The Hermite–Lindemann–WeierstraSS Transcendence Theorem

Manuel Eberl

December 14, 2021

Abstract

This article provides a formalisation of the Hermite–Lindemann–WeierstraSS Theorem (also known as simply Hermite–Lindemann or Lindemann–WeierstraSS). This theorem is one of the crowning achievements of 19th century number theory.

The theorem states that if $\alpha_1, \ldots, \alpha_n \in \mathbb{C}$ are algebraic numbers that are linearly independent over $\mathbb{Z}$, then $e^{\alpha_1}, \ldots, e^{\alpha_n}$ are algebraically independent over $\mathbb{Q}$.

Like the previous formalisation in Coq by Bernard [2], I proceeded by formalising Baker's alternative formulation of the theorem [1] and then deriving the original one from that. Baker's version states that for any algebraic numbers $\beta_1, \ldots, \beta_n \in \mathbb{C}$ and distinct algebraic numbers $\alpha_i, \ldots, \alpha_n \in \mathbb{C}$, we have:

$$\beta_1 e^{\alpha_1} + \ldots + \beta_n e^{\alpha_n} = 0 \qquad \text{iff} \qquad \forall i.\ \beta_i = 0$$

This has a number of immediate corollaries, e.g.:

- e and π are transcendental
- e^z, $\sin z$, $\tan z$, etc. are transcendental for algebraic $z \in \mathbb{C} \setminus \{0\}$
- $\ln z$ is transcendental for algebraic $z \in \mathbb{C} \setminus \{0, 1\}$

Contents

1 Divisibility of algebraic integers

theory *Algebraic-Integer-Divisibility*
  imports *Algebraic-Numbers.Algebraic-Numbers*
begin

In this section, we define a notion of divisibility of algebraic integers: y is divisible by x if $y\ /\ x$ is an algebraic integer (or if x and y are both zero).

Technically, the definition does not require x and y to be algebraic integers themselves, but we will always use it that way (in fact, in our case x will always be a rational integer).

definition *alg-dvd* :: *'a* :: *field* ⇒ *'a* ⇒ *bool* (**infix** *alg'-dvd 50*) **where**
  x alg-dvd y ⟷ (*x* = *0* ⟶ *y* = *0*) ∧ *algebraic-int* (*y* / *x*)

lemma *alg-dvd-imp-algebraic-int*:
  fixes *x y* :: *'a* :: *field-char-0*
  shows *x alg-dvd y* ⟹ *algebraic-int x* ⟹ *algebraic-int y*
  using *algebraic-int-times*[*of y* / *x x*] **by** (*auto simp*: *alg-dvd-def*)

lemma *alg-dvd-0-left-iff* [*simp*]: *0 alg-dvd x* ⟷ *x* = *0*
  by (*auto simp*: *alg-dvd-def*)

lemma *alg-dvd-0-right* [*iff*]: *x alg-dvd 0*
  by (*auto simp*: *alg-dvd-def*)

lemma *one-alg-dvd-iff* [*simp*]: *1 alg-dvd x* ⟷ *algebraic-int x*
  by (*auto simp*: *alg-dvd-def*)

lemma *alg-dvd-of-int* [*intro*]:
  assumes *x dvd y*
  shows *of-int x alg-dvd of-int y*
proof (*cases of-int x* = (*0* :: *'a*))
  case *False*
  from *assms* **obtain** *z* **where** *z*: *y* = *x* ∗ *z*
    by (*elim dvdE*)
  have *algebraic-int* (*of-int z*)
    by *auto*
  also have *of-int z* = *of-int y* / (*of-int x* :: *'a*)
    using *False* **by** (*simp add*: *z field-simps*)
  finally show *?thesis*
    using *False* **by** (*simp add*: *alg-dvd-def*)
qed (*use assms* **in** ‹*auto simp*: *alg-dvd-def*›)

lemma *alg-dvd-of-nat* [*intro*]:
  assumes *x dvd y*
  shows *of-nat x alg-dvd of-nat y*
  using *alg-dvd-of-int*[*of int x int y*] *assms* **by** *simp*

lemma *alg-dvd-of-int-iff* [*simp*]:

(*of-int x* :: *'a* :: *field-char-0*) *alg-dvd of-int y* ⟷ *x dvd y*
proof
assume (*of-int x* :: *'a*) *alg-dvd of-int y*
hence *of-int y* / (*of-int x* :: *'a*) ∈ ℤ **and** *nz*: *of-int x* = (*0*::*'a*) ⟶ *of-int y* = (*0*::*'a*)
by (*auto simp*: *alg-dvd-def dest*!: *rational-algebraic-int-is-int*)
then obtain *n* **where** *of-int y* / *of-int x* = (*of-int n* :: *'a*)
by (*elim Ints-cases*)
hence *of-int y* = (*of-int* (*x* ∗ *n*) :: *'a*)
unfolding *of-int-mult* **using** *nz* **by** (*auto simp*: *field-simps*)
hence *y* = *x* ∗ *n*
by (*subst* (*asm*) *of-int-eq-iff*)
thus *x dvd y*
by *auto*
qed *blast*

lemma *alg-dvd-of-nat-iff* [*simp*]:
(*of-nat x* :: *'a* :: *field-char-0*) *alg-dvd of-nat y* ⟷ *x dvd y*
proof −
have (*of-int* (*int x*) :: *'a*) *alg-dvd of-int* (*int y*) ⟷ *x dvd y*
by (*subst alg-dvd-of-int-iff*) *auto*
thus *?thesis* **unfolding** *of-int-of-nat-eq* **.**
qed

lemma *alg-dvd-add* [*intro*]:
fixes *x y z* :: *'a* :: *field-char-0*
shows *x alg-dvd y* ⟹ *x alg-dvd z* ⟹ *x alg-dvd* (*y* + *z*)
unfolding *alg-dvd-def* **by** (*auto simp*: *add-divide-distrib*)

lemma *alg-dvd-uminus-right* [*intro*]: *x alg-dvd y* ⟹ *x alg-dvd* −*y*
by (*auto simp*: *alg-dvd-def*)

lemma *alg-dvd-uminus-right-iff* [*simp*]: *x alg-dvd* −*y* ⟷ *x alg-dvd y*
using *alg-dvd-uminus-right*[*of x y*] *alg-dvd-uminus-right*[*of x* −*y*] **by** *auto*

lemma *alg-dvd-diff* [*intro*]:
fixes *x y z* :: *'a* :: *field-char-0*
shows *x alg-dvd y* ⟹ *x alg-dvd z* ⟹ *x alg-dvd* (*y* − *z*)
unfolding *alg-dvd-def* **by** (*auto simp*: *diff-divide-distrib*)

lemma *alg-dvd-triv-left* [*intro*]: *algebraic-int y* ⟹ *x alg-dvd x* ∗ *y*
by (*auto simp*: *alg-dvd-def*)

lemma *alg-dvd-triv-right* [*intro*]: *algebraic-int x* ⟹ *y alg-dvd x* ∗ *y*
by (*auto simp*: *alg-dvd-def*)

lemma *alg-dvd-triv-left-iff*: *x alg-dvd x* ∗ *y* ⟷ *x* = *0* ∨ *algebraic-int y*
by (*auto simp*: *alg-dvd-def*)

lemma *alg-dvd-triv-right-iff*: *y alg-dvd x * y ⟷ y = 0 ∨ algebraic-int x*
by (*auto simp*: *alg-dvd-def*)

lemma *alg-dvd-triv-left-iff'* [*simp*]: *x ≠ 0 ⟹ x alg-dvd x * y ⟷ algebraic-int y*
by (*simp add*: *alg-dvd-triv-left-iff*)

lemma *alg-dvd-triv-right-iff'* [*simp*]: *y ≠ 0 ⟹ y alg-dvd x * y ⟷ algebraic-int x*
by (*simp add*: *alg-dvd-triv-right-iff*)

lemma *alg-dvd-trans* [*trans*]:
fixes *x y z* :: *'a* :: *field-char-0*
shows *x alg-dvd y ⟹ y alg-dvd z ⟹ x alg-dvd z*
using *algebraic-int-times*[*of y / x z / y*] **by** (*auto simp*: *alg-dvd-def*)

lemma *alg-dvd-mono* [*simp*]:
fixes *a b c d* :: *'a* :: *field-char-0*
shows *a alg-dvd c ⟹ b alg-dvd d ⟹ (a * b) alg-dvd (c * d)*
using *algebraic-int-times*[*of c / a d / b*] **by** (*auto simp*: *alg-dvd-def*)

lemma *alg-dvd-mult* [*simp*]:
fixes *a b c* :: *'a* :: *field-char-0*
shows *a alg-dvd c ⟹ algebraic-int b ⟹ a alg-dvd (b * c)*
using *alg-dvd-mono*[*of a c 1 b*] **by** (*auto simp*: *mult.commute*)

lemma *alg-dvd-mult2* [*simp*]:
fixes *a b c* :: *'a* :: *field-char-0*
shows *a alg-dvd b ⟹ algebraic-int c ⟹ a alg-dvd (b * c)*
using *alg-dvd-mult*[*of a b c*] **by** (*simp add*: *mult.commute*)

A crucial theorem: if an integer x divides a rational number y, then y is in fact also an integer, and that integer is a multiple of x.

lemma *alg-dvd-int-rat*:
fixes *y* :: *'a* :: *field-char-0*
assumes *of-int x alg-dvd y* **and** *y* ∈ ℚ
shows ∃ *n*. *y = of-int n ∧ x dvd n*
proof (*cases x = 0*)
case *False*
have *y / of-int x* ∈ ℤ
by (*intro rational-algebraic-int-is-int*) (*use assms* **in** ‹*auto simp*: *alg-dvd-def*›)
then obtain *n* **where** *n*: *of-int n = y / (of-int x* :: *'a)*
by (*elim Ints-cases*) *auto*
hence *y = of-int (n * x)*
using *False* **by** (*simp add*: *field-simps*)
thus *?thesis* **by** (*intro exI*[*of - x * n*]) *auto*
qed (*use assms* **in** *auto*)

lemma *prod-alg-dvd-prod*:
fixes *f* :: *'a* ⇒ *'b* :: *field-char-0*

assumes $\bigwedge x.\ x \in A \Longrightarrow f\ x$ *alg-dvd* $g\ x$
shows *prod* $f\ A$ *alg-dvd prod* $g\ A$
using *assms* **by** (*induction* A *rule*: *infinite-finite-induct*) *auto*

lemma *alg-dvd-sum*:
fixes $f :: 'a \Rightarrow 'b ::$ *field-char-0*
assumes $\bigwedge x.\ x \in A \Longrightarrow y$ *alg-dvd* $f\ x$
shows y *alg-dvd sum* $f\ A$
using *assms* **by** (*induction* A *rule*: *infinite-finite-induct*) *auto*

lemma *not-alg-dvd-sum*:
fixes $f :: 'a \Rightarrow 'b ::$ *field-char-0*
assumes $\bigwedge x.\ x \in A - \{x'\} \Longrightarrow y$ *alg-dvd* $f\ x$
assumes $\neg y$ *alg-dvd* $f\ x'$
assumes $x' \in A$ *finite* A
shows $\neg y$ *alg-dvd sum* $f\ A$
proof
assume $*$: y *alg-dvd sum* $f\ A$
have y *alg-dvd sum* $f\ A - sum\ f\ (A - \{x'\})$
using $\langle x' \in A \rangle$ **by** (*intro alg-dvd-diff*[*OF* $*$ *alg-dvd-sum*] *assms*) *auto*
also have $\ldots = sum\ f\ (A - (A - \{x'\}))$
using *assms* **by** (*subst sum-diff*) *auto*
also have $A - (A - \{x'\}) = \{x'\}$
using *assms* **by** *auto*
finally show *False* **using** *assms* **by** *simp*
qed

lemma *fact-dvd-pochhammer*:
assumes $m \leq n + 1$
shows *fact* m *dvd pochhammer* (*int* n − *int* $m + 1$) m
proof −
have (*real* n *gchoose* m) $*$ *fact* $m =$ *of-int* (*pochhammer* (*int* n − *int* $m + 1$) m)
by (*simp add*: *gbinomial-pochhammer*′ *pochhammer-of-int* [*symmetric*])
also have (*real* n *gchoose* m) $*$ *fact* $m =$ *of-int* (*int* (n *choose* m) $*$ *fact* m)
by (*simp add*: *binomial-gbinomial*)
finally have *int* (n *choose* m) $*$ *fact* $m =$ *pochhammer* (*int* n − *int* $m + 1$) m
by (*subst* (*asm*) *of-int-eq-iff*)
from *this* [*symmetric*] **show** *?thesis* **by** *simp*
qed

lemma *coeff-higher-pderiv*:
coeff ((*pderiv* ^^ m) f) $n =$ *pochhammer* (*of-nat* (*Suc* n)) m $*$ *coeff* f ($n + m$)
by (*induction* m *arbitrary*: n) (*simp-all add*: *coeff-pderiv pochhammer-rec algebra-simps*)

lemma *fact-alg-dvd-poly-higher-pderiv*:
fixes $p :: 'a ::$ *field-char-0 poly*
assumes $\bigwedge i.$ *algebraic-int* (*poly.coeff* $p\ i$) *algebraic-int* x $m \leq k$
shows *fact* m *alg-dvd poly* ((*pderiv* ^^ k) p) x

unfolding *poly-altdef*
proof (*intro alg-dvd-sum*, *goal-cases*)
case (*1 i*)
have (*of-int* (*fact m*) :: *'a*) *alg-dvd* (*of-int* (*fact k*))
by (*intro alg-dvd-of-int fact-dvd assms*)
also have (*of-int* (*fact k*) :: *'a*) *alg-dvd of-int* (*pochhammer* (*int i* + *1*) *k*)
using *fact-dvd-pochhammer*[*of k i* + *k*]
by (*intro alg-dvd-of-int fact-dvd-pochhammer*) (*auto simp*: *algebra-simps*)
finally have *fact m alg-dvd* (*pochhammer* (*of-nat i* + *1*) *k* :: *'a*)
by (*simp flip*: *pochhammer-of-int*)
also have ... *alg-dvd pochhammer* (*of-nat i* + *1*) *k* ∗ *poly.coeff p* (*i* + *k*)
by (*rule alg-dvd-triv-left*) (*rule assms*)
also have ... = *poly.coeff* ((*pderiv* ^^ *k*) *p*) *i*
unfolding *coeff-higher-pderiv* **by** (*simp add*: *add-ac flip*: *pochhammer-of-int*)
also have ... *alg-dvd poly.coeff* ((*pderiv* ^^ *k*) *p*) *i* ∗ *x* ^ *i*
by (*intro alg-dvd-triv-left algebraic-int-power assms*)
finally show *?case* **.**
qed

end

2 Auxiliary facts about univariate polynomials

theory *More-Polynomial-HLW*
imports
HOL−Computational-Algebra.Computational-Algebra
Polynomial-Factorization.Gauss-Lemma
Power-Sum-Polynomials.Power-Sum-Polynomials-Library
Algebraic-Numbers.Algebraic-Numbers
begin

instance *poly* :: ({*idom-divide*,*normalization-semidom-multiplicative*,*factorial-ring-gcd*,
semiring-gcd-mult-normalize}) *factorial-semiring-multiplicative* **..**

lemma *lead-coeff-prod-mset*:
fixes *A* :: *'a*::{*comm-semiring-1*, *semiring-no-zero-divisors*} *poly multiset*
shows *Polynomial.lead-coeff* (*prod-mset A*) = *prod-mset* (*image-mset Polynomial.lead-coeff A*)
by (*induction A*) (*auto simp*: *Polynomial.lead-coeff-mult*)

lemma *content-normalize* [*simp*]:
fixes *p* :: *'a* :: {*factorial-semiring*, *idom-divide*, *semiring-gcd*, *normalization-semidom-multiplicative*} *poly*
shows *content* (*normalize p*) = *content p*
proof (*cases p* = *0*)
case [*simp*]: *False*
have *content p* = *content* (*unit-factor p* ∗ *normalize p*)
by *simp*
also have ... = *content* (*unit-factor p*) ∗ *content* (*normalize p*)

```
    by (rule content-mult)
  also have content (unit-factor p) = 1
    by (auto simp: unit-factor-poly-def)
  finally show ?thesis by simp
qed auto

lemma rat-to-normalized-int-poly-exists:
  fixes p :: rat poly
  assumes p ≠ 0
  obtains q lc where p = Polynomial.smult lc (of-int-poly q) lc > 0 content q =
1
proof −
  define lc where lc = fst (rat-to-normalized-int-poly p)
  define q where q = snd (rat-to-normalized-int-poly p)
  have eq: rat-to-normalized-int-poly p = (lc, q)
    by (simp add: lc-def q-def)
  show ?thesis
    using rat-to-normalized-int-poly[OF eq] assms
    by (intro that[of lc q]) auto
qed

lemma irreducible-imp-squarefree:
  assumes irreducible p
  shows   squarefree p
proof (rule squarefreeI)
  fix q assume q ^ 2 dvd p
  then obtain r where qr: p = q ^ 2 * r
    by (elim dvdE)
  have q dvd 1 ∨ q * r dvd 1
    by (intro irreducibleD[OF assms]) (use qr in ‹simp-all add: power2-eq-square
mult-ac›)
  thus q dvd 1
    by (meson dvd-mult-left)
qed

lemma squarefree-imp-rsquarefree:
  fixes p :: 'a :: idom poly
  assumes squarefree p
  shows   rsquarefree p
  unfolding rsquarefree-def
proof (intro conjI allI)
  fix x :: 'a
  have Polynomial.order x p < 2
  proof (rule ccontr)
    assume ¬(Polynomial.order x p < 2)
    hence [:−x, 1:] ^ 2 dvd p
      by (subst order-divides) auto
    from assms and this have [:−x, 1:] dvd 1
      by (rule squarefreeD)
```

```
    hence Polynomial.degree [:−x, 1:] ≤ Polynomial.degree (1 :: 'a poly)
      by (rule dvd-imp-degree-le) auto
    thus False by simp
  qed
  thus Polynomial.order x p = 0 ∨ Polynomial.order x p = 1
    by linarith
qed (use assms in auto)

lemma squarefree-imp-coprime-pderiv:
  fixes p :: 'a :: {factorial-ring-gcd,semiring-gcd-mult-normalize,semiring-char-0}
poly
  assumes squarefree p and content p = 1
  shows   Rings.coprime p (pderiv p)
proof (rule coprimeI-primes)
  fix d assume d: prime d d dvd p d dvd pderiv p
  show False
  proof (cases Polynomial.degree d = 0)
    case deg: False
    obtain q where dq: p = d * q
      using d by (elim dvdE)
    have ‹d dvd q * pderiv d›
      using d by (simp add: dq pderiv-mult dvd-add-right-iff)
    moreover have ¬d dvd pderiv d
    proof
      assume d dvd pderiv d
      hence Polynomial.degree d ≤ Polynomial.degree (pderiv d)
        using d deg by (intro dvd-imp-degree-le) (auto simp: pderiv-eq-0-iff)
      hence Polynomial.degree d = 0
        by (subst (asm) degree-pderiv) auto
      thus False using deg by contradiction
    qed
    ultimately have d dvd q
      using d(1) by (simp add: prime-dvd-mult-iff)
    hence d ^ 2 dvd p
      by (auto simp: dq power2-eq-square)
    from assms(1) and this have is-unit d
      by (rule squarefreeD)
    thus False using ‹prime d› by auto
  next
    case True
    then obtain d' where [simp]: d = [:d':]
      by (elim degree-eq-zeroE)
    from d have d' dvd content p
      by (simp add: const-poly-dvd-iff-dvd-content)
    with assms and prime-imp-prime-elem[OF ‹prime d›] show False
      by (auto simp: prime-elem-const-poly-iff)
  qed
qed (use assms in auto)
```

lemma *irreducible-imp-coprime-pderiv*:
fixes $p :: 'a :: \{idom\text{-}divide, semiring\text{-}char\text{-}0\}$ *poly*
assumes *irreducible p Polynomial.degree p* $\neq$ *0*
shows *Rings.coprime p* (*pderiv p*)
proof (*rule Rings.coprimeI*)
fix *d* **assume** *d*: *d dvd p d dvd pderiv p*
obtain *q* **where** *dq*: $p = d * q$
using *d* **by** (*elim dvdE*)
have *is-unit d* $\vee$ *is-unit q*
using *assms dq* **by** (*auto simp*: *irreducible-def*)
thus *is-unit d*
proof
assume *unit*: *is-unit q*
with *d* **have** *p dvd pderiv p*
using *algebraic-semidom-class.mult-unit-dvd-iff dq* **by** *blast*
hence *Polynomial.degree p* = *0*
by (*meson not-dvd-pderiv*)
with *assms*(*2*) **show** *?thesis* **by** *contradiction*
qed
qed

lemma *poly-gcd-eq-0I*:
assumes *poly p x* = *0 poly q x* = *0*
shows *poly* (*gcd p q*) *x* = *0*
using *assms* **by** (*simp add*: *poly-eq-0-iff-dvd*)

lemma *poly-eq-0-coprime*:
assumes *Rings.coprime p q p* $\neq$ *0 q* $\neq$ *0*
shows *poly p x* $\neq$ *0* $\vee$ *poly q x* $\neq$ *0*
proof −
have *False* **if** *poly p x* = *0 poly q x* = *0*
proof −
have [:−*x*, *1*:] *dvd p* [:−*x*, *1*:] *dvd q*
using *that* **by** (*simp-all add*: *poly-eq-0-iff-dvd*)
hence [:−*x*, *1*:] *dvd 1*
using ‹*Rings.coprime p q*› **by** (*meson not-coprimeI*)
thus *False*
by (*simp add*: *is-unit-poly-iff*)
qed
thus *?thesis*
by *blast*
qed

lemma *coprime-of-int-polyI*:
assumes *Rings.coprime p q*
shows *Rings.coprime* (*of-int-poly p*) (*of-int-poly q* :: $'a :: \{$*field-char-0*,*field-gcd*$\}$ *poly*)
using *assms gcd-of-int-poly*[*of p q*, **where** $?'a = 'a$] **unfolding** *coprime-iff-gcd-eq-1*
by *simp*

lemma *irreducible-imp-rsquarefree-of-int-poly*:
fixes p :: *int poly*
assumes *irreducible p* **and** *Polynomial.degree p* > *0*
shows *rsquarefree* (*of-int-poly p* :: $'a$ :: {*field-gcd*, *field-char-0*} *poly*)
proof −
{
fix x :: $'a$
assume x: *poly* (*of-int-poly p*) x = *0 poly* (*pderiv* (*of-int-poly p*)) x = *0*
define d **where** d = *gcd* (*of-int-poly p*) (*pderiv* (*of-int-poly p*) :: $'a$ *poly*)
have *poly d x* = *0*
using x **unfolding** *d-def* **by** (*intro poly-gcd-eq-0I*) *auto*
moreover have $d \neq 0$
using *assms* **by** (*auto simp*: *d-def*)
ultimately have *0* < *Polynomial.degree d*
by (*intro Nat.gr0I*) (*auto elim*!: *degree-eq-zeroE*)
also have *Polynomial.degree d* = *Polynomial.degree* (*gcd p* (*pderiv p*))
unfolding *d-def of-int-hom.map-poly-pderiv*[*symmetric*] *gcd-of-int-poly* **by** *simp*
finally have *deg*: ... > *0* **.**

have *gcd p* (*pderiv p*) *dvd p*
by *auto*
from *irreducibleD'*[*OF assms*(*1*) *this*] **and** *deg* **have** *p dvd gcd p* (*pderiv p*)
by *auto*
also have ... *dvd pderiv p*
by *auto*
finally have *Polynomial.degree p* = *0*
by *auto*
with *assms* **have** *False* **by** *simp*
}
thus *?thesis* **by** (*auto simp*: *rsquarefree-roots*)
qed

lemma *squarefree-of-int-polyI*:
assumes *squarefree p content p* = *1*
shows *squarefree* (*of-int-poly p* :: $'a$ :: {*field-char-0*,*field-gcd*} *poly*)
proof −
have *Rings.coprime p* (*pderiv p*)
by (*rule squarefree-imp-coprime-pderiv*) *fact*+
hence *Rings.coprime* (*of-int-poly p* :: $'a$ *poly*) (*of-int-poly* (*pderiv p*))
by (*rule coprime-of-int-polyI*)
also have *of-int-poly* (*pderiv p*) = *pderiv* (*of-int-poly p* :: $'a$ *poly*)
by (*simp add*: *of-int-hom.map-poly-pderiv*)
finally show *?thesis*
using *coprime-pderiv-imp-squarefree* **by** *blast*
qed

lemma *higher-pderiv-pcompose-linear*:

(*pderiv* ^^ *n*) (*pcompose p* [:*0*, *c*:]) =
Polynomial.smult (*c* ^ *n*) (*pcompose* ((*pderiv* ^^ *n*) *p*) [:*0*, *c*:])
by (*induction n*) (*simp-all add*: *pderiv-pcompose pderiv-smult pderiv-pCons pcompose-smult mult-ac*)

lemma *poly-poly-eq*:
poly (*poly p* [:*x*:]) *y* = *poly* (*eval-poly* (λ*p*. [:*poly p y*:]) *p* [:*0*, *1*:]) *x*
by (*induction p*) (*auto simp*: *eval-poly-def*)

lemma *poly-poly-poly-y-x* [*simp*]:
fixes *p* :: *'a* :: *idom poly poly*
shows *poly* (*poly* (*poly-y-x p*) [:*y*:]) *x* = *poly* (*poly p* [:*x*:]) *y*
proof (*induction p*)
case (*pCons a p*)
have *poly* (*poly* (*poly-y-x* (*pCons a p*)) [:*y*:]) *x* =
poly a y + *poly* (*poly* (*map-poly* (*pCons 0*) (*poly-y-x p*)) [:*y*:]) *x*
by (*simp add*: *poly-y-x-pCons eval-poly-def*)
also have *pCons 0* = (λ*p*::*'a poly*. *Polynomial.monom 1 1* ∗ *p*)
by (*simp add*: *Polynomial.monom-altdef*)
also have *map-poly* ... (*poly-y-x p*) = *Polynomial.smult* (*Polynomial.monom 1 1*) (*poly-y-x p*)
by (*simp add*: *smult-conv-map-poly*)
also have *poly* ... [:*y*:] = *Polynomial.monom 1 1* ∗ *poly* (*poly-y-x p*) [:*y*:]
by *simp*
also have *poly a y* + *poly* ... *x* = *poly* (*poly* (*pCons a p*) [:*x*:]) *y*
by (*simp add*: *pCons poly-monom*)
finally show *?case* **.**
qed *auto*

lemma (**in** *idom-hom*) *map-poly-higher-pderiv* [*hom-distribs*]:
map-poly hom ((*pderiv* ^^ *n*) *p*) = (*pderiv* ^^ *n*) (*map-poly hom p*)
by (*induction n*) (*simp-all add*: *map-poly-pderiv*)

