Factorization of Polynomials with Algebraic Coefficients*

Manuel Eberl   René Thiemann

May 26, 2024

Abstract

The AFP already contains a verified implementation of algebraic numbers. However, it is has a severe limitation in its factorization algorithm of real and complex polynomials: the factorization is only guaranteed to succeed if the coefficients of the polynomial are rational numbers. In this work, we verify an algorithm to factor all real and complex polynomials whose coefficients are algebraic. The existence of such an algorithm proves in a constructive way that the set of complex algebraic numbers is algebraically closed. Internally, the algorithm is based on resultants of multivariate polynomials and an approximation algorithm using interval arithmetic.

Contents

1 Introduction 2

2 Resultants and Multivariate Polynomials 2
2.1 Connecting Univariate and Multivariate Polynomials 2
2.2 Exact Division of Multivariate Polynomials 9
2.3 Implementation of Division on Multivariate Polynomials . . . 12
2.4 Class Instances for Multivariate Polynomials and Containers . 14
2.5 Resultants of Multivariate Polynomials 15

3 Testing for Integrality and Conversion to Integers 16

4 Representing Roots of Polynomials with Algebraic Coefficients 17
4.1 Preliminaries 17
4.2 More Facts about Resultants 19
4.3 Systems of Polynomials 19
4.4 Elimination of Auxiliary Variables 21

*Supported by FWF (Austrian Science Fund) project Y757.

4.5 A Representing Polynomial for the Roots of a Polynomial with Algebraic Coefficients 21
4.6 Soundness Proof for Complex Algebraic Polynomials 22
4.7 Soundness Proof for Real Algebraic Polynomials 22
4.8 Algebraic Closedness of Complex Algebraic Numbers 22
4.9 Executable Version to Compute Representative Polynomials . 23

5 Root Filter via Interval Arithmetic **24**
5.1 Generic Framework 24

6 Roots of Real and Complex Algebraic Polynomials **27**

7 Factorization of Polynomials with Algebraic Coefficients **30**
7.1 Complex Algebraic Coefficients 30
7.2 Real Algebraic Coefficients 31

1 Introduction

The formalization of algebraic numbers [4, 6] includes an algorithm that given a univariate polynomial f over $\mathbb{Z}$ or $\mathbb{Q}$, it computes all roots of f within $\mathbb{R}$ or $\mathbb{C}$. In this AFP entry we verify a generalized algorithm that also allows polynomials as input whose coefficients are complex or real algebraic numbers, following [5, Section 3].

The verified algorithm internally computes resultants of multivariate polynomials, where we utilize Braun and Traub's subresultant algorithm in our verified implementation [1, 2, 3]. In this way we achieve an efficient implementation with minimal effort: only a division algorithm for multivariate polynomials is required, but no algorithm for computing greatest common divisors of these polynomials.

Acknowledgments We thank Dmitriy Traytel for help with code generation for functions defined via **lift-definition**.

2 Resultants and Multivariate Polynomials

2.1 Connecting Univariate and Multivariate Polynomials

We define a conversion of multivariate polynomials into univariate polynomials w.r.t. a fixed variable x and multivariate polynomials as coefficients.

theory *Poly-Connection*
  imports
    Polynomials.MPoly-Type-Univariate
    Jordan-Normal-Form.Missing-Misc
    Polynomial-Interpolation.Ring-Hom-Poly

Hermite-Lindemann.More-Multivariate-Polynomial-HLW
Polynomials.MPoly-Type-Class
begin

lemma *mpoly-is-unitE*:
fixes *p* :: *'a* :: {*comm-semiring-1*, *semiring-no-zero-divisors*} *mpoly*
assumes *p dvd 1*
obtains *c* **where** *p* = *Const c c dvd 1*
⟨*proof*⟩

lemma *Const-eq-Const-iff* [*simp*]:
Const c = *Const c'* ⟷ *c* = *c'*
⟨*proof*⟩

lemma *is-unit-ConstI* [*intro*]: *c dvd 1* ⟹ *Const c dvd 1*
⟨*proof*⟩

lemma *is-unit-Const-iff*:
fixes *c* :: *'a* :: {*comm-semiring-1*, *semiring-no-zero-divisors*}
shows *Const c dvd 1* ⟷ *c dvd 1*
⟨*proof*⟩

lemma *vars-emptyE*: *vars p* = {} ⟹ (⋀*c*. *p* = *Const c* ⟹ *P*) ⟹ *P*
⟨*proof*⟩

lemma *degree-geI*:
assumes *MPoly-Type.coeff p m* ≠ *0*
shows *MPoly-Type.degree p i* ≥ *Poly-Mapping.lookup m i*
⟨*proof*⟩

lemma *monom-of-degree-exists*:
assumes *p* ≠ *0*
obtains *m* **where** *MPoly-Type.coeff p m* ≠ *0* *Poly-Mapping.lookup m i* = *MPoly-Type.degree p i*
⟨*proof*⟩

lemma *degree-leI*:
assumes ⋀*m*. *Poly-Mapping.lookup m i* > *n* ⟹ *MPoly-Type.coeff p m* = *0*
shows *MPoly-Type.degree p i* ≤ *n*
⟨*proof*⟩

lemma *coeff-gt-degree-eq-0*:
assumes *Poly-Mapping.lookup m i* > *MPoly-Type.degree p i*
shows *MPoly-Type.coeff p m* = *0*
⟨*proof*⟩

lemma *vars-altdef*: *vars p* = (⋃*m*∈{*m*. *MPoly-Type.coeff p m* ≠ *0*}. *keys m*)
⟨*proof*⟩

lemma *degree-pos-iff*: *MPoly-Type.degree p x* > *0* ⟷ *x* ∈ *vars p*
⟨*proof*⟩

lemma *degree-eq-0-iff*: *MPoly-Type.degree p x* = *0* ⟷ *x* ∉ *vars p*
⟨*proof*⟩

lemma *MPoly-Type-monom-zero*[*simp*]: *MPoly-Type.monom m 0* = *0*
⟨*proof*⟩

lemma *vars-monom-keys'*: *vars* (*MPoly-Type.monom m c*) = (*if c* = *0 then* {} *else keys m*)
⟨*proof*⟩

lemma *Const-eq-0-iff* [*simp*]: *Const c* = *0* ⟷ *c* = *0*
⟨*proof*⟩

lemma *monom-remove-key*: *MPoly-Type.monom m* (*a* :: *'a* :: *semiring-1*) = *MPoly-Type.monom* (*remove-key x m*) *a* ∗ *MPoly-Type.monom* (*Poly-Mapping.single x* (*lookup m x*)) *1*
⟨*proof*⟩

lemma *MPoly-Type-monom-0-iff*[*simp*]: *MPoly-Type.monom m x* = *0* ⟷ *x* = *0*
⟨*proof*⟩

lemma *vars-signof*[*simp*]: *vars* (*signof x*) = {}
⟨*proof*⟩

lemma *prod-mset-Const*: *prod-mset* (*image-mset Const A*) = *Const* (*prod-mset A*)
⟨*proof*⟩

lemma *Const-eq-product-iff*:
fixes *c* :: *'a* :: *idom*
assumes *c* ≠ *0*
shows *Const c* = *a* ∗ *b* ⟷ (∃ *a' b'*. *a* = *Const a'* ∧ *b* = *Const b'* ∧ *c* = *a'* ∗ *b'*)
⟨*proof*⟩

lemma *irreducible-Const-iff* [*simp*]:
irreducible (*Const* (*c* :: *'a* :: *idom*)) ⟷ *irreducible c*
⟨*proof*⟩

lemma *Const-dvd-Const-iff* [*simp*]: *Const a dvd Const b* ⟷ *a dvd b*
⟨*proof*⟩

The lemmas above should be moved into the right theories. The part below is on the new connection between multivariate polynomials and univariate polynomials.

The imported theories only allow a conversion from one-variable mpoly's to poly and vice-versa. However, we require a conversion from arbitrary mpoly's into poly's with mpolys as coefficients.

definition *mpoly-to-mpoly-poly* :: *nat* ⇒ *'a* :: *comm-ring-1 mpoly* ⇒ *'a mpoly poly*
where
mpoly-to-mpoly-poly x p = ($\sum$ *m* .
Polynomial.monom (*MPoly-Type.monom* (*remove-key x m*) (*MPoly-Type.coeff p m*)) (*lookup m x*))

lemma *mpoly-to-mpoly-poly-add* [*simp*]:
mpoly-to-mpoly-poly x (*p* + *q*) = *mpoly-to-mpoly-poly x p* + *mpoly-to-mpoly-poly x q*
⟨*proof*⟩

lemma *mpoly-to-mpoly-poly-monom*: *mpoly-to-mpoly-poly x* (*MPoly-Type.monom m a*) = *Polynomial.monom* (*MPoly-Type.monom* (*remove-key x m*) *a*) (*lookup m x*)
⟨*proof*⟩

lemma *remove-key-transfer* [*transfer-rule*]:
rel-fun (=) (*rel-fun* (*pcr-poly-mapping* (=) (=)) (*pcr-poly-mapping* (=) (=)))
(λ*k0 f k*. *f k when k* ≠ *k0*) *remove-key*
⟨*proof*⟩

lemma *remove-key-0* [*simp*]: *remove-key x 0* = *0*
⟨*proof*⟩

lemma *remove-key-single'* [*simp*]:
x ≠ *y* ⟹ *remove-key x* (*Poly-Mapping.single y n*) = *Poly-Mapping.single y n*
⟨*proof*⟩

