      by (auto simp: K_def ring_of_type_algebra_def)
    show "algebraic_closure L (range K.indexed_const)"
      unfolding * [symmetric] L_def by (rule K.alg_closureE)
  qed

  have [simp]: "Rep_alg_closure x ∈ carrier L" for x
    using Rep_alg_closure[of x] by (simp only: L_def K_def)

  have [simp]: "Rep_alg_closure x = Rep_alg_closure y ⟷ x = y" for
x y
    by (simp add: Rep_alg_closure_inject)
  have [simp]: "Rep_alg_closure x = 0_L ⟷ x = 0" for x
  proof -
    have "Rep_alg_closure x = Rep_alg_closure 0 ⟷ x = 0"
      by simp
    also have "Rep_alg_closure 0 = 0_L"
      by (simp add: zero_alg_closure.rep_eq L_def K_def)
    finally show ?thesis .
  qed

  have [simp]: "Rep_alg_closure (x ^ n) = Rep_alg_closure x [^]_L n"
    for x :: "'a alg_closure" and n
    by (induction n)
```

```
      (auto simp: one_alg_closure.rep_eq times_alg_closure.rep_eq m_comm
            simp flip: L_def K_def)
  have [simp]: "Rep_alg_closure (Abs_alg_closure x) = x" if "x ∈ carrier
L" for x
    using that unfolding L_def K_def by (rule Abs_alg_closure_inverse)

  show "∃x. poly p x = 0" if p: "monic p" "Polynomial.degree p > 0" for
p :: "'a alg_closure poly"
  proof -
    define P where "P = rev (map Rep_alg_closure (Polynomial.coeffs p))"
    have deg: "Polynomials.degree P = Polynomial.degree p"
      by (auto simp: P_def degree_eq_length_coeffs)
    have carrier_P: "P ∈ carrier (poly_ring L)"
      by (auto simp: univ_poly_def polynomial_def P_def hd_map hd_rev
last_map
                  last_coeffs_eq_coeff_degree)
    hence "splitted P"
      using roots_over_carrier by blast
    hence "roots P ≠ {#}"
      unfolding splitted_def using deg p by auto
    then obtain x where "x ∈# roots P"
      by blast
    hence x: "is_root P x"
      using roots_mem_iff_is_root[OF carrier_P] by auto
    hence [simp]: "x ∈ carrier L"
      by (auto simp: is_root_def)
    define x' where "x' = Abs_alg_closure x"
    define xs where "xs = rev (coeffs p)"