lemma *coeff-prod-linear-factors*:
fixes *f* :: *'a* ⇒ *'b* :: *comm-ring-1*
assumes [*intro*]: *finite A*
shows *Polynomial.coeff* (∏ *x*∈*A*. [:−*f x*, *1*:] ^ *e x*) *i* =
(∑ *X* | *X* ∈ *Pow* (*SIGMA x*:*A*. {..<*e x*}) ∧ *i* = *sum e A* − *card X*.
(−*1*) ^ *card X* ∗ (∏ *x*∈*X*. *f* (*fst x*)))
proof −
define *poly-X* **where** *poly-X* = (*Polynomial.monom 1 1* :: *'b poly*)
have [*simp*]: (− *1*) ^ *n* = [:(− *1*) ^ *n* :: *'b*:] **for** *n* :: *nat*
by (*simp flip*: *pCons-one add*: *poly-const-pow*)
have (∏ *x*∈*A*. [:−*f x*, *1*:] ^ *e x*) = (∏ (*x*,-)∈*Sigma A* (λ*x*. {..<*e x*}). [:−*f x*, *1*:])
by (*subst prod.Sigma* [*symmetric*]) *auto*
also have ... = (∏ (*x*,-)∈*Sigma A* (λ*x*. {..<*e x*}). *poly-X* − [:*f x*:])
by (*intro prod.cong*) (*auto simp*: *poly-X-def monom-altdef*)
also have ... = (∑ *X*∈*Pow* (*SIGMA x*:*A*. {..<*e x*}).
Polynomial.smult ((−*1*) ^ *card X* ∗ (∏ *x*∈*X*. *f* (*fst x*)))

```
                (poly-X ^ card ((SIGMA x:A. {..<e x}) - X)))
    unfolding case-prod-unfold
    by (subst prod-diff1) (auto simp: mult-ac simp flip: coeff-lift-hom.hom-prod)
  also have ... = (∑ X∈Pow (SIGMA x:A. {..<e x}).
       Polynomial.monom ((- 1) ^ card X * (∏ x∈X. f (fst x))) (card ((SIGMA
x:A. {..<e x}) - X)))
    unfolding poly-X-def monom-power Polynomial.smult-monom by simp
  also have Polynomial.coeff ... i = (∑ X∈{X∈Pow (SIGMA x:A. {..<e x}). i
=
                sum e A - card X}. (- 1) ^ card X * (∏ x∈X. f (fst x)))
    unfolding Polynomial.coeff-sum
  proof (intro sum.mono-neutral-cong-right ballI, goal-cases)
    case (3 X)
    hence X: X ⊆ (SIGMA x:A. {..<e x})
      by auto
    have card-le: card X ≤ card (SIGMA x:A. {..<e x})
      using X by (intro card-mono) auto
    have finite X
      by (rule finite-subset[OF X]) auto
    hence card ((SIGMA x:A. {..<e x}) - X) = card (SIGMA x:A. {..<e x}) -
card X
      using 3 by (intro card-Diff-subset) auto
    also have card-eq: card (SIGMA x:A. {..<e x}) = sum e A
      by (subst card-SigmaI) auto
    finally show ?case
      using 3 card-le card-eq by (auto simp: algebra-simps)
  next
    case (4 X)
    hence X: X ⊆ (SIGMA x:A. {..<e x})
      by auto
    have finite X
      by (rule finite-subset[OF X]) auto
    hence card ((SIGMA x:A. {..<e x}) - X) = card (SIGMA x:A. {..<e x}) -
card X
      using 4 by (intro card-Diff-subset) auto
    also have card-eq: card (SIGMA x:A. {..<e x}) = sum e A
      by (subst card-SigmaI) auto
    finally show ?case
      using 4 card-eq by (auto simp: algebra-simps)
  qed auto
  finally show ?thesis .
qed

lemma (in comm-ring-hom) synthetic-div-hom:
  synthetic-div (map-poly hom p) (hom x) = map-poly hom (synthetic-div p x)
  by (induction p) (auto simp: map-poly-pCons-hom)

lemma synthetic-div-altdef:
  fixes p :: 'a :: field poly
```

```
  shows synthetic-div p c = p div [:−c, 1:]
proof −
  define q where q = p div [:− c, 1:]
  have Polynomial.degree (p mod [:−c, 1:]) = 0
  proof (cases p mod [:−c, 1:] = 0)
    case False
    hence Polynomial.degree (p mod [:−c, 1:]) < Polynomial.degree [:−c, 1:]
      by (intro degree-mod-less′) auto
    thus ?thesis by simp
  qed auto
  then obtain d where d: p mod [:−c, 1:] = [:d:]
    by (elim degree-eq-zeroE)

  have p-eq: p = q ∗ [:−c, 1:] + [:d:]
    unfolding q-def d [symmetric] by presburger
  have [simp]: poly p c = d
    by (simp add: p-eq)
  have p + Polynomial.smult c q = pCons (poly p c) q
    by (subst p-eq) auto
  from synthetic-div-unique[OF this] show ?thesis
    by (auto simp: q-def)
qed

lemma (in ring-closed) poly-closed [intro]:
  assumes ⋀i. poly.coeff p i ∈ A x ∈ A
  shows   poly p x ∈ A
  unfolding poly-altdef by (intro sum-closed mult-closed power-closed assms)

lemma (in ring-closed) coeff-pCons-closed [intro]:
  assumes ⋀i. poly.coeff p i ∈ A x ∈ A
  shows   poly.coeff (pCons x p) i ∈ A
  unfolding poly-altdef using assms by (auto simp: coeff-pCons split: nat.splits)

lemma (in ring-closed) coeff-poly-mult-closed [intro]:
  assumes ⋀i. poly.coeff p i ∈ A ⋀i. poly.coeff q i ∈ A
  shows   poly.coeff (p ∗ q) i ∈ A
  unfolding coeff-mult using assms by auto

lemma (in ring-closed) coeff-poly-prod-closed [intro]:
  assumes ⋀x i. x ∈ X ⟹ poly.coeff (f x) i ∈ A
  shows   poly.coeff (prod f X) i ∈ A
  using assms by (induction X arbitrary: i rule: infinite-finite-induct) auto

lemma (in ring-closed) coeff-poly-power-closed [intro]:
  assumes ⋀i. poly.coeff p i ∈ A
  shows   poly.coeff (p ^ n) i ∈ A
  using coeff-poly-prod-closed[of {..<n} λ-. p i] assms by simp

lemma (in ring-closed) synthetic-div-closed:
```

assumes $\bigwedge i.$ *poly.coeff p i* $\in A$ $x \in A$
shows *poly.coeff* (*synthetic-div p x*) $i \in A$
proof −
from *assms*(*1*) **have** $\forall\, i.$ *poly.coeff p i* $\in A$
by *blast*
from *this* **and** *assms*(*2*) **show** *?thesis*
by (*induction p arbitrary*: *i*) (*auto simp*: *coeff-pCons split*: *nat.splits*)
qed

lemma *pcompose-monom*: *pcompose* (*Polynomial.monom c n*) *p* = *Polynomial.smult c* (*p* ^ *n*)
by (*simp add*: *monom-altdef pcompose-hom.hom-power pcompose-smult*)

lemma *poly-roots-uminus* [*simp*]: *poly-roots* (−*p*) = *poly-roots p*
using *poly-roots-smult*[*of* −*1 p*] **by** (*simp del*: *poly-roots-smult*)

lemma *poly-roots-normalize* [*simp*]:
fixes *p* :: *'a* :: {*normalization-semidom*, *idom-divide*} *poly*
shows *poly-roots* (*normalize p*) = *poly-roots p*
proof (*cases p* = *0*)
case [*simp*]: *False*
have *poly-roots p* = *poly-roots* (*unit-factor p* ∗ *normalize p*)
by *simp*
also have ... = *poly-roots* (*normalize p*)
unfolding *unit-factor-poly-def* **by** *simp*
finally show *?thesis* **..**
qed *auto*

lemma *poly-roots-of-int-normalize* [*simp*]:
poly-roots (*of-int-poly* (*normalize p*) :: *'a* :: {*idom*, *ring-char-0*} *poly*) =
poly-roots (*of-int-poly p*)
proof (*cases p* = *0*)
case [*simp*]: *False*
have *poly-roots* (*of-int-poly p* :: *'a poly*) = *poly-roots* (*of-int-poly* (*unit-factor p* ∗ *normalize p*))
by *simp*
also have ... = *poly-roots* (*Polynomial.smult* (*of-int* (*sgn* (*Polynomial.lead-coeff p*)))
(*of-int-poly* (*normalize p*)))
by (*simp add*: *unit-factor-poly-def of-int-hom.map-poly-hom-smult*)
also have ... = *poly-roots* (*Ring-Hom-Poly.of-int-poly* (*normalize p*) :: *'a poly*)
by (*intro poly-roots-smult*) (*auto simp*: *sgn-if*)
finally show *?thesis* **..**
qed *auto*

lemma *poly-roots-power* [*simp*]: *poly-roots* (*p* ^ *n*) = *repeat-mset n* (*poly-roots p*)
proof (*cases p* = *0*)
case *True*

thus *?thesis* **by** (*cases n*) *auto*
next
case *False*
thus *?thesis* **by** (*induction n*) (*auto simp*: *poly-roots-mult*)
qed

lemma *poly-roots-conv-sum-prime-factors*:
poly-roots q = (∑ *p*∈#*prime-factorization q*. *poly-roots p*)
proof (*cases q* = *0*)
case [*simp*]: *False*

have (∑ *p*∈#*prime-factorization q*. *poly-roots p*) =
poly-roots (*prod-mset* (*prime-factorization q*))
by (*rule poly-roots-prod-mset* [*symmetric*]) *auto*
also have ... = *poly-roots* (*normalize* (*prod-mset* (*prime-factorization q*)))
by *simp*
also have *normalize* (*prod-mset* (*prime-factorization q*)) = *normalize q*
by (*rule prod-mset-prime-factorization-weak*) *auto*
also have *poly-roots* ... = *poly-roots q*
by *simp*
finally show *?thesis* **..**
qed *auto*

lemma *poly-roots-of-int-conv-sum-prime-factors*:
poly-roots (*of-int-poly q* :: *'a* :: {*idom*, *ring-char-0*} *poly*) =
(∑ *p*∈#*prime-factorization q*. *poly-roots* (*of-int-poly p*))
proof (*cases q* = *0*)
case [*simp*]: *False*

have (∑ *p*∈#*prime-factorization q*. *poly-roots* (*of-int-poly p* :: *'a poly*)) =
poly-roots (∏ *p*∈#*prime-factorization q*. *of-int-poly p*)
by (*subst poly-roots-prod-mset*) (*auto simp*: *multiset.map-comp o-def*)
also have (∏ *p*∈#*prime-factorization q*. *of-int-poly p* :: *'a poly*) =
of-int-poly (*prod-mset* (*prime-factorization q*))
by *simp*
also have *poly-roots* ... = *poly-roots* (*of-int-poly* (*normalize* (*prod-mset* (*prime-factorization q*))))
by (*rule poly-roots-of-int-normalize* [*symmetric*])
also have *normalize* (*prod-mset* (*prime-factorization q*)) = *normalize q*
by (*rule prod-mset-prime-factorization-weak*) *auto*
also have *poly-roots* (*of-int-poly* ... :: *'a poly*) = *poly-roots* (*of-int-poly q*)
by *simp*
finally show *?thesis* **..**
qed *auto*

lemma *dvd-imp-poly-roots-subset*:
assumes *q* ≠ *0 p dvd q*
shows *poly-roots p* ⊆# *poly-roots q*
proof −

```
  from assms have p ≠ 0
    by auto
  thus ?thesis
    using assms by (intro mset-subset-eqI) (auto intro: dvd-imp-order-le)
qed

lemma abs-prod-mset: |prod-mset (A :: 'a :: idom-abs-sgn multiset)| = prod-mset
(image-mset abs A)
  by (induction A) (auto simp: abs-mult)

lemma content-1-imp-nonconstant-prime-factors:
  assumes content (p :: int poly) = 1 and q ∈ prime-factors p
  shows   Polynomial.degree q > 0
proof −
  let ?d = Polynomial.degree :: int poly ⇒ nat
  let ?lc = Polynomial.lead-coeff :: int poly ⇒ int
  define P where P = prime-factorization p
  define P1 where P1 = filter-mset (λp. ?d p = 0) P
  define P2 where P2 = filter-mset (λp. ?d p > 0) P
  have [simp]: p ≠ 0
    using assms by auto
  have 1 = content (normalize p)
    using assms by simp
  also have normalize p = prod-mset P
    unfolding P-def by (rule prod-mset-prime-factorization [symmetric]) auto
  also have P = filter-mset (λp. ?d p = 0) P + filter-mset (λp. ?d p > 0) P
    by (induction P) auto
  also have prod-mset ... = prod-mset P1 * prod-mset P2
    unfolding P1-def P2-def by (subst prod-mset.union) auto
  also have content ... = content (prod-mset P1) * content (prod-mset P2)
    unfolding content-mult ..
  also have image-mset id P1 = image-mset (λq. [:?lc q:]) P1
    by (intro image-mset-cong) (auto simp: P1-def elim!: degree-eq-zeroE)
  hence P1 = image-mset (λq. [:?lc q:]) P1
    by simp
  also have content (prod-mset ...) = |(∏ q∈#P1. ?lc q)|
    by (simp add: content-prod-mset multiset.map-comp o-def abs-prod-mset)
  finally have |(∏ q∈#P1. ?lc q)| * content (prod-mset P2) = 1 ..
  hence |(∏ q∈#P1. ?lc q)| dvd 1
    unfolding dvd-def by metis

  have set-mset P1 = {}
  proof (rule ccontr)
    assume set-mset P1 ≠ {}
    then obtain q where q: q ∈# P1
      by blast
    have |?lc q| dvd (∏ q∈#P1. |?lc q|)
      by (rule dvd-prod-mset) (use q in auto)
    also have ... = |(∏ q∈#P1. ?lc q)|
```

```
    by (simp add: abs-prod-mset multiset.map-comp o-def)
  also have ... dvd 1
    by fact
  finally have is-unit (?lc q)
    by simp
  hence is-unit q
    using q unfolding P1-def by (auto elim!: degree-eq-zeroE)
  moreover have prime q
    using q unfolding P1-def P-def by auto
  ultimately show False by auto
 qed
 with assms show ?thesis
  by (auto simp: P1-def P-def)
qed

end
```

This theory imports both univariate and multivariate polynomials and thereby causes several overlaps in notation of polynomials.

```
theory More-Min-Int-Poly
 imports
  Algebraic-Numbers.Min-Int-Poly
  HOL−Computational-Algebra.Computational-Algebra
  More-Polynomial-HLW
begin

lemma min-int-poly-squarefree [intro]:
 fixes x :: 'a :: {field-char-0, field-gcd}
 shows squarefree (min-int-poly x)
 by (rule irreducible-imp-squarefree) auto

lemma min-int-poly-conv-Gcd:
 fixes x :: 'a :: {field-char-0, field-gcd}
 assumes algebraic x
 shows min-int-poly x = Gcd {p. p ≠ 0 ∧ p represents x}
proof (rule sym, rule Gcd-eqI, (safe)?)
 fix p assume p: ⋀q. q ∈ {p. p ≠ 0 ∧ p represents x} ⟹ p dvd q
 show p dvd min-int-poly x
  using assms by (intro p) auto
next
 fix p assume p: p ≠ 0 p represents x
 have min-int-poly x represents x
  using assms by auto
 hence poly (gcd (of-int-poly (min-int-poly x)) (of-int-poly p)) x = 0
  using p by (intro poly-gcd-eq-0I) auto
 hence ipoly (gcd (min-int-poly x) p) x = 0
  by (subst (asm) gcd-of-int-poly) auto
 hence gcd (min-int-poly x) p represents x
  using p unfolding represents-def by auto
```

have *min-int-poly x dvd gcd* (*min-int-poly x*) *p* ∨ *is-unit* (*gcd* (*min-int-poly x*) *p*)
by (*intro irreducibleD'*) *auto*
moreover from ‹*gcd* (*min-int-poly x*) *p represents x*› **have** ¬*is-unit* (*gcd* (*min-int-poly x*) *p*)
by (*auto simp*: *represents-def*)
ultimately have *min-int-poly x dvd gcd* (*min-int-poly x*) *p*
by *blast*
also have ... *dvd p*
by *blast*
finally show *min-int-poly x dvd p* **.**
qed *auto*

end

3 The lexicographic ordering on complex numbers

theory *Complex-Lexorder*
imports *Complex-Main HOL−Library.Multiset*
begin

We define a lexicographic order on the complex numbers, comparing first the real parts and, if they are equal, the imaginary parts. This ordering is of course not compatible with multiplication, but it is compatible with addition.

definition *less-eq-complex-lex* (**infix** $\leq_{\mathbb{C}}$ *50*) **where**
less-eq-complex-lex x y ⟷ *Re x* < *Re y* ∨ *Re x* = *Re y* ∧ *Im x* ≤ *Im y*

definition *less-complex-lex* (**infix** $<_{\mathbb{C}}$ *50*) **where**
less-complex-lex x y ⟷ *Re x* < *Re y* ∨ *Re x* = *Re y* ∧ *Im x* < *Im y*

interpretation *complex-lex*:
linordered-ab-group-add (+) *0* (−) *uminus less-eq-complex-lex less-complex-lex*
by *standard* (*auto simp*: *less-eq-complex-lex-def less-complex-lex-def complex-eq-iff*)

lemmas [*trans*] =
complex-lex.order.trans complex-lex.less-le-trans
complex-lex.less-trans complex-lex.le-less-trans

lemma (**in** *ordered-comm-monoid-add*) *sum-mono-complex-lex*:
(⋀*i*. *i*∈*K* ⟹ *f i* $\leq_{\mathbb{C}}$ *g i*) ⟹ (∑ *i*∈*K*. *f i*) $\leq_{\mathbb{C}}$ (∑ *i*∈*K*. *g i*)
by (*induct K rule*: *infinite-finite-induct*) (*use complex-lex.add-mono* **in** *auto*)

lemma *sum-strict-mono-ex1-complex-lex*:
fixes *f g* :: *'i* ⇒ *complex*
assumes *finite A*
and ∀ *x*∈*A*. *f x* $\leq_{\mathbb{C}}$ *g x*
and ∃ *a*∈*A*. *f a* $<_{\mathbb{C}}$ *g a*

```
  shows sum f A <_C sum g A
proof−
  from assms(3) obtain a where a: a ∈ A f a <_C g a by blast
  have sum f A = sum f ((A − {a}) ∪ {a})
    by (simp add: insert-absorb[OF ‹a ∈ A›])
  also have ... = sum f (A − {a}) + sum f {a}
    using ‹finite A› by (subst sum.union-disjoint) auto
  also have ... ≤_C sum g (A − {a}) + sum f {a}
    by (intro complex-lex.add-mono sum-mono-complex-lex) (simp-all add: assms)
  also have ... <_C sum g (A − {a}) + sum g {a}
    using a by (intro complex-lex.add-strict-left-mono) auto
  also have ... = sum g ((A − {a}) ∪ {a})
    using ‹finite A› by (subst sum.union-disjoint[symmetric]) auto
  also have ... = sum g A by (simp add: insert-absorb[OF ‹a ∈ A›])
  finally show ?thesis
    by simp
qed

lemma sum-list-mono-complex-lex:
  assumes list-all2 (≤_C) xs ys
  shows   sum-list xs ≤_C sum-list ys
  using assms by induction (auto intro: complex-lex.add-mono)

lemma sum-mset-mono-complex-lex:
  assumes rel-mset (≤_C) A B
  shows   sum-mset A ≤_C sum-mset B
 using assms by (auto simp: rel-mset-def sum-mset-sum-list intro: sum-list-mono-complex-lex)

lemma rel-msetI:
  assumes list-all2 R xs ys mset xs = A mset ys = B
  shows   rel-mset R A B
  using assms by (auto simp: rel-mset-def)

lemma mset-replicate [simp]: mset (replicate n x) = replicate-mset n x
  by (induction n) auto

lemma rel-mset-replicate-mset-right:
  assumes ⋀x. x ∈# A ⟹ R x y size A = n
  shows   rel-mset R A (replicate-mset n y)
proof −
  obtain xs where [simp]: A = mset xs
    by (metis ex-mset)
  from assms have ∀x∈set xs. R x y
    by auto
  hence list-all2 R xs (replicate (length xs) y)
    by (induction xs) auto
  with assms(2) show ?thesis
    by (intro rel-msetI[of R xs replicate n y]) auto
qed
```

end

4 Additional facts about multivariate polynomials

theory *More-Multivariate-Polynomial-HLW*
imports *Power-Sum-Polynomials.Power-Sum-Polynomials-Library*
begin

4.1 Miscellaneous

lemma *Var-altdef*: *Var i* = *monom* (*Poly-Mapping.single i 1*) *1*
by *transfer'* (*simp add*: Var_0*-def*)

lemma *Const-conv-monom*: *Const c* = *monom 0 c*
by *transfer'* (*auto simp*: $Const_0$*-def*)

lemma *smult-conv-mult-Const*: *smult c p* = *Const c* ∗ *p*
by (*simp add*: *smult-conv-mult Const-conv-monom*)

lemma *mpoly-map-vars-Var* [*simp*]: *bij f* ⟹ *mpoly-map-vars f* (*Var i*) = *Var* (*f i*)
unfolding *Var-altdef*
by (*subst mpoly-map-vars-monom*) (*auto simp*: *permutep-single bij-imp-bij-inv inv-inv-eq*)

lemma *symmetric-mpoly-symmetric-prod'*:
assumes ⋀π. π *permutes A* ⟹ *g* π *permutes X*
assumes ⋀*x* π. *x* ∈ *X* ⟹ π *permutes A* ⟹ *mpoly-map-vars* π (*f x*) = *f* (*g* π *x*)
shows *symmetric-mpoly A* (∏ *x*∈*X*. *f x*)
unfolding *symmetric-mpoly-def*
proof *safe*
fix π **assume** π: π *permutes A*
have *mpoly-map-vars* π (*prod f X*) = (∏ *x*∈*X*. *mpoly-map-vars* π (*f x*))
by *simp*
also have ... = (∏ *x*∈*X*. *f* (*g* π *x*))
by (*intro prod.cong assms* π *refl*)
also have ... = (∏ *x*∈*g* π ‘*X*. *f x*)
using *assms*(*1*)[*OF* π] **by** (*subst prod.reindex*) (*auto simp*: *permutes-inj-on*)
also have *g* π ‘ *X* = *X*
using *assms*(*1*)[*OF* π] **by** (*simp add*: *permutes-image*)
finally show *mpoly-map-vars* π (*prod f X*) = *prod f X* **.**
qed

4.2 Converting a univariate polynomial into a multivariate one

lift-definition *mpoly-of-poly-aux* :: *nat* ⇒ *'a* :: *zero poly* ⇒ (*nat* ⇒$_0$ *nat*) ⇒$_0$ *'a*
is
  λi c m. if Poly-Mapping.keys m ⊆ {i} then c (Poly-Mapping.lookup m i) else 0
proof *goal-cases*
  case (*1 i c*)
  hence *fin*: *finite {n. c n ≠ 0}*
    by (*metis eventually-cofinite*)
  show *finite {x. (if keys x ⊆ {i} then c (lookup x i) else 0) ≠ 0}*
  proof (*rule finite-subset*)
    show *finite (Poly-Mapping.single i ' {n. c n ≠ 0})*
      by (*intro finite-imageI fin*)
  next
    show *{x. (if keys x ⊆ {i} then c (lookup x i) else 0) ≠ 0} ⊆*
          Poly-Mapping.single i ' {n. c n ≠ 0}
    proof (*safe*, *split if-splits*)
      fix *x* :: (*nat* ⇒$_0$ *nat*)
      assume *x*: *keys x ⊆ {i} c (lookup x i) ≠ 0*
      hence *x = Poly-Mapping.single i (lookup x i)*
        by (*metis Diff-eq-empty-iff keys-empty-iff lookup-single-eq*
              remove-key-keys remove-key-single remove-key-sum)
      thus *x ∈ Poly-Mapping.single i ' {n. c n ≠ 0}*
        using *x* **by** *blast*
    qed *auto*
  qed
qed

lift-definition *mpoly-of-poly* :: *nat* ⇒ *'a* :: *zero poly* ⇒ *'a mpoly* **is**
  mpoly-of-poly-aux **.**

lemma *mpoly-of-poly-0* [*simp*]: *mpoly-of-poly i 0 = 0*
  by (*transfer'*, *transfer*) *auto*

lemma *coeff-mpoly-of-poly1* [*simp*]:
  coeff (mpoly-of-poly i p) (Poly-Mapping.single i n) = poly.coeff p n
  by (*transfer'*, *transfer'*) *auto*

lemma *coeff-mpoly-of-poly2* [*simp*]:
  assumes *¬keys x ⊆ {i}*
  shows *coeff (mpoly-of-poly i p) x = 0*
  using *assms* **by** (*transfer'*, *transfer'*) *auto*

lemma *coeff-mpoly-of-poly*:
  coeff (mpoly-of-poly i p) m =
    (poly.coeff p (Poly-Mapping.lookup m i) when keys m ⊆ {i})
  by (*transfer'*, *transfer'*) *auto*

lemma *poly-mapping-single-eq-0-iff* [*simp*]: *Poly-Mapping.single i n = 0 ⟷ n =*

```
0
  by (metis lookup-single-eq single-zero)

lemma mpoly-of-poly-pCons [simp]:
  fixes p :: 'a :: semiring-1 poly
  shows mpoly-of-poly i (pCons c p) = Const c + Var i * mpoly-of-poly i p
proof (rule mpoly-eqI)
  fix mon :: nat ⇒₀ nat
  define moni :: nat ⇒₀ nat where moni = Poly-Mapping.single i 1
  have coeff (Var i * mpoly-of-poly i p) mon =
        (∑ l. (1 when l = moni) * (∑ q. coeff (mpoly-of-poly i p) q when mon =
moni + q))
    unfolding coeff-mpoly-times prod-fun-def coeff-Var moni-def
    by (rule Sum-any.cong) (auto simp: when-def)
  also have ... = (∑ a. coeff (mpoly-of-poly i p) a when mon = moni + a)
    by (subst Sum-any-left-distrib [symmetric]) simp-all
  finally have eq: coeff (Var i * mpoly-of-poly i p) mon = ... .

  show coeff (mpoly-of-poly i (pCons c p)) mon = coeff (Const c + Var i *
mpoly-of-poly i p) mon
  proof (cases keys mon ⊆ {i})
    case False
    hence [simp]: mon ≠ 0
      by auto
    obtain j where j: j ∈ keys mon j ≠ i
      using False by auto
    have coeff (mpoly-of-poly i p) mon' = 0 if mon-eq: mon = moni + mon' for
mon'
    proof −
      have Poly-Mapping.lookup mon j ≠ 0
        using j by (meson lookup-eq-zero-in-keys-contradict)
      also have Poly-Mapping.lookup mon j = Poly-Mapping.lookup mon' j
       unfolding mon-eq moni-def using j by (simp add: lookup-add lookup-single)
      finally have j ∈ keys mon'
        by (meson lookup-not-eq-zero-eq-in-keys)
      with j have ¬keys mon' ⊆ {i}
        by blast
      thus ?thesis by simp
    qed
    hence coeff (Var i * mpoly-of-poly i p) mon = 0
      unfolding eq by (intro Sum-any-zeroI) (auto simp: when-def)
    thus ?thesis using False
      by (simp add: mpoly-coeff-Const)
  next
    case True
    define n where n = Poly-Mapping.lookup mon i
    have mon-eq: mon = Poly-Mapping.single i n
      using True unfolding n-def
     by (metis Diff-eq-empty-iff add-cancel-right-left keys-empty-iff remove-key-keys
```

remove-key-sum)
    have *eq'*: *mon* = *moni* + *mon'* ⟷ *n* > *0* ∧ *mon'* = *Poly-Mapping.single i* (*n* − *1*) **for** *mon'*
    proof *safe*
      assume *eq*: *mon* = *moni* + *mon'*
      thus *n* > *0 mon'* = *Poly-Mapping.single i* (*n* − *1*)
        unfolding *moni-def mon-eq* **using** *gr0I* **by** (*force simp*: *single-diff*)+
    next
      assume *n* > *0 mon'* = *Poly-Mapping.single i* (*n* − *1*)
      thus *mon* = *moni* + *Poly-Mapping.single i* (*n* − *1*)
        unfolding *mon-eq moni-def* **by** (*subst single-add* [*symmetric*]) *auto*
    qed
    have *coeff* (*Var i* ∗ *mpoly-of-poly i p*) *mon* = (*poly.coeff p* (*n* − *1*) *when* (*n* > *0*))
      unfolding *eq eq'* **by** (*auto simp*: *when-def*)
    thus *?thesis*
     by (*auto simp*: *mon-eq when-def mpoly-coeff-Const coeff-pCons split*: *nat.splits*)
  qed
qed

lemma *mpoly-of-poly-1* [*simp*]: *mpoly-of-poly i 1* = *1*
  unfolding *one-pCons mpoly-of-poly-pCons mpoly-of-poly-0* **by** *simp*

lemma *mpoly-of-poly-uminus* [*simp*]: *mpoly-of-poly i* (−*p*) = −*mpoly-of-poly i p*
  by (*rule mpoly-eqI*) (*auto simp*: *coeff-mpoly-of-poly when-def*)

lemma *mpoly-of-poly-add* [*simp*]: *mpoly-of-poly i* (*p* + *q*) = *mpoly-of-poly i p* + *mpoly-of-poly i q*
  by (*rule mpoly-eqI*) (*auto simp*: *coeff-mpoly-of-poly when-def*)

lemma *mpoly-of-poly-diff* [*simp*]: *mpoly-of-poly i* (*p* − *q*) = *mpoly-of-poly i p* − *mpoly-of-poly i q*
  by (*rule mpoly-eqI*) (*auto simp*: *coeff-mpoly-of-poly when-def*)

lemma *mpoly-of-poly-smult* [*simp*]:
  mpoly-of-poly i (*Polynomial.smult c p*) = *smult c* (*mpoly-of-poly i p*)
  by (*rule mpoly-eqI*) (*auto simp*: *coeff-mpoly-of-poly when-def*)

lemma *mpoly-of-poly-mult* [*simp*]:
  fixes *p q* :: *'a* :: *comm-semiring-1 poly*
  shows *mpoly-of-poly i* (*p* ∗ *q*) = *mpoly-of-poly i p* ∗ *mpoly-of-poly i q*
  by (*induction p*) (*auto simp*: *algebra-simps smult-conv-mult-Const*)

lemma *insertion-mpoly-of-poly* [*simp*]: *insertion f* (*mpoly-of-poly i p*) = *poly p* (*f i*)
  by (*induction p*) (*auto simp*: *insertion-add insertion-mult*)

lemma *mapping-of-mpoly-of-poly* [*simp*]: *mapping-of* (*mpoly-of-poly i p*) = *mpoly-of-poly-aux i p*

by *transfer' simp*

lemma *vars-mpoly-of-poly*: *vars* (*mpoly-of-poly i p*) ⊆ {*i*}
proof −
have *x* = *i* **if** *xa* ∈ *keys* (*mpoly-of-poly-aux i p*) *x* ∈ *keys xa* **for** *x xa*
using *that*
by (*meson in-mono lookup-eq-zero-in-keys-contradict mpoly-of-poly-aux.rep-eq singletonD*)
thus *?thesis*
by (*auto simp*: *vars-def*)
qed

lemma *mpoly-map-vars-mpoly-of-poly* [*simp*]:
assumes *bij f*
shows *mpoly-map-vars f* (*mpoly-of-poly i p*) = *mpoly-of-poly* (*f i*) *p*
proof (*rule mpoly-eqI*, *goal-cases*)
case (*1 mon*)
have *f* −‘ *keys mon* ⊆ {*i*} ⟷ *keys mon* ⊆ {*f i*}
using *assms* **by** (*simp add*: *vimage-subset-eq*)
thus *?case* **using** *assms*
by (*simp add*: *coeff-mpoly-map-vars coeff-mpoly-of-poly lookup-permutep keys-permutep when-def*)
qed

end

5 More facts about algebraic numbers

theory *More-Algebraic-Numbers-HLW*
imports *Algebraic-Numbers.Algebraic-Numbers*
begin