lemma *poly-coeff-Sum-any*:
assumes *finite* {*x*. *f x* ≠ *0*}
shows *poly.coeff* (*Sum-any f*) *n* = *Sum-any* (λ*x*. *poly.coeff* (*f x*) *n*)
⟨*proof*⟩

lemma *coeff-coeff-mpoly-to-mpoly-poly*:
MPoly-Type.coeff (*poly.coeff* (*mpoly-to-mpoly-poly x p*) *n*) *m* =
(*MPoly-Type.coeff p* (*m* + *Poly-Mapping.single x n*) *when lookup m x* = *0*)
⟨*proof*⟩

lemma *mpoly-to-mpoly-poly-Const* [*simp*]:
mpoly-to-mpoly-poly x (*Const c*) = [:*Const c*:]
⟨*proof*⟩

lemma *mpoly-to-mpoly-poly-Var*:
mpoly-to-mpoly-poly x (*Var y*) = (*if x* = *y then* [:*0*, *1*:] *else* [:*Var y*:])
⟨*proof*⟩

lemma *mpoly-to-mpoly-poly-Var-this* [*simp*]:

mpoly-to-mpoly-poly x (*Var x*) = [:*0*, *1*:]
$x \neq y \Longrightarrow$ *mpoly-to-mpoly-poly x* (*Var y*) = [:*Var y*:]
⟨*proof*⟩

lemma *mpoly-to-mpoly-poly-uminus* [*simp*]:
mpoly-to-mpoly-poly x (−*p*) = −*mpoly-to-mpoly-poly x p*
⟨*proof*⟩

lemma *mpoly-to-mpoly-poly-diff* [*simp*]:
mpoly-to-mpoly-poly x (*p* − *q*) = *mpoly-to-mpoly-poly x p* − *mpoly-to-mpoly-poly x q*
⟨*proof*⟩

lemma *mpoly-to-mpoly-poly-0* [*simp*]:
mpoly-to-mpoly-poly x 0 = *0*
⟨*proof*⟩

lemma *mpoly-to-mpoly-poly-1* [*simp*]:
mpoly-to-mpoly-poly x 1 = *1*
⟨*proof*⟩

lemma *mpoly-to-mpoly-poly-of-nat* [*simp*]:
mpoly-to-mpoly-poly x (*of-nat n*) = *of-nat n*
⟨*proof*⟩

lemma *mpoly-to-mpoly-poly-of-int* [*simp*]:
mpoly-to-mpoly-poly x (*of-int n*) = *of-int n*
⟨*proof*⟩

lemma *mpoly-to-mpoly-poly-numeral* [*simp*]:
mpoly-to-mpoly-poly x (*numeral n*) = *numeral n*
⟨*proof*⟩

lemma *coeff-monom-mult'*:
MPoly-Type.coeff (*MPoly-Type.monom m a* ∗ *q*) *m'* =
(*a* ∗ *MPoly-Type.coeff q* (*m'* − *m*) *when lookup m'* ≥ *lookup m*)
⟨*proof*⟩

lemma *mpoly-to-mpoly-poly-mult-monom*:
mpoly-to-mpoly-poly x (*MPoly-Type.monom m a* ∗ *q*) =
Polynomial.monom (*MPoly-Type.monom* (*remove-key x m*) *a*) (*lookup m x*) ∗ *mpoly-to-mpoly-poly x q*
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *mpoly-to-mpoly-poly-mult* [*simp*]:
mpoly-to-mpoly-poly x (*p* ∗ *q*) = *mpoly-to-mpoly-poly x p* ∗ *mpoly-to-mpoly-poly x q*
⟨*proof*⟩

lemma *coeff-mpoly-to-mpoly-poly*:
  Polynomial.coeff (*mpoly-to-mpoly-poly x p*) *n* =
    Sum-any (λ*m*. *MPoly-Type.monom* (*remove-key x m*) (*MPoly-Type.coeff p m*) *when Poly-Mapping.lookup m x* = *n*)
  ⟨*proof*⟩

lemma *mpoly-coeff-to-mpoly-poly-coeff*:
  MPoly-Type.coeff p m = *MPoly-Type.coeff* (*poly.coeff* (*mpoly-to-mpoly-poly x p*) (*lookup m x*)) (*remove-key x m*)
⟨*proof*⟩

lemma *degree-mpoly-to-mpoly-poly* [*simp*]:
  Polynomial.degree (*mpoly-to-mpoly-poly x p*) = *MPoly-Type.degree p x*
⟨*proof*⟩

The upcoming lemma is similar to *reduce-nested-mpoly* (*extract-var ?p ?v*) = *?p*.

lemma *poly-mpoly-to-mpoly-poly*:
  poly (*mpoly-to-mpoly-poly x p*) (*Var x*) = *p*
⟨*proof*⟩

lemma *mpoly-to-mpoly-poly-eq-iff* [*simp*]:
  mpoly-to-mpoly-poly x p = *mpoly-to-mpoly-poly x q* ⟷ *p* = *q*
⟨*proof*⟩

Evaluation, i.e., insertion of concrete values is identical

lemma *insertion-mpoly-to-mpoly-poly*: **assumes** ⋀ *y*. *y* ≠ *x* ⟹ β *y* = α *y*
  shows *poly* (*map-poly* (*insertion* β) (*mpoly-to-mpoly-poly x p*)) (α *x*) = *insertion* α *p*
⟨*proof*⟩

lemma *mpoly-to-mpoly-poly-dvd-iff* [*simp*]:
  mpoly-to-mpoly-poly x p dvd mpoly-to-mpoly-poly x q ⟷ *p dvd q*
⟨*proof*⟩

lemma *vars-coeff-mpoly-to-mpoly-poly*: *vars* (*poly.coeff* (*mpoly-to-mpoly-poly x p*) *i*) ⊆ *vars p* − {*x*}
  ⟨*proof*⟩

locale *transfer-mpoly-to-mpoly-poly* =
  fixes *x* :: *nat*
begin

definition *R* :: ′*a* :: *comm-ring-1 mpoly poly* ⇒ ′*a mpoly* ⇒ *bool* **where**
  R p p′ ⟷ *p* = *mpoly-to-mpoly-poly x p*′

context

includes *lifting-syntax*
begin

lemma *transfer-0* [*transfer-rule*]: *R 0 0*
and *transfer-1* [*transfer-rule*]: *R 1 1*
and *transfer-Const* [*transfer-rule*]: *R* [:*Const c*:] (*Const c*)
and *transfer-uminus* [*transfer-rule*]: (*R* ===> *R*) *uminus uminus*
and *transfer-of-nat* [*transfer-rule*]: ((=) ===> *R*) *of-nat of-nat*
and *transfer-of-int* [*transfer-rule*]: ((=) ===> *R*) *of-nat of-nat*
and *transfer-numeral* [*transfer-rule*]: ((=) ===> *R*) *of-nat of-nat*
and *transfer-add* [*transfer-rule*]: (*R* ===> *R* ===> *R*) (+) (+)
and *transfer-diff* [*transfer-rule*]: (*R* ===> *R* ===> *R*) (+) (+)
and *transfer-mult* [*transfer-rule*]: (*R* ===> *R* ===> *R*) (∗) (∗)
and *transfer-dvd* [*transfer-rule*]: (*R* ===> *R* ===> (=)) (*dvd*) (*dvd*)
and *transfer-monom* [*transfer-rule*]:
((=) ===> (=) ===> *R*)
(λ*m a. Polynomial.monom* (*MPoly-Type.monom* (*remove-key x m*) *a*) (*lookup m x*))
MPoly-Type.monom
and *transfer-coeff* [*transfer-rule*]:
(*R* ===> (=) ===> (=))
(λ*p m. MPoly-Type.coeff* (*poly.coeff p* (*lookup m x*)) (*remove-key x m*))
MPoly-Type.coeff
and *transfer-degree* [*transfer-rule*]:
(*R* ===> (=)) *Polynomial.degree* (λ*p. MPoly-Type.degree p x*)
⟨*proof*⟩

lemma *transfer-vars* [*transfer-rule*]:
assumes [*transfer-rule*]: *R p p′*
shows (⋃*i. vars* (*poly.coeff p i*)) ∪ (*if Polynomial.degree p* = *0 then* {} *else* {*x*}) = *vars p′*
(**is** *?A* ∪ *?B* = -)
⟨*proof*⟩

lemma *right-total* [*transfer-rule*]: *right-total R*
⟨*proof*⟩

lemma *bi-unique* [*transfer-rule*]: *bi-unique R*
⟨*proof*⟩

end

end

lemma *mpoly-degree-mult-eq*:
fixes *p q* :: *′a* :: *idom mpoly*
assumes *p* ≠ *0 q* ≠ *0*

shows *MPoly-Type.degree* $(p * q)$ x = *MPoly-Type.degree* p x + *MPoly-Type.degree* q x
⟨*proof*⟩

Converts a multi-variate polynomial into a univariate polynomial via inserting values for all but one variable

definition *partial-insertion* :: (*nat* ⇒ *'a*) ⇒ *nat* ⇒ *'a* :: *comm-ring-1 mpoly* ⇒ *'a poly* **where**
partial-insertion α x p = *map-poly* (*insertion* α) (*mpoly-to-mpoly-poly* x p)

lemma *comm-ring-hom-insertion*: *comm-ring-hom* (*insertion* α)
⟨*proof*⟩

lemma *partial-insertion-add*: *partial-insertion* α x $(p + q)$ = *partial-insertion* α x p + *partial-insertion* α x q
⟨*proof*⟩

lemma *partial-insertion-monom*: *partial-insertion* α x (*MPoly-Type.monom* m a) = *Polynomial.monom* (*insertion* α (*MPoly-Type.monom* (*remove-key* x m) a)) (*lookup* m x)
⟨*proof*⟩

