Power Sum Polynomials and the Girard–Newton Theorem

Manuel Eberl

March 17, 2025

Abstract

This article provides a formalisation of the symmetric multivariate polynomials known as *power sum polynomials*. These are of the form $p_n(X_1, \ldots, X_k) = X_1^n + \ldots + X_k^n$. A formal proof of the Girard–Newton Theorem is also given. This theorem relates the power sum polynomials to the elementary symmetric polynomials s_k in the form of a recurrence relation $(-1)^k k s_k = \sum_{i=0}^{k-1} (-1)^i s_i p_{k-i}$.

As an application, this is then used to solve a generalised form of a puzzle given as an exercise in Dummit and Foote's *Abstract Algebra*: For k complex unknowns $x_1, \ldots, x_k$, define $p_j := x_1^j + \ldots + x_k^j$. Then for each vector $a \in \mathbb{C}^k$, show that there is exactly one solution to the system $p_1 = a_1, \ldots, p_k = a_k$ up to permutation of the x_i and determine the value of p_i for $i > k$.

Contents

1 Auxiliary material

theory *Power-Sum-Polynomials-Library*
imports
  Polynomial-Factorization.Fundamental-Theorem-Algebra-Factorized
  Symmetric-Polynomials.Symmetric-Polynomials
  HOL−Computational-Algebra.Computational-Algebra
begin

unbundle *multiset.lifting*

1.1 Miscellaneous

lemma *atLeastAtMost-nat-numeral*:
  atLeastAtMost m (*numeral k* :: *nat*) =
    (*if m* ≤ *numeral k then insert* (*numeral k*) (*atLeastAtMost m* (*pred-numeral k*))
        else {})
  by (*simp add*: *numeral-eq-Suc atLeastAtMostSuc-conv*)

lemma *sum-in-Rats* [*intro*]: (⋀*x*. *x* ∈ *A* ⟹ *f x* ∈ ℚ) ⟹ *sum f A* ∈ ℚ
  by (*induction A rule*: *infinite-finite-induct*) *auto*

lemma (**in** *monoid-mult*) *prod-list-distinct-conv-prod-set*:
  distinct xs ⟹ *prod-list* (*map f xs*) = *prod f* (*set xs*)
  by (*induct xs*) *simp-all*

lemma (**in** *monoid-mult*) *interv-prod-list-conv-prod-set-nat*:
  prod-list (*map f* [*m*..<*n*]) = *prod f* (*set* [*m*..<*n*])
  by (*simp add*: *prod-list-distinct-conv-prod-set*)

lemma (**in** *monoid-mult*) *prod-list-prod-nth*:
  prod-list xs = (∏ *i* = *0*..< *length xs*. *xs* ! *i*)
  using *interv-prod-list-conv-prod-set-nat* [*of* (!) *xs 0 length xs*] **by** (*simp add*: *map-nth*)

lemma *gcd-poly-code-aux-reduce*:
  gcd-poly-code-aux p 0 = *normalize p*
  q ≠ *0* ⟹ *gcd-poly-code-aux p q* = *gcd-poly-code-aux q* (*primitive-part* (*pseudo-mod p q*))
  by (*subst gcd-poly-code-aux.simps*; *simp*)+

lemma *coprimeI-primes*:
  fixes *a b* :: ′*a* :: *factorial-semiring*
  assumes *a* ≠ *0* ∨ *b* ≠ *0*
  assumes ⋀*p*. *prime p* ⟹ *p dvd a* ⟹ *p dvd b* ⟹ *False*
  shows *coprime a b*
proof (*rule coprimeI*)

```
  fix d assume d: d dvd a d dvd b
  with assms(1) have [simp]: d ≠ 0 by auto
  show is-unit d
  proof (rule ccontr)
    assume ¬is-unit d
    then obtain p where p: prime p p dvd d
      using prime-divisor-exists[of d] by auto
    from assms(2)[of p] and p and d show False
      using dvd-trans by auto
  qed
qed

lemma coprime-pderiv-imp-squarefree:
  assumes coprime p (pderiv p)
  shows   squarefree p
proof (rule squarefreeI)
  fix d assume d: d ^ 2 dvd p
  then obtain q where q: p = d ^ 2 * q
    by (elim dvdE)
  hence d dvd p d dvd pderiv p
    by (auto simp: pderiv-mult pderiv-power-Suc numeral-2-eq-2)
  with assms show is-unit d
    using not-coprimeI by blast
qed

lemma squarefree-field-poly-iff:
  fixes p :: 'a :: {field-char-0,euclidean-ring-gcd,semiring-gcd-mult-normalize} poly
  assumes [simp]: p ≠ 0
  shows squarefree p ⟷ coprime p (pderiv p)
proof
  assume squarefree p
  show coprime p (pderiv p)
  proof (rule coprimeI-primes)
    fix d assume d: d dvd p d dvd pderiv p prime d
    from d(1) obtain q where q: p = d * q
      by (elim dvdE)
    from d(2) and q have d dvd q * pderiv d
      by (simp add: pderiv-mult dvd-add-right-iff)
    with ‹prime d› have d dvd q ∨ d dvd pderiv d
      using prime-dvd-mult-iff by blast
    thus False
    proof
      assume d dvd q
      hence d ^ 2 dvd p
        by (auto simp: q power2-eq-square)
      with ‹squarefree p› show False
        using d(3) not-prime-unit squarefreeD by blast
    next
      assume d dvd pderiv d
```

hence *Polynomial.degree* $d = 0$ **by** *simp*
moreover have $d \neq 0$ **using** *d* **by** *auto*
ultimately show *False*
using *d*(*3*) *is-unit-iff-degree not-prime-unit* **by** *blast*
qed
qed *auto*
qed (*use coprime-pderiv-imp-squarefree*[*of p*] **in** *auto*)

lemma *coprime-pderiv-imp-rsquarefree*:
assumes *coprime* (*p* :: *'a* :: *field-char-0 poly*) (*pderiv p*)
shows *rsquarefree p*
unfolding *rsquarefree-roots*
proof *safe*
fix *x* **assume** *poly p x* = *0 poly* (*pderiv p*) *x* = *0*
hence [:−*x*, *1*:] *dvd p* [:−*x*, *1*:] *dvd pderiv p*
by (*auto simp*: *poly-eq-0-iff-dvd*)
with *assms* **have** *is-unit* [:−*x*, *1*:]
using *not-coprimeI* **by** *blast*
thus *False* **by** *auto*
qed

lemma *poly-of-nat* [*simp*]: *poly* (*of-nat n*) *x* = *of-nat n*
by (*induction n*) *auto*

lemma *poly-of-int* [*simp*]: *poly* (*of-int n*) *x* = *of-int n*
by (*cases n*) *auto*

lemma *order-eq-0-iff*: $p \neq 0 \Longrightarrow$ *order x p* = *0* $\longleftrightarrow$ *poly p x* $\neq$ *0*
by (*auto simp*: *order-root*)

lemma *order-pos-iff*: $p \neq 0 \Longrightarrow$ *order x p* > *0* $\longleftrightarrow$ *poly p x* = *0*
by (*auto simp*: *order-root*)

lemma *order-prod*:
assumes $\bigwedge x.\ x \in A \Longrightarrow f\ x \neq 0$
shows *order x* ($\prod y{\in}A.\ f\ y$) = ($\sum y{\in}A.$ *order x* (*f y*))
using *assms* **by** (*induction A rule*: *infinite-finite-induct*) (*auto simp*: *order-mult*)

lemma *order-prod-mset*:
assumes $0 \notin\# A$
shows *order x* (*prod-mset A*) = *sum-mset* (*image-mset* (*order x*) *A*)
using *assms* **by** (*induction A*) (*auto simp*: *order-mult*)

lemma *order-prod-list*:
assumes $0 \notin$ *set xs*
shows *order x* (*prod-list xs*) = *sum-list* (*map* (*order x*) *xs*)
using *assms* **by** (*induction xs*) (*auto simp*: *order-mult*)

lemma *order-power*: $p \neq 0 \Longrightarrow$ *order x* (*p* ^ *n*) = *n* ∗ *order x p*

by (*induction n*) (*auto simp*: *order-mult*)

lemma *smult-0-right* [*simp*]: *MPoly-Type.smult p 0 = 0*
by (*transfer*, *transfer*) *auto*

lemma *mult-smult-right* [*simp*]:
fixes *c* :: *'a* :: *comm-semiring-0*
shows *p* ∗ *MPoly-Type.smult c q = MPoly-Type.smult c* (*p* ∗ *q*)
by (*simp add*: *smult-conv-mult mult-ac*)

lemma *mapping-single-eq-iff* [*simp*]:
Poly-Mapping.single a b = Poly-Mapping.single c d ⟷ *b = 0* ∧ *d = 0* ∨ *a = c* ∧ *b = d*
by *transfer* (*unfold fun-eq-iff when-def*, *metis*)

lemma *monom-of-set-plus-monom-of-set*:
assumes *A* ∩ *B* = {} *finite A finite B*
shows *monom-of-set A + monom-of-set B = monom-of-set* (*A* ∪ *B*)
using *assms* **by** *transfer* (*auto simp*: *fun-eq-iff*)

lemma *mpoly-monom-0-eq-Const*: *monom 0 c = Const c*
by (*intro mpoly-eqI*) (*auto simp*: *coeff-monom when-def mpoly-coeff-Const*)

lemma *mpoly-Const-0* [*simp*]: *Const 0 = 0*
by (*intro mpoly-eqI*) (*auto simp*: *mpoly-coeff-Const mpoly-coeff-0*)

lemma *mpoly-Const-1* [*simp*]: *Const 1 = 1*
by (*intro mpoly-eqI*) (*auto simp*: *mpoly-coeff-Const mpoly-coeff-1*)

lemma *mpoly-Const-uminus*: *Const* (−*a*) = −*Const a*
by (*intro mpoly-eqI*) (*auto simp*: *mpoly-coeff-Const*)

lemma *mpoly-Const-add*: *Const* (*a* + *b*) = *Const a* + *Const b*
by (*intro mpoly-eqI*) (*auto simp*: *mpoly-coeff-Const*)

lemma *mpoly-Const-mult*: *Const* (*a* ∗ *b*) = *Const a* ∗ *Const b*
unfolding *mpoly-monom-0-eq-Const* [*symmetric*] *mult-monom* **by** *simp*

