Factorization of Polynomials with Algebraic Coefficients*

Manuel Eberl   René Thiemann

March 17, 2025

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

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

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 −
obtain *r* **where** *r*: *p* ∗ *r* = *1*
using *assms* **by** *auto*
from *r* **have** [*simp*]: *p* ≠ *0 r* ≠ *0*
by *auto*
have *0* = *lead-monom* (*1* :: *'a mpoly*)
by *simp*
also have *1* = *p* ∗ *r*
using *r* **by** *simp*
also have *lead-monom* (*p* ∗ *r*) = *lead-monom p* + *lead-monom r*
by (*intro lead-monom-mult*) *auto*
finally have *lead-monom p* = *0*
by *simp*
hence *vars p* = {}
by (*simp add*: *lead-monom-eq-0-iff*)
hence ∗: *p* = *Const* (*lead-coeff p*)
by (*auto simp*: *vars-empty-iff*)

have *1* = *lead-coeff* (*1* :: *'a mpoly*)
by *simp*
also have *1* = *p* ∗ *r*
using *r* **by** *simp*
also have *lead-coeff* (*p* ∗ *r*) = *lead-coeff p* ∗ *lead-coeff r*
by (*intro lead-coeff-mult*) *auto*
finally have *lead-coeff p dvd 1*
using *dvdI* **by** *blast*
with ∗ **show** *?thesis* **using** *that*
by *blast*
qed

lemma *Const-eq-Const-iff* [*simp*]:
Const c = *Const c'* ⟷ *c* = *c'*
by (*metis lead-coeff-Const*)

lemma *is-unit-ConstI* [*intro*]: *c dvd 1* ⟹ *Const c dvd 1*
by (*metis dvd-def mpoly-Const-1 mpoly-Const-mult*)

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

```
  assume Const c dvd 1
  thus c dvd 1
    by (auto elim!: mpoly-is-unitE)
qed auto

lemma vars-emptyE: vars p = {} ⟹ (⋀c. p = Const c ⟹ P) ⟹ P
  by (auto simp: vars-empty-iff)

lemma degree-geI:
  assumes MPoly-Type.coeff p m ≠ 0
  shows   MPoly-Type.degree p i ≥ Poly-Mapping.lookup m i
proof -
  have lookup m i ≤ Max (insert 0 ((λm. lookup m i) ` keys (mapping-of p)))
  proof (rule Max.coboundedI)
    show lookup m i ∈ insert 0 ((λm. lookup m i) ` keys (mapping-of p))
      using assms by (auto simp: coeff-keys)
  qed auto
  thus ?thesis unfolding MPoly-Type.degree-def by auto
qed

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 (cases MPoly-Type.degree p i = 0)
  case False
  have MPoly-Type.degree p i = Max (insert 0 ((λm. lookup m i) ` keys (mapping-of
p)))
    by (simp add: MPoly-Type.degree-def)
  also have ... ∈ insert 0 ((λm. lookup m i) ` keys (mapping-of p))
    by (rule Max-in) auto
  finally show ?thesis
    using False that by (auto simp: coeff-keys)
next
  case [simp]: True
  from assms obtain m where m: MPoly-Type.coeff p m ≠ 0
    using coeff-all-0 by blast
  show ?thesis using degree-geI[of p m i] m
    by (intro that[of m]) auto
qed

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 (cases p = 0)
  case False
  obtain m where m: MPoly-Type.coeff p m ≠ 0 Poly-Mapping.lookup m i =
MPoly-Type.degree p i
    using monom-of-degree-exists False by blast
```

with *assms* **show** *?thesis*
by *force*
qed *auto*

lemma *coeff-gt-degree-eq-0*:
assumes *Poly-Mapping.lookup m i > MPoly-Type.degree p i*
shows *MPoly-Type.coeff p m = 0*
using *assms degree-geI leD* **by** *blast*

lemma *vars-altdef*: *vars p* = ($\bigcup$ *m*∈{*m. MPoly-Type.coeff p m* ≠ *0*}. *keys m*)
unfolding *vars-def*
by (*intro arg-cong*[**where** *f* = $\bigcup$] *image-cong refl*) (*simp flip*: *coeff-keys*)

lemma *degree-pos-iff*: *MPoly-Type.degree p x* > *0* ⟷ *x* ∈ *vars p*
proof
assume *MPoly-Type.degree p x* > *0*
hence *p* ≠ *0* **by** *auto*
then obtain *m* **where** *m*: *lookup m x = MPoly-Type.degree p x MPoly-Type.coeff p m* ≠ *0*
using *monom-of-degree-exists*[*of p x*] **by** *metis*
from *m* **and** ‹*MPoly-Type.degree p x* > *0*› **have** *x* ∈ *keys m*
by (*simp add*: *in-keys-iff*)
with *m* **show** *x* ∈ *vars p*
by (*auto simp*: *vars-altdef*)
next
assume *x* ∈ *vars p*
then obtain *m* **where** *m*: *x* ∈ *keys m MPoly-Type.coeff p m* ≠ *0*
by (*auto simp*: *vars-altdef*)
have *0* < *lookup m x*
using *m* **by** (*auto simp*: *in-keys-iff*)
also from *m* **have** ... ≤ *MPoly-Type.degree p x*
by (*intro degree-geI*) *auto*
finally show *MPoly-Type.degree p x* > *0* **.**
qed

lemma *degree-eq-0-iff*: *MPoly-Type.degree p x* = *0* ⟷ *x* ∉ *vars p*
using *degree-pos-iff*[*of p x*] **by** *auto*

lemma *MPoly-Type-monom-zero*[*simp*]: *MPoly-Type.monom m 0* = *0*
by (*simp add*: *More-MPoly-Type.coeff-monom coeff-all-0*)

lemma *vars-monom-keys'*: *vars* (*MPoly-Type.monom m c*) = (*if c* = *0 then* {} *else keys m*)
by (*cases c* = *0*) (*auto simp*: *vars-monom-keys*)

lemma *Const-eq-0-iff* [*simp*]: *Const c* = *0* ⟷ *c* = *0*
by (*metis lead-coeff-Const mpoly-Const-0*)

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*
unfolding *MPoly-Type.mult-monom*
by (*rule arg-cong2*[*of - - - - MPoly-Type.monom*], *auto simp*: *remove-key-sum*)

lemma *MPoly-Type-monom-0-iff* [*simp*]: *MPoly-Type.monom m x* = *0* ⟷ *x* = *0*
by (*metis* (*full-types*) *MPoly-Type-monom-zero More-MPoly-Type.coeff-monom when-def*)

lemma *vars-signof* [*simp*]: *vars* (*signof x*) = {}
by (*simp add*: *sign-def*)

lemma *prod-mset-Const*: *prod-mset* (*image-mset Const A*) = *Const* (*prod-mset A*)
by (*induction A*) (*auto simp*: *mpoly-Const-mult*)

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
assume ∗: *Const c* = *a* ∗ *b*
have *lead-monom* (*a* ∗ *b*) = *0*
by (*auto simp flip*: ∗)
hence *lead-monom a* = *0* ∧ *lead-monom b* = *0*
by (*subst* (*asm*) *lead-monom-mult*) (*use assms* ∗ **in** *auto*)
hence *vars a* = {} *vars b* = {}
by (*auto simp*: *lead-monom-eq-0-iff*)
then obtain *a*′ *b*′ **where** *a* = *Const a*′ *b* = *Const b*′
by (*auto simp*: *vars-empty-iff*)
with ∗ **show** (∃ *a*′ *b*′. *a* = *Const a*′ ∧ *b* = *Const b*′ ∧ *c* = *a*′ ∗ *b*′)
by (*auto simp flip*: *mpoly-Const-mult*)
qed (*auto simp*: *mpoly-Const-mult*)

lemma *irreducible-Const-iff* [*simp*]:
irreducible (*Const* (*c* :: ′*a* :: *idom*)) ⟷ *irreducible c*
proof
assume ∗: *irreducible* (*Const c*)
show *irreducible c*
proof (*rule irreducibleI*)
fix *a b* **assume** *c* = *a* ∗ *b*
hence *Const c* = *Const a* ∗ *Const b*
by (*simp add*: *mpoly-Const-mult*)
with ∗ **have** *Const a dvd 1* ∨ *Const b dvd 1*
by *blast*
thus *a dvd 1* ∨ *b dvd 1*
by (*meson is-unit-Const-iff*)
qed (*use* ∗ **in** ‹*auto simp*: *irreducible-def*›)
next

assume ∗: *irreducible c*
have [*simp*]: *c* ≠ *0*
  using ∗ **by** *auto*
show *irreducible* (*Const c*)
proof (*rule irreducibleI*)
  fix *a b* **assume** *Const c* = *a* ∗ *b*
  then obtain *a′ b′* **where** [*simp*]: *a* = *Const a′ b* = *Const b′* **and** *c* = *a′* ∗ *b′*
    by (*auto simp*: *Const-eq-product-iff*)
  hence *a′ dvd 1* ∨ *b′ dvd 1*
    using ∗ **by** *blast*
  thus *a dvd 1* ∨ *b dvd 1*
    by *auto*
qed (*use* ∗ **in** ‹*auto simp*: *irreducible-def is-unit-Const-iff*›)
qed

lemma *Const-dvd-Const-iff* [*simp*]: *Const a dvd Const b* ⟷ *a dvd b*
proof
  assume *a dvd b*
  then obtain *c* **where** *b* = *a* ∗ *c*
    by *auto*
  hence *Const b* = *Const a* ∗ *Const c*
    by (*auto simp*: *mpoly-Const-mult*)
  thus *Const a dvd Const b*
    by *simp*
next
  assume *Const a dvd Const b*
  then obtain *p* **where** *p*: *Const b* = *Const a* ∗ *p*
    by *auto*
  have *MPoly-Type.coeff* (*Const b*) *0* = *MPoly-Type.coeff* (*Const a* ∗ *p*) *0*
    using *p* **by** *simp*
  also have ... = *MPoly-Type.coeff* (*Const a*) *0* ∗ *MPoly-Type.coeff p 0*
    using *mpoly-coeff-times-0* **by** *blast*
  finally show *a dvd b*
    by (*simp add*: *mpoly-coeff-Const*)
qed

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 = (∑ *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*
unfolding *mpoly-to-mpoly-poly-def More-MPoly-Type.coeff-add*[*symmetric*] *MPoly-Type.monom-add add-monom*[*symmetric*]
by (*rule Sum-any.distrib*) *auto*

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 −
have *mpoly-to-mpoly-poly x* (*MPoly-Type.monom m a*) =
($\sum$ *m′*. *Polynomial.monom* (*MPoly-Type.monom* (*remove-key x m′*) *a*) (*lookup m′ x*) *when m′* = *m*)
unfolding *mpoly-to-mpoly-poly-def*
by (*intro Sum-any.cong*, *auto simp*: *when-def More-MPoly-Type.coeff-monom*)
also have ... = *Polynomial.monom* (*MPoly-Type.monom* (*remove-key x m*) *a*) (*lookup m x*)
unfolding *Sum-any-when-equal* **..**
finally show *?thesis* **.**
qed

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*
unfolding *pcr-poly-mapping-def cr-poly-mapping-def OO-def*
by (*auto simp*: *rel-fun-def remove-key-lookup*)

lemma *remove-key-0* [*simp*]: *remove-key x 0* = *0*
by *transfer auto*

lemma *remove-key-single′* [*simp*]:
x ≠ *y* ⟹ *remove-key x* (*Poly-Mapping.single y n*) = *Poly-Mapping.single y n*
by *transfer* (*auto simp*: *when-def fun-eq-iff*)

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 −
have *Sum-any f* = ($\sum$ *x* | *f x* ≠ *0*. *f x*)
by (*rule Sum-any.expand-set*)
also have *poly.coeff* ... *n* = ($\sum$ *x* | *f x* ≠ *0*. *poly.coeff* (*f x*) *n*)
by (*simp add*: *Polynomial.coeff-sum*)
also have ... = *Sum-any* (λ*x*. *poly.coeff* (*f x*) *n*)
by (*rule Sum-any.expand-superset* [*symmetric*]) (*use assms* **in** *auto*)
finally show *?thesis* **.**
qed

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 −
have *MPoly-Type.coeff* (*poly.coeff* (*mpoly-to-mpoly-poly x p*) *n*) *m* =
MPoly-Type.coeff ($\sum$ *a*. *MPoly-Type.monom* (*remove-key x a*) (*MPoly-Type.coeff p a*) *when lookup a x* = *n*) *m*
unfolding *mpoly-to-mpoly-poly-def* **by** (*subst poly-coeff-Sum-any*) (*auto simp*: *when-def*)
also have . . . = ($\sum$ *xa*. *MPoly-Type.coeff* (*MPoly-Type.monom* (*remove-key x xa*) (*MPoly-Type.coeff p xa*)) *m when lookup xa x* = *n*)
by (*subst coeff-Sum-any*, *force*) (*auto simp*: *when-def intro*!: *Sum-any.cong*)
also have . . . = ($\sum$ *a*. *MPoly-Type.coeff p a when lookup a x* = *n* ∧ *m* = *remove-key x a*)
by (*intro Sum-any.cong*) (*simp add*: *More-MPoly-Type.coeff-monom when-def*)
also have (λ*a*. *lookup a x* = *n* ∧ *m* = *remove-key x a*) =
(λ*a*. *lookup m x* = *0* ∧ *a* = *m* + *Poly-Mapping.single x n*)
by (*rule ext*, *transfer*) (*auto simp*: *fun-eq-iff when-def*)
also have ($\sum$ *a*. *MPoly-Type.coeff p a when* . . . *a*) =
($\sum$ *a*. *MPoly-Type.coeff p a when lookup m x* = *0 when a* = *m* + *Poly-Mapping.single x n*)
by (*intro Sum-any.cong*) (*auto simp*: *when-def*)
also have . . . = (*MPoly-Type.coeff p* (*m* + *Poly-Mapping.single x n*) *when lookup m x* = *0*)
by (*rule Sum-any-when-equal*)
finally show *?thesis* **.**
qed

lemma *mpoly-to-mpoly-poly-Const* [*simp*]:
mpoly-to-mpoly-poly x (*Const c*) = [:*Const c*:]
proof −
have *mpoly-to-mpoly-poly x* (*Const c*) =
($\sum$ *m*. *Polynomial.monom* (*MPoly-Type.monom* (*remove-key x m*)
(*MPoly-Type.coeff* (*Const c*) *m*)) (*lookup m x*) *when m* = *0*)
unfolding *mpoly-to-mpoly-poly-def*
by (*intro Sum-any.cong*) (*auto simp*: *when-def mpoly-coeff-Const*)
also have . . . = [:*Const c*:]
by (*subst Sum-any-when-equal*)
(*auto simp*: *mpoly-coeff-Const monom-altdef simp flip*: *Const-conv-monom*)
finally show *?thesis* **.**
qed

lemma *mpoly-to-mpoly-poly-Var*:
mpoly-to-mpoly-poly x (*Var y*) = (*if x* = *y then* [:*0*, *1*:] *else* [:*Var y*:])
proof −
have *mpoly-to-mpoly-poly x* (*Var y*) =
($\sum$ *a*. *Polynomial.monom* (*MPoly-Type.monom* (*remove-key x a*) *1*) (*lookup a x*)
when a = *Poly-Mapping.single y 1*)

unfolding *mpoly-to-mpoly-poly-def* **by** (*intro Sum-any.cong*) (*auto simp*: *when-def coeff-Var*)
also have ... = (*if* $x = y$ *then* [:*0*, *1*:] *else* [:*Var y*:])
by (*auto simp*: *Polynomial.monom-altdef lookup-single Var-altdef*)
finally show *?thesis* **.**
qed

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*:]
by (*simp-all add*: *mpoly-to-mpoly-poly-Var*)

lemma *mpoly-to-mpoly-poly-uminus* [*simp*]:
mpoly-to-mpoly-poly x ($-p$) = $-$*mpoly-to-mpoly-poly x p*
unfolding *mpoly-to-mpoly-poly-def*
by (*auto simp*: *monom-uminus Sum-any-uminus simp flip*: *minus-monom*)

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*
by (*subst diff-conv-add-uminus*, *subst mpoly-to-mpoly-poly-add*) *auto*

lemma *mpoly-to-mpoly-poly-0* [*simp*]:
mpoly-to-mpoly-poly x 0 = *0*
unfolding *mpoly-Const-0* [*symmetric*] *mpoly-to-mpoly-poly-Const* **by** *simp*

lemma *mpoly-to-mpoly-poly-1* [*simp*]:
mpoly-to-mpoly-poly x 1 = *1*
unfolding *mpoly-Const-1* [*symmetric*] *mpoly-to-mpoly-poly-Const* **by** *simp*

lemma *mpoly-to-mpoly-poly-of-nat* [*simp*]:
mpoly-to-mpoly-poly x (*of-nat n*) = *of-nat n*
unfolding *of-nat-mpoly-eq mpoly-to-mpoly-poly-Const of-nat-poly* **..**

lemma *mpoly-to-mpoly-poly-of-int* [*simp*]:
mpoly-to-mpoly-poly x (*of-int n*) = *of-int n*
unfolding *of-nat-mpoly-eq mpoly-to-mpoly-poly-Const of-nat-poly* **by** (*cases n*) *auto*

lemma *mpoly-to-mpoly-poly-numeral* [*simp*]:
mpoly-to-mpoly-poly x (*numeral n*) = *numeral n*
using *mpoly-to-mpoly-poly-of-nat*[*of x numeral n*] **by** (*simp del*: *mpoly-to-mpoly-poly-of-nat*)

lemma *coeff-monom-mult'*:
MPoly-Type.coeff (*MPoly-Type.monom m a* $*$ *q*) m' =
(*a* $*$ *MPoly-Type.coeff q* ($m' - m$) *when lookup* $m' \geq$ *lookup m*)
proof (*cases lookup* $m' \geq$ *lookup m*)
case *True*
have *a* $*$ *MPoly-Type.coeff q* ($m' - m$) = *MPoly-Type.coeff* (*MPoly-Type.monom*

m a ∗ *q*) (*m* + (*m*′ − *m*))
by (*rule More-MPoly-Type.coeff-monom-mult* [*symmetric*])
also have *m* + (*m*′ − *m*) = *m*′
using *True* **by** *transfer* (*auto simp*: *le-fun-def*)
finally show *?thesis*
using *True* **by** (*simp add*: *when-def*)
next
case *False*
have *MPoly-Type.coeff* (*MPoly-Type.monom m a* ∗ *q*) *m*′ =
(∑ *m1*. *a* ∗ (∑ *m2*. *MPoly-Type.coeff q m2 when m*′ = *m1* + *m2*) *when m1* = *m*)
unfolding *coeff-mpoly-times prod-fun-def*
by (*intro Sum-any.cong*) (*auto simp*: *More-MPoly-Type.coeff-monom when-def*)
also have . . . = *a* ∗ (∑ *m2*. *MPoly-Type.coeff q m2 when m*′ = *m* + *m2*)
by (*subst Sum-any-when-equal*) *auto*
also have (λ*m2*. *m*′ = *m* + *m2*) = (λ*m2*. *False*)
by (*rule ext*) (*use False* **in** ‹*transfer*, *auto simp*: *le-fun-def*›)
finally show *?thesis*
using *False* **by** *simp*
qed