Partial insertion + insertion of last value is identical to (full) insertion

lemma *insertion-partial-insertion*: **assumes** $\bigwedge y.\ y \neq x \Longrightarrow \beta\ y = \alpha\ y$
shows *poly* (*partial-insertion* β x p) $(\alpha\ x)$ = *insertion* α p
⟨*proof*⟩

lemma *insertion-coeff-mpoly-to-mpoly-poly*[*simp*]:
insertion α (*coeff* (*mpoly-to-mpoly-poly* x p) k) = *coeff* (*partial-insertion* α x p) k
⟨*proof*⟩

lemma *degree-map-poly-Const*: *degree* (*map-poly* (*Const* :: *'a* :: *semiring-0* ⇒ -) f) = *degree* f
⟨*proof*⟩

lemma *degree-partial-insertion-le-mpoly*: *degree* (*partial-insertion* α x p) ≤ *degree* (*mpoly-to-mpoly-poly* x p)
⟨*proof*⟩

end

2.2 Exact Division of Multivariate Polynomials

theory *MPoly-Divide*
imports
Hermite-Lindemann.More-Multivariate-Polynomial-HLW
Polynomials.MPoly-Type-Class
Poly-Connection

begin

lemma *poly-lead-coeff-dvd-lead-coeff*:
  assumes *p dvd* (*q* :: ′*a* :: *idom poly*)
  shows *Polynomial.lead-coeff p dvd Polynomial.lead-coeff q*
  ⟨*proof*⟩

Since there is no particularly sensible algorithm for division with a remainder on multivariate polynomials, we define the following division operator that performs an exact division if possible and returns 0 otherwise.

instantiation *mpoly* :: (*comm-semiring-1*) *divide*
begin

definition *divide-mpoly* :: ′*a mpoly* ⇒ ′*a mpoly* ⇒ ′*a mpoly* **where**
  divide-mpoly x y = (*if* $y \neq 0 \wedge$ *y dvd x then THE z.* $x = y * z$ *else 0*)

instance ⟨*proof*⟩

end

instance *mpoly* :: (*idom*) *idom-divide*
  ⟨*proof*⟩

lemma (**in** *transfer-mpoly-to-mpoly-poly*) *transfer-div* [*transfer-rule*]:
  assumes [*transfer-rule*]: *R p′ p R q′ q*
  assumes *q dvd p*
  shows *R* (*p′ div q′*) (*p div q*)
  ⟨*proof*⟩

instantiation *mpoly* :: ({*normalization-semidom*, *idom*}) *normalization-semidom*
begin

definition *unit-factor-mpoly* :: ′*a mpoly* ⇒ ′*a mpoly* **where**
  unit-factor-mpoly p = *Const* (*unit-factor* (*lead-coeff p*))

definition *normalize-mpoly* :: ′*a mpoly* ⇒ ′*a mpoly* **where**
  normalize-mpoly p = *Rings.divide p* (*unit-factor p*)

lemma *unit-factor-mpoly-Const* [*simp*]:
  unit-factor (*Const c*) = *Const* (*unit-factor c*)
  ⟨*proof*⟩

lemma *normalize-mpoly-Const* [*simp*]:
  normalize (*Const c*) = *Const* (*normalize c*)
⟨*proof*⟩

instance ⟨*proof*⟩

end

The following is an exact division operator that can fail, i.e. if the divisor does not divide the dividend, it returns *None*.

definition *divide-option* :: *'a* :: *idom-divide* ⇒ *'a* ⇒ *'a option* (**infixl** *div? 70*)
where
divide-option p q = (*if q dvd p then Some* (*p div q*) *else None*)

We now show that exact division on the ring $R[X_1, \ldots, X_n]$ can be reduced to exact division on the ring $R[X_1, \ldots, X_n][X]$, i.e. we can go from *'a mpoly* to a *'a mpoly poly* where the coefficients have one variable less than the original multivariate polynomial. We basically simply use the isomorphism between these two rings.

```
lemma divide-option-mpoly:
  fixes p q :: 'a :: idom-divide mpoly
  shows p div? q = (let V = vars p ∪ vars q in
          (if V = {} then
             let a = MPoly-Type.coeff p 0; b = MPoly-Type.coeff q 0; c = a div b
             in  if b ∗ c = a then Some (Const c) else None
           else
             let x = Max V;
                 p′ = mpoly-to-mpoly-poly x p; q′ = mpoly-to-mpoly-poly x q
             in  case p′ div? q′ of
                    None ⇒ None
                  | Some r ⇒ Some (poly r (Var x)))) (is - = ?rhs)
⟨proof⟩
```

Next, we show that exact division on the ring $R[X_1, \ldots, X_n][Y]$ can be reduced to exact division on the ring $R[X_1, \ldots, X_n]$. This is essentially just polynomial division.

```
lemma divide-option-mpoly-poly:
  fixes p q :: 'a :: idom-divide mpoly poly
  shows p div? q =
          (if p = 0 then Some 0
           else if q = 0 then None
           else let dp = Polynomial.degree p; dq = Polynomial.degree q
                in  if dp < dq then None
                    else case Polynomial.lead-coeff p div? Polynomial.lead-coeff q of
                           None ⇒ None
                         | Some c ⇒ (
                             case (p − Polynomial.monom c (dp − dq) ∗ q) div? q of
                               None ⇒ None
                             | Some r ⇒ Some (Polynomial.monom c (dp − dq) + r)))
  (is - = ?rhs)
⟨proof⟩
```

These two equations now serve as two mutually recursive code equations that allow us to reduce exact division of multivariate polynomials to exact division of their coefficients. Termination of these code equations is not shown explicitly, but is obvious since one variable is eliminated in every step.

definition *divide-option-mpoly* :: *'a* :: *idom-divide mpoly* ⇒ -
  where *divide-option-mpoly* = *divide-option*

definition *divide-option-mpoly-poly* :: *'a* :: *idom-divide mpoly poly* ⇒ -
  where *divide-option-mpoly-poly* = *divide-option*

lemmas *divide-option-mpoly-code* [*code*] =
  divide-option-mpoly [*folded divide-option-mpoly-def divide-option-mpoly-poly-def*]

lemmas *divide-option-mpoly-poly-code* [*code*] =
  divide-option-mpoly-poly [*folded divide-option-mpoly-def divide-option-mpoly-poly-def*]

lemma *divide-mpoly-code* [*code*]:
  fixes *p q* :: *'a* :: *idom-divide mpoly*
  shows *p div q* = (*case divide-option-mpoly p q of None* ⇒ *0* | *Some r* ⇒ *r*)
  ⟨*proof*⟩

end

2.3 Implementation of Division on Multivariate Polynomials

theory *MPoly-Divide-Code*
  imports
    MPoly-Divide
    Polynomials.MPoly-Type-Class-FMap
    Polynomials.MPoly-Type-Univariate
begin

We now set up code equations for some of the operations that we will need, such as division, *mpoly-to-poly*, and *mpoly-to-mpoly-poly*.

lemma *mapping-of-MPoly*[*code*]: *mapping-of* (*MPoly p*) = *p*
  ⟨*proof*⟩

lift-definition *filter-pm* :: (*'a* ⇒ *bool*) ⇒ (*'a* $\Rightarrow_0$ *'b* :: *zero*) ⇒ (*'a* $\Rightarrow_0$ *'b*) **is**
  λP f x. if P x then f x else 0
  ⟨*proof*⟩

lemma *lookup-filter-pm*: *lookup* (*filter-pm P f*) *x* = (*if P x then lookup f x else 0*)
  ⟨*proof*⟩

lemma *filter-pm-code* [*code*]: *filter-pm P* (*Pm-fmap m*) = *Pm-fmap* (*fmfilter P m*)
  ⟨*proof*⟩

lemma *remove-key-conv-filter-pm* [*code*]: *remove-key x m* = *filter-pm* (λ*y*. *y* $\neq$ *x*) *m*
⟨*proof*⟩

lemma *finite-poly-coeff-nonzero*: *finite* {*n*. *poly.coeff p n* $\neq$ *0*}
⟨*proof*⟩

lemma *poly-degree-conv-Max*:
assumes *p* $\neq$ *0*
shows *Polynomial.degree p* = *Max* {*n*. *poly.coeff p n* $\neq$ *0*}
⟨*proof*⟩

lemma *mpoly-to-poly-code-aux*:
fixes *p* :: *'a* :: *comm-monoid-add mpoly* **and** *x* :: *nat*
defines *I* $\equiv$ (λ*m*. *lookup m x*) ‘ *Set.filter* (λ*m*. $\forall$ *y*$\in$*keys m*. *y* = *x*) (*keys* (*mapping-of p*))
shows *I* = {*n*. *poly.coeff* (*mpoly-to-poly x p*) *n* $\neq$ *0*}
and *mpoly-to-poly x p* = *0* $\longleftrightarrow$ *I* = {}
and *I* $\neq$ {} $\Longrightarrow$ *Polynomial.degree* (*mpoly-to-poly x p*) = *Max I*
⟨*proof*⟩

lemma *mpoly-to-poly-code* [*code*]:
Polynomial.coeffs (*mpoly-to-poly x p*) =
(*let I* = (λ*m*. *lookup m x*) ‘ *Set.filter* (λ*m*. $\forall$ *y*$\in$*keys m*. *y* = *x*) (*keys* (*mapping-of p*))
in if I = {} *then* [] *else map* (λ*n*. *MPoly-Type.coeff p* (*Poly-Mapping.single x n*)) [*0*..<*Max I* + *1*])
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