    have "cr_alg_closure (eval (map Rep_alg_closure xs) x) (poly (Poly
(rev xs)) x')"
      by (induction xs)
         (auto simp flip: K_def L_def simp: cr_alg_closure_def
                 zero_alg_closure.rep_eq plus_alg_closure.rep_eq
                 times_alg_closure.rep_eq Poly_append poly_monom
                 a_comm m_comm x'_def)
    also have "map Rep_alg_closure xs = P"
      by (simp add: xs_def P_def rev_map)
    also have "Poly (rev xs) = p"
      by (simp add: xs_def)
    finally have "poly p x' = 0"
      using x by (auto simp: is_root_def cr_alg_closure_def)
    thus "∃x. poly p x = 0" ..
  qed
qed
```

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
  unfolding K_def by rule

interpretation algebraic_closure L "range K.indexed_const"
proof -
  have *: "carrier K = UNIV"
    by (auto simp: K_def ring_of_type_algebra_def)
  show "algebraic_closure L (range K.indexed_const)"
    unfolding * [symmetric] L_def by (rule K.alg_closureE)
qed

lemma alg_closure_hom: "K.indexed_const ∈ Ring.ring_hom K L"
  unfolding L_def using K.alg_closureE(2) .

lift_definition to_ac :: "'a :: field ⇒ 'a alg_closure"
  is "ring.indexed_const K"
  by (fold K_def, fold L_def) (use mem_carrier in blast)

lemma to_ac_0 [simp]: "to_ac (0 :: 'a) = 0"
proof -
  have "to_ac (0_K) = 0"
  proof (transfer fixing: K, fold K_def, fold L_def)
    show "K.indexed_const 0_K = 0_L"
      using Ring.ring_hom_zero[OF alg_closure_hom] K.ring_axioms is_ring
      by simp
  qed
  thus ?thesis
    by (simp add: K_def ring_of_type_algebra_def)
qed

lemma to_ac_1 [simp]: "to_ac (1 :: 'a) = 1"
proof -
  have "to_ac (1_K) = 1"
  proof (transfer fixing: K, fold K_def, fold L_def)
    show "K.indexed_const 1_K = 1_L"
      using Ring.ring_hom_one[OF alg_closure_hom] K.ring_axioms is_ring
      by simp
  qed
  thus ?thesis
    by (simp add: K_def ring_of_type_algebra_def)
qed

lemma to_ac_add [simp]: "to_ac (x + y :: 'a) = to_ac x + to_ac y"
proof -
```

```
  have "to_ac (x ⊕K y) = to_ac x + to_ac y"
  proof (transfer fixing: K x y, fold K_def, fold L_def)
    show "K.indexed_const (x ⊕K y) = K.indexed_const x ⊕L K.indexed_const
y"
      using Ring.ring_hom_add[OF alg_closure_hom, of x y] K.ring_axioms
is_ring
      by (simp add: K_def ring_of_type_algebra_def)
  qed
  thus ?thesis
    by (simp add: K_def ring_of_type_algebra_def)
qed

lemma to_ac_minus [simp]: "to_ac (-x :: 'a) = -to_ac x"
  using to_ac_add to_ac_0 add_eq_0_iff by metis

lemma to_ac_diff [simp]: "to_ac (x - y :: 'a) = to_ac x - to_ac y"
  using to_ac_add[of x "-y"] by simp

lemma to_ac_mult [simp]: "to_ac (x * y :: 'a) = to_ac x * to_ac y"
proof -
  have "to_ac (x ⊗K y) = to_ac x * to_ac y"
  proof (transfer fixing: K x y, fold K_def, fold L_def)
    show "K.indexed_const (x ⊗K y) = K.indexed_const x ⊗L K.indexed_const
y"
      using Ring.ring_hom_mult[OF alg_closure_hom, of x y] K.ring_axioms
is_ring
      by (simp add: K_def ring_of_type_algebra_def)
  qed
  thus ?thesis
    by (simp add: K_def ring_of_type_algebra_def)
qed

lemma to_ac_inverse [simp]: "to_ac (inverse x :: 'a) = inverse (to_ac
x)"
  using to_ac_mult[of x "inverse x"] to_ac_1 to_ac_0
  by (metis divide_self_if field_class.field_divide_inverse field_class.field_inverse_zero
inverse_unique)

lemma to_ac_divide [simp]: "to_ac (x / y :: 'a) = to_ac x / to_ac y"
  using to_ac_mult[of x "inverse y"] to_ac_inverse[of y]
  by (simp add: field_class.field_divide_inverse)

lemma to_ac_power [simp]: "to_ac (x ^ n) = to_ac x ^ n"
  by (induction n) auto

lemma to_ac_of_nat [simp]: "to_ac (of_nat n) = of_nat n"
  by (induction n) auto

lemma to_ac_of_int [simp]: "to_ac (of_int n) = of_int n"
```

```
  by (induction n) auto

lemma to_ac_numeral [simp]: "to_ac (numeral n) = numeral n"
  using to_ac_of_nat[of "numeral n"] by (simp del: to_ac_of_nat)

lemma to_ac_sum: "to_ac (∑x∈A. f x) = (∑x∈A. to_ac (f x))"
  by (induction A rule: infinite_finite_induct) auto

lemma to_ac_prod: "to_ac (∏x∈A. f x) = (∏x∈A. to_ac (f x))"
  by (induction A rule: infinite_finite_induct) auto

lemma to_ac_sum_list: "to_ac (sum_list xs) = (∑x←xs. to_ac x)"
  by (induction xs) auto

lemma to_ac_prod_list: "to_ac (prod_list xs) = (∏x←xs. to_ac x)"
  by (induction xs) auto

lemma to_ac_sum_mset: "to_ac (sum_mset xs) = (∑x∈#xs. to_ac x)"
  by (induction xs) auto

lemma to_ac_prod_mset: "to_ac (prod_mset xs) = (∏x∈#xs. to_ac x)"
  by (induction xs) auto

end

lemma (in ring) indexed_const_eq_iff [simp]:
  "indexed_const x = (indexed_const y :: 'c multiset ⇒ 'a) ⟷ x = y"
proof
  assume "indexed_const x = (indexed_const y :: 'c multiset ⇒ 'a)"
  hence "indexed_const x ({#} :: 'c multiset) = indexed_const y ({#} ::
'c multiset)"
    by metis
  thus "x = y"
    by (simp add: indexed_const_def)
qed auto

lemma inj_to_ac: "inj to_ac"
  by (transfer, intro injI, subst (asm) ring.indexed_const_eq_iff) auto

lemma to_ac_eq_iff [simp]: "to_ac x = to_ac y ⟷ x = y"
  using inj_to_ac by (auto simp: inj_on_def)

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"
  using to_ac_eq_iff to_ac_0 to_ac_1 by metis+
```

```
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"