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 (*rule poly-eqI*, *rule mpoly-eqI*)
fix *n* :: *nat* **and** *mon* :: *nat* $\Rightarrow_0$ *nat*
have *MPoly-Type.coeff* (*poly.coeff ?lhs n*) *mon* =
(*a* ∗ *MPoly-Type.coeff q* (*mon* + *Poly-Mapping.single x n* − *m*)
when lookup m ≤ *lookup* (*mon* + *Poly-Mapping.single x n*) ∧ *lookup mon x* = *0*)
by (*simp add*: *coeff-coeff-mpoly-to-mpoly-poly coeff-monom-mult*′ *when-def*)
have *MPoly-Type.coeff* (*poly.coeff ?rhs n*) *mon* =
(*a* ∗ *MPoly-Type.coeff q* (*mon* − *remove-key x m* + *Poly-Mapping.single x* (*n* − *lookup m x*))
when lookup (*remove-key x m*) ≤ *lookup mon* ∧ *lookup m x* ≤ *n* ∧ *lookup mon x* = *0*)
by (*simp add*: *coeff-coeff-mpoly-to-mpoly-poly coeff-monom-mult*′ *lookup-minus-fun remove-key-lookup Missing-Polynomial.coeff-monom-mult when-def*)
also have *lookup* (*remove-key x m*) ≤ *lookup mon* ∧ *lookup m x* ≤ *n* ∧ *lookup mon x* = *0* ⟷
lookup m ≤ *lookup* (*mon* + *Poly-Mapping.single x n*) ∧ *lookup mon x* = *0* (**is** - = *?P*)
by *transfer* (*auto simp*: *when-def le-fun-def*)
also have *mon* − *remove-key x m* + *Poly-Mapping.single x* (*n* − *lookup m x*) = *mon* + *Poly-Mapping.single x n* − *m* **if** *?P*
using *that* **by** *transfer* (*auto simp*: *fun-eq-iff when-def*)
hence (*a* ∗ *MPoly-Type.coeff q* (*mon* − *remove-key x m* + *Poly-Mapping.single*

x (*n* − *lookup m x*)) *when ?P*) =
(*a* ∗ *MPoly-Type.coeff q* . . . *when ?P*)
by (*intro when-cong*) *auto*
also have . . . = *MPoly-Type.coeff* (*poly.coeff ?lhs n*) *mon*
by (*simp add*: *coeff-coeff-mpoly-to-mpoly-poly coeff-monom-mult′ when-def*)
finally show *MPoly-Type.coeff* (*poly.coeff ?lhs n*) *mon* = *MPoly-Type.coeff* (*poly.coeff ?rhs n*) *mon* **..**
qed

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*
by (*induction p arbitrary*: *q rule*: *mpoly-induct*)
(*simp-all add*: *mpoly-to-mpoly-poly-monom mpoly-to-mpoly-poly-mult-monom ring-distribs*)

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*)
unfolding *mpoly-to-mpoly-poly-def* **by** (*subst poly-coeff-Sum-any*) (*auto simp*: *when-def*)

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 −
have *MPoly-Type.coeff* (*poly.coeff* (*mpoly-to-mpoly-poly x p*) (*lookup m x*)) (*remove-key x m*) =
($\sum$*xa*. *MPoly-Type.coeff* (*MPoly-Type.monom* (*remove-key x xa*) (*MPoly-Type.coeff p xa*) *when*
lookup xa x = *lookup m x*) (*remove-key x m*))
by (*subst coeff-mpoly-to-mpoly-poly*, *subst coeff-Sum-any*) *auto*
also have . . . = ($\sum$*xa*. *MPoly-Type.coeff* (*MPoly-Type.monom* (*remove-key x xa*) (*MPoly-Type.coeff p xa*)) (*remove-key x m*)
when lookup xa x = *lookup m x*)
by (*intro Sum-any.cong*) (*auto simp*: *when-def*)
also have . . . = ($\sum$*xa*. *MPoly-Type.coeff p xa when remove-key x m* = *remove-key x xa* ∧ *lookup xa x* = *lookup m x*)
by (*intro Sum-any.cong*) (*auto simp*: *More-MPoly-Type.coeff-monom when-def*)
also have (λ*xa*. *remove-key x m* = *remove-key x xa* ∧ *lookup xa x* = *lookup m x*) = (λ*xa*. *xa* = *m*)
using *remove-key-sum* **by** *metis*
also have ($\sum$*xa*. *MPoly-Type.coeff p xa when xa* = *m*) = *MPoly-Type.coeff p m*
by *simp*
finally show *?thesis* **..**
qed

lemma *degree-mpoly-to-mpoly-poly* [*simp*]:

Polynomial.degree (*mpoly-to-mpoly-poly x p*) = *MPoly-Type.degree p x*
proof (*rule antisym*)
show *Polynomial.degree* (*mpoly-to-mpoly-poly x p*) ≤ *MPoly-Type.degree p x*
proof (*intro Polynomial.degree-le allI impI*)
fix *i* **assume** *i*: *i* > *MPoly-Type.degree p x*
have *poly.coeff* (*mpoly-to-mpoly-poly x p*) *i* =
($\sum$ *m*. *0 when lookup m x* = *i*)
unfolding *coeff-mpoly-to-mpoly-poly* **using** *i*
by (*intro Sum-any.cong when-cong refl*) (*auto simp*: *coeff-gt-degree-eq-0*)
also have ... = *0*
by *simp*
finally show *poly.coeff* (*mpoly-to-mpoly-poly x p*) *i* = *0* **.**
qed
next
show *Polynomial.degree* (*mpoly-to-mpoly-poly x p*) ≥ *MPoly-Type.degree p x*
proof (*cases p* = *0*)
case *False*
then obtain *m* **where** *m*: *MPoly-Type.coeff p m* ≠ *0 lookup m x* = *MPoly-Type.degree p x*
using *monom-of-degree-exists* **by** *blast*
show *Polynomial.degree* (*mpoly-to-mpoly-poly x p*) ≥ *MPoly-Type.degree p x*
proof (*rule Polynomial.le-degree*)
have *0* ≠ *MPoly-Type.coeff p m*
using *m* **by** *auto*
also have *MPoly-Type.coeff p m* = *MPoly-Type.coeff* (*poly.coeff* (*mpoly-to-mpoly-poly x p*) (*lookup m x*)) (*remove-key x m*)
by (*rule mpoly-coeff-to-mpoly-poly-coeff*)
finally show *poly.coeff* (*mpoly-to-mpoly-poly x p*) (*MPoly-Type.degree p x*) ≠ *0*
using *m* **by** *auto*
qed
qed *auto*
qed

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 (*induct p rule*: *mpoly-induct*)
case (*monom m a*)
show *?case* **unfolding** *mpoly-to-mpoly-poly-monom poly-monom*
by (*transfer*, *simp add*: Var_0*-power mult-single remove-key-sum*)
next
case (*sum p1 p2 m a*)
then show *?case* **by** (*simp add*: *mpoly-to-mpoly-poly-add*)
qed

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

proof
  assume *mpoly-to-mpoly-poly x p* = *mpoly-to-mpoly-poly x q*
  hence *poly* (*mpoly-to-mpoly-poly x p*) (*Var x*) =
        poly (*mpoly-to-mpoly-poly x q*) (*Var x*)
    by *simp*
  thus *p* = *q*
    by (*auto simp*: *poly-mpoly-to-mpoly-poly*)
qed *auto*

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

lemma *insertion-mpoly-to-mpoly-poly*: **assumes** $\bigwedge$ *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 (*induct p rule*: *mpoly-induct*)
  case (*monom m a*)
  let *?rkm* = *remove-key x m*
  have *to-alpha*: *insertion* β (*MPoly-Type.monom ?rkm a*) = *insertion* α (*MPoly-Type.monom*
?rkm a)
    by (*rule insertion-irrelevant-vars*, *rule assms*, *insert vars-monom-subset*[*of ?rkm*
a], *auto simp*: *remove-key-keys*[*symmetric*])
   have *main*: *insertion* α (*MPoly-Type.monom ?rkm a*) ∗ α *x* ^ *lookup m x* =
insertion α (*MPoly-Type.monom m a*)
    unfolding *monom-remove-key*[*of m a x*] *insertion-mult*
    by (*metis insertion-single mult.left-neutral*)
  show *?case* **using** *main to-alpha*
    by (*simp add*: *mpoly-to-mpoly-poly-monom map-poly-monom poly-monom*)
next
  case (*sum p1 p2 m a*)
  then show *?case* **by** (*simp add*: *mpoly-to-mpoly-poly-add insertion-add map-poly-add*)

qed

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
  assume *mpoly-to-mpoly-poly x p dvd mpoly-to-mpoly-poly x q*
  hence *poly* (*mpoly-to-mpoly-poly x p*) (*Var x*) *dvd poly* (*mpoly-to-mpoly-poly x q*)
(*Var x*)
    by (*intro poly-hom.hom-dvd*)
  thus *p dvd q*
    by (*simp add*: *poly-mpoly-to-mpoly-poly*)
qed *auto*

lemma *vars-coeff-mpoly-to-mpoly-poly*: *vars* (*poly.coeff* (*mpoly-to-mpoly-poly x p*)
i) ⊆ *vars p* − {*x*}
  unfolding *mpoly-to-mpoly-poly-def Sum-any.expand-set Polynomial.coeff-sum More-MPoly-Type.coeff-monon*
  apply (*rule order.trans*[*OF vars-setsum*], *force*)
  apply (*rule UN-least*, *simp*)
  apply (*intro impI order.trans*[*OF vars-monom-subset*])

by (*simp add*: *remove-key-keys*[*symmetric*] *Diff-mono SUP-upper2 coeff-keys vars-def*)

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*)
        (*λm a*. *Polynomial.monom* (*MPoly-Type.monom* (*remove-key x m*) *a*) (*lookup m x*))
      MPoly-Type.monom
  and *transfer-coeff* [*transfer-rule*]:
      (*R* ===> (=) ===> (=))
        (*λp 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* (*λp*. *MPoly-Type.degree p x*)
  unfolding *R-def*
  by (*auto simp*: *rel-fun-def mpoly-to-mpoly-poly-monom*
        simp flip: *mpoly-coeff-to-mpoly-poly-coeff*)

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 (*intro equalityI*)
  have *vars p'* = *vars* (*poly p* (*Var x*))
    using *assms* **by** (*simp add*: *R-def poly-mpoly-to-mpoly-poly*)

also have *poly p* (*Var x*) = ($\sum$ *i*≤*Polynomial.degree p. poly.coeff p i* ∗ *Var x* ^ *i*)
unfolding *poly-altdef* **..**
also have *vars* ... ⊆ ($\bigcup$ *i. vars* (*poly.coeff p i*) ∪ (*if Polynomial.degree p* = *0 then* {} *else* {*x*}))
proof (*intro order.trans*[*OF vars-sum*] *UN-mono order.trans*[*OF vars-mult*] *Un-mono*)
fix *i* :: *nat*
assume *i*: *i* ∈ {..*Polynomial.degree p*}
show *vars* (*Var x* ^ *i*) ⊆ (*if Polynomial.degree p* = *0 then* {} *else* {*x*})
proof (*cases Polynomial.degree p* = *0*)
case *False*
thus *?thesis*
by (*intro order.trans*[*OF vars-power*]) (*auto simp*: *vars-Var*)
qed (*use i* **in** *auto*)
qed *auto*
finally show *vars p′* ⊆ *?A* ∪ *?B* **by** *blast*
next
have *?A* ⊆ *vars p′*
using *assms vars-coeff-mpoly-to-mpoly-poly* **by** (*auto simp*: *R-def*)
moreover have *?B* ⊆ *vars p′*
using *assms* **by** (*auto simp*: *R-def degree-pos-iff*)
ultimately show *?A* ∪ *?B* ⊆ *vars p′*
by *blast*
qed

lemma *right-total* [*transfer-rule*]: *right-total R*
unfolding *right-total-def*
proof *safe*
fix *p′* :: *′a mpoly*
show ∃ *p. R p p′*
by (*rule exI*[*of - mpoly-to-mpoly-poly x p′*]) (*auto simp*: *R-def*)
qed

lemma *bi-unique* [*transfer-rule*]: *bi-unique R*
unfolding *bi-unique-def* **by** (*auto simp*: *R-def*)

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 −
interpret *transfer-mpoly-to-mpoly-poly x* **.**
define *deg* :: *′a mpoly* ⇒ *nat* **where** *deg* = (*λp. MPoly-Type.degree p x*)

have [*transfer-rule*]: *rel-fun R* (=) *Polynomial.degree deg*
using *transfer-degree* **unfolding** *deg-def* **.**

have *deg* (*p* ∗ *q*) = *deg p* + *deg q*
using *assms* **unfolding** *deg-def* [*symmetric*]
by *transfer* (*simp add*: *degree-mult-eq*)
thus *?thesis*
by (*simp add*: *deg-def*)
qed

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)
by (*unfold-locales*, *auto simp*: *insertion-add insertion-mult*)

lemma *partial-insertion-add*: *partial-insertion* α *x* (*p* + *q*) = *partial-insertion* α *x p* + *partial-insertion* α *x q*
proof −
interpret *i*: *comm-ring-hom insertion* α **by** (*rule comm-ring-hom-insertion*)
show *?thesis* **unfolding** *partial-insertion-def mpoly-to-mpoly-poly-add hom-distribs*
..
qed

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*)
unfolding *partial-insertion-def mpoly-to-mpoly-poly-monom*
by (*subst map-poly-monom*, *auto*)

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

lemma *insertion-partial-insertion*: **assumes** ⋀ *y*. *y* ≠ *x* ⟹ β *y* = α *y*
shows *poly* (*partial-insertion* β *x p*) (α *x*) = *insertion* α *p*
proof (*induct p rule*: *mpoly-induct*)
case (*monom m a*)
let *?rkm* = *remove-key x m*
have *to-alpha*: *insertion* β (*MPoly-Type.monom ?rkm a*) = *insertion* α (*MPoly-Type.monom ?rkm a*)
by (*rule insertion-irrelevant-vars*, *rule assms*, *insert vars-monom-subset*[*of ?rkm a*], *auto simp*: *remove-key-keys*[*symmetric*])
have *main*: *insertion* α (*MPoly-Type.monom ?rkm a*) ∗ α *x* ^ *lookup m x* = *insertion* α (*MPoly-Type.monom m a*)
unfolding *monom-remove-key*[*of m a x*] *insertion-mult*
by (*metis insertion-single mult.left-neutral*)
show *?case* **using** *main to-alpha* **by** (*simp add*: *partial-insertion-monom poly-monom*)

next
  case (*sum p1 p2 m a*)
  then show *?case* **by** (*simp add*: *partial-insertion-add insertion-add map-poly-add*)

qed

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*
  unfolding *partial-insertion-def*
  by (*subst coeff-map-poly*, *auto*)

lemma *degree-map-poly-Const*: *degree* (*map-poly* (*Const* :: *'a* :: *semiring-0* ⇒ -) *f*) = *degree f*
  by (*rule degree-map-poly*, *auto*)

lemma *degree-partial-insertion-le-mpoly*: *degree* (*partial-insertion* α *x p*) ≤ *degree* (*mpoly-to-mpoly-poly x p*)
  unfolding *partial-insertion-def* **by** (*rule degree-map-poly-le*)

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*
  using *assms* **by** (*elim dvdE*) (*auto simp*: *Polynomial.lead-coeff-mult*)

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* ≠ *0* ∧ *y dvd x then THE z*. *x* = *y* ∗ *z else 0*)

instance ..

end

instance *mpoly* :: (*idom*) *idom-divide*
  by *standard* (*auto simp*: *divide-mpoly-def*)

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*)
  using *assms*
  by (*smt* (*z3*) *div-by-0 dvd-imp-mult-div-cancel-left mpoly-to-mpoly-poly-0 mpoly-to-mpoly-poly-eq-iff mpoly-to-mpoly-poly-mult nonzero-mult-div-cancel-left transfer-mpoly-to-mpoly-poly.R-def*)

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*)
  unfolding *unit-factor-mpoly-def* **by** *simp*

lemma *normalize-mpoly-Const* [*simp*]:
  normalize (*Const c*) = *Const* (*normalize c*)
proof (*cases c* = *0*)
  case *False*
  have *normalize* (*Const c*) = *Const c div Const* (*unit-factor c*)
    by (*simp add*: *normalize-mpoly-def*)
  also have ... = *Const* (*unit-factor c* ∗ *normalize c*) *div Const* (*unit-factor c*)
    by *simp*
  also have ... = *Const* (*unit-factor c*) ∗ *Const* (*normalize c*) *div Const* (*unit-factor c*)
    by (*subst mpoly-Const-mult*) *auto*
  also have ... = *Const* (*normalize c*)
    using ‹*c* ≠ *0*›
    by (*subst nonzero-mult-div-cancel-left*) *auto*
  finally show *?thesis* **.**
qed (*auto simp*: *normalize-mpoly-def*)