5.1 Miscellaneous

lemma *in-Ints-imp-algebraic* [*simp*, *intro*]: *x* ∈ ℤ ⟹ *algebraic x*
by (*intro algebraic-int-imp-algebraic int-imp-algebraic-int*)

lemma *in-Rats-imp-algebraic* [*simp*, *intro*]: *x* ∈ ℚ ⟹ *algebraic x*
by (*auto elim*!: *Rats-cases' intro*: *algebraic-div*)

lemma *algebraic-uminus-iff* [*simp*]: *algebraic* (−*x*) ⟷ *algebraic x*
using *algebraic-uminus*[*of x*] *algebraic-uminus*[*of* −*x*] **by** *auto*

lemma *algebraic-0* [*simp*]: *algebraic* (*0* :: *'a* :: *field-char-0*)
and *algebraic-1* [*simp*]: *algebraic* (*1* :: *'a* :: *field-char-0*)
by *auto*

lemma *algebraic-sum* [*intro*]:
(⋀*x*. *x* ∈ *A* ⟹ *algebraic* (*f x*)) ⟹ *algebraic* (*sum f A*)

by (*induction A rule*: *infinite-finite-induct*) (*auto intro*!: *algebraic-plus*)

lemma *algebraic-prod* [*intro*]:
(⋀*x*. *x* ∈ *A* ⟹ *algebraic* (*f x*)) ⟹ *algebraic* (*prod f A*)
by (*induction A rule*: *infinite-finite-induct*) (*auto intro*!: *algebraic-times*)

lemma *algebraic-sum-list* [*intro*]:
(⋀*x*. *x* ∈ *set xs* ⟹ *algebraic x*) ⟹ *algebraic* (*sum-list xs*)
by (*induction xs*) (*auto intro*!: *algebraic-plus*)

lemma *algebraic-prod-list* [*intro*]:
(⋀*x*. *x* ∈ *set xs* ⟹ *algebraic x*) ⟹ *algebraic* (*prod-list xs*)
by (*induction xs*) (*auto intro*!: *algebraic-times*)

lemma *algebraic-sum-mset* [*intro*]:
(⋀*x*. *x* ∈# *A* ⟹ *algebraic x*) ⟹ *algebraic* (*sum-mset A*)
by (*induction A*) (*auto intro*!: *algebraic-plus*)

lemma *algebraic-prod-mset* [*intro*]:
(⋀*x*. *x* ∈# *A* ⟹ *algebraic x*) ⟹ *algebraic* (*prod-mset A*)
by (*induction A*) (*auto intro*!: *algebraic-times*)

lemma *algebraic-power* [*intro*]: *algebraic x* ⟹ *algebraic* (*x* ^ *n*)
by (*induction n*) (*auto intro*: *algebraic-times*)

lemma *algebraic-csqrt* [*intro*]: *algebraic x* ⟹ *algebraic* (*csqrt x*)
by (*rule algebraic-nth-root*[*of 2 x*]) *auto*

lemma *algebraic-csqrt-iff* [*simp*]: *algebraic* (*csqrt x*) ⟷ *algebraic x*
proof
assume *algebraic* (*csqrt x*)
hence *algebraic* (*csqrt x* ^ *2*)
by (*rule algebraic-power*)
also have *csqrt x* ^ *2* = *x*
by *simp*
finally show *algebraic x* **.**
qed *auto*

lemmas [*intro*] = *algebraic-plus algebraic-times algebraic-uminus algebraic-div*

lemma *algebraic-power-iff* [*simp*]:
assumes *n* > *0*
shows *algebraic* (*x* ^ *n*) ⟷ *algebraic x*
using *algebraic-nth-root*[*of n x* ^ *n x*] *assms* **by** *auto*

lemma *algebraic-ii* [*simp*]: *algebraic* i
by (*intro algebraic-int-imp-algebraic*) *auto*

lemma *algebraic-int-fact* [*simp*, *intro*]: *algebraic-int* (*fact n*)

by (*intro int-imp-algebraic-int fact-in-Ints*)

lemma *algebraic-minus* [*intro*]: *algebraic x* $\Longrightarrow$ *algebraic y* $\Longrightarrow$ *algebraic* $(x - y)$
using *algebraic-plus*[*of x* $-y$] **by** *simp*

lemma *algebraic-add-cancel-left* [*simp*]:
assumes *algebraic x*
shows *algebraic* $(x + y) \longleftrightarrow$ *algebraic y*
proof
assume *algebraic* $(x + y)$
hence *algebraic* $(x + y - x)$
using *assms* **by** (*intro algebraic-minus*) *auto*
thus *algebraic y* **by** *simp*
qed (*auto intro*: *algebraic-plus assms*)

lemma *algebraic-add-cancel-right* [*simp*]:
assumes *algebraic y*
shows *algebraic* $(x + y) \longleftrightarrow$ *algebraic x*
using *algebraic-add-cancel-left*[*of y x*] *assms*
by (*simp add*: *add.commute del*: *algebraic-add-cancel-left*)

lemma *algebraic-diff-cancel-left* [*simp*]:
assumes *algebraic x*
shows *algebraic* $(x - y) \longleftrightarrow$ *algebraic y*
using *algebraic-add-cancel-left*[*of x* $-y$] *assms* **by** (*simp del*: *algebraic-add-cancel-left*)

lemma *algebraic-diff-cancel-right* [*simp*]:
assumes *algebraic y*
shows *algebraic* $(x - y) \longleftrightarrow$ *algebraic x*
using *algebraic-add-cancel-right*[*of* $-y$ *x*] *assms* **by** (*simp del*: *algebraic-add-cancel-right*)

lemma *algebraic-mult-cancel-left* [*simp*]:
assumes *algebraic x* $x \neq 0$
shows *algebraic* $(x * y) \longleftrightarrow$ *algebraic y*
proof
assume *algebraic* $(x * y)$
hence *algebraic* $(x * y / x)$
using *assms* **by** (*intro algebraic-div*) *auto*
also have $x * y / x = y$
using *assms* **by** *auto*
finally show *algebraic y* **.**
qed (*auto intro*: *algebraic-times assms*)

lemma *algebraic-mult-cancel-right* [*simp*]:
assumes *algebraic y* $y \neq 0$
shows *algebraic* $(x * y) \longleftrightarrow$ *algebraic x*
using *algebraic-mult-cancel-left*[*of y x*] *assms*
by (*simp add*: *mult.commute del*: *algebraic-mult-cancel-left*)

lemma *algebraic-inverse-iff* [*simp*]: *algebraic* (*inverse y*) ⟷ *algebraic y*
proof
  assume *algebraic* (*inverse y*)
  hence *algebraic* (*inverse* (*inverse y*))
    by (*rule algebraic-inverse*)
  thus *algebraic y* **by** *simp*
qed (*auto intro*: *algebraic-inverse*)

lemma *algebraic-divide-cancel-left* [*simp*]:
  assumes *algebraic x x* ≠ *0*
  shows *algebraic* (*x* / *y*) ⟷ *algebraic y*
proof −
  have *algebraic* (*x* ∗ *inverse y*) ⟷ *algebraic* (*inverse y*)
    by (*intro algebraic-mult-cancel-left assms*)
  also have ... ⟷ *algebraic y*
    by (*intro algebraic-inverse-iff*)
  finally show *?thesis* **by** (*simp add*: *field-simps*)
qed

lemma *algebraic-divide-cancel-right* [*simp*]:
  assumes *algebraic y y* ≠ *0*
  shows *algebraic* (*x* / *y*) ⟷ *algebraic x*
proof −
  have *algebraic* (*x* ∗ *inverse y*) ⟷ *algebraic x*
    using *assms* **by** (*intro algebraic-mult-cancel-right*) *auto*
  thus *?thesis* **by** (*simp add*: *field-simps*)
qed

5.2 Turning an algebraic number into an algebraic integer

5.3 Multiplying an algebraic number with a suitable integer turns it into an algebraic integer.

lemma *algebraic-imp-algebraic-int*:
  fixes *x* :: *'a* :: *field-char-0*
  assumes *ipoly p x* = *0 p* ≠ *0*
  defines *c* ≡ *Polynomial.lead-coeff p*
  shows *algebraic-int* (*of-int c* ∗ *x*)
proof −
  define *n* **where** *n* = *Polynomial.degree p*
  define *p'* **where** *p'* = *Abs-poly* (λ*i*. *if i* = *n then 1 else c* ^ (*n* − *i* − *1*) ∗ *poly.coeff p i*)
  have *n* > *0*
    using *assms* **unfolding** *n-def* **by** (*intro Nat.gr0I*) (*auto elim*!: *degree-eq-zeroE*)

  have *coeff-p'*: *poly.coeff p' i* =
                (*if i* = *n then 1 else c* ^ (*n* − *i* − *1*) ∗ *poly.coeff p i*)
    (**is** - = *?f i*) **for** *i* **unfolding** *p'-def*
  proof (*subst poly.Abs-poly-inverse*)
    have *eventually* (λ*i*. *poly.coeff p i* = *0*) *cofinite*

```
    using MOST-coeff-eq-0 by blast
  hence eventually (λi. ?f i = 0) cofinite
    by eventually-elim (use assms in ‹auto simp: n-def›)
  thus ?f ∈ {f. eventually (λi. f i = 0) cofinite} by simp
qed auto

have deg-p': Polynomial.degree p' = n
proof −
  from assms have (λn. ∀ i>n. poly.coeff p' i = 0) = (λn. ∀ i>n. poly.coeff p i
= 0)
    by (auto simp: coeff-p' fun-eq-iff n-def)
  thus ?thesis
    by (simp add: Polynomial.degree-def n-def)
qed

have lead-coeff-p': Polynomial.lead-coeff p' = 1
  by (simp add: coeff-p' deg-p')

have 0 = of-int (c ^ (n − 1)) * (∑ i≤n. of-int (poly.coeff p i) * x ^ i)
  using assms unfolding n-def poly-altdef by simp
also have ... = (∑ i≤n. of-int (c ^ (n − 1) * poly.coeff p i) * x ^ i)
  by (simp add: sum-distrib-left sum-distrib-right mult-ac)
also have ... = (∑ i≤n. of-int (poly.coeff p' i) * (of-int c * x) ^ i)
proof (intro sum.cong, goal-cases)
  case (2 i)
  have of-int (poly.coeff p' i) * (of-int c * x) ^ i =
        of-int (c ^ i * poly.coeff p' i) * x ^ i
    by (simp add: algebra-simps)
  also have c ^ i * poly.coeff p' i = c ^ (n − 1) * poly.coeff p i
  proof (cases i = n)
    case True
    hence c ^ i * poly.coeff p' i = c ^ n
      by (auto simp: coeff-p' simp flip: power-Suc)
    also have n = Suc (n − 1)
      using ‹n > 0› by simp
    also have c ^ ... = c * c ^ (n − 1)
      by simp
    finally show ?thesis
      using True by (simp add: c-def n-def)
  next
    case False
    thus ?thesis using 2
      by (auto simp: coeff-p' simp flip: power-add)
  qed
  finally show ?case ..
qed auto
also have ... = ipoly p' (of-int c * x)
  by (simp add: poly-altdef n-def deg-p')
finally have ipoly p' (of-int c * x) = 0 ..
```

with *lead-coeff-p'* **show** *?thesis*
unfolding *algebraic-int-altdef-ipoly* **by** *blast*
qed

lemma *algebraic-imp-algebraic-int'*:
fixes $x :: 'a :: field\text{-}char\text{-}0$
assumes *ipoly p x = 0 p ≠ 0 Polynomial.lead-coeff p dvd c*
shows *algebraic-int* (*of-int c * x*)
proof −
from *assms*(*3*) **obtain** *c'* **where** *c-eq*: *c = Polynomial.lead-coeff p * c'*
by *auto*
have *algebraic-int* (*of-int c' * (of-int* (*Polynomial.lead-coeff p*) * *x*))
by (*rule algebraic-int-times*[*OF - algebraic-imp-algebraic-int*]) (*use assms* **in** *auto*)
also have *of-int c' * (of-int* (*Polynomial.lead-coeff p*) * *x*) = *of-int c * x*
by (*simp add*: *c-eq mult-ac*)
finally show *?thesis* **.**
qed

end

6 Miscellaneous facts

theory *Misc-HLW*
imports
Complex-Main
HOL−Library.Multiset
HOL−Library.FuncSet
HOL−Library.Groups-Big-Fun
HOL−Library.Poly-Mapping
HOL−Library.Landau-Symbols
HOL−Combinatorics.Permutations
HOL−Computational-Algebra.Computational-Algebra
begin

lemma *set-mset-subset-singletonD*:
assumes *set-mset A ⊆ {x}*
shows *A = replicate-mset* (*size A*) *x*
using *assms* **by** (*induction A*) *auto*

lemma *image-mset-eq-replicate-msetD*:
assumes *image-mset f A = replicate-mset n y*
shows $\forall x \in\# A.\ f\ x = y$
proof −
have *f ' set-mset A = set-mset* (*image-mset f A*)
by *simp*
also note *assms*
finally show *?thesis* **by** (*auto split*: *if-splits*)

qed

lemma *bij-betw-permutes-compose-left*:
assumes π *permutes* A
shows *bij-betw* $(\lambda\sigma.\ \pi \circ \sigma)$ $\{\sigma.\ \sigma$ *permutes* $A\}$ $\{\sigma.\ \sigma$ *permutes* $A\}$
proof (*rule bij-betwI*)
show $(\circ)\ \pi \in \{\sigma.\ \sigma$ *permutes* $A\} \to \{\sigma.\ \sigma$ *permutes* $A\}$
by (*auto intro*: *permutes-compose assms*)
show $(\circ)$ (*inv-into UNIV* π) $\in \{\sigma.\ \sigma$ *permutes* $A\} \to \{\sigma.\ \sigma$ *permutes* $A\}$
by (*auto intro*: *permutes-compose assms permutes-inv*)
qed (*use permutes-inverses*[*OF assms*] **in** *auto*)

lemma *bij-betw-compose-left-perm-Pi*:
assumes π *permutes* B
shows *bij-betw* $(\lambda f.\ (\pi \circ f))$ $(A \to B)$ $(A \to B)$
proof (*rule bij-betwI*)
have $*$: $(\lambda f.\ (\pi \circ f)) \in (A \to B) \to A \to B$ **if** π: π *permutes* B **for** π
by (*auto simp*: *permutes-in-image*[*OF* π])
show $(\lambda f.\ (\pi \circ f)) \in (A \to B) \to A \to B$
by (*rule* $*$) *fact*
show $(\lambda f.\ (\mathit{inv\text{-}into}\ \mathit{UNIV}\ \pi \circ f)) \in (A \to B) \to A \to B$
by (*intro* $*$ *permutes-inv*) *fact*
qed (*auto simp*: *permutes-inverses*[*OF assms*] *fun-eq-iff*)

lemma *bij-betw-compose-left-perm-PiE*:
assumes π *permutes* B
shows *bij-betw* $(\lambda f.\ \mathit{restrict}\ (\pi \circ f)\ A)$ $(A \to_E B)$ $(A \to_E B)$
proof (*rule bij-betwI*)
have $*$: $(\lambda f.\ \mathit{restrict}\ (\pi \circ f)\ A) \in (A \to_E B) \to A \to_E B$ **if** π: π *permutes* B
for π
by (*auto simp*: *permutes-in-image*[*OF* π])
show $(\lambda f.\ \mathit{restrict}\ (\pi \circ f)\ A) \in (A \to_E B) \to A \to_E B$
by (*rule* $*$) *fact*
show $(\lambda f.\ \mathit{restrict}\ (\mathit{inv\text{-}into}\ \mathit{UNIV}\ \pi \circ f)\ A) \in (A \to_E B) \to A \to_E B$
by (*intro* $*$ *permutes-inv*) *fact*
qed (*auto simp*: *permutes-inverses*[*OF assms*] *fun-eq-iff*)

lemma *bij-betw-image-mset-set*:
assumes *bij-betw f A B*
shows *image-mset f* (*mset-set A*) = *mset-set B*
using *assms* **by** (*simp add*: *bij-betw-def image-mset-mset-set*)

lemma *finite-multisets-of-size*:
assumes *finite A*
shows *finite* $\{X.\ \mathit{set\text{-}mset}\ X \subseteq A \wedge \mathit{size}\ X = n\}$
proof (*rule finite-subset*)
show $\{X.\ \mathit{set\text{-}mset}\ X \subseteq A \wedge \mathit{size}\ X = n\} \subseteq \mathit{mset}\ `\ \{xs.\ \mathit{set}\ xs \subseteq A \wedge \mathit{length}\ xs = n\}$
proof

```
    fix X assume X: X ∈ {X. set-mset X ⊆ A ∧ size X = n}
    obtain xs where [simp]: X = mset xs
      by (metis ex-mset)
    thus X ∈ mset ‘ {xs. set xs ⊆ A ∧ length xs = n}
      using X by auto
  qed
next
  show finite (mset ‘ {xs. set xs ⊆ A ∧ length xs = n})
    by (intro finite-imageI finite-lists-length-eq assms)
qed

lemma sum-mset-image-mset-sum-mset-image-mset:
   sum-mset (image-mset g (sum-mset (image-mset f A))) =
    sum-mset (image-mset (λx. sum-mset (image-mset g (f x))) A)
  by (induction A) auto

lemma sum-mset-image-mset-singleton: sum-mset (image-mset (λx. {#f x#}) A)
= image-mset f A
  by (induction A) auto

lemma sum-mset-conv-sum:
  sum-mset (image-mset f A) = (∑ x∈set-mset A. of-nat (count A x) * f x)
proof (induction A rule: full-multiset-induct)
  case (less A)
  show ?case
  proof (cases A = {#})
    case False
    then obtain x where x: x ∈# A
      by auto
    define n where n = count A x
    define A′ where A′ = filter-mset (λy. y ≠ x) A
    have A-eq: A = replicate-mset n x + A′
      by (intro multiset-eqI) (auto simp: A′-def n-def)
    have [simp]: x ∉# A′ count A′ x = 0
      by (auto simp: A′-def)
    have n ≠ 0
      using x by (auto simp: n-def)

    have sum-mset (image-mset f A) = of-nat n * f x + sum-mset (image-mset f
A′)
      by (simp add: A-eq)
    also have A′ ⊂# A
      unfolding A′-def using x by (simp add: filter-mset-eq-conv subset-mset-def)
     with less.IH have sum-mset (image-mset f A′) = (∑ x∈set-mset A′. of-nat
(count A′ x) * f x)
      by simp
    also have ... = (∑ x∈set-mset A′. of-nat (count A x) * f x)
      by (intro sum.cong) (auto simp: A-eq)
   also have of-nat n * f x + ... = (∑ x∈insert x (set-mset A′). of-nat (count A
```

```
x) * f x)
      by (subst sum.insert) (auto simp: A-eq)
    also from ‹n ≠ 0› have insert x (set-mset A') = set-mset A
      by (auto simp: A-eq)
    finally show ?thesis .
  qed auto
qed

lemma sum-mset-conv-Sum-any:
  sum-mset (image-mset f A) = Sum-any (λx. of-nat (count A x) * f x)
proof −
  have sum-mset (image-mset f A) = (∑ x∈set-mset A. of-nat (count A x) * f x)
    by (rule sum-mset-conv-sum)
  also have ... = Sum-any (λx. of-nat (count A x) * f x)
  proof (rule Sum-any.expand-superset [symmetric])
    show {x. of-nat (count A x) * f x ≠ 0} ⊆ set-mset A
    proof
      fix x assume x ∈ {x. of-nat (count A x) * f x ≠ 0}
      hence count A x ≠ 0
        by (intro notI) auto
      thus x ∈# A
        by auto
    qed
  qed auto
  finally show ?thesis .
qed

lemma Sum-any-sum-swap:
  assumes finite A ⋀y. finite {x. f x y ≠ 0}
  shows   Sum-any (λx. sum (f x) A) = (∑ y∈A. Sum-any (λx. f x y))
proof −
  have Sum-any (λx. sum (f x) A) = Sum-any (λx. Sum-any (λy. f x y when y ∈
A))
    unfolding when-def by (subst Sum-any.conditionalize) (use assms in simp-all)
  also have ... = Sum-any (λy. Sum-any (λx. f x y when y ∈ A))
    by (intro Sum-any.swap[of (⋃ y∈A. {x. f x y ≠ 0}) × A] finite-SigmaI fi-
nite-UN-I assms) auto
  also have (λy. Sum-any (λx. f x y when y ∈ A)) = (λy. Sum-any (λx. f x y)
when y ∈ A)
    by (auto simp: when-def)
  also have Sum-any ... = (∑ y∈A. Sum-any (λx. f x y))
    unfolding when-def by (subst Sum-any.conditionalize) (use assms in simp-all)
  finally show ?thesis .
qed

lemma (in landau-pair) big-power:
  assumes f ∈ L F g
  shows   (λx. f x ^ n) ∈ L F (λx. g x ^ n)
  using big-prod[of {..<n} λ-. f F λ-. g] assms by simp
```

lemma (**in** *landau-pair*) *small-power*:
assumes $f \in l\ F\ g\ n > 0$
shows $(\lambda x.\ f\ x\ \widehat{}\ n) \in l\ F\ (\lambda x.\ g\ x\ \widehat{}\ n)$
using *assms*(*2*,*1*)
by (*induction rule*: *nat-induct-non-zero*) (*auto intro*!: *small.mult*)

lemma *pairwise-imp-disjoint-family-on*:
assumes *pairwise R A*
assumes $\bigwedge m\ n.\ m \in A \Longrightarrow n \in A \Longrightarrow R\ m\ n \Longrightarrow f\ m \cap f\ n = \{\}$
shows *disjoint-family-on f A*
using *assms*
unfolding *disjoint-family-on-def pairwise-def* **by** *blast*

lemma (**in** *comm-monoid-set*) *If-eq*:
assumes $y \in A$ *finite A*
shows $F\ (\lambda x.\ g\ x\ (\textit{if } x = y \textit{ then } h1\ x \textit{ else } h2\ x))\ A = f\ (g\ y\ (h1\ y))\ (F\ (\lambda x.\ g\ x\ (h2\ x))\ (A - \{y\}))$
proof −
have $F\ (\lambda x.\ g\ x\ (\textit{if } x = y \textit{ then } h1\ x \textit{ else } h2\ x))\ A =$
$f\ (g\ y\ (h1\ y))\ (F\ (\lambda x.\ g\ x\ (\textit{if } x = y \textit{ then } h1\ x \textit{ else } h2\ x))\ (A - \{y\}))$
using *assms* **by** (*subst remove*[*of* - *y*]) *auto*
also have $F\ (\lambda x.\ g\ x\ (\textit{if } x = y \textit{ then } h1\ x \textit{ else } h2\ x))\ (A - \{y\}) = F\ (\lambda x.\ g\ x\ (h2\ x))\ (A - \{y\})$
by (*intro cong*) *auto*
finally show *?thesis* **by** *simp*
qed

lemma *prod-nonzeroI*:
fixes $f :: 'a \Rightarrow 'b :: \{\textit{semiring-no-zero-divisors}, \textit{comm-semiring-1}\}$
assumes $\bigwedge x.\ x \in A \Longrightarrow f\ x \neq 0$
shows $\textit{prod}\ f\ A \neq 0$
using *assms* **by** (*induction rule*: *infinite-finite-induct*) *auto*

lemma *frequently-prime-cofinite*: *frequently* (*prime* :: *nat* ⇒ *bool*) *cofinite*
unfolding *INFM-nat-le* **by** (*meson bigger-prime less-imp-le*)

lemma *frequently-eventually-mono*:
assumes *frequently Q F eventually P F* $\bigwedge x.\ P\ x \Longrightarrow Q\ x \Longrightarrow R\ x$
shows *frequently R F*
proof (*rule frequently-mp*[*OF* - *assms*(*1*)])
show *eventually* $(\lambda x.\ Q\ x \longrightarrow R\ x)\ F$
using *assms*(*2*) **by** *eventually-elim* (*use assms*(*3*) **in** *blast*)
qed

lemma *bij-betw-Diff*:
assumes *bij-betw f A B bij-betw f A' B' A'* ⊆ *A B'* ⊆ *B*
shows *bij-betw f* $(A - A')\ (B - B')$
unfolding *bij-betw-def*

proof
have *inj-on f A*
using *assms*(*1*) **by** (*auto simp*: *bij-betw-def*)
thus *inj-on f* (*A* − *A′*)
by (*rule inj-on-subset*) *auto*
have *f* ‘ (*A* − *A′*) = *f* ‘ *A* − *f* ‘ *A′*
by (*intro inj-on-image-set-diff*[*OF* ‹*inj-on f A*›]) (*use* ‹*A′* ⊆ *A*› **in** *auto*)
also have ... = *B* − *B′*
using *assms*(*1*,*2*) **by** (*auto simp*: *bij-betw-def*)
finally show *f*‘ (*A* − *A′*) = *B* − *B′* **.**
qed

lemma *bij-betw-singleton*: *bij-betw f* {*x*} {*y*} ⟷ *f x* = *y*
by (*auto simp*: *bij-betw-def*)

end

7 The Hermite–Lindemann–WeierstraSS Transcendence Theorem

theory *Hermite-Lindemann*
imports
Pi-Transcendental.Pi-Transcendental
Algebraic-Numbers.Algebraic-Numbers
Algebraic-Integer-Divisibility
More-Min-Int-Poly
Complex-Lexorder
More-Polynomial-HLW
More-Multivariate-Polynomial-HLW
More-Algebraic-Numbers-HLW
Misc-HLW
begin

The Hermite–Lindemann–WeierstraSS theorem answers questions about the transcendence of the exponential function and other related complex functions. It proves that a large number of combinations of exponentials is always transcendental.

A first (much weaker) version of the theorem was proven by Hermite. Lindemann and WeierstraSS then successively generalised it shortly afterwards, and finally Baker gave another, arguably more elegant formulation (which is the one that we will prove, and then derive the traditional version from it).

To honour the contributions of all three of these 19th-century mathematicians, I refer to the theorem as the Hermite–Lindemann–WeierstraSS theorem, even though in other literature it is often called Hermite–Lindemann or Lindemann–WeierstraSS. To keep things short, the Isabelle name of the

theorem, however, will omit WeierstraSS's name.

7.1 Main proof

Following Baker, We first prove the following special form of the theorem: Let $m > 0$ and $q_1, \ldots, q_m \in \mathbb{Z}[X]$ be irreducible, non-constant, and pairwise coprime polynomials. Let $\beta_1, \ldots, \beta_m$ be non-zero integers. Then

$$\sum_{i=1}^{m} \beta_i \sum_{q_i(\alpha)=0} e^{\alpha} \neq 0$$

The difference to the final theorem is that

1. The coefficients β_i are non-zero integers (as opposed to arbitrary algebraic numbers)
2. The exponents α_i occur in full sets of conjugates, and each set has the same coefficient.

