Symmetric Polynomials

Manuel Eberl

March 17, 2025

Abstract

A symmetric polynomial is a polynomial in variables $X_1, \ldots, X_n$ that does not discriminate between its variables, i. e. it is invariant under any permutation of them. These polynomials are important in the study of the relationship between the coefficients of a univariate polynomial and its roots in its algebraic closure.

This article provides a definition of symmetric polynomials and the elementary symmetric polynomials $e_1, \ldots, e_n$ and proofs of their basic properties, including three notable ones:

- Vieta's formula, which gives an explicit expression for the k-th coefficient of a univariate monic polynomial in terms of its roots $x_1, \ldots, x_n$, namely $c_k = (-1)^{n-k} e_{n-k}(x_1, \ldots, x_n)$.
- Second, the Fundamental Theorem of Symmetric Polynomials, which states that any symmetric polynomial is itself a uniquely determined polynomial combination of the elementary symmetric polynomials.
- Third, as a corollary of the previous two, that given a polynomial over some ring R, any symmetric polynomial combination of its roots is also in R even when the roots are not.

Both the symmetry property itself and the witness for the Fundamental Theorem are executable.

Contents

1 Vieta's Formulas

theory *Vieta*
imports
  HOL−Library.FuncSet
  HOL−Computational-Algebra.Computational-Algebra
begin

1.1 Auxiliary material

lemma *card-vimage-inter*:
  assumes *inj*: *inj-on f A* **and** *subset*: $X \subseteq f\ `\ A$
  shows $card\ (f\ -`\ X \cap A) = card\ X$
proof −
  have $card\ (f\ -`\ X \cap A) = card\ (f\ `\ (f\ -`\ X \cap A))$
    by (*subst card-image*) (*auto intro*!: *inj-on-subset*[*OF inj*])
  also have $f\ `\ (f\ -`\ X \cap A) = X$
    using *assms* **by** *auto*
  finally show *?thesis* **.**
qed

lemma *bij-betw-image-fixed-card-subset*:
  assumes *inj-on f A*
  shows $bij\text{-}betw\ (\lambda X.\ f\ `\ X)\ \{X.\ X \subseteq A \wedge card\ X = k\}\ \{X.\ X \subseteq f\ `\ A \wedge card\ X = k\}$
  using *assms inj-on-subset*[*OF assms*]
  by (*intro bij-betwI*[*of* - - - $\lambda X.\ f\ -`\ X \cap A$]) (*auto simp*: *card-image card-vimage-inter*)

lemma *image-image-fixed-card-subset*:
  assumes *inj-on f A*
  shows $(\lambda X.\ f\ `\ X)\ `\ \{X.\ X \subseteq A \wedge card\ X = k\} = \{X.\ X \subseteq f\ `\ A \wedge card\ X = k\}$
  using *bij-betw-imp-surj-on*[*OF bij-betw-image-fixed-card-subset*[*OF assms*, *of k*]]
.

lemma *prod-uminus*: $(\prod x \in A.\ -f\ x :: 'a :: comm\text{-}ring\text{-}1) = (-1)\ \hat{}\ card\ A * (\prod x \in A.\ f\ x)$
  by (*induction A rule*: *infinite-finite-induct*) (*auto simp*: *algebra-simps*)

theorem *prod-sum-PiE*:
  fixes $f :: 'a \Rightarrow 'b \Rightarrow 'c :: comm\text{-}semiring\text{-}1$
  assumes *finite*: *finite A* **and** *finite*: $\bigwedge x.\ x \in A \Longrightarrow finite\ (B\ x)$
  shows $(\prod x \in A.\ \sum y \in B\ x.\ f\ x\ y) = (\sum g \in PiE\ A\ B.\ \prod x \in A.\ f\ x\ (g\ x))$
  using *assms*
proof (*induction A rule*: *finite-induct*)
  case *empty*
  thus *?case* **by** *auto*
next
  case (*insert x A*)
  have $(\sum g \in Pi_E\ (insert\ x\ A)\ B.\ \prod x \in insert\ x\ A.\ f\ x\ (g\ x)) =$

$(\sum g{\in}Pi_E\ (insert\ x\ A)\ B.\ f\ x\ (g\ x) * (\prod x'{\in}A.\ f\ x'\ (g\ x')))$
using *insert* **by** *simp*
also have $(\lambda g.\ \prod x'{\in}A.\ f\ x'\ (g\ x')) = (\lambda g.\ \prod x'{\in}A.\ f\ x'\ (if\ x' = x\ then\ undefined\ else\ g\ x'))$
using *insert* **by** (*intro ext prod.cong*) *auto*
also have $(\sum g{\in}Pi_E\ (insert\ x\ A)\ B.\ f\ x\ (g\ x) * \ldots\ g) =$
$(\sum (y,g){\in}B\ x \times Pi_E\ A\ B.\ f\ x\ y * (\prod x'{\in}A.\ f\ x'\ (g\ x')))$
using *insert.prems insert.hyps*
by (*intro sum.reindex-bij-witness*[*of* - $\lambda(y,g).\ g(x := y)\ \lambda g.\ (g\ x,\ g(x := undefined))$]])
(*auto simp*: *PiE-def extensional-def*)
also have $\ldots = (\sum y{\in}B\ x.\ \sum g{\in}Pi_E\ A\ B.\ f\ x\ y * (\prod x'{\in}A.\ f\ x'\ (g\ x')))$
by (*subst sum.cartesian-product*) *auto*
also have $\ldots = (\sum y{\in}B\ x.\ f\ x\ y) * (\sum g{\in}Pi_E\ A\ B.\ \prod x'{\in}A.\ f\ x'\ (g\ x'))$
using *insert* **by** (*subst sum.swap*) (*simp add*: *sum-distrib-left sum-distrib-right*)
also have $(\sum g{\in}Pi_E\ A\ B.\ \prod x'{\in}A.\ f\ x'\ (g\ x')) = (\prod x{\in}A.\ \sum y{\in}B\ x.\ f\ x\ y)$
using *insert.prems* **by** (*intro insert.IH* [*symmetric*]) *auto*
also have $(\sum y{\in}B\ x.\ f\ x\ y) * \ldots = (\prod x{\in}insert\ x\ A.\ \sum y{\in}B\ x.\ f\ x\ y)$
using *insert.hyps* **by** *simp*
finally show *?case* **..**
qed