lemma *mpoly-Const-power*: *Const* (*a* ^ *n*) = *Const a* ^ *n*
by (*induction n*) (*auto simp*: *mpoly-Const-mult*)

lemma *of-nat-mpoly-eq*: *of-nat n = Const* (*of-nat n*)
proof (*induction n*)
case *0*
have *0* = (*Const 0* :: *'a mpoly*)
by (*intro mpoly-eqI*) (*auto simp*: *mpoly-coeff-Const*)
thus *?case*
by *simp*

next
  case (*Suc n*)
  have *1* + *Const* (*of-nat n*) = *Const* (*1* + *of-nat n*)
    by (*intro mpoly-eqI*) (*auto simp*: *mpoly-coeff-Const mpoly-coeff-1*)
  thus *?case*
    using *Suc* **by** *auto*
qed

lemma *insertion-of-nat* [*simp*]: *insertion f* (*of-nat n*) = *of-nat n*
  by (*simp add*: *of-nat-mpoly-eq*)

lemma *insertion-monom-of-set* [*simp*]:
  insertion f (*monom* (*monom-of-set X*) *c*) = *c* ∗ ($\prod$ *i*∈*X*. *f i*)
proof (*cases finite X*)
  case [*simp*]: *True*
  have *insertion f* (*monom* (*monom-of-set X*) *c*) = *c* ∗ ($\prod$ *a*. *f a* ^(*if a* ∈ *X then 1 else 0*))
    by (*auto simp*: *lookup-monom-of-set*)
  also have ($\prod$ *a*. *f a* ^(*if a* ∈ *X then 1 else 0*)) = ($\prod$ *i*∈*X*. *f i* ^(*if i* ∈ *X then 1 else 0*))
    by (*intro Prod-any.expand-superset*) *auto*
  also have . . . = ($\prod$ *i*∈*X*. *f i*)
    by (*intro prod.cong*) *auto*
  finally show *?thesis* **.**
qed (*auto simp*: *lookup-monom-of-set*)

lemma *symmetric-mpoly-symmetric-sum*:
  assumes $\bigwedge$π. π *permutes A* ⟹ *g* π *permutes X*
  assumes $\bigwedge$*x* π. *x* ∈ *X* ⟹ π *permutes A* ⟹ *mpoly-map-vars* π (*f x*) = *f* (*g* π *x*)
  shows *symmetric-mpoly A* ($\sum$ *x*∈*X*. *f x*)
  unfolding *symmetric-mpoly-def*
proof *safe*
  fix π **assume** π: π *permutes A*
  have *mpoly-map-vars* π (*sum f X*) = ($\sum$ *x*∈*X*. *mpoly-map-vars* π (*f x*))
    by *simp*
  also have . . . = ($\sum$ *x*∈*X*. *f* (*g* π *x*))
    by (*intro sum.cong assms* π *refl*)
  also have . . . = ($\sum$ *x*∈*g* π‘*X*. *f x*)
    using *assms*(*1*)[*OF* π] **by** (*subst sum.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 (*sum f X*) = *sum f X* **.**
qed

lemma *sym-mpoly-0* [*simp*]:
  assumes *finite A*

shows *sym-mpoly A 0 = 1*
using *assms* **by** (*transfer*, *transfer*) (*auto simp*: *fun-eq-iff when-def*)

lemma *sym-mpoly-eq-0* [*simp*]:
assumes $k > card\ A$
shows *sym-mpoly A k = 0*
proof (*transfer fixing*: *A k*, *transfer fixing*: *A k*, *intro ext*)
fix *mon*
have ¬(*finite A* ∧ (∃ *Y*⊆*A*. *card Y = k* ∧ *mon = monom-of-set Y*))
proof *safe*
fix *Y* **assume** *Y*: *finite A* $Y \subseteq A$ *k = card Y mon = monom-of-set Y*
hence $card\ Y \leq card\ A$ **by** (*intro card-mono*) *auto*
with *Y* **and** *assms* **show** *False* **by** *simp*
qed
thus (*if finite A* ∧ (∃ *Y*⊆*A*. *card Y = k* ∧ *mon = monom-of-set Y*) *then 1 else 0*) = *0*
by *auto*
qed

lemma *coeff-sym-mpoly-monom-of-set-eq-0*:
assumes *finite X* $Y \subseteq X$ *card* $Y \neq k$
shows *MPoly-Type.coeff* (*sym-mpoly X k*) (*monom-of-set Y*) = *0*
using *assms finite-subset*[*of - X*] **by** (*auto simp*: *coeff-sym-mpoly*)

lemma *coeff-sym-mpoly-monom-of-set-eq-0'*:
assumes *finite X* $\neg Y \subseteq X$ *finite Y*
shows *MPoly-Type.coeff* (*sym-mpoly X k*) (*monom-of-set Y*) = *0*
using *assms finite-subset*[*of - X*] **by** (*auto simp*: *coeff-sym-mpoly*)

1.2 The set of roots of a univariate polynomial

lift-definition *poly-roots* :: *'a* :: *idom poly* ⇒ *'a multiset* **is**
λ*p x*. *if p = 0 then 0 else order x p*
proof −
fix *p* :: *'a poly*
show *finite* {*x*. *0* < (*if p = 0 then 0 else order x p*)}
by (*cases p = 0*) (*auto simp*: *order-pos-iff poly-roots-finite*)
qed

lemma *poly-roots-0* [*simp*]: *poly-roots 0* = {#}
by *transfer auto*

lemma *poly-roots-1* [*simp*]: *poly-roots 1* = {#}
by *transfer auto*

lemma *count-poly-roots* [*simp*]:
assumes $p \neq 0$
shows *count* (*poly-roots p*) *x = order x p*
using *assms* **by** *transfer auto*

lemma *in-poly-roots-iff* [*simp*]: $p \neq 0 \Longrightarrow x \in\# \text{ poly-roots } p \longleftrightarrow \text{poly } p\ x = 0$
by (*subst count-greater-zero-iff* [*symmetric*], *subst count-poly-roots*) (*auto simp*: *order-pos-iff*)

lemma *set-mset-poly-roots*: $p \neq 0 \Longrightarrow \text{set-mset (poly-roots } p) = \{x.\ \text{poly } p\ x = 0\}$
using *in-poly-roots-iff*[*of p*] **by** *blast*

lemma *count-poly-roots'*: *count* (*poly-roots p*) *x* = (*if p* = *0 then 0 else order x p*)
by *transfer' auto*

lemma *poly-roots-const* [*simp*]: *poly-roots* [:*c*:] = {#}
by (*intro multiset-eqI*) (*auto simp*: *count-poly-roots' order-eq-0-iff*)

lemma *poly-roots-linear* [*simp*]: *poly-roots* [:$-x$, *1*:] = {#*x*#}
by (*intro multiset-eqI*) (*auto simp*: *count-poly-roots' order-eq-0-iff*)

lemma *poly-roots-monom* [*simp*]: $c \neq 0 \Longrightarrow$ *poly-roots* (*Polynomial.monom c n*) = *replicate-mset n 0*
by (*intro multiset-eqI*) (*auto simp*: *count-poly-roots' order-eq-0-iff poly-monom*)

lemma *poly-roots-smult* [*simp*]: $c \neq 0 \Longrightarrow$ *poly-roots* (*Polynomial.smult c p*) = *poly-roots p*
by (*intro multiset-eqI*) (*auto simp*: *count-poly-roots' order-smult*)

lemma *poly-roots-mult*: $p \neq 0 \Longrightarrow q \neq 0 \Longrightarrow$ *poly-roots* ($p * q$) = *poly-roots p* + *poly-roots q*
by (*intro multiset-eqI*) (*auto simp*: *count-poly-roots' order-mult*)

lemma *poly-roots-prod*:
assumes $\bigwedge x.\ x \in A \Longrightarrow f\ x \neq 0$
shows *poly-roots* (*prod f A*) = ($\sum x \in A$. *poly-roots* (*f x*))
using *assms* **by** (*induction A rule*: *infinite-finite-induct*) (*auto simp*: *poly-roots-mult*)

lemma *poly-roots-prod-mset*:
assumes $0 \notin\# A$
shows *poly-roots* (*prod-mset A*) = *sum-mset* (*image-mset poly-roots A*)
using *assms* **by** (*induction A*) (*auto simp*: *poly-roots-mult*)

lemma *poly-roots-prod-list*:
assumes $0 \notin$ *set xs*
shows *poly-roots* (*prod-list xs*) = *sum-list* (*map poly-roots xs*)
using *assms* **by** (*induction xs*) (*auto simp*: *poly-roots-mult*)

lemma *poly-roots-power*: $p \neq 0 \Longrightarrow$ *poly-roots* (p ^ n) = *repeat-mset n* (*poly-roots p*)
by (*induction n*) (*auto simp*: *poly-roots-mult*)

lemma *rsquarefree-poly-roots-eq*:

assumes *rsquarefree p*
shows *poly-roots p = mset-set {x. poly p x = 0}*
proof (*rule multiset-eqI*)
fix *x* :: *'a*
from *assms* **show** *count* (*poly-roots p*) *x* = *count* (*mset-set* {*x. poly p x = 0*}) *x*
by (*cases poly p x = 0*) (*auto simp*: *poly-roots-finite order-eq-0-iff rsquare-free-def*)
qed

lemma *rsquarefree-imp-distinct-roots*:
assumes *rsquarefree p* **and** *mset xs = poly-roots p*
shows *distinct xs*
proof (*cases p = 0*)
case [*simp*]: *False*
have ∗: *mset xs = mset-set* {*x. poly p x = 0*}
using *assms* **by** (*simp add*: *rsquarefree-poly-roots-eq*)
hence *set-mset* (*mset xs*) = *set-mset* (*mset-set* {*x. poly p x = 0*})
by (*simp only*:)
hence [*simp*]: *set xs* = {*x. poly p x = 0*}
by (*simp add*: *poly-roots-finite*)
from ∗ **show** *?thesis*
by (*subst distinct-count-atmost-1*) (*auto simp*: *poly-roots-finite*)
qed (*use assms* **in** *auto*)

