Elimination of Repeated Factors Algorithm

Katharina Kreuzer, Manuel Eberl

April 18, 2024

Abstract

This article formalises the Elimination of Repeated Factors (ERF) Algorithm. This is an algorithm to find the square-free part of polynomials over perfect fields. Notably, this encompasses all fields of characteristic 0 and all finite fields.

For fields with characteristic 0, the ERF algorithm proceeds similarly to the classical Yun algorithm (formalized in [3, File `Square_Free_Factorization.thy`]). However, for fields with non-zero characteristic p, Yun's algorithm can fail because the derivative of a non-zero polynomial can be 0. The ERF algorithm detects this case and therefore also works in this more general setting.

To state the ERF Algorithm in this general form, we build on the entry on perfect fields [1]. We show that the ERF algorithm is correct and returns a list of pairwise coprime square-free polynomials whose product is the input polynomial. Indeed, through this, the ERF algorithm also yields executable code for calculating the square-free part of a polynomial (denoted by the function *radical*).

The definition and proof of the ERF have been taken from Algorithm 1 in [2].

Contents

```
theory ERF_Library
imports
  Mason_Stothers.Mason_Stothers
  Berlekamp_Zassenhaus.Berlekamp_Type_Based
  Perfect_Fields.Perfect_Fields
begin
```

```
hide_const (open) Formal_Power_Series.radical
```

1 Auxiliary Lemmas

If all factors are monic, the product is monic as well (i.e. the normalization is itself).

```
lemma normalize_prod_monics:
  assumes "∀x∈A. monic x"
  shows "normalize (∏x∈A. x^(e x)) = (∏x∈A. x^(e x))"
  by (simp add: assms monic_power monic_prod normalize_monic)
```

All primes are monic.

```
lemma prime_monic:
  fixes p :: "'a :: {euclidean_ring_gcd,field} poly"
  assumes "p≠0" "prime p" shows "monic p"
  using normalize_prime[OF assms(2)] monic_normalize[OF assms(1)] by auto
```

If we know the factorization of a polynomial, we can explicitly characterize the derivative of said polynomial.

```
lemma pderiv_exp_prod_monic:
assumes "p = prod_mset fs"
shows "pderiv p = (sum (λ fi. let ei = count fs fi in
    Polynomial.smult (of_nat ei) (pderiv fi) * fi^(ei-1) * prod (λ fj.
fj^(count fs fj))
    ((set_mset fs) - {fi})) (set_mset fs))"
proof -
  have pderiv_fi: "pderiv (fi ^ count fs fi) =
    Polynomial.smult (of_nat (count fs fi)) (pderiv fi * (fi ^ (count
fs fi - Suc 0)))"
    if "fi ∈# fs" for fi
  proof -
    obtain i where i: "Suc i = count fs fi" by (metis ‹fi ∈# fs› in_countE)
    show ?thesis unfolding i[symmetric] by (subst pderiv_power_Suc) (auto
simp add: algebra_simps)
  qed
  show ?thesis unfolding assms prod_mset_multiplicity pderiv_prod sum_distrib_left
Let_def
```

```
    by (rule sum.cong[OF refl]) (auto simp add: algebra_simps pderiv_fi)
qed
```

Any element that divides a prime is either congruent to the prime (i.e. *p dvd c*) or a unit itself. Careful: This does not mean that $p = c$ since there could be another unit u such that $p = u * c$.

```
lemma prime_factors_prime:
  assumes "c dvd p" "prime p"
  shows "is_unit c ∨ p dvd c"
  using assms unfolding normalization_semidom_class.prime_def
  by (meson prime_elemD2)
```

A prime polynomial has degree greater than zero. This is clear since any polynomial of degree 0 is constant and thus also a unit.

```
lemma prime_degree_gt_zero:
  fixes p::"'a::{idom_divide,semidom_divide_unit_factor,field} poly"
  assumes "prime p"
  shows "degree p > 0"
  using assms by fastforce
```

This lemma helps to reason that if a sum is zero, under some conditions we can follow that the summands must also be zero.

```
lemma one_summand_zero:
  fixes a2::"'a ::field poly"
  assumes "Polynomial.smult a1 a2 + b = 0""c dvd b" "¬ c dvd a2"
  shows "a1 = 0"
  by (metis assms dvd_0_right dvd_add_triv_right_iff dvd_smult_cancel
dvd_trans)
```

1.1 Lemmas for the *radical* of polynomials

Properties of the function *radical*. Note: The radical polynomial in algebra denotes something else. Here, *radical* denotes the square-free and monic part of a polynomial (i.e. the product of all prime factors). This notion corresponds to radical ideals generated by square-free polynomials.

```
lemma squarefree_radical [intro]: "f ≠ 0 ⟹ squarefree (radical f)"
  by (simp add: in_prime_factors_iff multiplicity_radical_prime squarefree_factorial_semiri
```

```
lemma (in normalization_semidom_multiplicative) normalize_prod:
  "normalize (∏x∈A. f (x :: 'b) :: 'a) = (∏x∈A. normalize (f x))"
  by (induction A rule: infinite_finite_induct) (auto simp: normalize_mult)
```

```
lemma normalize_radical [simp]:
  fixes f :: "'a :: factorial_semiring_multiplicative"
  shows "normalize (radical f) = radical f"
```

```
  by (auto simp: radical_def normalize_prod in_prime_factors_iff normalize_prime
intro!: prod.cong)

lemma radical_of_squarefree:
  assumes "squarefree f"
  shows   "normalize (radical f) = normalize f"
proof -
  from assms have [simp]: "f ≠ 0"
    by auto
  have "normalize (∏# (prime_factorization f)) = normalize f"
    by (intro prod_mset_prime_factorization_weak) auto
  also have "prime_factorization f = mset_set (prime_factors f)"
    using assms
    by (intro multiset_eqI)
       (auto simp: count_prime_factorization count_mset_set' squarefree_factorial_semiring'
                   in_prime_factors_iff not_dvd_imp_multiplicity_0)
  also have "prod_mset (mset_set (prime_factors f)) = radical f"
    by (simp add: radical_def prod_unfold_prod_mset)
  finally show ?thesis
    by simp
qed
```

A constant polynomial has no primes in its prime factorization and its radical is 1.

```
lemma prime_factorization_degree0:
  fixes f :: "'a :: {factorial_ring_gcd,semiring_gcd_mult_normalize,field}
poly"
  assumes "degree f = 0"
  shows "prime_factorization f = {#}"
  by (simp add: assms prime_factorization_empty_iff)


lemma prime_factors_degree0:
  fixes f :: "'a :: {factorial_ring_gcd,semiring_gcd_mult_normalize,field}
poly"
  assumes "degree f = 0" "f≠0"
  shows "prime_factors f = {}"
  using prime_factorization_degree0 assms by auto

lemma radical_degree0:
  fixes f :: "'a :: {factorial_ring_gcd,semiring_gcd_mult_normalize,field}
poly"
  assumes "degree f = 0" "f≠0"
  shows "radical f = 1"
  by (simp add: assms is_unit_iff_degree)
```

A polynomial is square-free iff its normalization is also square-free.

```
lemma squarefree_normalize:
  "squarefree f ⟷ squarefree (normalize f)"
```

```
  by (simp add: squarefree_def)
```

Important: The zeros of a polynomial are also zeros of its *radical* and vice versa.

```
lemma same_zeros_radical: "(poly f a = 0) = (poly (radical f) a = 0)"
proof (cases "f = 0")
  case True show ?thesis unfolding True radical_def by auto
next
  case False
  have fin: "finite (prime_factors f)" by simp
  have f: "f = unit_factor f * prod_mset (prime_factorization f)"
    by (metis False in_prime_factors_imp_prime normalize_prime normalized_prod_msetI

      prod_mset_prime_factorization_weak unit_factor_mult_normalize)
  have "poly (unit_factor f) a≠0" using False poly_zero by fastforce
  moreover have "((∏p∈#prime_factorization f. poly p a) = 0)=((∏k∈prime_factors
f. poly k a) = 0)"
    by (subst prod_mset_zero_iff,subst prod_zero_iff[OF fin]) auto
  ultimately have "(poly f a = 0) = (poly (∏(prime_factors f)) a = 0)"

    by (subst f, subst poly_prod, subst poly_mult, subst poly_hom.hom_prod_mset)
auto
  then show ?thesis unfolding radical_def using False by auto
qed
```

1.2 More on square-free polynomials

We need to relate two different versions of the definition of a square-free polynomial (i.e. the functions *squarefree* and *square_free*). Over fields, they differ only in their behavior at 0.)

```
lemma squarefree_square_free:
  fixes x :: "'a :: {field} poly"
  assumes "x ≠ 0"
  shows "squarefree x = square_free x"
  using assms unfolding squarefree_def square_free_def proof (safe, goal_cases)
  case (1 q)
  have "q dvd 1" using 1(2,4) by (metis power2_eq_square)
  then have "degree q = 0" using poly_dvd_1[of q] by auto
  then show ?case using 1(3) by auto
next
  case (2 y)
  then have "degree y = 0" by (metis bot_nat_0.not_eq_extremum power2_eq_square)
  have "y≠0" using 2(2,4) by fastforce
  show ?case using is_unit_iff_degree[OF ‹y≠0›] ‹degree y = 0› by auto
qed

lemma (in comm_monoid_mult) prod_list_distinct_conv_prod_set:
  "distinct xs ⟹ prod_list (map (f :: 'b ⇒ 'a) xs) = prod f (set xs)"
```

```
  by (simp add: local.prod.distinct_set_conv_list)

lemma (in comm_monoid_mult) interv_prod_list_conv_prod_set_nat:
  "prod_list (map (f :: nat ⇒ 'a) [m..<n]) = prod f (set [m..<n])"
  by (metis distinct_upt local.prod.distinct_set_conv_list)

lemma (in comm_monoid_mult) prod_list_prod_nth:
  "prod_list (xs :: 'a list) = (∏ i = 0 ..< length xs. xs ! i)"
  using interv_prod_list_conv_prod_set_nat [of "(!) xs" 0 "length xs"]
by (simp add: map_nth)

lemma squarefree_mult_imp_coprime [dest]:
  assumes "squarefree (x * y)"
  shows   "coprime x y"
proof (rule coprimeI)
  fix d assume "d dvd x" "d dvd y"
  hence "d ^ 2 dvd x * y"
    by (simp add: mult_dvd_mono power2_eq_square)
  thus "d dvd 1"
    using assms unfolding squarefree_def by blast
qed

end


theory ERF_Perfect_Field_Factorization

imports ERF_Library

begin
```

Here we subsume properties of the factorization of a polynomial and its derivative in perfect fields. There are two main examples for perfect fields: fields with characteristic 0 and finite fields (i.e. $\mathbb{F}_q[x]$ where $q = p^n$, $n \in \mathbb{N}$ and p prime). For fields with characteristic 0, most of the lemmas below become trivial. But in the case of finite fields we get interesting results.