fun *mpoly-to-mpoly-poly-impl-aux1* :: *nat* $\Rightarrow$ ((*nat* $\Rightarrow_0$ *nat*) $\times$ *'a*) *list* $\Rightarrow$ *nat* $\Rightarrow$ ((*nat* $\Rightarrow_0$ *nat*) $\times$ *'a*) *list* **where**
mpoly-to-mpoly-poly-impl-aux1 i [] *j* = []
| *mpoly-to-mpoly-poly-impl-aux1 i* ((*mon'*, *c*) # *xs*) *j* =
(*if lookup mon' i* = *j then* [(*remove-key i mon'*, *c*)] *else* []) @ *mpoly-to-mpoly-poly-impl-aux1 i xs j*

lemma *mpoly-to-mpoly-poly-impl-aux1-altdef*:
mpoly-to-mpoly-poly-impl-aux1 i xs j =
map (λ(*mon*, *c*). (*remove-key i mon*, *c*)) (*filter* (λ(*mon*, *c*). *lookup mon i* = *j*) *xs*)
⟨*proof*⟩

lemma *map-of-mpoly-to-mpoly-poly-impl-aux1*:
map-of (*mpoly-to-mpoly-poly-impl-aux1 i xs j*) = (λ*mon*.
(*if lookup mon i* > *0 then None*

else map-of xs (mon + Poly-Mapping.single i j)))
⟨*proof*⟩

lemma *lookup0-fmap-of-list-mpoly-to-mpoly-poly-impl-aux1*:
lookup0 (fmap-of-list (mpoly-to-mpoly-poly-impl-aux1 i xs j)) = (λmon. lookup0 (fmap-of-list xs) (mon + Poly-Mapping.single i j) when lookup mon i = 0)
⟨*proof*⟩

definition *mpoly-to-mpoly-poly-impl-aux2* **where**
mpoly-to-mpoly-poly-impl-aux2 i p j = poly.coeff (mpoly-to-mpoly-poly i p) j

lemma *coeff-MPoly*: *MPoly-Type.coeff (MPoly f) m = lookup f m*
⟨*proof*⟩

lemma *mpoly-to-mpoly-poly-impl-aux2-code* [*code*]:
mpoly-to-mpoly-poly-impl-aux2 i (MPoly (Pm-fmap (fmap-of-list xs))) j = MPoly (Pm-fmap (fmap-of-list (mpoly-to-mpoly-poly-impl-aux1 i xs j)))
⟨*proof*⟩

definition *mpoly-to-mpoly-poly-impl* :: *nat ⇒ 'a :: comm-ring-1 mpoly ⇒ 'a mpoly list* **where**
mpoly-to-mpoly-poly-impl x p = (if p = 0 then [] else map (mpoly-to-mpoly-poly-impl-aux2 x p) [0..<Suc (MPoly-Type.degree p x)])

lemma *mpoly-to-mpoly-poly-eq-0-iff* [*simp*]: *mpoly-to-mpoly-poly x p = 0 ⟷ p = 0*
⟨*proof*⟩

lemma *mpoly-to-mpoly-poly-code* [*code*]:
Polynomial.coeffs (mpoly-to-mpoly-poly x p) = mpoly-to-mpoly-poly-impl x p
⟨*proof*⟩

value *mpoly-to-mpoly-poly 0 (Var 0 ^ 2 + Var 0 * Var 1 + Var 1 ^ 2 :: int mpoly)*

value *Rings.divide (Var 0 ^ 2 * Var 1 + Var 0 * Var 1 ^ 2 :: int mpoly) (Var 1)*

end

2.4 Class Instances for Multivariate Polynomials and Containers

theory *MPoly-Container*
imports
Polynomials.MPoly-Type-Class
Containers.Set-Impl
begin

Basic setup for using multivariate polynomials in combination with container framework.

derive (*eq*) *ceq poly-mapping*
derive (*dlist*) *set-impl poly-mapping*
derive (*no*) *ccompare poly-mapping*

end

2.5 Resultants of Multivariate Polynomials

We utilize the conversion of multivariate polynomials into univariate polynomials for the definition of the resultant of multivariate polynomials via the resultant for univariate polynomials. In this way, we can use the algorithm to efficiently compute resultants for the multivariate case.

theory *Multivariate-Resultant*
  imports
    Poly-Connection
    Algebraic-Numbers.Resultant
    Subresultants.Subresultant
    MPoly-Divide-Code
    MPoly-Container
begin

hide-const (**open**)
  MPoly-Type.degree
  MPoly-Type.coeff
  Symmetric-Polynomials.lead-coeff

lemma *det-sylvester-matrix-higher-degree*:
  det (*sylvester-mat-sub* (*degree f* + *n*) (*degree g*) *f g*)
  = *det* (*sylvester-mat-sub* (*degree f*) (*degree g*) *f g*) ∗ (*lead-coeff g* ∗ (−*1*)^(*degree g*))^*n*
⟨*proof*⟩

The conversion of multivariate into univariate polynomials permits us to define resultants in the multivariate setting. Since in our application one of the polynomials is already univariate, we use a non-symmetric definition where only one of the input polynomials is multivariate.

definition *resultant-mpoly-poly* :: *nat* ⇒ *'a* :: *comm-ring-1 mpoly* ⇒ *'a poly* ⇒ *'a mpoly* **where**
  resultant-mpoly-poly x p q = *resultant* (*mpoly-to-mpoly-poly x p*) (*map-poly Const q*)

This lemma tells us that there is only a minor difference between computing the multivariate resultant and then plugging in values, or first inserting values and then evaluate the univariate resultant.

lemma *insertion-resultant-mpoly-poly*: *insertion α* (*resultant-mpoly-poly x p q*) = *resultant* (*partial-insertion α x p*) *q* ∗

(*lead-coeff q* $*$ ($-$*1*)^ *degree q*)^(*degree* (*mpoly-to-mpoly-poly x p*) $-$ *degree* (*partial-insertion* α *x p*))
⟨*proof*⟩

lemma *insertion-resultant-mpoly-poly-zero*: **fixes** *q* :: *'a* :: *idom poly*
assumes *q*: *q* $\neq$ *0*
shows *insertion* α (*resultant-mpoly-poly x p q*) = *0* $\longleftrightarrow$ *resultant* (*partial-insertion* α *x p*) *q* = *0*
⟨*proof*⟩

lemma *vars-resultant*: *vars* (*resultant p q*) $\subseteq \bigcup$ (*vars* ‘ (*range* (*coeff p*) $\cup$ *range* (*coeff q*)))
⟨*proof*⟩

By taking the resultant, one variable is deleted.

lemma *vars-resultant-mpoly-poly*: *vars* (*resultant-mpoly-poly x p q*) $\subseteq$ *vars p* $-$ {*x*}

⟨*proof*⟩

For resultants, we manually have to select the implementation that works on integral domains, because there is no factorial ring instance for *int mpoly*.

lemma *resultant-mpoly-poly-code*[*code*]:
resultant-mpoly-poly x p q = *resultant-impl-basic* (*mpoly-to-mpoly-poly x p*) (*map-poly Const q*)
⟨*proof*⟩

end

3 Testing for Integrality and Conversion to Integers

theory *Is-Int-To-Int*
imports
Polynomial-Interpolation.Is-Rat-To-Rat
begin

lemma *inv-of-rat*: *inv of-rat* (*of-rat x*) = *x*
⟨*proof*⟩

lemma *of-rat-Ints-iff*: ((*of-rat x* :: *'a* :: *field-char-0*) $\in \mathbb{Z}$) = ($x \in \mathbb{Z}$)
⟨*proof*⟩

lemma *is-int-code*[*code-unfold*]:
shows ($x \in \mathbb{Z}$) = (*is-rat x* $\wedge$ *is-int-rat* (*to-rat x*))
⟨*proof*⟩

definition *to-int* :: *'a* :: *is-rat* $\Rightarrow$ *int* **where**
to-int x = *int-of-rat* (*to-rat x*)

lemma *of-int-to-int*: $x \in \mathbb{Z} \Longrightarrow$ *of-int* (*to-int* x) = x
⟨*proof*⟩

lemma *to-int-of-int*: *to-int* (*of-int* x) = x
⟨*proof*⟩

lemma *to-rat-complex-of-real*[*simp*]: *to-rat* (*complex-of-real* x) = *to-rat* x
⟨*proof*⟩

lemma *to-int-complex-of-real*[*simp*]: *to-int* (*complex-of-real* x) = *to-int* x
⟨*proof*⟩

end

4 Representing Roots of Polynomials with Algebraic Coefficients

We provide an algorithm to compute a non-zero integer polynomial q from a polynomial p with algebraic coefficients such that all roots of p are also roots of q.