lemma *poly-roots-factorization*:
fixes *p c A*
assumes [*simp*]: *c ≠ 0*
defines *p ≡ Polynomial.smult c* (*prod-mset* (*image-mset* (*λx.* [:−*x*, *1*:]) *A*))
shows *poly-roots p = A*
proof −
have *poly-roots p = poly-roots* (∏ *x*∈#*A*. [:−*x*, *1*:])
by (*auto simp*: *p-def*)
also have . . . = *A*
by (*subst poly-roots-prod-mset*) (*auto simp*: *image-mset.compositionality o-def*)
finally show *?thesis* **.**
qed

lemma *fundamental-theorem-algebra-factorized'*:
fixes *p* :: *complex poly*
shows *p = Polynomial.smult* (*Polynomial.lead-coeff p*)
(*prod-mset* (*image-mset* (*λx.* [:−*x*, *1*:]) (*poly-roots p*)))
proof (*cases p = 0*)
case [*simp*]: *False*
obtain *xs* **where**
xs: *Polynomial.smult* (*Polynomial.lead-coeff p*) (∏ *x*←*xs*. [:−*x*, *1*:]) = *p*
length xs = Polynomial.degree p
using *fundamental-theorem-algebra-factorized*[*of p*] **by** *auto*
define *A* **where** *A = mset xs*

note *xs*(*1*)
also have ($\prod x{\leftarrow}xs.$ [:$-x$, *1*:]) = *prod-mset* (*image-mset* ($\lambda x.$ [:$-x$, *1*:]) *A*)
unfolding *A-def* **by** (*induction xs*) *auto*
finally have ∗: *Polynomial.smult* (*Polynomial.lead-coeff p*) ($\prod x{\in}\#A.$ [:$-$ *x*, *1*:]) = *p* **.**
also have *A* = *poly-roots p*
using *poly-roots-factorization*[*of Polynomial.lead-coeff p A*]
by (*subst* ∗ [*symmetric*]) *auto*
finally show *?thesis* **..**
qed *auto*

lemma *poly-roots-eq-imp-eq*:
fixes *p q* :: *complex poly*
assumes *Polynomial.lead-coeff p* = *Polynomial.lead-coeff q*
assumes *poly-roots p* = *poly-roots q*
shows *p* = *q*
proof (*cases p* = *0* ∨ *q* = *0*)
case *False*
hence [*simp*]: *p* ≠ *0 q* ≠ *0*
by *auto*
have *p* = *Polynomial.smult* (*Polynomial.lead-coeff p*)
(*prod-mset* (*image-mset* ($\lambda x.$ [:$-x$, *1*:]) (*poly-roots p*)))
by (*rule fundamental-theorem-algebra-factorized′*)
also have ... = *Polynomial.smult* (*Polynomial.lead-coeff q*)
(*prod-mset* (*image-mset* ($\lambda x.$ [:$-x$, *1*:]) (*poly-roots q*)))
by (*simp add*: *assms*)
also have ... = *q*
by (*rule fundamental-theorem-algebra-factorized′* [*symmetric*])
finally show *?thesis* **.**
qed (*use assms* **in** *auto*)

lemma *Sum-any-zeroI′*: ($\bigwedge x.$ *P x* ⟹ *f x* = *0*) ⟹ *Sum-any* ($\lambda x.$ *f x when P x*) = *0*
by (*auto simp*: *Sum-any.expand-set*)

lemma *sym-mpoly-insert*:
assumes *finite X x* ∉ *X*
shows (*sym-mpoly* (*insert x X*) (*Suc k*) :: *′a* :: *semiring-1 mpoly*) =
monom (*monom-of-set* {*x*}) *1* ∗ *sym-mpoly X k* + *sym-mpoly X* (*Suc k*) (**is** *?lhs* = *?A* + *?B*)
proof (*rule mpoly-eqI*)
fix *mon*
show *coeff ?lhs mon* = *coeff* (*?A* + *?B*) *mon*
proof (*cases* ∀ *i. lookup mon i* ≤ *1* ∧ (*i* ∉ *insert x X* ⟶ *lookup mon i* = *0*))
case *False*
then obtain *i* **where** *i*: *lookup mon i* > *1* ∨ *i* ∉ *insert x X* ∧ *lookup mon i* > *0*
by (*auto simp*: *not-le*)

have *coeff ?A mon* = *prod-fun* (*coeff* (*monom* (*monom-of-set* {*x*}) *1*))
(*coeff* (*sym-mpoly X k*)) *mon*
by (*simp add*: *coeff-mpoly-times*)
also have ... = ($\sum$ *l*. $\sum$ *q*. *coeff* (*monom* (*monom-of-set* {*x*}) *1*) *l* ∗ *coeff* (*sym-mpoly X k*) *q*
when mon = *l* + *q*)
unfolding *prod-fun-def*
by (*intro Sum-any.cong*, *subst Sum-any-right-distrib*, *force*)
(*auto simp*: *Sum-any-right-distrib when-def intro*!: *Sum-any.cong*)
also have ... = *0*
proof (*rule Sum-any-zeroI*, *rule Sum-any-zeroI'*)
fix *ma mb* **assume** ∗: *mon* = *ma* + *mb*
show *coeff* (*monom* (*monom-of-set* {*x*}) (*1*::'*a*)) *ma* ∗ *coeff* (*sym-mpoly X k*) *mb* = *0*
proof (*cases i* = *x*)
case [*simp*]: *True*
show *?thesis*
proof (*cases lookup mb i* > *0*)
case *True*
hence *coeff* (*sym-mpoly X k*) *mb* = *0* **using** ‹*x* ∉ *X*›
by (*auto simp*: *coeff-sym-mpoly lookup-monom-of-set split*: *if-splits*)
thus *?thesis*
using *mult-not-zero* **by** *blast*
next
case *False*
hence *coeff* (*monom* (*monom-of-set* {*x*}) *1*) *ma* = *0*
using *i* **by** (*auto simp*: *coeff-monom when-def* ∗ *lookup-add*)
thus *?thesis*
using *mult-not-zero* **by** *blast*
qed
next
case [*simp*]: *False*
show *?thesis*
proof (*cases lookup ma i* > *0*)
case *False*
hence *lookup mb i* = *lookup mon i*
using ∗ **by** (*auto simp*: *lookup-add*)
hence *coeff* (*sym-mpoly X k*) *mb* = *0* **using** *i*
by (*auto simp*: *coeff-sym-mpoly lookup-monom-of-set split*: *if-splits*)
thus *?thesis*
using *mult-not-zero* **by** *blast*
next
case *True*
hence *coeff* (*monom* (*monom-of-set* {*x*}) *1*) *ma* = *0*
using *i* **by** (*auto simp*: *coeff-monom when-def* ∗ *lookup-add*)
thus *?thesis*
using *mult-not-zero* **by** *blast*
qed

qed
qed
finally have *coeff ?A mon = 0* **.**
moreover from *False* **have** *coeff ?lhs mon = 0*
by (*subst coeff-sym-mpoly*) (*auto simp*: *lookup-monom-of-set split*: *if-splits*)
moreover from *False* **have** *coeff* (*sym-mpoly X* (*Suc k*)) *mon = 0*
by (*subst coeff-sym-mpoly*) (*auto simp*: *lookup-monom-of-set split*: *if-splits*)
ultimately show *?thesis*
by *auto*
next
case *True*
define *A* **where** *A = keys mon*
have *A*: $A \subseteq$ *insert x X*
using *True* **by** (*auto simp*: *A-def*)
have [*simp*]: *mon = monom-of-set A*
unfolding *A-def* **using** *True* **by** *transfer* (*force simp*: *fun-eq-iff le-Suc-eq*)
have *finite A*
using *finite-subset A assms* **by** *blast*
show *?thesis*
proof (*cases* $x \in A$)
case *False*
have *coeff ?A mon = prod-fun* (*coeff* (*monom* (*monom-of-set* {*x*}) *1*))
(*coeff* (*sym-mpoly X k*)) (*monom-of-set A*)
by (*simp add*: *coeff-mpoly-times*)
also have ... = ($\sum$ *l*. $\sum$ *q*. *coeff* (*monom* (*monom-of-set* {*x*}) *1*) *l* ∗ *coeff* (*sym-mpoly X k*) *q*
when monom-of-set A = l + q)
unfolding *prod-fun-def*
by (*intro Sum-any.cong*, *subst Sum-any-right-distrib*, *force*)
(*auto simp*: *Sum-any-right-distrib when-def intro*!: *Sum-any.cong*)
also have ... = *0*
proof (*rule Sum-any-zeroI*, *rule Sum-any-zeroI'*)
fix *ma mb* **assume** ∗: *monom-of-set A = ma + mb*
hence *keys ma* ⊆ *A*
using ‹*finite A*› **by** *transfer* (*auto simp*: *fun-eq-iff split*: *if-splits*)
thus *coeff* (*monom* (*monom-of-set* {*x*}) (*1*::'*a*)) *ma* ∗ *coeff* (*sym-mpoly X k*) *mb = 0*
using ‹$x \notin A$› **by** (*auto simp*: *coeff-monom when-def*)
qed
finally show *?thesis*
using *False A assms finite-subset*[*of - insert x X*] *finite-subset*[*of - X*]
by (*auto simp*: *coeff-sym-mpoly*)
next
case *True*
have *mon = monom-of-set* {*x*} + *monom-of-set* (*A* − {*x*})
using ‹$x \in A$› ‹*finite A*› **by** (*auto simp*: *monom-of-set-plus-monom-of-set*)
also have *coeff ?A ... = coeff* (*sym-mpoly X k*) (*monom-of-set* (*A* − {*x*}))
by (*subst coeff-monom-mult*) *auto*
also have ... = (*if card A = Suc k then 1 else 0*)

proof (*cases card A = Suc k*)
  case *True*
  thus *?thesis*
    using *assms* ‹*finite A*› ‹*x* ∈ *A*› *A*
    by (*subst coeff-sym-mpoly-monom-of-set*) *auto*
next
  case *False*
  thus *?thesis*
    using *assms* ‹*x* ∈ *A*› *A* ‹*finite A*› *card-Suc-Diff1*[*of A x*]
    by (*subst coeff-sym-mpoly-monom-of-set-eq-0*) *auto*
qed
moreover have *coeff ?B* (*monom-of-set A*) = *0*
  using *assms* ‹*x* ∈ *A*› ‹*finite A*›
  by (*subst coeff-sym-mpoly-monom-of-set-eq-0'*) *auto*
moreover have *coeff ?lhs* (*monom-of-set A*) = (*if card A* = *Suc k then 1 else 0*)
  using *assms A* ‹*finite A*› *finite-subset*[*of - insert x X*] **by** (*auto simp*: *coeff-sym-mpoly*)
ultimately show *?thesis* **by** *simp*
qed
qed
qed

lifting-update *multiset.lifting*
lifting-forget *multiset.lifting*

end

2 Power sum polynomials

theory *Power-Sum-Polynomials*
imports
  Symmetric-Polynomials.Symmetric-Polynomials
  HOL−Computational-Algebra.Field-as-Ring
  Power-Sum-Polynomials-Library
begin