instance proof
  show *unit-factor* (*0* :: *′a mpoly*) = *0*
    by (*simp add*: *unit-factor-mpoly-def*)
next
  show *unit-factor x* = *x* **if** *x dvd 1* **for** *x* :: *′a mpoly*

```
    using that by (auto elim!: mpoly-is-unitE simp: is-unit-unit-factor)
next
  fix x :: 'a mpoly
  assume x ≠ 0
  thus unit-factor x dvd 1
    by (auto simp: unit-factor-mpoly-def)
next
  fix x y :: 'a mpoly
  assume x dvd 1
  hence x ≠ 0
    by auto
  show unit-factor (x * y) = x * unit-factor y
  proof (cases y = 0)
    case False
    have Const (unit-factor (lead-coeff x * lead-coeff y)) =
            x * Const (unit-factor (lead-coeff y)) using ‹x dvd 1›
      by (subst unit-factor-mult-unit-left)
         (auto elim!: mpoly-is-unitE simp: mpoly-Const-mult)
    thus ?thesis using ‹x ≠ 0› False
      by (simp add: unit-factor-mpoly-def lead-coeff-mult)
  qed (auto simp: unit-factor-mpoly-def)
next
  fix p :: 'a mpoly
  let ?c = Const (unit-factor (lead-coeff p))
  show unit-factor p * normalize p = p
  proof (cases p = 0)
    case False
    hence ?c dvd 1
      by (intro is-unit-ConstI) auto
    also have 1 dvd p
      by simp
    finally have ?c * (p div ?c) = p
      by (rule dvd-imp-mult-div-cancel-left)
    thus ?thesis
      by (auto simp: unit-factor-mpoly-def normalize-mpoly-def)
  qed (auto simp: normalize-mpoly-def)
next
  show normalize (0 :: 'a mpoly) = 0
    by (simp add: normalize-mpoly-def)
qed

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 −
  define *x* **where** *x* = *Max* (*vars p* ∪ *vars q*)
  define *p'* **where** *p'* = *mpoly-to-mpoly-poly x p*
  define *q'* **where** *q'* = *mpoly-to-mpoly-poly x q*
  interpret *transfer-mpoly-to-mpoly-poly x* **.**
  have [*transfer-rule*]: *R p' p R q' q*
    by (*auto simp*: *p'-def q'-def R-def*)

  show *?thesis*
  proof (*cases vars p* ∪ *vars q* = {})
    case *True*
    define *a* **where** *a* = *MPoly-Type.coeff p 0*
    define *b* **where** *b* = *MPoly-Type.coeff q 0*
    have [*simp*]: *p* = *Const a q* = *Const b*
      using *True* **by** (*auto elim*!: *vars-emptyE simp*: *a-def b-def mpoly-coeff-Const*)
    show *?thesis*
      apply (*cases b* = *0*)
      apply (*auto simp*: *Let-def mpoly-coeff-Const mpoly-Const-mult divide-option-def elim*!: *dvdE*)
      by (*metis dvd-triv-left*)
  next
    case *False*
    have *?rhs* =
          (*case p' div? q' of None* ⇒ *None*
            | *Some r* ⇒ *Some* (*poly r* (*Var x*)))
      using *False*
      unfolding *Let-def*
      apply (*simp only*:)
      apply (*subst if-False*)
      apply (*simp flip*: *x-def p'-def q'-def cong*: *option.case-cong*)
      done

also have ... = (*if* q' *dvd* p' *then Some* (*poly* (p' *div* q') (*Var x*)) *else None*)
  using *False* **by** (*auto simp*: *divide-option-def*)
also have ... = *p div? q*
  unfolding *divide-option-def*
proof (*intro if-cong refl arg-cong*[**where** *f* = *Some*])
  show (q' *dvd* p') = (*q dvd p*)
    by *transfer-prover*
next
  assume [*transfer-rule*]: *q dvd p*
  have *R* (p' *div* q') (*p div q*)
    by *transfer-prover*
  thus *poly* (p' *div* q') (*Var x*) = *p div q*
    by (*simp add*: *R-def poly-mpoly-to-mpoly-poly*)
qed
finally show *?thesis* **..**
qed
qed

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 (*cases p* = *0*; *cases q* = *0*)
  assume [*simp*]: *p* ≠ *0 q* ≠ *0*
  define *dp* **where** *dp* = *Polynomial.degree p*
  define *dq* **where** *dq* = *Polynomial.degree q*
  define *cp* **where** *cp* = *Polynomial.lead-coeff p*
  define *cq* **where** *cq* = *Polynomial.lead-coeff q*
  define *mon* **where** *mon* = *Polynomial.monom* (*cp div cq*) (*dp* − *dq*)
  show *?thesis*
  proof (*cases dp* < *dq*)
    case *True*
    hence ¬*q dvd p*
      unfolding *dp-def dq-def*
      by (*meson* ‹*p* ≠ *0*› *divides-degree leD*)
    thus *?thesis*

using *True* **by** (*simp add*: *divide-option-def dp-def dq-def*)
next
case *deg*: *False*
show *?thesis*
proof (*cases cq dvd cp*)
case *False*
hence ¬*q dvd p*
unfolding *cq-def cp-def* **using** *poly-lead-coeff-dvd-lead-coeff* **by** *blast*
thus *?thesis*
using *deg False* **by** (*simp add*: *dp-def dq-def Let-def divide-option-def cp-def cq-def*)
next
case *dvd1*: *True*
show *?thesis*
proof (*cases q dvd* (*p* − *mon* ∗ *q*))
case *False*
hence ¬*q dvd p*
by (*meson dvd-diff dvd-triv-right*)
thus *?thesis*
using *deg dvd1 False*
by (*simp add*: *dp-def dq-def Let-def divide-option-def cp-def cq-def mon-def*)
next
case *dvd2*: *True*
hence *q dvd p*
by (*metis diff-eq-eq dvd-add dvd-triv-right*)
have *?rhs* = *Some* (*mon* + (*p* − *mon* ∗ *q*) *div q*)
using *deg dvd1 dvd2*
by (*simp add*: *dp-def dq-def Let-def divide-option-def cp-def cq-def mon-def*)
also have *mon* + (*p* − *mon* ∗ *q*) *div q* = *p div q*
using *dvd2* **by** (*elim dvdE*) (*auto simp*: *algebra-simps*)
also have *Some* . . . = *p div? q*
using ‹*q dvd p*› **by** (*simp add*: *divide-option-def*)
finally show *?thesis* **..**
qed
qed
qed
qed (*auto simp*: *divide-option-def*)

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*)
by (*auto simp*: *divide-option-mpoly-def divide-option-def divide-mpoly-def*)

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*
by (*simp add*: *MPoly-inverse*)

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
by (*erule finite-subset*[*rotated*]) *auto*

lemma *lookup-filter-pm*: *lookup* (*filter-pm P f*) *x* = (*if P x then lookup f x else 0*)
by *transfer auto*

lemma *filter-pm-code* [*code*]: *filter-pm P* (*Pm-fmap m*) = *Pm-fmap* (*fmfilter P m*)
by (*auto intro*!: *poly-mapping-eqI simp*: *fmlookup-default-def lookup-filter-pm*)

lemma *remove-key-conv-filter-pm* [*code*]: *remove-key x m* = *filter-pm* (*λy. y* ≠ *x*) *m*
by *transfer auto*

lemma *finite-poly-coeff-nonzero*: *finite* {*n. poly.coeff p n* ≠ *0*}
by (*metis MOST-coeff-eq-0 eventually-cofinite*)

lemma *poly-degree-conv-Max*:
assumes *p* ≠ *0*
shows *Polynomial.degree p* = *Max* {*n. poly.coeff p n* ≠ *0*}
using *assms*

proof (*intro antisym degree-le Max.boundedI*)
  fix *n* **assume** $n \in \{n.\ poly.coeff\ p\ n \neq 0\}$
  thus $n \leq Polynomial.degree\ p$
    by (*simp add*: *le-degree*)
qed (*auto simp*: *poly-eq-iff finite-poly-coeff-nonzero*)

lemma *mpoly-to-poly-code-aux*:
  fixes *p* :: *'a* :: *comm-monoid-add mpoly* **and** *x* :: *nat*
  defines $I \equiv (\lambda m.\ lookup\ m\ x)$ ' *Set.filter* $(\lambda m.\ \forall y \in keys\ m.\ y = x)$ (*keys* (*mapping-of p*))
  shows $I = \{n.\ poly.coeff\ (mpoly\text{-}to\text{-}poly\ x\ p)\ n \neq 0\}$
    and $mpoly\text{-}to\text{-}poly\ x\ p = 0 \longleftrightarrow I = \{\}$
    and $I \neq \{\} \Longrightarrow Polynomial.degree\ (mpoly\text{-}to\text{-}poly\ x\ p) = Max\ I$
proof −
  have $n \in I \longleftrightarrow poly.coeff\ (mpoly\text{-}to\text{-}poly\ x\ p)\ n \neq 0$ **for** *n*
  proof −
    have $I = (\lambda m.\ lookup\ m\ x)$ ' $(keys\ (mapping\text{-}of\ p) \cap \{m.\ \forall y \in keys\ m.\ y = x\})$
      by (*auto simp*: *I-def Set.filter-def*)
    also have $\{m.\ \forall y \in keys\ m.\ y = x\} = range\ (\lambda n.\ monomial\ n\ x)$ (**is** *?lhs* = *?rhs*)
    proof (*intro equalityI subsetI*)
      fix *m* **assume** $m \in ?lhs$
      hence $m = monomial\ (lookup\ m\ x)\ x$
        by *transfer* (*auto simp*: *fun-eq-iff when-def*)
      thus $m \in ?rhs$ **by** *auto*
    qed (*auto split*: *if-splits*)
    also have $n \in (\lambda m.\ lookup\ m\ x)$ ' $(keys\ (mapping\text{-}of\ p) \cap \ldots) \longleftrightarrow$
              $monomial\ n\ x \in keys\ (mapping\text{-}of\ p)$ **by** *force*
    also have $\ldots \longleftrightarrow poly.coeff\ (mpoly\text{-}to\text{-}poly\ x\ p)\ n \neq 0$
      by (*simp add*: *coeff-def in-keys-iff*)
    finally show *?thesis* **.**
  qed
  thus *I*: $I = \{n.\ poly.coeff\ (mpoly\text{-}to\text{-}poly\ x\ p)\ n \neq 0\}$
    by *blast*
  show *eq-0-iff*: $mpoly\text{-}to\text{-}poly\ x\ p = 0 \longleftrightarrow I = \{\}$
    unfolding *I* **by** (*auto simp*: *poly-eq-iff*)
  show $I \neq \{\} \Longrightarrow Polynomial.degree\ (mpoly\text{-}to\text{-}poly\ x\ p) = Max\ I$
    by (*subst poly-degree-conv-Max*) (*use eq-0-iff I* **in** *auto*)
qed

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

```
  define I where I = (λm. lookup m x) ` Set.filter (λm. ∀ y∈keys m. y = x) (keys
(mapping-of p))
  show ?thesis
  proof (cases I = {})
    case True
    thus ?thesis using mpoly-to-poly-code-aux(2)[of x p]
      by (simp add: I-def)
  next
    case False
    have [simp]: mpoly-to-poly x p ≠ 0
      using mpoly-to-poly-code-aux(2)[of x p] False by (simp add: I-def)
    from False have ?rhs = map (λn. MPoly-Type.coeff p (Poly-Mapping.single x
n)) [0..<Max I + 1]
      (is - = ?rhs')
      by (simp add: I-def Let-def)
    also have ... = ?lhs
    proof (rule nth-equalityI)
      show length ?rhs' = length ?lhs
        using mpoly-to-poly-code-aux(3)[of x p] False
        by (simp add: I-def length-coeffs-degree)
      thus ?rhs' ! n = ?lhs ! n if n < length ?rhs' for n using that
        by (auto simp del: upt-Suc simp: nth-coeffs-coeff)
    qed
    finally show ?thesis ..
  qed
qed


fun mpoly-to-mpoly-poly-impl-aux1 :: nat ⇒ ((nat ⇒₀ nat) × 'a) list ⇒ nat ⇒
((nat ⇒₀ nat) × '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)
  by (induction xs) auto

lemma map-of-mpoly-to-mpoly-poly-impl-aux1:
  map-of (mpoly-to-mpoly-poly-impl-aux1 i xs j) = (λmon.
     (if lookup mon i > 0 then None
      else map-of xs (mon + Poly-Mapping.single i j)))
  apply (rule ext)
  apply (induction i xs j rule: mpoly-to-mpoly-poly-impl-aux1.induct)
   apply (auto simp: remove-key-lookup)
    apply (meson remove-key-sum)
```

apply (*metis add-left-cancel lookup-single-eq remove-key-sum*)
apply (*metis remove-key-add remove-key-single remove-key-sum single-zero*)
done

lemma *lookup0-fmap-of-list-mpoly-to-mpoly-poly-impl-aux1*:
lookup0 (*fmap-of-list* (*mpoly-to-mpoly-poly-impl-aux1 i xs j*)) = (λ*mon.*
lookup0 (*fmap-of-list xs*) (*mon* + *Poly-Mapping.single i j*) *when lookup mon i* = *0*)
by (*auto simp add*: *fmlookup-default-def fmlookup-of-list map-of-mpoly-to-mpoly-poly-impl-aux1*)

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*
by (*simp add*: *coeff-def mpoly.MPoly-inverse*)

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*)))
unfolding *mpoly-to-mpoly-poly-impl-aux2-def*
by (*rule mpoly-eqI*)
(*simp add*: *coeff-coeff-mpoly-to-mpoly-poly coeff-MPoly*
lookup0-fmap-of-list-mpoly-to-mpoly-poly-impl-aux1)

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 −
interpret *transfer-mpoly-to-mpoly-poly x* **.**
define *p'* **where** *p'* = *mpoly-to-mpoly-poly x p*
have [*transfer-rule*]: *R p' p*
by (*auto simp*: *R-def p'-def*)
show *?thesis*
unfolding *p'-def* [*symmetric*] **by** *transfer-prover*
qed

lemma *mpoly-to-mpoly-poly-code* [*code*]:
Polynomial.coeffs (*mpoly-to-mpoly-poly x p*) = *mpoly-to-mpoly-poly-impl x p*
by (*intro nth-equalityI*)
(*auto simp*: *mpoly-to-mpoly-poly-impl-def length-coeffs-degree*
mpoly-to-mpoly-poly-impl-aux2-def coeffs-nth simp del: *upt-Suc*)

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 (*induct n*)
  case (*Suc n*)
  let *?A* = *sylvester-mat-sub* (*degree f* + *Suc n*) (*degree g*) *f g*
  let *?d* = *degree f* + *Suc n* + *degree g*

define *h* **where** *h i* = *?A* $$ (*i*,*0*) ∗ *cofactor ?A i 0* **for** *i*
have *mult-left-zero*: *x* = *0* ⟹ *x* ∗ *y* = *0* **for** *x y* :: *'a* **by** *auto*
have *det ?A* = (∑ *i*<*?d*. *h i*)
unfolding *h-def*
by (*rule laplace-expansion-column*[*OF sylvester-mat-sub-carrier*, *of 0*], *force*)
also have ... = *sum h* ({*degree g*} ∪ ({..<*?d*} − {*degree g*}))
by (*rule sum.cong*, *auto*)
also have ... = *sum h* {*degree g*} + *sum h* ({..<*?d*} − {*degree g*})
by (*rule sum.union-disjoint*, *auto*)
also have *sum h* ({..<*?d*} − {*degree g*}) = *0*
unfolding *h-def*
by (*intro sum.neutral ballI mult-left-zero*, *auto simp*: *sylvester-mat-sub-def coeff-eq-0*)
also have *sum h* {*degree g*} = *h* (*degree g*) **by** *simp*
also have ... = *lead-coeff g* ∗ *cofactor ?A* (*degree g*) *0* **unfolding** *h-def*
by (*rule arg-cong*[*of - - λ x. x* ∗ -], *simp add*: *sylvester-mat-sub-def*)
also have *cofactor ?A* (*degree g*) *0* = (−*1*)^(*degree g*) ∗ *det* (*sylvester-mat-sub* (*degree f* + *n*) (*degree g*) *f g*)
unfolding *cofactor-def*
proof (*intro arg-cong2*[*of - - - - λ x y*. (−*1*)^*x* ∗ *det y*], *force*)
show *mat-delete ?A* (*degree g*) *0* = *sylvester-mat-sub* (*degree f* + *n*) (*degree g*) *f g*
unfolding *sylvester-mat-sub-def*
by (*intro eq-matI*, *auto simp*: *mat-delete-def coeff-eq-0*)
qed
finally show *?case* **unfolding** *Suc* **by** *simp*
qed *simp*

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 −
let *?pa* = *partial-insertion* α *x*
let *?a* = *insertion* α
let *?q* = *map-poly Const q*

let *?m* = *mpoly-to-mpoly-poly x*
interpret *a*: *comm-ring-hom ?a* **by** (*rule comm-ring-hom-insertion*)
define *m* **where** *m* = *degree* (*?m p*) − *degree* (*?pa p*)
from *degree-partial-insertion-le-mpoly*[*of α x p*] **have** *deg*: *degree* (*?m p*) = *degree* (*?pa p*) + *m* **unfolding** *m-def* **by** *simp*
define *k* **where** *k* = *degree* (*?pa p*) + *m*
define *l* **where** *l* = *degree q*
have *resultant* (*?pa p*) *q* = *det* (*sylvester-mat-sub* (*degree* (*?pa p*)) (*degree q*) (*?pa p*) *q*)
unfolding *resultant-def sylvester-mat-def* **by** *simp*
have *?a* (*resultant-mpoly-poly x p q*) = *?a* (*det* (*sylvester-mat-sub* (*degree* (*?pa p*) + *m*) (*degree q*) (*?m p*) *?q*))
unfolding *resultant-mpoly-poly-def resultant-def sylvester-mat-def degree-map-poly-Const deg* **..**
also have ... =
det (*a.mat-hom* (*sylvester-mat-sub* (*degree* (*?pa p*) + *m*) (*degree q*) (*?m p*) *?q*))

unfolding *a.hom-det* **..**
also have *a.mat-hom* (*sylvester-mat-sub* (*degree* (*?pa p*) + *m*) (*degree q*) (*?m p*) *?q*)
= *sylvester-mat-sub* (*degree* (*?pa p*) + *m*) (*degree q*) (*?pa p*) *q*
unfolding *k-def*[*symmetric*] *l-def*[*symmetric*]
by (*intro eq-matI*, *auto simp*: *sylvester-mat-sub-def coeff-map-poly*)
also have *det* ... = *det* (*sylvester-mat-sub* (*degree* (*?pa p*)) (*degree q*) (*?pa p*) *q*) ∗ (*lead-coeff q* ∗ (− *1*) ^ *degree q*) ^ *m*
by (*subst det-sylvester-matrix-higher-degree*, *simp*)
also have *det* (*sylvester-mat-sub* (*degree* (*?pa p*)) (*degree q*) (*?pa p*) *q*) = *resultant* (*?pa p*) *q*
unfolding *resultant-def sylvester-mat-def* **by** *simp*
finally show *?thesis* **unfolding** *m-def* **by** *auto*
qed

lemma *insertion-resultant-mpoly-poly-zero*: **fixes** *q* :: *'a* :: *idom poly*
assumes *q*: *q* ≠ *0*
shows *insertion α* (*resultant-mpoly-poly x p q*) = *0* ⟷ *resultant* (*partial-insertion α x p*) *q* = *0*
unfolding *insertion-resultant-mpoly-poly* **using** *q* **by** *auto*

lemma *vars-resultant*: *vars* (*resultant p q*) ⊆ ⋃ (*vars* ' (*range* (*coeff p*) ∪ *range* (*coeff q*)))
unfolding *resultant-def det-def sylvester-mat-def sylvester-mat-sub-def*
apply *simp*
apply (*rule order.trans*[*OF vars-setsum*])
subgoal using *finite-permutations* **by** *blast*
apply (*rule UN-least*)
apply (*rule order.trans*[*OF vars-mult*])
apply *simp*
apply (*rule order.trans*[*OF vars-prod*])
apply (*rule UN-least*)

by *auto*

By taking the resultant, one variable is deleted.

lemma *vars-resultant-mpoly-poly*: *vars* (*resultant-mpoly-poly x p q*) ⊆ *vars p* − {*x*}

proof
fix *y*
assume *y* ∈ *vars* (*resultant-mpoly-poly x p q*)
from *set-mp*[*OF vars-resultant this*[*unfolded resultant-mpoly-poly-def*]] **obtain** *i*
where *y* ∈ *vars* (*coeff* (*mpoly-to-mpoly-poly x p*) *i*) ∨ *y* ∈ *vars* (*coeff* (*map-poly Const q*) *i*)
by *auto*
moreover have *vars* (*coeff* (*map-poly Const q*) *i*) = {}
by (*subst coeff-map-poly*, *auto*)
ultimately have *y* ∈ *vars* (*coeff* (*mpoly-to-mpoly-poly x p*) *i*) **by** *auto*
thus *y* ∈ *More-MPoly-Type.vars p* − {*x*} **using** *vars-coeff-mpoly-to-mpoly-poly* **by** *blast*
qed

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*)
unfolding *resultant-mpoly-poly-def div-exp-basic.resultant-impl* **by** *simp*

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*
by (*meson injI inv-f-eq of-rat-eq-iff*)

lemma *of-rat-Ints-iff*: ((*of-rat x* :: *'a* :: *field-char-0*) ∈ ℤ) = (*x* ∈ ℤ)
by (*metis Ints-cases Ints-of-int inv-of-rat of-rat-of-int-eq*)

lemma *is-int-code*[*code-unfold*]:
shows (*x* ∈ ℤ) = (*is-rat x* ∧ *is-int-rat* (*to-rat x*))
proof −
have *x* ∈ ℤ ⟷ *x* ∈ ℚ ∧ *x* ∈ ℤ
by (*metis Ints-cases Rats-of-int*)
also have ... = (*is-rat x* ∧ *is-int-rat* (*to-rat x*))

proof (*simp*, *intro conj-cong*[*OF refl*])
assume $x \in \mathbb{Q}$
then obtain y **where** x: $x = \textit{of-rat}\ y$ **unfolding** *Rats-def* **by** *auto*
show $(x \in \mathbb{Z}) = (\textit{to-rat}\ x \in \mathbb{Z})$ **unfolding** x
by (*simp add*: *of-rat-Ints-iff*)
qed
finally show *?thesis* **.**
qed

definition *to-int* :: *'a* :: *is-rat* ⇒ *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*
by (*metis Ints-cases int-of-rat*(*1*) *of-rat-of-int-eq to-int-def to-rat-of-rat*)

lemma *to-int-of-int*: *to-int* (*of-int x*) = *x*
by (*metis int-of-rat*(*1*) *of-rat-of-int-eq to-int-def to-rat-of-rat*)

lemma *to-rat-complex-of-real*[*simp*]: *to-rat* (*complex-of-real x*) = *to-rat x*
by (*metis Re-complex-of-real complex-of-real-of-rat of-rat-to-rat to-rat to-rat-of-rat*)

lemma *to-int-complex-of-real*[*simp*]: *to-int* (*complex-of-real x*) = *to-int x*
by (*simp add*: *to-int-def*)

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* ⟹ *map-mpoly f* (*Const i*) = *Const* (*f i*)

by (*intro mpoly-eqI*, *auto simp*: *coeff-map-mpoly mpoly-coeff-Const*)

lemma *map-mpoly-Var*: *f 1* = *1* ⟹ *map-mpoly* (*f* :: *'b* :: *zero-neq-one* ⇒ -) (*Var i*) = *Var i*
by (*intro mpoly-eqI*, *auto simp*: *coeff-map-mpoly coeff-Var when-def*)

lemma *map-mpoly-monom*: *f 0* = *0* ⟹ *map-mpoly f* (*MPoly-Type.monom m a*) = (*MPoly-Type.monom m* (*f a*))
by (*intro mpoly-eqI*, *unfold coeff-map-mpoly if-distrib coeff-monom*, *simp add*: *when-def*)

lemma *remove-key-single'*:
remove-key v (*Poly-Mapping.single w n*) = (*if v* = *w then 0 else Poly-Mapping.single w n*)
by (*metis add.right-neutral lookup-single-not-eq remove-key-single remove-key-sum single-zero*)

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*))
unfolding *Sum-any.expand-set hom-sum*
by (*rule sum.mono-neutral-right*[*OF fin*], *auto*)

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

by (*unfold-locales*; *intro mpoly-eqI*, *auto simp*: *hom-add*)

lemma *map-mpoly-hom-Const*: *map-mpoly hom* (*Const i*) = *Const* (*hom i*)
by (*rule map-mpoly-Const*, *simp*)

lemma *map-mpoly-hom-monom*: *map-mpoly hom* (*MPoly-Type.monom m a*) = *MPoly-Type.monom m* (*hom a*)
by (*rule map-mpoly-monom*, *simp*)
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*)
by (*rule poly-eqI*, *unfold coeff-mpoly-to-poly coeff-map-poly-hom*, *subst coeff-map-mpoly'*, *auto*)

lemma *comm-ring-hom-mpoly-map*: *comm-ring-hom* (*map-mpoly hom*)
proof −
interpret *mp*: *comm-monoid-add-hom map-mpoly hom* **by** (*rule comm-monoid-add-hom-mpoly-map*)
show *?thesis*
proof (*unfold-locales*)
show *map-mpoly hom 1* = *1*

by (*intro mpoly-eqI*, *simp add*: *MPoly-Type.coeff-def*, *transfer fixing*: *hom*, *transfer fixing*: *hom*, *auto simp*: *when-def*)
fix *x y*
show *map-mpoly hom* (*x* ∗ *y*) = *map-mpoly hom x* ∗ *map-mpoly hom y*
apply (*intro mpoly-eqI*)
apply (*subst coeff-map-mpoly′*, *force*)
apply (*unfold coeff-mpoly-times*)
apply (*subst prod-fun-unfold-prod*, *blast*, *blast*)
apply (*subst prod-fun-unfold-prod*, *blast*, *blast*)
apply (*subst coeff-map-mpoly′*, *force*)
apply (*subst coeff-map-mpoly′*, *force*)
apply (*subst hom-Sum-any*)
subgoal
proof −
let *?X* = {*a*. *MPoly-Type.coeff x a* ≠ *0*}
let *?Y* = {*a*. *MPoly-Type.coeff y a* ≠ *0*}
have *fin*: *finite* (*?X* × *?Y*) **by** *auto*
show *?thesis*
by (*rule finite-subset*[*OF - fin*], *auto*)
qed
apply (*rule Sum-any.cong*)
subgoal for *mon pair* **by** (*cases pair*, *auto simp*: *hom-mult when-def*)
done
qed
qed

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 −
interpret *mp*: *comm-ring-hom map-mpoly hom* **by** (*rule comm-ring-hom-mpoly-map*)
interpret *mmp*: *map-poly-comm-monoid-add-hom map-mpoly hom* **..**
show *?thesis* **unfolding** *mpoly-to-mpoly-poly-def*
apply (*subst mmp.hom-Sum-any*, *force*)
apply (*rule Sum-any.cong*)
apply (*unfold mp.map-poly-hom-monom map-mpoly-hom-monom*)
by *auto*
qed
end

context *inj-comm-ring-hom*
begin
lemma *inj-comm-ring-hom-mpoly-map*: *inj-comm-ring-hom* (*map-mpoly hom*)
proof −
interpret *mp*: *comm-ring-hom map-mpoly hom* **by** (*rule comm-ring-hom-mpoly-map*)
show *?thesis*
proof (*unfold-locales*)
fix *x*
assume *0*: *map-mpoly hom x* = *0*

show *x* = *0*
proof (*intro mpoly-eqI*)
fix *m*
show *MPoly-Type.coeff x m* = *MPoly-Type.coeff 0 m*
using *arg-cong*[*OF 0*, *of* λ *p*. *MPoly-Type.coeff p m*] **by** *simp*
qed
qed
qed

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 −
interpret *mp*: *inj-comm-ring-hom map-mpoly hom* **by** (*rule inj-comm-ring-hom-mpoly-map*)
show *?thesis*
unfolding *resultant-mpoly-poly-def*
unfolding *mpoly-to-mpoly-poly-map-mpoly-hom*
apply (*subst mp.resultant-map-poly*[*symmetric*])
subgoal by (*subst mp.degree-map-poly-hom*, *unfold-locales*, *auto*)
subgoal by (*subst mp.degree-map-poly-hom*, *unfold-locales*, *auto*)
subgoal
apply (*rule arg-cong*[*of - - resultant -*], *intro poly-eqI*)
apply (*subst coeff-map-poly*, *force*)+
by (*simp add*: *map-mpoly-hom-Const*)
done
qed
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*)
by (*induct xs*, *auto*)

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*)
by (*induct xs*, *auto simp*: *map-insort-key*)

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 (*cases resultant f g* = *0*)

case *True*
from *resultant-zero-imp-common-factor*[*OF deg True*] *True*
show *?thesis* **by** *simp*
next
case *False*
from *deg* **have** *fg*: $f \neq 0 \vee g \neq 0$ **by** *auto*
from *resultant-non-zero-imp-coprime*[*OF False fg*] *deg False*
show *?thesis* **by** *auto*
qed

lemma *resultant-zero-iff-coprime*: **fixes** *f g* :: *'a* :: *field poly*
assumes $f \neq 0 \vee g \neq 0$
shows *resultant f g* $= 0 \longleftrightarrow \neg$ *coprime f g*
proof (*cases degree f* $> 0 \vee$ *degree g* > 0)
case *True*
thus *?thesis* **using** *resultant-iff-coprime-main*[*OF True*] **by** *simp*
next
case *False*
hence *degree f* $= 0$ *degree g* $= 0$ **by** *auto*
then obtain *c d* **where** *f*: *f* = [:*c*:] **and** *g*: *g* = [:*d*:] **using** *degree0-coeffs* **by** *metis+*
from *assms* **have** *cd*: $c \neq 0 \vee d \neq 0$ **unfolding** *f g* **by** *auto*
have *res*: *resultant f g* = *1* **unfolding** *f g resultant-const* **by** *auto*
have *coprime f g*
by (*metis assms one-neq-zero res resultant-non-zero-imp-coprime*)
with *res* **show** *?thesis* **by** *auto*
qed

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* $= 0 \longleftrightarrow (\exists\ x.$ *poly f x* $= 0 \wedge$ *poly g x* $= 0)$
proof −
from *irr root* **have** *deg*: *degree g* $\neq 0$ **using** *degree0-coeffs*[*of g*] **by** *fastforce*
show *?thesis*
proof
assume $\exists\ x.$ *poly f x* $= 0 \wedge$ *poly g x* $= 0$
then obtain *x* **where** *poly f x* = *0* *poly g x* = *0* **by** *auto*
from *resultant-zero*[*OF - this*] *deg* **show** *resultant f g* = *0* **by** *auto*
next
assume *resultant f g* = *0*
from *resultant-zero-imp-common-factor*[*OF - this*] *deg*
have ¬ *coprime f g* **by** *auto*
from *this*[*unfolded not-coprime-iff-common-factor*] **obtain** *r* **where**
rf: *r dvd f* **and** *rg*: *r dvd g* **and** *r*: ¬ *is-unit r* **by** *auto*