Since fields are not instantiated with gcd, we need the additional type class constraint `field_gcd`.

```
locale perfect_field_poly_factorization =
  fixes e :: "'e :: {perfect_field, field_gcd} itself"
    and f :: "'e poly"
    and p :: nat
  assumes p_def: "p = CHAR('e)"
    and deg: "degree f ≠ 0"
begin
```

Definitions to shorten the terms.

```
definition fm where "fm = normalize f"
```

definition *fac* **where** *"fac = prime_factorization fm"*
definition *fac_set* **where** *"fac_set = prime_factors fm"*
definition *ex* **where** *"ex = (λp. multiplicity p fm)"*

The split of all prime factors into *P1* and *P2* only affects fields with prime characteristic. For fields with characteristic 0, *P2* is always empty.

definition *P1* **where** *"P1 = {f∈fac_set. ¬ p dvd ex f}"*
definition *P2* **where** *"P2 = {f∈fac_set. p dvd ex f}"*

Assumptions on the degree of f rewritten.

lemma *deg_f_gr_0[simp]: "degree f > 0"* **using** *deg* **by** *auto*
lemma *f_nonzero[simp]: "f≠0"* **using** *deg degree_0* **by** *blast*
lemma *fm_nonzero: "fm ≠ 0"* **using** *deg_f_gr_0 fm_def* **by** *auto*

Lemmas on *fac_set*, *P1* and *P2*. *P1* and *P2* are a partition of *fac_set*.

lemma *fac_set_nonempty[simp]: "fac_set ≠ {}"* **unfolding** *fac_set_def*
  by *(metis deg_f_gr_0 degree_0 degree_1 degree_normalize fm_def nat_less_le prod_mset.empty prod_mset_prime_factorization_weak set_mset_eq_empty_iff)*
lemma *fac_set_P1_P2: "fac_set = P1 ∪ P2"*
  unfolding *P1_def P2_def* **by** *auto*

lemma *P1_P2_intersect[simp]: "P1 ∩ P2 = {}"*
  unfolding *P1_def P2_def* **by** *auto*

lemma *finites[simp]: "finite fac_set" "finite P1" "finite P2"*
  unfolding *P1_def P2_def fac_set_def* **by** *auto*

All elements of *fac_set* (and thus of *P1* and *P2*) are monic, irreducible, prime and prime elements.

lemma *fac_set_prime[simp]: "prime x"* **if** *"x∈fac_set"*
  using *fac_set_def that* **by** *blast*

lemma *P1_prime[simp]: "prime x"* **if** *"x∈P1"*
  using *P1_def fac_set_prime that* **by** *blast*

lemma *P2_prime[simp]: "prime x"* **if** *"x∈P2"*
  using *P2_def fac_set_prime that* **by** *blast*

lemma *fac_set_monic[simp]: "monic x"* **if** *"x∈fac_set"*
  using *fac_set_def that* **by** *(metis in_prime_factors_imp_prime monic_normalize normalize_prime not_prime_0)*
lemma *P1_monic[simp]: "monic x"* **if** *"x∈P1"*
  using *P1_def fac_set_monic that* **by** *blast*
lemma *P2_monic[simp]: "monic x"* **if** *"x∈P2"*
  using *P2_def fac_set_monic that* **by** *blast*

lemma *fac_set_prime_elem[simp]: "prime_elem x"* **if** *"x∈fac_set"*

```
  using fac_set_def that in_prime_factors_imp_prime by blast
lemma P1_prime_elem[simp]: "prime_elem x" if "x∈P1"
  using P1_def fac_set_prime that by blast
lemma P2_prime_elem[simp]: "prime_elem x" if "x∈P2"
  using P2_def fac_set_prime that by blast

lemma fac_set_irreducible[simp]: "irreducible x" if "x∈fac_set"
  using fac_set_def that fac_set_prime_elem by auto
lemma P1_irreducible[simp]: "irreducible x" if "x∈P1"
  using P1_def fac_set_prime that by blast
lemma P2_irreducible[simp]: "irreducible x" if "x∈P2"
  using P2_def fac_set_prime that by blast
```

All prime factors are nonzero. Also the derivative of a prime factor is nonzero. The exponent of a prime factor is also nonzero.

```
lemma nonzero[simp]: "fj ≠ 0" if "fj∈ fac_set"
  using fac_set_def that zero_not_in_prime_factors by blast

lemma nonzero_deriv[simp]: "pderiv fj ≠ 0" if "fj∈ fac_set"
  by (intro irreducible_imp_pderiv_nonzero)
    (use that fac_set_def in_prime_factors_imp_prime in ‹auto›)

lemma P1_ex_nonzero: "of_nat (ex x) ≠ (0:: 'e)" if "x∈P1"
  using that P1_def p_def by (simp add: of_nat_eq_0_iff_char_dvd)
```

A prime factor and its derivative are coprime. Also elements of *P1* and *P2* are coprime.

```
lemma deriv_coprime: "algebraic_semidom_class.coprime x (pderiv x)"
  if "x∈fac_set" for x using irreducible_imp_separable that
  using fac_set_def in_prime_factors_imp_prime by blast


lemma P1_P2_coprime: "algebraic_semidom_class.coprime x (∏f∈P2. f^ex
f)" if "x∈P1"
  by (smt (verit) P1_def P2_def as_ufd.prime_elem_iff_irreducible fac_set_def

      in_prime_factors_imp_prime irreducible_dvd_prod mem_Collect_eq
      normalization_semidom_class.prime_def prime_dvd_power prime_imp_coprime
primes_dvd_imp_eq that)

lemma P1_ex_P2_coprime: "algebraic_semidom_class.coprime (x^ex x) (∏f∈P2.
f^ex f)" if "x∈P1"
  using P1_P2_coprime by (simp add: that)
```

We now come to the interesting factorizations of the normalization of a polynomial. It can be represented in Isabelle as the multi-set product *prod_mset* of the multi-set of its prime factors, or as a product of prime factors to the power of its multiplicity. We can also split the product into two parts: The prime factors with exponent divisible by the cardinality of the finite field *p* (= the set *P2*) and those not divisible by *p* (= the set *P1*).

```
lemma f_fac: "fm = prod_mset fac"
  by (metis deg_f_gr_0 bot_nat_0.extremum_strict degree_0 fac_def fm_def
in_prime_factors_iff
      normalize_eq_0_iff normalize_prime normalized_prod_msetI prod_mset_prime_factorizatio

lemma fm_P1_P2: "fm = (∏fj∈P1. fj^(ex fj)) * (∏fj∈P2. fj^(ex fj))"
proof -
  have *: "fm = (∏fj∈fac_set. fj^(ex fj))" unfolding f_fac unfolding
fac_def fac_set_def
    by (smt (verit, best) count_prime_factorization_prime ex_def in_prime_factors_imp_prime

        prod.cong prod_mset_multiplicity)
  show ?thesis unfolding * using fac_set_P1_P2
      prod.union_disjoint[OF finites(2) finites(3) P1_P2_intersect] by
auto
qed
```

We now want to look at the derivative and its explicit form. The problem for polynomials over fields with prime characteristic is that for prime factors with exponent divisible by the characteristic, the exponent as a field element equals 0 and cancels out the respective term, i.e.: In a finite field $\mathbb{F}_{p^n}[x]$, if $f = g^p$ where g is a prime polynomial and p is the cardinality, then $f' = p \cdot g^{p-1} = 0$. This has nasty side effects in the elimination of repeated factors (ERF) algorithm. As all summands with a derivative of a factor in *P2* cancel out, we can also write the derivative as a sum over all derivatives over *P1* only.

```
definition deriv_part where
  "deriv_part = (λy. Polynomial.smult (of_nat (ex y)) (pderiv y * y ^
(ex y - Suc 0) *
                (∏fj∈fac_set - {y}. fj ^ ex fj)))"

definition deriv_monic where
  "deriv_monic = (λy. pderiv y * y ^ (ex y - Suc 0) * (∏fj∈fac_set -
{y}. fj ^ ex fj))"

lemma pderiv_fm: "pderiv fm = (∑f∈fac_set. deriv_part f)"
  unfolding deriv_part_def pderiv_exp_prod_monic[OF f_fac] Let_def fac_set_def
fac_def ex_def
    count_prime_factorization by (intro sum.cong, simp)
    (smt (verit) DiffD1 One_nat_def in_prime_factors_iff mult_smult_left
prod.cong)

lemma sumP2_deriv_zero: "(∑f∈P2. deriv_part f) = 0"
  unfolding deriv_part_def unfolding P2_def
  by (intro sum.neutral, use P2_def p_def of_nat_eq_0_iff_char_dvd in
‹auto›)
```

```
lemma pderiv_fm': "pderiv fm = (∑f∈P1. deriv_part f)"
  by (subst pderiv_fm, subst fac_set_P1_P2,
      subst sum.union_disjoint[OF finites(2) finites(3) P1_P2_intersect])
    (use sumP2_deriv_zero in ‹auto›)

definition deriv_P1 where
  "deriv_P1 = (λy. Polynomial.smult (of_nat (ex y)) (pderiv y * y ^ (ex
y - Suc 0) *
              (∏fj∈P1 - {y}. fj ^ ex fj)))"

lemma pderiv_fm'': "pderiv fm = (∏f∈P2. f^ex f) * (∑x∈P1. deriv_P1
x)"
proof (subst pderiv_fm', subst sum_distrib_left, intro sum.cong, safe,
goal_cases)
  case (1 x)
  have *: "fac_set -{x} = P2 ∪ (P1-{x})" unfolding fac_set_P1_P2
    using 1 P1_P2_intersect by blast
  have **: "P2 ∩ (P1 - {x}) = {}" using 1 P1_P2_intersect by blast
  have "(∏fj∈fac_set - {x}. fj ^ ex fj) = (∏f∈P2. f ^ ex f) * (∏fj∈P1
- {x}. fj ^ ex fj)"
    unfolding * by (intro prod.union_disjoint, auto simp add: **)
  then show ?case unfolding deriv_part_def deriv_P1_def by (auto simp
add: algebra_simps)
qed
```