In a similar fashion to the proofs of the transcendence of e and π, we define some number J depending on the α_i and β_i and an arbitrary sufficiently large prime p. We then show that, on one hand, J is an integer multiple of $(p-1)!$, but on the other hand it is bounded from above by a term of the form $C_1 \cdot C_2^p$. This is then clearly a contradiction if p is chosen large enough.

lemma *Hermite-Lindemann-aux1*:
fixes P :: *int poly set* **and** β :: *int poly* $\Rightarrow$ *int*
assumes *finite* P **and** $P \neq \{\}$
assumes *distinct*: *pairwise Rings.coprime P*
assumes *irred*: $\bigwedge p.\ p \in P \Longrightarrow$ *irreducible p*
assumes *nonconstant*: $\bigwedge p.\ p \in P \Longrightarrow$ *Polynomial.degree* $p > 0$
assumes β-*nz*: $\bigwedge p.\ p \in P \Longrightarrow \beta\ p \neq 0$
defines *Roots* $\equiv (\lambda p.\ \{\alpha$::*complex. poly* (*of-int-poly p*) $\alpha = 0\})$
shows $(\sum p{\in}P.$ *of-int* $(\beta\ p) * (\sum \alpha{\in}$*Roots p. exp* $\alpha)) \neq 0$
proof
note [*intro*] = ‹*finite P*›
assume *sum-eq-0*: $(\sum p{\in}P.$ *of-int* $(\beta\ p) * (\sum \alpha{\in}$*Roots p. exp* $\alpha)) = 0$

define *Roots'* **where** *Roots'* $= (\bigcup p{\in}P.$ *Roots p*)
have *finite-Roots* [*intro*]: *finite* (*Roots p*) **if** $p \in P$ **for** p
using *nonconstant*[*of p*] *that* **by** (*auto intro*: *poly-roots-finite simp*: *Roots-def*)
have [*intro*]: *finite Roots'*
by (*auto simp*: *Roots'-def*)
have [*simp*]: $0 \notin P$
using *nonconstant*[*of 0*] **by** *auto*
have [*simp*]: $p \neq 0$ **if** $p \in P$ **for** p
using *that* **by** *auto*

The polynomials in P do not have multiple roots:

have *rsquarefree*: *rsquarefree* (*of-int-poly q* :: *complex poly*) **if** $q \in P$ **for** q
by (*rule irreducible-imp-rsquarefree-of-int-poly*) (*use that* **in** ‹*auto intro*: *irred nonconstant*›)

No two different polynomials in P have roots in common:

have *disjoint*: *disjoint-family-on Roots P*
using *distinct*
proof (*rule pairwise-imp-disjoint-family-on*)
fix *p q* **assume** *P*: $p \in P$ $q \in P$ **and** *Rings.coprime p q*
hence *Rings.coprime* (*of-int-poly p* :: *complex poly*) (*of-int-poly q*)
by (*intro coprime-of-int-polyI*)
thus *Roots p* ∩ *Roots q* = {}
using *poly-eq-0-coprime*[*of of-int-poly p of-int-poly q* :: *complex poly*] *P*
by (*auto simp*: *Roots-def*)
qed

define *n-roots* :: *int poly* ⇒ *nat* (♯-)
where *n-roots* = (λ*p*. *card* (*Roots p*))
define *n* **where** *n* = (∑ *p*∈*P*. ♯*p*)
have *n-altdef*: *n* = *card Roots*′
unfolding *n-def Roots*′*-def n-roots-def* **using** *disjoint*
by (*subst card-UN-disjoint*) (*auto simp*: *disjoint-family-on-def*)
have *Roots-nonempty*: *Roots p* ≠ {} **if** $p \in P$ **for** *p*
using *nonconstant*[*OF that*] **by** (*auto simp*: *Roots-def fundamental-theorem-of-algebra constant-degree*)
have *Roots*′ ≠ {}
using *Roots-nonempty* ‹*P* ≠ {}› **by** (*auto simp*: *Roots*′*-def*)
have *n* > *0*
using ‹*Roots*′ ≠ {}› ‹*finite Roots*′› **by** (*auto simp*: *n-altdef*)

We can split each polynomial in P into a product of linear factors:

have *of-int-poly-P*:
of-int-poly q = *Polynomial.smult* (*Polynomial.lead-coeff q*) (∏ *x*∈*Roots q*. [:−*x*, *1*:])
if $q \in P$ **for** *q*
using *complex-poly-decompose-rsquarefree*[*OF rsquarefree*[*OF that*]] **by** (*simp add*: *Roots-def*)

We let l be an integer such that $l\alpha$ is an algebraic integer for all our roots α:

define *l* **where** *l* = (*LCM q*∈*P*. *Polynomial.lead-coeff q*)
have *alg-int*: *algebraic-int* (*of-int l* ∗ *x*) **if** $x \in$ *Roots*′ **for** *x*
proof −
from *that* **obtain** *q* **where** *q*: $q \in P$ *ipoly q x* = *0*
by (*auto simp*: *Roots*′*-def Roots-def*)
show *?thesis*
by (*rule algebraic-imp-algebraic-int*′[*of q*]) (*use q* **in** ‹*auto simp*: *l-def*›)

```
  qed
  have l ≠ 0
    using ‹finite P› by (auto simp: l-def Lcm-0-iff)
  moreover have l ≥ 0
    unfolding l-def by (rule Lcm-int-greater-eq-0)
  ultimately have l > 0 by linarith
```

We can split the product of all the polynomials in P into linear factors:

```
  define lc-factor where lc-factor = (∏ q∈P. l ^ Polynomial.degree q div Polyno-
mial.lead-coeff q)
  have lc-factor: Polynomial.smult (of-int l ^ n) (∏ α'∈Roots'. [:−α',1:]) =
                    of-int-poly (Polynomial.smult lc-factor (∏ P))
  proof −
    define lc where lc = (λq. Polynomial.lead-coeff q :: int)
    define d where d = (Polynomial.degree :: int poly ⇒ nat)
    have (∏ q∈P. of-int-poly q) =
          (∏ q∈P. Polynomial.smult (lc q) (∏ x∈Roots q. [:−x, 1:]) :: complex poly)
      unfolding lc-def by (intro prod.cong of-int-poly-P refl)
    also have ... = Polynomial.smult (∏ q∈P. lc q) (∏ q∈P. (∏ x∈Roots q. [:−x,
1:]))
      by (simp add: prod-smult)
    also have (∏ q∈P. (∏ x∈Roots q. [:−x, 1:])) = (∏ x∈Roots'. [:−x, 1:])
      unfolding Roots'-def using disjoint
    by (intro prod.UNION-disjoint [symmetric]) (auto simp: disjoint-family-on-def)
    also have Polynomial.smult (of-int lc-factor) (Polynomial.smult (∏ q∈P. lc q)
... ) =
                Polynomial.smult (∏ q∈P. of-int (l ^ d q div lc q * lc q)) (∏ x∈Roots'.
pCons (− x) 1)
      by (simp add: lc-factor-def prod.distrib lc-def d-def)
    also have (∏ q∈P. of-int (l ^ d q div lc q * lc q)) = (∏ q∈P. of-int l ^ d q ::
complex)
    proof (intro prod.cong, goal-cases)
      case (2 q)
      have lc q dvd l
        unfolding l-def lc-def using 2 by auto
      also have ... dvd l ^ d q
        using 2 nonconstant[of q] by (intro dvd-power) (auto simp: d-def)
      finally show ?case by simp
    qed auto
    also have ... = l ^ (∑ q∈P. d q)
      by (simp add: power-sum)
    also have (∑ q∈P. d q) = (∑ q∈P. n-roots q)
    proof (intro sum.cong, goal-cases)
      case (2 q)
      thus ?case using rsquarefree[OF 2]
          by (subst (asm) rsquarefree-card-degree) (auto simp: d-def n-roots-def
Roots-def)
    qed auto
    also have ... = n
```

```
    by (simp add: n-def)
  finally show ?thesis
    by (simp add: of-int-hom.map-poly-hom-smult of-int-poly-hom.hom-prod)
qed
```

We define R to be the radius of the smallest circle around the origin in which all our roots lie:

```
define R :: real where R = Max (norm ` Roots')
have R-ge: R ≥ norm α if α ∈ Roots' for α
  unfolding R-def using that by (intro Max-ge) auto
have R ≥ 0
proof −
  from ‹Roots' ≠ {}› obtain α where α ∈ Roots'
    by auto
  have 0 ≤ norm α
    by simp
  also have ... ≤ R
    by (intro R-ge) fact
  finally show R ≥ 0
    by simp
qed
```

Now the main part of the proof: for any sufficiently large prime p, our assumptions imply $(p-1)!^n \leq C' l^{np} (2R)^{np-1}$ for some constant C':

```
define C :: nat ⇒ real where C = (λp. l ^ (n * p) * (2*R) ^ (n * p − 1))
define C' where
  C' = (∏x∈Roots'. ∑q∈P. real-of-int |β q| * (∑α∈Roots q. cmod α * exp
(cmod α)))
```

We commence with the proof of the main inequality.

```
have ineq: fact (p − 1) ^ n ≤ C' * C p ^ n
  if p: prime p
  and p-ineqs: ∀q∈P. p > |β q|
               real p > norm (∏α∈Roots'. of-int (l^n) * (∏x∈Roots'−{α}. α −
x))
  for p :: nat
proof −
  have p > 1
    using prime-gt-1-nat[OF p] .
```

We define the polynomial function

$$f_i(X) = l^{np} \frac{\prod_\alpha (X-\alpha)^p}{X - \alpha_i}$$

where the product runs over all roots α.

```
  define f-poly :: complex ⇒ complex poly where
    f-poly = (λα. Polynomial.smult (l^(n*p)) ((∏α'∈Roots'. [:−α', 1:]^p) div
[:−α, 1:]))
```

have *f-poly-altdef*: *f-poly* α = *Polynomial.smult* ($l \hat{} (n*p)$)
($(\prod \alpha' \in Roots'.\ [:-\alpha',\ 1:] \hat{} (\textit{if}\ \alpha' = \alpha\ \textit{then}\ p - 1\ \textit{else}\ p))$)
if $\alpha \in Roots'$ **for** α
proof −
have $(\prod \alpha' \in Roots'.\ [:-\alpha',\ 1:] \hat{}\ (\textit{if}\ \alpha' = \alpha\ \textit{then}\ p-1\ \textit{else}\ p)) * [:-\alpha,\ 1:] =$
$[:-\alpha,\ 1:] \hat{}\ (p - 1) * (\prod x \in Roots' - \{\alpha\}.\ [:-x,\ 1:] \hat{}\ p) * [:-\alpha,\ 1:]$
using *that* **by** (*subst prod.If-eq*) (*auto simp*: *algebra-simps*)
also have ... = $(\prod x \in Roots' - \{\alpha\}.\ [:-x,\ 1:] \hat{}\ p) * [:-\alpha,\ 1:] \hat{}\ Suc\ (p - 1)$
by (*simp only*: *power-Suc mult-ac*)
also have $Suc\ (p - 1) = p$
using ‹$p > 1$› **by** *auto*
also have $(\prod x \in Roots' - \{\alpha\}.\ [:-x,\ 1:] \hat{}\ p) * [:-\alpha,\ 1:] \hat{}\ p = (\prod x \in Roots'.\ [:-x,\ 1:] \hat{}\ p)$
using *that* **by** (*subst prod.remove*[*of* - α]) *auto*
finally have *eq*: $(\prod \alpha' \in Roots'.\ [:-\alpha',\ 1:] \hat{}\ (\textit{if}\ \alpha' = \alpha\ \textit{then}\ p-1\ \textit{else}\ p)) * [:-\alpha,\ 1:] =$
$(\prod x \in Roots'.\ [:-x,\ 1:] \hat{}\ p)$ **.**
show *?thesis*
unfolding *f-poly-def eq*[*symmetric*] **by** (*subst nonzero-mult-div-cancel-right*) *auto*
qed

define f :: *complex* ⇒ *complex* ⇒ *complex*
where $f = (\lambda \alpha\ x.\ l \hat{} (n*p) * (\prod \alpha' \in Roots'.\ (x - \alpha') \hat{}\ (\textit{if}\ \alpha' = \alpha\ \textit{then}\ p - 1\ \textit{else}\ p)))$
have *eval-f*: *poly* (*f-poly* α) $x = f\ \alpha\ x$ **if** $\alpha \in Roots'$ **for** $\alpha\ x$
using *that* **by** (*simp add*: *f-poly-altdef poly-prod f-def*)
have *deg-f*: *Polynomial.degree* (*f-poly* α) = $n * p - 1$ **if** $\alpha \in Roots'$ **for** α
proof −
have *Polynomial.degree* (*f-poly* α) = $p - 1 + (n - 1) * p$
unfolding *f-poly-altdef*[*OF that*] **using** *that* ‹$l > 0$› ‹*finite Roots'*›
by (*subst prod.If-eq*) (*auto simp*: *degree-prod-eq degree-power-eq degree-mult-eq n-altdef*)
also have $p - 1 + (n - 1) * p = n * p - 1$
using ‹$n > 0$› ‹$p > 1$› **by** (*cases n*) *auto*
finally show *?thesis* **.**
qed

Next, we define the function $I_i(z) = \int_0^z e^{z-t} f_i(t) \mathrm{d}t$, and, based on that, the numbers $J_i = \sum_{i=1}^m \beta_i \sum_{q_i(x)=0} I_i(x)$, and the number J, which is the product of all the J_i:

define I :: *complex* ⇒ *complex* ⇒ *complex*
where $I = (\lambda \alpha\ x.$ *lindemann-weierstrass-aux.I* (*f-poly* α) $x)$
define J :: *complex* ⇒ *complex*
where $J = (\lambda \alpha.\ \sum q \in P.\ \beta\ q * (\sum x \in Roots\ q.\ I\ \alpha\ x))$

define J' :: *complex*
where $J' = (\prod \alpha \in Roots'.\ J\ \alpha)$

Reusing some of the machinery from the proof that e is transcendental, we find the following equality for J_i:

have *J-eq*: $J\ \alpha = -(\sum q{\in}P.\ \textit{of-int}\ (\beta\ q) * (\sum x{\in}\textit{Roots}\ q.\ \sum j{<}n{*}p.\ \textit{poly}\ ((\textit{pderiv}\ \hat{\ }\hat{\ }\ j)\ (\textit{f-poly}\ \alpha))\ x))$
if $\alpha \in \textit{Roots}'$ **for** α
proof −
have $n * p \geq 1 * 2$
using ‹$n > 0$› ‹$p > 1$› **by** (*intro mult-mono*) *auto*
hence [*simp*]: $\{..n{*}p{-}\textit{Suc}\ 0\} = \{..{<}n{*}p\}$
by *auto*
have $J\ \alpha = (\sum q{\in}P.\ \beta\ q * (\sum x{\in}\textit{Roots}\ q.\ I\ \alpha\ x))$
unfolding *J-def* **..**
also have $\ldots = (\sum q{\in}P.\ \textit{of-int}\ (\beta\ q) * (\sum x{\in}\textit{Roots}\ q.\ \textit{exp}\ x * (\sum j{<}n{*}p.\ \textit{poly}\ ((\textit{pderiv}\ \hat{\ }\hat{\ }\ j)\ (\textit{f-poly}\ \alpha))\ 0))) -$
$(\sum q{\in}P.\ \textit{of-int}\ (\beta\ q) * (\sum x{\in}\textit{Roots}\ q.\ \sum j{<}n{*}p.\ \textit{poly}\ ((\textit{pderiv}\ \hat{\ }\hat{\ }\ j)\ (\textit{f-poly}\ \alpha))\ x))$
unfolding *I-def lindemann-weierstrass-aux.I-def*
by (*simp add*: *deg-f that ring-distribs sum-subtractf sum-distrib-left sum-distrib-right mult-ac*)
also have $\ldots = -(\sum q{\in}P.\ \textit{of-int}\ (\beta\ q) * (\sum x{\in}\textit{Roots}\ q.\ \sum j{<}n{*}p.\ \textit{poly}\ ((\textit{pderiv}\ \hat{\ }\hat{\ }\ j)\ (\textit{f-poly}\ \alpha))\ x))$
unfolding *sum-distrib-right* [*symmetric*] *mult.assoc* [*symmetric*] *sum-eq-0*
by *simp*
finally show *?thesis* **.**
qed

The next big step is to show that $(p-1)!\mid J_i$ as an algebraic integer (i.e. $J_i/(p-1)!$ is an algebraic integer), but $p \not| J_i$. This is done by brute force: We show that every summand in the above sum has $p!$ as a factor, except for the one corresponding to $x = \alpha_i$, $j = p-1$, which has $(p-1)!$ as a factor but not p.

have *J*: *fact* $(p-1)$ *alg-dvd* $J\ \alpha\ \neg$*of-nat p alg-dvd* $J\ \alpha$ **if** α: $\alpha \in \textit{Roots}'$ **for** α
proof −
define h **where** $h = (\lambda\alpha'\ j.\ \textit{poly}\ ((\textit{pderiv}\ \hat{\ }\hat{\ }\ j)\ (\textit{f-poly}\ \alpha))\ \alpha')$
from α **obtain** q **where** q: $q \in P\ \alpha \in \textit{Roots}\ q$
by (*auto simp*: *Roots'-def*)

have $J\ \alpha = -(\sum (q,\ \alpha'){\in}\textit{Sigma}\ P\ \textit{Roots}.\ \sum j{<}n{*}p.\ \textit{of-int}\ (\beta\ q) * h\ \alpha'\ j)$
unfolding *J-eq*[*OF* α] *h-def sum-distrib-left* **by** (*subst* (2) *sum.Sigma*) *auto*
also have $\ldots = -(\sum ((q,\alpha'),i){\in}\textit{Sigma}\ P\ \textit{Roots}{\times}\{..{<}n{*}p\}.\ \textit{of-int}\ (\beta\ q) * h\ \alpha'\ i)$
by (*subst* (2) *sum.Sigma* [*symmetric*]) (*auto simp*: *case-prod-unfold*)
finally have *J-eq'*: $J\ \alpha = -(\sum ((q,\ \alpha'),\ i){\in}\textit{Sigma}\ P\ \textit{Roots} \times \{..{<}n * p\}.\ \textit{of-int}\ (\beta\ q) * h\ \alpha'\ i)$ **.**

have *h-α-pm1-eq*: $h\ \alpha\ (p{-}1) = \textit{of-int}\ (l\hat{\ }(n{*}p)) * \textit{fact}\ (p{-}1) * (\prod \alpha'{\in}\textit{Roots}'{-}\{\alpha\}.\ (\alpha{-}\alpha')\hat{\ }p)$
proof −

```
    have h α (p − 1) = of-int (l ^ (n * p)) *
          poly ((pderiv ^^ (p−1)) (∏α′∈Roots′. [:−α′,1:] ^ (if α′ = α then p
− 1 else p))) α
      unfolding h-def f-poly-altdef[OF α] higher-pderiv-smult poly-smult ..
    also have (∏α′∈Roots′. [:−α′,1:] ^ (if α′ = α then p − 1 else p)) =
          [:−α,1:]^(p−1) * (∏α′∈Roots′−{α}. [:−α′,1:]^p)
      using α by (subst prod.If-eq) auto
    also have poly ((pderiv ^^ (p−1)) ...) α = fact (p − 1) * (∏α′∈Roots′ −
{α}. (α − α′) ^ p)
      by (subst poly-higher-pderiv-aux2) (simp-all add: poly-prod)
    finally show ?thesis by (simp only: mult.assoc)
  qed

  have fact (p−1) alg-dvd h α (p−1)
  proof −
    have fact (p−1) alg-dvd fact (p−1) * (of-int (l^p) * (∏α′∈Roots′−{α}.
(l*α−l*α′)^p))
      by (intro alg-dvd-triv-left algebraic-int-times[of of-int (l^p)]
          algebraic-int-prod algebraic-int-power algebraic-int-diff
          alg-int α algebraic-int-of-int) auto
    also have (∏α′∈Roots′−{α}. (l*α−l*α′)^p) = (∏α′∈Roots′−{α}. of-int
l^p * (α−α′)^p)
      by (subst power-mult-distrib [symmetric]) (simp-all add: algebra-simps)
    also have ... = of-int (l ^ (p * (n−1))) * (∏α′∈Roots′−{α}. (α−α′)^p)
      using α by (subst prod.distrib) (auto simp: card-Diff-subset n-altdef simp
flip: power-mult)
    also have of-int (l^p) * ... = of-int (l^(p+p*(n−1))) * (∏α′∈Roots′−{α}.
(α−α′)^p)
      unfolding mult.assoc [symmetric] power-add [symmetric] of-int-power ..
    also have p + p * (n − 1) = n * p
      using ‹n > 0› by (cases n) (auto simp: mult-ac)
    also have fact (p − 1) * (of-int (l^(n*p)) * (∏α′∈Roots′−{α}. (α−α′)^p))
= h α (p−1)
      unfolding h-α-pm1-eq by (simp add: mult-ac)
    finally show ?thesis .
  qed

  have ¬of-nat p alg-dvd of-int (β q) * h α (p−1)
    unfolding h-α-pm1-eq mult.assoc [symmetric] of-int-mult [symmetric]
  proof
    define r where r = (λα. of-int (l^n) * (∏α′∈Roots′−{α}. α−α′))
    have alg-int-r: algebraic-int (r α) if α ∈ Roots′ for α
    proof −
      have algebraic-int (of-int l * (∏α′∈Roots′−{α}. of-int l * α − of-int l *
α′))
        by (intro algebraic-int-times[OF algebraic-int-of-int] algebraic-int-prod
            algebraic-int-power algebraic-int-diff alg-int that) auto
      also have ... = of-int l * (∏α′∈Roots′−{α}. of-int l * (α − α′))
        by (simp add: algebra-simps flip: power-mult-distrib)
```

```
    also have ... = of-int (l^(1 + (n-1))) * (∏α'∈Roots'-{α}. α - α')
      using that by (simp add: r-def prod.distrib card-Diff-subset
                               n-altdef power-add mult-ac flip: power-mult)
    also have 1 + (n - 1) = n
      using ‹n > 0› by auto
    finally show algebraic-int (r α)
      unfolding r-def .
  qed

  have (∏α'∈Roots'. r α') ∈ ℚ
  proof -
    obtain Root where Root-bij: bij-betw Root {..<n} Roots'
        using ex-bij-betw-nat-finite[OF ‹finite Roots'›] unfolding n-altdef
atLeast0LessThan by metis
    have Root-in-Roots': Root i ∈ Roots' if i < n for i
      using Root-bij that by (auto simp: bij-betw-def)

    define R :: complex mpoly where
      R = (∏i<n. Const (of-int (l^n)) * (∏j∈{..<n}-{i}. Var i - Var j))
    have insertion Root R ∈ ℚ
    proof (rule symmetric-poly-of-roots-in-subring)
      show symmetric-mpoly {..<n} R
        unfolding R-def
      proof (rule symmetric-mpoly-symmetric-prod'[of - λπ. π], goal-cases)
        case (2 i π)
        from ‹π permutes {..<n}› have [simp]: bij π
          by (rule permutes-bij)
        have mpoly-map-vars π (Const (of-int (l ^ n)) *
              (∏j∈{..<n} - {i}. Var i - Var j):: complex mpoly) =
            Const (of-int l ^ n) * (∏j∈{..<n} - {i}. Var (π i) - Var (π j))
          by simp
        also have (∏j∈{..<n} - {i}. Var (π i) - Var (π j)) =
                  (∏j∈{..<n} - {π i}. Var (π i) - Var j)
          using 2 permutes-in-image[OF 2(2), of i]
            by (intro prod.reindex-bij-betw bij-betw-Diff permutes-imp-bij[OF
2(2)])
               (auto simp: bij-betw-singleton)
        finally show ?case by simp
      qed
    next
      show vars R ⊆ {..<n} unfolding R-def
       by (intro order.trans[OF vars-prod] UN-least order.trans[OF vars-mult]
                 Un-least order.trans[OF vars-power] order.trans[OF vars-diff])
          (auto simp: vars-Var)
    next
      show ring-closed (ℚ :: complex set)
        by unfold-locales auto
      then interpret ring-closed ℚ :: complex set .
      show ∀m. MPoly-Type.coeff R m ∈ ℚ
```

```
      unfolding R-def
      by (intro allI coeff-prod-closed coeff-mult-closed coeff-power-closed)
        (auto simp: mpoly-coeff-Const coeff-Var when-def)
    next
      let ?lc = of-int (∏ p∈P. Polynomial.lead-coeff p) :: complex
      have (∏ q∈P. of-int-poly q) = (∏ q∈P. Polynomial.smult
          (of-int (Polynomial.lead-coeff q)) (∏ x∈Roots q. [:−x, 1:])))
        by (intro prod.cong of-int-poly-P refl)
      also have ... = Polynomial.smult ?lc (∏ q∈P. ∏ x∈Roots q. [:−x, 1:])
        by (simp add: prod-smult)
      also have (∏ q∈P. ∏ x∈Roots q. [:−x, 1:]) = (∏ x∈Roots'. [:−x, 1:])
        unfolding Roots'-def using disjoint
      by (intro prod.UNION-disjoint [symmetric]) (auto simp: disjoint-family-on-def)
      also have ... = (∏ i<n. [:− Root i, 1:])
        by (intro prod.reindex-bij-betw [symmetric] Root-bij)
      finally show of-int-poly (∏ P) = Polynomial.smult ?lc (∏ i<n. [:− Root
i, 1:])
        by (simp add: of-int-poly-hom.hom-prod)
      have prod Polynomial.lead-coeff P ≠ 0
        by (intro prod-nonzeroI) auto
      thus inverse ?lc * ?lc = 1 inverse ?lc ∈ ℚ
        by (auto simp: field-simps simp flip: of-int-prod)
    qed auto
    also have insertion Root R = (∏ i<n. of-int (l^n) * (∏ j∈{..<n}−{i}.
Root i − Root j))
        by (simp add: R-def insertion-prod insertion-mult insertion-power
insertion-diff)
    also have ... = (∏ i<n. of-int (l^n) * (∏ α'∈Roots'−{Root i}. Root i −
α'))
    proof (intro prod.cong, goal-cases)
      case (2 i)
      hence (∏ j∈{..<n}−{i}. Root i − Root j) = (∏ α'∈Roots'−{Root i}.
Root i − α')
        by (intro prod.reindex-bij-betw bij-betw-Diff Root-bij)
          (auto intro: Root-in-Roots' simp: bij-betw-singleton)
      thus ?case by simp
    qed auto
    also have ... = (∏ α'∈Roots'. r α')
      unfolding r-def by (intro prod.reindex-bij-betw Root-bij)
    finally show (∏ α'∈Roots'. r α') ∈ ℚ .
  qed
  moreover have algebraic-int (∏ α'∈Roots'. r α')
    by (intro algebraic-int-prod alg-int-r)
  ultimately have is-int: (∏ α'∈Roots'. r α') ∈ ℤ
    using rational-algebraic-int-is-int by blast
  then obtain R' where R': (∏ α'∈Roots'. r α') = of-int R'
    by (elim Ints-cases)
  have (∏ α'∈Roots'. r α') ≠ 0
    using ‹l > 0› by (intro prod-nonzeroI) (auto simp: r-def ‹finite Roots'›)
```

with R' **have** [*simp*]: $R' \neq 0$
by *auto*

assume *of-nat p alg-dvd of-int* (β $q * l\hat{}(n*p)$) $*$ *fact* $(p-1) * (\prod \alpha' \in Roots' - \{\alpha\}.$
$(\alpha - \alpha')\ \hat{}\ p)$
also have $\ldots$ $=$ *of-int* (β q) $*$ *fact* $(p-1) * r\ \alpha\ \hat{}\ p$
by (*simp add*: *r-def mult-ac power-mult-distrib power-mult prod-power-distrib*)
also have $\ldots$ *alg-dvd of-int* (β q) $*$ *fact* $(p-1) * r\ \alpha\ \hat{}\ p * (\prod \alpha' \in Roots' - \{\alpha\}.$
$r\ \alpha')\ \hat{}\ p$
by (*intro alg-dvd-triv-left algebraic-int-prod alg-int-r algebraic-int-power*)
auto
also have $\ldots$ $=$ *of-int* (β q) $*$ *fact* $(p-1) * (\prod \alpha' \in Roots'.\ r\ \alpha')\ \hat{}\ p$
using α **by** (*subst* (*2*) *prod.remove*[*of* - α]) (*auto simp*: *mult-ac power-mult-distrib*)
also have $\ldots$ $=$ *of-int* (β $q * fact\ (p - 1) * R'\ \hat{}\ p$)
by (*simp add*: R')
also have *of-nat p* $=$ *of-int* (*int p*)
by *simp*
finally have *int p dvd* β $q * fact\ (p - 1) * R'\ \hat{}\ p$
by (*subst* (*asm*) *alg-dvd-of-int-iff*)
moreover have *prime* (*int p*)
using ‹*prime p*› **by** *auto*
ultimately have *int p dvd* β $q \vee$ *int p dvd fact* $(p - 1) \vee$ *int p dvd* $R'\ \hat{}\ p$
by (*simp add*: *prime-dvd-mult-iff*)
moreover have $\neg$*int p dvd* β q
proof
assume *int p dvd* β q
hence *int p* $\leq |\beta\ q|$
using β-*nz*[*of q*] *dvd-imp-le-int*[*of* β q *int p*] q **by** *auto*
with *p-ineqs*(*1*) q **show** *False* **by** *auto*
qed
moreover have $\neg$*int p dvd fact* $(p - 1)$
proof −
have $\neg p$ *dvd fact* $(p - 1)$
using ‹$p > 1$› p **by** (*subst prime-dvd-fact-iff*) *auto*
hence $\neg$*int p dvd int* (*fact* $(p - 1)$)
by (*subst int-dvd-int-iff*)
thus *?thesis* **unfolding** *of-nat-fact* **.**
qed
moreover have $\neg$*int p dvd* $R'\ \hat{}\ p$
proof
assume *int p dvd* $R'\ \hat{}\ p$
hence *int p dvd* R'
using ‹*prime* (*int p*)› *prime-dvd-power* **by** *metis*
hence *int p* $\leq |R'|$
using β-*nz*[*of q*] *dvd-imp-le-int*[*of* R' *int p*] q **by** *auto*
hence *real p* $\leq$ *real-of-int* $|R'|$
by *linarith*
also have $\ldots$ $=$ *norm* $(\prod \alpha \in Roots'.\ r\ \alpha)$
unfolding R' **by** *simp*

```
    finally show False unfolding r-def using p-ineqs(2)
      by linarith
  qed
  ultimately show False
    by blast
qed

have fact-p-dvd: fact p alg-dvd h α' j if α' ∈ Roots' α' ≠ α ∨ j ≠ p − 1 for
α' j
proof (cases j ≥ p)
  case False
  with that have j: j < (if α' = α then p − 1 else p)
    by auto
  have h α' j = 0
    unfolding h-def f-poly-altdef[OF α]
    by (intro poly-higher-pderiv-aux1'[OF j] dvd-smult dvd-prodI that) auto
  thus ?thesis by simp
next
  case True
  define e where e = (λx. if x = α then p − 1 else p)
  define Q where Q = (∏ x∈Roots'. [:−x, 1:] ^ e x)
  define Q' where Q' = Polynomial.smult (of-int (l^(n*p+j))) (pcompose Q
[:0, 1 / of-int l:])
  have poly ((pderiv ^^ j) Q) α' / l ^ j =
        poly ((pderiv ^^ j) (pcompose Q [:0, 1 / of-int l:])) (l * α')
    using ‹l > 0› by (simp add: higher-pderiv-pcompose-linear poly-pcompose
field-simps)

  have sum e Roots' = (n − 1) * p + (p − 1)
    unfolding e-def using α
    by (subst sum.If-eq) (auto simp: card-Diff-subset n-altdef algebra-simps)
  also have ... = n * p − 1
    using ‹n > 0› ‹p > 1› by (cases n) auto
  finally have [simp]: sum e Roots' = n * p − 1 .

  have h α' j = of-int (l^(n*p)) * poly ((pderiv ^^ j) Q) α'
    unfolding h-def f-poly-altdef[OF α] higher-pderiv-smult poly-smult e-def
Q-def ..
  also have poly ((pderiv ^^ j) Q) α' =
              of-int l ^ j * poly ((pderiv ^^ j) (pcompose Q [:0, 1 / of-int l:]))
(l * α')
    using ‹l > 0› by (simp add: higher-pderiv-pcompose-linear poly-pcompose
field-simps)
  also have of-int (l ^ (n * p)) * ... = poly ((pderiv ^^ j) Q') (l * α')
    by (simp add: Q'-def higher-pderiv-smult power-add)
  also have fact p alg-dvd ...
  proof (rule fact-alg-dvd-poly-higher-pderiv)
    show j ≥ p by fact
    show algebraic-int (of-int l * α')
```