```
    from rg r irr have g dvd r
      by (meson algebraic-semidom-class.irreducible-altdef)
    with rf have g dvd f by auto
    with root show ∃ x. poly f x = 0 ∧ poly g x = 0
      by (intro exI[of - a], auto simp: dvd-def)
  qed
qed
```

```
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 (cases degree g = 0)
  case deg: False
  show ?thesis
  proof
    assume ∃ x. poly f x = 0 ∧ poly g x = 0
    then obtain x where poly f x = 0 poly g x = 0 by auto
    from resultant-zero[OF - this] deg show resultant f g = 0 by auto
  next
    assume resultant f g = 0
    from resultant-zero-imp-common-factor[OF - this] deg
    have ¬ coprime f g by auto
    from this[unfolded not-coprime-iff-common-factor] obtain r where
      rf: r dvd f and rg: r dvd g and r: ¬ is-unit r by auto
    from rg g have r0: r ≠ 0 by auto
    with r have degr: degree r ≠ 0 by simp
    hence ¬ constant (poly r)
      by (simp add: constant-degree)
    from fundamental-theorem-of-algebra[OF this] obtain a where root: poly r a
= 0 by auto
    from rf rg root show ∃ x. poly f x = 0 ∧ poly g x = 0
      by (intro exI[of - a], auto simp: dvd-def)
  qed
next
  case deg: True
  from degree0-coeffs[OF deg] obtain c where gc: g = [:c:] by auto
  from gc g have c: c ≠ 0 by auto
  hence resultant f g ≠ 0 unfolding gc resultant-const by simp
  with gc c show ?thesis by auto
qed
```

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 ∧
```

(∀ i ∈ N. poly (qs i) (α (Suc i)) = 0))

The upcoming lemma shows how to eliminate single variables in multivariate 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 ∉ qs ‘ N
  and i: i ∈ N
shows mpoly-polys-solution (resultant-mpoly-poly (Suc i) p (qs i)) qs (N − {i}) α
  ⟷ (∃ v. mpoly-polys-solution p qs N (α((Suc i) := v)))
proof −
  let ?x = Suc i
  let ?q = qs i
  let ?mres = resultant-mpoly-poly ?x p ?q
  from i obtain M where N: N = insert i M and MN: M = N − {i} and iM:
i ∉ M by auto
  from nz i have nzq: ?q ≠ 0 by auto
  hence lc0: lead-coeff (qs i) ≠ 0 by auto
  have mpoly-polys-solution ?mres qs (N − {i}) α ⟷
   insertion α ?mres = 0 ∧ (∀ i ∈ M. poly (qs i) (α (Suc i)) = 0)
    unfolding mpoly-polys-solution-def MN ..
  also have insertion α ?mres = 0 ⟷ resultant (partial-insertion α ?x p) ?q =
0
    by (rule insertion-resultant-mpoly-poly-zero[OF nzq])
  also have ... ⟷ (∃ v. poly (partial-insertion α ?x p) v = 0 ∧ poly ?q v = 0)
    by (rule resultant-zero-iff-common-root-complex[OF nzq])
  also have ... ⟷ (∃ v. insertion (α(?x := v)) p = 0 ∧ poly ?q v = 0) (is ?lhs
= ?rhs)
  proof (intro iff-exI conj-cong refl arg-cong[of - - λ x. x = 0])
    fix v
    have poly (partial-insertion α ?x p) v = poly (partial-insertion α ?x p) ((α(?x
:= v)) ?x) by simp
    also have ... = insertion (α(?x := v)) p
      by (rule insertion-partial-insertion, auto)
    finally show poly (partial-insertion α ?x p) v = insertion (α(?x := v)) p .
  qed
  also have ... ∧ (∀ i∈M. poly (qs i) (α (Suc i)) = 0)
    ⟷ (∃ v. insertion (α(?x := v)) p = 0 ∧ poly (qs i) v = 0 ∧ (∀ i∈M. poly (qs
i) ((α(?x := v)) (Suc i)) = 0))
    using iM by auto
  also have ... ⟷ (∃ v. mpoly-polys-solution p qs N (α((Suc i) := v)))
    unfolding mpoly-polys-solution-def N by (intro iff-exI, auto)
  finally
  show ?thesis .
qed
```

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
∧ (∀ *i*. *i* ∉ *insert 0* (*Suc* ' *N*) ⟶ α *i* = *0*))

lemma *resultant-mpoly-polys-zero-solution*: **fixes** *p* :: *complex mpoly*
assumes *nz*: *0* ∉ *qs* ' *N*
and *i*: *i* ∈ *N*
shows
mpoly-polys-zero-solution (*resultant-mpoly-poly* (*Suc i*) *p* (*qs i*)) *qs* (*N* − {*i*}) α
⟹ ∃ *v*. *mpoly-polys-zero-solution p qs N* (α(*Suc i* := *v*))
mpoly-polys-zero-solution p qs N α
⟹ *mpoly-polys-zero-solution* (*resultant-mpoly-poly* (*Suc i*) *p* (*qs i*)) *qs* (*N* − {*i*}) (α(*Suc i* := *0*))
proof −
assume *mpoly-polys-zero-solution* (*resultant-mpoly-poly* (*Suc i*) *p* (*qs i*)) *qs* (*N* − {*i*}) α
hence *1*: *mpoly-polys-solution* (*resultant-mpoly-poly* (*Suc i*) *p* (*qs i*)) *qs* (*N* − {*i*}) α **and** *2*: (∀ *i*. *i* ∉ *insert 0* (*Suc* ' (*N* − {*i*})) ⟶ α *i* = *0*)
unfolding *mpoly-polys-zero-solution-def* **by** *auto*
from *resultant-mpoly-polys-solution*[*of qs N - p* α, *OF nz i*] *1* **obtain** *v* **where** *mpoly-polys-solution p qs N* (α(*Suc i* := *v*)) **by** *auto*
with *2* **have** *mpoly-polys-zero-solution p qs N* (α(*Suc i* := *v*)) **using** *i* **unfolding** *mpoly-polys-zero-solution-def* **by** *auto*
thus ∃ *v*. *mpoly-polys-zero-solution p qs N* (α(*Suc i* := *v*)) **..**
next
assume *mpoly-polys-zero-solution p qs N* α
from *this*[*unfolded mpoly-polys-zero-solution-def*] **have** *1*: *mpoly-polys-solution p qs N* α **and** *2*: ∀ *i*. *i* ∉ *insert 0* (*Suc* ' *N*) ⟶ α *i* = *0* **by** *auto*
from *1* **have** *mpoly-polys-solution p qs N* (α(*Suc i* := α (*Suc i*))) **by** *auto*
hence ∃ *v*. *mpoly-polys-solution p qs N* (α(*Suc i* := *v*)) **by** *blast*
with *resultant-mpoly-polys-solution*[*of qs N - p* α, *OF nz i*] **have** *mpoly-polys-solution* (*resultant-mpoly-poly* (*Suc i*) *p* (*qs i*)) *qs* (*N* − {*i*}) α **by** *auto*
hence *mpoly-polys-solution* (*resultant-mpoly-poly* (*Suc i*) *p* (*qs i*)) *qs* (*N* − {*i*}) (α (*Suc i* := *0*))
unfolding *mpoly-polys-solution-def*
apply *simp*
apply (*subst insertion-irrelevant-vars*[*of - -* α])
by (*insert vars-resultant-mpoly-poly*, *auto*)
thus *mpoly-polys-zero-solution* (*resultant-mpoly-poly* (*Suc i*) *p* (*qs i*)) *qs* (*N* − {*i*}) (α(*Suc i* := *0*))
unfolding *mpoly-polys-zero-solution-def* **using** *2* **by** *auto*
qed

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 −
let *?solN* = *mpoly-polys-zero-solution p qs N*
let *?solN1* = *mpoly-polys-zero-solution* (*resultant-mpoly-poly* (*Suc i*) *p* (*qs i*)) *qs* (*N* − {*i*})
let *?x* = *Suc i*
note *defs* = *mpoly-polys-zero-solution-def*
define *zero* **where** *zero* α = α(*?x* := *0*) **for** α :: *nat* ⇒ *complex*
{
fix α
assume *sol*: *?solN1* α
from *sol*[*unfolded defs*] **have** *0*: α *?x* = *0* **by** *auto*
from *resultant-mpoly-polys-zero-solution*(*1*)[*of qs N i p*, *OF nz i sol*] **obtain** *v* **where** *?solN* (α(*?x* := *v*)) **by** *auto*
hence *sol*: α(*?x* := *v*) ∈ {α. *?solN* α} **by** *auto*
hence *zero* (α(*?x* := *v*)) ∈ *zero* ‘ {α. *?solN* α} **by** *auto*
also have *zero* (α(*?x* := *v*)) = α **using** *0* **by** (*auto simp*: *zero-def*)
finally have α ∈ *zero* ‘ {α. *?solN* α} **.**
}
hence {α. *?solN1* α} ⊆ *zero* ‘ {α. *?solN* α} **by** *blast*
from *finite-subset*[*OF this finite-imageI*[*OF fin*]]
show *?thesis* **.**
qed

lemma *finite-resultant-mpoly-polys-empty*: **fixes** *p* :: *complex mpoly*
assumes *finite* {α. *mpoly-polys-zero-solution p qs* {} α}
shows *p* ≠ *0*
proof
define *g* **where** *g x* = (λ *i* :: *nat*. *if i* = *0 then x else 0*) **for** *x* :: *complex*
assume *p* = *0*
hence ∀ *x*. *mpoly-polys-zero-solution p qs* {} (*g x*)
unfolding *mpoly-polys-zero-solution-def mpoly-polys-solution-def g-def* **by** *auto*
hence *range g* ⊆ {α. *mpoly-polys-zero-solution p qs* {} α} **by** *auto*
from *finite-subset*[*OF this assms*] **have** *finite* (*range g*) **.**
moreover have *inj g* **unfolding** *g-def inj-on-def* **by** *metis*
ultimately have *finite* (*UNIV* :: *complex set*) **by** *simp*
thus *False* **using** *infinite-UNIV-char-0* **by** *auto*
qed

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 −
let *?h* = *of-int* :: - ⇒ ′*a*
interpret *mp*: *comm-ring-hom* (*map-mpoly ?h*)
by (*rule of-int-hom.comm-ring-hom-mpoly-map*)
show *?thesis*
proof (*induct is arbitrary*: *mp*)
case *Nil*
show *?case* **by** (*simp add*: *of-int-hom.mpoly-to-poly-map-mpoly-hom*)
next
case (*Cons i is mp*)
show *?case* **unfolding** *eliminate-aux-vars.simps Cons*[*symmetric*]
apply (*rule arg-cong*[*of* - - λ *x*. *eliminate-aux-vars x* - -], *unfold o-def*)
by (*rule of-int-hom.resultant-mpoly-poly-hom*)
qed
qed

The polynomial of the elimination process will represent the first value *α 0* 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*
using *assms*
proof (*induct is arbitrary*: *p*)
case (*Nil p*)
from *Nil*(*3*) *finite-resultant-mpoly-polys-empty*[*of p*]
have *p0*: *p* ≠ *0* **by** *auto*
from *Nil*(*2*) **have** *vars*: *vars p* ⊆ {*0*} **by** *auto*
note [*simp*] = *poly-eq-insertion*[*OF this*]
from *Nil*(*5*)[*unfolded mpoly-polys-solution-def*]
have *insertion α p* = *0* **by** *auto*
also have *insertion α p* = *insertion* (*λv*. *α 0*) *p*
by (*rule insertion-irrelevant-vars*, *insert vars*, *auto*)
finally
show *?case* **using** *p0 mpoly-to-poly-inverse*[*OF vars*] **by** (*auto simp*: *poly-to-mpoly0*)
next
case (*Cons i is p*)
let *?x* = *Suc i*

let ?p = resultant-mpoly-poly ?x p (qs i)
have dist: distinct is **using** Cons(2) **by** auto
have vars: vars ?p ⊆ insert 0 (Suc ‘ set is) **using** Cons(3) vars-resultant-mpoly-poly[of ?x p qs i] **by** auto
have fin: finite {α. mpoly-polys-zero-solution ?p qs (set is) α}
using finite-resultant-mpoly-polys-non-empty[of qs set (i # is) i p, OF Cons(5)] Cons(2,4) **by** auto
have 0: 0 ∉ qs ‘ set is **using** Cons(5) **by** auto
have (∃ v. mpoly-polys-solution p qs (set (i # is)) (α(?x := v)))
using Cons(6) **by** (intro exI[of - α ?x], auto)
from this resultant-mpoly-polys-solution[OF Cons(5), of i p α]
have mpoly-polys-solution ?p qs (set (i # is) − {i}) α
by auto
also have set (i # is) − {i} = set is **using** Cons(2) **by** auto
finally have mpoly-polys-solution ?p qs (set is) α **by** auto
note IH = Cons(1)[OF dist vars fin 0 this]
show ?case **unfolding** eliminate-aux-vars.simps **using** IH **by** simp
qed

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 −
  define *rcs* **where** *rcs* = *remdups* (*filter* (λ*c*. *c* ∉ ℤ) (*coeffs p*))
  define *spairs* **where** *spairs* = *sort-key* (λ(*c*, *f*). *degree f*) (*map* (λ*c*. (*c*, *min-int-poly c*)) *rcs*)
  let *?find* = λ *i*. *find* (λ(*j*, *d*, *f*). *d* = *coeff p i*) *triples*
  define *trans* **where** *trans i* = (*case ?find i of None* ⇒ *Const* (*to-int* (*coeff p i*))
    | *Some* (*j*, *pair*) ⇒ *Var* (*Suc j*)) **for** *i*
  note *res* = *res*[*unfolded initial-root-problem-def Let-def*, *folded rcs-def*, *folded spairs-def*]
  have *triples*: *triples* = *zip* [*0*..<*length spairs*] *spairs* **using** *res* **by** *auto*
  note *res* = *res*[*folded triples*, *folded trans-def*]
  have *mp*: *mp* = (∑ *i*≤*degree p*. *MPoly-Type.monom* (*Poly-Mapping.single 0 i*) *1* ∗ *trans i*) **using** *res* **by** *auto*
  have *dist-rcs*: *distinct rcs* **unfolding** *rcs-def* **by** *auto*
  hence *distinct* (*map fst* (*map* (λ*c*. (*c*, *min-int-poly c*)) *rcs*)) **by** (*simp add*: *o-def*)
  hence *dist-spairs*: *distinct* (*map fst spairs*) **unfolding** *spairs-def*
    by (*metis* (*no-types*, *lifting*) *distinct-map distinct-sort set-sort*)
  {
    fix *c*
    assume *c* ∈ *set rcs*
    hence *c* ∈ *set* (*coeffs p*) **unfolding** *rcs-def* **by** *auto*
    with *algebraic* **have** *algebraic c* **by** *auto*
  } **note** *rcs-alg* = *this*
  {
    fix *c*
    assume *c*: *c* ∈ *range* (*coeff p*) *c* ∉ ℤ
    hence *c* ∈ *set* (*coeffs p*) **unfolding** *range-coeff* **by** *auto*
    with *c* **have** *crcs*: *c* ∈ *set rcs* **unfolding** *rcs-def* **by** *auto*
    from *rcs-alg*[*OF crcs*] **have** *algebraic c* **.**
    from *min-int-poly-represents*[*OF this*]
    have *min-int-poly c represents c* **.**
    hence ∃ *f*. (*c*,*f*) ∈ *set spairs* ∧ *f represents c* **using** *crcs* **unfolding** *spairs-def* **by** *auto*
  }
  have *dist-is*: *distinct is* **unfolding** *is triples* **by** *simp*
  note *eliminate* = *eliminate-aux-vars*[*OF dist-is*]
  let *?mp* = *map-mpoly of-int mp* :: *complex mpoly*
  have *vars-mp*: *vars mp* ⊆ *insert 0* (*Suc* ‘ *set is*)
    unfolding *mp*
    apply (*rule order.trans*[*OF vars-setsum*], *force*)

apply (*rule UN-least*, *rule order.trans*[*OF vars-mult*], *rule Un-least*)
apply (*intro order.trans*[*OF vars-monom-single*], *force*)
subgoal for *i*
proof −
show *?thesis*
proof (*cases ?find i*)
case *None*
show *?thesis* **unfolding** *trans-def None* **by** *auto*
next
case (*Some j-pair*)
then obtain *j c f* **where** *find*: *?find i = Some* (*j*,*c*,*f*) **by** (*cases j-pair*, *auto*)
from *find-Some-D*[*OF find*] **have** *Suc j* ∈ *Suc* ‘ (*fst* ‘ *set triples*) **by** *force*
thus *?thesis* **unfolding** *trans-def find* **by** (*simp add*: *vars-Var is*)
qed
qed
done
hence *varsMp*: *vars ?mp* ⊆ *insert 0* (*Suc* ‘ *set is*) **using** *vars-map-mpoly-subset* **by** *auto*
note *eliminate* = *eliminate*[*OF this*]
let *?f* = λ *j*. *snd* (*snd* (*triples* ! *j*))
let *?c* = λ *j*. *fst* (*snd* (*triples* ! *j*))
{
fix *j*
assume *j* ∈ *set is*
hence (*?c j*, *?f j*) ∈ *set spairs* **unfolding** *is triples* **by** *simp*
hence *?f j represents ?c j ?f j* = *min-int-poly* (*?c j*) **unfolding** *spairs-def*
by (*auto intro*: *min-int-poly-represents*[*OF rcs-alg*])
} **note** *is-repr* = *this*
let *?qs* = (*of-int-poly o qs*) :: *nat* ⇒ *complex poly*
{
fix *j*
assume *j* ∈ *set is*
hence *j* < *length is* **unfolding** *is triples* **by** *simp*
} **note** *j-len* = *this*
have *qs-0*: *0* ∉ *qs* ‘ *set is*
proof
assume *0* ∈ *qs* ‘ *set is*
then obtain *j* **where** *j*: *j* ∈ *set is* **and** *0*: *qs j* = *0* **by** *auto*
from *is-repr*[*OF j*] **have** *?f j* ≠ *0* **by** *auto*
with *0* **show** *False* **unfolding** *qs*[*OF j-len*[*OF j*]] **by** *auto*
qed
hence *qs0*: *0* ∉ *?qs* ‘ *set is* **by** *auto*
note *eliminate* = *eliminate*[*OF - this*]
define *roots* **where** *roots p* = (*SOME xs*. *set xs* = {*x* . *poly p x* = *0*}) **for** *p* :: *complex poly*
{
fix *p* :: *complex poly*
assume *p* ≠ *0*
from *someI-ex*[*OF finite-list*[*OF poly-roots-finite*[*OF this*]], *folded roots-def*]