2.1 Definition

For n indeterminates $X_1, \ldots, X_n$, we define the k-th power sum polynomial as

$$p_k(X_1, \ldots, X_n) = X_1^k + \ldots + X_n^k \; .$$

lift-definition *powsum-mpoly-aux* :: *nat set* ⇒ *nat* ⇒ (*nat* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ *'a* :: {*semiring-1*,*zero-neq-one*} **is**
  λX k mon. if infinite X ∨ *k* = *0* ∧ *mon* ≠ *0 then 0*
    else if k = *0* ∧ *mon* = *0 then of-nat* (*card X*)
    else if finite X ∧ (∃ *x*∈*X*. *mon* = *Poly-Mapping.single x k*) *then 1 else 0*

by *auto*

lemma *lookup-powsum-mpoly-aux*:
Poly-Mapping.lookup (*powsum-mpoly-aux X k*) *mon* =
(*if infinite X* ∨ *k* = *0* ∧ *mon* ≠ *0 then 0*
else if k = *0* ∧ *mon* = *0 then of-nat* (*card X*)
else if finite X ∧ (∃ *x*∈*X*. *mon* = *Poly-Mapping.single x k*) *then 1 else 0*)
by *transfer′ simp*

lemma *lookup-sym-mpoly-aux-monom-singleton* [*simp*]:
assumes *finite X x* ∈ *X k* > *0*
shows *Poly-Mapping.lookup* (*powsum-mpoly-aux X k*) (*Poly-Mapping.single x k*) = *1*
using *assms* **by** (*auto simp*: *lookup-powsum-mpoly-aux*)

lemma *lookup-sym-mpoly-aux-monom-singleton′*:
assumes *finite X k* > *0*
shows *Poly-Mapping.lookup* (*powsum-mpoly-aux X k*) (*Poly-Mapping.single x k*) = (*if x* ∈ *X then 1 else 0*)
using *assms* **by** (*auto simp*: *lookup-powsum-mpoly-aux*)

lemma *keys-powsum-mpoly-aux*: *m* ∈ *keys* (*powsum-mpoly-aux A k*) ⟹ *keys m* ⊆ *A*
by *transfer′* (*auto split*: *if-splits simp*: *keys-monom-of-set*)

lift-definition *powsum-mpoly* :: *nat set* ⇒ *nat* ⇒ ′*a* :: {*semiring-1*,*zero-neq-one*} *mpoly* **is**
powsum-mpoly-aux **.**

lemma *vars-powsum-mpoly-subset*: *vars* (*powsum-mpoly A k*) ⊆ *A*
using *keys-powsum-mpoly-aux* **by** (*auto simp*: *vars-def powsum-mpoly.rep-eq*)

lemma *powsum-mpoly-infinite*: ¬*finite A* ⟹ *powsum-mpoly A k* = *0*
by (*transfer*, *transfer*) *auto*

lemma *coeff-powsum-mpoly*:
MPoly-Type.coeff (*powsum-mpoly X k*) *mon* =
(*if infinite X* ∨ *k* = *0* ∧ *mon* ≠ *0 then 0*
else if k = *0* ∧ *mon* = *0 then of-nat* (*card X*)
else if finite X ∧ (∃ *x*∈*X*. *mon* = *Poly-Mapping.single x k*) *then 1 else 0*)
by *transfer′* (*simp add*: *lookup-powsum-mpoly-aux*)

lemma *coeff-powsum-mpoly-0-right*:
MPoly-Type.coeff (*powsum-mpoly X 0*) *mon* = (*if mon* = *0 then of-nat* (*card X*) *else 0*)
by *transfer′* (*auto simp add*: *lookup-powsum-mpoly-aux*)

lemma *coeff-powsum-mpoly-singleton*:
assumes *finite X k > 0*
shows *MPoly-Type.coeff* (*powsum-mpoly X k*) (*Poly-Mapping.single x k*) = (*if x ∈ X then 1 else 0*)
using *assms* **by** *transfer′* (*simp add*: *lookup-powsum-mpoly-aux*)

lemma *coeff-powsum-mpoly-singleton-eq-1* [*simp*]:
assumes *finite X x ∈ X k > 0*
shows *MPoly-Type.coeff* (*powsum-mpoly X k*) (*Poly-Mapping.single x k*) = *1*
using *assms* **by** (*simp add*: *coeff-powsum-mpoly-singleton*)

lemma *coeff-powsum-mpoly-singleton-eq-0* [*simp*]:
assumes *finite X x ∉ X k > 0*
shows *MPoly-Type.coeff* (*powsum-mpoly X k*) (*Poly-Mapping.single x k*) = *0*
using *assms* **by** (*simp add*: *coeff-powsum-mpoly-singleton*)

lemma *powsum-mpoly-0* [*simp*]: *powsum-mpoly X 0* = *of-nat* (*card X*)
by (*intro mpoly-eqI ext*) (*auto simp*: *coeff-powsum-mpoly-0-right of-nat-mpoly-eq mpoly-coeff-Const*)

lemma *powsum-mpoly-empty* [*simp*]: *powsum-mpoly* {} *k* = *0*
by (*intro mpoly-eqI*) (*auto simp*: *coeff-powsum-mpoly*)

lemma *powsum-mpoly-altdef*: *powsum-mpoly X k* = (∑ *x*∈*X*. *monom* (*Poly-Mapping.single x k*) *1*)
proof (*cases finite X*)
  case [*simp*]: *True*
  show *?thesis*
  proof (*cases k* = *0*)
    case *True*
    thus *?thesis* **by** *auto*
  next
    case *False*
    show *?thesis*
    proof (*intro mpoly-eqI*, *goal-cases*)
      case (*1 mon*)
      show *?case* **using** *False*
        by (*cases* ∃ *x*∈*X*. *mon* = *Poly-Mapping.single x k*)
          (*auto simp*: *coeff-powsum-mpoly coeff-monom when-def*)
    qed
  qed
qed (*auto simp*: *powsum-mpoly-infinite*)

Power sum polynomials are symmetric:

lemma *symmetric-powsum-mpoly* [*intro*]:
assumes *A ⊆ B*
shows *symmetric-mpoly A* (*powsum-mpoly B k*)
unfolding *powsum-mpoly-altdef*

proof (*rule symmetric-mpoly-symmetric-sum*)
  fix *x* π
  assume $x \in B$ π *permutes A*
  thus *mpoly-map-vars* π (*MPoly-Type.monom* (*Poly-Mapping.single x k*) *1*) =
      MPoly-Type.monom (*Poly-Mapping.single* (π *x*) *k*) *1*
   using *assms* **by** (*auto simp*: *mpoly-map-vars-monom permutes-bij permutep-single bij-imp-bij-inv permutes-inv-inv*)
qed (*use assms* **in** ‹*auto simp*: *permutes-subset*›)

lemma *insertion-powsum-mpoly* [*simp*]: *insertion f* (*powsum-mpoly X k*) = ($\sum$ *i*∈*X*. *f i* ^ *k*)
  unfolding *powsum-mpoly-altdef insertion-sum insertion-single* **by** *simp*

lemma *powsum-mpoly-nz*:
  assumes *finite X X* ≠ {} *k* > *0*
  shows (*powsum-mpoly X k* :: *'a* :: {*semiring-1*, *zero-neq-one*} *mpoly*) ≠ *0*
proof −
  from *assms* **obtain** *x* **where** *x* ∈ *X* **by** *auto*
  hence *coeff* (*powsum-mpoly X k*) (*Poly-Mapping.single x k*) = (*1* :: *'a*)
    using *assms* **by** (*auto simp*: *coeff-powsum-mpoly*)
  thus *?thesis* **by** *auto*
qed

lemma *powsum-mpoly-eq-0-iff*:
  assumes *k* > *0*
  shows *powsum-mpoly X k* = *0* ⟷ *infinite X* ∨ *X* = {}
  using *assms powsum-mpoly-nz*[*of X k*] **by** (*auto simp*: *powsum-mpoly-infinite*)

2.2 The Girard–Newton Theorem

The following is a nice combinatorial proof of the Girard–Newton Theorem due to Doron Zeilberger [2].

The precise statement is this:

Let e_k denote the k-th elementary symmetric polynomial in $X_1, \ldots, X_n$. This is the sum of all monomials that can be formed by taking the product of k distinct variables.

Next, let $p_k = X_1^k + \ldots + X_n^k$ denote that k-th symmetric power sum polynomial in $X_1, \ldots, X_n$.