by (*rule alg-int*) *fact*
interpret *alg-int*: *ring-closed* {*x*::*complex*. *algebraic-int x*}
by *standard auto*
show *algebraic-int* (*poly.coeff Q' i*) **for** *i*
proof (*cases i* ≤ *Polynomial.degree Q'*)
case *False*
thus *?thesis*
by (*simp add*: *coeff-eq-0*)
next
case *True*
hence *i* ≤ *n* ∗ *p* − *1* **using** ‹*l* > *0*›
by (*simp add*: *Q'-def degree-prod-eq Q-def degree-power-eq*)
also have *n* ∗ *p* > *0*
using ‹*n* > *0*› ‹*p* > *1*› **by** *auto*
hence *n* ∗ *p* − *1* < *n* ∗ *p*
by *simp*
finally have *i*: *i* < *n* ∗ *p* **.**

have *poly.coeff Q' i* = *of-int l* ^ (*n* ∗ *p* + *j*) / *of-int l* ^ *i* ∗ *poly.coeff Q i*
by (*simp add*: *Q'-def coeff-pcompose-linear field-simps*)
also have *of-int l* ^ (*n* ∗ *p* + *j*) = (*of-int l* ^ (*n* ∗ *p* + *j* − *i*) :: *complex*) ∗ *of-int l* ^ *i*
unfolding *power-add* [*symmetric*] **using** *i* **by** *simp*
hence *of-int l* ^ (*n* ∗ *p* + *j*) / *of-int l* ^ *i* = (*of-int l* ^ (*n* ∗ *p* + *j* − *i*) :: *complex*)
using ‹*l* > *0*› **by** (*simp add*: *field-simps*)
also have ... ∗ *poly.coeff Q i* =
(∑ *X*∈{*X*. *X* ⊆ (*SIGMA x*:*Roots'*. {..<*e x*}) ∧ *i* = *n* ∗ *p* − *Suc* (*card X*)}.
of-int l ^ (*n* ∗ *p* + *j* − (*n* ∗ *p* − *Suc* (*card X*))) ∗ ((− *1*) ^ *card X* ∗ *prod fst X*))
unfolding *Q-def* **by** (*subst coeff-prod-linear-factors*) (*auto simp*: *sum-distrib-left*)
also have *algebraic-int* ...
proof (*intro algebraic-int-sum*, *goal-cases*)
case (*1 X*)
hence *X*: *X* ⊆ (*SIGMA x*:*Roots'*. {..<*e x*})
by *auto*
have *card-eq*: *card* (*SIGMA x*:*Roots'*. {..<*e x*}) = *n* ∗ *p* − *1*
by (*subst card-SigmaI*) *auto*
from *X* **have** *card X* ≤ *card* (*SIGMA x*:*Roots'*. {..<*e x*})
by (*intro card-mono*) *auto*
hence *card X* ≤ *n* ∗ *p* − *1*
using *card-eq* **by** *auto*
also have ... < *n* ∗ *p*
using ‹*n* ∗ *p* > *0*› **by** *simp*
finally have *card-less*: *card X* < *n* ∗ *p* **.**
have *algebraic-int* ((−*1*) ^ *card X* ∗ *of-int l* ^ (*j* + *1*) ∗ (∏ *x*∈*X*. *of-int l* ∗ *fst x*))

```
          using X by (intro algebraic-int-times algebraic-int-prod alg-int) auto
        thus ?case
          using card-less by (simp add: power-add prod.distrib mult-ac)
      qed
      finally show ?thesis .
    qed
  qed
  finally show ?thesis .
qed

have p-dvd: of-nat p alg-dvd h α' j if α' ∈ Roots' α' ≠ α ∨ j ≠ p − 1 for α' j
proof −
  have of-nat p alg-dvd (of-nat (fact p) :: complex)
    by (intro alg-dvd-of-nat dvd-fact) (use ‹p > 1› in auto)
  hence of-nat p alg-dvd (fact p :: complex)
    by simp
  also have ... alg-dvd h α' j
    using that by (intro fact-p-dvd)
  finally show ?thesis .
qed

show fact (p − 1) alg-dvd J α
  unfolding J-eq'
proof (intro alg-dvd-uminus-right alg-dvd-sum, safe intro!: alg-dvd-mult algebraic-int-of-int)
  fix q α' j
  assume q ∈ P α' ∈ Roots q j < n * p
  hence α' ∈ Roots'
    by (auto simp: Roots'-def)
  show fact (p − 1) alg-dvd h α' j
  proof (cases α' = α ∧ j = p − 1)
    case True
    thus ?thesis using ‹fact (p − 1) alg-dvd h α (p − 1)›
      by simp
  next
    case False
    have of-int (fact (p − 1)) alg-dvd (of-int (fact p) :: complex)
      by (intro alg-dvd-of-int fact-dvd) auto
    hence fact (p − 1) alg-dvd (fact p :: complex)
      by simp
    also have ... alg-dvd h α' j
      using False ‹α' ∈ Roots'› by (intro fact-p-dvd) auto
    finally show ?thesis .
  qed
qed

show ¬of-nat p alg-dvd J α
  unfolding J-eq' alg-dvd-uminus-right-iff
```

```
    proof (rule not-alg-dvd-sum)
      have p − 1 < 1 ∗ p
        using ‹p > 1› by simp
      also have 1 ∗ p ≤ n ∗ p
        using ‹n > 0› by (intro mult-right-mono) auto
      finally show ((q, α), p − 1) ∈ Sigma P Roots × {..<n∗p}
        using q ‹n > 0› by auto
    next
      fix z assume z: z ∈ Sigma P Roots × {..<n∗p}−{((q,α),p−1)}
      from z have snd (fst z) ∈ Roots′
        by (auto simp: Roots′-def)
      moreover have fst (fst z) = q if α ∈ Roots (fst (fst z))
      proof −
        have α ∈ Roots (fst (fst z)) ∩ Roots q q ∈ P fst (fst z) ∈ P
          using that q z by auto
        with disjoint show ?thesis
          unfolding disjoint-family-on-def by blast
      qed
      ultimately have of-nat p alg-dvd h (snd (fst z)) (snd z)
        using z by (intro p-dvd) auto
      thus of-nat p alg-dvd (case z of (x, xa) ⇒ (case x of (q, α′) ⇒ λi. of-int
(β q) ∗ h α′ i) xa)
        using z by auto
    qed (use ‹¬of-nat p alg-dvd of-int (β q) ∗ h α (p−1)› in auto)
  qed
```

Our next goal is to show that J is rational. This is done by repeated applications of the fundamental theorem of symmetric polynomials, exploiting the fact that J is symmetric in all the α_i for each set of conjugates.

```
  define g :: int poly poly
    where g = synthetic-div (map-poly (λx. [:x:])
              ((Polynomial.smult lc-factor (∏ P)) ^ p)) [:0, 1:]
  have g: map-poly (λp. ipoly p α) g = f-poly α if α: α ∈ Roots′ for α
  proof −
    interpret α: comm-ring-hom λp. ipoly p α
    by standard (auto simp: of-int-hom.poly-map-poly-eval-poly of-int-poly-hom.hom-mult)
    define Q :: int poly where Q = (Polynomial.smult lc-factor (∏ P)) ^ p
  have f-poly α = Polynomial.smult (of-int (l^(n∗p))) ((∏ α′∈Roots′. [:−α′,1:]) ^p)
div [:−α,1:]
    unfolding f-poly-def div-smult-left [symmetric] prod-power-distrib[symmetric]
..
    also have of-int (l^(n∗p)) = (of-int l^n)^p
      by (simp add: power-mult)
    also have Polynomial.smult ... ((∏ α′∈Roots′. [:−α′,1:])^p) =
              (Polynomial.smult (of-int l ^ n) (∏ α′∈Roots′. [:−α′,1:])) ^ p
      by (simp only: smult-power)
    also have ... = of-int-poly Q
      by (subst lc-factor) (simp-all add: Q-def of-int-poly-hom.hom-power)
    also have ... div [:−α, 1:] = synthetic-div (of-int-poly Q) α
```

```
      unfolding synthetic-div-altdef ..
    also have ... = synthetic-div (map-poly (λp. ipoly p α) (map-poly (λx. [:x:])
Q)) (ipoly [:0, 1:] α)
      by (simp add: map-poly-map-poly o-def)
    also have ... = map-poly (λp. ipoly p α) g
      unfolding g-def Q-def by (rule α.synthetic-div-hom)
    finally show ?thesis ..
  qed

  obtain Q where Q: J α = −(∑ q∈P. of-int (β q) ∗ eval-poly of-rat (Q q) α)
    if α ∈ Roots' for α
  proof −
    define g' :: nat ⇒ complex poly poly
      where g' = (λj. (map-poly of-int-poly ((pderiv ^^ j) g)))
    obtain root :: int poly ⇒ nat ⇒ complex
      where root: ⋀q. q ∈ P ⟹ bij-betw (root q) {..<♯q} (Roots q)
    using ex-bij-betw-nat-finite[OF finite-Roots] unfolding n-roots-def atLeast0LessThan
      by metis
    have ∃ Q'. map-poly of-rat Q' = (∑ x∈Roots q. poly (g' j) [:x:]) if q: q ∈ P
for q j
    proof −
      define Q :: nat ⇒ complex poly mpoly
        where Q = (λj. (∑ i<♯q. mpoly-of-poly i (g' j)))
      define ratpolys :: complex poly set where ratpolys = {p. ∀ i. poly.coeff p i
∈ ℚ}
      have insertion ((λx. [:x:]) ∘ root q) (Q j) ∈ ratpolys
      proof (rule symmetric-poly-of-roots-in-subring)
        show ring-closed ratpolys
          by standard (auto simp: ratpolys-def intro!: coeff-mult-semiring-closed)
        show ∀ m. MPoly-Type.coeff (Q j) m ∈ ratpolys
        by (auto simp: Q-def ratpolys-def Polynomial.coeff-sum coeff-mpoly-of-poly
when-def g'-def
                  intro!: sum-in-Rats)
        show vars (Q j) ⊆ {..<♯q} unfolding Q-def
      by (intro order.trans[OF vars-sum] UN-least order.trans[OF vars-mpoly-of-poly])
auto
        show symmetric-mpoly {..<♯q} (Q j) unfolding Q-def
            by (rule symmetric-mpoly-symmetric-sum[of - id]) (auto simp: per-
mutes-bij)
        interpret coeff-lift-hom: map-poly-idom-hom λx. [:x:]
          by standard
        define lc :: complex where lc = of-int (Polynomial.lead-coeff q)
      have of-int-poly q = Polynomial.smult (Polynomial.lead-coeff q) (∏ x∈Roots
q. [:−x, 1:])
          by (rule of-int-poly-P) fact
        also have poly-lift ... = Polynomial.smult [:lc:] (∏ a∈Roots q. [:−[:a:],
1:])
          by (simp add: poly-lift-def map-poly-smult coeff-lift-hom.hom-prod lc-def)
        also have (∏ a∈Roots q. [:−[:a:], 1:]) = (∏ i<♯q. [:−[:root q i:], 1:])
```

```
        by (intro prod.reindex-bij-betw [symmetric] root q)
      also have ... = (∏ i<♯q. [:− ((λx. [:x:]) ∘ root q) i, 1:])
        by simp
      finally show poly-lift (Ring-Hom-Poly.of-int-poly q) = Polynomial.smult
[:lc:] ... .
      have lc ≠ 0
        using q by (auto simp: lc-def)
      thus [:inverse lc:] ∗ [:lc:] = 1
        by (simp add: field-simps)
    qed (auto simp: ratpolys-def coeff-pCons split: nat.splits)

    also have insertion ((λx. [:x:]) ∘ root q) (Q j) = (∑ i<♯q. poly (g' j) [:root
q i:])
      by (simp add: Q-def insertion-sum poly-sum)
    also have ... = (∑ x∈Roots q. poly (g' j) [:x:])
      by (intro sum.reindex-bij-betw root q)
    finally have ∀ i. poly.coeff (∑ x∈Roots q. poly (g' j) [:x:]) i ∈ ℚ
      by (auto simp: ratpolys-def)
    thus ?thesis
      using ratpolyE by metis
  qed
    then obtain Q where Q: ⋀q j. q ∈ P ⟹ map-poly of-rat (Q q j) =
(∑ x∈Roots q. poly (g' j) [:x:])
    by metis
  define Q' where Q' = (λq. ∑ j<n∗p. Q q j)

  have J α = − (∑ q∈P. of-int (β q) ∗ eval-poly of-rat (Q' q) α) if α: α ∈
Roots' for α
  proof −
  have J α = −(∑ q∈P. of-int (β q) ∗ (∑ x∈Roots q. ∑ j<n ∗ p. poly ((pderiv
^^ j) (f-poly α)) x))
      (is - = −?S) unfolding J-eq[OF α] ..
    also have ?S = (∑ q∈P. of-int (β q) ∗ eval-poly of-rat (Q' q) α)
    proof (rule sum.cong, goal-cases)
      case q: (2 q)
      interpret α: idom-hom λp. ipoly p α
    by standard (auto simp: of-int-hom.poly-map-poly-eval-poly of-int-poly-hom.hom-mult)

      have (∑ x∈Roots q. ∑ j<n ∗ p. poly ((pderiv ^^ j) (f-poly α)) x) =
          (∑ j<n ∗ p. ∑ x∈Roots q. poly ((pderiv ^^ j) (f-poly α)) x)
        by (rule sum.swap)
      also have ... = (∑ j<n ∗ p. eval-poly of-rat (Q q j) α)
      proof (rule sum.cong, goal-cases)
        case j: (2 j)
        have (∑ x∈Roots q. poly ((pderiv ^^ j) (f-poly α)) x) =
            (∑ x∈Roots q. poly (poly (g' j) [:x:]) α)
        proof (rule sum.cong, goal-cases)
          case (2 x)
          have poly ((pderiv ^^ j) (f-poly α)) x =
```

```
        poly ((pderiv ^^ j) (map-poly (λp. ipoly p α) g)) x
      by (subst g[OF α, symmetric]) (rule refl)
    also have ... = poly (eval-poly ((λp. [:poly p α:]) ∘ of-int-poly) ((pderiv
^^ j) g) [:0, 1:]) x
      unfolding o-def α.map-poly-higher-pderiv [symmetric]
      by (simp only: α.map-poly-eval-poly)
    also have ... = poly (eval-poly (λp. [:poly p α:])
                    (map-poly of-int-poly ((pderiv ^^ j) g)) [:0, 1:]) x
      unfolding eval-poly-def by (subst map-poly-map-poly) auto
    also have ... = poly (poly (map-poly of-int-poly ((pderiv ^^ j) g)) [:x:])
α
      by (rule poly-poly-eq [symmetric])
    also have ... = poly (poly (g' j) [:x:]) α
      by (simp add: g'-def)
    finally show ?case .
  qed auto
  also have ... = poly (∑x∈Roots q. poly (g' j) [:x:]) α
    by (simp add: poly-sum)
  also have ... = eval-poly of-rat (Q q j) α
    using q by (simp add: Q eval-poly-def)
  finally show ?case .
 qed auto
 also have ... = eval-poly of-rat (Q' q) α
  by (simp add: Q'-def of-rat-hom.eval-poly-sum)
 finally show ?case by simp
qed auto
finally show J α = - (∑q∈P. of-int (β q) * eval-poly of-rat (Q' q) α) .
qed
thus ?thesis using that[of Q'] by metis
qed

have J' ∈ ℚ
proof -
 have (∏α∈Roots q. J α) ∈ ℚ if q: q ∈ P for q
 proof -
  obtain root where root: bij-betw root {..<♯q} (Roots q)
   using ex-bij-betw-nat-finite[OF finite-Roots[OF q]]
   unfolding atLeast0LessThan n-roots-def by metis
  define Q' :: complex poly
   where Q' = -(∑q∈P. Polynomial.smult (of-int (β q)) (map-poly of-rat
(Q q)))

  have (∏α∈Roots q. J α) = (∏α∈Roots q. -(∑q∈P. of-int (β q) * eval-poly
of-rat (Q q) α))
   by (intro prod.cong refl Q) (auto simp: Roots'-def q)
  also have ... = (∏α∈Roots q. poly Q' α)
   by (simp add: Q'-def poly-sum eval-poly-def)
  also have ... = (∏i<♯q. poly Q' (root i))
   by (intro prod.reindex-bij-betw [symmetric] root)
```

```
    also have ... = insertion root (∏ i<♯q. mpoly-of-poly i Q')
      by (simp add: insertion-prod)
    also have ... ∈ ℚ
    proof (rule symmetric-poly-of-roots-in-subring)
      show ring-closed (ℚ :: complex set)
        by standard auto
      then interpret Q: ring-closed ℚ :: complex set .
      show ∀ m. MPoly-Type.coeff (∏ i<♯q. mpoly-of-poly i Q') m ∈ ℚ
        by (auto intro!: Q.coeff-prod-closed sum-in-Rats
              simp: coeff-mpoly-of-poly when-def Q'-def Polynomial.coeff-sum)
      show symmetric-mpoly {..<♯q} (∏ i<♯q. mpoly-of-poly i Q')
         by (intro symmetric-mpoly-symmetric-prod'[of - id]) (auto simp: per-
mutes-bij)
      show vars (∏ i<♯q. mpoly-of-poly i Q') ⊆ {..<♯q}
        by (intro order.trans[OF vars-prod] order.trans[OF vars-mpoly-of-poly]
UN-least) auto
      define lc where lc = (of-int (Polynomial.lead-coeff q) :: complex)
      have of-int-poly q = Polynomial.smult lc (∏ x∈Roots q. [:- x, 1:])
        unfolding lc-def by (rule of-int-poly-P) fact
      also have (∏ x∈Roots q. [:- x, 1:]) = (∏ i<♯q. [:- root i, 1:])
        by (intro prod.reindex-bij-betw [symmetric] root)
      finally show of-int-poly q = Polynomial.smult lc (∏ i<♯q. [:- root i, 1:])
.
      have lc ≠ 0
        using q by (auto simp: lc-def)
      thus inverse lc * lc = 1 inverse lc ∈ ℚ
        by (auto simp: lc-def)
    qed auto
    finally show ?thesis .
  qed
  hence (∏ q∈P. ∏ α∈Roots q. J α) ∈ ℚ
    by (rule prod-in-Rats)
  also have (∏ q∈P. ∏ α∈Roots q. J α) = J'
    unfolding Roots'-def J'-def using disjoint
  by (intro prod.UNION-disjoint [symmetric]) (auto simp: disjoint-family-on-def)
  finally show J' ∈ ℚ .
qed
```

Since J' is clearly an algebraic integer, we now know that it is in fact an integer.

```
moreover have algebraic-int J'
  unfolding J'-def
proof (intro algebraic-int-prod)
  fix x assume x ∈ Roots'
  hence fact (p - 1) alg-dvd J x
    by (intro J)
  thus algebraic-int (J x)
    by (rule alg-dvd-imp-algebraic-int) auto
qed
```

ultimately have $J' \in \mathbb{Z}$
using *rational-algebraic-int-is-int* **by** *blast*

It is also non-zero, as none of the J_i have p as a factor and such cannot be zero.

have $J' \neq 0$
unfolding *J'-def*
proof (*intro prod-nonzeroI*)
fix α **assume** $\alpha \in$ *Roots'*
hence $\neg$*of-nat p alg-dvd J* α
using *J*(*2*)[*of* α] **by** *auto*
thus *J* $\alpha \neq 0$
by *auto*
qed

It then clearly follows that $(p-1)!^n \leq J$:

have *fact* (*p* − *1*) ^ *n alg-dvd J'*
proof −
have *fact* (*p* − *1*) ^ *n* = (∏ α∈*Roots'*. *fact* (*p* − *1*))
by (*simp add*: *n-altdef*)
also have ... *alg-dvd J'*
unfolding *J'-def* **by** (*intro prod-alg-dvd-prod J*(*1*))
finally show *?thesis* **.**
qed

have *fact* (*p* − *1*) ^ *n* ≤ *norm J'*
proof −
from ‹*J'* ∈ ℤ› **obtain** *J''* **where** [*simp*]: *J'* = *of-int J''*
by (*elim Ints-cases*)
have *of-int* (*fact* (*p* − *1*) ^ *n*) = (*fact* (*p* − *1*) ^ *n* :: *complex*)
by *simp*
also have ... *alg-dvd J'*
by *fact*
also have *J'* = *of-int J''*
by *fact*
finally have *fact* (*p* − *1*) ^ *n dvd J''*
by (*subst* (*asm*) *alg-dvd-of-int-iff*)
moreover from ‹*J'* ≠ *0*› **have** *J''* ≠ *0*
by *auto*
ultimately have |*J''*| ≥ |*fact* (*p* − *1*) ^ *n*|
by (*intro dvd-imp-le-int*)
hence *real-of-int* |*J''*| ≥ *real-of-int* |*fact* (*p* − *1*) ^ *n*|
by *linarith*
also have *real-of-int* |*J''*| = *norm J'*
by *simp*
finally show *?thesis*
by *simp*
qed

The standard M-L bound for $I_i(x)$ shows the following inequality:

```
  also have norm J' ≤ C' * C p ^ n
  proof −
    have norm J' = (∏ x∈Roots'. norm (J x))
      unfolding J'-def prod-norm [symmetric] ..
   also have ... ≤ (∏ x∈Roots'. ∑ q∈P. real-of-int |β q| * (∑ α∈Roots q. cmod
α * exp (cmod α) * C p))
    proof (intro prod-mono conjI)
      fix x assume x: x ∈ Roots'
       show norm (J x) ≤ (∑ q∈P. real-of-int |β q| * (∑ α∈Roots q. norm α *
exp (norm α) * C p))
        unfolding J-def
       proof (intro sum-norm-le)
        fix q assume q ∈ P
        show norm (of-int (β q) * sum (I x) (Roots q)) ≤
              real-of-int |β q| * (∑ α∈Roots q. norm α * exp (norm α) * C p)
          unfolding norm-mult norm-of-int of-int-abs
        proof (intro mult-left-mono sum-norm-le)
          fix α assume α ∈ Roots q
          hence α: α ∈ Roots'
            using ‹q ∈ P› by (auto simp: Roots'-def)
          show norm (I x α) ≤ norm α * exp (norm α) * C p
            unfolding I-def
       proof (intro lindemann-weierstrass-aux.lindemann-weierstrass-integral-bound)
          fix t assume t ∈ closed-segment 0 α
          also have closed-segment 0 α ⊆ cball 0 R
            using ‹R ≥ 0› R-ge[OF α] by (intro closed-segment-subset) auto
          finally have norm t ≤ R by simp

          have norm-diff-le: norm (t − y) ≤ 2 * R if y ∈ Roots' for y
          proof −
            have norm (t − y) ≤ norm t + norm y
              by (meson norm-triangle-ineq4)
            also have ... ≤ R + R
              by (intro add-mono[OF ‹norm t ≤ R› R-ge] that)
            finally show ?thesis by simp
          qed

          have norm (poly (f-poly x) t) =
                |real-of-int l| ^ (n * p) * (∏ y∈Roots'. cmod (t − y) ^ (if y = x
then p − 1 else p))
            by (simp add: eval-f x f-def norm-mult norm-power flip: prod-norm)
          also have ... ≤ |real-of-int l| ^ (n * p) * (∏ y∈Roots'. (2*R) ^ (if y
= x then p − 1 else p))
           by (intro mult-left-mono prod-mono conjI power-mono norm-diff-le)
auto
             also have ... = |real-of-int l|^(n*p) * (2^(p−1) * R^(p−1) *
(2^p*R^p)^(n−1))
          using x by (subst prod.If-eq) (auto simp: card-Diff-subset n-altdef)
```

```
      also have 2^(p-1) * R^(p-1) * (2^p*R^p)^(n-1) = (2^((p-1)+p*(n-1)))
* (R^((p-1)+p*(n-1)))
        unfolding power-mult power-mult-distrib power-add by (simp add:
mult-ac)
       also have (p-1)+p*(n-1) = p*n - 1
         using ‹n > 0› ‹p > 1› by (cases n) (auto simp: algebra-simps)
       also have 2 ^ (p * n - 1) * R ^ (p * n - 1) = (2*R)^(n * p-1)
         unfolding power-mult-distrib by (simp add: mult-ac)
       finally show norm (poly (f-poly x) t) ≤ C p
         unfolding C-def using ‹l > 0› by simp
     qed (use ‹R ≥ 0› ‹l > 0› in ‹auto simp: C-def›)
    qed auto
   qed
  qed auto
  also have ... = C' * C p ^ n
    by (simp add: C'-def power-mult-distrib n-altdef flip: sum-distrib-right
mult.assoc)
  finally show ?thesis .
 qed
```

And with that, we have our inequality:

```
  finally show fact (p - 1) ^ n ≤ C' * C p ^ n .
qed
```