Some properties that $f_i^{e_i}$ for prime factors f_i divides the summands of the derivative or not.

```
lemma ex_min_1_power_dvd_P1: "x ^ (ex x - 1) dvd deriv_part a" if "x∈P1"
"a∈P1" for x a
proof (cases "x = a")
  case True
  then show ?thesis unfolding deriv_part_def
    by (intro dvd_smult,subst dvd_mult2,subst dvd_mult) auto
next
  case False
  then have "x ^ (ex x - 1) dvd (∏fj∈fac_set - {a}. fj ^ ex fj)"
    by (metis (no_types, lifting) Num.of_nat_simps(1) P1_def P1_ex_nonzero
dvd_prod dvd_triv_right
        finite_Diff finites(1) insertE insert_Diff mem_Collect_eq power_eq_if
that(1) that(2))
  then show ?thesis unfolding deriv_part_def by (intro dvd_smult, subst
dvd_mult) auto
qed

lemma ex_power_dvd_P2: "x ^ ex x dvd deriv_part a" if "x∈P2" "a∈P1" un-
folding deriv_part_def
  by (intro dvd_smult, intro dvd_mult) (use P1_def P2_def that(1) that(2)
in ‹auto›)
```

```
lemma ex_power_not_dvd: "¬ y^ex y dvd deriv_monic y" if "y∈fac_set"
proof
  assume "y^ex y dvd deriv_monic y"
  then have "y * (y^(ex y-1)) dvd (pderiv y * (∏fj∈fac_set - {y}. fj
^ ex fj)) * (y^(ex y-1))"
    unfolding deriv_monic_def
    by (metis (no_types, lifting) count_prime_factorization_prime ex_def
fac_set_def
        in_prime_factors_imp_prime more_arith_simps(11) mult.commute not_in_iff
numeral_nat(7)
        power_eq_if that)
  then have *: "y dvd pderiv y * (∏fj∈fac_set - {y}. fj ^ ex fj)"
    unfolding dvd_mult_cancel_right dvd_smult_cancel by auto
  then have "y dvd (∏fj∈fac_set - {y}. fj ^ ex fj)"
    using deriv_coprime[THEN coprime_dvd_mult_right_iff] <y∈fac_set> by
auto
  then obtain fj where fj_def: "y dvd fj ^ ex fj" "fj∈fac_set - {y}"
using prime_dvd_prod_iff
    by (metis (no_types, lifting) finites(1) <y ∈ fac_set> fac_set_def
finite_Diff
        in_prime_factors_iff)
  then have "y dvd fj" using prime_dvd_power
    by (metis fac_set_def in_prime_factors_imp_prime that)
  then have "coprime y fj" using fj_def(2)
    by (metis Diff_iff Diff_not_in fac_set_prime primes_dvd_imp_eq that)
  then show False by (metis <y dvd fj> coprimeE dvd_refl fac_set_def
in_prime_factors_imp_prime
        not_prime_unit that)
qed

lemma P1_ex_power_not_dvd: "¬ y^ex y dvd deriv_part y" if "y∈P1"
proof
  assume ass: "y^ex y dvd deriv_part y"
  have "y^ex y dvd deriv_monic y"
    using P1_ex_nonzero ass dvd_smult_iff that unfolding deriv_part_def
deriv_monic_def by blast
  then show False using ex_power_not_dvd that unfolding P1_def by auto
qed

lemma P1_ex_power_not_dvd': "¬ y^ex y dvd deriv_P1 y" if "y∈P1"
proof
  assume "y^ex y dvd deriv_P1 y"
  then have ass: "y^ex y dvd pderiv y * y ^ (ex y - Suc 0) * (∏fj∈P1
- {y}. fj ^ ex fj)"
    using P1_ex_nonzero dvd_smult_iff that unfolding deriv_P1_def by blast
  then have "y * (y^(ex y-1)) dvd (pderiv y * (∏fj∈P1 - {y}. fj ^ ex
fj)) * (y^(ex y-1))"
```

```
    by (metis (no_types, lifting) Num.of_nat_simps(1) P1_ex_nonzero more_arith_simps(11)

        mult.commute numeral_nat(7) power_eq_if that)
  then have *: "y dvd pderiv y * (∏fj∈P1 - {y}. fj ^ ex fj)"
    unfolding dvd_mult_cancel_right dvd_smult_cancel by auto
  then have "y dvd (∏fj∈P1 - {y}. fj ^ ex fj)"
    using deriv_coprime[THEN coprime_dvd_mult_right_iff] ‹y∈P1› fac_set_P1_P2
by blast
  then obtain fj where fj_def: "y dvd fj ^ ex fj" "fj∈P1 - {y}" using
prime_dvd_prod_iff
    by (metis (no_types, lifting) P1_def finites(2) ‹y ∈ P1› fac_set_def
finite_Diff
        in_prime_factors_iff mem_Collect_eq)
  then have "y dvd fj" using prime_dvd_power
    by (metis UnCI fac_set_P1_P2 fac_set_def in_prime_factors_iff that)
  then show False
    by (metis DiffD1 Diff_not_in P1_prime fj_def(2) primes_dvd_imp_eq
that)
qed
```

If the derivative of the normalized polynomial *fm* is zero, then all prime factors have an exponent divisible by the cardinality *p*.

```
lemma pderiv0_p_dvd_count: "p dvd ex fj" if "fj∈fac_set" "pderiv fm =
0"
proof -
  have "(∑f∈fac_set. deriv_part f) = 0" using pderiv_fm ‹pderiv fm =
0› by auto
  then have zero: "Polynomial.smult (of_nat (ex fj)) (deriv_monic fj)
+ (∑f∈fac_set-{fj}. deriv_part f) = 0"
    unfolding deriv_part_def deriv_monic_def
    by (metis (no_types, lifting) finites(1) sum.remove that(1))
  have dvd: "fj ^ ex fj dvd (∑f∈fac_set - {fj}. deriv_part f)"
    unfolding deriv_part_def
    by (intro dvd_sum,intro dvd_smult,intro dvd_mult)
      (use finites(1) that(1) in ‹blast›)
  have nondvd: "¬ fj ^ ex fj dvd deriv_monic fj"
    using ex_power_not_dvd[OF ‹fj∈fac_set›] unfolding deriv_monic_def
by auto
  have "of_nat (ex fj) = (0::'e)" by (rule one_summand_zero[OF zero dvd
nondvd])
  then show ?thesis using p_def of_nat_eq_0_iff_char_dvd by blast
qed
```