  unfolding of_ac_def by (meson inj_to_ac inv_f_f range_eqI)

lemma of_ac_0 [simp]: "of_ac 0 = 0"
  and of_ac_1 [simp]: "of_ac 1 = 1"
  by (rule of_ac_eqI; simp; fail)+

lemma of_ac_to_ac [simp]: "of_ac (to_ac x) = x"
  by (rule of_ac_eqI) auto

lemma to_ac_of_ac: "x ∈ range to_ac ⟹ to_ac (of_ac x) = x"
  by auto


lemma CHAR_alg_closure [simp]:
  "CHAR('a :: field alg_closure) = CHAR('a)"
proof (rule CHAR_eqI)
  show "of_nat CHAR('a) = (0 :: 'a alg_closure)"
    by (metis of_nat_CHAR to_ac_0 to_ac_of_nat)
next
  show "CHAR('a) dvd n" if "of_nat n = (0 :: 'a alg_closure)" for n
    using that by (metis of_nat_eq_0_iff_char_dvd to_ac_eq_0_iff' to_ac_of_nat)
qed

instance alg_closure :: (field_char_0) field_char_0
proof
  show "inj (of_nat :: nat ⇒ 'a alg_closure)"
    by (metis injD inj_of_nat inj_on_def inj_to_ac to_ac_of_nat)
qed


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 -
  define K where "K ≡ (ring_of_type_algebra :: 'a ring)"
  define L where "L ≡ field.alg_closure K"

  interpret K: field K
    unfolding K_def by rule

  interpret algebraic_closure L "range K.indexed_const"
  proof -
    have *: "carrier K = UNIV"
      by (auto simp: K_def ring_of_type_algebra_def)
    show "algebraic_closure L (range K.indexed_const)"
      unfolding * [symmetric] L_def by (rule K.alg_closureE)
  qed

  let ?K = "range K.indexed_const"
  have sr: "subring ?K L"
    by (rule subring_axioms)
  define x' where "x' = Rep_alg_closure x"
  have "x' ∈ carrier L"
    unfolding x'_def L_def K_def by (rule Rep_alg_closure)
  hence alg: "(algebraic over range K.indexed_const) x'"
    using algebraic_extension by blast
  then obtain p where p: "p ∈ carrier (?K[X]_L)" "p ≠ []" "eval p x'
= 0_L"
    using algebraicE[OF sr ‹x' ∈ carrier L› alg] by blast

  have [simp]: "Rep_alg_closure x ∈ carrier L" for x
    using Rep_alg_closure[of x] by (simp only: L_def K_def)
  have [simp]: "Abs_alg_closure x = 0 ⟷ x = 0_L" if "x ∈ carrier L"
for x
    using that unfolding L_def K_def
    by (metis Abs_alg_closure_inverse zero_alg_closure.rep_eq zero_alg_closure_def)
  have [simp]: "Rep_alg_closure (x ^ n) = Rep_alg_closure x [^]_L n"
    for x :: "'a alg_closure" and n
    by (induction n)
       (auto simp: one_alg_closure.rep_eq times_alg_closure.rep_eq m_comm
             simp flip: L_def K_def)
  have [simp]: "Rep_alg_closure (Abs_alg_closure x) = x" if "x ∈ carrier
L" for x
    using that unfolding L_def K_def by (rule Abs_alg_closure_inverse)
  have [simp]: "Rep_alg_closure x = 0_L ⟷ x = 0" for x
```

```
    by (metis K_def L_def Rep_alg_closure_inverse zero_alg_closure.rep_eq)

  define p' where "p' = Poly (map Abs_alg_closure (rev p))"
  have "p' ≠ 0"
  proof
    assume "p' = 0"
    then obtain n where n: "map Abs_alg_closure (rev p) = replicate n
0"
      by (auto simp: p'_def Poly_eq_0)
    with ‹p ≠ []› have "n > 0"
      by (auto intro!: Nat.gr0I)
    have "last (map Abs_alg_closure (rev p)) = 0"
      using ‹n > 0› by (subst n) auto
    moreover have "Polynomials.lead_coeff p ≠ 0_L" "Polynomials.lead_coeff
p ∈ carrier L"
      using p ‹p ≠ []› local.subset
      by (fastforce simp: polynomial_def univ_poly_def)+
    ultimately show False
      using ‹p ≠ []› by (auto simp: last_map last_rev)
  qed

  have "set p ⊆ carrier L"
    using local.subset p by (auto simp: univ_poly_def polynomial_def)
  hence "cr_alg_closure (eval p x') (poly p' x)"
    unfolding p'_def
    by (induction p)
       (auto simp flip: K_def L_def simp: cr_alg_closure_def
              zero_alg_closure.rep_eq plus_alg_closure.rep_eq
              times_alg_closure.rep_eq Poly_append poly_monom
              a_comm m_comm x'_def)
  hence "poly p' x = 0"
    using p by (auto simp: cr_alg_closure_def x'_def)

  have coeff_p': "Polynomial.coeff p' i ∈ range to_ac" for i
  proof (cases "i ≥ length p")
    case False
    have "Polynomial.coeff p' i = Abs_alg_closure (rev p ! i)"
      unfolding p'_def using False
      by (auto simp: nth_default_def)
    moreover have "rev p ! i ∈ ?K"
      using p(1) False by (auto simp: univ_poly_def polynomial_def rev_nth)
    ultimately show ?thesis
      unfolding to_ac.abs_eq K_def by fastforce
  qed (auto simp: p'_def nth_default_def)


  define p'' where "p'' = map_poly of_ac p'"
  have p'_eq: "p' = map_poly to_ac p''"
    by (rule poly_eqI) (auto simp: coeff_map_poly p''_def to_ac_of_ac[OF
```

```
coeff_p'])

  interpret to_ac: map_poly_inj_comm_ring_hom "to_ac :: 'a ⇒ 'a alg_closure"
    by unfold_locales auto

  show ?thesis
  proof (rule that)
    show "p'' ≠ 0"
      using ‹p' ≠ 0› by (auto simp: p'_eq)
  next
    show "poly (map_poly to_ac p'') x = 0"
      using ‹poly p' x = 0› by (simp add: p'_eq)
  qed
qed


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
  by standard
    (simp_all add: dvd_field_iff field_split_simps split: if_splits)

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 by standard (simp_all add: gcd_alg_closure_def lcm_alg_closure_def
Gcd_alg_closure_def Lcm_alg_closure_def)

end

instance alg_closure :: (field) semiring_gcd_mult_normalize
  ..

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
..
```