corollary *prod-add*:
fixes $f1\ f2 :: 'a \Rightarrow 'c :: comm\text{-}semiring\text{-}1$
assumes *finite*: *finite A*
shows $(\prod x{\in}A.\ f1\ x + f2\ x) = (\sum X{\in}Pow\ A.\ (\prod x{\in}X.\ f1\ x) * (\prod x{\in}A{-}X.\ f2\ x))$
proof −
have $(\prod x{\in}A.\ f1\ x + f2\ x) = (\sum g{\in}A \to_E UNIV.\ \prod x{\in}A.\ if\ g\ x\ then\ f1\ x\ else\ f2\ x)$
using *prod-sum-PiE*[*of A* λ-. *UNIV* :: *bool set* $\lambda x\ y.\ if\ y\ then\ f1\ x\ else\ f2\ x$] *assms*
by (*simp-all add*: *UNIV-bool add-ac*)
also have $\ldots = (\sum X{\in}Pow\ A.\ \prod x{\in}A.\ if\ x \in X\ then\ f1\ x\ else\ f2\ x)$
by (*intro sum.reindex-bij-witness*
[*of* - $\lambda X\ x.\ if\ x \in A\ then\ x \in X\ else\ undefined\ \lambda P.\ \{x{\in}A.\ P\ x\}$]) *auto*
also have $\ldots = (\sum X{\in}Pow\ A.\ (\prod x{\in}X.\ f1\ x) * (\prod x{\in}A{-}X.\ f2\ x))$
proof (*intro sum.cong refl*, *goal-cases*)
case (*1 X*)
let $?f = \lambda x.\ if\ x \in X\ then\ f1\ x\ else\ f2\ x$
have $prod\ f1\ X * prod\ f2\ (A - X) = prod\ ?f\ X * prod\ ?f\ (A - X)$
by (*intro arg-cong2*[*of* - - - - (*)] *prod.cong*) *auto*
also have $\ldots = prod\ ?f\ (X \cup (A - X))$
using *1* **by** (*subst prod.union-disjoint*) (*auto intro*: *finite-subset*[*OF* - *finite*])
also have $X \cup (A - X) = A$ **using** *1* **by** *auto*
finally show *?case* **..**
qed
finally show *?thesis* **.**
qed

corollary *prod-diff1*:
fixes *f1 f2* :: *'a* ⇒ *'c* :: *comm-ring-1*
assumes *finite*: *finite A*
shows $(\prod x{\in}A.\ f1\ x - f2\ x) = (\sum X{\in}Pow\ A.\ (-1)\ \widehat{}\ card\ X * (\prod x{\in}X.\ f2\ x) * (\prod x{\in}A{-}X.\ f1\ x))$
proof −
have $(\prod x{\in}A.\ f1\ x - f2\ x) = (\prod x{\in}A.\ -f2\ x + f1\ x)$
by *simp*
also have $\ldots = (\sum X{\in}Pow\ A.\ (\prod x{\in}X.\ - f2\ x) * prod\ f1\ (A - X))$
by (*rule prod-add*) *fact+*
also have $\ldots = (\sum X{\in}Pow\ A.\ (-1)\ \widehat{}\ card\ X * (\prod x{\in}X.\ f2\ x) * prod\ f1\ (A - X))$
by (*simp add*: *prod-uminus*)
finally show *?thesis* **.**
qed

corollary *prod-diff2*:
fixes *f1 f2* :: *'a* ⇒ *'c* :: *comm-ring-1*
assumes *finite*: *finite A*
shows $(\prod x{\in}A.\ f1\ x - f2\ x) = (\sum X{\in}Pow\ A.\ (-1)\ \widehat{}\ (card\ A - card\ X) * (\prod x{\in}X.\ f1\ x) * (\prod x{\in}A{-}X.\ f2\ x))$
proof −
have $(\prod x{\in}A.\ f1\ x - f2\ x) = (\prod x{\in}A.\ f1\ x + (-f2\ x))$
by *simp*
also have $\ldots = (\sum X{\in}Pow\ A.\ (\prod x{\in}X.\ f1\ x) * (\prod x{\in}A{-}X.\ -f2\ x))$
by (*rule prod-add*) *fact+*
also have $\ldots = (\sum X{\in}Pow\ A.\ (-1)\ \widehat{}\ card\ (A - X) * (\prod x{\in}X.\ f1\ x) * (\prod x{\in}A{-}X.\ f2\ x))$
by (*simp add*: *prod-uminus mult-ac*)
also have $\ldots = (\sum X{\in}Pow\ A.\ (-1)\ \widehat{}\ (card\ A - card\ X) * (\prod x{\in}X.\ f1\ x) * (\prod x{\in}A{-}X.\ f2\ x))$
using *finite-subset*[*OF - assms*] **by** (*intro sum.cong refl*, *subst card-Diff-subset*) *auto*
finally show *?thesis* **.**
qed

1.2 Main proofs

Our goal is to determine the coefficients of some fully factored polynomial $p(X) = c(X - x_1) \ldots (X - x_n)$ in terms of the x_i. It is clear that it is sufficient to consider monic polynomials (i.e. $c = 1$), since the general case follows easily from this one.

We start off by expanding the product over the linear factors:

lemma *poly-from-roots*:
fixes *f* :: *'a* ⇒ *'b* :: *comm-ring-1* **assumes** *fin*: *finite A*
shows $(\prod x{\in}A.\ [:-f\ x,\ 1:]) = (\sum X{\in}Pow\ A.\ monom\ ((-1)\ \widehat{}\ card\ X * (\prod x{\in}X.\ f\ x))\ (card\ (A - X)))$

proof $-$
  have $(\prod x{\in}A.\ [{:}-f\ x,\ 1{:}]) = (\prod x{\in}A.\ [{:}0,\ 1{:}] - [{:}f\ x{:}])$
    by *simp*
  also have $\ldots = (\sum X{\in}Pow\ A.\ (-1)\ \hat{}\ card\ X * (\prod x{\in}X.\ [{:}f\ x{:}]) * monom\ 1\ (card\ (A - X)))$
    using *fin* **by** (*subst prod-diff1*) (*auto simp*: *monom-altdef mult-ac*)
  also have $\ldots = (\sum X{\in}Pow\ A.\ monom\ ((-1)\ \hat{}\ card\ X * (\prod x{\in}X.\ f\ x))\ (card\ (A - X)))$
  proof (*intro sum.cong refl*, *goal-cases*)
    case (*1 X*)
    have $(-1 :: 'b\ poly)\ \hat{}\ card\ X = [{:}(-1)\ \hat{}\ card\ X{:}]$
      by (*induction X rule*: *infinite-finite-induct*) (*auto simp*: *one-pCons algebra-simps*)
    moreover have $(\prod x{\in}X.\ [{:}f\ x{:}]) = [{:}\prod x{\in}X.\ f\ x{:}]$
      by (*induction X rule*: *infinite-finite-induct*) *auto*
    ultimately show *?case* **by** (*simp add*: *smult-monom*)
  qed
  finally show *?thesis* **.**
qed