Properties on the multiplicity (i.e. the exponents) of prime factors in the factorization of the derivative.

```
lemma mult_fm[simp]: "count fac x = ex x" if "x∈fac_set"
  by (simp add: count_prime_factorization_prime ex_def fac_def that)

lemma mult_deriv1: "multiplicity x (pderiv fm) = ex x - 1"
```

```
  if "x∈P1" "pderiv fm ≠ 0" for x
proof (subst multiplicity_eq_Max[OF that(2)])
  show "¬ is_unit x" using that(1) using P1_def fac_set_def not_prime_unit
by blast
  then have fin: "finite {n. x ^ n dvd pderiv fm}"
    using is_unit_iff_infinite_divisor_powers that(2) by blast
  show "Max {n. x ^ n dvd pderiv fm} = ex x - 1"
  proof (subst Max_eq_iff, goal_cases)
    case 2 then show ?case by (metis empty_Collect_eq one_dvd power_0)
  next
    case 3
    have dvd: "x ^(ex x-1) dvd pderiv fm" unfolding pderiv_fm' by(intro
dvd_sum)
        (use ex_min_1_power_dvd_P1[OF ‹x∈P1›] in ‹auto›)
    have not :"¬ x^ex x dvd pderiv fm"
    proof
      assume ass: "x ^ ex x dvd pderiv fm"
      have coprime: "algebraic_semidom_class.coprime (x^ex x) (∏f∈P2.
f ^ ex f)"
        using P1_ex_P2_coprime that(1) by auto
      then have "x ^ ex x dvd (∑y∈P1. deriv_P1 y)"
        using ass coprime_dvd_mult_right_iff[OF coprime] unfolding pderiv_fm''
by auto
      also have "(∑y∈P1. deriv_P1 y) = deriv_P1 x + (∑y∈P1-{x}. deriv_P1
y)"
        by (intro sum.remove, auto simp add: that)
      also have "... = deriv_P1 x + (x^ex x) * (∑y∈P1 - {x}. Polynomial.smult
(of_nat (ex y))
      (pderiv y * y ^ (ex y - Suc 0) * (∏fj∈(P1 - {x})- {y}. fj ^ ex
fj)))"
      proof -
        have *: "(pderiv xa * xa ^ (ex xa - Suc 0) * (∏fj∈P1 - {xa}.
fj ^ ex fj)) =
          (x ^ ex x * (pderiv xa * xa ^ (ex xa - Suc 0) * (∏fj∈P1 - {x}
- {xa}. fj ^ ex fj)))"
          if "xa∈P1-{x}" for xa
        proof -
          have "x∈P1-{xa}" using that ‹x∈P1› by auto
          have fin: "finite (P1 - {xa})" by auto
          show ?thesis by (subst prod.remove[OF fin ‹x∈P1-{xa}›])
              (smt (verit, del_insts) Diff_insert2 Groups.mult_ac(3) insert_commute)
        qed
        show ?thesis unfolding deriv_P1_def by (auto simp add: sum_distrib_left
*)
      qed
      finally have "x ^ ex x dvd deriv_P1 x + (x^ex x) * (∑y∈P1 - {x}.
Polynomial.smult (of_nat (ex y))
      (pderiv y * y ^ (ex y - Suc 0) * (∏fj∈(P1 - {x})- {y}. fj ^ ex
fj)))" by auto
```

```
      then have "x ^ ex x dvd deriv_P1 x" using dvd_add_times_triv_right_iff
        by (simp add: dvd_add_left_iff)
      then show False using P1_ex_power_not_dvd'[OF that(1)] by auto
    qed
    then have less: "a ≤ ex x - 1" if "a∈{n. x ^ n dvd pderiv fm}" for
a
      by (metis IntI Int_Collect Suc_pred' algebraic_semidom_class.unit_imp_dvd

          bot_nat_0.not_eq_extremum is_unit_power_iff not_less_eq_eq power_le_dvd
that)
    show ?case using dvd less by auto
  qed (use fin in ‹auto›)
qed


lemma mult_deriv: "multiplicity x (pderiv fm) ≥ (if p dvd ex x then
ex x else ex x - 1)"
  if "x∈fac_set" "pderiv fm ≠ 0"
proof (subst multiplicity_eq_Max[OF that(2)])
  show "¬ is_unit x" using that(1) using fac_set_def not_prime_unit by
blast
  then have fin: "finite {n. x ^ n dvd pderiv fm}"
    using is_unit_iff_infinite_divisor_powers that(2) by blast
  show "Max {n. x ^ n dvd pderiv fm} ≥ (if p dvd ex x then ex x else
ex x - 1)"
  proof (split if_splits, safe, goal_cases)
    case 1
    then have "x∈P2" unfolding P2_def using that by auto
    have dvd: "x ^ ex x dvd pderiv fm" unfolding pderiv_fm' by(intro
dvd_sum)
        (use ‹x ∈ P2› ex_power_dvd_P2 in ‹blast›)
    then show ?case by (intro Max_ge, auto simp add: fin)
  next
    case 2
    then have "x∈P1" unfolding P1_def using that by auto
    have dvd: "x ^(ex x-1) dvd pderiv fm" unfolding pderiv_fm' by(intro
dvd_sum)
        (use ‹x ∈ P1› ex_min_1_power_dvd_P1 in ‹blast›)
    then show ?case by (intro Max_ge, auto simp add: fin)
  qed
qed

end
end

theory ERF_Algorithm
imports
  ERF_Perfect_Field_Factorization
begin
```

2 Elimination of Repeated Factors Algorithm

This file contains the elimnation of repeated factors (ERF) algorithm for polynomials over perfect fields. This algorithm does not only work over fields with characteristic 0 like the classical Yun Algorithm but also for example over finite fields with prime characteristic (i.e. $\mathbb{F}_q[x]$ for $q = p^n$, $n \in \mathbb{N}$ and p prime). Intuitively, the ERF algorithm proceeds similarly to the classical Yun algorithm, taking the gcd of the polynomial and its derivative and thus eliminating repeated factors iteratively. However, if we work over finite characteristic, prime factors with exponent divisible by the characteristic p are cancelled out since $p \equiv 0$. Therefore, we separate prime factors with exponent divisible by the characteristic from the rest and treat them seperately in the ERF algorithm.

Since we use the *gcd*, we need the additional type constraint *field_gcd*.

context
assumes *"SORT_CONSTRAINT('e::{perfect_field, field_gcd})"*
begin

The funtion *ERF_step* describes the main body of the ERF algorithm. Let us walk through the algorithm step by step.

- A polynomial of degree 0 is constant and thus there is nothing to do.
- We only consider the monic part of our polynomial f using the *normalize* function.
- u is the gcd of the monic f and its derivative.
- $u = 1$ iff f is already square-free. If the characteristic is zero, this property is already fulfilled. Otherwise we continue and denote the (prime) characteristic by p.
- If $u \neq 1$, we split f in a part v and w. v is already square-free and contains all prime factors with exponent not divisible by p.
- w contains all prime factors with exponent divisible by p. Thus we can take the p-th root of w (by using the inverse Frobenius homomorphism *inv_frob_poly*) and obtain z (which we will further reduce in an iterative step).

```
definition ERF_step ::"'e poly ⇒ _" where
  "ERF_step f = (if degree f = 0 then None else (let
     f_mono = normalize f;
     u = gcd f_mono (pderiv f_mono);
     n = degree f
  in (if u = 1 then None else let
```

```
    v = f_mono div u;
    w = u div gcd u (v^n);
    z = inv_frob_poly w
    in Some (v, z)
  )
))"
```

lemma *ERF_step_0 [simp]: "ERF_step 0 = None"*
  unfolding *ERF_step_def* **by** *auto*

lemma *ERF_step_const: "degree f = 0 ⟹ ERF_step f = None"*
  unfolding *ERF_step_def* **by** *auto*

For the correctness proof of the `local.ERF_step` algorithm, we need to show that u, v and w have the correct form.

Let $f = \prod_i f_i^{e_i}$ where we assume f to be monic and f_i are the prime factors with exponents e_i. Let furthermore $P_1 = \{f_i.\ p \nmid e_i\}$ and $P_2 = \{f_i.\ p \mid e_i\}$. Then we have

$$u = \prod_{f_i \in P_1} f_i^{e_i - 1} \cdot \prod_{f_i \in P_2} f_i^{e_i}$$

.

```
lemma u_characterization :
  fixes f::"'e poly"
  assumes "degree f ≠ 0"
  and u_def: "u = gcd (normalize f) (pderiv (normalize f))"
  shows "u = (let fm' = normalize f in
                (∏fj∈prime_factors fm'. let ej = multiplicity fj fm'
in
                (if CHAR('e) dvd ej then  fj ^ ej else fj ^(ej-1))))"
(is ?u)
    and "u = (let fm' = normalize f; P1 = {f∈prime_factors fm'. ¬ CHAR('e)
dvd multiplicity f fm'};
               P2 = {f∈prime_factors fm'. CHAR('e) dvd multiplicity f
fm'} in
              (∏fj∈P1. fj^(multiplicity fj fm' -1)) * (∏fj∈P2. fj^(multiplicity
fj fm')))"
(is ?u')
proof -
  define p where "p = CHAR('e)"
  — Here we import the lemmas on the factorization of polynomials over a finite
field
  interpret perfect_field_poly_factorization "TYPE('e)" f p
  proof
    show "degree f ≠ 0" using assms by auto
  qed (auto simp add: p_def)

  have "u =(∏fj∈fac_set. let ej = ex fj in (if p dvd ej then  fj ^ ej
else fj ^(ej-1)))"
```

```
    if "pderiv fm = 0"
  proof -
    have "u = fm" unfolding u_def ‹pderiv fm = 0› using fm_def that by
auto
    moreover have "fm = (∏fj∈fac_set. let ej = ex fj in (if p dvd ej
then  fj ^ ej else
      fj ^(ej-1)))"
      using pderiv0_p_dvd_count[OF _ that] unfolding Let_def f_fac prod_mset_multiplicity
      by (intro prod.cong) (simp add:fac_set_def fac_def, auto)
    ultimately show ?thesis by auto
  qed

  moreover have "u =(∏fj∈fac_set. let ej = ex fj in (if p dvd ej then
fj ^ ej else fj ^(ej-1)))"
    if "pderiv fm ≠ 0"
  unfolding u_def fm_def[symmetric] proof (subst gcd_eq_factorial', goal_cases)
    case 3
    let ?prod_pow = "(∏p∈prime_factors fm ∩ prime_factors (pderiv fm).
        p ^ min (multiplicity p fm) (multiplicity p (pderiv fm)))"
    have norm: "normalize ?prod_pow = ?prod_pow" by (intro normalize_prod_monics)
      (metis Int_iff dvd_0_left_iff in_prime_factors_iff monic_normalize
normalize_prime)
    have subset: "prime_factors fm ∩ prime_factors (pderiv fm) ⊆ fac_set"

      unfolding fac_set_def by auto
    show ?case unfolding norm proof (subst prod.mono_neutral_left[OF
_ subset], goal_cases)
      case 2
      have "i ∈# prime_factorization (pderiv fm)"  if "i ∈ fac_set" "ei
= count fac i"
           "i ^ min ei (multiplicity i (pderiv fm)) ≠ 1" for i ei
      proof (intro prime_factorsI)
        have "min ei (multiplicity i (pderiv fm)) ≠ 0" using that(3)
by (metis power_0)
        then have "multiplicity i (pderiv fm) ≥ 1" by simp
        then show "i dvd pderiv fm"
          using not_dvd_imp_multiplicity_0 by fastforce
        show "pderiv fm ≠ 0" "prime i" using ‹pderiv fm ≠ 0› ‹i ∈ fac_set›

          unfolding fac_set_def by auto
      qed
      then show ?case  using mult_fm unfolding fac_set_def Let_def us-
ing ex_def by fastforce
    next
      case 3
      have "x ^ min (multiplicity x fm) (multiplicity x (pderiv fm)) =
x ^ multiplicity x fm"
        if "x ∈ fac_set" "p dvd multiplicity x fm" for x
        using ‹pderiv fm ≠ 0› ex_def mult_deriv that(1) that(2) by fastforce
```

```
      moreover have "x ^ min (multiplicity x fm) (multiplicity x (pderiv
fm)) =
        x ^ (multiplicity x fm - Suc 0)" if "x ∈ fac_set" "¬ p dvd multiplicity
x fm" for x
        using P1_def ‹pderiv fm ≠ 0› ex_def mult_deriv1 that(1) that(2)
by auto
      ultimately show ?case by (intro prod.cong, simp, unfold Let_def,
          auto simp add: ex_def mult_deriv[OF _ ‹pderiv fm ≠ 0›])
    qed (auto simp add: fac_set_def)
  qed (auto simp add: fm_nonzero that)

  ultimately have u: "u =(∏fj∈fac_set. let ej = ex fj in (if p dvd ej
then  fj ^ ej else fj ^(ej-1)))"
    by blast
  then have u': "u = (∏fj∈P1. fj^(ex fj -1)) * (∏fj∈P2. fj^(ex fj))"
    unfolding Let_def by (smt (verit) P1_def P2_def P1_P2_intersect fac_set_P1_P2

      finites(2) finites(3) mem_Collect_eq prod.cong prod.union_disjoint)

  show ?u using u unfolding ex_def fm_def fac_set_def unfolding Let_def
p_def by auto
  show ?u' using u' unfolding local.P1_def local.P2_def
    unfolding p_def ex_def fm_def Let_def fac_set_def by auto
qed
```