Then the following equality holds:

$$(-1)^k k e_k + \sum_{i=0}^{k-1} (-1)^i e_i p_{k-i}$$

theorem *Girard-Newton*:
  assumes *finite X*
  shows (−*1*) ^ *k* ∗ *of-nat k* ∗ *sym-mpoly X k* +
      ($\sum$ *i*<*k*. (−*1*) ^ *i* ∗ *sym-mpoly X i* ∗ *powsum-mpoly X* (*k* − *i*)) =

(*0* :: *'a* :: *comm-ring-1 mpoly*)
(**is** *?lhs* = *0*)
proof −
write *Poly-Mapping.single* (‹*sng*›)

define *n* **where** *n* = *card X*
define $\mathcal{A}$ :: (*nat set* × *nat*) *set*
where $\mathcal{A}$ = {(*A*, *j*). *A* ⊆ *X* ∧ *card A* ≤ *k* ∧ *j* ∈ *X* ∧ (*card A* = *k* ⟶ *j* ∈ *A*)}
define $\mathcal{A}$*1* :: (*nat set* × *nat*) *set*
where $\mathcal{A}$*1* = {*A*∈*Pow X*. *card A* < *k*} × *X*
define $\mathcal{A}$*2* :: (*nat set* × *nat*) *set*
where $\mathcal{A}$*2* = (*SIGMA A*:{*A*∈*Pow X*. *card A* = *k*}. *A*)

have $\mathcal{A}$*-split*: $\mathcal{A}$ = $\mathcal{A}$*1* ∪ $\mathcal{A}$*2* $\mathcal{A}$*1* ∩ $\mathcal{A}$*2* = {}
by (*auto simp*: $\mathcal{A}$*-def* $\mathcal{A}$*1-def* $\mathcal{A}$*2-def*)
have [*intro*]: *finite* $\mathcal{A}$*1* *finite* $\mathcal{A}$*2*
using *assms finite-subset*[*of - X*] **by** (*auto simp*: $\mathcal{A}$*1-def* $\mathcal{A}$*2-def intro*!: *finite-SigmaI*)
have [*intro*]: *finite* $\mathcal{A}$
by (*subst* $\mathcal{A}$*-split*) *auto*

— We define a 'weight' function w from $\mathcal{A}$ to the ring of polynomials as

$$w(A,j) = (-1)^{|A|} x_j^{k-|A|} \prod_{i\in A} x_i \ .$$

define *w* :: *nat set* × *nat* ⇒ *'a mpoly*
where *w* = (λ(*A*, *j*). *monom* (*monom-of-set A* + *sng j* (*k* − *card A*)) ((−*1*) ^ *card A*))

— The sum of these weights over all of $\mathcal{A}$ is precisely the sum that we want to show equals 0:
have *?lhs* = (∑*x*∈$\mathcal{A}$. *w x*)
proof −
have (∑*x*∈$\mathcal{A}$. *w x*) = (∑*x*∈$\mathcal{A}$*1*. *w x*) + (∑*x*∈$\mathcal{A}$*2*. *w x*)
by (*subst* $\mathcal{A}$*-split*, *subst sum.union-disjoint*, *use* $\mathcal{A}$*-split*(*2*) **in** *auto*)

also have (∑*x*∈$\mathcal{A}$*1*. *w x*) = (∑*i*<*k*. (−*1*) ^*i* ∗ *sym-mpoly X i* ∗ *powsum-mpoly X* (*k* − *i*))
proof −
have (∑*x*∈$\mathcal{A}$*1*. *w x*) = (∑*A* | *A* ⊆ *X* ∧ *card A* < *k*. ∑*j*∈*X*. *w* (*A*, *j*))
using *assms* **by** (*subst sum.Sigma*) (*auto simp*: $\mathcal{A}$*1-def*)
also have ... = (∑*A* | *A* ⊆ *X* ∧ *card A* < *k*. ∑*j*∈*X*.
monom (*monom-of-set A*) ((−*1*) ^ *card A*) ∗ *monom* (*sng j* (*k* − *card A*)) *1*)
unfolding *w-def* **by** (*intro sum.cong*) (*auto simp*: *mult-monom*)
also have ... = (∑*A* | *A* ⊆ *X* ∧ *card A* < *k*. *monom* (*monom-of-set A*) ((−*1*) ^ *card A*) ∗
powsum-mpoly X (*k* − *card A*))
by (*simp add*: *sum-distrib-left powsum-mpoly-altdef*)

also have ... = (∑(*i*,*A*) ∈ (*SIGMA i*:{..<*k*}. {*A*. *A* ⊆ *X* ∧ *card A* = *i*}).
monom (*monom-of-set A*) ((−*1*) ^ *i*) ∗ *powsum-mpoly X* (*k* − *i*))
by (*rule sum.reindex-bij-witness*[*of* - *snd* λ*A*. (*card A*, *A*)]) *auto*
also have ... = (∑*i*<*k*. ∑*A* | *A* ⊆ *X* ∧ *card A* = *i*.
monom (*monom-of-set A*) *1* ∗ *monom 0* ((−*1*) ^ *i*) ∗ *powsum-mpoly X* (*k* − *i*))
using *assms* **by** (*subst sum.Sigma*) (*auto simp*: *mult-monom*)
also have ... = (∑*i*<*k*. (−*1*) ^ *i* ∗ *sym-mpoly X i* ∗ *powsum-mpoly X* (*k* − *i*))
by (*simp add*: *sum-distrib-left sum-distrib-right mpoly-monom-0-eq-Const*
mpoly-Const-power mpoly-Const-uminus algebra-simps sym-mpoly-altdef)
finally show *?thesis* **.**
qed

also have (∑*x*∈𝒜*2*. *w x*) = (−*1*) ^ *k* ∗ *of-nat k* ∗ *sym-mpoly X k*
proof −
have (∑*x*∈𝒜*2*. *w x*) = (∑(*A*,*j*)∈𝒜*2*. *monom* (*monom-of-set A*) ((− *1*) ^ *k*))
by (*intro sum.cong*) (*auto simp*: 𝒜*2-def w-def mpoly-monom-0-eq-Const intro*!: *sum.cong*)
also have ... = (∑*A* | *A* ⊆ *X* ∧ *card A* = *k*. ∑*j*∈*A*. *monom* (*monom-of-set A*) ((− *1*) ^ *k*))
using *assms finite-subset*[*of* - *X*] **by** (*subst sum.Sigma*) (*auto simp*: 𝒜*2-def*)
also have (λ*A*. *monom* (*monom-of-set A*) ((− *1*) ^ *k*) :: ′*a mpoly*) =
(λ*A*. *monom 0* ((−*1*) ^ *k*) ∗ *monom* (*monom-of-set A*) *1*)
by (*auto simp*: *fun-eq-iff mult-monom*)
also have *monom 0* ((−*1*) ^ *k*) = (−*1*) ^ *k*
by (*auto simp*: *mpoly-monom-0-eq-Const mpoly-Const-power mpoly-Const-uminus*)
also have (∑*A* | *A* ⊆ *X* ∧ *card A* = *k*. ∑*j*∈*A*. (− *1*) ^ *k* ∗ *monom* (*monom-of-set A*) *1*) =
((−*1*) ^ *k* ∗ *of-nat k* ∗ *sym-mpoly X k* :: ′*a mpoly*)
by (*auto simp*: *sum-distrib-left sum-distrib-right mult-ac sym-mpoly-altdef*)
finally show *?thesis* **.**
qed

finally show *?thesis* **by** (*simp add*: *algebra-simps*)
qed

— Next, we show that the weights sum to 0:
also have (∑*x*∈𝒜. *w x*) = *0*
proof −
— We define a function T that is a involutory permutation of $\mathcal{A}$. To be more precise, it bijectively maps those elements (A,j) of $\mathcal{A}$ with $j \in A$ to those where $j \notin A$ and the other way round. 'Involutory' means that T is its own inverse function, i. e. $T(T(x)) = x$.
define *T* :: *nat set* × *nat* ⇒ *nat set* × *nat*
where *T* = (λ(*A*, *j*). *if j* ∈ *A then* (*A* − {*j*}, *j*) *else* (*insert j A*, *j*))

have [*simp*]: $T\ (T\ x) = x$ **for** x
  by (*auto simp*: *T-def split*: *prod.splits*)
have [*simp*]: $T\ x \in \mathcal{A}$ **if** $x \in \mathcal{A}$ **for** x
proof −
  have [*simp*]: $n \leq n - Suc\ 0 \longleftrightarrow n = 0$ **for** n
    by *auto*
  show *?thesis* **using** *that assms finite-subset*[*of - X*]
    by (*auto simp*: *T-def* $\mathcal{A}$*-def split*: *prod.splits*)
qed
have $snd\ (T\ x) \in fst\ (T\ x) \longleftrightarrow snd\ x \notin fst\ x$ **if** $x \in \mathcal{A}$ **for** x
  by (*auto simp*: *T-def split*: *prod.splits*)
hence *bij*: *bij-betw* $T\ \{x{\in}\mathcal{A}.\ snd\ x \in fst\ x\}\ \{x{\in}\mathcal{A}.\ snd\ x \notin fst\ x\}$
  by (*intro bij-betwI*[*of - - - T*]) *auto*

— Crucially, we show that T flips the weight of each element:
have [*simp*]: $w\ (T\ x) = -w\ x$ **if** $x \in \mathcal{A}$ **for** x
proof −
  obtain $A\ j$ **where** [*simp*]: $x = (A, j)$ **by** *force*

  — Since T is an involution, we can assume w. l. o. g. that $j \in A$:
  have *aux*: $w\ (T\ (A, j)) = -\ w\ (A, j)$ **if** $(A, j) \in \mathcal{A}\ j \in A$ **for** $j\ A$
  proof −
    from *that* **have** [*simp*]: $j \in A\ A \subseteq X$ **and** $k > 0$
     using *finite-subset*[*OF - assms*, *of A*] **by** (*auto simp*: $\mathcal{A}$*-def intro*!: *Nat.gr0I*)
    have [*simp*]: *finite A*
      using *finite-subset*[*OF - assms*, *of A*] **by** *auto*
    from *that* **have** $card\ A \leq k$
      by (*auto simp*: $\mathcal{A}$*-def*)

    have *card*: $card\ A = Suc\ (card\ (A - \{j\}))$
      using *card.remove*[*of A j*] **by** *auto*
    hence *card-less*: $card\ (A - \{j\}) < card\ A$ **by** *linarith*

     have $w\ (T\ (A, j)) = monom\ (monom\text{-}of\text{-}set\ (A - \{j\}) + sng\ j\ (k - card\ (A - \{j\})))$
                 $((-\ 1)\ \hat{}\ card\ (A - \{j\}))$ **by** (*simp add*: *w-def T-def*)
    also have $(-\ 1)\ \hat{}\ card\ (A - \{j\}) = ((-\ 1)\ \hat{}\ Suc\ (Suc\ (card\ (A - \{j\})))\ ::\ 'a)$
      by *simp*
    also have $Suc\ (card\ (A - \{j\})) = card\ A$
      using *card* **by** *simp*
    also have $k - card\ (A - \{j\}) = Suc\ (k - card\ A)$
      using ‹$k > 0$› ‹$card\ A \leq k$› *card-less* **by** (*subst card*) *auto*
    also have $monom\text{-}of\text{-}set\ (A - \{j\}) + sng\ j\ (Suc\ (k - card\ A)) =$
            $monom\text{-}of\text{-}set\ A + sng\ j\ (k - card\ A)$
      by (*transfer fixing*: *A j k*) (*auto simp*: *fun-eq-iff*)
    also have $monom\ \ldots\ ((-1)\hat{}\ Suc\ (card\ A)) = -w\ (A, j)$
      by (*simp add*: *w-def monom-uminus*)
    finally show *?thesis* **.**

qed

show *?thesis*
proof (*cases* $j \in A$)
case *True*
with *aux*[*of A j*] *that* **show** *?thesis* **by** *auto*
next
case *False*
hence *snd* (*T x*) $\in$ *fst* (*T x*)
by (*auto simp*: *T-def split*: *prod.splits*)
with *aux*[*of fst* (*T x*) *snd* (*T x*)] *that* **show** *?thesis* **by** *auto*
qed
qed