Comparing coefficients yields Vieta's formula:

theorem *coeff-poly-from-roots*:
  fixes $f :: 'a \Rightarrow 'b :: comm\text{-}ring\text{-}1$
  assumes *fin*: *finite A* **and** *k*: $k \le card\ A$
  shows $coeff\ (\prod x{\in}A.\ [{:}-f\ x,\ 1{:}])\ k =$
    $(-1)\ \hat{}(card\ A - k) * (\sum X \mid X \subseteq A \wedge card\ X = card\ A - k.\ (\prod x{\in}X.\ f\ x))$
proof $-$
  have $(\prod x{\in}A.\ [{:}-f\ x,\ 1{:}]) = (\sum X{\in}Pow\ A.\ monom\ ((-1)\ \hat{}\ card\ X * (\prod x{\in}X.\ f\ x))\ (card\ (A - X)))$
    by (*intro poly-from-roots fin*)
  also have $coeff\ \ldots\ k = (\sum X \mid X \subseteq A \wedge card\ X = card\ A - k.\ (-1)\ \hat{}(card\ A - k) * (\prod x{\in}X.\ f\ x))$
    unfolding *coeff-sum coeff-monom* **using** *finite-subset*[*OF - fin*] *k card-mono*[*OF fin*]
    by (*intro sum.mono-neutral-cong-right*) (*auto simp*: *card-Diff-subset*)
  also have $\ldots = (-1)\ \hat{}(card\ A - k) * (\sum X \mid X \subseteq A \wedge card\ X = card\ A - k.\ (\prod x{\in}X.\ f\ x))$
    by (*simp add*: *sum-distrib-left*)
  finally show *?thesis* **.**
qed

If the roots are all distinct, we can get the following alternative representation:

corollary *coeff-poly-from-roots'*:
  fixes $f :: 'a \Rightarrow 'b :: comm\text{-}ring\text{-}1$
  assumes *fin*: *finite A* **and** *inj*: *inj-on f A* **and** *k*: $k \le card\ A$
  shows $coeff\ (\prod x{\in}A.\ [{:}-f\ x,\ 1{:}])\ k =$
    $(-1)\ \hat{}(card\ A - k) * (\sum X \mid X \subseteq f\ `\ A \wedge card\ X = card\ A - k.\ \prod X)$

proof −
  have $coeff\ (\prod x{\in}A.\ [{:}{-}f\ x,\ 1{:}])\ k =$
  $(-1)\ \hat{}\ (card\ A - k) * (\sum X \mid X \subseteq A \wedge card\ X = card\ A - k.\ (\prod x{\in}X.\ f\ x))$
  by $(intro\ coeff\text{-}poly\text{-}from\text{-}roots\ assms)$
  also have $(\sum X \mid X \subseteq A \wedge card\ X = card\ A - k.\ (\prod x{\in}X.\ f\ x)) =$
  $(\sum X \mid X \subseteq A \wedge card\ X = card\ A - k.\ \prod (f`X))$
  by $(intro\ sum.cong\ refl,\ subst\ prod.reindex)\ (auto\ intro{:}\ inj\text{-}on\text{-}subset[OF\ inj])$
  also have $\ldots = (\sum X \in (\lambda X.\ f`X)\ `\ \{X.\ X \subseteq A \wedge card\ X = card\ A - k\}.\ \prod X)$
  by $(subst\ sum.reindex)\ (auto\ intro!{:}\ inj\text{-}on\text{-}image\ inj\text{-}on\text{-}subset[OF\ inj])$
  also have $(\lambda X.\ f\ `\ X)\ `\ \{X.\ X \subseteq A \wedge card\ X = card\ A - k\} = \{X.\ X \subseteq f\ `\ A \wedge card\ X = card\ A - k\}$
  by $(intro\ image\text{-}image\text{-}fixed\text{-}card\text{-}subset\ inj)$
  finally show *?thesis* **.**
qed

end

2 Symmetric Polynomials

theory *Symmetric-Polynomials*
imports
  Vieta
  Polynomials.More-MPoly-Type
  HOL−Combinatorics.Permutations
begin

2.1 Auxiliary facts

An infinite set has infinitely many infinite subsets.

lemma *infinite-infinite-subsets*:
  assumes *infinite A*
  shows $infinite\ \{X.\ X \subseteq A \wedge infinite\ X\}$
proof −
  have $\forall k.\ \exists X.\ X \subseteq A \wedge infinite\ X \wedge card\ (A - X) = k$ **for** $k :: nat$
  proof
    fix $k :: nat$ **obtain** Y **where** $finite\ Y\ card\ Y = k\ Y \subseteq A$
      using $infinite\text{-}arbitrarily\text{-}large[of\ A\ k]$ *assms* **by** *auto*
    moreover from *this* **have** $A - (A - Y) = Y$ **by** *auto*
    ultimately show $\exists X.\ X \subseteq A \wedge infinite\ X \wedge card\ (A - X) = k$
      using *assms* **by** $(intro\ exI[of\ \text{-}\ A - Y])$ *auto*
  qed
  from $choice[OF\ this]$ **obtain** f
    where $f{:}\ \bigwedge k.\ f\ k \subseteq A \wedge infinite\ (f\ k) \wedge card\ (A - f\ k) = k$ **by** *blast*
  have $k = l$ **if** $f\ k = f\ l$ **for** $k\ l$
  proof $(rule\ ccontr)$
    assume $k \neq l$
    hence $card\ (A - f\ k) \neq card\ (A - f\ l)$

```
    using f[of k] f[of l] by auto
  with ‹f k = f l› show False by simp
qed
hence inj f by (auto intro: injI)
moreover have range f ⊆ {X. X ⊆ A ∧ infinite X}
  using f by auto
ultimately show ?thesis
  by (subst infinite-iff-countable-subset) auto
qed
```