Continuing our calculations, we get:

$$v = \prod_{f_i \in P_1} f_i$$

Therefore, v is already square-free and v's prime factors are exactly P_1.

```
lemma v_characterization:
assumes "ERF_step f = Some (v,z)"
shows "v = (let fm = normalize f in fm div (gcd fm (pderiv fm)))" (is
?a)
and "v = ∏{x∈prime_factors (normalize f). ¬ CHAR('e) dvd multiplicity
x (normalize f)}" (is ?b)
and "prime_factors v = {x∈prime_factors (normalize f). ¬ CHAR('e) dvd
multiplicity x (normalize f)}"(is ?c)
and "squarefree v"(is ?d)
proof -
  define p where "p = CHAR('e)"
  have [simp]: "degree f ≠ 0" using assms unfolding ERF_step_def by (metis
not_None_eq)


  interpret perfect_field_poly_factorization "TYPE('e)" f p
  proof (unfold_locales)
    show "p = CHAR('e)" by (rule p_def)
```

```
    show "degree f ≠ 0" by auto
  qed

  define u where "u = gcd fm (pderiv fm)"
  have u_def': "u = gcd (normalize f) (pderiv (normalize f))" unfold-
ing u_def fm_def by auto
  have u: "u =(∏fj∈fac_set. let ej = ex fj in (if p dvd ej then  fj
^ ej else fj ^(ej-1)))"
    using u_characterization[OF ‹degree f ≠ 0›] u_def
    unfolding fm_def Let_def fac_set_def ex_def p_def
    by blast
  have u': "u = (∏fj∈P1. fj^(ex fj -1)) * (∏fj∈P2. fj^(ex fj))"
    using u_characterization(2)[OF ‹degree f ≠ 0› u_def'] unfolding fm_def[symmetric]
Let_def
      fac_set_def[symmetric] ex_def[symmetric] p_def[symmetric]
    using P1_def P2_def ex_def by presburger

  have v_def: "v = fm div u" unfolding fm_def u_def using assms unfold-
ing ERF_step_def
    by (auto split: if_splits simp add: Let_def)
  then show ?a unfolding u_def fm_def Let_def by auto

  have v: "v = ∏P1"
  proof -
    have "v = ((∏fj∈P1. fj^(ex fj)) * (∏fj∈P2. fj^(ex fj))) div u"

      unfolding v_def fm_P1_P2 by auto
    also have "... = (∏fj∈P1. fj^(ex fj)) div (∏fj∈P1. fj^(ex fj-1))"
unfolding u Let_def
    by (metis (no_types, lifting) fm_nonzero div_div_div_same dvd_triv_right
fm_P1_P2 mult_not_zero
      nonzero_mult_div_cancel_right semiring_gcd_class.gcd_dvd1 u u' u_def)
    also have "... = ∏P1"
    proof -
      have *: "(∏fj∈P1. fj^(ex fj)) = (∏P1) * (∏fj∈P1. fj^(ex fj-1))"
      by (smt (z3) P1_def dvd_0_right mem_Collect_eq power_eq_if prod.cong
prod.distrib)
      show ?thesis unfolding * by auto
    qed
    finally show ?thesis by auto
  qed
  then show ?b unfolding P1_def fac_set_def fm_def ex_def unfolding p_def
by auto

  have prime_factors_v: "prime_factors v = P1" unfolding v
  proof (subst prime_factors_prod[OF finites(2)], goal_cases)
    case 1
    then show ?case using fac_set_P1_P2 nonzero by blast
  next
```

```
    case 2
    have prime: "prime x" if "x∈P1" for x using P1_def fac_set_def that
by blast
    have "⋃ (prime_factors ` P1) = P1" using prime[THEN prime_prime_factors]
by auto
    then show ?case by simp
  qed
  then show ?c unfolding P1_def fac_set_def ex_def fm_def unfolding p_def
by auto

  show "squarefree v" proof (subst squarefree_factorial_semiring', safe,
goal_cases)
    case (2 p)
    have "0 ∉ (λx. x) ` P1" using prime_factors_v P1_prime_elem by fastforce
    moreover have "prime_elem p" using "2" in_prime_factors_imp_prime
by blast
    moreover have "sum (multiplicity p) P1 = 1"
      by (metis "2" finites(2) calculation(2) in_prime_factors_imp_prime
multiplicity_prime
      prime_factors_v prime_multiplicity_other sum_eq_1_iff)
    ultimately show ?case using 2 unfolding prime_factors_v unfolding
v
      by (subst prime_elem_multiplicity_prod_distrib) auto
  qed (simp add: v P1_def)
qed
```

For the definition of w, we only want to get the prime factors in P_2. Therefore, we kick out all prime factors in P_1 from f by calculating this gcd.