```
    have set (roots p) = {x. poly p x = 0} .
  } note roots = this
  define qs-roots where qs-roots = concat-lists (map (λ i. roots (?qs i)) [0 ..<
length triples])
  define evals where evals = concat (map (λ part. let
    q = partial-insertion (λ i. part ! (i − 1)) 0 ?mp;
    new-roots = roots q
    in map (λ r. r # part) new-roots) qs-roots)
  define conv where conv roots i = (if i ≤ length triples then roots ! i else 0 ::
complex) for roots i
  define alphas where alphas = map conv evals
  {
    fix n
    assume n: n ∈ {..degree p}
    let ?cn = coeff p n
    from n have mem: ?cn ∈ set (coeffs p) using p unfolding Polynomial.coeffs-def
by force
    {
      assume ?cn ∉ ℤ
      with mem have ?cn ∈ set rcs unfolding rcs-def by auto
      hence (?cn, min-int-poly ?cn) ∈ set spairs unfolding spairs-def by auto
      hence ∃ i. (i, ?cn, min-int-poly ?cn) ∈ set triples unfolding triples set-zip
set-conv-nth
        by force
      hence ?find n ≠ None unfolding find-None-iff by auto
    }
  } note non-int-find = this
  have fin: finite {α. mpoly-polys-zero-solution ?mp ?qs (set is) α}
  proof (rule finite-subset[OF - finite-set[of alphas]], standard, clarify)
    fix α
    assume sol: mpoly-polys-zero-solution ?mp ?qs (set is) α
    define part where part = map (λ i. α (Suc i)) [0 ..< length triples]
    {
      fix i
      assume i > length triples
      hence i ∉ insert 0 (Suc ‘ set is) unfolding triples is by auto
      hence α i = 0 using sol[unfolded mpoly-polys-zero-solution-def] by auto
    } note alpha0 = this
    {
      fix i
      assume i < length triples
      hence i: i ∈ set is unfolding triples is by auto
      from qs0 i have 0: ?qs i ≠ 0 by auto
      from i sol[unfolded mpoly-polys-zero-solution-def mpoly-polys-solution-def]
      have poly (?qs i) (α (Suc i)) = 0 by auto
      hence α (Suc i) ∈ set (roots (?qs i)) poly (?qs i) (α (Suc i)) = 0 using
roots[OF 0] by auto
    } note roots2 = this
    hence part: part ∈ set qs-roots
```

unfolding *part-def qs-roots-def concat-lists-listset listset* **by** *auto*
let *?gamma* = (*λi. part* ! (*i* − *1*))
let *?f* = *partial-insertion ?gamma 0 ?mp*
have *α 0* ∈ *set* (*roots ?f*)
proof −
from *sol*[*unfolded mpoly-polys-zero-solution-def mpoly-polys-solution-def*]
have *0* = *insertion α ?mp* **by** *simp*
also have ... = *insertion* (*λ i. if i* ≤ *length triples then α i else part* ! (*i* − *1*)) *?mp*
(**is** - = *insertion ?beta* -)
proof (*rule insertion-irrelevant-vars*)
fix *i*
assume *i* ∈ *vars ?mp*
from *set-mp*[*OF varsMp this*] **have** *i* ≤ *length triples* **unfolding** *triples is* **by** *auto*
thus *α i* = *?beta i* **by** *auto*
qed
also have ... = *poly* (*partial-insertion* (*?beta*(*0* := *part* ! *0*)) *0 ?mp*) (*?beta 0*)
by (*subst insertion-partial-insertion*, *auto*)
also have *?beta*(*0* := *part* ! *0*) = *?gamma* **unfolding** *part-def*
by (*intro ext*, *auto*)
finally have *root*: *poly ?f* (*α 0*) = *0* **by** *auto*
have *?f* ≠ *0*
proof
interpret *mp*: *inj-comm-ring-hom map-mpoly complex-of-int*
by (*rule of-int-hom.inj-comm-ring-hom-mpoly-map*)
assume *?f* = *0*
hence *0* = *coeff ?f* (*degree p*) **by** *simp*
also have ... = *insertion ?gamma* (*coeff* (*mpoly-to-mpoly-poly 0 ?mp*) (*degree p*))
unfolding *insertion-coeff-mpoly-to-mpoly-poly*[*symmetric*] **..**
also have *coeff* (*mpoly-to-mpoly-poly 0 ?mp*) (*degree p*) = *map-mpoly of-int* (*coeff* (*mpoly-to-mpoly-poly 0 mp*) (*degree p*))
unfolding *of-int-hom.mpoly-to-mpoly-poly-map-mpoly-hom*
by (*subst coeff-map-poly*, *auto*)
also have *coeff* (*mpoly-to-mpoly-poly 0 mp*) (*degree p*) =
(∑ *x. MPoly-Type.monom* (*remove-key 0 x*) (*MPoly-Type.coeff mp x*) *when lookup x 0* = *degree p*)
unfolding *mpoly-to-mpoly-poly-def when-def*
by (*subst coeff-hom.hom-Sum-any*, *force*, *unfold Polynomial.coeff-monom*, *auto*)
also have ... = (∑ *x. MPoly-Type.monom* (*remove-key 0 x*)
(∑ *xa*≤*degree p. let xx* = *Poly-Mapping.single 0 xa in*
∑ (*a*, *b*). *MPoly-Type.coeff* (*trans xa*) *b when x* = *xx* + *b when*
a = *xx*) *when*
lookup x 0 = *degree p*) **unfolding** *mp coeff-sum More-MPoly-Type.coeff-monom coeff-mpoly-times Let-def*
apply (*subst prod-fun-unfold-prod*, *force*, *force*)

apply (*unfold when-mult*, *subst when-commute*)
by (*auto simp*: *when-def intro*!: *Sum-any.cong sum.cong if-cong arg-cong*[*of - - MPoly-Type.monom -*])
also have ... = ($\sum$ *x*. *MPoly-Type.monom* (*remove-key 0 x*)
($\sum$ *i*≤*degree p*. $\sum$ *m*. *MPoly-Type.coeff* (*trans i*) *m when x* = *Poly-Mapping.single 0 i* + *m*) *when*
lookup x 0 = *degree p*)
unfolding *Sum-any-when-dependent-prod-left Let-def* **by** *simp*
also have ... = ($\sum$ *x*. *MPoly-Type.monom* (*remove-key 0 x*)
($\sum$ *i* ∈ {*degree p*}. $\sum$ *m*. *MPoly-Type.coeff* (*trans i*) *m when x* = *Poly-Mapping.single 0 i* + *m*) *when*
lookup x 0 = *degree p*)
apply (*intro Sum-any.cong when-cong refl arg-cong*[*of - - MPoly-Type.monom -*] *sum.mono-neutral-right*, *force*+)
apply (*intro ballI Sum-any-zeroI*, *auto simp*: *when-def*)
subgoal for *i x*
proof (*goal-cases*)
case *1*
hence *lookup x 0* > *0* **by** (*auto simp*: *lookup-add*)
moreover have *0* ∉ *vars* (*trans i*) **unfolding** *trans-def*
by (*auto split*: *option.splits simp*: *vars-Var*)
ultimately show *?thesis*
by (*metis set-mp coeff-notin-vars in-keys-iff neq0-conv*)
qed
done
also have ... = ($\sum$ *x*. *MPoly-Type.monom* (*remove-key 0 x*)
($\sum$ *m*. *MPoly-Type.coeff* (*trans* (*degree p*)) *m when x* = *Poly-Mapping.single 0* (*degree p*) + *m*) *when*
lookup x 0 = *degree p*) (**is** *-* = *?mid*)
by *simp*
also have *insertion ?gamma* (*map-mpoly of-int* ...) ≠ *0*
proof (*cases ?find* (*degree p*))
case *None*
from *non-int-find*[*of degree p*] *None*
have *lcZ*: *lead-coeff p* ∈ ℤ **by** *auto*
have *?mid* = ($\sum$ *x*. *MPoly-Type.monom* (*remove-key 0 x*)
($\sum$ *m*. (*to-int* (*lead-coeff p*) *when*
x = *Poly-Mapping.single 0* (*degree p*) + *m when m* = *0*)) *when*
lookup x 0 = *degree p*)
using *None* **unfolding** *trans-def None option.simps mpoly-coeff-Const when-def*
by (*intro Sum-any.cong if-cong refl*, *intro arg-cong*[*of - - MPoly-Type.monom -*] *Sum-any.cong*, *auto*)
also have ... = ($\sum$ *x*. *MPoly-Type.monom* (*remove-key 0 x*)
(*to-int* (*lead-coeff p*) *when x* = *Poly-Mapping.single 0* (*degree p*)) *when*
lookup x 0 = *degree p when x* = *Poly-Mapping.single 0* (*degree p*))
unfolding *Sum-any-when-equal*[*of - 0*]
by (*intro Sum-any.cong*, *auto simp*: *when-def*)
also have ... = *MPoly-Type.monom* (*remove-key 0* (*Poly-Mapping.single*

0 (*degree p*)))
(*to-int* (*lead-coeff p*))
unfolding *Sum-any-when-equal* **by** *simp*
also have . . . = *Const* (*to-int* (*lead-coeff p*)) **by** (*simp add*: *mpoly-monom-0-eq-Const*)
also have *map-mpoly of-int* . . . = *Const* (*lead-coeff p*)
unfolding *of-int-hom.map-mpoly-hom-Const of-int-to-int*[*OF lcZ*] **by** *simp*
also have *insertion ?gamma* . . . = *lead-coeff p* **by** *simp*
also have . . . $\neq$ *0* **using** *p* **by** *auto*
finally show *?thesis* **.**
next
case *Some*
from *find-Some-D*[*OF this*] *Some* **obtain** *j f* **where** *mem*: (*j*,*lead-coeff p*,*f*) $\in$ *set triples* **and**
Some: *?find* (*degree p*) = *Some* (*j*, *lead-coeff p*, *f*) **by** *auto*
from *mem* **have** *j*: *j* < *length triples* **unfolding** *triples set-zip* **by** *auto*
have *?mid* = ($\sum$ *x*. *if lookup x 0* = *degree p*
then MPoly-Type.monom (*remove-key 0 x*)
($\sum$ *m*. *1 when m* = *Poly-Mapping.single* (*Suc j*) *1 when x* = *Poly-Mapping.single 0* (*degree p*) + *m*)
else 0)
unfolding *trans-def Some option.simps split when-def coeff-Var* **by** *auto*
also have . . . = ($\sum$ *x*. *if lookup x 0* = *degree p*
then MPoly-Type.monom (*remove-key 0 x*) *1*
when x = *Poly-Mapping.single 0* (*degree p*) + *Poly-Mapping.single* (*Suc j*) *1*
else 0 when x = *Poly-Mapping.single 0* (*degree p*) + *Poly-Mapping.single* (*Suc j*) *1*)
apply (*subst when-commute*)
apply (*unfold Sum-any-when-equal*)
by (*rule Sum-any.cong*, *auto simp*: *when-def*)
also have . . . = ($\sum$ *x*. (*MPoly-Type.monom* (*remove-key 0 x*) *1 when lookup x 0* = *degree p*)
when x = *Poly-Mapping.single 0* (*degree p*) + *Poly-Mapping.single* (*Suc j*) *1*)
by (*rule Sum-any.cong*, *auto simp*: *when-def*)
also have . . . = *MPoly-Type.monom* (*Poly-Mapping.single* (*Suc j*) *1*) *1*
unfolding *Sum-any-when-equal* **unfolding** *when-def*
by (*simp add*: *lookup-add remove-key-add*[*symmetric*]
remove-key-single′ lookup-single)
also have . . . = *Var* (*Suc j*)
by (*intro mpoly-eqI*, *simp add*: *coeff-Var coeff-monom*)
also have *map-mpoly complex-of-int* . . . = *Var* (*Suc j*)
by (*simp add*: *map-mpoly-Var*)
also have *insertion ?gamma* . . . = *part* ! *j* **by** *simp*
also have . . . = α (*Suc j*) **unfolding** *part-def* **using** *j* **by** *auto*
also have . . . $\neq$ *0*
proof
assume α (*Suc j*) = *0*

with roots2(2)[OF j] **have** root0: poly (?qs j) 0 = 0 **by** auto
from j is **have** ji: j < length is **by** auto
hence jis: j ∈ set is **unfolding** is triples set-zip **by** auto
from mem **have** tj: triples ! j = (j, lead-coeff p, f) **unfolding** triples set-zip **by** auto
from root0[unfolded qs[OF ji] o-def tj]
have rootf: poly f 0 = 0 **by** auto
from is-repr[OF jis, unfolded tj] **have** rootlc: ipoly f (lead-coeff p) = 0
and f: f = min-int-poly (lead-coeff p) **by** auto
from f **have** irr: irreducible f **by** auto
from rootf **have** [:0,1:] dvd f **using** dvd-iff-poly-eq-0 **by** fastforce
from this[unfolded dvd-def] **obtain** g **where** f: f = [:0, 1:] ∗ g **by** auto
from irreducibleD[OF irr f] **have** is-unit g
by (metis is-unit-poly-iff one-neq-zero one-pCons pCons-eq-iff)
then obtain c **where** g: g = [:c:] **and** c: c dvd 1 **unfolding** is-unit-poly-iff **by** auto
from rootlc[unfolded f g] c **have** lead-coeff p = 0 **by** auto
with p **show** False **by** auto
qed
finally show ?thesis **.**
qed
finally show False **by** auto
qed
from roots[OF this] root **show** ?thesis **by** auto
qed
hence α 0 # part ∈ set evals
unfolding evals-def set-concat Let-def set-map
by (auto intro!: bexI[OF - part])
hence map α [0 ..< Suc (length triples)] ∈ set evals **unfolding** part-def
by (metis Utility.map-upt-Suc)
hence conv (map α [0 ..< Suc (length triples)]) ∈ set alphas **unfolding** alphas-def **by** auto
also have conv (map α [0 ..< Suc (length triples)]) = α
proof
fix i
show conv (map α [0..<Suc (length triples)]) i = α i
unfolding conv-def **using** alpha0
by (cases i < length triples; cases i = length triples; auto simp: nth-append)
qed
finally show α ∈ set alphas **.**
qed
note eliminate = eliminate[OF this]
define α **where** α x j = (if j = 0 then x else ?c (j − 1)) **for** x j
have α: α x (Suc j) = ?c j α x 0 = x **for** j x **unfolding** α-def **by** auto
interpret mp: inj-comm-ring-hom map-mpoly complex-of-int **by** (rule of-int-hom.inj-comm-ring-hom-mpoly-r
have ins: insertion (α x) ?mp = poly p x **for** x
unfolding poly-altdef mp mp.hom-sum insertion-sum insertion-mult mp.hom-mult
proof (rule sum.cong[OF refl], subst mult.commute, rule arg-cong2[of - - - - (∗)])
fix n

assume *n*: $n \in \{..degree\ p\}$
let *?cn* = *coeff p n*
from *n* **have** *mem*: *?cn* ∈ *set* (*coeffs p*) **using** *p* **unfolding** *Polynomial.coeffs-def*
by *force*
have *insertion* (α *x*) (*map-mpoly complex-of-int* (*MPoly-Type.monom* (*Poly-Mapping.single 0 n*) *1*)) = (∏ *a*. α *x a* ^ (*n when a* = *0*))
unfolding *of-int-hom.map-mpoly-hom-monom* **by** (*simp add*: *lookup-single*)
also have ... = (∏ *a*. *if a* = *0 then* α *x a* ^ *n else 1*)
by (*rule Prod-any.cong*, *auto simp*: *when-def*)
also have ... = α *x 0* ^ *n* **by** *simp*
also have ... = *x* ^ *n* **unfolding** α **..**
finally show *insertion* (α *x*) (*map-mpoly complex-of-int* (*MPoly-Type.monom* (*Poly-Mapping.single 0 n*) *1*)) = *x* ^ *n* **.**
show *insertion* (α *x*) (*map-mpoly complex-of-int* (*trans n*)) = *?cn*
proof (*cases ?find n*)
case *None*
with *non-int-find*[*OF n*] **have** *ints*: *?cn* ∈ ℤ **by** *auto*
from *None* **show** *?thesis* **unfolding** *trans-def* **using** *ints*
by (*simp add*: *of-int-hom.map-mpoly-hom-Const of-int-to-int*)
next
case (*Some triple*)
from *find-Some-D*[*OF this*] *this* **obtain** *j f*
where *mem*: (*j*,*?cn*,*f*) ∈ *set triples* **and** *Some*: *?find n* = *Some* (*j*,*?cn*,*f*)
by (*cases triple*, *auto*)
from *mem* **have** *triples* ! *j* = (*j*,*?cn*,*f*) **unfolding** *triples set-zip* **by** *auto*
thus *?thesis* **unfolding** *trans-def Some* **by** (*simp add*: *map-mpoly-Var* α*-def*)
qed
qed
from *root* **have** *insertion* (α *x*) *?mp* = *0* **unfolding** *ins* **by** *auto*
hence *mpoly-polys-solution ?mp ?qs* (*set is*) (α *x*)
unfolding *mpoly-polys-solution-def*
proof (*standard*, *intro ballI*)
fix *j*
assume *j*: *j* ∈ *set is*
from *is-repr*[*OF this*]
show *poly* (*?qs j*) (α *x* (*Suc j*)) = *0* **unfolding** α *qs*[*OF j-len*[*OF j*]] *o-def* **by** *auto*
qed
note *eliminate* = *eliminate*[*OF this*, *unfolded* α *eliminate-aux-vars-of-int-poly*]
thus *eliminate-aux-vars mp qs is represents x* **by** *auto*
qed

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 −
obtain *mp triples* **where** *init*: *initial-root-problem p* = (*mp*, *triples*) **by** *force*