An infinite set contains infinitely many finite subsets of any fixed nonzero cardinality.

```
lemma infinite-card-subsets:
  assumes infinite A k > 0
  shows   infinite {X. X ⊆ A ∧ finite X ∧ card X = k}
proof −
  obtain B where B: B ⊆ A finite B card B = k − 1
    using infinite-arbitrarily-large[OF assms(1), of k − 1] by blast
  define f where f = (λx. insert x B)
  have f ‘ (A − B) ⊆ {X. X ⊆ A ∧ finite X ∧ card X = k}
    using assms B by (auto simp: f-def)
  moreover have inj-on f (A − B)
    by (auto intro!: inj-onI simp: f-def)
  hence infinite (f ‘ (A − B))
    using assms B by (subst finite-image-iff) auto
  ultimately show ?thesis
    by (rule infinite-super)
qed

lemma comp-bij-eq-iff:
  assumes bij f
  shows   g ∘ f = h ∘ f ⟷ g = h
proof
  assume ∗: g ∘ f = h ∘ f
  show g = h
  proof
    fix x
    obtain y where [simp]: x = f y using bij-is-surj[OF assms] by auto
    have (g ∘ f) y = (h ∘ f) y by (simp only: ∗)
    thus g x = h x by simp
  qed
qed auto

lemma sum-list-replicate [simp]:
  sum-list (replicate n x) = of-nat n ∗ (x :: 'a :: semiring-1)
  by (induction n) (auto simp: algebra-simps)

lemma ex-subset-of-card:
  assumes finite A card A ≥ k
```

```
  shows  ∃B. B ⊆ A ∧ card B = k
  using assms
proof (induction arbitrary: k rule: finite-induct)
  case empty
  thus ?case by auto
next
  case (insert x A k)
  show ?case
  proof (cases k = 0)
    case True
    thus ?thesis by (intro exI[of - {}]) auto
  next
    case False
    from insert have ∃B⊆A. card B = k − 1 by (intro insert.IH) auto
    then obtain B where B: B ⊆ A card B = k − 1 by auto
    with insert have [simp]: x ∉ B by auto
    have insert x B ⊆ insert x A
      using B insert by auto
    moreover have card (insert x B) = k
      using insert B finite-subset[of B A] False by (subst card.insert-remove) auto
    ultimately show ?thesis by blast
  qed
qed

lemma length-sorted-list-of-set [simp]: length (sorted-list-of-set A) = card A
  using distinct-card[of sorted-list-of-set A] by (cases finite A) simp-all

lemma upt-add-eq-append′: i ≤ j ⟹ j ≤ k ⟹ [i..<k] = [i..<j] @ [j..<k]
  using upt-add-eq-append[of i j k − j] by simp
```

2.2 Subrings and ring homomorphisms

```
locale ring-closed =
  fixes A :: ′a :: comm-ring-1 set
  assumes zero-closed [simp]: 0 ∈ A
  assumes one-closed [simp]: 1 ∈ A
  assumes add-closed [simp]: x ∈ A ⟹ y ∈ A ⟹ (x + y) ∈ A
  assumes mult-closed [simp]: x ∈ A ⟹ y ∈ A ⟹ (x ∗ y) ∈ A
  assumes uminus-closed [simp]: x ∈ A ⟹ −x ∈ A
begin

lemma minus-closed [simp]: x ∈ A ⟹ y ∈ A ⟹ x − y ∈ A
  using add-closed[of x −y] uminus-closed[of y] by auto

lemma sum-closed [intro]: (⋀x. x ∈ X ⟹ f x ∈ A) ⟹ sum f X ∈ A
  by (induction X rule: infinite-finite-induct) auto

lemma power-closed [intro]: x ∈ A ⟹ x ^ n ∈ A
  by (induction n) auto
```

lemma *Sum-any-closed* [*intro*]: $(\bigwedge x.\ f\ x \in A) \Longrightarrow$ *Sum-any* $f \in A$
unfolding *Sum-any.expand-set* **by** (*rule sum-closed*)

lemma *prod-closed* [*intro*]: $(\bigwedge x.\ x \in X \Longrightarrow f\ x \in A) \Longrightarrow$ *prod* $f\ X \in A$
by (*induction X rule*: *infinite-finite-induct*) *auto*

lemma *Prod-any-closed* [*intro*]: $(\bigwedge x.\ f\ x \in A) \Longrightarrow$ *Prod-any* $f \in A$
unfolding *Prod-any.expand-set* **by** (*rule prod-closed*)

lemma *prod-fun-closed* [*intro*]: $(\bigwedge x.\ f\ x \in A) \Longrightarrow (\bigwedge x.\ g\ x \in A) \Longrightarrow$ *prod-fun* $f\ g$ $x \in A$
by (*auto simp*: *prod-fun-def when-def intro*!: *Sum-any-closed mult-closed*)

lemma *of-nat-closed* [*simp*, *intro*]: *of-nat* $n \in A$
by (*induction n*) *auto*

lemma *of-int-closed* [*simp*, *intro*]: *of-int* $n \in A$
by (*induction n*) *auto*

end

locale *ring-homomorphism* =
fixes f :: $'a$ :: *comm-ring-1* $\Rightarrow$ $'b$ :: *comm-ring-1*
assumes *add*[*simp*]: $f\ (x + y) = f\ x + f\ y$
assumes *uminus*[*simp*]: $f\ (-x) = -f\ x$
assumes *mult*[*simp*]: $f\ (x * y) = f\ x * f\ y$
assumes *zero*[*simp*]: $f\ 0 = 0$
assumes *one* [*simp*]: $f\ 1 = 1$
begin

lemma *diff* [*simp*]: $f\ (x - y) = f\ x - f\ y$
using *add*[*of x* $-y$] **by** (*simp del*: *add*)

lemma *power* [*simp*]: $f\ (x\ \widehat{}\ n) = f\ x\ \widehat{}\ n$
by (*induction n*) *auto*

lemma *sum* [*simp*]: $f\ (sum\ g\ A) = (\sum x{\in}A.\ f\ (g\ x))$
by (*induction A rule*: *infinite-finite-induct*) *auto*

lemma *prod* [*simp*]: $f\ (prod\ g\ A) = (\prod x{\in}A.\ f\ (g\ x))$
by (*induction A rule*: *infinite-finite-induct*) *auto*

end

lemma *ring-homomorphism-id* [*intro*]: *ring-homomorphism id*
by *standard auto*

lemma *ring-homomorphism-id′* [*intro*]: *ring-homomorphism* $(\lambda x.\ x)$

by *standard auto*

lemma *ring-homomorphism-of-int* [*intro*]: *ring-homomorphism of-int*
by *standard auto*

2.3 Various facts about multivariate polynomials

lemma *poly-mapping-nat-ge-0* [*simp*]: $(m :: nat \Rightarrow_0 nat) \geq 0$
proof (*cases* $m = 0$)
case *False*
hence *Poly-Mapping.lookup* $m \neq$ *Poly-Mapping.lookup* 0 **by** *transfer auto*
hence $\exists k$. *Poly-Mapping.lookup* $m\ k \neq 0$ **by** (*auto simp*: *fun-eq-iff*)
from *LeastI-ex*[*OF this*] *Least-le*[*of* λk. *Poly-Mapping.lookup* $m\ k \neq 0$] **show** *?thesis*
by (*force simp*: *less-eq-poly-mapping-def less-fun-def*)
qed *auto*

lemma *poly-mapping-nat-le-0* [*simp*]: $(m :: nat \Rightarrow_0 nat) \leq 0 \longleftrightarrow m = 0$
unfolding *less-eq-poly-mapping-def poly-mapping-eq-iff less-fun-def* **by** *auto*

lemma *of-nat-diff-poly-mapping-nat*:
assumes $m \geq n$
shows *of-nat* $(m - n) =$ (*of-nat* $m -$ *of-nat* n :: $'a$ :: *monoid-add* $\Rightarrow_0$ *nat*)
by (*auto intro*!: *poly-mapping-eqI simp*: *lookup-of-nat lookup-minus when-def*)