$$\gcd(u, v^{\deg f}) = \prod_{f_i \in P_1} f_i^{e_i - 1}$$

```
lemma gcd_u_v:
  assumes "ERF_step f = Some (v,z)"
  shows "let fm = normalize f; u = gcd fm (pderiv fm);
  P1 = {x∈prime_factors fm. ¬ CHAR('e) dvd multiplicity x fm} in
gcd u (v^(degree f)) = (∏ fj∈P1. fj ^(multiplicity fj fm -1))"
proof -
  define p where "p = CHAR('e)"
  have [simp]: "degree f ≠ 0" using assms unfolding ERF_step_def by (metis
not_None_eq)

  — We import the lemmas on factorization, the characterizations of u and v
  interpret perfect_field_poly_factorization "TYPE('e)" f p
  proof
    show "p = CHAR('e)" by (rule p_def)
    show "degree f ≠ 0" by auto
  qed
```

```
  define u where "u = gcd (normalize f) (pderiv (normalize f))"
  have u': "u = (∏fj∈P1. fj^(ex fj -1)) * (∏fj∈P2. fj^(ex fj))"
    using u_characterization(2)[OF ‹degree f ≠ 0› u_def] unfolding fm_def[symmetric]
Let_def
      fac_set_def[symmetric] ex_def[symmetric] p_def[symmetric]
    using P1_def P2_def ex_def by presburger
  have v: "v = ∏P1" using v_characterization(2)[OF assms] unfolding P1_def
fac_set_def fm_def ex_def
    using p_def by auto
  have prime_factors_v: "prime_factors v = P1" using v_characterization(3)[OF
assms]
    unfolding P1_def fac_set_def ex_def fm_def using p_def by auto
  have v_def: "v = fm div u" unfolding fm_def u_def using assms unfold-
ing ERF_step_def
    by (auto split: if_splits simp add: Let_def)

  have "gcd u (v^(degree f)) = (∏fj∈P1. fj ^(multiplicity fj fm -1))"
unfolding u' v
  proof (subst gcd_mult_left_right_cancel, goal_cases)
    case 1
    then show ?case by (simp add: P1_P2_coprime prod_coprime_left)
  next
    case 2
    have nonzero1: "(∏fj∈P1. fj ^ (ex fj - 1)) ≠ 0" using ex_def by
auto
    have nonzero2: "∏P1 ^ (degree f) ≠ 0" using prime_factors_v v v_def
by fastforce
    have null: "0 ∉ (λfj. fj ^ (ex fj - Suc 0)) ` P1" using prime_factors_v
nonzero1 by force
    have null': "0 ∉ (λx. x) ` P1" using prime_factors_v P1_irreducible
by blast
    have P1: "prime_factors (∏fj∈P1. fj ^ (ex fj - Suc 0)) ∩ prime_factors
(∏P1 ^ (degree f)) =
      {x∈P1. ex x > 1}"
    proof (subst prime_factors_prod[OF finites(2) null],
           subst prime_factors_power[OF deg_f_gr_0],
           subst prime_factors_prod[OF finites(2) null'],
           unfold comp_def, safe, goal_cases)
      case (1 x xa xb) then show ?case by (metis dvd_trans in_prime_factors_iff
prime_factors_v)
    next
      case (2 x xa xb)
      then have "x = xa"
        by (metis in_prime_factors_iff prime_dvd_power prime_factors_v
primes_dvd_imp_eq)
      then have "x ∈# prime_factorization (x ^ (ex x - 1))" using 2 by
auto
      then show ?case
      by (metis gr0I in_prime_factors_iff not_prime_unit power_0 zero_less_diff)
```

```
    next
      case (3 x) then show ?case
      by (smt (verit) One_nat_def Totient.prime_factors_power dvd_refl
image_ident
        in_prime_factors_iff mem_simps(8) null' prime_factors_v zero_less_diff)

    next
      case (4 x) then show ?case
      by (metis dvd_refl in_prime_factors_iff mem_simps(8) not_prime_0
prime_factors_v)
    qed
    have n: "ex fj ≤ degree f" if "fj∈P1" for fj
    proof -
      have "fj ≠ 0" using that unfolding P1_def by auto
      have "degree (fj ^ multiplicity fj fm) ≤ degree f"
        using divides_degree[OF multiplicity_dvd] fm_nonzero
        by (subst degree_normalize[of f, symmetric], unfold fm_def[symmetric])
auto
      then have "degree fj * ex fj ≤ degree f" by (subst degree_power_eq[OF
‹fj≠0›, symmetric],
        unfold ex_def)
      then show ?thesis
      by (metis Missing_Polynomial.is_unit_field_poly ‹fj ≠ 0› bot_nat_0.not_eq_extremum

        dual_order.trans dvd_imp_le dvd_triv_right empty_iff in_prime_factors_iff

        linordered_semiring_strict_class.mult_pos_pos prime_factorization_1
prime_factors_v
        semiring_norm(160) set_mset_empty that)
    qed
    then have min: "min (ex x - Suc 0) (degree f) = ex x -1" if "x∈P1"
for x using n[OF that] by auto
    have mult1: "multiplicity g (∏fj∈P1. fj ^ (ex fj - Suc 0)) = ex
g -1" if "g∈P1" for g
    proof -
      have "prime_elem g" using in_prime_factors_imp_prime prime_factors_v
that by blast
      have "multiplicity g (∏fj∈P1. fj ^ (ex fj - Suc 0)) =
            multiplicity g (g^(ex g -1) * (∏fj∈P1-{g}. fj ^ (ex fj -
Suc 0)))"
        by (subst prod.remove[OF finites(2) ‹g∈P1›], auto)
      also have "... = multiplicity g ((∏fj∈P1-{g}. fj ^ (ex fj - Suc
0)) * g^(ex g -1))"
        by (auto simp add: algebra_simps)
      also have "... = multiplicity g (g^(ex g -1))"
      proof (intro multiplicity_prime_elem_times_other[OF ‹prime_elem
g›], rule ccontr, safe)
        assume ass: "g dvd (∏fj∈P1 - {g}. fj ^ (ex fj - Suc 0))"
        have "irreducible g" using ‹prime_elem g› by blast
```

```
        obtain a where "a∈P1-{g}" "g dvd a ^ (ex a -1)"
          using irreducible_dvd_prod[OF ‹irreducible g› ass]
          by (metis dvd_1_left nat_dvd_1_iff_1)
        then have "g dvd a" by (meson ‹prime_elem g› prime_elem_dvd_power)
        then show False
        by (metis DiffD1 DiffD2 ‹a ∈ P1 - {g}› in_prime_factors_imp_prime
insertI1
          prime_factors_v primes_dvd_imp_eq that)
      qed
      also have "... = ex g -1" by (metis image_ident in_prime_factors_imp_prime

        multiplicity_same_power not_prime_unit null' prime_factors_v that)
      finally show ?thesis by blast
    qed
    have mult2: "multiplicity g (∏P1 ^ (degree f)) = (degree f)" if "g∈P1"
for g
    proof -
      have "∏P1≠0" unfolding P1_def
      using P1_def in_prime_factors_iff prime_factors_v that v by simp
      have "prime_elem g" using in_prime_factors_imp_prime prime_factors_v
that by blast
      have "multiplicity g (∏P1 ^ (degree f)) = (degree f) * (multiplicity
g (∏P1))"
        by (subst prime_elem_multiplicity_power_distrib[OF ‹prime_elem
g› ‹∏P1≠0›], auto)
      also have "... = (degree f) * multiplicity g (g * ∏(P1-{g}))" by
(metis finites(2) prod.remove that)
      also have "... = (degree f) * multiplicity g (∏(P1-{g}) * g)" by
(auto simp add: algebra_simps)
      also have "... = (degree f)"
      proof (subst multiplicity_prime_elem_times_other[OF ‹prime_elem
g›])
        show "¬ g dvd ∏(P1 - {g})" by (metis DiffD1 DiffD2 ‹prime_elem
g›
          as_ufd.prime_elem_iff_irreducible in_prime_factors_imp_prime
irreducible_dvd_prod
          prime_factors_v primes_dvd_imp_eq singletonI that)
        show "(degree f) * multiplicity g g = (degree f)"
          by (auto simp add: multiplicity_prime[OF ‹prime_elem g›])
      qed
      finally show ?thesis by blast
    qed
    have split: "(∏x∈{x ∈ P1. Suc 0 < ex x}. x ^ (ex x - Suc 0)) =
      (∏fj∈P1. fj ^ (ex fj - Suc 0))"
    proof -
      have *: "ex x ≠ 0" if "x∈P1" for x by (metis P1_ex_nonzero of_nat_0
that)
      have "Suc 0 < ex x" if "x∈P1" "ex x ≠ Suc 0" for x using *[OF that(1)]
that(2) by auto
```

```
      then have union: "P1 = {x∈P1. 1 < ex x} ∪ {x∈P1. ex x = 1}" by
auto
      show ?thesis by (subst (2) union, subst prod.union_disjoint) auto

    qed
    show ?case by (subst gcd_eq_factorial'[OF nonzero1 nonzero2],subst
normalize_prod_monics)
      (auto simp add: P1 mult1 mult2 min normalize_prod_monics split,
auto simp add: ex_def)
  qed
  then show ?thesis unfolding Let_def u_def P1_def fm_def ex_def fac_set_def
using p_def by auto
qed
```

Finally, we can calculate

$$w = \prod_{f_i \in P_2} f_i^{p \cdot (e_i/p)}$$

and

$$z = \sqrt[p]{w} = \prod_{f_i \in P_2} f_i^{e_i/p}$$

Now, we can show the correctness of the `local.ERF_step` function. These properties comprise:

- prime factors of f are either in v or in z
- v is already square-free
- z is non-zero and the p-th power of z divides f (important for the termination of the ERF)

```
lemma ERF_step_correct:
  assumes "ERF_step f = Some (v, z)"
  shows   "radical f = v * radical z"
          "squarefree v"
          "z ^ CHAR('e) dvd f"
          "z≠0"
          "CHAR('e) = 0 ⟹ z = 1"
proof -
  define p where "p = CHAR('e)"

  have [simp]: "degree f ≠ 0" using assms unfolding ERF_step_def by (metis
not_None_eq)


  interpret perfect_field_poly_factorization "TYPE('e)" f p
  proof (unfold_locales)
```

```
    show "p = CHAR('e)" by (rule p_def)
    show "degree f ≠ 0" by auto
  qed

  define u where "u = gcd fm (pderiv fm)"
  define n where "n = degree f"
  define w where "w = u div (gcd u (v^n))"
  have u_def': "u = gcd (normalize f) (pderiv (normalize f))" unfolding u_def fm_def by auto

  have u: "u =(∏fj∈fac_set. let ej = ex fj in (if p dvd ej then  fj
^ ej else fj ^(ej-1)))"
    using u_characterization[OF ‹degree f ≠ 0›] u_def
    unfolding fm_def Let_def fac_set_def ex_def p_def
    by blast
  have u': "u = (∏fj∈P1. fj^(ex fj -1)) * (∏fj∈P2. fj^(ex fj))"
    using u_characterization(2)[OF ‹degree f ≠ 0› u_def'] unfolding fm_def[symmetric]
Let_def
      fac_set_def[symmetric] ex_def[symmetric] p_def[symmetric]
    using P1_def P2_def ex_def by presburger

  have v_def: "v = fm div u" unfolding fm_def u_def using assms unfolding ERF_step_def
    by (auto split: if_splits simp add: Let_def)
  have v: "v = ∏P1" using v_characterization(2)[OF assms] unfolding P1_def
fac_set_def fm_def ex_def
    using p_def by auto

  have prime_factors_v: "prime_factors v = P1"
    using v_characterization(3)[OF assms] unfolding P1_def fac_set_def
fm_def ex_def
    using p_def by auto

  show "squarefree v" by (rule v_characterization(4)[OF assms])

  have gcd_u_v: "gcd u (v^n) = (∏fj∈P1. fj ^(ex fj -1))" using gcd_u_v[OF
assms]
    unfolding Let_def u_def fm_def P1_def fac_set_def ex_def using p_def
n_def by force


  have w: "w = (∏fj∈P2. fj^(ex fj))" unfolding w_def gcd_u_v unfolding u'
  by (metis (no_types, lifting) fm_nonzero gcd_eq_0_iff gcd_u_v nonzero_mult_div_cancel_lef
u_def)

  have w_power: "w = (∏fj∈P2. fj^(ex fj div p))^p"
  proof -
    have w: "w = (∏fj∈P2. fj^((ex fj div p)*p))" unfolding w P2_def
```

```
by auto
    show ?thesis unfolding w by (auto simp add: power_mult prod_power_distrib[symmetric])
  qed

  have z_def: "z = inv_frob_poly w"
    unfolding p_def w_def u_def fm_def n_def using assms unfolding ERF_step_def
    by (auto simp add: Let_def split: if_splits)

  show "CHAR('e) = 0 ⟹ z = 1"
    by (auto simp: z_def p_def w_power)

  have z: "z = (∏x∈P2. x ^(ex x div p))"
    by (cases "p = 0") (auto simp: z_def p_def w_power inv_frob_poly_power')

  have zw: "z^CHAR('e) = w" unfolding w_power z p_def[symmetric] by auto

  show "z ^ CHAR('e) dvd f" unfolding zw w
  by (metis (full_types) dvd_mult_right dvd_normalize_iff dvd_refl fm_P1_P2
fm_def)

  show "z≠0" by (simp add: fac_set_P1_P2 z)

  have prime_factors_z: "∏(prime_factors z) = ∏P2" unfolding z
  proof (subst prime_factors_prod)
    show "finite P2" by auto
    show "0 ∉ (λx. x ^ (ex x div p)) ` P2" using fac_set_P1_P2 by force
    have pos: "0 < ex x div p" if "x∈P2" for x
    by (metis (no_types, lifting) P2_def count_prime_factorization_prime
dvd_div_eq_0_iff
        ex_def fac_set_def gr0I in_prime_factors_imp_prime mem_Collect_eq
not_in_iff that)
    have "prime_factors (x ^ (ex x div p)) = {x}" if "x∈P2" for x
      unfolding prime_factors_power[OF pos[OF that]] using that by (simp
add: prime_prime_factors)
    then have *: "(⋃x∈P2. prime_factors (x ^ (ex x div p))) = P2" by
auto
    show "∏(⋃ ((prime_factors ∘ (λx. x ^ (ex x div p))) ` P2)) = ∏P2"

      unfolding comp_def * by auto
  qed

  show "radical f = v * radical z"
  proof -
    have factors: "prime_factors f = fac_set" unfolding fac_set_def fm_def
by auto
    have "∏(prime_factors f) = ∏P1 * ∏P2" unfolding factors fac_set_P1_P2

      by (subst prod.union_disjoint, auto)
    also have "... = v * ∏(prime_factors z)" unfolding v using ‹z≠0›
```

```
prime_factors_z by auto
    finally have "∏(prime_factors f) = v * ∏(prime_factors z)" by auto
    then show ?thesis  unfolding radical_def using ‹z≠0› f_nonzero by
auto
  qed

qed
```