In this way, we have a constructive proof that the set of complex algebraic numbers is algebraically closed.

theory *Roots-of-Algebraic-Poly*
imports
Algebraic-Numbers.Complex-Algebraic-Numbers
Multivariate-Resultant
Is-Int-To-Int
begin

4.1 Preliminaries

hide-const (**open**) *up-ring.monom*
hide-const (**open**) *MPoly-Type.monom*

lemma *map-mpoly-Const*: f *0* = *0* $\Longrightarrow$ *map-mpoly* f (*Const* i) = *Const* (f i)
⟨*proof*⟩

lemma *map-mpoly-Var*: f *1* = *1* $\Longrightarrow$ *map-mpoly* (f :: $'b$ :: *zero-neq-one* ⇒ -) (*Var* i) = *Var* i
⟨*proof*⟩

lemma *map-mpoly-monom*: f *0* = *0* $\Longrightarrow$ *map-mpoly* f (*MPoly-Type.monom* m a) = (*MPoly-Type.monom* m (f a))
⟨*proof*⟩

lemma *remove-key-single'*:

remove-key v (*Poly-Mapping.single w n*) = (*if v* = *w then 0 else Poly-Mapping.single w n*)
⟨*proof*⟩

context *comm-monoid-add-hom*
begin
lemma *hom-Sum-any*: **assumes** *fin*: *finite* {*x*. *f x* ≠ *0*}
shows *hom* (*Sum-any f*) = *Sum-any* (λ *x*. *hom* (*f x*))
⟨*proof*⟩

lemma *comm-monoid-add-hom-mpoly-map*: *comm-monoid-add-hom* (*map-mpoly hom*)
⟨*proof*⟩

lemma *map-mpoly-hom-Const*: *map-mpoly hom* (*Const i*) = *Const* (*hom i*)
⟨*proof*⟩

lemma *map-mpoly-hom-monom*: *map-mpoly hom* (*MPoly-Type.monom m a*) = *MPoly-Type.monom m* (*hom a*)
⟨*proof*⟩
end

context *comm-ring-hom*
begin
lemma *mpoly-to-poly-map-mpoly-hom*: *mpoly-to-poly x* (*map-mpoly hom p*) = *map-poly hom* (*mpoly-to-poly x p*)
⟨*proof*⟩

lemma *comm-ring-hom-mpoly-map*: *comm-ring-hom* (*map-mpoly hom*)
⟨*proof*⟩

lemma *mpoly-to-mpoly-poly-map-mpoly-hom*:
mpoly-to-mpoly-poly x (*map-mpoly hom p*) = *map-poly* (*map-mpoly hom*) (*mpoly-to-mpoly-poly x p*)
⟨*proof*⟩
end

context *inj-comm-ring-hom*
begin
lemma *inj-comm-ring-hom-mpoly-map*: *inj-comm-ring-hom* (*map-mpoly hom*)
⟨*proof*⟩

lemma *resultant-mpoly-poly-hom*: *resultant-mpoly-poly x* (*map-mpoly hom p*) (*map-poly hom q*) = *map-mpoly hom* (*resultant-mpoly-poly x p q*)
⟨*proof*⟩
end

lemma *map-insort-key*: **assumes** [*simp*]: ⋀ *x y*. *g1 x* ≤ *g1 y* ⟷ *g2* (*f x*) ≤ *g2* (*f y*)

shows *map f* (*insort-key g1 a xs*) = *insort-key g2* (*f a*) (*map f xs*)
⟨*proof*⟩

lemma *map-sort-key*: **assumes** [*simp*]: ⋀ *x y*. *g1 x* ≤ *g1 y* ⟷ *g2* (*f x*) ≤ *g2* (*f y*)
shows *map f* (*sort-key g1 xs*) = *sort-key g2* (*map f xs*)
⟨*proof*⟩

hide-const (**open**) *MPoly-Type.degree*
hide-const (**open**) *MPoly-Type.coeffs*
hide-const (**open**) *MPoly-Type.coeff*
hide-const (**open**) *Symmetric-Polynomials.lead-coeff*

4.2 More Facts about Resultants

lemma *resultant-iff-coprime-main*:
fixes *f g* :: ′*a* :: *field poly*
assumes *deg*: *degree f* > *0* ∨ *degree g* > *0*
shows *resultant f g* = *0* ⟷ ¬ *coprime f g*
⟨*proof*⟩

lemma *resultant-zero-iff-coprime*: **fixes** *f g* :: ′*a* :: *field poly*
assumes *f* ≠ *0* ∨ *g* ≠ *0*
shows *resultant f g* = *0* ⟷ ¬ *coprime f g*
⟨*proof*⟩

The problem with the upcoming lemma is that ”root” and ”irreducibility” refer to the same type. In the actual application we interested in ”irreducibility” over the integers, but the roots we are interested in are either real or complex.

lemma *resultant-zero-iff-common-root-irreducible*: **fixes** *f g* :: ′*a* :: *field poly*
assumes *irr*: *irreducible g*
and *root*: *poly g a* = *0*
shows *resultant f g* = *0* ⟷ (∃ *x*. *poly f x* = *0* ∧ *poly g x* = *0*)
⟨*proof*⟩

lemma *resultant-zero-iff-common-root-complex*: **fixes** *f g* :: *complex poly*
assumes *g*: *g* ≠ *0*
shows *resultant f g* = *0* ⟷ (∃ *x*. *poly f x* = *0* ∧ *poly g x* = *0*)
⟨*proof*⟩

4.3 Systems of Polynomials

Definition of solving a system of polynomials, one being multivariate

definition *mpoly-polys-solution* :: ′*a* :: *field mpoly* ⇒ (*nat* ⇒ ′*a poly*) ⇒ *nat set* ⇒ (*nat* ⇒ ′*a*) ⇒ *bool* **where**
mpoly-polys-solution p qs N α = (

insertion α *p* = *0* $\wedge$
($\forall$ *i* $\in$ *N*. *poly* (*qs i*) (α (*Suc i*)) = *0*))

The upcoming lemma shows how to eliminate single variables in multi-variate root-problems. Because of the problem mentioned in *resultant-zero-iff-common-root-irreducible* we here restrict to polynomials over the complex numbers. Since the result computations are homomorphisms, we are able to lift it to integer polynomials where we are interested in real or complex roots.

lemma *resultant-mpoly-polys-solution*: **fixes** *p* :: *complex mpoly*
assumes *nz*: *0* $\notin$ *qs* ‘ *N*
and *i*: *i* $\in$ *N*
shows *mpoly-polys-solution* (*resultant-mpoly-poly* (*Suc i*) *p* (*qs i*)) *qs* (*N* − {*i*}) α
$\longleftrightarrow$ ($\exists$ *v*. *mpoly-polys-solution p qs N* (α((*Suc i*) := *v*)))
⟨*proof*⟩

We now restrict solutions to be evaluated to zero outside the variable range. Then there are only finitely many solutions for our applications.

definition *mpoly-polys-zero-solution* :: *'a* :: *field mpoly* ⇒ (*nat* ⇒ *'a poly*) ⇒ *nat set* ⇒ (*nat* ⇒ *'a*) ⇒ *bool* **where**
mpoly-polys-zero-solution p qs N α = (*mpoly-polys-solution p qs N* α
$\wedge$ ($\forall$ *i*. *i* $\notin$ *insert 0* (*Suc* ‘ *N*) $\longrightarrow$ α *i* = *0*))

lemma *resultant-mpoly-polys-zero-solution*: **fixes** *p* :: *complex mpoly*
assumes *nz*: *0* $\notin$ *qs* ‘ *N*
and *i*: *i* $\in$ *N*
shows
mpoly-polys-zero-solution (*resultant-mpoly-poly* (*Suc i*) *p* (*qs i*)) *qs* (*N* − {*i*}) α
$\Longrightarrow$ $\exists$ *v*. *mpoly-polys-zero-solution p qs N* (α(*Suc i* := *v*))
mpoly-polys-zero-solution p qs N α
$\Longrightarrow$ *mpoly-polys-zero-solution* (*resultant-mpoly-poly* (*Suc i*) *p* (*qs i*)) *qs* (*N* − {*i*}) (α(*Suc i* := *0*))
⟨*proof*⟩

The following two lemmas show that if we start with a system of polynomials with finitely many solutions, then the resulting polynomial cannot be the zero-polynomial.

lemma *finite-resultant-mpoly-polys-non-empty*: **fixes** *p* :: *complex mpoly*
assumes *nz*: *0* $\notin$ *qs* ‘ *N*
and *i*: *i* $\in$ *N*
and *fin*: *finite* {α. *mpoly-polys-zero-solution p qs N* α}
shows *finite* {α. *mpoly-polys-zero-solution* (*resultant-mpoly-poly* (*Suc i*) *p* (*qs i*)) *qs* (*N* − {*i*}) α}
⟨*proof*⟩

lemma *finite-resultant-mpoly-polys-empty*: **fixes** *p* :: *complex mpoly*
assumes *finite* {α. *mpoly-polys-zero-solution p qs* {} α}
shows *p* $\neq$ *0*
⟨*proof*⟩

4.4 Elimination of Auxiliary Variables

fun *eliminate-aux-vars* :: *'a* :: *comm-ring-1 mpoly* ⇒ (*nat* ⇒ *'a poly*) ⇒ *nat list* ⇒ *'a poly* **where**
eliminate-aux-vars p qs [] = *mpoly-to-poly 0 p*
| *eliminate-aux-vars p qs* (*i* # *is*) = *eliminate-aux-vars* (*resultant-mpoly-poly* (*Suc i*) *p* (*qs i*)) *qs is*

lemma *eliminate-aux-vars-of-int-poly*:
eliminate-aux-vars (*map-mpoly* (*of-int* :: - ⇒ *'a* :: {*comm-ring-1*,*ring-char-0*}) *mp*) (*of-int-poly* ∘ *qs*) *is*
= *of-int-poly* (*eliminate-aux-vars mp qs is*)
⟨*proof*⟩

The polynomial of the elimination process will represent the first value α (0::$'a$) of any solution to the multi-polynomial problem.

lemma *eliminate-aux-vars*: **fixes** *p* :: *complex mpoly*
assumes *distinct is*
and *vars p* ⊆ *insert 0* (*Suc* ' *set is*)
and *finite* {α. *mpoly-polys-zero-solution p qs* (*set is*) α}
and *0* ∉ *qs* ' *set is*
and *mpoly-polys-solution p qs* (*set is*) α
shows *poly* (*eliminate-aux-vars p qs is*) (α *0*) = *0* ∧ *eliminate-aux-vars p qs is* ≠ *0*
⟨*proof*⟩