lemma *mpoly-coeff-transfer* [*transfer-rule*]:
rel-fun cr-mpoly (=) *poly-mapping.lookup MPoly-Type.coeff*
unfolding *MPoly-Type.coeff-def* **by** *transfer-prover*

lemma *mapping-of-sum*: ($\sum x \in A$. *mapping-of* $(f\ x)$) = *mapping-of* (*sum* $f\ A$)
by (*induction A rule*: *infinite-finite-induct*) (*auto simp*: *plus-mpoly.rep-eq zero-mpoly.rep-eq*)

lemma *mapping-of-eq-0-iff* [*simp*]: *mapping-of* $p = 0 \longleftrightarrow p = 0$
by *transfer auto*

lemma *Sum-any-mapping-of*: *Sum-any* (λx. *mapping-of* $(f\ x)$) = *mapping-of* (*Sum-any* f)
by (*simp add*: *Sum-any.expand-set mapping-of-sum*)

lemma *Sum-any-parametric-cr-mpoly* [*transfer-rule*]:
(*rel-fun* (*rel-fun* (=) *cr-mpoly*) *cr-mpoly*) *Sum-any Sum-any*
by (*auto simp*: *rel-fun-def cr-mpoly-def Sum-any-mapping-of*)

lemma *lookup-mult-of-nat* [*simp*]: *lookup* (*of-nat* $n * m$) $k = n *$ *lookup* $m\ k$
proof −
have *of-nat* $n * m = (\sum i<n.\ m)$ **by** *simp*
also have *lookup* ... $k = (\sum i<n.$ *lookup* $m\ k)$
by (*simp only*: *lookup-sum*)
also have ... $= n *$ *lookup* $m\ k$

by *simp*
finally show *?thesis* **.**
qed

lemma *mpoly-eqI*:
assumes $\bigwedge mon.$ *MPoly-Type.coeff p mon* = *MPoly-Type.coeff q mon*
shows $p = q$
using *assms* **by** (*transfer*, *transfer*) (*auto simp*: *fun-eq-iff*)

lemma *coeff-mpoly-times*:
MPoly-Type.coeff $(p * q)$ *mon* = *prod-fun* (*MPoly-Type.coeff p*) (*MPoly-Type.coeff q*) *mon*
by (*transfer'*, *transfer'*) *auto*

lemma (**in** *ring-closed*) *coeff-mult-closed* [*intro*]:
($\bigwedge x.$ *coeff p x* $\in A$) $\Longrightarrow$ ($\bigwedge x.$ *coeff q x* $\in A$) $\Longrightarrow$ *coeff* $(p * q)$ $x \in A$
by (*auto simp*: *coeff-mpoly-times prod-fun-closed*)

lemma *coeff-notin-vars*:
assumes $\neg$(*keys m* $\subseteq$ *vars p*)
shows *coeff p m* = *0*
using *assms* **unfolding** *vars-def* **by** *transfer'* (*auto simp*: *in-keys-iff*)

lemma *finite-coeff-support* [*intro*]: *finite* {*m. coeff p m* $\neq$ *0*}
by *transfer simp*

lemma *insertion-altdef*:
insertion f p = *Sum-any* ($\lambda m.$ *coeff p m* $*$ *Prod-any* ($\lambda i.$ *f i* ^ *lookup m i*))
by (*transfer'*, *transfer'*) (*simp add*: *insertion-fun-def*)

lemma *mpoly-coeff-uminus* [*simp*]: *coeff* $(-p)$ *m* = $-$*coeff p m*
by *transfer auto*

lemma *Sum-any-uminus*: *Sum-any* ($\lambda x. -f\ x$ :: *'a* :: *ab-group-add*) = $-$*Sum-any f*
by (*simp add*: *Sum-any.expand-set sum-negf*)

lemma *insertion-uminus* [*simp*]: *insertion f* $(-p$ :: *'a* :: *comm-ring-1 mpoly*) = $-$*insertion f p*
by (*simp add*: *insertion-altdef Sum-any-uminus*)

lemma *Sum-any-lookup*: *finite* {*x. g x* $\neq$ *0*} $\Longrightarrow$ *Sum-any* ($\lambda x.$ *lookup* (*g x*) *y*) = *lookup* (*Sum-any g*) *y*
by (*auto simp*: *Sum-any.expand-set lookup-sum intro*!: *sum.mono-neutral-left*)

lemma *Sum-any-diff*:
assumes *finite* {*x. f x* $\neq$ *0*}
assumes *finite* {*x. g x* $\neq$ *0*}
shows *Sum-any* ($\lambda x.$ *f x* $-$ *g x* :: *'a* :: *ab-group-add*) = *Sum-any f* $-$ *Sum-any g*

proof $-$
  have $\{x.\ f\ x - g\ x \neq 0\} \subseteq \{x.\ f\ x \neq 0\} \cup \{x.\ g\ x \neq 0\}$ **by** *auto*
  moreover have *finite* $(\{x.\ f\ x \neq 0\} \cup \{x.\ g\ x \neq 0\})$
    by (*subst finite-Un*) (*insert assms*, *auto*)
  ultimately have *finite* $\{x.\ f\ x - g\ x \neq 0\}$
    by (*rule finite-subset*)
  with *assms* **show** *?thesis*
    by (*simp add*: *algebra-simps Sum-any.distrib* [*symmetric*])
qed

lemma *insertion-diff*:
  insertion f $(p - q :: {'a} :: \mathit{comm\text{-}ring\text{-}1\ mpoly}) = \mathit{insertion}\ f\ p - \mathit{insertion}\ f\ q$
proof (*transfer*, *transfer*)
  fix $f :: nat \Rightarrow {'a}$ **and** $p\ q :: (nat \Rightarrow_0 nat) \Rightarrow {'a}$
  assume *fin*: *finite* $\{x.\ p\ x \neq 0\}$ *finite* $\{x.\ q\ x \neq 0\}$
  have *insertion-fun* f $(\lambda x.\ p\ x - q\ x) =$
        $(\sum m.\ p\ m * (\prod v.\ f\ v\ \widehat{}\ \mathit{lookup}\ m\ v) - q\ m * (\prod v.\ f\ v\ \widehat{}\ \mathit{lookup}\ m\ v))$
    by (*simp add*: *insertion-fun-def algebra-simps Sum-any-diff*)
  also have $\ldots = (\sum m.\ p\ m * (\prod v.\ f\ v\ \widehat{}\ \mathit{lookup}\ m\ v)) - (\sum m.\ q\ m * (\prod v.\ f\ v$
$\widehat{}\ \mathit{lookup}\ m\ v))$
    by (*subst Sum-any-diff*) (*auto intro*: *finite-subset*[*OF* - *fin*(*1*)] *finite-subset*[*OF*
- *fin*(*2*)])
  also have $\ldots =$ *insertion-fun* f p $-$ *insertion-fun* f q
    by (*simp add*: *insertion-fun-def*)
  finally show *insertion-fun* f $(\lambda x.\ p\ x - q\ x) = \ldots$ **.**
qed