If the algorithm stops, then the input was already square-free or zero.

```
lemma ERF_step_correct_None:
  assumes "ERF_step f = None"
    shows "degree f = 0 ∨ radical f = normalize f"
          "f≠0 ⟹ squarefree f"
proof -
  define p where "p = CHAR('e)"
  define fm' where "fm' = normalize f"
  define u where "u = gcd fm' (pderiv fm')"
  have or: "degree f = 0 ∨ u = 1" using assms unfolding ERF_step_def

    by (smt (verit, best) option.simps(3) u_def fm'_def)
  have rad: "radical f = normalize f" if "u =1" "degree f≠0"
  proof -
    interpret perfect_field_poly_factorization "TYPE('e)" f p
    proof (unfold_locales)
      show "p = CHAR('e)" by (rule p_def)
      show "degree f ≠ 0" using that(2) by auto
    qed
    have u_def': "u = gcd (normalize f) (pderiv (normalize f))" unfold-
ing u_def fm'_def by auto
    have u': "u = (∏fj∈P1. fj^(ex fj -1)) * (∏fj∈P2. fj^(ex fj))"
      using u_characterization(2)[OF ‹degree f ≠ 0› u_def'] unfolding
fm_def[symmetric] Let_def
        fac_set_def[symmetric] ex_def[symmetric] p_def[symmetric]
      using P1_def P2_def ex_def by presburger
    have P2_1: "(∏fj∈P2. fj^(ex fj)) = 1"  using u' ‹u=1›
    by (smt (verit, best) class_cring.finprod_all1 dvd_def dvd_mult2 dvd_prod
dvd_refl
      dvd_triv_right ex_power_not_dvd perfect_field_poly_factorization.P2_def

      perfect_field_poly_factorization_axioms finites(3) idom_class.unit_imp_dvd
mem_Collect_eq)
    then have "P2 = {}"
    by (smt (verit, ccfv_threshold) Collect_cong Collect_mem_eq UnCI dvd_prodI
empty_iff
      ex_power_not_dvd fac_set_P1_P2 finites(3) idom_class.unit_imp_dvd)

    moreover have mult: "multiplicity fj fm = 1" if "fj∈P1" for fj
    by (metis (no_types, lifting) One_nat_def P1_ex_power_not_dvd Suc_pred
```

```
      P2_1 ‹u = 1› algebraic_semidom_class.unit_imp_dvd dvd_prod dvd_refl

      perfect_field_poly_factorization.ex_def perfect_field_poly_factorization_axioms
finites(2)
      gr0I is_unit_power_iff mult_1_right that u')
    ultimately have "fm = ∏P1" unfolding fm_P1_P2 ‹(∏fj∈P2. fj ^ ex
fj) = 1› unfolding ex_def
      by (subst mult_1_right, intro prod.cong, simp) (auto simp add: mult)
    also have "... = radical f"
    by (metis P1_P2_intersect ‹P2 = {}› f_nonzero fac_set_P1_P2 fac_set_def
finites(2) finites(3)
      fm_def one_neq_zero prime_factorization_1 prime_factorization_normalize
prod.union_disjoint
      radical_1 radical_def set_mset_empty verit_prod_simplify(2))
    finally show ?thesis unfolding fm_def by auto
  qed
  show *: "degree f = 0 ∨ radical f = normalize f" using or rad by auto
  show "squarefree f" if "f ≠ 0"
  proof -
    from ‹f ≠ 0› and * have "radical f = normalize f"
      by (metis Missing_Polynomial_Factorial.is_unit_field_poly normalize_1_iff
radical_unit)
    thus "squarefree f"
      using ‹f ≠ 0› squarefree_normalize squarefree_radical by metis
  qed
qed
```

The degree of z is less than the degree of f. This guarantees the termination of ERF.

```
lemma degree_ERF_step_less [termination_simp]:
  assumes "ERF_step f = Some (v, z)"
  shows   "degree z < degree f"
proof -
  define u where "u = gcd f (pderiv (normalize f))"
  define w where "w = u div gcd u (v ^ degree f)"
  from assms have "degree f > 0"
    by (auto split: if_splits simp: Let_def u_def w_def ERF_step_def)

  have "z ≠ 0"
    using assms ERF_step_correct(4) by blast
  have le: "degree z * CHAR('e) ≤ degree f"
    using divides_degree[OF ERF_step_correct(3)[OF assms]] ‹degree f >
0›
    unfolding degree_power_eq[OF ‹z ≠ 0›] by auto

  show ?thesis
  proof (cases "CHAR('e) = 0")
    case True
    thus ?thesis
```

```
    using ERF_step_correct(5)[OF assms] ‹degree f > 0› by auto
  next
    case False
    hence "CHAR('e) > 1"
      by (metis CHAR_nonzero less_one nat_neq_iff of_nat_1 of_nat_CHAR
zero_neq_one)
    show ?thesis
    proof (cases "degree z = 0")
      case False
      hence "degree z * 1 < degree z * CHAR('e)"
        by (intro mult_less_mono2) (use ‹CHAR('e) > 1› in auto)
      also have "... ≤ degree f"
        by (rule le)
      finally show ?thesis
        by simp
    qed (use ‹degree f > 0› in auto)
  qed
qed

lemma is_measure_degree [measure_function]: "is_measure Polynomial.degree"
  by (rule is_measure_trivial)
```

Finally, we state the full ERF algorithm. We show correctness as well.

```
fun ERF ::"'e poly ⇒ 'e poly list" where
  "ERF f = (
     case ERF_step f of
       None ⇒ if degree f = 0 then [] else [normalize f]
     | Some (v, z) ⇒ v # ERF z)"

lemmas [simp del] = ERF.simps


lemma ERF_0 [simp]: "ERF 0 = []"
  by (auto simp add: ERF.simps)

lemma ERF_const [simp]:
  assumes "degree f = 0"
  shows   "ERF f = []"
  by (auto simp add: ERF_step_const[OF assms] assms ERF.simps)

theorem ERF_correct:
  assumes "f ≠ 0"
  shows   "prod_list (ERF f) = radical f"
          "g ∈ set (ERF f) ⟹ squarefree g"
proof -
  show "prod_list (ERF f) = radical f"
  using assms proof (induction f rule: ERF.induct)
  case (1 f) then show ?case proof (cases "ERF_step f")
      case None
```

```
      have "prod_list (ERF f) = (if degree f = 0 then 1 else normalize
f)"
        using None by (auto simp: ERF.simps)
      moreover have "radical f = 1" if "degree f = 0" using radical_degree0[OF
that ‹f≠0›]
        by simp
      moreover have "radical f = normalize f" if "degree f ≠ 0"
        using ERF_step_correct_None[OF None] that by auto
      ultimately show ?thesis by auto
    next
      case (Some a)
      obtain v z where vz: "(v,z) = a" by (metis surj_pair)
      then have Some': "ERF_step f = Some (v,z)" using Some by auto
      have "prod_list (ERF f) = v * prod_list(ERF z)"
        by (auto simp add: Some' ERF.simps)
      also have "... = v * radical z"
        by (subst 1)(auto simp add: Some vz[symmetric] ERF_step_correct(4)[OF
Some'])
      also have "... = radical f" using ERF_step_correct(1)[OF Some', symmetric]
by auto
      finally show ?thesis by auto
    qed
  qed

  show "g ∈ set (ERF f) ⟹ squarefree g"
  using assms proof (induction f rule: ERF.induct)
  case (1 f) then show ?case proof (cases "ERF_step f")
      case None
      have "set (ERF f) = (if degree f = 0 then {} else {normalize f})"
        using None by (auto simp: ERF.simps)
      moreover have "degree f ≠ 0" by (metis "1.prems"(1) calculation
emptyE)
      moreover have "squarefree (normalize f)" if "degree f ≠ 0"
        using ERF_step_correct_None(2)[OF None] that squarefree_normalize

        "1"(3) by blast
      ultimately show ?thesis using 1 by auto
    next
      case (Some a)
      obtain v z where vz: "(v,z) = a" by (metis surj_pair)
      then have Some': "ERF_step f = Some (v,z)" using Some by auto
      have "set (ERF f) = {v} ∪  set (ERF z)"
        by (auto simp add: Some' ERF.simps)
      moreover have "squarefree g" if "g∈{v}" using ERF_step_correct(2)[OF
Some']
        that by auto
      moreover have "squarefree g" if "g∈set (ERF z)"
        using 1 ERF_step_correct(4)[OF Some'] that Some' by blast
      ultimately show ?thesis using 1(2) by blast
```

```
    qed
  qed
qed
```