```
  from get-representative-complex[OF p algebraic init refl - root]
  show ?thesis unfolding representative-poly-def init Let-def by auto
qed
```

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 −
  let ?c = of-real :: real ⇒ complex
  let ?cp = map-poly ?c
  let ?p = ?cp p :: complex poly
  define cn where cn = degree ?p
  define n where n = degree p
  have n: cn = n unfolding n-def cn-def by simp
  note def = initial-root-problem-def[of ?p]
  note def = def[folded cn-def, unfolded n]
  define ccs where ccs = coeffs ?p
  define cs where cs = coeffs p
  have cs: ccs = map ?c cs
    unfolding ccs-def cs-def by auto
  note def = def[folded ccs-def]
  define crcs where crcs = remdups (filter (λc. c ∉ ℤ) ccs)
  define rcs where rcs = remdups (filter (λc. c ∉ ℤ) cs)
  have rcs: crcs = map ?c rcs
    unfolding crcs-def rcs-def cs by (induct cs, auto)
  define cpairs where cpairs = map (λc. (c, min-int-poly c)) crcs
  define pairs where pairs = map (λc. (c, min-int-poly c)) rcs
  have pairs: cpairs = map (map-prod ?c id) pairs
    unfolding pairs-def cpairs-def rcs by auto
  define cspairs where cspairs = sort-key (λ(c, y). degree y) cpairs
  define spairs where spairs = sort-key (λ(c, y). degree y) pairs
  have spairs: cspairs = map (map-prod ?c id) spairs
    unfolding spairs-def cspairs-def pairs
    by (rule sym, rule map-sort-key, auto)
  define ctriples where ctriples = zip [0..<length cspairs] cspairs
  define triples where triples = zip [0..<length spairs] spairs
  have triples: ctriples = map (map-prod id (map-prod ?c id)) triples
    unfolding ctriples-def triples-def spairs by (rule nth-equalityI, auto)
 note def = def[unfolded Let-def, folded crcs-def, folded cpairs-def, folded cspairs-def,
folded ctriples-def,
     unfolded of-real-hom.coeff-map-poly-hom]
  note def2 = initial-root-problem-def[of p, unfolded Let-def, folded n-def cs-def,
folded rcs-def, folded pairs-def,
     folded spairs-def, folded triples-def]
```

show *initial-root-problem ?p* = *map-prod id* (*map* (*map-prod id* (*map-prod ?c id*))) (*initial-root-problem p*)
unfolding *def def2 triples to-int-complex-of-real*
by (*simp*, *intro sum.cong refl arg-cong*[*of* - - λ *x*. - ∗ *x*], *induct triples*, *auto*)
qed

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 −
obtain *mp triples* **where** *init*: *initial-root-problem p* = (*mp*, *triples*) **by** *force*
define *is* **where** *is* = *map fst triples*
define *qs* **where** *qs* = (λ *j*. *snd* (*snd* (*triples* ! *j*)))
let *?c* = *of-real* :: *real* ⇒ *complex*
let *?cp* = *map-poly ?c*
let *?ct* = *map* (*map-prod id* (*map-prod ?c id*))
let *?p* = *?cp p* :: *complex poly*
have *p*: *?p* ≠ *0* **using** *p* **by** *auto*
have *initial-root-problem ?p* = *map-prod id ?ct* (*initial-root-problem p*)
by (*rule initial-root-problem-complex-of-real-poly*)
from *this*[*unfolded init*]
have *res*: *initial-root-problem ?p* = (*mp*, *?ct triples*)
by *auto*
from *root* **have** *0* = *?c* (*poly p x*) **by** *simp*
also have . . . = *poly ?p* (*?c x*) **by** *simp*
finally have *root*: *poly ?p* (*?c x*) = *0* **by** *simp*
have *qs*: *j* < *length is* =⇒ *qs j* = *snd* (*snd* (*?ct triples* ! *j*)) **for** *j*
unfolding *is-def qs-def* **by** (*auto simp*: *set-conv-nth*)
have *is*: *is* = *map fst* (*?ct triples*) **unfolding** *is-def* **by** *auto*
{
fix *cc*
assume *cc* ∈ *set* (*coeffs ?p*)
then obtain *c* **where** *c* ∈ *set* (*coeffs p*) **and** *cc*: *cc* = *?c c* **by** *auto*
from *algebraic this*(*1*) **have** *algebraic cc*
unfolding *cc algebraic-complex-iff* **by** *auto*
}
hence *algebraic*: *Ball* (*set* (*coeffs ?p*)) *algebraic* **..**
from *get-representative-complex*[*OF p this res is qs root*]
have *eliminate-aux-vars mp qs is represents ?c x* **.**
hence *eliminate-aux-vars mp qs is represents x* **by** *simp*
thus *?thesis* **unfolding** *representative-poly-def res init split Let-def qs-def is-def*
.
qed

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 −
  from *fundamental-theorem-of-algebra*[*OF nc*] **obtain** *z* **where**
    root: *poly p z* = *0* **by** *auto*
  from *algebraic-representsI*[*OF representative-poly-complex*[*OF - alg root*]] *nc root*
  have *algebraic z* ∧ *poly p z* = *0*
    using *constant-degree degree-0* **by** *blast*
  thus *?thesis* **..**
qed

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*
  by (*intro ext*, *simp*)+

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*
by (*intro ext*, *simp*)+

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* $\in_c$ *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
  unfolding zero-interval-def by auto

lemma zero-in-complex-interval-0 [simp, intro]: 0 ∈c 0
  unfolding zero-complex-interval-def by (auto simp: in-complex-interval-def)

lemma length-coeffs-degree':
  length (coeffs p) = (if p = 0 then 0 else Suc (degree p))
  by (cases p = 0) (auto simp: length-coeffs-degree)

lemma poly-in-poly-interval-complex:
  assumes list-all2 (λc ivl. c ∈c ivl) (coeffs p) cs x ∈c ivl
  shows   poly p x ∈c poly-interval cs ivl
proof −
  have len-eq: length (coeffs p) = length cs
    using assms(1) list-all2-lengthD by blast
  have coeffs p = map (λi. coeffs p ! i) [0..<length cs]
    by (subst len-eq [symmetric], rule map-nth [symmetric])
  also have ... = map (poly.coeff p) [0..<length cs]
    by (intro map-cong) (auto simp: nth-coeffs-coeff len-eq)
  finally have list-all2 (λc ivl. c ∈c ivl) (map (poly.coeff p) [0..<length cs]) cs
    using assms by simp
  moreover have length cs ≥ length (coeffs p)
    using len-eq by simp
  ultimately show ?thesis using assms(2)
  proof (induction cs ivl arbitrary: p x rule: poly-interval.induct)
    case (1 ivl p x)
    thus ?case by auto
  next
    case (2 c ivl p x)
    have degree p = 0
      using 2 by (auto simp: degree-eq-length-coeffs)
    then obtain c' where [simp]: p = [:c':]
      by (meson degree-eq-zeroE)
    show ?case using 2 by auto
  next
    case (3 c1 c2 cs ivl p x)
    obtain q c where [simp]: p = pCons c q
      by (cases p rule: pCons-cases)
    have list-all2 in-complex-interval (map (poly.coeff p) [0..<length (c1 # c2 #
cs)])
```

```
              (c1 # c2 # cs)
      using 3.prems(1) by simp
    also have [0..<length (c1 # c2 # cs)] = 0 # map Suc [0..<length (c2 # cs)]
      by (metis length-Cons map-Suc-upt upt-conv-Cons zero-less-Suc)
    also have map (poly.coeff p) ... = c # map (poly.coeff q) [0..<length (c2 #
cs)]
      by auto
    finally have c ∈c c1 and
        list-all2 in-complex-interval (map (poly.coeff q) [0..<length (c2 # cs)]) (c2
# cs)
      using 3.prems by (simp-all del: upt-Suc)

    have IH: poly q x ∈c poly-interval (c2 # cs) ivl
    proof (rule 3.IH)
      show length (coeffs q) ≤ length (c2 # cs)
        using 3.prems(2) unfolding length-coeffs-degree' by auto
    qed fact+

    show ?case
      using IH 3.prems ‹c ∈c c1›
      by (auto intro!: plus-complex-interval times-complex-interval)
  qed
qed

lemma poly-in-poly-interval-real: fixes x :: real
  assumes list-all2 (λc ivl. c ∈i ivl) (coeffs p) cs x ∈i ivl
  shows   poly p x ∈i poly-interval cs ivl
proof -
  have len-eq: length (coeffs p) = length cs
    using assms(1) list-all2-lengthD by blast
  have coeffs p = map (λi. coeffs p ! i) [0..<length cs]
    by (subst len-eq [symmetric], rule map-nth [symmetric])
  also have ... = map (poly.coeff p) [0..<length cs]
    by (intro map-cong) (auto simp: nth-coeffs-coeff len-eq)
  finally have list-all2 (λc ivl. c ∈i ivl) (map (poly.coeff p) [0..<length cs]) cs
    using assms by simp
  moreover have length cs ≥ length (coeffs p)
    using len-eq by simp
  ultimately show ?thesis using assms(2)
  proof (induction cs ivl arbitrary: p x rule: poly-interval.induct)
    case (1 ivl p x)
    thus ?case by auto
  next
    case (2 c ivl p x)
    have degree p = 0
      using 2 by (auto simp: degree-eq-length-coeffs)
    then obtain c' where [simp]: p = [:c':]
      by (meson degree-eq-zeroE)
    show ?case using 2 by auto
```

```
  next
    case (3 c1 c2 cs ivl p x)
    obtain q c where [simp]: p = pCons c q
      by (cases p rule: pCons-cases)
    have list-all2 in-interval (map (poly.coeff p) [0..<length (c1 # c2 # cs)])
                (c1 # c2 # cs)
      using 3.prems(1) by simp
    also have [0..<length (c1 # c2 # cs)] = 0 # map Suc [0..<length (c2 # cs)]
      by (metis length-Cons map-Suc-upt upt-conv-Cons zero-less-Suc)
    also have map (poly.coeff p) ... = c # map (poly.coeff q) [0..<length (c2 #
cs)]
      by auto
    finally have c ∈i c1 and
        list-all2 in-interval (map (poly.coeff q) [0..<length (c2 # cs)]) (c2 # cs)
      using 3.prems by (simp-all del: upt-Suc)

    have IH: poly q x ∈i poly-interval (c2 # cs) ivl
    proof (rule 3.IH)
      show length (coeffs q) ≤ length (c2 # cs)
        using 3.prems(2) unfolding length-coeffs-degree' by auto
    qed fact+

    show ?case
      using IH 3.prems ‹c ∈i c1›
      by (auto intro!: plus-in-interval times-in-interval)
  qed
qed


lemma in-interval-of-real [simp, intro]: x ∈i interval-of-real prec x
  unfolding interval-of-real-def by (auto simp: Let-def of-rat-rat field-simps)

lemma in-interval-of-complex [simp, intro]: z ∈c interval-of-complex prec z
  unfolding interval-of-complex-def in-complex-interval-def by auto

lemma distinct-genuine-roots [simp, intro]:
  distinct xs ⟹ distinct (genuine-roots p xs)
  by (simp add: genuine-roots-def)

definition filter-fun :: 'a poly ⇒ (nat ⇒ 'a :: comm-ring ⇒ bool) ⇒ bool where
  filter-fun p f = (∀ n x. poly p x = 0 ⟶ f n x)

lemma filter-fun-complex: filter-fun p (filter-fun-complex p)
  unfolding filter-fun-def
proof (intro impI allI)
  fix prec x
  assume root: poly p x = 0
  define cs where cs = map (interval-of-complex prec) (coeffs p)
  have cs: list-all2 in-complex-interval (coeffs p) cs
```

```
    unfolding cs-def list-all2-map2 by (intro list-all2-refl in-interval-of-complex)
  define P where P = (λx. 0 ∈c poly-interval cs (interval-of-complex prec x))
  have P x
  proof −
    have poly p x ∈c poly-interval cs (interval-of-complex prec x)
      by (intro poly-in-poly-interval-complex in-interval-of-complex cs)
    with root show ?thesis
      by (simp add: P-def)
  qed
  thus filter-fun-complex p prec x unfolding filter-fun-complex-def Let-def P-def
    using cs-def by blast
qed

lemma filter-fun-real: filter-fun p (filter-fun-real p)
  unfolding filter-fun-def
proof (intro impI allI)
  fix prec x
  assume root: poly p x = 0
  define cs where cs = map (interval-of-real prec) (coeffs p)
  have cs: list-all2 in-interval (coeffs p) cs
    unfolding cs-def list-all2-map2 by (intro list-all2-refl in-interval-of-real)
  define P where P = (λx. 0 ∈i poly-interval cs (interval-of-real prec x))
  have P x
  proof −
    have poly p x ∈i poly-interval cs (interval-of-real prec x)
      by (intro poly-in-poly-interval-real in-interval-of-real cs)
    with root show ?thesis
      by (simp add: P-def)
  qed
  thus filter-fun-real p prec x unfolding filter-fun-real-def Let-def P-def
    using cs-def by blast
qed