lemma *insertion-power*: *insertion* f $(p\ \widehat{}\ n) =$ *insertion* f $p\ \widehat{}\ n$
  by (*induction* n) (*simp-all add*: *insertion-mult*)

lemma *insertion-sum*: *insertion* f $(\mathit{sum}\ g\ A) = (\sum x{\in}A.\ \mathit{insertion}\ f\ (g\ x))$
  by (*induction A rule*: *infinite-finite-induct*) (*auto simp*: *insertion-add*)

lemma *insertion-prod*: *insertion* f $(\mathit{prod}\ g\ A) = (\prod x{\in}A.\ \mathit{insertion}\ f\ (g\ x))$
  by (*induction A rule*: *infinite-finite-induct*) (*auto simp*: *insertion-mult*)

lemma *coeff-Var*: *coeff* $(\mathit{Var}\ i)\ m = (\mathit{1}\ \mathit{when}\ m = \mathit{Poly\text{-}Mapping.single}\ i\ \mathit{1})$
  by *transfer'* (*auto simp*: Var_0*-def lookup-single when-def*)

lemma *vars-Var*: *vars* $(\mathit{Var}\ i :: {'a} :: \{one,zero\}\ mpoly) = (\mathit{if}\ (0{::}{'a}) = 1\ \mathit{then}\ \{\}$
$\mathit{else}\ \{i\})$
  unfolding *vars-def* **by** (*auto simp*: *Var.rep-eq* Var_0*-def*)

lemma *insertion-Var* [*simp*]: *insertion* f $(\mathit{Var}\ i) = f\ i$
proof $-$
  have *insertion* f $(\mathit{Var}\ i) = (\sum m.\ (\mathit{1}\ \mathit{when}\ m = \mathit{Poly\text{-}Mapping.single}\ i\ \mathit{1}) *$
                    $(\prod i.\ f\ i\ \widehat{}\ \mathit{lookup}\ m\ i))$
    by (*simp add*: *insertion-altdef coeff-Var*)
  also have $\ldots = (\prod j.\ f\ j\ \widehat{}\ \mathit{lookup}\ (\mathit{Poly\text{-}Mapping.single}\ i\ \mathit{1})\ j)$

by (*subst Sum-any.expand-superset*[*of* {*Poly-Mapping.single i 1*}]) (*auto simp*: *when-def*)
also have ... = *f i*
by (*subst Prod-any.expand-superset*[*of* {*i*}]) (*auto simp*: *when-def lookup-single*)
finally show *?thesis* **.**
qed

lemma *insertion-Sum-any*:
assumes *finite* {*x*. *g x* $\neq$ *0*}
shows *insertion f* (*Sum-any g*) = *Sum-any* (λ*x*. *insertion f* (*g x*))
unfolding *Sum-any.expand-set insertion-sum*
by (*intro sum.mono-neutral-right*) (*auto intro*!: *finite-subset*[*OF - assms*])

lemma *keys-diff-subset*:
keys (*f* − *g*) $\subseteq$ *keys f* $\cup$ *keys g*
by *transfer auto*

lemma *keys-empty-iff* [*simp*]: *keys p* = {} $\longleftrightarrow$ *p* = *0*
by *transfer auto*

lemma *mpoly-coeff-0* [*simp*]: *MPoly-Type.coeff 0 m* = *0*
by *transfer auto*

lemma *lookup-1*: *lookup 1 m* = (*if m* = *0 then 1 else 0*)
by *transfer* (*simp add*: *when-def*)

lemma *mpoly-coeff-1*: *MPoly-Type.coeff 1 m* = (*if m* = *0 then 1 else 0*)
by (*simp add*: *MPoly-Type.coeff-def one-mpoly.rep-eq lookup-1*)

lemma *lookup-Const*$_0$: *lookup* (*Const*$_0$ *c*) *m* = (*if m* = *0 then c else 0*)
unfolding *Const*$_0$*-def* **by** (*simp add*: *lookup-single when-def*)

lemma *mpoly-coeff-Const*: *MPoly-Type.coeff* (*Const c*) *m* = (*if m* = *0 then c else 0*)
by (*simp add*: *MPoly-Type.coeff-def Const.rep-eq lookup-Const*$_0$)

lemma *coeff-smult* [*simp*]: *coeff* (*smult c p*) *m* = (*c* :: *'a* :: *mult-zero*) $*$ *coeff p m*
by *transfer* (*auto simp*: *map-lookup*)

lemma *in-keys-mapI*: *x* $\in$ *keys m* $\Longrightarrow$ *f* (*lookup m x*) $\neq$ *0* $\Longrightarrow$ *x* $\in$ *keys* (*Poly-Mapping.map f m*)
by *transfer auto*

lemma *keys-uminus* [*simp*]: *keys* (−*m*) = *keys m*
by *transfer auto*

lemma *vars-uminus* [*simp*]: *vars* (−*p*) = *vars p*
unfolding *vars-def* **by** *transfer' auto*

lemma *vars-smult*: *vars* (*smult c p*) ⊆ *vars p*
unfolding *vars-def* **by** (*transfer'*, *transfer'*) *auto*

lemma *vars-0* [*simp*]: *vars 0* = {}
unfolding *vars-def* **by** *transfer' simp*

lemma *vars-1* [*simp*]: *vars 1* = {}
unfolding *vars-def* **by** *transfer' simp*

lemma *vars-sum*: *vars* (*sum f A*) ⊆ (⋃*x*∈*A*. *vars* (*f x*))
using *vars-add* **by** (*induction A rule*: *infinite-finite-induct*) *auto*

lemma *vars-prod*: *vars* (*prod f A*) ⊆ (⋃*x*∈*A*. *vars* (*f x*))
using *vars-mult* **by** (*induction A rule*: *infinite-finite-induct*) *auto*

lemma *vars-Sum-any*: *vars* (*Sum-any h*) ⊆ (⋃*i*. *vars* (*h i*))
unfolding *Sum-any.expand-set* **by** (*intro order.trans*[*OF vars-sum*]) *auto*

lemma *vars-Prod-any*: *vars* (*Prod-any h*) ⊆ (⋃*i*. *vars* (*h i*))
unfolding *Prod-any.expand-set* **by** (*intro order.trans*[*OF vars-prod*]) *auto*

lemma *vars-power*: *vars* (*p* ^ *n*) ⊆ *vars p*
using *vars-mult* **by** (*induction n*) *auto*