Some simple asymptotic estimates show that this is clearly a contradiction, since the left-hand side grows much faster than the right-hand side and there are infinitely many sufficiently large primes:

```
have freq: frequently prime sequentially
  using frequently-prime-cofinite unfolding cofinite-eq-sequentially .
have ev: eventually (λp. (∀ q∈P.  int p > |β q|) ∧
        real p > norm (∏ α∈Roots'. of-int (l^n) * (∏ α'∈Roots'-{α}. (α-α')))
sequentially
  by (intro eventually-ball-finite ‹finite P› ballI eventually-conj filterlim-real-sequentially
      eventually-compose-filterlim[OF eventually-gt-at-top] filterlim-int-sequentially)

have frequently (λp. fact (p - 1) ^ n ≤ C' * C p ^ n) sequentially
  by (rule frequently-eventually-mono[OF freq ev]) (use ineq in blast)
moreover have eventually (λp. fact (p - 1) ^ n > C' * C p ^ n) sequentially
proof (cases R = 0)
  case True
  have eventually (λp. p * n > 1) at-top using ‹n > 0›
   by (intro eventually-compose-filterlim[OF eventually-gt-at-top] mult-nat-right-at-top)
  thus ?thesis
    by eventually-elim (use ‹n > 0› True in ‹auto simp: C-def power-0-left
mult-ac›)
next
  case False
  hence R > 0
    using ‹R ≥ 0› by auto
```

define $D :: real$ **where** $D = (2 * R * |real\text{-}of\text{-}int\ l|)\ \hat{}\ n$
have $D > 0$
using ‹$R > 0$› ‹$l > 0$› **unfolding** *D-def* **by** (*intro zero-less-power*) *auto*

have $(\lambda p.\ C' * C\ p\ \hat{}\ n) \in O(\lambda p.\ C\ p\ \hat{}\ n)$
by *simp*
also have $(\lambda p.\ C\ p\ \hat{}\ n) \in O(\lambda p.\ ((2 * R * l)\ \hat{}\ (n * p))\ \hat{}\ n)$
proof (*rule landau-o.big-power*[*OF bigthetaD1*])
have *np*: *eventually* $(\lambda p.\ p * n > 0)$ *at-top* **using** ‹$n > 0$›
by (*intro eventually-compose-filterlim*[*OF eventually-gt-at-top*] *mult-nat-right-at-top*)
have *eventually* $(\lambda p.\ (2 * R) * C\ p = (2 * R * l)\ \hat{}\ (n * p))$ *at-top*
using *np*
proof *eventually-elim*
case (*elim p*)
have $2 * R * C\ p = l\ \hat{}\ (n * p) * (2 * R)\ \hat{}\ (Suc\ (n * p - 1))$
by (*simp add*: *C-def algebra-simps*)
also have $Suc\ (n * p - 1) = n * p$
using *elim* **by** *auto*
finally show *?case*
by (*simp add*: *algebra-simps*)
qed
hence $(\lambda p.\ (2 * R) * C\ p) \in \Theta(\lambda p.\ (2 * R * l)\ \hat{}\ (n * p))$
by (*intro bigthetaI-cong*)
thus $C \in \Theta(\lambda p.\ (2 * R * l)\ \hat{}\ (n * p))$
using ‹$R > 0$› **by** *simp*
qed
also have $\ldots = O(\lambda p.\ (D\ \hat{}\ p)\ \hat{}\ n)$
using ‹$l > 0$› **by** (*simp flip*: *power-mult add*: *power2-eq-square mult-ac D-def*)
also have $(\lambda p.\ (D\ \hat{}\ p)\ \hat{}\ n) \in o(\lambda p.\ fact\ (p - 1)\ \hat{}\ n)$
proof (*intro landau-o.small-power*)
have *eventually* $(\lambda p.\ D\ \hat{}\ p = D * D\ \hat{}\ (p - 1))$ *at-top*
using *eventually-gt-at-top*[*of 0*]
by *eventually-elim* (*use* ‹$D > 0$› **in** ‹*auto simp flip*: *power-Suc*›)
hence $(\lambda p.\ D\ \hat{}\ p) \in \Theta(\lambda p.\ D * D\ \hat{}\ (p - 1))$
by (*intro bigthetaI-cong*)
hence $(\lambda p.\ D\ \hat{}\ p) \in \Theta(\lambda p.\ D\ \hat{}\ (p - 1))$
using ‹$D > 0$› **by** *simp*
also have $(\lambda p.\ D\ \hat{}\ (p - 1)) \in o(\lambda p.\ fact\ (p - 1))$
by (*intro smalloI-tendsto*[*OF filterlim-compose*[*OF power-over-fact-tendsto-0*]]
filterlim-minus-nat-at-top) *auto*
finally show $(\lambda p.\ D\ \hat{}\ p) \in o(\lambda x.\ fact\ (x - 1))$ **.**
qed *fact+*
finally have *smallo*: $(\lambda p.\ C' * C\ p\ \hat{}\ n) \in o(\lambda p.\ fact\ (p - 1)\ \hat{}\ n)$ **.**
have *eventually* $(\lambda p.\ |C' * C\ p\ \hat{}\ n| \leq 1/2 * fact\ (p - 1)\ \hat{}\ n)$ *at-top*
using *landau-o.smallD*[*OF smallo*, *of 1/2*] **by** *simp*
thus *eventually* $(\lambda p.\ C' * C\ p\ \hat{}\ n < fact\ (p - 1)\ \hat{}\ n)$ *at-top*
proof *eventually-elim*
case (*elim p*)
have $C' * C\ p\ \hat{}\ n \leq |C' * C\ p\ \hat{}\ n|$

```
      by simp
    also have ... ≤ 1/2 * fact (p - 1) ^ n
      by fact
    also have ... < fact (p - 1) ^ n
      by simp
    finally show ?case .
  qed
 qed
 ultimately have frequently (λp::nat. False) sequentially
  by (rule frequently-eventually-mono) auto
 thus False
  by simp
qed
```

7.2 Removing the restriction of full sets of conjugates

We will now remove the restriction that the α_i must occur in full sets of conjugates by multiplying the equality with all permutations of roots.

```
lemma Hermite-Lindemann-aux2:
 fixes X :: complex set and β :: complex ⇒ int
 assumes finite X
 assumes nz: ⋀x. x ∈ X ⟹ β x ≠ 0
 assumes alg: ⋀x. x ∈ X ⟹ algebraic x
 assumes sum0: (∑ x∈X. of-int (β x) * exp x) = 0
 shows   X = {}
proof (rule ccontr)
 assume X ≠ {}
 note [intro] = ‹finite X›
```

Let P be the smallest integer polynomial whose roots are a superset of X:

```
 define P :: int poly where P = ∏ (min-int-poly ` X)
 define Roots :: complex set where Roots = {x. ipoly P x = 0}
 have [simp]: P ≠ 0
  using ‹finite X› by (auto simp: P-def)
 have [intro]: finite Roots
  unfolding Roots-def by (intro poly-roots-finite) auto

 have X ⊆ Roots
 proof safe
  fix x assume x ∈ X
  hence ipoly (min-int-poly x) x = 0
   by (intro ipoly-min-int-poly alg)
  thus x ∈ Roots
   using ‹finite X› ‹x ∈ X›
   by (auto simp: Roots-def P-def of-int-poly-hom.hom-prod poly-prod)
 qed

 have squarefree (of-int-poly P :: complex poly)
```

```
    unfolding P-def of-int-poly-hom.hom-prod
  proof (rule squarefree-prod-coprime; safe)
    fix x assume x ∈ X
    thus squarefree (of-int-poly (min-int-poly x) :: complex poly)
      by (intro squarefree-of-int-polyI) auto
  next
    fix x y assume xy: x ∈ X y ∈ X min-int-poly x ≠ min-int-poly y
    thus Rings.coprime (of-int-poly (min-int-poly x)) (of-int-poly (min-int-poly y))
:: complex poly)
      by (intro coprime-of-int-polyI[OF primes-coprime]) auto
  qed
```

Since we will need a numbering of these roots, we obtain one:

```
  define n where n = card Roots
  obtain Root where Root: bij-betw Root {..<n} Roots
   using ex-bij-betw-nat-finite[OF ‹finite Roots›] unfolding n-def atLeast0LessThan
by metis
  define unRoot :: complex ⇒ nat where unRoot = inv-into {..<n} Root
  have unRoot: bij-betw unRoot Roots {..<n}
    unfolding unRoot-def by (intro bij-betw-inv-into Root)
  have unRoot-Root [simp]: unRoot (Root i) = i if i < n for i
     unfolding unRoot-def using Root that by (subst inv-into-f-f) (auto simp:
bij-betw-def)
  have Root-unRoot [simp]: Root (unRoot x) = x if x ∈ Roots for x
     unfolding unRoot-def using Root that by (subst f-inv-into-f) (auto simp:
bij-betw-def)
  have [simp, intro]: Root i ∈ Roots if i < n for i
    using Root that by (auto simp: bij-betw-def)
  have [simp, intro]: unRoot x < n if x ∈ Roots for x
    using unRoot that by (auto simp: bij-betw-def)
```

We will also need to convert between permutations of natural numbers less than n and permutations of the roots:

```
  define convert-perm :: (nat ⇒ nat) ⇒ (complex ⇒ complex) where
    convert-perm = (λσ x. if x ∈ Roots then Root (σ (unRoot x)) else x)
  have bij-convert: bij-betw convert-perm {σ. σ permutes {..<n}} {σ. σ permutes
Roots}
    using bij-betw-permutations[OF Root] unfolding convert-perm-def unRoot-def
.
  have permutes-convert-perm [intro]: convert-perm σ permutes Roots if σ permutes
{..<n} for σ
    using that bij-convert unfolding bij-betw-def by blast
   have convert-perm-compose: convert-perm (π ∘ σ) = convert-perm π ∘ con-
vert-perm σ
    if π permutes {..<n} σ permutes {..<n} for σ π
  proof (intro ext)
    fix x show convert-perm (π ∘ σ) x = (convert-perm π ∘ convert-perm σ) x
    proof (cases x ∈ Roots)
      case True
```

```
    thus ?thesis
      using permutes-in-image[OF that(2), of unRoot x]
      by (auto simp: convert-perm-def bij-betw-def)
  qed (auto simp: convert-perm-def)
qed
```

We extend the coefficient vector to the new roots by setting their coefficients to 0:

```
define β' where β' = (λx. if x ∈ X then β x else 0)
```

We now define the set of all permutations of our roots:

```
define perms where perms = {π. π permutes Roots}
have [intro]: finite perms
  unfolding perms-def by (rule finite-permutations) auto
have [simp]: card perms = fact n
  unfolding perms-def n-def by (intro card-permutations) auto
```

The following is the set of all $n!$-tuples of roots, disregarding permutation of components. In other words: all multisets of roots with size $n!$.

```
define Roots-ms :: complex multiset set where
  Roots-ms = {X. set-mset X ⊆ Roots ∧ size X = fact n}
have [intro]: finite Roots-ms
  unfolding Roots-ms-def by (rule finite-multisets-of-size) auto
```

Next, the following is the set of $n!$-tuples whose entries are precisely the multiset X:

```
define tuples :: complex multiset ⇒ ((complex ⇒ complex) ⇒ complex) set where
  tuples = (λX. {f∈perms →E Roots. image-mset f (mset-set perms) = X})
have fin-tuples [intro]: finite (tuples X) for X
   unfolding tuples-def by (rule finite-subset[of - perms →E Roots, OF - finite-PiE]) auto
 define tuples' :: (complex multiset × ((complex ⇒ complex) ⇒ complex)) set
where
  tuples' = (SIGMA X:Roots-ms. tuples X)
```

The following shows that our *tuples* definition is stable under permutation of the roots.

```
have bij-convert': bij-betw (λf. f ∘ (λg. σ ∘ g)) (tuples X) (tuples X)
  if σ: σ permutes Roots for σ X
proof (rule bij-betwI)
  have ∗: (λf. f ∘ (∘) σ) ∈ tuples X → tuples X if σ: σ permutes Roots for σ
  proof
    fix f assume f: f ∈ tuples X
    show f ∘ (∘) σ ∈ tuples X
      unfolding tuples-def
    proof safe
      fix σ'
      assume σ': σ' ∈ perms
```

```
      show (f ∘ (∘) σ) σ' ∈ Roots
        using permutes-compose[OF - σ, of σ'] σ σ' f by (auto simp: perms-def
tuples-def)
    next
      fix σ'
      assume σ': σ' ∉ perms
      have ¬(σ ∘ σ') permutes Roots
      proof
        assume (σ ∘ σ') permutes Roots
        hence inv-into UNIV σ ∘ (σ ∘ σ') permutes Roots
          by (rule permutes-compose) (use permutes-inv[OF σ] in simp-all)
        also have inv-into UNIV σ ∘ (σ ∘ σ') = σ'
          by (auto simp: fun-eq-iff permutes-inverses[OF σ])
        finally show False using σ' by (simp add: perms-def)
      qed
      thus (f ∘ (∘) σ) σ' = undefined
        using f by (auto simp: perms-def tuples-def)
    next
      have image-mset (f ∘ (∘) σ) (mset-set perms) =
          image-mset f (image-mset ((∘) σ) (mset-set perms))
        by (rule multiset.map-comp [symmetric])
      also have image-mset ((∘) σ) (mset-set perms) = mset-set perms
        using bij-betw-permutes-compose-left[OF σ]
        by (subst image-mset-mset-set) (auto simp: bij-betw-def perms-def)
      also have image-mset f ... = X
        using f by (auto simp: tuples-def)
      finally show image-mset (f ∘ (∘) σ) (mset-set perms) = X .
    qed
  qed

  show (λf. f ∘ (∘) σ) ∈ tuples X → tuples X
    by (rule *) fact
  show (λf. f ∘ (∘) (inv-into UNIV σ)) ∈ tuples X → tuples X
    by (intro * permutes-inv) fact
  show f ∘ (∘) σ ∘ (∘) (inv-into UNIV σ) = f if f ∈ tuples X for f
    by (auto simp: fun-eq-iff o-def permutes-inverses[OF σ])
  show f ∘ (∘) (inv-into UNIV σ) ∘ (∘) σ = f if f ∈ tuples X for f
    by (auto simp: fun-eq-iff o-def permutes-inverses[OF σ])
qed
```

Next, we define the multiset of of possible exponents that we can get for a given $n!$-multiset of roots,

```
define R :: complex multiset ⇒ complex multiset where
  R = (λX. image-mset (λf. ∑σ∈perms. σ (f σ)) (mset-set (tuples X)))
```

We show that, for each such multiset, there is a content-free integer polynomial that has exactly these exponents as roots. This shows that they form a full set of conjugates (but note this polynomial is not necessarily squarefree). The proof is yet another application of the fundamental theorem of sym-

metric polynomials.

```
  obtain Q :: complex multiset ⇒ int poly
    where Q: ⋀X. X ∈ Roots-ms ⟹ poly-roots (of-int-poly (Q X)) = R X
          ⋀X. X ∈ Roots-ms ⟹ content (Q X) = 1
  proof −
    {
      fix X :: complex multiset
      assume X: X ∈ Roots-ms
      define Q :: complex poly mpoly where
        Q = (∏f∈tuples X. Const [:0, 1:] −
              (∑σ | σ permutes {..<n}. Var (σ (unRoot (f (convert-perm σ))))))
      define Q1 where Q1 = (∏f∈tuples X. [:− (∑σ | σ permutes Roots. σ (f
σ)), 1:])
      define ratpolys :: complex poly set where ratpolys = {p. ∀i. poly.coeff p i ∈
ℚ}

      have insertion (λx. [:Root x:]) Q ∈ ratpolys
      proof (rule symmetric-poly-of-roots-in-subring[where l = λx. [:x:]])
        show ring-closed ratpolys
         unfolding ratpolys-def by standard (auto intro: coeff-mult-semiring-closed)
        then interpret ratpolys: ring-closed ratpolys .
        have pCons 0 1 ∈ ratpolys
          by (auto simp: ratpolys-def coeff-pCons split: nat.splits)
        thus ∀m. MPoly-Type.coeff Q m ∈ ratpolys
          unfolding Q-def
          by (intro allI ratpolys.coeff-prod-closed)
         (auto intro!: ratpolys.minus-closed ratpolys.sum-closed ratpolys.uminus-closed
simp: coeff-Var mpoly-coeff-Const when-def)
      next
        show ring-homomorphism (λx::complex. [:x:]) ..
      next
         have σ (unRoot (f (convert-perm σ))) < n if f ∈ tuples X σ permutes
{..<n} for f σ
        proof −
          have convert-perm σ ∈ perms
            using bij-convert that(2) by (auto simp: bij-betw-def perms-def)
          hence f (convert-perm σ) ∈ Roots
            using that by (auto simp: tuples-def)
          thus ?thesis
            using permutes-in-image[OF that(2)] by simp
        qed
        thus vars Q ⊆ {..<n}
          unfolding Q-def
          by (intro order.trans[OF vars-prod] UN-least order.trans[OF vars-sum]
                order.trans[OF vars-diff] Un-least) (auto simp: vars-Var)
      next
        define lc :: complex where lc = of-int (Polynomial.lead-coeff P)
        show [:inverse lc:] ∈ ratpolys
          by (auto simp: ratpolys-def coeff-pCons lc-def split: nat.splits)
```

show $\forall\, i.$ [:*poly.coeff* (*of-int-poly P*) *i*:] ∈ *ratpolys*
by (*auto simp*: *ratpolys-def coeff-pCons split*: *nat.splits*)
have *lc* ≠ *0*
by (*auto simp*: *lc-def*)
thus [:*inverse lc*:] ∗ [:*lc*:] = *1*
by *auto*
have *rsquarefree* (*of-int-poly P* :: *complex poly*)
using ‹*squarefree* (*of-int-poly P* :: *complex poly*)› **by** (*intro squarefree-imp-rsquarefree*)
hence *of-int-poly P* = *Polynomial.smult lc* (∏ *x*∈*Roots*. [:−*x*, *1*:])
unfolding *lc-def Roots-def of-int-hom.hom-lead-coeff*[*symmetric*]
by (*rule complex-poly-decompose-rsquarefree* [*symmetric*])
also have (∏ *x*∈*Roots*. [:−*x*, *1*:]) = (∏ *i*<*n*. [:−*Root i*, *1*:])
by (*rule prod.reindex-bij-betw*[*OF Root*, *symmetric*])
finally show *of-int-poly P* = *Polynomial.smult lc* (∏ *i*<*n*. [:− *Root i*, *1*:]) **.**
next
show *symmetric-mpoly* {..<*n*} *Q*
unfolding *symmetric-mpoly-def*
proof *safe*
fix π **assume** π: π *permutes* {..<*n*}
have *mpoly-map-vars* π *Q* = (∏ *f*∈*tuples X*. *Const* (*pCons 0 1*) − (∑ σ | σ *permutes* {..<*n*}.
Var ((π ∘ σ) (*unRoot* (*f* (*convert-perm* σ))))))
by (*simp add*: *Q-def permutes-bij*[*OF* π])
also have ... = (∏ *f*∈*tuples X*. *Const* (*pCons 0 1*) − (∑ σ | σ *permutes* {..<*n*}.
Var ((π ∘ σ) (*unRoot* ((*f* ∘ (λσ. *convert-perm* π ∘ σ)) (*convert-perm* σ))))))
using π **by** (*intro prod.reindex-bij-betw* [*OF bij-convert*′, *symmetric*]) *auto*
also have ... = *Q*
unfolding *Q-def*
proof (*rule prod.cong*, *goal-cases*)
case (*2 f*)
have (∑ σ | σ *permutes* {..<*n*}. *Var* ((π ∘ σ) (*unRoot* ((*f* ∘ (λσ. *convert-perm* π ∘ σ)) (*convert-perm* σ))))) =
(∑ σ | σ *permutes* {..<*n*}. *Var* ((π ∘ σ) (*unRoot* (*f* (*convert-perm* (π ∘ σ))))))
using π **by** (*intro sum.cong refl*, *subst convert-perm-compose*) *simp-all*
also have ... = (∑ σ | σ *permutes* {..<*n*}. *Var* (σ (*unRoot* (*f* (*convert-perm* σ)))))
using π **by** (*rule setum-permutations-compose-left* [*symmetric*])
finally show *?case* **by** *simp*
qed *auto*
finally show *mpoly-map-vars* π *Q* = *Q* **.**
qed
qed *auto*
also have *insertion* (λ*x*. [:*Root x*:]) *Q* = *Q1*
unfolding *Q-def Q1-def insertion-prod insertion-sum insertion-diff inser-*

tion-Const insertion-Var
proof (*intro prod.cong*, *goal-cases*)
case *f*: (*2 f*)
have ($\sum\sigma$ | σ *permutes* {..<*n*}. [:*Root* (σ (*unRoot* (*f* (*convert-perm* σ)))):])
=
($\sum\sigma$ | σ *permutes* {..<*n*}. [:*convert-perm* σ (*f* (*convert-perm* σ)):])
proof (*rule sum.cong*, *goal-cases*)
case (*2* σ)
have *convert-perm* σ *permutes Roots*
using *bij-convert 2* **by** (*auto simp*: *bij-betw-def*)
hence *f* (*convert-perm* σ) ∈ *Roots*
using *f* **by** (*auto simp*: *tuples-def perms-def*)
thus *?case* **by** (*simp add*: *convert-perm-def*)
qed *simp-all*
also have ... = ($\sum\sigma$ | σ *permutes Roots*. [:σ (*f* σ):])
by (*rule sum.reindex-bij-betw*[*OF bij-convert*])
finally show *?case*
by (*simp flip*: *pCons-one coeff-lift-hom.hom-sum*)
qed *simp-all*
finally have *Q1* ∈ *ratpolys*
by *auto*
then obtain *Q2* :: *rat poly* **where** *Q2*: *Q1* = *map-poly of-rat Q2*
unfolding *ratpolys-def* **using** *ratpolyE*[*of Q1*] **by** *blast*

have *Q1* ≠ *0*
unfolding *Q1-def* **using** *fin-tuples*[*of X*] **by** *auto*
with *Q2* **have** *Q2* ≠ *0*
by *auto*
obtain *Q3* :: *int poly* **and** *lc* :: *rat*
where *Q3*: *Q2* = *Polynomial.smult lc* (*of-int-poly Q3*) **and** *lc* > *0* **and** *content Q3* = *1*
using *rat-to-normalized-int-poly-exists*[*OF* ‹*Q2* ≠ *0*›] **by** *metis*

have *poly-roots* (*of-int-poly Q3*) = *poly-roots* (*map-poly* (*of-rat* ∘ *of-int*) *Q3*)
by *simp*
also have *map-poly* (*of-rat* ∘ *of-int*) *Q3* = *map-poly of-rat* (*map-poly of-int Q3*)
by (*subst map-poly-map-poly*) *auto*
also have *poly-roots* ... = *poly-roots* (*Polynomial.smult* (*of-rat lc*) ...)
using ‹*lc* > *0*› **by** *simp*
also have *Polynomial.smult* (*of-rat lc*) (*map-poly of-rat* (*map-poly of-int Q3*))
=
map-poly of-rat (*Polynomial.smult lc* (*map-poly of-int Q3*))
by (*simp add*: *of-rat-hom.map-poly-hom-smult*)
also have ... = *Q1*
by (*simp only*: *Q3* [*symmetric*] *Q2* [*symmetric*])
also have *poly-roots Q1* = *R X*
unfolding *Q1-def*
by (*subst poly-roots-prod*, *force*, *subst poly-roots-linear*)

```
    (auto simp: R-def perms-def sum-mset-image-mset-singleton sum-unfold-sum-mset)
   finally have ∃ Q. poly-roots (of-int-poly Q) = R X ∧ content Q = 1
    using ‹content Q3 = 1› by metis
  }
  hence ∃ Q. ∀ X∈Roots-ms. poly-roots (of-int-poly (Q X)) = R X ∧ content (Q
X) = 1
   by metis
  thus ?thesis using that by metis
 qed
```

We can now collect all the $e^{\sum \alpha_i}$ that happen to be equal and let the following be their coefficients:

```
 define β'' :: int poly ⇒ int
  where β'' = (λq. ∑ X∈Roots-ms. int (count (prime-factorization (Q X)) q) *
(∏ x∈#X. β' x))
 have supp-β'': {q. β'' q ≠ 0} ⊆ (⋃ X∈Roots-ms. prime-factors (Q X))
  unfolding β''-def using sum.not-neutral-contains-not-neutral by fastforce
```

We have to prove that β'' is not zero everywhere. We do this by selecting the nonzero term with the maximal exponent (w.r.t. the lexicographic ordering on the complex numbers) in every factor of the product and show that there is no other summand corresponding to these, so that their non-zero coefficient cannot get cancelled.

```
 have {q. β'' q ≠ 0} ≠ {}
 proof −
  define f where f = restrict (λσ. inv-into UNIV σ (complex-lex.Max (σ ‘ X)))
perms
  have f: f ∈ perms → X
  proof
   fix σ assume σ: σ ∈ perms
   have complex-lex.Max (σ ‘ X) ∈ σ ‘ X
    using ‹X ≠ {}› by (intro complex-lex.Max-in finite-imageI) auto
   thus f σ ∈ X
    using σ by (auto simp: f-def permutes-inverses[of σ Roots] perms-def)
  qed
  hence f': f ∈ perms →E Roots
   using ‹X ⊆ Roots› by (auto simp: f-def PiE-def)

  define Y where Y = image-mset f (mset-set perms)
  have Y ∈ Roots-ms using f' ‹finite perms›
   by (auto simp: Roots-ms-def Y-def)

  have (∑ σ∈perms. σ (f σ)) ∈# R Y
  proof −
   from f' have f ∈ tuples Y
    unfolding tuples-def Y-def by simp
   thus ?thesis
    unfolding R-def using fin-tuples[of Y] by auto
```

```
  qed
  also have R Y = poly-roots (of-int-poly (Q Y))
    by (rule Q(1) [symmetric]) fact
  also have ... = (∑ p∈#prime-factorization (Q Y). poly-roots (of-int-poly p))
    by (rule poly-roots-of-int-conv-sum-prime-factors)
  finally obtain q where q: q ∈ prime-factors (Q Y) (∑ σ∈perms. σ (f σ)) ∈#
poly-roots (of-int-poly q)
    by auto

  have β'' q = (∑ X∈{Y}. int (count (prime-factorization (Q X)) q) * prod-mset
(image-mset β' X))
    unfolding β''-def
  proof (intro sum.mono-neutral-right ballI)
    fix Y' assume Y': Y' ∈ Roots-ms − {Y}
    show int (count (prime-factorization (Q Y')) q) * ∏# (image-mset β' Y')
= 0
    proof (cases set-mset Y' ⊆ X)
      case Y'-subset: True
      have q ∉ prime-factors (Q Y')
      proof
        assume q': q ∈ prime-factors (Q Y')
        have poly-roots (of-int-poly q :: complex poly) ⊆#
                    poly-roots (of-int-poly (Q Y'))
          using q' by (intro dvd-imp-poly-roots-subset of-int-poly-hom.hom-dvd)
auto
        with q(2) have (∑ σ∈perms. σ (f σ)) ∈# poly-roots (of-int-poly (Q Y'))
          by (meson mset-subset-eqD)
        also have poly-roots (of-int-poly (Q Y')) = R Y'
          using Q(1)[of Y'] Y' by auto
          finally obtain g where g: g ∈ tuples Y' (∑ σ∈perms. σ (f σ)) =
(∑ σ∈perms. σ (g σ))
          unfolding R-def using fin-tuples[of Y'] by auto

        moreover have (∑ σ∈perms. σ (g σ)) <_c (∑ σ∈perms. σ (f σ))
        proof (rule sum-strict-mono-ex1-complex-lex)
          show le: ∀ σ∈perms. σ (g σ) ≤_c σ (f σ)
          proof
            fix σ assume σ: σ ∈ perms
            hence σ': σ permutes Roots
              by (auto simp: perms-def)
            have image-mset g (mset-set perms) = Y'
              using g by (auto simp: tuples-def)
            also have set-mset ... ⊆ X
              by fact
            finally have g ‘ perms ⊆ X
              using ‹finite perms› by auto
            hence σ (g σ) ≤_c complex-lex.Max (σ ‘ X)
              using ‹finite perms› σ
              by (intro complex-lex.Max.coboundedI finite-imageI imageI)
```

```
            (auto simp: tuples-def)
          also have ... = σ (f σ)
            using σ by (simp add: f-def permutes-inverses[OF σ'])
          finally show σ (g σ) ≤𝐜 σ (f σ) .
        qed

        have image-mset g (mset-set perms) ≠ image-mset f (mset-set perms)
          using Y' g by (auto simp: tuples-def Y-def)
        then obtain σ where σ: σ ∈# mset-set perms g σ ≠ f σ
          by (meson multiset.map-cong)
        have σ permutes Roots
          using σ ‹finite perms› by (auto simp: perms-def)
        have σ (g σ) ≠ σ (f σ)
          using permutes-inj[OF ‹σ permutes Roots›] σ by (auto simp: inj-def)
        moreover have σ (g σ) ≤𝐜 σ (f σ)
          using le σ ‹finite perms› by auto
        ultimately have σ (g σ) <𝐜 σ (f σ)
          by simp
        thus ∃σ∈perms. σ (g σ) <𝐜 σ (f σ)
          using σ ‹finite perms› by auto
      qed (use ‹finite perms› in simp-all)
      ultimately show False by simp
    qed
    thus ?thesis by auto
  qed (auto simp: β'-def)
 qed (use ‹Y ∈ Roots-ms› in auto)
 also have ... = int (count (prime-factorization (Q Y)) q) * prod-mset (image-mset
β' Y)
   by simp
 also have ... ≠ 0
   using q nz ‹finite X› ‹X ≠ {}› ‹finite perms› f by (auto simp: β'-def Y-def)
 finally show {q. β'' q ≠ 0} ≠ {}
   by auto
qed
```