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)
  unfolding genuine-roots-def filter-filter
  using ff[unfolded filter-fun-def, rule-format, of - prec] by metis

lemma genuine-roots-step-preserve-invar:
  assumes {z. poly p z = 0} ⊆ set xs
  shows   {z. poly p z = 0} ⊆ set (filter (f prec) xs)
proof −
  have {z. poly p z = 0} = set (genuine-roots p xs)
    using assms by (auto simp: genuine-roots-def)
```

also have ... = *set* (*genuine-roots p* (*filter* (*f prec*) *xs*))
  using *genuine-roots-step*[*of - prec*] **by** *simp*
also have ... ⊆ *set* (*filter* (*f prec*) *xs*)
  by (*auto simp*: *genuine-roots-def*)
finally show *?thesis* **.**
qed
end

lemma *genuine-roots-finish*:
  fixes *p* :: *'a* :: *field-char-0 poly*
  assumes {*z*. *poly p z* = *0*} ⊆ *set xs distinct xs*
  assumes *length xs* = *card* {*z*. *poly p z* = *0*}
  shows *genuine-roots p xs* = *xs*
proof −
  have [*simp*]: *p* ≠ *0*
    using *finite-subset*[*OF assms*(*1*) *finite-set*] *infinite-UNIV-char-0* **by** *auto*
  have *length* (*genuine-roots p xs*) = *length xs*
    unfolding *genuine-roots-def* **using** *assms*
    by (*simp add*: *Int-absorb2 distinct-length-filter*)
  thus *?thesis*
    unfolding *genuine-roots-def*
    by (*metis filter-True length-filter-less linorder-not-less order-eq-iff*)
qed

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-char-0 poly*, *xs*, *n*, *ff*).
    distinct xs ∧
    {*z*. *poly p z* = *0*} ⊆ *set xs* ∧
    card {*z*. *poly p z* = *0*} = *n* ∧
    filter-fun p ff}
  by (*rule exI*[*of -* (*1*, [], *0*, λ - -. *False*)], *auto simp*: *filter-fun-def*)

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

lift-definition *genuine-roots'* :: *nat* ⇒ *'a* :: *field-char-0 genuine-roots-aux* ⇒ *'a list* **is**
  λ*prec* (*p*, *xs*, *n*, *ff*). *genuine-roots p xs* **.**

lift-definition *genuine-roots-impl-step'* :: *nat* ⇒ *'a* :: *field-char-0 genuine-roots-aux* ⇒ *'a genuine-roots-aux* **is**
  λ*prec* (*p*, *xs*, *n*, *ff*). (*p*, *filter* (*ff prec*) *xs*, *n*, *ff*)
  by (*safe*, *intro distinct-filter*, *auto simp*: *filter-fun-def*)

lift-definition *gr-poly* :: *'a* :: *field-char-0 genuine-roots-aux* ⇒ *'a poly* **is**
  λ(*p* :: *'a poly*, -, -, -). *p* **.**

lift-definition *gr-list* :: *'a* :: *field-char-0 genuine-roots-aux* ⇒ *'a list* **is**
λ(-, *xs* :: *'a list*, -, -). *xs* **.**

lift-definition *gr-numroots* :: *'a* :: *field-char-0 genuine-roots-aux* ⇒ *nat* **is**
λ(-, -, *n*, -). *n* **.**

lemma *genuine-roots'-code* [*code*]:
genuine-roots' prec gr =
(*if length* (*gr-list gr*) = *gr-numroots gr then gr-list gr*
else genuine-roots' (*2* ∗ *prec*) (*genuine-roots-impl-step' prec gr*))
proof (*transfer*, *clarify*)
fix *prec* :: *nat* **and** *p* :: *'a poly* **and** *xs* :: *'a list* **and** *ff*
assume ∗: {*z*. *poly p z* = *0*} ⊆ *set xs distinct xs filter-fun p ff*
show *genuine-roots p xs* =
(*if length xs* = *card* {*z*. *poly p z* = *0*} *then xs*
else genuine-roots p (*filter* (*ff prec*) *xs*))
using *genuine-roots-finish*[*of p xs*] *genuine-roots-step*[*of p*] ∗ **by** *auto*
qed

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*)
unfolding *genuine-roots-impl-def*
by (*transfer*, *auto simp*: *genuine-roots-def*, *transfer*, *auto*)

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*}
  by (*intro exI*[*of* - *1*], *auto simp*: *rsquarefree-def*)

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* **.**

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*)
  by *simp*

lemma *roots-of-poly-dummy*: **assumes** *p*: *p* ≠ *0*
  shows *set* (*roots-of-poly-dummy p*) = {*x*. *poly p x* = *0*} *distinct* (*roots-of-poly-dummy p*)
proof −
  from *someI-ex*[*OF finite-distinct-list*[*OF poly-roots-finite*[*OF p*]], *folded roots-of-poly-dummy-def*]
  show *set* (*roots-of-poly-dummy p*) = {*x*. *poly p x* = *0*} *distinct* (*roots-of-poly-dummy p*) **by** *auto*
qed

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*)
  subgoal for *p*
  proof −
    assume *rp*: *rsquarefree p*
    hence *card*: *card* {*x*. *poly p x* = *0*} = *Polynomial.degree p*
      using *rsquarefree-card-degree rsquarefree-def* **by** *blast*
    from *rp* **have** *p*: *p* ≠ *0* **unfolding** *rsquarefree-def* **by** *auto*
    have *ff*: *filter-fun p* (*filter-fun-complex p*) **by** (*rule filter-fun-complex*)
    show *?thesis*
    proof (*cases Ball* (*set* (*Polynomial.coeffs p*)) *algebraic*)
      case *False*
      with *roots-of-poly-dummy*[*OF p*] *ff*

show *?thesis* **using** *rp card* **by** *auto*
next
case *True*
from *rp card representative-poly-complex*[*of p*]
complex-roots-of-int-poly[*of representative-poly p*] *ff*
show *?thesis* **unfolding** *Let-def rsquarefree-def* **using** *True* **by** *auto*
qed
qed
done

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*)
subgoal for *p*
proof −
assume *rp*: *rsquarefree p*
from *rp* **have** *p*: $p \neq 0$ **unfolding** *rsquarefree-def* **by** *auto*
have *ff*: *filter-fun p* (*filter-fun-real p*) **by** (*rule filter-fun-real*)
show *?thesis*
proof (*cases Ball* (*set* (*Polynomial.coeffs p*)) *algebraic*)
case *False*
with *roots-of-poly-dummy*[*OF p*] *ff*
show *?thesis* **using** *rp* **by** (*auto simp*: *Let-def count-roots-correct*)
next
case *True*
from *rp representative-poly-real*[*of p*]
real-roots-of-int-poly[*of representative-poly p*] *ff*
show *?thesis* **unfolding** *Let-def rsquarefree-def* **using** *True*
by (*auto simp*: *count-roots-correct*)
qed
qed
done

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*)
unfolding *roots-of-complex-rf-poly-def genuine-roots-impl*
by (*transfer*, *auto simp*: *genuine-roots-impl*)

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*)
unfolding *roots-of-real-rf-poly-def genuine-roots-impl*
by (*transfer*, *auto simp*: *genuine-roots-impl Let-def*)

typedef (**overloaded**) *'a rf-polys* = { (*a* :: *'a* :: *idom*, *ps* :: (*'a poly* × *nat*) *list*). *Ball* (*fst* ‘ *set ps*) *rsquarefree*}
by (*intro exI*[*of* - (-,*Nil*)], *auto*)

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*
subgoal for *p*
apply *auto*
apply (*intro square-free-rsquarefree*)
apply (*insert yun-factorization*[*of p*, *OF refl*])
by (*cases yun-factorization gcd p*, *auto dest*: *square-free-factorizationD*)
done

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*
by (*auto simp*: *list-all-iff*, *force*)
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*)
using *assms* **unfolding** *polys-rf-def o-def*
apply *transfer*
subgoal for *p x*
proof −
assume *p*: *p* ≠ *0*
obtain *c ps* **where** *yun*: *yun-factorization gcd p* = (*c*,*ps*) **by** *force*
from *yun-factorization*[*OF this*] **have** *sff*: *square-free-factorization p* (*c*, *ps*) **by** *auto*
from *square-free-factorizationD'*(*1*)[*OF sff*] *p* **have** *c0*: *c* ≠ *0* **by** *auto*
show *?thesis* **unfolding** *yun*
unfolding *square-free-factorizationD'*(*1*)[*OF sff*] *poly-smult poly-prod-list*

snd-conv
mult-eq-0-iff prod-list-zero-iff
using *c0 square-free-factorizationD*(*2*)[*OF sff*] **by** *force*
qed
done

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* }
unfolding *roots-of-complex-rf-polys-def set-concat set-map image-comp o-def roots-of-complex-rf-poly* **by** *auto*

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* }
unfolding *roots-of-real-rf-polys-def set-concat set-map image-comp o-def roots-of-real-rf-poly* **by** *auto*

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*}
using *p* **unfolding** *roots-of-complex-poly-def*
by (*simp add*: *roots-of-complex-rf-polys yun-polys*[*OF p*])

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* ≠ *0*
shows *set* (*roots-of-real-poly p*) = {*x*. *poly p x* = *0*}
using *p* **unfolding** *roots-of-real-poly-def*
by (*simp add*: *roots-of-real-rf-polys yun-polys*[*OF p*])

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*)
by (*induct xs*, *auto split*: *if-splits*)

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*))))
  using *assms* **unfolding** *polys-rf-def*
proof (*transfer*, *goal-cases*)
  case (*1 rop p*)
  obtain *c fs* **where** *yun*: *yun-factorization gcd p* = (*c*,*fs*) **by** *force*
  note *sff* = *yun-factorization*(*1*)[*OF yun*]
  note *sff1* = *square-free-factorizationD*[*OF sff*]
  note *sff2* = *square-free-factorizationD'*[*OF sff*]
  have *rs*: (*p*,*i*) ∈ *set fs* ⟹ *rsquarefree p* **for** *p i*
    by (*intro square-free-rsquarefree*, *insert sff1*(*2*), *auto*)
  note *1* = *1*[*OF rs*]
  show *?case* **unfolding** *yun snd-conv map-map o-def* **using** *1 sff1*(*3*,*5*)
  proof (*induct fs*)
    case (*Cons pi fs*)
    obtain *p i* **where** *pi*: *pi* = (*p*,*i*) **by** *force*
    hence (*p*,*i*) ∈ *set* (*pi* # *fs*) **by** *auto*
    note *p-i* = *Cons*(*2*−*4*)[*OF this*]
    have *IH*: *distinct* (*concat* (*map* (λ*x*. *rop* (*fst x*)) *fs*))
      by (*rule Cons*(*1*)[*OF Cons*(*2*,*3*,*4*)], *insert Cons*(*5*), *auto*)
    {
      fix *x*
      assume *x*: *x* ∈ *set* (*rop p*) *x* ∈ (⋃*x*∈*set fs*. *set* (*rop* (*fst x*)))
      from *x*[*unfolded p-i*] **have** *rtp*: *poly p x* = *0* **by** *auto*
      from *x* **obtain** *q j* **where** *qj*: (*q*,*j*) ∈ *set fs* **and** *x*: *x* ∈ *set* (*rop q*) **by** *force*
      from *Cons*(*2*)[*of q j*] *x qj* **have** *rtq*: *poly q x* = *0* **by** *auto*
      from *Cons*(*5*)[*unfolded pi*] *qj* **have** (*p*,*i*) ≠ (*q*,*j*) **by** *auto*
      from *p-i*(*3*)[*OF - this*] *qj* **have** *cop*: *algebraic-semidom-class.coprime p q* **by**
auto
      from *rtp* **have** *dvdp*: [:−*x*,*1*:] *dvd p* **using** *poly-eq-0-iff-dvd* **by** *blast*
      from *rtq* **have** *dvdq*: [:−*x*,*1*:] *dvd q* **using** *poly-eq-0-iff-dvd* **by** *blast*
      from *cop dvdp dvdq* **have** *is-unit* [:−*x*,*1*:] **by** (*metis coprime-common-divisor*)
      hence *False* **by** *simp*
    }
    thus *?case* **unfolding** *pi* **by** (*auto simp*: *p-i*(*2*) *IH*)
  qed *simp*
qed

lemma *distinct-roots-of-real-poly*: *distinct* (*roots-of-real-poly p*)
  unfolding *roots-of-real-poly-def roots-of-real-rf-polys-def*
  using *roots-of-rf-yun-polys-distinct*[*of roots-of-real-rf-poly p*, *OF roots-of-real-rf-poly*]
  by *auto*

lemma *distinct-roots-of-complex-poly*: *distinct* (*roots-of-complex-poly p*)
  unfolding *roots-of-complex-poly-def roots-of-complex-rf-polys-def*
  using *roots-of-rf-yun-polys-distinct*[*of roots-of-complex-rf-poly p*, *OF roots-of-complex-rf-poly*]

by *auto*

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 (*cases p* = *0*)
  case *True*
  have [*simp*]: *yun-rf* (*yun-polys 0*) = (*0*,[])
    by (*transfer*, *simp*)
  show *?thesis*
    unfolding *factor-complex-main-def Let-def roots-of-complex-poly-def True*
    by *simp*
next
  case *False*
  hence *p*: (*p* = *0*) = *False* **by** *simp*
  obtain *c rts* **where** *yun*: *yun-rf* (*yun-polys p*) = (*c*,*rts*) **by** *force*
  show *?thesis*
    unfolding *factor-complex-main-def Let-def roots-of-complex-poly-def p if-False*
      roots-of-complex-rf-polys-def polys-rf-def o-def yun split snd-conv map-map
    by (*induct rts*, *auto simp*: *o-def*)
qed

lemma *distinct-factor-complex-main*:
  distinct (*map fst* (*snd* (*factor-complex-main p*)))
  unfolding *roots-of-complex-poly-via-factor-complex-main*
  by (*rule distinct-roots-of-complex-poly*)

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

obtain *d pis* **where** *yun*: *yun-factorization gcd p* = (*d*,*pis*) **by** *force*
obtain *d′ pis′* **where** *yun-rf*: *yun-rf* (*yun-polys p*) = (*d′*,*pis′*) **by** *force*
let *?p* = *poly-rf*
let *?map* = *map* (λ (*p*,*i*). (*?p p*, *i*))
from *yun yun-rf* **have** *d′*: *d′* = *d* **and** *pis*: *pis* = *?map pis′*
  by (*atomize*(*full*), *transfer*, *auto*)
from *rt*[*unfolded factor-complex-main-def yun-rf split Let-def d′*]
have *xis*: *xis* = *concat* (*map* (λ(*p*, *i*). *map* (λ*r*. (*r*, *i*)) (*roots-of-complex-rf-poly p*)) *pis′*)
  and *d*: *d* = *c*
  by (*auto split*: *if-splits*)
note *yun* = *yun-factorization*[*OF yun*[*unfolded d*]]
note *yun* = *square-free-factorizationD*[*OF yun*(*1*)] *yun*(*2*)[*unfolded snd-conv*]
let *?exp* = λ *pis*. ∏(*x*, *i*)←*concat*
  (*map* (λ(*p*, *i*). *map* (λ*r*. (*r*, *i*)) (*roots-of-complex-rf-poly p*)) *pis*). [:− *x*, *1*:] ^ *i*
from *yun*(*1*) **have** *p*: *p* = *smult c* (∏(*a*, *i*)∈*set pis*. *a* ^ *i*) **.**
also have (∏(*a*, *i*)∈*set pis*. *a* ^ *i*) = (∏(*a*, *i*)←*pis*. *a* ^ *i*)
  by (*rule prod.distinct-set-conv-list*[*OF yun*(*5*)])
also have . . . = *?exp pis′* **using** *yun*(*2*,*6*) **unfolding** *pis*
proof (*induct pis′*)
  case (*Cons pi pis*)
  obtain *p i* **where** *pi*: *pi* = (*p*,*i*) **by** *force*
  let *?rts* = *roots-of-complex-rf-poly p*
  note *Cons* = *Cons*[*unfolded pi*]
  have *IH*: (∏(*a*, *i*)←*?map pis*. *a* ^ *i*) = (*?exp pis*)
    by (*rule Cons*(*1*)[*OF Cons*(*2*−*3*)], *auto*)
  from *Cons*(*2*−*3*)[*of ?p p i*] **have** *p*: *square-free* (*?p p*) *degree* (*?p p*) ≠ *0 ?p p* ≠ *0 monic* (*?p p*) **by** *auto*
  have (∏(*a*, *i*)←*?map* (*pi* # *pis*). *a* ^ *i*) = *?p p* ^ *i* ∗ (∏(*a*, *i*)←*?map pis*. *a* ^ *i*)
    unfolding *pi* **by** *simp*
  also have (∏(*a*, *i*)←*?map pis*. *a* ^ *i*) = *?exp pis* **by** (*rule IH*)
  finally have *id*: (∏(*a*, *i*)←*?map* (*pi* # *pis*). *a* ^ *i*) = *?p p* ^ *i* ∗ *?exp pis* **by** *simp*
  have *?exp* (*pi* # *pis*) = *?exp* [(*p*,*i*)] ∗ *?exp pis* **unfolding** *pi* **by** *simp*
  also have *?exp* [(*p*,*i*)] = (∏(*x*, *i*)← (*map* (λ*r*. (*r*, *i*)) *?rts*). [:− *x*, *1*:] ^ *i*)
    by *simp*
  also have . . . = (∏ *x* ← *?rts*. [:− *x*, *1*:])^ *i*
    unfolding *prod-list-power* **by** (*rule arg-cong*[*of - - prod-list*], *auto*)
  also have (∏ *x* ← *?rts*. [:− *x*, *1*:]) = *?p p*
  proof −
    from *fundamental-theorem-algebra-factorized*[*of ?p p*, *unfolded* ‹*monic* (*?p p*)›]
    obtain *as* **where** *as*: *?p p* = (∏*a*←*as*. [:− *a*, *1*:]) **by** (*metis smult-1-left*)
    also have . . . = (∏*a*∈*set as*. [:− *a*, *1*:])
    proof (*rule sym*, *rule prod.distinct-set-conv-list*, *rule ccontr*)
      assume ¬ *distinct as*
      from *not-distinct-decomp*[*OF this*] **obtain** *as1 as2 as3 a* **where**
        a: *as* = *as1* @ [*a*] @ *as2* @ [*a*] @ *as3* **by** *blast*

```
      define q where q = (∏ a←as1 @ as2 @ as3. [:− a, 1:])
      have ?p p = (∏ a←as. [:− a, 1:]) by fact
      also have ... = (∏ a←([a] @ [a]). [:− a, 1:]) ∗ q
        unfolding q-def a map-append prod-list.append by (simp only: ac-simps)
      also have ... = [:−a,1:] ∗ [:−a,1:] ∗ q by simp
      finally have ?p p = ([:−a,1:] ∗ [:−a,1:]) ∗ q by simp
      hence [:−a,1:] ∗ [:−a,1:] dvd ?p p unfolding dvd-def ..
    with ‹square-free (?p p)›[unfolded square-free-def, THEN conjunct2, rule-format,
of [:−a,1:]]
      show False by auto
    qed
    also have set as = {x. poly (?p p) x = 0} unfolding as poly-prod-list
      by (simp add: o-def, induct as, auto)
    also have ... = set ?rts by (simp add: roots-of-complex-rf-poly(1))
    also have (∏ a∈set ?rts. [:− a, 1:]) = (∏ a←?rts. [:− a, 1:])
      by (rule prod.distinct-set-conv-list[OF roots-of-complex-rf-poly(2)])
    finally show ?thesis by simp
  qed
  finally have id2: ?exp (pi # pis) = ?p p ^ i ∗ ?exp pis by simp
  show ?case unfolding id id2 ..
 qed simp
 also have ?exp pis′ = (∏ (x, i)←xis. [:− x, 1:] ^ i) unfolding xis ..
 finally show p = smult c (∏ (x, i)←xis. [:− x, 1:] ^ i) unfolding p xis by simp

 from yun(2) have 0 ∉ snd ‘ set pis by force
 with pis have 0 ∉ snd ‘ set pis′ by force
 thus 0 ∉ snd ‘ set xis unfolding xis by force
qed

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 −
 obtain c ris where main: factor-complex-main p = (c,ris) by force
 show ?thesis unfolding factor-complex-poly-def main split
  using distinct-factor-complex-main[of p, unfolded main]
  unfolding snd-conv o-def
  unfolding distinct-map by (force simp: inj-on-def)
qed


theorem factor-complex-poly: assumes fp: factor-complex-poly p = (c,qis)
 shows
 p = smult c (∏ (q, i)←qis. q ^ i)
 (q,i) ∈ set qis ⟹ irreducible q ∧ i ≠ 0 ∧ monic q ∧ degree q = 1
```