We can now show fairly easily that the sum is equal to zero.

have $*$: $\mathcal{A} = \{x{\in}\mathcal{A}.\ snd\ x \in fst\ x\} \cup \{x{\in}\mathcal{A}.\ snd\ x \notin fst\ x\}$
by *auto*
have $(\sum x{\in}\mathcal{A}.\ w\ x) = (\sum x \mid x \in \mathcal{A} \wedge snd\ x \in fst\ x.\ w\ x) + (\sum x \mid x \in \mathcal{A} \wedge snd\ x \notin fst\ x.\ w\ x)$
using ‹*finite* $\mathcal{A}$› **by** (*subst* $*$, *subst sum.union-disjoint*) *auto*
also have $(\sum x \mid x \in \mathcal{A} \wedge snd\ x \notin fst\ x.\ w\ x) = (\sum x \mid x \in \mathcal{A} \wedge snd\ x \in fst\ x.\ w\ (T\ x))$
using *sum.reindex-bij-betw*[*OF bij*, *of w*] **by** *simp*
also have $\ldots = -(\sum x \mid x \in \mathcal{A} \wedge snd\ x \in fst\ x.\ w\ x)$
by (*simp add*: *sum-negf*)
finally show $(\sum x{\in}\mathcal{A}.\ w\ x) = 0$
by *simp*
qed

finally show *?thesis* **.**
qed

The following variant of the theorem holds for $k > n$. Note that this is now a linear recurrence relation with constant coefficients for p_k in terms of $e_0, \ldots, e_n$.

corollary *Girard-Newton′*:
assumes *finite X* **and** $k >$ *card X*
shows $(\sum i{\le}card\ X.\ (-1)\ \hat{}\ i * sym\text{-}mpoly\ X\ i * powsum\text{-}mpoly\ X\ (k - i)) = (0 :: 'a :: comm\text{-}ring\text{-}1\ mpoly)$
proof −
have $(0 :: 'a\ mpoly) = (\sum i{<}k.\ (-\ 1)\ \hat{}\ i * sym\text{-}mpoly\ X\ i * powsum\text{-}mpoly\ X\ (k - i))$
using *Girard-Newton*[*of X k*] *assms* **by** *simp*
also have $\ldots = (\sum i{\le}card\ X.\ (-\ 1)\ \hat{}\ i * sym\text{-}mpoly\ X\ i * powsum\text{-}mpoly\ X\ (k - i))$
using *assms* **by** (*intro sum.mono-neutral-right*) *auto*
finally show *?thesis* **..**
qed

The following variant is the Newton–Girard Theorem solved for e_k, giving us an explicit way to determine e_k from $e_0, \ldots, e_{k-1}$ and $p_1, \ldots, p_k$:

corollary *sym-mpoly-recurrence*:
  assumes *k*: *k > 0* **and** *finite X*
  shows (*sym-mpoly X k :: 'a :: field-char-0 mpoly*) =
    −*smult* (*1 / of-nat k*) (∑ *i=1..k*. (−*1*) ^ *i* ∗ *sym-mpoly X* (*k* − *i*) ∗ *powsum-mpoly X i*)
proof −
  define *e p* :: *nat* ⇒ *'a mpoly* **where** [*simp*]: *e* = *sym-mpoly X p* = *powsum-mpoly X*
  have ∗: *0* = (−*1*) ^ *k* ∗ *of-nat k* ∗ *e k* +
      (∑ *i*<*k*. (− *1*) ^ *i* ∗ *e i* ∗ *p* (*k* − *i*) :: *'a mpoly*)
    using *Girard-Newton*[*of X k*] *assms* **by** *simp*

  have *0* = (−*1*) ^ *k* ∗ *smult* (*1 / of-nat k*) (*0* :: *'a mpoly*)
    by *simp*
  also have ... = *smult* (*1 / of-nat k*) (*of-nat k*) ∗ *e k* +
      smult (*1 / of-nat k*) (∑ *i*<*k*. (−*1*)^(*k*+*i*) ∗ *e i* ∗ *p* (*k* − *i*))
    unfolding *smult-conv-mult*
    using *k* **by** (*subst* ∗) (*simp add*: *power-add sum-distrib-left sum-distrib-right field-simps*
      del: *div-mult-self3 div-mult-self4 div-mult-self2 div-mult-self1*)
  also have *smult* (*1 / of-nat k* :: *'a*) (*of-nat k*) = *1*
    using *k* **by** (*simp add*: *of-nat-monom smult-conv-mult mult-monom del*: *monom-of-nat*)
  also have (∑ *i*<*k*. (−*1*) ^(*k*+*i*) ∗ *e i* ∗ *p* (*k* − *i*)) = (∑ *i=1..k*. (−*1*) ^ *i* ∗ *e* (*k*−*i*) ∗ *p i*)
    by (*intro sum.reindex-bij-witness*[*of* - λ*i*. *k* − *i* λ*i*. *k* − *i*])
      (*auto simp*: *minus-one-power-iff*)
  finally show *?thesis* **unfolding** *e-p-def* **by** *algebra*
qed

Analogously, the following is the theorem solved for p_k, giving us a way to determine p_k from $e_0, \ldots, e_k$ and $p_1, \ldots, p_{k-1}$:

corollary *powsum-mpoly-recurrence*:
  assumes *k*: *k > 0* **and** *X*: *finite X*
  shows (*powsum-mpoly X k :: 'a :: comm-ring-1 mpoly*) =
    (−*1*) ^(*k* + *1*) ∗ *of-nat k* ∗ *sym-mpoly X k* −
    (∑ *i=1..<k*. (−*1*) ^ *i* ∗ *sym-mpoly X i* ∗ *powsum-mpoly X* (*k* − *i*))
proof −
  define *e p* :: *nat* ⇒ *'a mpoly* **where** [*simp*]: *e* = *sym-mpoly X p* = *powsum-mpoly X*
  have ∗: *0* = (−*1*) ^ *k* ∗ *of-nat k* ∗ *e k* +
      (∑ *i*<*k*. (−*1*) ^ *i* ∗ *e i* ∗ *p* (*k* − *i*) :: *'a mpoly*)
    using *Girard-Newton*[*of X k*] *assms* **by** *simp*
  also have {..<*k*} = *insert 0* {*1..<k*}
    using *assms* **by** *auto*
  finally have (−*1*) ^ *k* ∗ *of-nat k* ∗ *e k* + (∑ *i=1..<k*. (−*1*) ^ *i* ∗ *e i* ∗ *p* (*k* − *i*)) + *p k* = *0*
    using *assms* **by** (*simp add*: *algebra-simps*)

from *add.inverse-unique*[*OF this*] **show** *?thesis* **by** *simp*
qed

Again, if we assume $k > n$, the above takes a much simpler form and is, in fact, a linear recurrence with constant coefficients:

lemma *powsum-mpoly-recurrence′*:
assumes *k*: *k* > *card X* **and** *X*: *finite X*
shows (*powsum-mpoly X k* :: *′a* :: *comm-ring-1 mpoly*) =
−(∑ *i*=*1*..*card X*. (−*1*) ^ *i* ∗ *sym-mpoly X i* ∗ *powsum-mpoly X* (*k* − *i*))
proof −
define *e p* :: *nat* ⇒ *′a mpoly* **where** [*simp*]: *e* = *sym-mpoly X p* = *powsum-mpoly X*
have *p k* = (−*1*) ^ (*k* + *1*) ∗ *of-nat k* ∗ *e k* − (∑ *i*=*1*..<*k*. (−*1*) ^ *i* ∗ *e i* ∗ *p* (*k* − *i*))
unfolding *e-p-def* **using** *assms* **by** (*intro powsum-mpoly-recurrence*) *auto*
also have ... = −(∑ *i*=*1*..<*k*. (−*1*) ^ *i* ∗ *e i* ∗ *p* (*k* − *i*))
using *assms* **by** *simp*
also have (∑ *i*=*1*..<*k*. (−*1*) ^ *i* ∗ *e i* ∗ *p* (*k* − *i*)) = (∑ *i*=*1*..*card X*. (−*1*) ^ *i* ∗ *e i* ∗ *p* (*k* − *i*))
using *assms* **by** (*intro sum.mono-neutral-right*) *auto*
finally show *?thesis* **by** *simp*
qed

end

3 Power sum puzzles

theory *Power-Sum-Puzzle*
imports
Power-Sum-Polynomials
Polynomial-Factorization.Rational-Root-Test
begin

3.1 General setting and results

We now consider the following situation: Given unknown complex numbers $x_1, \ldots, x_n$, define $p_k = x_1^k + \ldots + x_n^k$. Also, define $e_k := e_k(x_1, \ldots, x_n)$ where $e_k(X_1, \ldots, X_n)$ is the k-th elementary symmetric polynomial.