lemma *vars-diff*: *vars* (*p1* − *p2*) ⊆ *vars p1* ∪ *vars p2*
unfolding *vars-def*
proof *transfer'*
fix *p1 p2* :: (*nat* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ *'a*
show ⋃ (*keys* ‘ *keys* (*p1* − *p2*)) ⊆ ⋃(*keys* ‘ (*keys p1*)) ∪ ⋃(*keys* ‘ (*keys p2*))
using *keys-diff-subset*[*of p1 p2*] **by** (*auto simp flip*: *not-in-keys-iff-lookup-eq-zero*)
qed

lemma *insertion-smult* [*simp*]: *insertion f* (*smult c p*) = *c* ∗ *insertion f p*
unfolding *insertion-altdef*
by (*subst Sum-any-right-distrib*)
(*auto intro*: *finite-subset*[*OF - finite-coeff-support*[*of p*]] *simp*: *mult.assoc*)

lemma *coeff-add* [*simp*]: *coeff* (*p* + *q*) *m* = *coeff p m* + *coeff q m*
by *transfer'* (*simp add*: *lookup-add*)

lemma *coeff-diff* [*simp*]: *coeff* (*p* − *q*) *m* = *coeff p m* − *coeff q m*
by *transfer'* (*simp add*: *lookup-minus*)

lemma *insertion-monom* [*simp*]:
insertion f (*monom m c*) = *c* ∗ *Prod-any* (λ*x*. *f x* ^ *lookup m x*)
proof −
have *insertion f* (*monom m c*) =
(∑*m'*. (*c when m* = *m'*) ∗ (∏*v*. *f v* ^ *lookup m' v*))
by (*simp add*: *insertion-def insertion-aux-def insertion-fun-def lookup-single*)

also have ... = $c * (\prod v.\ f\ v$ ^ *lookup m v*)
  by (*subst Sum-any.expand-superset*[*of* {*m*}]) (*auto simp*: *when-def*)
finally show *?thesis* **.**
qed

lemma *insertion-aux-Const*$_0$ [*simp*]: *insertion-aux f* (*Const*$_0$ *c*) = *c*
proof −
  have *insertion-aux f* (*Const*$_0$ *c*) = ($\sum$ *m*. (*c when m* = *0*) * ($\prod$ *v*. *f v* ^ *lookup m v*))
    by (*simp add*: *Const*$_0$*-def insertion-aux-def insertion-fun-def lookup-single*)
  also have ... = ($\sum$ *m*∈{*0*}. (*c when m* = *0*) * ($\prod$ *v*. *f v* ^ *lookup m v*))
    by (*intro Sum-any.expand-superset*) (*auto simp*: *when-def*)
  also have ... = *c* **by** *simp*
  finally show *?thesis* **.**
qed

lemma *insertion-Const* [*simp*]: *insertion f* (*Const c*) = *c*
  by (*simp add*: *insertion-def Const.rep-eq*)

lemma *coeffs-0* [*simp*]: *coeffs 0* = {}
  by *transfer auto*

lemma *coeffs-1* [*simp*]: *coeffs 1* = {*1*}
  by *transfer auto*

lemma *coeffs-Const*: *coeffs* (*Const c*) = (*if c* = *0 then* {} *else* {*c*})
  unfolding *Const-def Const*$_0$*-def* **by** *transfer' auto*

lemma *coeffs-subset*: *coeffs* (*Const c*) ⊆ {*c*}
  by (*auto simp*: *coeffs-Const*)

lemma *keys-Const*$_0$: *keys* (*Const*$_0$ *c*) = (*if c* = *0 then* {} *else* {*0*})
  unfolding *Const*$_0$*-def* **by** *transfer' auto*

lemma *vars-Const* [*simp*]: *vars* (*Const c*) = {}
  unfolding *vars-def* **by** *transfer'* (*auto simp*: *keys-Const*$_0$)

lemma *prod-fun-compose-bij*:
  assumes *bij f* **and** *f*: $\bigwedge$*x y*. *f* (*x* + *y*) = *f x* + *f y*
  shows *prod-fun m1 m2* (*f x*) = *prod-fun* (*m1* ∘ *f*) (*m2* ∘ *f*) *x*
proof −
  have [*simp*]: *f x* = *f y* ⟷ *x* = *y* **for** *x y*
    using ‹*bij f*› **by** (*auto dest*!: *bij-is-inj inj-onD*)
  have *prod-fun* (*m1* ∘ *f*) (*m2* ∘ *f*) *x* =
        Sum-any ((λ*l*. *m1 l* * *Sum-any* ((λ*q*. *m2 q when f x* = *l* + *q*) ∘ *f*)) ∘ *f*)
    by (*simp add*: *prod-fun-def f*(*1*) [*symmetric*] *o-def*)
  also have ... = *Sum-any* ((λ*l*. *m1 l* * *Sum-any* ((λ*q*. *m2 q when f x* = *l* + *q*))))
    by (*simp only*: *Sum-any.reindex-cong*[*OF assms*(*1*) *refl*, *symmetric*])
  also have ... = *prod-fun m1 m2* (*f x*)

by (*simp add*: *prod-fun-def*)
finally show *?thesis* **..**
qed

lemma *add-nat-poly-mapping-zero-iff* [*simp*]:
$(a + b :: 'a \Rightarrow_0 nat) = 0 \longleftrightarrow a = 0 \wedge b = 0$
by *transfer* (*auto simp*: *fun-eq-iff*)

lemma *prod-fun-nat-0*:
fixes $f\ g :: ('a \Rightarrow_0 nat) \Rightarrow 'b::semiring\text{-}0$
shows $prod\text{-}fun\ f\ g\ 0 = f\ 0 * g\ 0$
proof −
have $prod\text{-}fun\ f\ g\ 0 = (\sum l.\ f\ l * (\sum q.\ g\ q\ when\ 0 = l + q))$
unfolding *prod-fun-def* **..**
also have $(\lambda l.\ \sum q.\ g\ q\ when\ 0 = l + q) = (\lambda l.\ \sum q{\in}\{0\}.\ g\ q\ when\ 0 = l + q)$
by (*intro ext Sum-any.expand-superset*) (*auto simp*: *when-def*)
also have $(\sum l.\ f\ l * \ldots\ l) = (\sum l{\in}\{0\}.\ f\ l * \ldots\ l)$
by (*intro ext Sum-any.expand-superset*) (*auto simp*: *when-def*)
finally show *?thesis* **by** *simp*
qed

lemma *mpoly-coeff-times-0*: $coeff\ (p * q)\ 0 = coeff\ p\ 0 * coeff\ q\ 0$
by (*simp add*: *coeff-mpoly-times prod-fun-nat-0*)

lemma *mpoly-coeff-prod-0*: $coeff\ (\prod x{\in}A.\ f\ x)\ 0 = (\prod x{\in}A.\ coeff\ (f\ x)\ 0)$
by (*induction A rule*: *infinite-finite-induct*) (*auto simp*: *mpoly-coeff-times-0 mpoly-coeff-1*)

