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))"
  ⟨proof⟩
```

All primes are monic.

```
lemma prime_monic:
  fixes p :: "'a :: {euclidean_ring_gcd,field} poly"
  assumes "p≠0" "prime p" shows "monic p"
  ⟨proof⟩
```

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⟩
```

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"
  ⟨proof⟩
```

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"*
  ⟨*proof*⟩

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"*
  ⟨*proof*⟩

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)"*
  ⟨*proof*⟩

lemma **(in** *normalization_semidom_multiplicative)* *normalize_prod:*
  "normalize (∏x∈A. f (x :: 'b) :: 'a) = (∏x∈A. normalize (f x))"
  ⟨*proof*⟩

lemma *normalize_radical [simp]:*
  fixes *f :: "'a :: factorial_semiring_multiplicative"*
  shows *"normalize (radical f) = radical f"*
  ⟨*proof*⟩

lemma *radical_of_squarefree:*
  assumes *"squarefree f"*
  shows *"normalize (radical f) = normalize f"*
⟨*proof*⟩

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 = {#}"*
  ⟨*proof*⟩

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 = {}"*
  ⟨*proof*⟩

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"*
  ⟨*proof*⟩

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

lemma *squarefree_normalize:*
  "squarefree f ⟷ squarefree (normalize f)"
  ⟨*proof*⟩

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*⟩

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"*
  ⟨*proof*⟩

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

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])"
  ⟨*proof*⟩

lemma **(in** *comm_monoid_mult) prod_list_prod_nth:*
  "prod_list (xs :: 'a list) = (∏ i = 0 ..< length xs. xs ! i)"
  ⟨*proof*⟩

```
lemma squarefree_mult_imp_coprime [dest]:
  assumes "squarefree (x * y)"
  shows   "coprime x y"
⟨proof⟩

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" ⟨proof⟩
lemma f_nonzero[simp]: "f≠0" ⟨proof⟩
lemma fm_nonzero: "fm ≠ 0" ⟨proof⟩
```

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

```
lemma fac_set_nonempty[simp]: "fac_set ≠ {}" ⟨proof⟩
```

lemma *fac_set_P1_P2: "fac_set = P1 ∪ P2"*
⟨*proof*⟩

lemma *P1_P2_intersect[simp]: "P1 ∩ P2 = {}"*
⟨*proof*⟩

lemma *finites[simp]: "finite fac_set" "finite P1" "finite P2"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *P1_prime[simp]: "prime x"* **if** *"x∈P1"*
⟨*proof*⟩

lemma *P2_prime[simp]: "prime x"* **if** *"x∈P2"*
⟨*proof*⟩

lemma *fac_set_monic[simp]: "monic x"* **if** *"x∈fac_set"*
⟨*proof*⟩
lemma *P1_monic[simp]: "monic x"* **if** *"x∈P1"*
⟨*proof*⟩
lemma *P2_monic[simp]: "monic x"* **if** *"x∈P2"*
⟨*proof*⟩

lemma *fac_set_prime_elem[simp]: "prime_elem x"* **if** *"x∈fac_set"*
⟨*proof*⟩
lemma *P1_prime_elem[simp]: "prime_elem x"* **if** *"x∈P1"*
⟨*proof*⟩
lemma *P2_prime_elem[simp]: "prime_elem x"* **if** *"x∈P2"*
⟨*proof*⟩

lemma *fac_set_irreducible[simp]: "irreducible x"* **if** *"x∈fac_set"*
⟨*proof*⟩
lemma *P1_irreducible[simp]: "irreducible x"* **if** *"x∈P1"*
⟨*proof*⟩
lemma *P2_irreducible[simp]: "irreducible x"* **if** *"x∈P2"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *nonzero_deriv[simp]: "pderiv fj ≠ 0"* **if** *"fj∈ fac_set"*
⟨*proof*⟩

lemma *P1_ex_nonzero: "of_nat (ex x) ≠ (0:: 'e)"* **if** *"x∈P1"*
⟨*proof*⟩

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* ⟨*proof*⟩

lemma *P1_P2_coprime: "algebraic_semidom_class.coprime x (∏f∈P2. f^ex f)"* **if** *"x∈P1"*
⟨*proof*⟩

lemma *P1_ex_P2_coprime: "algebraic_semidom_class.coprime (x^ex x) (∏f∈P2. f^ex f)"* **if** *"x∈P1"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *fm_P1_P2: "fm = (∏fj∈P1. fj^(ex fj)) * (∏fj∈P2. fj^(ex fj))"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

lemma *sumP2_deriv_zero: "(∑f∈P2. deriv_part f) = 0"*
⟨*proof*⟩

lemma *pderiv_fm': "pderiv fm = (∑f∈P1. deriv_part f)"*
⟨*proof*⟩

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*⟩

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*⟩

lemma *ex_power_dvd_P2: "x ^ ex x dvd deriv_part a"* **if** *"x∈P2" "a∈P1"* ⟨*proof*⟩

lemma *ex_power_not_dvd: "¬ y^ex y dvd deriv_monic y"* **if** *"y∈fac_set"*
⟨*proof*⟩

lemma *P1_ex_power_not_dvd: "¬ y^ex y dvd deriv_part y"* **if** *"y∈P1"*
⟨*proof*⟩

lemma *P1_ex_power_not_dvd': "¬ y^ex y dvd deriv_P1 y"* **if** *"y∈P1"*
⟨*proof*⟩

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*⟩

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"*
⟨*proof*⟩

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

```
  if "x∈P1" "pderiv fm ≠ 0" for x
⟨proof⟩


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⟩

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"
  ⟨proof⟩

lemma ERF_step_const: "degree f = 0 ⟹ ERF_step f = None"
  ⟨proof⟩
```

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⟩
```

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⟩
```

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⟩
```

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⟩
```

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⟩
```

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⟩

lemma is_measure_degree [measure_function]: "is_measure Polynomial.degree"
  ⟨proof⟩
```

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 = []"
  ⟨proof⟩

lemma ERF_const [simp]:
  assumes "degree f = 0"
```

```
  shows "ERF f = []"
  ⟨proof⟩

theorem ERF_correct:
  assumes "f ≠ 0"
  shows "prod_list (ERF f) = radical f"
        "g ∈ set (ERF f) ⟹ squarefree g"
⟨proof⟩
```

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⟩
```

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))"
  ⟨proof⟩
```

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⟩

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⟩

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"*
  ⟨*proof*⟩

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
  ⟨*proof*⟩
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)"
  ⟨*proof*⟩

⟨*ML*⟩

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
  ⟨*proof*⟩
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
  ⟨*proof*⟩
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
  ⟨*proof*⟩
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
  ⟨*proof*⟩
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))"
  ⟨*proof*⟩

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.