What is the relationship between the sequences e_k and p_k; in particular, how can we determine one from the other?

locale *power-sum-puzzle* =
fixes *x* :: *nat* ⇒ *complex*
fixes *n* :: *nat*
begin

We first introduce the notation $p_k := x_1^k + \ldots + x_n^k$:

definition *p* **where** *p k* = (∑ *i*<*n*. *x i* ^ *k*)

lemma *p-0* [*simp*]: *p 0* = *of-nat n*
  by (*simp add*: *p-def*)

lemma *p-altdef*: *p k* = *insertion x* (*powsum-mpoly* {..<*n*} *k*)
  by (*simp add*: *p-def*)

Similarly, we introduce the notation $e_k = e_k(x_1, \ldots, x_n)$ where $e_k(X_1, \ldots, X_n)$ is the k-th elementary symmetric polynomial (i. e. the sum of all monomials that can be formed by taking the product of exactly k distinct variables).

definition *e* **where** *e k* = (∑ *Y* | *Y* ⊆ {..<*n*} ∧ *card Y* = *k*. *prod x Y*)

lemma *e-altdef*: *e k* = *insertion x* (*sym-mpoly* {..<*n*} *k*)
  by (*simp add*: *e-def insertion-sym-mpoly*)

It is clear that e_k vanishes for $k > n$.

lemma *e-eq-0* [*simp*]: *k* > *n* ⟹ *e k* = *0*
  by (*simp add*: *e-altdef*)

lemma *e-0* [*simp*]: *e 0* = *1*
  by (*simp add*: *e-altdef*)

The recurrences we got from the Girard–Newton Theorem earlier now directly give us analogous recurrences for e_k and p_k:

lemma *e-recurrence*:
  assumes *k*: *k* > *0*
  shows *e k* = −(∑ *i*=*1*..*k*. (− *1*) ^ *i* ∗ *e* (*k* − *i*) ∗ *p i*) / *of-nat k*
  using *assms* **unfolding** *e-altdef p-altdef*
  by (*subst sym-mpoly-recurrence*)
    (*auto simp*: *insertion-sum insertion-add insertion-mult insertion-power insertion-sym-mpoly*)

lemma *p-recurrence*:
  assumes *k*: *k* > *0*
  shows *p k* = −*of-nat k* ∗ (−*1*) ^ *k* ∗ *e k* − (∑ *i*=*1*..<*k*. (−*1*) ^ *i* ∗ *e i* ∗ *p* (*k* − *i*))
  using *assms* **unfolding** *e-altdef p-altdef*
  by (*subst powsum-mpoly-recurrence*)
    (*auto simp*: *insertion-sum insertion-add insertion-mult insertion-diff*
      insertion-power insertion-sym-mpoly)

lemma *p-recurrence′*:
  assumes *k*: *k* > *n*
  shows *p k* = −(∑ *i*=*1*..*n*. (−*1*) ^ *i* ∗ *e i* ∗ *p* (*k* − *i*))
  using *assms* **unfolding** *e-altdef p-altdef*
  by (*subst powsum-mpoly-recurrence′*)
    (*auto simp*: *insertion-sum insertion-add insertion-mult insertion-diff*

```
                  insertion-power insertion-sym-mpoly)
```

It is clear from this recurrence that if p_1 to p_n are rational, then so are the e_k:

```
lemma e-in-Rats:
  assumes ⋀k. k ∈ {1..n} ⟹ p k ∈ ℚ
  shows   e k ∈ ℚ
proof (cases k ≤ n)
  case True
  thus ?thesis
  proof (induction k rule: less-induct)
    case (less k)
    show ?case
    proof (cases k = 0)
      case False
      thus ?thesis using assms less
        by (subst e-recurrence) (auto intro!: Rats-divide)
    qed auto
  qed
qed auto
```

Analogously, if p_1 to p_n are rational, then so are all the other p_k:

```
lemma p-in-Rats:
  assumes ⋀k. k ∈ {1..n} ⟹ p k ∈ ℚ
  shows   p k ∈ ℚ
proof (induction k rule: less-induct)
  case (less k)
  consider k = 0 | k ∈ {1..n} | k > n
    by force
  thus ?case
  proof cases
    assume k > n
    thus ?thesis
     using less assms by (subst p-recurrence'') (auto intro!: sum-in-Rats Rats-mult
e-in-Rats)
  qed (use assms in auto)
qed
```

Next, we define the unique monic polynomial that has $x_1, \ldots, x_n$ as its roots (respecting multiplicity):

```
definition Q :: complex poly where Q = (∏ i<n. [:−x i, 1:])

lemma degree-Q [simp]: Polynomial.degree Q = n
  by (simp add: Q-def degree-prod-eq-sum-degree)

lemma lead-coeff-Q [simp]: Polynomial.coeff Q n = 1
  using monic-prod[of {..<n} λi. [:−x i, 1:]]
  by (simp add: Q-def degree-prod-eq-sum-degree)
```

By Vieta's Theorem, we then have:

$$Q(X) = \sum_{k=0}^{n} (-1)^{n-k} e_{n-k} X^k$$

In other words: The above allows us to determine the $x_1, \ldots, x_n$ explicitly. They are, in fact, precisely the roots of the above polynomial (respecting multiplicity). Since this polynomial depends only on the e_k, which are in turn determined by $p_1, \ldots, p_n$, this means that these are the *only* solutions of this puzzle (up to permutation of the x_i).

lemma *coeff-Q*: *Polynomial.coeff Q k* = (*if k* > *n then 0 else* (−*1*) ^(*n* − *k*) ∗ *e* (*n* − *k*))
proof (*cases k* ≤ *n*)
  case *True*
  thus *?thesis*
    using *coeff-poly-from-roots*[*of* {..<*n*} *k x*] **by** (*auto simp*: *Q-def e-def*)
qed (*auto simp*: *Polynomial.coeff-eq-0*)

lemma *Q-altdef*: *Q* = (∑ *k*≤*n*. *Polynomial.monom* ((−*1*) ^(*n* − *k*) ∗ *e* (*n* − *k*)) *k*)
  by (*subst poly-as-sum-of-monoms* [*symmetric*]) (*simp add*: *coeff-Q*)

The following theorem again shows that $x_1, \ldots, x_n$ are precisely the roots of Q, respecting multiplicity.

theorem *mset-x-eq-poly-roots-Q*: {#*x i*. *i* ∈# *mset-set* {..<*n*}#} = *poly-roots Q*
proof −
  have *poly-roots Q* = (∑ *i*<*n*. {#*x i*#})
    by (*simp add*: *Q-def poly-roots-prod*)
  also have ... = {#*x i*. *i* ∈# *mset-set* {..<*n*}#}
    by (*induction n*) (*auto simp*: *lessThan-Suc*)
  finally show *?thesis* **..**
qed

end

3.2 Existence of solutions

So far, we have assumed a solution to the puzzle and then shown the properties that this solution must fulfil. However, we have not yet shown that there *is* a solution. We will do that now.

Let n be a natural number and f_k some sequence of complex numbers. We will show that there are $x_1, \ldots, x_n$ so that $x_1^k + \ldots + x_n^k = f_k$ for any $1 \leq k \leq n$.

locale *power-sum-puzzle-existence* =
  fixes *f* :: *nat* ⇒ *complex* **and** *n* :: *nat*
begin

First, we define a sequence of numbers e' analogously to the sequence e before, except that we replace all occurrences of the power sum p_k with f_k (recall that in the end we want $p_k = f_k$).

```
fun e' :: nat ⇒ complex
  where e' k = (if k = 0 then 1 else if k > n then 0
               else −(∑ i=1..k. (−1) ^ i ∗ e' (k − i) ∗ f i) / of-nat k)

lemmas [simp del] = e'.simps

lemma e'-0 [simp]: e' 0 = 1
  by (simp add: e'.simps)

lemma e'-eq-0 [simp]: k > n ⟹ e' k = 0
  by (auto simp: e'.simps)
```

Just as before, we can show the following recurrence for f in terms of e':

```
lemma f-recurrence:
  assumes k: k > 0 k ≤ n
  shows   f k = −of-nat k ∗ (−1) ^ k ∗ e' k − (∑ i=1..<k. (− 1) ^ i ∗ e' i ∗ f (k − i))
proof −
  have −of-nat k ∗ e' k = (∑ i=1..k. (− 1) ^ i ∗ e' (k − i) ∗ f i)
    using assms by (subst e'.simps) (simp add: field-simps)
  hence (−1)^k ∗ (−of-nat k ∗ e' k) = (−1)^k ∗ (∑ i=1..k. (− 1) ^ i ∗ e' (k − i) ∗ f i)
    by simp
  also have ... = f k + (−1) ^ k ∗ (∑ i=1..<k. (− 1) ^ i ∗ e' (k − i) ∗ f i)
    using assms by (subst sum.last-plus) (auto simp: minus-one-power-iff)
  also have (−1) ^ k ∗ (∑ i=1..<k. (− 1) ^ i ∗ e' (k − i) ∗ f i) =
            (∑ i=1..<k. (− 1) ^ (k − i) ∗ e' (k − i) ∗ f i)
   unfolding sum-distrib-left by (intro sum.cong) (auto simp: minus-one-power-iff)
  also have ... = (∑ i=1..<k. (− 1) ^ i ∗ e' i ∗ f (k − i))
    by (intro sum.reindex-bij-witness[of - λi. k − i λi. k − i]) auto
  finally show ?thesis
    by (simp add: algebra-simps)
qed
```

We now define a polynomial whose roots will be precisely the solution $x_1, \ldots, x_n$ to our problem.

```
lift-definition Q' :: complex poly is λk. if k > n then 0 else (−1) ^ (n − k) ∗ e' (n − k)
  using eventually-gt-at-top[of n] unfolding cofinite-eq-sequentially
  by eventually-elim auto

lemma coeff-Q': Polynomial.coeff Q' k = (if k > n then 0 else (−1) ^ (n − k) ∗ e' (n − k))
  by transfer auto
```

lemma *lead-coeff-Q′*: *Polynomial.coeff Q′ n = 1*
  by (*simp add*: *coeff-Q′*)

lemma *degree-Q′* [*simp*]: *Polynomial.degree Q′ = n*
proof (*rule antisym*)
  show *Polynomial.degree Q′ ≥ n*
    by (*rule le-degree*) (*auto simp*: *coeff-Q′*)
  show *Polynomial.degree Q′ ≤ n*
    by (*rule degree-le*) (*auto simp*: *coeff-Q′*)
qed