It is also easy to see that any two polynomials in the list returned by *local.ERF* are coprime.

```
lemma ERF_pairwise_coprime: "sorted_wrt coprime (ERF p)"
proof (cases "p = 0")
  case [simp]: False
  show ?thesis
    unfolding sorted_wrt_iff_nth_less
  proof safe
    fix i j :: nat
    assume ij: "i < j" "j < length (ERF p)"
    have "(∏k∈{i,j}. ERF p ! k) dvd (∏k<length (ERF p). ERF p ! k)"
      by (rule prod_dvd_prod_subset) (use ij in auto)
    also have "... = prod_list (ERF p)"
      by (simp add: prod_list_prod_nth atLeast0LessThan)
    also have "... = radical p"
      by (simp add: ERF_correct)
    finally have "ERF p ! i * ERF p ! j dvd radical p"
      using ij by simp
    hence "squarefree (ERF p ! i * ERF p ! j)"
      using False squarefree_mono by blast
    thus "coprime (ERF p ! i) (ERF p ! j)"
      by blast
  qed
qed auto
```

We can also compute the radical of a polynomial with the ERF algorithm by simply multiplying together the individual parts we found.

```
lemma radical_code [code_unfold]: "radical f = (if f = 0 then 0 else
prod_list (ERF f))"
  using ERF_correct(1)[of f] by simp
```

With this, the ERF algorithm can also serve as an executable test for the square-freeness of a polynomial (especially over a finite field):

```
lemma squarefree_poly_code [code_unfold]:
  fixes p :: "'a :: field_gcd poly"
  shows "squarefree p ⟷ p ≠ 0 ∧ Polynomial.degree p = Polynomial.degree
(radical p)"
proof
  assume *: "p ≠ 0 ∧ Polynomial.degree p = Polynomial.degree (radical
p)"
  have "p dvd radical p"
    by (rule dvd_euclidean_size_eq_imp_dvd) (use * in ‹auto simp: euclidean_size_poly_def›)
  thus "squarefree p"
    using * squarefree_mono squarefree_radical by blast
```

```
next
  assume "squarefree p"
  have "Polynomial.degree (radical p) = Polynomial.degree (normalize (radical
p))"
    by auto
  also have "normalize (radical p) = normalize p"
    using ‹squarefree p› radical_of_squarefree by blast
  finally show "p ≠ 0 ∧ Polynomial.degree p = Polynomial.degree (radical
p)"
    using ‹squarefree p› by auto
qed

end

end

theory ERF_Code_Fixes
  imports Berlekamp_Zassenhaus.Finite_Field
  Perfect_Fields.Perfect_Fields
begin
```

3 Code Generation for ERF and Example

```
lemma inverse_mod_ring_altdef:
  fixes x :: "'p :: prime_card mod_ring"
  defines "x' ≡ Rep_mod_ring x"
  shows   "Rep_mod_ring (inverse x) = fst (bezout_coefficients x' CARD('p))
mod CARD('p)"
proof (cases "x' = 0")
  case False
  define y where "y = fst (bezout_coefficients x' CARD('p))"
  define z where "z = fst (bezout_coefficients x' CARD('p))"
  define p where "p = CARD('p)"
  from False have "coprime x' p"
    by (metis Rep_mod_ring_mod algebraic_semidom_class.coprime_commute

        dvd_imp_mod_0 prime_card_int prime_imp_coprime p_def assms)
  have "[x' * (y mod p) = x' * y] (mod p)"
    by (intro cong_mult cong_refl) auto
  also have "x' * y = x' * y + 0"
    by simp
  also have "[x' * y + 0 = x' * y + z * p] (mod p)"
    by (intro cong_add cong_refl) (auto simp: cong_def)
  also have "[x' * y + z * p = gcd x' p] (mod p)"
    by (metis bezout_coefficients_fst_snd cong_def mod_mult_self2 mult.commute
y_def z_def p_def)
  also have "gcd x' p = 1"
    using ‹coprime x' p› by auto
  finally have "(x' * (y mod p)) mod p = 1"
```

```
    by (simp add: cong_def p_def)
  thus ?thesis
    unfolding p_def y_def x'_def
    by (metis Rep_mod_ring_inverse inverse_unique of_int_mod_ring.rep_eq
one_mod_ring.rep_eq times_mod_ring.rep_eq)
next
  case True
  hence "x = 0"
    by (metis Rep_mod_ring_inverse x'_def zero_mod_ring_def)
  thus ?thesis unfolding True
    by (auto simp: x'_def bezout_coefficients_left_0 inverse_mod_ring_def
zero_mod_ring.rep_eq)
qed

lemmas inverse_mod_ring_code' [code] =
  inverse_mod_ring_altdef [where 'p = "'p :: {prime_card, card_UNIV}"]

lemma divide_mod_ring_code' [code]:
  "x / (y :: 'p :: {prime_card, card_UNIV} mod_ring) = x * inverse y"
  by (fact divide_inverse)

instantiation mod_ring :: ("{finite, card_UNIV}") card_UNIV
begin
definition "card_UNIV = Phantom('a mod_ring) (of_phantom (card_UNIV ::
'a card_UNIV))"
definition "finite_UNIV = Phantom('a mod_ring) True"
instance
  by intro_classes
     (simp_all add: finite_UNIV_mod_ring_def finite_UNIV card_UNIV_mod_ring_def
card_UNIV)
end

lemmas of_int_mod_ring_code [code] =
  of_int_mod_ring.rep_eq[where ?'a = "'a :: {finite, card_UNIV}"]

lemmas plus_mod_ring_code [code] =
  plus_mod_ring.rep_eq[where ?'a = "'a :: {finite, card_UNIV}"]

lemmas minus_mod_ring_code [code] =
  minus_mod_ring.rep_eq[where ?'a = "'a :: {finite, card_UNIV}"]

lemmas uminus_mod_ring_code [code] =
  uminus_mod_ring.rep_eq[where ?'a = "'a :: {finite, card_UNIV}"]

lemmas times_mod_ring_code [code] =
  times_mod_ring.rep_eq[where ?'a = "'a :: {finite, card_UNIV}"]

lemmas inverse_mod_ring_code [code] =
  inverse_mod_ring_def[where ?'a = "'a :: {prime_card, finite, card_UNIV}"]
```

```
lemmas divide_mod_ring_code [code] =
  divide_mod_ring_def[where ?'a = "'a :: {prime_card, finite, card_UNIV}"]

lemma card_UNIV_code:
  "card (UNIV :: 'a :: card_UNIV set) = of_phantom (card_UNIV :: ('a,
nat) phantom)"
  by (simp add: card_UNIV)

setup ‹
  Code_Preproc.map_pre (fn ctxt =>
    ctxt addsimprocs
      [Simplifier.make_simproc context
        {name = "card_UNIV",
         lhss = [term‹card UNIV›],
         proc = fn _ => fn _ => fn ct =>
          SOME @{thm card_UNIV_code [THEN eq_reflection]},
         identifier = []}])
›


class semiring_char_code = semiring_1 +
  fixes semiring_char_code :: "('a, nat) phantom"
  assumes semiring_char_code_correct: "semiring_char_code = Phantom('a)
CHAR('a)"

instantiation mod_ring :: ("{finite,nontriv,card_UNIV}") semiring_char_code
begin
definition semiring_char_code_mod_ring :: "('a mod_ring, nat) phantom"
where
  "semiring_char_code_mod_ring = Phantom('a mod_ring) (of_phantom (card_UNIV
:: ('a, nat) phantom))"
instance
  by standard (auto simp: semiring_char_code_mod_ring_def card_UNIV)
end

instantiation poly :: ("{semiring_char_code, comm_semiring_1}") semiring_char_code
begin
definition
  "semiring_char_code_poly =
      Phantom('a poly) (of_phantom (semiring_char_code :: ('a, nat) phantom))"
instance
  by standard (auto simp: semiring_char_code_poly_def semiring_char_code_correct)
end

instantiation fps :: ("{semiring_char_code, comm_semiring_1}") semiring_char_code
begin
definition
  "semiring_char_code_fps =
```

```
      Phantom('a fps) (of_phantom (semiring_char_code :: ('a, nat) phantom))"
instance
  by standard (auto simp: semiring_char_code_fps_def semiring_char_code_correct)
end

instantiation fls :: ("{semiring_char_code, comm_semiring_1}") semiring_char_code
begin
definition
  "semiring_char_code_fls =
      Phantom('a fls) (of_phantom (semiring_char_code :: ('a, nat) phantom))"
instance
  by standard (auto simp: semiring_char_code_fls_def semiring_char_code_correct)
end


lemma semiring_char_code [code]:
  "semiring_char x =
     (if x = TYPE('a :: semiring_char_code) then
        of_phantom (semiring_char_code :: ('a, nat) phantom) else
        Code.abort STR ''semiring_char'' (λ_. semiring_char x))"
  by (auto simp: semiring_char_code_correct)

end


theory ERF_Code_Test
imports
  "HOL-Library.Code_Target_Numeral"
  ERF_Algorithm
  ERF_Code_Fixes
begin

hide_const (open) Formal_Power_Series.radical
notation (output) Abs_mod_ring ("_")
```

3.1 Example for the code generation with *GF(2)*

```
type_synonym gf2 = "bool mod_ring"

definition x where "x = [:0, 1:]"
definition p :: "gf2 poly"
  where "p = x^16 + x^15 + x^13 + x^11 + x^9 + x^8 + x^6 + x^5 + x^4
+ x^2 + x + 1"

value "ERF p"
value "radical p"
```

end

References

[1] M. Eberl and K. Kreuzer. Perfect fields. *Archive of Formal Proofs*, November 2023. https://isa-afp.org/entries/Perfect_Fields.html, Formal proof development.

[2] M. Scott. Factoring polynomials over finite fields, May 2019. https://carleton.ca/math/wp-content/uploads/Factoring-Polynomials-over-Finite-Fields_Melissa-Scott.pdf.

[3] R. Thiemann and A. Yamada. Polynomial factorization. *Archive of Formal Proofs*, January 2016. https://isa-afp.org/entries/Polynomial_Factorization.html, Formal proof development.