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* ∈ *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 ≤ m < p* since *X - α* is prime.

However, the coefficient of X^{m-1} in $(X - \alpha)^m$ is *-mα*, and since *Q ∈ K[X]* we must have *-mα ∈ K* and therefore *α ∈ 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 (cases "CHAR('a) = 0")
  case False
  let ?p = "CHAR('a)"
  from False have "Factorial_Ring.prime ?p"
    by (simp add: prime_CHAR_semidom)
  hence "?p > 1"
    using prime_gt_1_nat by blast
  note p = ‹Factorial_Ring.prime ?p› ‹?p > 1›

  interpret to_ac: map_poly_inj_comm_ring_hom "to_ac :: 'a ⇒ 'a alg_closure"
    by unfold_locales auto

  have "surj (frob :: 'a ⇒ 'a)"
  proof safe
    fix c :: 'a
    obtain α :: "'a alg_closure" where α: "α ^ ?p = to_ac c"
      using p nth_root_exists[of ?p "to_ac c"] by auto
    define P where "P = Polynomial.monom 1 ?p + [:-c:]"
    define P' where "P' = map_poly to_ac P"
    have deg: "Polynomial.degree P = ?p"
      unfolding P_def using p by (subst degree_add_eq_left) (auto simp:
degree_monom_eq)