4.5 A Representing Polynomial for the Roots of a Polynomial with Algebraic Coefficients

First convert an algebraic polynomial into a system of integer polynomials.

definition *initial-root-problem* :: *'a* :: {*is-rat*,*field-gcd*} *poly* ⇒ *int mpoly* × (*nat* × *'a* × *int poly*) *list* **where**
initial-root-problem p = (*let*
n = *degree p*;
cs = *coeffs p*;
rcs = *remdups* (*filter* (λ *c*. *c* ∉ ℤ) *cs*);
pairs = *map* (λ *c*. (*c*, *min-int-poly c*)) *rcs*;
spairs = *sort-key* (λ (*c*,*f*). *degree f*) *pairs*; — sort by degree so that easy computations will be done first
triples = *zip* [*0* ..< *length spairs*] *spairs*;
mpoly = (*sum* (λ *i*. *let c* = *coeff p i in*
MPoly-Type.monom (*Poly-Mapping.single 0 i*) *1* ∗ — x_0^i ∗ ...
(*case find* (λ (*j*,*d*,*f*). *d* = *c*) *triples of*
None ⇒ *Const* (*to-int c*)
| *Some* (*j*,*pair*) ⇒ *Var* (*Suc j*)))
{..*n*})
in (*mpoly*, *triples*))

And then eliminate all auxiliary variables

definition *representative-poly* :: *'a* :: {*is-rat*,*field-char-0*,*field-gcd*} *poly* ⇒ *int poly*
where
  representative-poly p = (*case initial-root-problem p of*
    (*mp*, *triples*) ⇒
    let is = *map fst triples*;
        qs = (λ *j*. *snd* (*snd* (*triples* ! *j*)))
      in eliminate-aux-vars mp qs is)

4.6 Soundness Proof for Complex Algebraic Polynomials

lemma *get-representative-complex*: **fixes** *p* :: *complex poly*
  assumes *p*: *p* ≠ *0*
  and *algebraic*: *Ball* (*set* (*coeffs p*)) *algebraic*
  and *res*: *initial-root-problem p* = (*mp*, *triples*)
  and *is*: *is* = *map fst triples*
  and *qs*: ⋀ *j*. *j* < *length is* ⟹ *qs j* = *snd* (*snd* (*triples* ! *j*))
  and *root*: *poly p x* = *0*
shows *eliminate-aux-vars mp qs is represents x*
⟨*proof*⟩

lemma *representative-poly-complex*: **fixes** *x* :: *complex*
  assumes *p*: *p* ≠ *0*
    and *algebraic*: *Ball* (*set* (*coeffs p*)) *algebraic*
    and *root*: *poly p x* = *0*
  shows *representative-poly p represents x*
⟨*proof*⟩

4.7 Soundness Proof for Real Algebraic Polynomials

We basically use the result for complex algebraic polynomials which are a superset of real algebraic polynomials.

lemma *initial-root-problem-complex-of-real-poly*:
  initial-root-problem (*map-poly complex-of-real p*) =
  map-prod id (*map* (*map-prod id* (*map-prod complex-of-real id*))) (*initial-root-problem p*)
⟨*proof*⟩

lemma *representative-poly-real*: **fixes** *x* :: *real*
  assumes *p*: *p* ≠ *0*
  and *algebraic*: *Ball* (*set* (*coeffs p*)) *algebraic*
  and *root*: *poly p x* = *0*
shows *representative-poly p represents x*
⟨*proof*⟩

4.8 Algebraic Closedness of Complex Algebraic Numbers

lemma *complex-algebraic-numbers-are-algebraically-closed*:

assumes *nc*: ¬ *constant* (*poly p*)
and *alg*: *Ball* (*set* (*coeffs p*)) *algebraic*
shows ∃ *z* :: *complex*. *algebraic z* ∧ *poly p z* = *0*
⟨*proof*⟩

end

4.9 Executable Version to Compute Representative Polynomials

theory *Roots-of-Algebraic-Poly-Impl*
imports
Roots-of-Algebraic-Poly
Polynomials.MPoly-Type-Class-FMap
begin

We need to specialize our code to real and complex polynomials, since *algebraic* and *min-int-poly* are not executable in their parametric versions.

definition *initial-root-problem-real* :: *real poly* ⇒ - **where**
[*simp*]: *initial-root-problem-real p* = *initial-root-problem p*

definition *initial-root-problem-complex* :: *complex poly* ⇒ - **where**
[*simp*]: *initial-root-problem-complex p* = *initial-root-problem p*

lemmas *initial-root-problem-code* =
initial-root-problem-real-def[*unfolded initial-root-problem-def*]
initial-root-problem-complex-def[*unfolded initial-root-problem-def*]

declare *initial-root-problem-code*[*code*]

lemma *initial-root-problem-code-unfold*[*code-unfold*]:
initial-root-problem = *initial-root-problem-complex*
initial-root-problem = *initial-root-problem-real*
⟨*proof*⟩

definition *representative-poly-real* :: *real poly* ⇒ - **where**
[*simp*]: *representative-poly-real p* = *representative-poly p*

definition *representative-poly-complex* :: *complex poly* ⇒ - **where**
[*simp*]: *representative-poly-complex p* = *representative-poly p*

lemmas *representative-poly-code* =
representative-poly-real-def[*unfolded representative-poly-def*]
representative-poly-complex-def[*unfolded representative-poly-def*]

declare *representative-poly-code*[*code*]

lemma *representative-poly-code-unfold*[*code-unfold*]:

```
  representative-poly = representative-poly-complex
  representative-poly = representative-poly-real
  ⟨proof⟩

end
```

5 Root Filter via Interval Arithmetic

5.1 Generic Framework

We provide algorithms for finding all real or complex roots of a polynomial from a superset of the roots via interval arithmetic. These algorithms are much faster than just evaluating the polynomial via algebraic number computations.

```
theory Roots-via-IA
  imports
    Algebraic-Numbers.Interval-Arithmetic
begin

definition interval-of-real :: nat ⇒ real ⇒ real interval where
  interval-of-real prec x =
     (if is-rat x then Interval x x
      else let n = 2 ^ prec; x' = x * of-int n
           in  Interval (of-rat (Rat.Fract ⌊x'⌋ n)) (of-rat (Rat.Fract ⌈x'⌉ n)))

definition interval-of-complex :: nat ⇒ complex ⇒ complex-interval where
  interval-of-complex prec z =
     Complex-Interval (interval-of-real prec (Re z)) (interval-of-real prec (Im z))

fun poly-interval :: 'a :: {plus,times,zero} list ⇒ 'a ⇒ 'a where
  poly-interval [] - = 0
| poly-interval [c] - = c
| poly-interval (c # cs) x = c + x * poly-interval cs x

definition filter-fun-complex :: complex poly ⇒ nat ⇒ complex ⇒ bool where
  filter-fun-complex p = (let c = coeffs p in
     (λ prec. let cs = map (interval-of-complex prec) c
     in (λ x. 0 ∈c poly-interval cs (interval-of-complex prec x))))

definition filter-fun-real :: real poly ⇒ nat ⇒ real ⇒ bool where
  filter-fun-real p = (let c = coeffs p in
     (λ prec. let cs = map (interval-of-real prec) c
     in (λ x. 0 ∈i poly-interval cs (interval-of-real prec x))))

definition genuine-roots :: - poly ⇒ - list ⇒ - list where
  genuine-roots p xs = filter (λx. poly p x = 0) xs

lemma zero-in-interval-0 [simp, intro]: 0 ∈i 0
```

⟨*proof*⟩

lemma *zero-in-complex-interval-0* [*simp*, *intro*]: $0 \in_c 0$
⟨*proof*⟩

lemma *length-coeffs-degree'*:
length (*coeffs p*) = (*if p* = *0 then 0 else Suc* (*degree p*))
⟨*proof*⟩

lemma *poly-in-poly-interval-complex*:
assumes *list-all2* (λc *ivl*. $c \in_c$ *ivl*) (*coeffs p*) *cs* $x \in_c$ *ivl*
shows *poly p* $x \in_c$ *poly-interval cs ivl*
⟨*proof*⟩

lemma *poly-in-poly-interval-real*: **fixes** *x* :: *real*
assumes *list-all2* (λc *ivl*. $c \in_i$ *ivl*) (*coeffs p*) *cs* $x \in_i$ *ivl*
shows *poly p* $x \in_i$ *poly-interval cs ivl*
⟨*proof*⟩

lemma *in-interval-of-real* [*simp*, *intro*]: $x \in_i$ *interval-of-real prec x*
⟨*proof*⟩

lemma *in-interval-of-complex* [*simp*, *intro*]: $z \in_c$ *interval-of-complex prec z*
⟨*proof*⟩

lemma *distinct-genuine-roots* [*simp*, *intro*]:
distinct xs $\Longrightarrow$ *distinct* (*genuine-roots p xs*)
⟨*proof*⟩

definition *filter-fun* :: *'a poly* $\Rightarrow$ (*nat* $\Rightarrow$ *'a* :: *comm-ring* $\Rightarrow$ *bool*) $\Rightarrow$ *bool* **where**
filter-fun p f = ($\forall$ *n x*. *poly p x* = *0* $\longrightarrow$ *f n x*)

lemma *filter-fun-complex*: *filter-fun p* (*filter-fun-complex p*)
⟨*proof*⟩

lemma *filter-fun-real*: *filter-fun p* (*filter-fun-real p*)
⟨*proof*⟩

context
fixes *p* :: *'a* :: *comm-ring poly* **and** *f*
assumes *ff*: *filter-fun p f*
begin

lemma *genuine-roots-step*:
genuine-roots p xs = *genuine-roots p* (*filter* (*f prec*) *xs*)
⟨*proof*⟩

lemma *genuine-roots-step-preserve-invar*:

assumes $\{z.\ poly\ p\ z = 0\} \subseteq set\ xs$
shows $\{z.\ poly\ p\ z = 0\} \subseteq set\ (filter\ (f\ prec)\ xs)$
⟨*proof*⟩
end

lemma *genuine-roots-finish*:
fixes $p :: {'}a :: field\text{-}char\text{-}0\ poly$
assumes $\{z.\ poly\ p\ z = 0\} \subseteq set\ xs\ distinct\ xs$
assumes $length\ xs = card\ \{z.\ poly\ p\ z = 0\}$
shows $genuine\text{-}roots\ p\ xs = xs$
⟨*proof*⟩