We are now ready for the final push: we start with the original sum that we know to be zero, multiply it with the other permutations, and then multiply out the sum.

```
have 0 = (∑ x∈X. β x * exp x)
  using sum0 ..
also have ... = (∑ x∈Roots. β' x * exp x)
  by (intro sum.mono-neutral-cong-left ‹X ⊆ Roots›) (auto simp: β'-def)
also have ... dvd (∏ σ∈perms. ∑ x∈Roots. β' x * exp (σ x))
  by (rule dvd-prodI[OF ‹finite perms›])
     (use permutes-id[of Roots] in ‹simp-all add: id-def perms-def›)
also have ... = (∑ f∈perms →E Roots. ∏ σ∈perms. β' (f σ) * exp (σ (f σ)))
  by (rule prod-sum-PiE) auto
also have ... = (∑ f∈perms →E Roots. (∏ σ∈perms. β' (f σ)) * exp (∑ σ∈perms.
σ (f σ)))
```

using ‹*finite perms*› **by** (*simp add*: *prod.distrib exp-sum*)
also have ... = ($\sum$ (*X*,*f*)∈*tuples'*. ($\prod$ σ∈*perms*. β' (*f* σ)) * *exp* ($\sum$ σ∈*perms*. σ (*f* σ)))
using ‹*finite perms*›
by (*intro sum.reindex-bij-witness*[*of* - *snd* λ*f*. (*image-mset f* (*mset-set perms*), *f*)])
(*auto simp*: *tuples'-def tuples-def Roots-ms-def PiE-def Pi-def*)
also have ... = ($\sum$ (*X*,*f*)∈*tuples'*. ($\prod$ *x*∈#*X*. β' *x*) * *exp* ($\sum$ σ∈*perms*. σ (*f* σ)))
proof (*safe intro*!: *sum.cong*)
fix *X* :: *complex multiset* **and** *f* :: (*complex* ⇒ *complex*) ⇒ *complex*
assume (*X*, *f*) ∈ *tuples'*
hence *X*: *X* ∈ *Roots-ms X* = *image-mset f* (*mset-set perms*) **and** *f*: *f* ∈ *perms* →$_E$ *Roots*
by (*auto simp*: *tuples'-def tuples-def*)
have ($\prod$ σ∈*perms*. β' (*f* σ)) = ($\prod$ σ∈#*mset-set perms*. β' (*f* σ))
by (*meson prod-unfold-prod-mset*)
also have ... = ($\prod$ *x*∈#*X*. β' *x*)
unfolding *X*(*2*) **by** (*simp add*: *multiset.map-comp o-def*)
finally show ($\prod$ σ∈*perms*. β' (*f* σ)) * *exp* ($\sum$ σ∈*perms*. σ (*f* σ)) =
($\prod$ *x*∈#*X*. β' *x*) * *exp* ($\sum$ σ∈*perms*. σ (*f* σ)) **by** *simp*
qed
also have ... = ($\sum$ *X*∈*Roots-ms*. $\sum$ *f*∈*tuples X*. ($\prod$ *x*∈#*X*. β' *x*) * *exp* ($\sum$ σ∈*perms*. σ (*f* σ)))
unfolding *tuples'-def* **by** (*intro sum.Sigma* [*symmetric*]) *auto*
also have ... = ($\sum$ *X*∈*Roots-ms*. *of-int* ($\prod$ *x*∈#*X*. β' *x*) * ($\sum$ *f*∈*tuples X*. *exp* ($\sum$ σ∈*perms*. σ (*f* σ))))
by (*simp add*: *sum-distrib-left*)
also have ... = ($\sum$ *X*∈*Roots-ms*. *of-int* ($\prod$ *x*∈#*X*. β' *x*) * ($\sum$ *x*∈#*R X*. *exp x*))
by (*simp only*: *R-def multiset.map-comp o-def sum-unfold-sum-mset*)
also have ... = ($\sum$ *X*∈*Roots-ms*. *of-int* ($\prod$ *x*∈#*X*. β' *x*) * ($\sum$ *x*∈#*poly-roots* (*of-int-poly* (*Q X*)). *exp x*))
by (*intro sum.cong*) (*simp-all flip*: *Q*)

Our problem now is that the polynomials $Q\ X$ can still contain multiple roots and that their roots might not be disjoint. We therefore split them all into irreducible factors and collect equal terms.

also have ... = ($\sum$ *X*∈*Roots-ms*. ($\sum$ *p*. *of-int* (*int* (*count* (*prime-factorization* (*Q X*)) *p*) *
($\prod$ *x*∈#*X*. β' *x*)) * ($\sum$ *x* | *ipoly p x* = *0*. *exp x*)))
proof (*rule sum.cong*, *goal-cases*)
case (*2 X*)
have ($\sum$ *x*∈#*poly-roots* (*of-int-poly* (*Q X*) :: *complex poly*). *exp x*) =
($\sum$ *x* ∈# ($\sum$ *p*∈#*prime-factorization* (*Q X*). *poly-roots* (*of-int-poly p*)). *exp x*)
by (*subst poly-roots-of-int-conv-sum-prime-factors*) (*rule refl*)
also have ... = ($\sum$ *p*∈#*prime-factorization* (*Q X*). $\sum$ *x*∈#*poly-roots* (*of-int-poly p*). *exp x*)
by (*rule sum-mset-image-mset-sum-mset-image-mset*)

```
  also have rsquarefree (of-int-poly p :: complex poly) if p ∈ prime-factors (Q X)
for p
    proof (rule irreducible-imp-rsquarefree-of-int-poly)
      have prime p
        using that by auto
      thus irreducible p
        by blast
    next
      show Polynomial.degree p > 0
        by (intro content-1-imp-nonconstant-prime-factors[OF Q(2) that] 2)
    qed
    hence (∑ p∈#prime-factorization (Q X). ∑ x∈#poly-roots (of-int-poly p). exp
x) =
          (∑ p∈#prime-factorization (Q X). ∑ x | ipoly p x = 0. exp (x :: complex))
      unfolding sum-unfold-sum-mset
      by (intro arg-cong[of - - sum-mset] image-mset-cong sum.cong refl,
          subst rsquarefree-poly-roots-eq) auto
    also have ... = (∑ p. count (prime-factorization (Q X)) p * (∑ x | ipoly p x
= 0. exp (x :: complex)))
      by (rule sum-mset-conv-Sum-any)
    also have of-int (∏ x∈#X. β' x) * ... =
                (∑ p. of-int (int (count (prime-factorization (Q X)) p) * (∏ x∈#X.
β' x)) * (∑ x | ipoly p x = 0. exp x))
      by (subst Sum-any-right-distrib) (auto simp: mult-ac)
    finally show ?case by simp
  qed auto
  also have ... = (∑ q. of-int (β'' q) * (∑ x | ipoly q x = 0. exp x))
    unfolding β''-def of-int-sum
    by (subst Sum-any-sum-swap [symmetric]) (auto simp: sum-distrib-right)
  also have ... = (∑ q | β'' q ≠ 0. of-int (β'' q) * (∑ x | ipoly q x = 0. exp x))
    by (intro Sum-any.expand-superset finite-subset[OF supp-β''] auto
  finally have (∑ q | β'' q ≠ 0. of-int (β'' q) * (∑ x | ipoly q x = 0. exp (x ::
complex))) = 0
    by simp
```

We are now in the situation of our the specialised Hermite–Lindemann Theorem we proved earlier and can easily derive a contradiction.

```
  moreover have (∑ q | β'' q ≠ 0. of-int (β'' q) * (∑ x | ipoly q x = 0. exp (x
:: complex))) ≠ 0
  proof (rule Hermite-Lindemann-aux1)
    show finite {q. β'' q ≠ 0}
      by (rule finite-subset[OF supp-β''] auto
  next
    show pairwise Rings.coprime {q. β'' q ≠ 0}
    proof (rule pairwiseI, clarify)
      fix p q assume pq: p ≠ q β'' p ≠ 0 β'' q ≠ 0
      hence prime p prime q
        using supp-β'' Q(2) by auto
      with pq show Rings.coprime p q
```

```
      by (simp add: primes-coprime)
    qed
  next
    fix q :: int poly
    assume q: q ∈ {q. β'' q ≠ 0}
    also note supp-β''
    finally obtain X where X: X ∈ Roots-ms q ∈ prime-factors (Q X)
      by blast
    show irreducible q
      using X by (intro prime-elem-imp-irreducible prime-imp-prime-elem) auto
    show Polynomial.degree q > 0 using X
      by (intro content-1-imp-nonconstant-prime-factors[OF Q(2)[of X]])
  qed (use ‹{x. β'' x ≠ 0} ≠ {}› in auto)

  ultimately show False by contradiction
qed
```

7.3 Removing the restriction to integer coefficients

Next, we weaken the restriction that the β_i must be integers to the restriction that they must be rationals. This is done simply by multiplying with the least common multiple of the demoninators.

```
lemma Hermite-Lindemann-aux3:
  fixes X :: complex set and β :: complex ⇒ rat
  assumes finite X
  assumes nz:  ⋀x. x ∈ X ⟹ β x ≠ 0
  assumes alg: ⋀x. x ∈ X ⟹ algebraic x
  assumes sum0: (∑ x∈X. of-rat (β x) * exp x) = 0
  shows   X = {}
proof −
  define l :: int where l = Lcm ((snd ∘ quotient-of ∘ β) ' X)
  have [simp]: snd (quotient-of r) ≠ 0 for r
    using quotient-of-denom-pos'[of r] by simp
  have [simp]: l ≠ 0
    using ‹finite X› by (auto simp: l-def Lcm-0-iff)

  have of-int l * β x ∈ ℤ if x ∈ X for x
  proof −
    define a b where a = fst (quotient-of (β x)) and b = snd (quotient-of (β x))
    have b > 0
      using quotient-of-denom-pos'[of β x] by (auto simp: b-def)
    have β x = of-int a / of-int b
      by (intro quotient-of-div) (auto simp: a-def b-def)
    also have of-int l * ... = of-int (l * a) / of-int b
      using ‹b > 0› by (simp add: field-simps)
    also have ... ∈ ℤ using that
      by (intro of-int-divide-in-Ints) (auto simp: l-def b-def)
    finally show ?thesis .
  qed
```

```
  hence ∀ x∈X. ∃ n. of-int n = of-int l * β x
    using Ints-cases by metis
  then obtain β' where β': of-int (β' x) = of-int l * β x if x ∈ X for x
    by metis

  show ?thesis
  proof (rule Hermite-Lindemann-aux2)
    have 0 = of-int l * (∑ x∈X. of-rat (β x) * exp x :: complex)
      by (simp add: sum0)
    also have ... = (∑ x∈X. of-int (β' x) * exp x)
      unfolding sum-distrib-left
    proof (rule sum.cong, goal-cases)
      case (2 x)
      have of-int l * of-rat (β x) = of-rat (of-int l * β x)
        by (simp add: of-rat-mult)
      also have of-int l * β x = of-int (β' x)
        using 2 by (rule β' [symmetric])
      finally show ?case by (simp add: mult-ac)
    qed simp-all
    finally show ... = 0 ..
  next
    fix x assume x ∈ X
    hence of-int (β' x) ≠ (0 :: rat) using nz
      by (subst β') auto
    thus β' x ≠ 0
      by auto
  qed (use alg ‹finite X› in auto)
qed
```

Next, we weaken the restriction that the β_i must be rational to them being algebraic. Similarly to before, this is done by multiplying over all possible permutations of the β_i (in some sense) to introduce more symmetry, from which it then follows by the fundamental theorem of symmetric polynomials that the resulting coefficients are rational.

```
lemma Hermite-Lindemann-aux4:
  fixes β :: complex ⇒ complex
  assumes [intro]: finite X
  assumes alg1: ⋀x. x ∈ X ⟹ algebraic x
  assumes alg2: ⋀x. x ∈ X ⟹ algebraic (β x)
  assumes nz:   ⋀x. x ∈ X ⟹ β x ≠ 0
  assumes sum0: (∑ x∈X. β x * exp x) = 0
  shows   X = {}
proof (rule ccontr)
  assume X: X ≠ {}
  note [intro!] = finite-PiE
```

We now take more or less the same approach as before, except that now we find a polynomial that has all of the conjugates of the coefficients β as roots. Note that this is a slight deviation from Baker's proof, who picks

one polynomial for each β independently. I did it this way because, as Bernard [2] observed, it makes the proof a bit easier.

```
define P :: int poly where P = ∏ ((min-int-poly ∘ β) ` X)
define Roots :: complex set where Roots = {x. ipoly P x = 0}
have 0 ∉ Roots using ‹finite X› alg2 nz
  by (auto simp: Roots-def P-def poly-prod)
have [simp]: P ≠ 0
  using ‹finite X› by (auto simp: P-def)
have [intro]: finite Roots
  unfolding Roots-def by (intro poly-roots-finite) auto

have β ` X ⊆ Roots
proof safe
  fix x assume x ∈ X
  hence ipoly (min-int-poly (β x)) (β x) = 0
    by (intro ipoly-min-int-poly alg2)
  thus β x ∈ Roots
    using ‹finite X› ‹x ∈ X›
    by (auto simp: Roots-def P-def of-int-poly-hom.hom-prod poly-prod)
qed

have squarefree (of-int-poly P :: complex poly)
  unfolding P-def of-int-poly-hom.hom-prod o-def
proof (rule squarefree-prod-coprime; safe)
  fix x assume x ∈ X
  thus squarefree (of-int-poly (min-int-poly (β x)) :: complex poly)
    by (intro squarefree-of-int-polyI) auto
next
  fix x y assume xy: x ∈ X y ∈ X min-int-poly (β x) ≠ min-int-poly (β y)
  thus Rings.coprime (of-int-poly (min-int-poly (β x)))
         (of-int-poly (min-int-poly (β y)) :: complex poly)
    by (intro coprime-of-int-polyI[OF primes-coprime]) auto
qed

define n where n = card Roots
define m where m = card X
have Roots ≠ {}
  using ‹β ` X ⊆ Roots› ‹X ≠ {}› by auto
hence n > 0 m > 0
  using ‹finite Roots› ‹finite X› ‹X ≠ {}› by (auto simp: n-def m-def)
have fin1 [simp]: finite (X →E Roots)
  by auto
have [simp]: card (X →E Roots) = n ^ m
  by (subst card-PiE) (auto simp: m-def n-def)
```

We again find a bijection between the roots and the natural numbers less than n:

```
obtain Root where Root: bij-betw Root {..<n} Roots
```

using *ex-bij-betw-nat-finite*[*OF* ‹*finite Roots*›] **unfolding** *n-def atLeast0LessThan*
by *metis*
define *unRoot* :: *complex* ⇒ *nat* **where** *unRoot* = *inv-into* {..<*n*} *Root*
have *unRoot*: *bij-betw unRoot Roots* {..<*n*}
unfolding *unRoot-def* **by** (*intro bij-betw-inv-into Root*)
have *unRoot-Root* [*simp*]: *unRoot* (*Root i*) = *i* **if** *i* < *n* **for** *i*
unfolding *unRoot-def* **using** *Root that* **by** (*subst inv-into-f-f*) (*auto simp*: *bij-betw-def*)
have *Root-unRoot* [*simp*]: *Root* (*unRoot x*) = *x* **if** *x* ∈ *Roots* **for** *x*
unfolding *unRoot-def* **using** *Root that* **by** (*subst f-inv-into-f*) (*auto simp*: *bij-betw-def*)
have [*simp*, *intro*]: *Root i* ∈ *Roots* **if** *i* < *n* **for** *i*
using *Root that* **by** (*auto simp*: *bij-betw-def*)
have [*simp*, *intro*]: *unRoot x* < *n* **if** *x* ∈ *Roots* **for** *x*
using *unRoot that* **by** (*auto simp*: *bij-betw-def*)

And we again define the set of multisets and tuples that we will get in the expanded product.

define *Roots-ms* :: *complex multiset set* **where**
Roots-ms = {*Y*. *set-mset Y* ⊆ *X* ∧ *size Y* = *n* ^ *m*}
have [*intro*]: *finite Roots-ms*
unfolding *Roots-ms-def* **by** (*rule finite-multisets-of-size*) *auto*
define *tuples* :: *complex multiset* ⇒ ((*complex* ⇒ *complex*) ⇒ *complex*) *set*
where *tuples* = (λ*Y*. {*f*∈(*X* →*E Roots*) →*E X*. *image-mset f* (*mset-set* (*X* →*E Roots*)) = *Y*})
have [*intro*]: *finite* (*tuples Y*) **for** *Y*
unfolding *tuples-def* **by** (*rule finite-subset*[*of* - (*X* →*E Roots*) →*E X*]) *auto*

We will also need to convert permutations over the natural and over the roots again.

define *convert-perm* :: (*nat* ⇒ *nat*) ⇒ (*complex* ⇒ *complex*) **where**
convert-perm = (λσ *x*. *if x* ∈ *Roots then Root* (σ (*unRoot x*)) *else x*)
have *bij-convert*: *bij-betw convert-perm* {σ. σ *permutes* {..<*n*}} {σ. σ *permutes Roots*}
using *bij-betw-permutations*[*OF Root*] **unfolding** *convert-perm-def unRoot-def*
.
have *permutes-convert-perm* [*intro*]: *convert-perm* σ *permutes Roots* **if** σ *permutes* {..<*n*} **for** σ
using *that bij-convert* **unfolding** *bij-betw-def* **by** *blast*

We also need a small lemma showing that our tuples are stable under permutation of the roots.

have *bij-betw-compose-perm*:
bij-betw (λ*f*. *restrict* (λ*g*. *f* (*restrict* (π ∘ *g*) *X*)) (*X* →*E Roots*)) (*tuples Y*) (*tuples Y*)
if π: π *permutes Roots* **and** *Y* ∈ *Roots-ms* **for** π *Y*
proof (*rule bij-betwI*)
have ∗: (λ*f*. *restrict* (λ*g*. *f* (*restrict* (π ∘ *g*) *X*)) (*X* →*E Roots*)) ∈ *tuples Y* → *tuples Y*

```
    if π: π permutes Roots for π
  proof
    fix f assume f: f ∈ tuples Y
    hence f': f ∈ (X →E Roots) →E X
      by (auto simp: tuples-def)
    define f' where f' = (λg. f (restrict (π ∘ g) X))
    have f' ∈ (X →E Roots) → X unfolding f'-def
      using f' bij-betw-apply[OF bij-betw-compose-left-perm-PiE[OF π, of X]] by
blast
    hence restrict f' (X →E Roots) ∈ (X →E Roots) →E X
      by simp
     moreover have image-mset (restrict f' (X →E Roots)) (mset-set (X →E
Roots)) = Y
    proof −
      have image-mset (restrict f' (X →E Roots)) (mset-set (X →E Roots)) =
          image-mset f' (mset-set (X →E Roots))
        by (intro image-mset-cong) auto
     also have ... = image-mset f (image-mset (λg. restrict (π ∘ g) X) (mset-set
(X →E Roots)))
        unfolding f'-def o-def multiset.map-comp by (simp add: o-def)
      also have image-mset (λg. restrict (π ∘ g) X) (mset-set (X →E Roots)) =
            mset-set (X →E Roots)
        by (intro bij-betw-image-mset-set bij-betw-compose-left-perm-PiE π)
      also have image-mset f ... = Y
        using f by (simp add: tuples-def)
      finally show ?thesis .
    qed
    ultimately show restrict f' (X →E Roots) ∈ tuples Y
      by (auto simp: tuples-def)
  qed
  show (λf. restrict (λg. f (restrict (π ∘ g) X)) (X →E Roots)) ∈ tuples Y →
tuples Y
    by (intro ∗ π)
  show (λf. restrict (λg. f (restrict (inv-into UNIV π ∘ g) X)) (X →E Roots))
∈ tuples Y → tuples Y
    by (intro ∗ permutes-inv π)
next
  have ∗: (λg∈X →E Roots. (λg∈X →E Roots. f (restrict (π ∘ g) X))
          (restrict (inv-into UNIV π ∘ g) X)) = f (is ?lhs = -)
    if f: f ∈ tuples Y and π: π permutes Roots for f π
  proof
    fix g show ?lhs g = f g
    proof (cases g ∈ X →E Roots)
      case True
      have restrict (π ∘ restrict (inv-into UNIV π ∘ g) X) X = g
        using True
        by (intro ext) (auto simp: permutes-inverses[OF π])
      thus ?thesis using True
        by (auto simp: permutes-in-image[OF permutes-inv[OF π]])
```

```
    qed (use f in ‹auto simp: tuples-def›)
  qed
  show (λg∈X →E Roots. (λg∈X →E Roots. f (restrict (π ∘ g) X))
          (restrict (inv-into UNIV π ∘ g) X)) = f if f ∈ tuples Y for f
    using *[OF that π] .
  show (λg∈X →E Roots. (λg∈X →E Roots. f (restrict (inv-into UNIV π ∘ g)
X))
          (restrict (π ∘ g) X)) = f if f ∈ tuples Y for f
    using *[OF that permutes-inv[OF π]] permutes-inv-inv[OF π] by simp
qed
```

We show that the coefficients in the expanded new sum are rational – again using the fundamental theorem of symmetric polynomials.

```
define β' :: complex multiset ⇒ complex
  where β' = (λY. ∑f∈tuples Y. ∏g∈X →E Roots. g (f g))

have β' Y ∈ ℚ if Y: Y ∈ Roots-ms for Y
proof −
  define Q :: complex mpoly
    where Q = (∑f∈tuples Y. ∏g∈X →E Roots. Var (unRoot (g (f g))))

  have insertion Root Q ∈ ℚ
  proof (rule symmetric-poly-of-roots-in-subring)
    show ring-closed (ℚ :: complex set)
      by standard auto
    then interpret ring-closed ℚ :: complex set .
    show ∀m. coeff Q m ∈ ℚ
    by (auto simp: Q-def coeff-Var when-def intro!: sum-in-Rats coeff-prod-closed)
  next
    show symmetric-mpoly {..<n} Q
      unfolding symmetric-mpoly-def
    proof safe
      fix π assume π: π permutes {..<n}
      define π' where π' = convert-perm (inv-into UNIV π)
      have π': π' permutes Roots
        unfolding π'-def by (intro permutes-convert-perm permutes-inv π)
       have mpoly-map-vars π Q = (∑f∈tuples Y. ∏g∈X →E Roots. Var (π
(unRoot (g (f g)))))
        unfolding Q-def by (simp add: permutes-bij[OF π])
        also have ... = (∑f∈tuples Y. ∏g∈X →E Roots. Var (unRoot (g (f
(restrict (π' ∘ g) X)))))
      proof (rule sum.cong, goal-cases)
        case (2 f)
        have f: f ∈ (X →E Roots) →E X
          using 2 by (auto simp: tuples-def)
        have (∏g∈X →E Roots. Var (π (unRoot (g (f g))))) =
              (∏g∈X →E Roots. Var (π (unRoot (restrict (π' ∘ g) X (f (restrict
(π' ∘ g) X))))))
       using π' by (intro prod.reindex-bij-betw [symmetric] bij-betw-compose-left-perm-PiE)
```

```
      also have ... = (∏ g∈X →E Roots. Var (unRoot (g (f (restrict (π' ∘ g)
X)))))
      proof (intro prod.cong refl arg-cong[of - - Var])
        fix g assume g: g ∈ X →E Roots
        have restrict (π' ∘ g) X ∈ X →E Roots
            using bij-betw-compose-left-perm-PiE[OF π', of X] g unfolding
bij-betw-def by blast
        hence ∗: f (restrict (π' ∘ g) X) ∈ X
          by (rule PiE-mem[OF f])
        hence ∗∗: g (f (restrict (π' ∘ g) X)) ∈ Roots
          by (rule PiE-mem[OF g])

        have unRoot (restrict (π' ∘ g) X (f (restrict (π' ∘ g) X))) =
              unRoot (Root (inv-into UNIV π (unRoot (g (f (restrict (π' ∘ g)
X))))))
          using ∗ ∗∗ by (subst π'-def) (auto simp: convert-perm-def)
          also have inv-into UNIV π (unRoot (g (f (restrict (π' ∘ g) X)))) ∈
{..<n}
          using ∗∗ by (subst permutes-in-image[OF permutes-inv[OF π]]) auto
        hence unRoot (Root (inv-into UNIV π (unRoot (g (f (restrict (π' ∘ g)
X))))))) =
              inv-into UNIV π (unRoot (g (f (restrict (π' ∘ g) X))))
          by (intro unRoot-Root) auto
        also have π ... = unRoot (g (f (restrict (π' ∘ g) X)))
          by (rule permutes-inverses[OF π])
        finally show π (unRoot (restrict (π' ∘ g) X (f (restrict (π' ∘ g) X)))) =
                  unRoot (g (f (restrict (π' ∘ g) X))) .
      qed
      finally show ?case .
    qed simp-all
    also have ... = (∑ x∈tuples Y. ∏ g∈X →E Roots. Var (unRoot (g ((λg∈X
→E Roots. x (restrict (π' ∘ g) X)) g))))
      by (intro sum.cong prod.cong refl) auto
    also have ... = Q
      unfolding Q-def
      by (rule sum.reindex-bij-betw[OF bij-betw-compose-perm]) (use π' Y in
simp-all)
    finally show mpoly-map-vars π Q = Q .
  qed
next
  show vars Q ⊆ {..<n}
    unfolding Q-def
    by (intro order.trans[OF vars-sum] UN-least order.trans[OF vars-prod])
      (auto simp: vars-Var tuples-def)
next
  define lc where lc = Polynomial.lead-coeff P
  have lc ≠ 0
    unfolding lc-def by auto
  thus inverse (of-int lc) ∗ (of-int lc :: complex) = 1 and inverse (of-int lc) ∈
```

$\mathbb{Q}$
by *auto*
have *rsquarefree* (*of-int-poly P* :: *complex poly*)
using ‹*squarefree* (*of-int-poly P* :: *complex poly*)› **by** (*intro squarefree-imp-rsquarefree*)
hence *of-int-poly P* = *Polynomial.smult* (*of-int lc*) ($\prod$ *x*∈*Roots*. [:−*x*, *1*:])
unfolding *lc-def of-int-hom.hom-lead-coeff*[*symmetric*] *Roots-def*
by (*rule complex-poly-decompose-rsquarefree* [*symmetric*])
also have ($\prod$ *x*∈*Roots*. [:−*x*, *1*:]) = ($\prod$ *i*<*n*. [:−*Root i*, *1*:])
by (*rule prod.reindex-bij-betw*[*OF Root*, *symmetric*])
finally show *of-int-poly P* = *Polynomial.smult* (*of-int lc*) ($\prod$ *i*<*n*. [:− *Root i*, *1*:]) **.**
qed *auto*
also have *insertion Root Q* = ($\sum$ *f*∈*tuples Y*. $\prod$ *g*∈*X* →$_E$ *Roots*. *Root* (*unRoot* (*g* (*f g*))))
by (*simp add*: *Q-def insertion-sum insertion-prod*)
also have ... = β' *Y*
unfolding β'*-def* **by** (*intro sum.cong prod.cong refl Root-unRoot*) (*auto simp*: *tuples-def*)
finally show *?thesis* **.**
qed
hence ∀ *Y*∈*Roots-ms*. ∃ *x*. β' *Y* = *of-rat x*
by (*auto elim*!: *Rats-cases*)
then obtain β'' :: *complex multiset* ⇒ *rat*
where β'': ⋀*Y*. *Y* ∈ *Roots-ms* ⟹ β' *Y* = *of-rat* (β'' *Y*)
by *metis*

We again collect all the terms that happen to have equal exponents and call their coefficients β'':

define β''' :: *complex* ⇒ *rat* **where** β''' = (λα. $\sum$ *Y*∈*Roots-ms*. (β'' *Y when* $\sum_\#$ *Y* = α))
have *supp-*β''': {*x*. β''' *x* ≠ *0*} ⊆ *sum-mset* ‘ *Roots-ms*
by (*auto simp*: β'''*-def when-def elim*!: *sum.not-neutral-contains-not-neutral split*: *if-splits*)

We again start with the sum that we now to be zero and multiply it with all the sums that can be obtained with different choices for the roots.

have *0* = ($\sum$ *x*∈*X*. β *x* ∗ *exp x*)
using *sum0* **..**
also have ... = ($\sum$ *x*∈*X*. *restrict* β *X x* ∗ *exp x*)
by (*intro sum.cong*) *auto*
also have ... *dvd* ($\prod$ *f* ∈ *X* →$_E$ *Roots*. $\sum$ *x*∈*X*. *f x* ∗ *exp x*)
by (*rule dvd-prodI*) (*use* ‹β ‘ *X* ⊆ *Roots*› **in** ‹*auto simp*: *id-def*›)
also have ... = ($\sum$ *f*∈(*X* →$_E$ *Roots*) →$_E$ *X*. $\prod$ *g*∈*X* →$_E$ *Roots*. *g* (*f g*) ∗ *exp* (*f g*))
by (*rule prod-sum-PiE*) *auto*
also have ... = ($\sum$ *f*∈(*X* →$_E$ *Roots*) →$_E$ *X*. ($\prod$ *g*∈*X* →$_E$ *Roots*. *g* (*f g*)) ∗ *exp* ($\sum$ *g*∈*X* →$_E$ *Roots*. *f g*))
by (*simp add*: *prod.distrib exp-sum*)
also have ... = ($\sum$ (*Y*,*f*)∈*Sigma Roots-ms tuples*.

```
                (∏ g∈X →E Roots. g (f g)) * exp (∑ g∈X →E Roots. f g))
    by (intro sum.reindex-bij-witness[of - snd λf. (image-mset f (mset-set (X →E
Roots)), f)])
       (auto simp: Roots-ms-def tuples-def)
  also have ... = (∑ (Y,f)∈Sigma Roots-ms tuples. (∏ g∈X →E Roots. g (f g))
* exp (∑# Y))
    by (intro sum.cong) (auto simp: tuples-def sum-unfold-sum-mset)
  also have ... = (∑ Y∈Roots-ms. β' Y * exp (∑# Y))
    unfolding β'-def sum-distrib-right by (rule sum.Sigma [symmetric]) auto
  also have ... = (∑ Y∈Roots-ms. of-rat (β'' Y) * exp (∑# Y))
    by (intro sum.cong) (auto simp: β'')
  also have ... = (∑ Y∈Roots-ms. Sum-any (λα. of-rat (β'' Y when ∑# Y =
α) * exp α))
  proof (rule sum.cong, goal-cases)
    case (2 Y)
    have Sum-any (λα. of-rat (β'' Y when ∑# Y = α) * exp α) =
          (∑ α∈{∑# Y}. of-rat (β'' Y when ∑# Y = α) * exp α)
      by (intro Sum-any.expand-superset) auto
    thus ?case by simp
  qed auto
  also have ... = Sum-any (λα. of-rat (β''' α) * exp α)
    unfolding β'''-def of-rat-sum sum-distrib-right by (subst Sum-any-sum-swap)
auto
  also have ... = (∑ α | β''' α ≠ 0. of-rat (β''' α) * exp α)
    by (intro Sum-any.expand-superset finite-subset[OF supp-β''']) auto
  finally have (∑ α | β''' α ≠ 0. of-rat (β''' α) * exp α) = 0
    by auto
```