Since the complex numbers are algebraically closed, this polynomial splits into linear factors:

definition *Root* :: *nat ⇒ complex*
  where *Root* = (*SOME Root. Q′* = (∏ *i*<*n*. [:−*Root i*, *1*:]))

lemma *Root*: *Q′* = (∏ *i*<*n*. [:−*Root i*, *1*:])
proof −
  obtain *rs* **where** *rs*: (∏ *r*←*rs*. [:−*r*, *1*:]) = *Q′ length rs = n*
    using *fundamental-theorem-algebra-factorized*[*of Q′*] *lead-coeff-Q′* **by** *auto*
  have *Q′* = (∏ *r*←*rs*. [:−*r*, *1*:])
    by (*simp add*: *rs*)
  also have ... = (∏ *r*=*0*..<*n*. [:−*rs* ! *r*, *1*:])
    by (*subst prod-list-prod-nth*) (*auto simp*: *rs*)
  also have {*0*..<*n*} = {..<*n*}
    by *auto*
  finally have ∃ *Root*. *Q′* = (∏ *i*<*n*. [:−*Root i*, *1*:])
    by *blast*
  thus *?thesis*
    unfolding *Root-def* **by** (*rule someI-ex*)
qed

We can therefore now use the results from before for these $x_1, \ldots, x_n$.

sublocale *power-sum-puzzle Root n* **.**

Vieta's theorem gives us an expression for the coefficients of Q' in terms of $e_k(x_1, \ldots, x_n)$. This shows that our e' is indeed exactly the same as e.

lemma *e′-eq-e*: *e′ k = e k*
proof (*cases k ≤ n*)
  case *True*
  from *True* **have** *e′ k* = (−*1*) ^ *k* ∗ *poly.coeff Q′* (*n* − *k*)
    by (*simp add*: *coeff-Q′*)
  also have *Q′* = (∏ *x*<*n*. [:−*Root x*, *1*:])
    using *Root* **by** *simp*
  also have (−*1*) ^ *k* ∗ *poly.coeff* ... (*n* − *k*) = *e k*
    using *True coeff-poly-from-roots*[*of* {..<*n*} *n* − *k Root*]
    by (*simp add*: *insertion-sym-mpoly e-altdef*)
  finally show *e′ k = e k* **.**

qed *auto*

It then follows by a simple induction that $p_k = f_k$ for $1 \leq k \leq n$, as intended:

```
lemma p-eq-f:
  assumes k > 0 k ≤ n
  shows   p k = f k
  using assms
proof (induction k rule: less-induct)
  case (less k)
  thus p k = f k
    using p-recurrence[of k] f-recurrence[of k] less by (simp add: e′-eq-e)
qed

end
```

Here is a more condensed form of the above existence theorem:

```
theorem power-sum-puzzle-has-solution:
  fixes f :: nat ⇒ complex
  shows ∃Root. ∀k∈{1..n}. (∑i<n. Root i ^ k) = f k
proof −
  interpret power-sum-puzzle-existence f .
  from p-eq-f have ∀k∈{1..n}. (∑i<n. Root i ^ k) = f k
    by (auto simp: p-def)
  thus ?thesis by blast
qed
```

3.3 A specific puzzle

We now look at one particular instance of this puzzle, which was given as an exercise in *Abstract Algebra* by Dummit and Foote (Exercise 23 in Section 14.6) [1].

Suppose we know that $x + y + z = 1$, $x^2 + y^2 + z^2 = 2$, and $x^3 + y^3 + z^3 = 3$. Then what is $x^5 + y^5 + z^5$? What about any arbitrary $x^n + y^n + z^n$?

```
locale power-sum-puzzle-example =
  fixes x y z :: complex
  assumes xyz: x   + y   + z   = 1
               x^2 + y^2 + z^2 = 2
               x^3 + y^3 + z^3 = 3
begin
```

We reuse the results we have shown in the general case before.

```
definition f where f n = [x,y,z] ! n

sublocale power-sum-puzzle f 3 .
```

We can simplify p a bit more now.

```
lemma p-altdef′: p k = x ^ k + y ^ k + z ^ k
```

unfolding *p-def f-def* **by** (*simp add*: *eval-nat-numeral*)

lemma *p-base* [*simp*]: *p* (*Suc 0*) = *1 p 2* = *2 p 3* = *3*
using *xyz* **by** (*simp-all add*: *p-altdef′*)

We can easily compute all the non-zero values of e recursively:

lemma *e-Suc-0* [*simp*]: *e* (*Suc 0*) = *1*
by (*subst e-recurrence*; *simp*)

lemma *e-2* [*simp*]: *e 2* = −*1*/*2*
by (*subst e-recurrence*; *simp add*: *atLeastAtMost-nat-numeral*)

lemma *e-3* [*simp*]: *e 3* = *1*/*6*
by (*subst e-recurrence*; *simp add*: *atLeastAtMost-nat-numeral*)

Plugging in all the values, the recurrence relation for p now looks like this:

lemma *p-recurrence′′′*: *k* > *3* ⟹ *p k* = *p* (*k*−*3*) / *6* + *p* (*k*−*2*) / *2* + *p* (*k*−*1*)
using *p-recurrence′′*[*of k*] **by** (*simp add*: *atLeastAtMost-nat-numeral*)

Also note again that all p_k are rational:

lemma *p-in-Rats′*: *p k* ∈ ℚ
proof −
have ∗: {*1*..*3*} = {*1*, *2*, (*3*::*nat*)}
by *auto*
also have ∀ *k*∈. . . . *p k* ∈ ℚ
by *auto*
finally show *?thesis*
using *p-in-Rats*[*of k*] **by** *simp*
qed

The above recurrence has the characteristic polynomial $X^3 - X^2 - \frac{1}{2}X - \frac{1}{6}$ (which is exactly our Q), so we know that can now specify x, y, and z more precisely: They are the roots of that polynomial (in unspecified order).

lemma *xyz-eq*: {#*x*, *y*, *z*#} = *poly-roots* [:−*1*/*6*, −*1*/*2*, −*1*, *1*:]
proof −
have *image-mset f* (*mset-set* {..<*3*}) = *poly-roots Q*
using *mset-x-eq-poly-roots-Q* **.**
also have *image-mset f* (*mset-set* {..<*3*}) = {#*x*, *y*, *z*#}
by (*simp add*: *numeral-3-eq-3 lessThan-Suc f-def Multiset.union-ac*)
also have *Q* = [:−*1*/*6*, −*1*/*2*, −*1*, *1*:]
by (*simp add*: *Q-altdef atMost-nat-numeral Polynomial.monom-altdef power3-eq-cube power2-eq-square*)
finally show *?thesis* **.**
qed

Using the rational root test, we can easily show that x, y, and z are irrational.

lemma *xyz-irrational*: *set-mset* (*poly-roots* [:−*1*/*6*, −*1*/*2*, −*1*, *1*::*complex*:]) ∩ ℚ = {}

proof –
  define *p* :: *rat poly* **where** *p* = [:−*1/6*, −*1/2*, −*1*, *1*:]
  have *rational-root-test p* = *None*
    unfolding *p-def* **by** *code-simp*
  hence ¬(∃ *x*::*rat*. *poly p x* = *0*)
    by (*rule rational-root-test*)
  hence ¬(∃ *x*∈ℚ. *poly* (*map-poly of-rat p*) *x* = (*0* :: *complex*))
    by (*auto simp*: *Rats-def*)
  also have *map-poly of-rat p* = [:−*1/6*, −*1/2*, −*1*, *1* :: *complex*:]
    by (*simp add*: *p-def of-rat-minus of-rat-divide*)
  finally show *?thesis*
    by *auto*
qed

This polynomial is *squarefree*, so these three roots are, in fact, unique (so that there are indeed $3! = 6$ possible permutations).

lemma *rsquarefree*: *rsquarefree* [:−*1/6*, −*1/2*, −*1*, *1* :: *complex*:]
  by (*rule coprime-pderiv-imp-rsquarefree*)
      (*auto simp*: *pderiv-pCons coprime-iff-gcd-eq-1 gcd-poly-code gcd-poly-code-def content-def*
        primitive-part-def gcd-poly-code-aux-reduce pseudo-mod-def pseudo-divmod-def
          Let-def Polynomial.monom-altdef normalize-poly-def)

lemma *distinct-xyz*: *distinct* [*x*, *y*, *z*]
  by (*rule rsquarefree-imp-distinct-roots*[*OF rsquarefree*]) (*simp-all add*: *xyz-eq*)

While these roots *can* be written more explicitly in radical form, they are not very pleasant to look at. We therefore only compute a few values of p just for fun:

lemma *p 4* = *25* / *6* **and** *p 5* = *6* **and** *p 10* = *15539* / *432*
  by (*simp-all add*: *p-recurrence'''*)

Lastly, let us (informally) examine the asymptotics of this problem.

Two of the roots have a norm of roughly $\beta \approx 0.341$, while the remaining root α is roughly 1.431. Consequently, $x^n + y^n + z^n$ is asymptotically equivalent to α^n, with the error being bounded by $2 \cdot \beta^n$ and therefore goes to 0 very quickly.

For $p(10) = \frac{15539}{432} \approx 35.97$, for instance, this approximation is correct up to 6 decimals (a relative error of about 0.0001 %).

end

To really emphasise that the above puzzle has a solution and the locale is not 'vacuous', here is an interpretation of the locale using the existence theorem from before:

notepad
begin
  define *f* :: *nat* ⇒ *complex* **where** *f* = (λ*k*. [*1*,*2*,*3*] ! (*k* − *1*))

obtain *Root* :: *nat* ⇒ *complex* **where** *Root*: $\bigwedge k.\ k \in \{1..3\} \Longrightarrow (\sum i{<}3.\ Root\ i\ \widehat{}\ k) = f\ k$
using *power-sum-puzzle-has-solution*[*of 3 f*] **by** *metis*
define *x y z* **where** *x* = *Root 0* *y* = *Root 1* *z* = *Root 2*
have *x* + *y* + *z* = *1* **and** *x*^*2* + *y*^*2* + *z*^*2* = *2* **and** *x*^*3* + *y*^*3* + *z*^*3* = *3*
using *Root*[*of 1*] *Root*[*of 2*] *Root*[*of 3*] **by** (*simp-all add*: *eval-nat-numeral x-y-z-def f-def*)
then interpret *power-sum-puzzle-example x y z*
by *unfold-locales*
have *p 5* = *6*
by (*simp add*: *p-recurrence'''*)
end

end

References

[1] D. S. Dummit and R. M. Foote. *Abstract Algebra.* Wiley, 2003.

[2] D. Zeilberger. A combinatorial proof of Newton's identities. *Discrete Mathematics*, 49(3):319, 1984.