This is type of the initial search problem. It consists of a polynomial p, a list xs of candidate roots, the cardinality of the set of roots of p and a filter function to drop non-roots that is parametric in a precision parameter.

typedef (**overloaded**) *'a genuine-roots-aux* =
$\{(p :: {'}a :: field\text{-}char\text{-}0\ poly,\ xs,\ n,\ ff).$
$distinct\ xs \wedge$
$\{z.\ poly\ p\ z = 0\} \subseteq set\ xs \wedge$
$card\ \{z.\ poly\ p\ z = 0\} = n \wedge$
$filter\text{-}fun\ p\ ff\}$
⟨*proof*⟩

setup-lifting *type-definition-genuine-roots-aux*

lift-definition $genuine\text{-}roots' :: nat \Rightarrow {'}a :: field\text{-}char\text{-}0\ genuine\text{-}roots\text{-}aux \Rightarrow {'}a\ list$ **is**
$\lambda prec\ (p,\ xs,\ n,\ ff).\ genuine\text{-}roots\ p\ xs$ ⟨*proof*⟩

lift-definition $genuine\text{-}roots\text{-}impl\text{-}step' :: nat \Rightarrow {'}a :: field\text{-}char\text{-}0\ genuine\text{-}roots\text{-}aux \Rightarrow {'}a\ genuine\text{-}roots\text{-}aux$ **is**
$\lambda prec\ (p,\ xs,\ n,\ ff).\ (p,\ filter\ (ff\ prec)\ xs,\ n,\ ff)$
⟨*proof*⟩

lift-definition $gr\text{-}poly :: {'}a :: field\text{-}char\text{-}0\ genuine\text{-}roots\text{-}aux \Rightarrow {'}a\ poly$ **is**
$\lambda(p :: {'}a\ poly,\ \text{-},\ \text{-},\ \text{-}).\ p$ ⟨*proof*⟩

lift-definition $gr\text{-}list :: {'}a :: field\text{-}char\text{-}0\ genuine\text{-}roots\text{-}aux \Rightarrow {'}a\ list$ **is**
$\lambda(\text{-},\ xs :: {'}a\ list,\ \text{-},\ \text{-}).\ xs$ ⟨*proof*⟩

lift-definition $gr\text{-}numroots :: {'}a :: field\text{-}char\text{-}0\ genuine\text{-}roots\text{-}aux \Rightarrow nat$ **is**
$\lambda(\text{-},\ \text{-},\ n,\ \text{-}).\ n$ ⟨*proof*⟩

lemma *genuine-roots'-code* [*code*]:
$genuine\text{-}roots'\ prec\ gr =$
$(if\ length\ (gr\text{-}list\ gr) = gr\text{-}numroots\ gr\ then\ gr\text{-}list\ gr$
$else\ genuine\text{-}roots'\ (2 * prec)\ (genuine\text{-}roots\text{-}impl\text{-}step'\ prec\ gr))$
⟨*proof*⟩

definition *initial-precision* :: *nat* **where** *initial-precision* = *10*

definition *genuine-roots-impl* :: *'a genuine-roots-aux* ⇒ *'a* :: *field-char-0 list* **where**
genuine-roots-impl = *genuine-roots' initial-precision*

lemma *genuine-roots-impl*: *set* (*genuine-roots-impl p*) = {*z*. *poly* (*gr-poly p*) *z* = *0*}
distinct (*genuine-roots-impl p*)
⟨*proof*⟩

end

6 Roots of Real and Complex Algebraic Polynomials

We are now able to actually compute all roots of polynomials with real and complex algebraic coefficients. The main addition to calculating the representative polynomial for a superset of all roots is to find the genuine roots. For this we utilize the approximation algorithm via interval arithmetic.

theory *Roots-of-Real-Complex-Poly*
imports
Roots-of-Algebraic-Poly-Impl
Roots-via-IA
MPoly-Container
begin

hide-const (**open**) *Module.smult*

typedef (**overloaded**) *'a rf-poly* = { *p* :: *'a* :: *idom poly*. *rsquarefree p*}
⟨*proof*⟩

setup-lifting *type-definition-rf-poly*

context
begin
lifting-forget *poly.lifting*

lift-definition *poly-rf* :: *'a* :: *idom rf-poly* ⇒ *'a poly* **is** λ *x*. *x* ⟨*proof*⟩

definition *roots-of-poly-dummy* :: *'a*::{*comm-ring-1*,*ring-no-zero-divisors*} *poly* ⇒ -
where *roots-of-poly-dummy p* = (*SOME xs*. *set xs* = {*r*. *poly p r* = *0*} ∧ *distinct xs*)

lemma *roots-of-poly-dummy-code*[*code*]:
roots-of-poly-dummy p = *Code.abort* (*STR ''roots−of−poly−dummy''*) (λ *x*. *roots-of-poly-dummy p*)

⟨proof⟩

lemma *roots-of-poly-dummy*: **assumes** *p*: $p \neq 0$
shows *set* (*roots-of-poly-dummy p*) = {*x*. *poly p x* = *0*} *distinct* (*roots-of-poly-dummy p*)
⟨proof⟩

lift-definition *roots-of-complex-rf-poly-part1* :: *complex rf-poly* ⇒ *complex genuine-roots-aux* **is**
λ *p*. *if Ball* (*set* (*Polynomial.coeffs p*)) *algebraic then*
let q = *representative-poly p*;
zeros = *complex-roots-of-int-poly q*
in (*p*,*zeros*,*Polynomial.degree p*, *filter-fun-complex p*)
else (*p*,*roots-of-poly-dummy p*,*Polynomial.degree p*, *filter-fun-complex p*)
⟨proof⟩

lift-definition *roots-of-real-rf-poly-part1* :: *real rf-poly* ⇒ *real genuine-roots-aux* **is**
λ *p*. *let n* = *count-roots p in*
if Ball (*set* (*Polynomial.coeffs p*)) *algebraic then*
let q = *representative-poly p*;
zeros = *real-roots-of-int-poly q*
in (*p*,*zeros*,*n*, *filter-fun-real p*)
else (*p*,*roots-of-poly-dummy p*,*n*, *filter-fun-real p*)
⟨proof⟩

definition *roots-of-complex-rf-poly* :: *complex rf-poly* ⇒ *complex list* **where**
roots-of-complex-rf-poly p = *genuine-roots-impl* (*roots-of-complex-rf-poly-part1 p*)

lemma *roots-of-complex-rf-poly*: *set* (*roots-of-complex-rf-poly p*) = {*x*. *poly* (*poly-rf p*) *x* = *0*}
distinct (*roots-of-complex-rf-poly p*)
⟨proof⟩

definition *roots-of-real-rf-poly* :: *real rf-poly* ⇒ *real list* **where**
roots-of-real-rf-poly p = *genuine-roots-impl* (*roots-of-real-rf-poly-part1 p*)

lemma *roots-of-real-rf-poly*: *set* (*roots-of-real-rf-poly p*) = {*x*. *poly* (*poly-rf p*) *x* = *0*}
distinct (*roots-of-real-rf-poly p*)
⟨proof⟩

typedef (**overloaded**) *'a rf-polys* = { (*a* :: *'a* :: *idom*, *ps* :: (*'a poly* × *nat*) *list*). *Ball* (*fst* ‘ *set ps*) *rsquarefree*}
⟨proof⟩

setup-lifting *type-definition-rf-polys*

lift-definition *yun-polys* :: *'a* :: {*euclidean-ring-gcd,field-char-0,semiring-gcd-mult-normalize*} *poly* ⇒ *'a rf-polys*
is λ *p. yun-factorization gcd p*
⟨*proof*⟩

context
notes [[*typedef-overloaded*]]
begin
lift-definition (*code-dt*) *yun-rf* :: *'a* :: *idom rf-polys* ⇒ *'a* × (*'a rf-poly* × *nat*) *list*
is λ *x. x*
⟨*proof*⟩
end
end
definition *polys-rf* :: *'a* :: *idom rf-polys* ⇒ *'a rf-poly list* **where**
polys-rf = *map fst o snd o yun-rf*

lemma *yun-polys*: **assumes** *p* ≠ *0*
shows *poly p x* = *0* ⟷ (∃ *q* ∈ *set* (*polys-rf* (*yun-polys p*)). *poly* (*poly-rf q*) *x* = *0*)
⟨*proof*⟩

definition *roots-of-complex-rf-polys* :: *complex rf-polys* ⇒ *complex list* **where**
roots-of-complex-rf-polys ps = *concat* (*map roots-of-complex-rf-poly* (*polys-rf ps*))

lemma *roots-of-complex-rf-polys*:
set (*roots-of-complex-rf-polys ps*) = {*x*. ∃ *p* ∈ *set* (*polys-rf ps*). *poly* (*poly-rf p*) *x* = *0* }
⟨*proof*⟩