We are now in the situation of our previous version of the theorem and can apply it to find that all the coefficients are zero.

```
  have {α. β''' α ≠ 0} = {}
  proof (rule Hermite-Lindemann-aux3)
    show finite {α. β''' α ≠ 0}
      by (rule finite-subset[OF supp-β'''])  auto
  next
    show (∑ α | β''' α ≠ 0. of-rat (β''' α) * exp α) = 0
    by fact
  next
    fix α assume α: α ∈ {α. β''' α ≠ 0}
    then obtain Y where Y: Y ∈ Roots-ms α = sum-mset Y
      using supp-β''' by auto
    thus algebraic α using alg1
      by (auto simp: Roots-ms-def)
  qed auto
```

However, similarly to before, we can show that the coefficient corresponding to the term with the lexicographically greatest exponent (which is obtained by picking the term with the lexicographically greatest term in each of the factors of our big product) is non-zero.

moreover have $\exists\,\alpha.\ \beta'''\ \alpha \neq 0$
proof −
define α-*max* **where** α-*max* = *complex-lex.Max X*
have [*simp*]: α-*max* ∈ *X*
unfolding α-*max-def* **using** ‹*X* ≠ {}› **by** (*intro complex-lex.Max-in*) *auto*
define *Y-max* :: *complex multiset* **where** *Y-max* = *replicate-mset* (*n* ^ *m*) α-*max*
define *f-max* **where** *f-max* = *restrict* (λ-. α-*max*) (*X* →$_E$ *Roots*)
have [*simp*]: *Y-max* ∈ *Roots-ms*
by (*auto simp*: *Y-max-def Roots-ms-def*)
have *tuples Y-max* = {*f-max*}
proof *safe*
have *image-mset* (λ-∈*X* →$_E$ *Roots*. α-*max*) (*mset-set* (*X* →$_E$ *Roots*)) =
image-mset (λ-. α-*max*) (*mset-set* (*X* →$_E$ *Roots*))
by (*intro image-mset-cong*) *auto*
thus *f-max* ∈ *tuples Y-max*
by (*auto simp*: *f-max-def tuples-def Y-max-def image-mset-const-eq*)
next
fix *f* **assume** *f* ∈ *tuples Y-max*
hence *f*: *f* ∈ (*X* →$_E$ *Roots*) →$_E$ *X image-mset f* (*mset-set* (*X* →$_E$ *Roots*)) = *Y-max*
by (*auto simp*: *tuples-def*)
hence ∀ *g* ∈# *mset-set* (*X* →$_E$ *Roots*). *f g* = α-*max*
by (*intro image-mset-eq-replicate-msetD*[**where** *n* = *n* ^ *m*]) (*auto simp*: *Y-max-def*)
thus *f* = *f-max*
using *f* **by** (*auto simp*: *Y-max-def fun-eq-iff f-max-def*)
qed

have β''' (*of-nat* (*n* ^ *m*) ∗ α-*max*) = (∑ *Y*∈*Roots-ms*. β'' *Y when* ∑$_\#$ *Y* = *of-nat* (*n* ^ *m*) ∗ α-*max*)
unfolding β'''-*def Roots-ms-def* **..**
also have ∑$_\#$ *Y* ≠ *of-nat n* ^ *m* ∗ α-*max* **if** *Y* ∈ *Roots-ms Y* ≠ *Y-max* **for** *Y*
proof −
have ¬*set-mset Y* ⊆ {α-*max*}
using *set-mset-subset-singletonD*[*of Y* α-*max*] *that*
by (*auto simp*: *Roots-ms-def Y-max-def split*: *if-splits*)
then obtain *y* **where** *y*: *y* ∈# *Y y* ≠ α-*max*
by *auto*
have *y* ∈ *X set-mset* (*Y* − {#*y*#}) ⊆ *X*
using *y that* **by** (*auto simp*: *Roots-ms-def dest*: *in-diffD*)
hence *y* ≤$_{\mathbb{C}}$ α-*max*
using *y* **unfolding** α-*max-def* **by** (*intro complex-lex.Max-ge*) *auto*
with *y* **have** *y* <$_{\mathbb{C}}$ α-*max*
by *auto*
have ∗: *Y* = {#*y*#} + (*Y* − {#*y*#})
using *y* **by** *simp*
have *sum-mset Y* = *y* + *sum-mset* (*Y* − {#*y*#})

```
      by (subst *) auto
    also have ... <_c α-max + sum-mset (Y - {#y#})
      by (intro complex-lex.add-strict-right-mono) fact
    also have ... ≤_c α-max + sum-mset (replicate-mset (n ^ m - 1) α-max)
      unfolding α-max-def using that y ‹set-mset (Y - {#y#}) ⊆ X›
      by (intro complex-lex.add-left-mono sum-mset-mono-complex-lex
            rel-mset-replicate-mset-right complex-lex.Max-ge)
         (auto simp: Roots-ms-def size-Diff-singleton)
    also have ... = of-nat (Suc (n ^ m - 1)) * α-max
      by (simp add: algebra-simps)
    also have Suc (n ^ m - 1) = n ^ m
      using ‹n > 0› by simp
    finally show ?thesis by simp
  qed
  hence (∑ Y∈Roots-ms. β'' Y when ∑# Y = of-nat (n ^ m) * α-max) =
(∑ Y∈{Y-max}. β'' Y when ∑# Y = of-nat (n ^ m) * α-max)
    by (intro sum.mono-neutral-right ballI) auto
  also have ... = β'' Y-max
    by (auto simp: when-def Y-max-def)
  also have of-rat ... = β' Y-max
    using β''[of Y-max] by auto
  also have ... = (∏ g∈X →_E Roots. g (f-max g))
    by (auto simp: β'-def ‹tuples Y-max = {f-max}›)
  also have ... = (∏ g∈X →_E Roots. g α-max)
    by (intro prod.cong) (auto simp: f-max-def)
  also have ... ≠ 0
    using ‹0 ∉ Roots› ‹α-max ∈ X› by (intro prod-nonzeroI) (metis PiE-mem)
  finally show ?thesis by blast
 qed

 ultimately show False by blast
qed
```

7.4 The final theorem

We now additionally allow some of the β_i to be zero:

```
lemma Hermite-Lindemann':
 fixes β :: complex ⇒ complex
 assumes finite X
 assumes ⋀x. x ∈ X ⟹ algebraic x
 assumes ⋀x. x ∈ X ⟹ algebraic (β x)
 assumes (∑ x∈X. β x * exp x) = 0
 shows   ∀ x∈X. β x = 0
proof -
 have {x∈X. β x ≠ 0} = {}
 proof (rule Hermite-Lindemann-aux4)
   have (∑ x | x ∈ X ∧ β x ≠ 0. β x * exp x) = (∑ x∈X. β x * exp x)
     by (intro sum.mono-neutral-left assms(1)) auto
   also have ... = 0
```

```
    by fact
  finally show (∑ x | x ∈ X ∧ β x ≠ 0. β x * exp x) = 0 .
 qed (use assms in auto)
 thus ?thesis by blast
qed
```

Lastly, we switch to indexed summation in order to obtain a version of the theorem that is somewhat nicer to use:

```
theorem Hermite-Lindemann:
 fixes α β :: 'a ⇒ complex
 assumes finite I
 assumes ⋀x. x ∈ I ⟹ algebraic (α x)
 assumes ⋀x. x ∈ I ⟹ algebraic (β x)
 assumes inj-on α I
 assumes (∑ x∈I. β x * exp (α x)) = 0
 shows   ∀ x∈I. β x = 0
proof −
 define f where f = inv-into I α
 have [simp]: f (α x) = x if x ∈ I for x
   using that by (auto simp: f-def inv-into-f-f[OF assms(4)])
 have ∀ x∈α ' I. (β ∘ f) x = 0
 proof (rule Hermite-Lindemann')
   have 0 = (∑ x∈I. β x * exp (α x))
     using assms(5) ..
   also have ... = (∑ x∈I. (β ∘ f) (α x) * exp (α x))
     by (intro sum.cong) auto
   also have ... = (∑ x∈α ' I. (β ∘ f) x * exp x)
     using assms(4) by (subst sum.reindex) auto
   finally show (∑ x∈α ' I. (β ∘ f) x * exp x) = 0 ..
 qed (use assms in auto)
 thus ?thesis by auto
qed
```

The following version using lists instead of sequences is even more convenient to use in practice:

```
corollary Hermite-Lindemann-list:
 fixes xs :: (complex × complex) list
 assumes alg:      ∀ (x,y)∈set xs. algebraic x ∧ algebraic y
 assumes distinct: distinct (map snd xs)
 assumes sum0:     (∑ (c,α)←xs. c * exp α) = 0
 shows   ∀ c∈(fst ' set xs). c = 0
proof −
 define n where n = length xs
 have *: ∀ i∈{..<n}. fst (xs ! i) = 0
 proof (rule Hermite-Lindemann)
   from distinct have inj-on (λi. map snd xs ! i) {..<n}
     by (intro inj-on-nth) (auto simp: n-def)
   also have ?this ⟷ inj-on (λi. snd (xs ! i)) {..<n}
     by (intro inj-on-cong) (auto simp: n-def)
```

```
    finally show inj-on (λi. snd (xs ! i)) {..<n} .
  next
    have 0 = (∑(c,α)←xs. c * exp α)
      using sum0 ..
    also have ... = (∑i<n. fst (xs ! i) * exp (snd (xs ! i)))
      unfolding sum-list-sum-nth
      by (intro sum.cong) (auto simp: n-def case-prod-unfold)
    finally show ... = 0 ..
  next
    fix i assume i: i ∈ {..<n}
    hence (fst (xs ! i), snd (xs ! i)) ∈ set xs
      by (auto simp: n-def)
    with alg show algebraic (fst (xs ! i)) algebraic (snd (xs ! i))
      by blast+
  qed auto

  show ?thesis
  proof (intro ballI, elim imageE)
    fix c x assume cx: c = fst x x ∈ set xs
    then obtain i where i ∈ {..<n} x = xs ! i
      by (auto simp: set-conv-nth n-def)
    with * cx show c = 0 by blast
  qed
qed
```

7.5 The traditional formulation of the theorem

What we proved above was actually Baker's reformulation of the theorem. Thus, we now also derive the original one, which uses linear independence and algebraic independence.

It states that if $\alpha_1, \ldots, \alpha_n$ are algebraic numbers that are linearly independent over $\mathbb{Z}$, then $e^{\alpha_1}, \ldots, e^{\alpha_n}$ are algebraically independent over $\mathbb{Q}$.

Linear independence over the integers is just independence of a set of complex numbers when viewing the complex numbers as a $\mathbb{Z}$-module.

```
definition linearly-independent-over-int :: 'a :: field-char-0 set ⇒ bool where
  linearly-independent-over-int = module.independent (λr x. of-int r * x)
```

Algebraic independence over the rationals means that the given set X of numbers fulfils no non-trivial polynomial equation with rational coefficients, i.e. there is no non-zero multivariate polynomial with rational coefficients that, when inserting the numbers from X, becomes zero.

Note that we could easily replace 'rational coefficients' with 'algebraic coefficients' here and the proof would still go through without any modifications.

```
definition algebraically-independent-over-rat :: nat ⇒ (nat ⇒ 'a :: field-char-0)
⇒ bool where
  algebraically-independent-over-rat n a ⟷
```

$(\forall\, p.\ vars\ p \subseteq \{..<n\} \wedge (\forall\, m.\ coeff\ p\ m \in \mathbb{Q}) \wedge insertion\ a\ p = 0 \longrightarrow p = 0)$

corollary *Hermite-Lindemann-original*:
fixes $n :: nat$ **and** $\alpha :: nat \Rightarrow complex$
assumes *inj-on* α $\{..<n\}$
assumes $\bigwedge i.\ i < n \Longrightarrow algebraic\ (\alpha\ i)$
assumes *linearly-independent-over-int* $(\alpha\ `\ \{..<n\})$
shows *algebraically-independent-over-rat* n $(\lambda i.\ exp\ (\alpha\ i))$
unfolding *algebraically-independent-over-rat-def*
proof *safe*
fix p **assume** p: $vars\ p \subseteq \{..<n\}\ \forall\, m.\ coeff\ p\ m \in \mathbb{Q}\ insertion\ (\lambda i.\ exp\ (\alpha\ i))\ p = 0$
define α' **where** $\alpha' = (\lambda m.\ \sum i{<}n.\ of\text{-}nat\ (lookup\ m\ i) * \alpha\ i)$
define I **where** $I = \{m.\ coeff\ p\ m \neq 0\}$

have *lookup-eq-0*: $lookup\ m\ i = 0$ **if** $m \in I\ i \notin \{..<n\}$ **for** $i\ m$
proof −
have $keys\ m \subseteq vars\ p$
using *that coeff-notin-vars*[*of m p*] **by** (*auto simp*: *I-def*)
thus $lookup\ m\ i = 0$
using *in-keys-iff*[*of i m*] *that p*(*1*) **by** *blast*
qed

have $\forall\, x{\in}I.\ coeff\ p\ x = 0$
proof (*rule Hermite-Lindemann*)
show *finite I*
by (*auto simp*: *I-def*)
next
show $algebraic\ (\alpha'\ m)$ **if** $m \in I$ **for** m
unfolding α'*-def* **using** *assms*(*2*) **by** *fastforce*
next
show $algebraic\ (coeff\ p\ m)$ **if** $m \in I$ **for** m
unfolding α'*-def* **using** *p*(*2*) **by** *blast*
next
show *inj-on* $\alpha'\ I$
proof
fix *m1 m2* **assume** *m12*: $m1 \in I\ m2 \in I\ \alpha'\ m1 = \alpha'\ m2$
define *lu* :: $(nat \Rightarrow_0 nat) \Rightarrow nat \Rightarrow int$ **where** $lu = (\lambda m\ i.\ int\ (lookup\ m\ i))$
interpret *int*: *Modules.module* $\lambda r\ x.\ of\text{-}int\ r * (x :: complex)$
by *standard* (*auto simp*: *algebra-simps of-rat-mult of-rat-add*)
define *idx* **where** $idx = inv\text{-}into\ \{..<n\}\ \alpha$

have $lu\ m1\ i = lu\ m2\ i$ **if** $i < n$ **for** i
proof −
have $lu\ m1\ (idx\ (\alpha\ i)) - lu\ m2\ (idx\ (\alpha\ i)) = 0$
proof (*rule int.independentD*)
show *int.independent* $(\alpha\ `\ \{..<n\})$
using *assms*(*3*) **by** (*simp add*: *linearly-independent-over-int-def*)
next

```
        have (∑ x∈α ‘{..<n}. of-int (lu m1 (idx x) − lu m2 (idx x)) ∗ x) =
              (∑ i<n. of-int (lu m1 (idx (α i)) − lu m2 (idx (α i))) ∗ α i)
          using assms(1) by (subst sum.reindex) auto
        also have ... = (∑ i<n. of-int (lu m1 i − lu m2 i) ∗ α i)
          by (intro sum.cong) (auto simp: idx-def inv-into-f-f[OF assms(1)])
        also have ... = 0
          using m12 by (simp add: α'-def ring-distribs of-rat-diff sum-subtractf
lu-def)
        finally show (∑ x∈α ‘{..<n}. of-int (lu m1 (idx x) − lu m2 (idx x)) ∗ x)
= 0
          by (simp add: α'-def ring-distribs of-rat-diff sum-subtractf lu-def)
      qed (use that in auto)
      thus ?thesis
        using that by (auto simp: idx-def inv-into-f-f[OF assms(1)])
    qed
    hence lookup m1 i = lookup m2 i for i
      using m12 by (cases i < n) (auto simp: lu-def lookup-eq-0)
    thus m1 = m2
      by (rule poly-mapping-eqI)
  qed
 next
  have 0 = insertion (λi. exp (α i)) p
    using p(3) ..
  also have ... = (∑ m∈I. coeff p m ∗ Prod-any (λi. exp (α i) ^ lookup m i))
    unfolding insertion-altdef by (rule Sum-any.expand-superset) (auto simp:
I-def)
  also have ... = (∑ m∈I. coeff p m ∗ exp (α' m))
  proof (intro sum.cong, goal-cases)
    case (2 m)
    have Prod-any (λi. exp (α i) ^ lookup m i) = (∏ i<n. exp (α i) ^ lookup m
i)
      using 2 lookup-eq-0[of m] by (intro Prod-any.expand-superset; force)
    also have ... = exp (α' m)
      by (simp add: exp-sum exp-of-nat-mult α'-def)
    finally show ?case by simp
  qed simp-all
  finally show (∑ m∈I. coeff p m ∗ exp (α' m)) = 0 ..
 qed
 thus p = 0
  by (intro mpoly-eqI) (auto simp: I-def)
qed
```

7.6 Simple corollaries

Now, we derive all the usual obvious corollaries of the theorem in the obvious way.

First, the exponential of a non-zero algebraic number is transcendental.

corollary *algebraic-exp-complex-iff*:

assumes *algebraic x*
shows *algebraic* (*exp x* :: *complex*) ⟷ *x = 0*
using *Hermite-Lindemann-list*[*of* [(*1*, *x*), (−*exp x*, *0*)]] *assms* **by** *auto*

More generally, any sum of exponentials with algebraic coefficients and exponents is transcendental if the exponents are all distinct and non-zero and at least one coefficient is non-zero.

corollary *sum-of-exp-transcendentalI*:
fixes *xs* :: (*complex* × *complex*) *list*
assumes ∀(*x*,*y*)∈*set xs*. *algebraic x* ∧ *algebraic y* ∧ *y* ≠ *0*
assumes ∃*x*∈*fst*‘*set xs*. *x* ≠ *0*
assumes *distinct*: *distinct* (*map snd xs*)
shows ¬*algebraic* (∑(*c*,*α*)←*xs*. *c* ∗ *exp α*)
proof
define *S* **where** *S* = (∑(*c*,*α*)←*xs*. *c* ∗ *exp α*)
assume *S*: *algebraic S*
have ∀*c*∈*fst*‘*set* ((−*S*,*0*) # *xs*). *c* = *0*
proof (*rule Hermite-Lindemann-list*)
show (∑(*c*, *α*)←(− *S*, *0*) # *xs*. *c* ∗ *exp α*) = *0*
by (*auto simp*: *S-def*)
qed (*use S assms* **in** *auto*)
with *assms*(*2*) **show** *False*
by *auto*
qed

Any complex logarithm of an algebraic number other than 1 is transcendental (no matter which branch cut).

corollary *transcendental-complex-logarithm*:
assumes *algebraic x exp y* = (*x* :: *complex*) *x* ≠ *1*
shows ¬*algebraic y*
using *algebraic-exp-complex-iff*[*of y*] *assms* **by** *auto*

In particular, this holds for the standard branch of the logarithm.

corollary *transcendental-Ln*:
assumes *algebraic x x* ≠ *0 x* ≠ *1*
shows ¬*algebraic* (*Ln x*)
by (*rule transcendental-complex-logarithm*) (*use assms* **in** *auto*)

The transcendence of e and π, which I have already formalised directly in other AFP entries, now follows as a simple corollary.

corollary *exp-1-complex-transcendental*: ¬*algebraic* (*exp 1* :: *complex*)
by (*subst algebraic-exp-complex-iff*) *auto*

corollary *pi-transcendental*: ¬*algebraic pi*
proof −
have ¬*algebraic* (i ∗ *pi*)
by (*rule transcendental-complex-logarithm*[*of* −*1*]) *auto*
thus *?thesis* **by** *simp*
qed

7.7 Transcendence of the trigonometric and hyperbolic functions

In a similar fashion, we can also prove the transcendence of all the trigonometric and hyperbolic functions such as sin, tan, sinh, arcsin, etc.

lemma *transcendental-sinh*:
  assumes *algebraic z z* ≠ *0*
  shows ¬*algebraic* (*sinh z* :: *complex*)
proof −
  have ¬*algebraic* (∑ (*a*,*b*)←[(*1*/*2*, *z*), (−*1*/*2*, −*z*)]. *a* ∗ *exp b*)
    using *assms* **by** (*intro sum-of-exp-transcendentalI*) *auto*
  also have (∑ (*a*,*b*)←[(*1*/*2*, *z*), (−*1*/*2*, −*z*)]. *a* ∗ *exp b*) = *sinh z*
    by (*simp add*: *sinh-def field-simps scaleR-conv-of-real*)
  finally show *?thesis* **.**
qed

lemma *transcendental-cosh*:
  assumes *algebraic z z* ≠ *0*
  shows ¬*algebraic* (*cosh z* :: *complex*)
proof −
  have ¬*algebraic* (∑ (*a*,*b*)←[(*1*/*2*, *z*), (*1*/*2*, −*z*)]. *a* ∗ *exp b*)
    using *assms* **by** (*intro sum-of-exp-transcendentalI*) *auto*
  also have (∑ (*a*,*b*)←[(*1*/*2*, *z*), (*1*/*2*, −*z*)]. *a* ∗ *exp b*) = *cosh z*
    by (*simp add*: *cosh-def field-simps scaleR-conv-of-real*)
  finally show *?thesis* **.**
qed

lemma *transcendental-sin*:
  assumes *algebraic z z* ≠ *0*
  shows ¬*algebraic* (*sin z* :: *complex*)
  unfolding *sin-conv-sinh* **using** *transcendental-sinh*[*of* i ∗ *z*] *assms* **by** *simp*

lemma *transcendental-cos*:
  assumes *algebraic z z* ≠ *0*
  shows ¬*algebraic* (*cos z* :: *complex*)
  unfolding *cos-conv-cosh* **using** *transcendental-cosh*[*of* i ∗ *z*] *assms* **by** *simp*

lemma *tan-square-neq-neg1*: *tan* (*z* :: *complex*) ^ *2* ≠ −*1*
proof
  assume *tan z* ^ *2* = −*1*
  hence *sin z* ^ *2* = −(*cos z* ^ *2*)
    by (*auto simp*: *tan-def divide-simps split*: *if-splits*)
  also have *cos z* ^ *2* = *1* − *sin z* ^ *2*
    by (*simp add*: *cos-squared-eq*)
  finally show *False*
    by *simp*
qed

```
lemma transcendental-tan:
  assumes algebraic z z ≠ 0
  shows   ¬algebraic (tan z :: complex)
proof
  assume algebraic (tan z)

  have nz1: real-of-int n + 1 / 2 ≠ 0 for n
  proof −
    have real-of-int (2 * n + 1) / real-of-int 2 ∉ ℤ
      by (intro fraction-not-in-ints) auto
    also have real-of-int (2 * n + 1) / real-of-int 2 = real-of-int n + 1 / 2
      by simp
    finally show ... ≠ 0
      by auto
  qed

  have nz2: 1 + tan z ^ 2 ≠ 0
    using tan-square-neq-neg1[of z] by (subst add-eq-0-iff)

  have nz3: cos z ≠ 0
  proof
    assume cos z = 0
   then obtain n where z = complex-of-real (real-of-int n * pi) + complex-of-real
pi / 2
      by (subst (asm) cos-eq-0) blast
    also have ... = complex-of-real ((real-of-int n + 1 / 2) * pi)
      by (simp add: ring-distribs)
    also have algebraic ... ⟷ algebraic ((real-of-int n + 1 / 2) * pi)
      by (rule algebraic-of-real-iff)
    also have ¬algebraic ((real-of-int n + 1 / 2) * pi)
      using nz1[of n] transcendental-pi by simp
    finally show False using assms(1) by contradiction
  qed

  from nz3 have *: sin z ^ 2 = tan z ^ 2 * cos z ^ 2
    by (simp add: tan-def field-simps)
  also have cos z ^ 2 = 1 − sin z ^ 2
    by (simp add: cos-squared-eq)
  finally have sin z ^ 2 * (1 + tan z ^ 2) = tan z ^ 2
    by (simp add: algebra-simps)
  hence sin z ^ 2 = tan z ^ 2 / (1 + tan z ^ 2)
    using nz2 by (simp add: field-simps)
  also have algebraic (tan z ^ 2 / (1 + tan z ^ 2))
    using ‹algebraic (tan z)› by auto
  finally have algebraic (sin z ^ 2) .
  hence algebraic (sin z)
    by simp
  thus False
    using transcendental-sin[OF ‹algebraic z› ‹z ≠ 0›] by contradiction
```

qed

lemma *transcendental-cot*:
assumes *algebraic z z* $\neq$ *0*
shows $\neg$*algebraic* (*cot z* :: *complex*)
proof −
have $\neg$*algebraic* (*tan z*)
by (*rule transcendental-tan*) *fact*+
also have *algebraic* (*tan z*) $\longleftrightarrow$ *algebraic* (*inverse* (*tan z*))
by *simp*
also have *inverse* (*tan z*) = *cot z*
by (*simp add*: *cot-def tan-def*)
finally show *?thesis* **.**
qed

lemma *transcendental-tanh*:
assumes *algebraic z z* $\neq$ *0 cosh z* $\neq$ *0*
shows $\neg$*algebraic* (*tanh z* :: *complex*)
using *transcendental-tan*[*of* i ∗ *z*] *assms* **unfolding** *tanh-conv-tan* **by** *simp*

lemma *transcendental-Arcsin*:
assumes *algebraic z z* $\neq$ *0*
shows $\neg$*algebraic* (*Arcsin z*)
proof −
have i ∗ *z* + *csqrt* (*1* − z^2) $\neq$ *0*
using *Arcsin-body-lemma* **by** *blast*
moreover have i ∗ *z* + *csqrt* (*1* − z^2) $\neq$ *1*
proof
assume i ∗ *z* + *csqrt* (*1* − z^2) = *1*
hence *Arcsin z* = *0*
by (*simp add*: *Arcsin-def*)
hence *sin* (*Arcsin z*) = *0*
by (*simp only*: *sin-zero*)
also have *sin* (*Arcsin z*) = *z*
by *simp*
finally show *False* **using** ‹*z* $\neq$ *0*› **by** *simp*
qed
ultimately have ¬ *algebraic* (*Ln* (i ∗ *z* + *csqrt* (*1* − z^2)))
using *assms* **by** (*intro transcendental-Ln*) *auto*
thus *?thesis*
by (*simp add*: *Arcsin-def*)
qed

lemma *transcendental-Arccos*:
assumes *algebraic z z* $\neq$ *1*
shows $\neg$*algebraic* (*Arccos z*)
proof −
have *z* + i ∗ *csqrt* (*1* − z^2) $\neq$ *0*
using *Arccos-body-lemma* **by** *blast*

```
  moreover have z + i * csqrt (1 - z^2) ≠ 1
  proof
    assume z + i * csqrt (1 - z^2) = 1
    hence Arccos z = 0
      by (simp add: Arccos-def)
    hence cos (Arccos z) = 1
      by (simp only: cos-zero)
    also have cos (Arccos z) = z
      by simp
    finally show False using ‹z ≠ 1› by simp
  qed
  ultimately have ¬ algebraic (Ln (z + i * csqrt (1 - z^2)))
    using assms by (intro transcendental-Ln) auto
  thus ?thesis
    by (simp add: Arccos-def)
qed

lemma transcendental-Arctan:
  assumes algebraic z z ∉ {0, i, -i}
  shows   ¬algebraic (Arctan z)
proof -
  have i * z ≠ 1 1 + i * z ≠ 0
    using assms(2) by (auto simp: complex-eq-iff)
  hence ¬ algebraic (Ln ((1 - i * z) / (1 + i * z)))
    using assms by (intro transcendental-Ln) auto
  thus ?thesis
    by (simp add: Arctan-def)
qed

end
```

References

[1] A. Baker. *Transcendental Number Theory*. Cambridge Mathematical Library. Cambridge University Press, 1975.

[2] S. Bernard. Formalization of the Lindemann-Weierstrass theorem. In M. Ayala-Rincón and C. A. Muñoz, editors, *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*, volume 10499 of *Lecture Notes in Computer Science*, pages 65–80. Springer, 2017.

[3] R. Steinberg and R. M. Redheffer. Analytic proof of the Lindemann theorem. *Pacific Journal of Mathematics*, 2(2):231–242, 1952.