    have "[:-α, 1:] ^ ?p = ([:0, 1:] + [:-α:]) ^ ?p"
      by (simp add: one_pCons)
    also have "... = [:0, 1:] ^ ?p - [:α^?p:]"
      using p by (subst freshmans_dream) (auto simp: poly_const_pow minus_power_prime_CHAR)
    also have "α ^ ?p = to_ac c"
      by (simp add: α)
    also have "[:0, 1:] ^ CHAR('a) - [:to_ac c:] = P'"
      by (simp add: P_def P'_def to_ac.hom_add to_ac.hom_power
                 to_ac.base.map_poly_pCons_hom monom_altdef)
    finally have eq: "P' = [:-α, 1:] ^ ?p" ..

    have "¬is_unit P" "P ≠ 0"
      using deg p by auto
    then obtain Q where Q: "Factorial_Ring.prime Q" "Q dvd P"
      by (metis prime_divisor_exists)
    have "monic Q"
      using unit_factor_prime[OF Q(1)] by (auto simp: unit_factor_poly_def
one_pCons)

    from Q(2) have "map_poly to_ac Q dvd P'"
      by (auto simp: P'_def)
    hence "map_poly to_ac Q dvd [:-α, 1:] ^ ?p"
      by (simp add: ‹P' = [:-α, 1:] ^ ?p›)
    moreover have "Factorial_Ring.prime_elem [:-α, 1:]"
      by (intro prime_elem_linear_field_poly) auto
    hence "Factorial_Ring.prime [:-α, 1:]"
```

```
      unfolding Factorial_Ring.prime_def by (auto simp: normalize_monic)
    ultimately obtain m where "m ≤ ?p" "normalize (map_poly to_ac Q)
= [:-α, 1:] ^ m"
      using divides_primepow by blast
    hence "map_poly to_ac Q = [:-α, 1:] ^ m"
      using ‹monic Q› by (subst (asm) normalize_monic) auto
    moreover from this have "m > 0"
      using Q by (intro Nat.gr0I) auto
    moreover have "m ≠ ?p"
    proof
      assume "m = ?p"
      hence "Q = P"
        using ‹map_poly to_ac Q = [:-α, 1:] ^ m› eq
        by (simp add: P'_def to_ac.injectivity)
      with Q have "Factorial_Ring.irreducible P"
        using idom_class.prime_elem_imp_irreducible by blast
      with assms have "pderiv P ≠ 0"
        by blast
      thus False
        by (auto simp: P_def pderiv_add pderiv_monom of_nat_eq_0_iff_char_dvd)
    qed
    ultimately have m: "m ∈ {0<..<?p}" "map_poly to_ac Q = [:-α, 1:]
^ m"
      using ‹m ≤ ?p› by auto

    from m(1) have "¬?p dvd m"
      using p by auto
    have "poly.coeff ([:-α, 1:] ^ m) (m - 1) = - of_nat (m choose (m -
1)) * α"
      using m(1) by (subst coeff_linear_poly_power) auto
    also have "m choose (m - 1) = m"
      using ‹0 < m› by (subst binomial_symmetric) auto
    also have "[:-α, 1:] ^ m = map_poly to_ac Q"
      using m(2) ..
    also have "poly.coeff ... (m - 1) = to_ac (poly.coeff Q (m - 1))"
      by simp
    finally have "α = to_ac (-poly.coeff Q (m - 1) / of_nat m)"
      using m(1) p ‹¬?p dvd m› by (auto simp: field_simps of_nat_eq_0_iff_char_dvd)
    hence "(- poly.coeff Q (m - 1) / of_nat m) ^ ?p = c"
      using α by (metis to_ac.base.eq_iff to_ac.base.hom_power)
    thus "c ∈ range frob"
      unfolding frob_def by blast
  qed auto
  thus ?thesis ..
qed auto

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 (rule perfect_field_alt)
  fix p :: "'a poly"
  assume p: "Factorial_Ring.irreducible p"
  with assms[OF p] show "pderiv p ≠ 0"
    by auto
qed

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].