definition *roots-of-real-rf-polys* :: *real rf-polys* ⇒ *real list* **where**
roots-of-real-rf-polys ps = *concat* (*map roots-of-real-rf-poly* (*polys-rf ps*))

lemma *roots-of-real-rf-polys*:
set (*roots-of-real-rf-polys ps*) = {*x*. ∃ *p* ∈ *set* (*polys-rf ps*). *poly* (*poly-rf p*) *x* = *0* }
⟨*proof*⟩

definition *roots-of-complex-poly* :: *complex poly* ⇒ *complex list* **where**
roots-of-complex-poly p = (*if p* = *0 then* [] *else roots-of-complex-rf-polys* (*yun-polys p*))

lemma *roots-of-complex-poly*: **assumes** *p*: *p* ≠ *0*
shows *set* (*roots-of-complex-poly p*) = {*x*. *poly p x* = *0*}
⟨*proof*⟩

definition *roots-of-real-poly* :: *real poly* ⇒ *real list* **where**
roots-of-real-poly p = (*if p* = *0 then* [] *else roots-of-real-rf-polys* (*yun-polys p*))

lemma *roots-of-real-poly*: **assumes** *p*: $p \neq 0$
shows *set* (*roots-of-real-poly p*) = {*x*. *poly p x* = *0*}
⟨*proof*⟩

lemma *distinct-concat′*:
⟦ *distinct* (*list-neq xs* []);
⋀ *ys*. *ys* ∈ *set xs* ⟹ *distinct ys*;
⋀ *ys zs*. ⟦ *ys* ∈ *set xs* ; *zs* ∈ *set xs* ; *ys* ≠ *zs* ⟧ ⟹ *set ys* ∩ *set zs* = {}
⟧ ⟹ *distinct* (*concat xs*)
⟨*proof*⟩

lemma *roots-of-rf-yun-polys-distinct*: **assumes**
rt: ⋀ *p*. *set* (*rop p*) = {*x*. *poly* (*poly-rf p*) *x* = *0*}
and *dist*: ⋀ *p*. *distinct* (*rop p*)
shows *distinct* (*concat* (*map rop* (*polys-rf* (*yun-polys p*))))
⟨*proof*⟩

lemma *distinct-roots-of-real-poly*: *distinct* (*roots-of-real-poly p*)
⟨*proof*⟩

lemma *distinct-roots-of-complex-poly*: *distinct* (*roots-of-complex-poly p*)
⟨*proof*⟩

end

7 Factorization of Polynomials with Algebraic Coefficients

7.1 Complex Algebraic Coefficients

theory *Factor-Complex-Poly*
imports
Roots-of-Real-Complex-Poly
begin
hide-const (**open**) *MPoly-Type.smult MPoly-Type.degree MPoly-Type.coeff MPoly-Type.coeffs*

definition *factor-complex-main* :: *complex poly* ⇒ *complex* × (*complex* × *nat*) *list*
where
factor-complex-main p ≡ *let* (*c*,*pis*) = *yun-rf* (*yun-polys p*) *in*
(*c*, *concat* (*map* (λ (*p*,*i*). *map* (λ *r*. (*r*,*i*)) (*roots-of-complex-rf-poly p*)) *pis*))

lemma *roots-of-complex-poly-via-factor-complex-main*:
map fst (*snd* (*factor-complex-main p*)) = *roots-of-complex-poly p*
⟨*proof*⟩

lemma *distinct-factor-complex-main*:
distinct (*map fst* (*snd* (*factor-complex-main p*)))

⟨*proof*⟩

lemma *factor-complex-main*: **assumes** *rt*: *factor-complex-main p* = (*c*,*xis*)
shows *p* = *smult c* ($\prod$(*x*, *i*)←*xis*. [:− *x*, *1*:] ^ *i*)
0 ∉ *snd* ‘ *set xis*
⟨*proof*⟩

definition *factor-complex-poly* :: *complex poly* ⇒ *complex* × (*complex poly* × *nat*) *list* **where**
factor-complex-poly p = (*case factor-complex-main p of*
(*c*,*ris*) ⇒ (*c*, *map* (λ (*r*,*i*). ([:−*r*,*1*:],*i*)) *ris*))

lemma *distinct-factor-complex-poly*:
distinct (*map fst* (*snd* (*factor-complex-poly p*)))
⟨*proof*⟩

theorem *factor-complex-poly*: **assumes** *fp*: *factor-complex-poly p* = (*c*,*qis*)
shows
p = *smult c* ($\prod$(*q*, *i*)←*qis*. *q* ^ *i*)
(*q*,*i*) ∈ *set qis* ⟹ *irreducible q* ∧ *i* ≠ *0* ∧ *monic q* ∧ *degree q* = *1*
⟨*proof*⟩

end

7.2 Real Algebraic Coefficients

We basically perform a factorization via complex algebraic numbers, take all real roots, and then merge each pair of conjugate roots into a quadratic factor.

theory *Factor-Real-Poly*
imports
Factor-Complex-Poly
begin

hide-const (**open**) *Coset.order*

fun *delete-cnj* :: *complex* ⇒ *nat* ⇒ (*complex* × *nat*) *list* ⇒ (*complex* × *nat*) *list* **where**
delete-cnj x i ((*y*,*j*) # *yjs*) = (*if x* = *y then if j* = *i then yjs else if j* > *i then*
((*y*,*j* − *i*) # *yjs*) *else delete-cnj x* (*i* − *j*) *yjs else* (*y*,*j*) # *delete-cnj x i yjs*)
| *delete-cnj* - - [] = []

lemma *delete-cnj-length*[*termination-simp*]: *length* (*delete-cnj x i yjs*) ≤ *length yjs*
⟨*proof*⟩

fun *complex-roots-to-real-factorization* :: (*complex* × *nat*) *list* ⇒ (*real poly* × *nat*)*list* **where**
complex-roots-to-real-factorization [] = []

| *complex-roots-to-real-factorization* $((x,i)\ \#\ xs) = (\mathit{if}\ x \in \mathbb{R}\ \mathit{then}$
$([:-(\mathit{Re}\ x),1:],i)\ \#\ \mathit{complex\text{-}roots\text{-}to\text{-}real\text{-}factorization}\ xs\ \mathit{else}$
$\mathit{let}\ xx = \mathit{cnj}\ x;\ ys = \mathit{delete\text{-}cnj}\ xx\ i\ xs;\ p = \mathit{map\text{-}poly}\ \mathit{Re}\ ([:-x,1:] * [:-xx,1:])$
$\mathit{in}\ (p,i)\ \#\ \mathit{complex\text{-}roots\text{-}to\text{-}real\text{-}factorization}\ ys)$

definition *factor-real-poly* :: *real poly* ⇒ *real* × (*real poly* × *nat*) *list* **where**
factor-real-poly p ≡ *case factor-complex-main* (*map-poly of-real p*) *of*
(*c*,*ris*) ⇒ (*Re c*, *complex-roots-to-real-factorization ris*)

lemma *monic-imp-nonzero*: *monic x* ⟹ *x* ≠ *0* **for** *x* :: *'a* :: *semiring-1 poly* ⟨*proof*⟩

lemma *delete-cnj-0*: **assumes** *0* ∉ *snd* ‘ *set xis*
shows *0* ∉ *snd* ‘ *set* (*delete-cnj x si xis*)
⟨*proof*⟩

lemma *delete-cnj*: **assumes**
$\mathit{order}\ x\ (\prod (x,\ i)\leftarrow xis.\ [:-\ x,\ 1:]\ \widehat{}\ i) \geq si\ \ si \neq 0$
shows $(\prod (x,\ i)\leftarrow xis.\ [:-\ x,\ 1:]\ \widehat{}\ i) =$
$[:-\ x,\ 1:]\ \widehat{}\ si * (\prod (x,\ i)\leftarrow \mathit{delete\text{-}cnj}\ x\ si\ xis.\ [:-\ x,\ 1:]\ \widehat{}\ i)$
⟨*proof*⟩

theorem *factor-real-poly*: **assumes** *fp*: *factor-real-poly p* = (*c*,*qis*)
shows $p = \mathit{smult}\ c\ (\prod (q,\ i)\leftarrow qis.\ q\ \widehat{}\ i)$
$(q,j) \in \mathit{set}\ qis \Longrightarrow \mathit{irreducible}\ q \wedge j \neq 0 \wedge \mathit{monic}\ q \wedge \mathit{degree}\ q \in \{1,2\}$
⟨*proof*⟩

end

References

[1] W. S. Brown. The subresultant PRS algorithm. *ACM Trans. Math. Softw.*, 4(3):237–249, 1978.

[2] W. S. Brown and J. F. Traub. On Euclid's algorithm and the theory of subresultants. *Journal of the ACM*, 18(4):505–514, 1971.

[3] S. Joosten, R. Thiemann, and A. Yamada. Subresultants. *Archive of Formal Proofs*, Apr. 2017. https://isa-afp.org/entries/Subresultants.html, Formal proof development.

[4] S. J. C. Joosten, R. Thiemann, and A. Yamada. A verified implementation of algebraic numbers in Isabelle/HOL. *J. Autom. Reason.*, 64(3):363–389, 2020.

[5] A. W. Strzeboński. Computing in the field of complex algebraic numbers. *J. Symbolic Computation*, 24:647–656, 1997.

[6] R. Thiemann, A. Yamada, and S. Joosten. Algebraic numbers in Isabelle/HOL. *Archive of Formal Proofs*, Dec. 2015. https://isa-afp.org/entries/Algebraic_Numbers.html, Formal proof development.