```
proof –
  from fp[unfolded factor-complex-poly-def]
  obtain pis where fp: factor-complex-main p = (c,pis)
    and qis: qis = map (λ(r, i). ([:− r, 1:], i)) pis
    by (cases factor-complex-main p, auto)
  from factor-complex-main[OF fp] have p: p = smult c (∏ (x, i)←pis. [:− x, 1:]
^ i) and 0: 0 ∉ snd ‘ set pis by auto
  show p = smult c (∏ (q, i)←qis. q ^ i) unfolding p qis
    by (rule arg-cong[of - - λ p. smult c (prod-list p)], auto)
  show (q,i) ∈ set qis ⟹ irreducible q ∧ i ≠ 0 ∧ monic q ∧ degree q = 1
    using linear-irreducible-field[of q] using 0 unfolding qis by force
qed

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
  by (induct x i yjs rule: delete-cnj.induct, auto)

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) = (if x ∈ ℝ then
    ([:−(Re x),1:],i) # complex-roots-to-real-factorization xs else
    let xx = cnj x; ys = delete-cnj xx i xs; p = map-poly Re ([:−x,1:] ∗ [:−xx,1:])
    in (p,i) # complex-roots-to-real-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* **by** *auto*

lemma *delete-cnj-0*: **assumes** *0* ∉ *snd* ‘ *set xis*
 shows *0* ∉ *snd* ‘ *set* (*delete-cnj x si xis*)
 using *assms* **by** (*induct x si xis rule*: *delete-cnj.induct*, *auto*)

lemma *delete-cnj*: **assumes**
 order x (∏ (*x*, *i*)←*xis*. [:− *x*, *1*:] ^ *i*) ≥ *si si* ≠ *0*
 shows (∏ (*x*, *i*)←*xis*. [:− *x*, *1*:] ^ *i*) =
 [:− *x*, *1*:] ^ *si* ∗ (∏ (*x*, *i*)←*delete-cnj x si xis*. [:− *x*, *1*:] ^ *i*)
using *assms*
proof (*induct x si xis rule*: *delete-cnj.induct*)
 case (*2 x si*)
 hence *order x 1* ≥ *1* **by** *auto*
 hence [:−*x*,*1*:]^*1 dvd 1* **unfolding** *order-divides* **by** *simp*
 from *power-le-dvd*[*OF this*, *of 1*] ‹*si* ≠ *0*› **have** [:− *x*, *1*:] *dvd 1* **by** *simp*
 from *divides-degree*[*OF this*]
 show *?case* **by** *auto*
next
 case (*1 x i y j yjs*)
 note *IH* = *1*(*1*−*2*)
 let *?yj* = [:−*y*,*1*:]^*j*
 let *?yjs* = (∏ (*x*,*i*)←*yjs*. [:− *x*, *1*:] ^ *i*)
 let *?x* = [: − *x*, *1* :]
 let *?xi* = *?x* ^ *i*
 have *monic* (∏ (*x*,*i*)←(*y*, *j*) # *yjs*. [:− *x*, *1*:] ^ *i*)
 by (*intro monic-prod-list*, *auto intro*: *monic-power*)
 then have *monic* (*?yj* ∗ *?yjs*) **by** *simp*
 from *monic-imp-nonzero*[*OF this*] **have** *yy0*: *?yj* ∗ *?yjs* ≠ *0* **by** *auto*
 have *id*: (∏ (*x*,*i*)←(*y*, *j*) # *yjs*. [:− *x*, *1*:] ^ *i*) = *?yj* ∗ *?yjs* **by** *simp*
 from *1*(*3*−) **have** *ord*: *i* ≤ *order x* (*?yj* ∗ *?yjs*) **and** *i*: *i* ≠ *0* **unfolding** *id* **by** *auto*
 from *ord*[*unfolded order-mult*[*OF yy0*]] **have** *ord*: *i* ≤ *order x ?yj* + *order x ?yjs* .
 from *this*[*unfolded order-linear-power*]
 have *ord*: *i* ≤ (*if y* = *x then j else 0*) + *order x ?yjs* **by** *simp*
 show *?case*
 proof (*cases x* = *y*)
  case *False*
  from *ord False* **have** *i* ≤ *order x ?yjs* **by** *simp*
  note *IH* = *IH*(*2*)[*OF False this i*]
  from *False* **have** *del*: *delete-cnj x i* ((*y*, *j*) # *yjs*) = (*y*,*j*) # *delete-cnj x i yjs* **by** *simp*
  show *?thesis* **unfolding** *del id IH*
   by (*simp add*: *ac-simps*)
 next
  case *True* **note** *xy* = *this*
  note *IH* = *IH*(*1*)[*OF True*]

show *?thesis*
proof (*cases* $j \geq i$)
case *False*
from *ord* **have** *ord*: $i - j \leq$ *order x ?yjs* **unfolding** *xy* **by** *simp*
have $?xi = ?x \,\hat{}\, (j + (i - j))$ **using** *False* **by** *simp*
also have $\ldots = ?x \,\hat{}\, j * ?x \,\hat{}\, (i - j)$
unfolding *power-add* **by** *simp*
finally have *xi*: $?xi = ?x \,\hat{}\, j * ?x \,\hat{}\, (i - j)$ **.**
from *False* **have** $j \neq i \neg i < j\ i - j \neq 0$ **by** *auto*
note *IH* = *IH*[*OF this*(*1*,*2*) *ord this*(*3*)]
from *xy False* **have** *del*: *delete-cnj x i* ((*y*, *j*) # *yjs*) = *delete-cnj x* ($i - j$) *yjs* **by** *auto*
show *?thesis* **unfolding** *del id* **unfolding** *IH xi* **unfolding** *xy* **by** *simp*
next
case *True*
hence $j = i \vee i < j$ **by** *auto*
thus *?thesis*
proof
assume *i*: $j = i$
from *xy i* **have** *del*: *delete-cnj x i* ((*y*, *j*) # *yjs*) = *yjs* **by** *simp*
show *?thesis* **unfolding** *id del* **unfolding** *xy i* **by** *simp*
next
assume *ij*: $i < j$
with *xy i* **have** *del*: *delete-cnj x i* ((*y*, *j*) # *yjs*) = (*y*, $j - i$) # *yjs* **by** *simp*
from *ij* **have** *idd*: $j = i + (j - i)$ **by** *simp*
show *?thesis*
apply (*unfold id del*)
apply (*subst idd*)
apply (*unfold power-add xy*)
by *simp*
qed
qed
qed
qed

theorem *factor-real-poly*: **assumes** *fp*: *factor-real-poly p* = (*c*,*qis*)
shows *p* = *smult c* ($\prod$(*q*, *i*)←*qis*. $q \,\hat{}\, i$)
(*q*,*j*) ∈ *set qis* ⟹ *irreducible q* ∧ $j \neq 0$ ∧ *monic q* ∧ *degree q* ∈ {*1*,*2*}
proof −
interpret *map-poly-inj-idom-hom of-real*..
have (*p* = *smult c* ($\prod$(*q*, *i*)←*qis*. $q \,\hat{}\, i$)) ∧ ((*q*,*j*) ∈ *set qis* ⟶ *irreducible q* ∧ $j \neq 0$ ∧ *monic q* ∧ *degree q* ∈ {*1*,*2*})
proof (*cases p* = *0*)
case *True*
have *yun*: *yun-rf* (*yun-polys* (*0* :: *complex poly*)) = (*0*,[])
by (*transfer*, *auto simp*: *yun-factorization-def*)
have *factor-real-poly p* = (*0*,[]) **unfolding** *True*
by (*simp add*: *factor-real-poly-def factor-complex-main-def yun*)

with *fp* **have** *id*: *c* = *0 qis* = [] **by** *auto*
thus *?thesis* **unfolding** *True* **by** *simp*
next
case *False* **note** *p0* = *this*
let *?c* = *complex-of-real*
let *?rp* = *map-poly Re*
let *?cp* = *map-poly ?c*
let *?p* = *?cp p*
from *fp*[*unfolded factor-real-poly-def*]
obtain *d xis* **where** *fp*: *factor-complex-main ?p* = (*d*,*xis*)
and *c*: *c* = *Re d* **and** *qis*: *qis* = *complex-roots-to-real-factorization xis*
by (*cases factor-complex-main ?p*, *auto*)
from *factor-complex-main*[*OF fp*] **have** *p*: *?p* = *smult d* (∏(*x*, *i*)←*xis*. [:− *x*, *1*:] ^ *i*)
(**is** - = *smult d ?q*) **and** *0*: *0* ∉ *snd* ' *set xis* **.**
from *arg-cong*[*OF this*(*1*), *of* λ *p*. *coeff p* (*degree p*)]
have *coeff ?p* (*degree ?p*) = *coeff* (*smult d ?q*) (*degree* (*smult d ?q*)) **.**
also have *coeff ?p* (*degree ?p*) = *?c* (*coeff p* (*degree p*)) **by** *simp*
also have *coeff* (*smult d ?q*) (*degree* (*smult d ?q*)) = *d* ∗ *coeff ?q* (*degree ?q*)
by *simp*
also have *monic ?q* **by** (*rule monic-prod-list*, *auto intro*: *monic-power*)
finally have *d*: *d* = *?c* (*coeff p* (*degree p*)) **by** *auto*
from *arg-cong*[*OF this*, *of Re*, *folded c*] **have** *c*: *c* = *coeff p* (*degree p*) **by** *auto*
have *set* (*coeffs ?p*) ⊆ ℝ **by** *auto*
with *p* **have** *q′*: *set* (*coeffs* (*smult d ?q*)) ⊆ ℝ **by** *auto*
from *d p0* **have** *d0*: *d* ≠ *0* **by** *auto*
have *smult d ?q* = [:*d*:] ∗ *?q* **by** *auto*
from *real-poly-factor*[*OF q′*[*unfolded this*]] *d0 d*
have *q*: *set* (*coeffs ?q*) ⊆ ℝ **by** *auto*
have *p* = *?rp ?p*
by (*rule sym*, *subst map-poly-map-poly*, *force*, *rule map-poly-idI*, *auto*)
also have . . . = *?rp* (*smult d ?q*) **unfolding** *p* **..**
also have *?q* = *?cp* (*?rp ?q*)
by (*rule sym*, *rule map-poly-of-real-Re*, *insert q*, *auto*)
also have *d* = *?c c* **unfolding** *d c* **..**
also have *smult* (*?c c*) (*?cp* (*?rp ?q*)) = *?cp* (*smult c* (*?rp ?q*)) **by** (*simp add*: *hom-distribs*)
also have *?rp* . . . = *smult c* (*?rp ?q*)
by (*subst map-poly-map-poly*, *force*, *rule map-poly-idI*, *auto*)
finally have *p*: *p* = *smult c* (*?rp ?q*) **.**
let *?fact* = *complex-roots-to-real-factorization*
have *?rp ?q* = (∏(*q*, *i*)←*qis*. *q* ^ *i*) ∧
((*q*, *j*) ∈ *set qis* ⟶ *irreducible q* ∧ *j* ≠ *0* ∧ *monic q* ∧ *degree q* ∈ {*1*, *2*})
using *q 0* **unfolding** *qis*
proof (*induct xis rule*: *complex-roots-to-real-factorization.induct*)
case *1*
show *?case* **by** *simp*
next
case (*2 x i xis*)

```
note IH = 2(1−2)
note prems = 2(3)
from 2(4) have i: i ≠ 0 and 0: 0 ∉ snd ‘ set xis by auto
let ?xi = [:− x, 1:] ^ i
let ?xis = (∏(x, i)←xis. [:− x, 1:] ^ i)
have id: (∏(x, i)←((x,i) # xis). [:− x, 1:] ^ i) = ?xi ∗ ?xis
  by simp
show ?case
proof (cases x ∈ ℝ)
  case True
  have xi: set (coeffs ?xi) ⊆ ℝ
    by (rule real-poly-power, insert True, auto)
  have xis: set (coeffs ?xis) ⊆ ℝ
 by (rule real-poly-factor[OF prems[unfolded id] xi], rule linear-power-nonzero)
  note IH = IH(1)[OF True xis 0]
  have ?rp (?xi ∗ ?xis) = ?rp ?xi ∗ ?rp ?xis
    by (rule map-poly-Re-mult[OF xi xis])
  also have ?rp ?xi = (?rp [: −x,1 :])^i
    by (rule map-poly-Re-power, insert True, auto)
  also have ?rp [: −x,1 :] = [:−(Re x),1:] by auto
  also have ?rp ?xis = (∏ (a,b) ← ?fact xis. a ^ b)
    using IH by auto
  also have [:− Re x, 1:] ^ i ∗ (∏ (a,b) ← ?fact xis. a ^ b) =
    (∏ (a,b) ← ?fact ((x,i) # xis). a ^ b) using True by simp
  finally have idd: ?rp (?xi ∗ ?xis) = (∏ (a,b) ← ?fact ((x,i) # xis). a ^ b)
.
  show ?thesis unfolding id idd
  proof (intro conjI, force, intro impI)
    assume (q, j) ∈ set (?fact ((x, i) # xis))
    hence (q,j) ∈ set (?fact xis) ∨ (q = [:− Re x, 1:] ∧ j = i)
      using True by auto
    thus irreducible q ∧ j ≠ 0 ∧ monic q ∧ degree q ∈ {1, 2}
    proof
      assume (q,j) ∈ set (?fact xis)
      with IH show ?thesis by auto
    next
      assume q = [:− Re x, 1:] ∧ j = i
      with linear-irreducible-field[of [:− Re x, 1:]] i show ?thesis by auto
    qed
  qed
next
  case False
  define xi where xi = [:Re x ∗ Re x + Im x ∗ Im x, − (2 ∗ Re x), 1:]
  obtain xx where xx: xx = cnj x by auto
  have xi: xi = ?rp ([:−x,1:] ∗ [:−xx,1:]) unfolding xx xi-def by auto
  have cpxi: ?cp xi = [:−x,1:] ∗ [:−xx,1:] unfolding xi-def
    by (cases x, auto simp: xx legacy-Complex-simps)
  obtain yis where yis: yis = delete-cnj xx i xis by auto
  from delete-cnj-0[OF 0] have 0: 0 ∉ snd ‘ set yis unfolding yis .
```

```
    from False have fact: ?fact ((x,i) # xis) = ((xi,i) # ?fact yis)
      unfolding xi-def xx yis by simp
    note IH = IH(2)[OF False xx yis xi - 0]
    have irreducible xi
      apply (fold irreducible-connect-field)
    proof (rule irreducible_d I)
      show degree xi > 0 unfolding xi by auto
      fix q p :: real poly
      assume degree q > 0 degree q < degree xi and qp: xi = q ∗ p
      hence dq: degree q = 1 unfolding xi by auto
      have dxi: degree xi = 2 xi ≠ 0 unfolding xi by auto
      with qp have q ≠ 0 p ≠ 0 by auto
      hence degree xi = degree q + degree p unfolding qp
        by (rule degree-mult-eq)
      with dq have dp: degree p = 1 unfolding dxi by simp
      {
        fix c :: complex
        assume rt: poly (?cp xi) c = 0
        hence poly (?cp q ∗ ?cp p) c = 0 by (simp add: qp hom-distribs)
        hence (poly (?cp q) c = 0 ∨ poly (?cp p) c = 0) by auto
        hence c = roots1 (?cp q) ∨ c = roots1 (?cp p)
          using roots1[of ?cp q] roots1[of ?cp p] dp dq by auto
        hence c ∈ ℝ unfolding roots1-def by auto
        hence c ≠ x using False by auto
      }
      hence poly (?cp xi) x ≠ 0 by auto
      thus False unfolding cpxi by simp
    qed
    hence xi': irreducible xi monic xi degree xi = 2
      unfolding xi by auto
    let ?xxi = [:− xx, 1:] ^ i
    let ?yis = (∏(x, i)←yis. [:− x, 1:] ^ i)
    let ?yi = (?cp xi)^ i
    have yi: set (coeffs ?yi) ⊆ ℝ
      by (rule real-poly-power, auto simp: xi)
    have mon: monic (∏(x, i)←(x, i) # xis. [:− x, 1:] ^ i)
      by (rule monic-prod-list, auto intro: monic-power)
    from monic-imp-nonzero[OF this] have xixis: ?xi ∗ ?xis ≠ 0 unfolding id
by auto
      from False have xxx: xx ≠ x unfolding xx by (cases x, auto simp:
legacy-Complex-simps Reals-def)
    from prems[unfolded id] have prems: set (coeffs (?xi ∗ ?xis)) ⊆ ℝ .
    from id have [:− x, 1:] ^ i dvd ?xi ∗ ?xis by auto
    from xixis this[unfolded order-divides]
    have order x (?xi ∗ ?xis) ≥ i by auto
    with complex-conjugate-order[OF prems xixis, of x, folded xx]
    have order xx (?xi ∗ ?xis) ≥ i by auto
    hence order xx ?xi + order xx ?xis ≥ i unfolding order-mult[OF xixis] .
     also have order xx ?xi = 0 unfolding order-linear-power using xxx by
```

simp

finally have *order xx ?xis* ≥ *i* **by** *simp*
hence *yis*: *?xis* = *?xxi* ∗ *?yis* **unfolding** *yis* **using** *i*
by (*intro delete-cnj*, *simp*)
hence *?xi* ∗ *?xis* = (*?xi* ∗ *?xxi*) ∗ *?yis* **by** (*simp only*: *ac-simps*)
also have *?xi* ∗ *?xxi* = ([:− *x*, *1*:] ∗ [:− *xx*, *1*:])^*i*
by (*metis power-mult-distrib*)
also have [:− *x*, *1*:] ∗ [:− *xx*, *1*:] = *?cp xi* **unfolding** *cpxi* **..**
finally have *idd*: *?xi* ∗ *?xis* = (*?cp xi*)^*i* ∗ *?yis* **by** *simp*
from *prems*[*unfolded idd*] **have** *R*: *set* (*coeffs* ((*?cp xi*)^*i* ∗ *?yis*)) ⊆ ℝ **.**
have *yis*: *set* (*coeffs ?yis*) ⊆ ℝ
by (*rule real-poly-factor*[*OF R yi*], *auto*, *auto simp*: *xi-def*)
note *IH* = *IH*[*OF yis*]
have *?rp* (*?xi* ∗ *?xis*) = *?rp ?yi* ∗ *?rp ?yis* **unfolding** *idd*
by (*rule map-poly-Re-mult*[*OF yi yis*])
also have *?rp ?yi* = *xi*^*i* **by** (*fold hom-distribs*, *rule map-poly-Re-of-real*)
also have *?rp ?yis* = (∏ (*a*,*b*) ← *?fact yis*. *a* ^ *b*)
using *IH* **by** *auto*
also have *xi* ^ *i* ∗ (∏ (*a*,*b*) ← *?fact yis*. *a* ^ *b*) =
(∏ (*a*,*b*) ← *?fact* ((*x*,*i*) # *xis*). *a* ^ *b*) **unfolding** *fact* **by** *simp*
finally have *idd*: *?rp* (*?xi* ∗ *?xis*) = (∏ (*a*,*b*) ← *?fact* ((*x*,*i*) # *xis*). *a* ^ *b*)
.
show *?thesis* **unfolding** *id idd fact* **using** *IH xi′ i* **by** *auto*
qed
qed
thus *?thesis* **unfolding** *p* **by** *simp*
qed
thus *p* = *smult c* (∏ (*q*, *i*)←*qis*. *q* ^ *i*)
(*q*,*j*) ∈ *set qis* ⟹ *irreducible q* ∧ *j* ≠ *0* ∧ *monic q* ∧ *degree q* ∈ {*1*,*2*} **by** *blast+*
qed

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.