lemma *mpoly-coeff-power-0*: $coeff\ (p\ \hat{}\ n)\ 0 = coeff\ p\ 0\ \hat{}\ n$
by (*induction n*) (*auto simp*: *mpoly-coeff-times-0 mpoly-coeff-1*)

lemma *prod-fun-max*:
fixes $f\ g :: 'a::\{linorder,\ ordered\text{-}cancel\text{-}comm\text{-}monoid\text{-}add\} \Rightarrow 'b::semiring\text{-}0$
assumes *zero*: $\bigwedge m.\ m > a \Longrightarrow f\ m = 0\ \bigwedge m.\ m > b \Longrightarrow g\ m = 0$
assumes *fin*: $finite\ \{m.\ f\ m \neq 0\}\ finite\ \{m.\ g\ m \neq 0\}$
shows $prod\text{-}fun\ f\ g\ (a + b) = f\ a * g\ b$
proof −
note $fin' = finite\text{-}subset[OF\ \text{-}\ fin(1)]\ finite\text{-}subset[OF\ \text{-}\ fin(2)]$
have $prod\text{-}fun\ f\ g\ (a + b) = (\sum l.\ f\ l * (\sum q.\ g\ q\ when\ a + b = l + q))$
by (*simp add*: *prod-fun-def Sum-any-right-distrib*)
also have $\ldots = (\sum l.\ \sum q.\ f\ l * g\ q\ when\ a + b = l + q)$
by (*subst Sum-any-right-distrib*) (*auto intro*!: *Sum-any.cong fin'*(*2*) *simp*: *when-def*)
also {
fix *l q* **assume** *lq*: $a + b = l + q\ (a,\ b) \neq (l,\ q)$ **and** *nz*: $f\ l * g\ q \neq 0$
from *nz* **and** *zero* **have** $l \le a\ q \le b$ **by** (*auto intro*: *leI*)
moreover from *this* **and** *lq*(*2*) **have** $l < a \vee q < b$ **by** *auto*
ultimately have $l + q < a + b$
by (*auto intro*: *add-less-le-mono add-le-less-mono*)
with *lq*(*1*) **have** *False* **by** *simp*
}

hence $(\sum l. \sum q. f\ l * g\ q\ when\ a + b = l + q) = (\sum l. \sum q. f\ l * g\ q\ when\ (a, b) = (l, q))$
  by (*intro Sum-any.cong refl*) (*auto simp*: *when-def*)
 also have $\ldots = (\sum (l,q). f\ l * g\ q\ when\ (a, b) = (l, q))$
  by (*intro Sum-any.cartesian-product*[*of* $\{(a, b)\}$]) *auto*
 also have $\ldots = (\sum (l,q)\in\{(a,b)\}. f\ l * g\ q\ when\ (a, b) = (l, q))$
  by (*intro Sum-any.expand-superset*) *auto*
 also have $\ldots = f\ a * g\ b$ **by** *simp*
 finally show *?thesis* **.**
qed

lemma *prod-fun-gt-max-eq-zero*:
 fixes $f\ g$ :: $'a$::{*linorder*, *ordered-cancel-comm-monoid-add*} $\Rightarrow$ $'b$::*semiring-0*
 assumes $m > a + b$
 assumes *zero*: $\bigwedge m.\ m > a \Longrightarrow f\ m = 0$ $\bigwedge m.\ m > b \Longrightarrow g\ m = 0$
 assumes *fin*: *finite* $\{m.\ f\ m \neq 0\}$ *finite* $\{m.\ g\ m \neq 0\}$
 shows *prod-fun* $f\ g\ m = 0$
proof −
 note *fin'* = *finite-subset*[*OF - fin*(*1*)] *finite-subset*[*OF - fin*(*2*)]
 have *prod-fun* $f\ g\ m = (\sum l. f\ l * (\sum q. g\ q\ when\ m = l + q))$
  by (*simp add*: *prod-fun-def Sum-any-right-distrib*)
 also have $\ldots = (\sum l. \sum q. f\ l * g\ q\ when\ m = l + q)$
  by (*subst Sum-any-right-distrib*) (*auto intro*!: *Sum-any.cong fin'*(*2*) *simp*: *when-def*)
 also {
  fix $l\ q$ **assume** *lq*: $m = l + q$ **and** *nz*: $f\ l * g\ q \neq 0$
  from *nz* **and** *zero* **have** $l \leq a$ $q \leq b$ **by** (*auto intro*: *leI*)
  hence $l + q \leq a + b$ **by** (*intro add-mono*)
  also have $\ldots < m$ **by** *fact*
  finally have $l + q < m$ **.**
 }
 hence $(\sum l. \sum q. f\ l * g\ q\ when\ m = l + q) = (\sum l. \sum q. f\ l * g\ q\ when\ False)$
  by (*intro Sum-any.cong refl*) (*auto simp*: *when-def*)
 also have $\ldots = 0$ **by** *simp*
 finally show *?thesis* **.**
qed

2.4 Restricting a monomial to a subset of variables

lift-definition *restrictpm* :: $'a$ *set* $\Rightarrow$ $('a \Rightarrow_0 'b :: zero) \Rightarrow ('a \Rightarrow_0 'b)$ **is**
 $\lambda A\ f\ x.$ *if* $x \in A$ *then* $f\ x$ *else* 0
 by (*erule finite-subset*[*rotated*]) *auto*

lemma *lookup-restrictpm*: *lookup* (*restrictpm* $A\ m$) x = (*if* $x \in A$ *then lookup* $m\ x$ *else* 0)
 by *transfer auto*

lemma *lookup-restrictpm-in* [*simp*]: $x \in A \Longrightarrow$ *lookup* (*restrictpm* $A\ m$) x = *lookup* $m\ x$
 and *lookup-restrict-pm-not-in* [*simp*]: $x \notin A \Longrightarrow$ *lookup* (*restrictpm* $A\ m$) $x = 0$

by (*simp-all add*: *lookup-restrictpm*)

lemma *keys-restrictpm* [*simp*]: *keys* (*restrictpm* A *m*) = *keys* $m \cap A$
by *transfer auto*

lemma *restrictpm-add*: *restrictpm* X (*m1* + *m2*) = *restrictpm* X *m1* + *restrictpm* X *m2*
by *transfer auto*

lemma *restrictpm-id* [*simp*]: *keys* $m \subseteq X \Longrightarrow$ *restrictpm* X $m = m$
by *transfer* (*auto simp*: *fun-eq-iff*)

lemma *restrictpm-orthogonal* [*simp*]: *keys* $m \subseteq -X \Longrightarrow$ *restrictpm* X $m = 0$
by *transfer* (*auto simp*: *fun-eq-iff*)

lemma *restrictpm-add-disjoint*:
$X \cap Y = \{\} \Longrightarrow$ *restrictpm* X m + *restrictpm* Y m = *restrictpm* $(X \cup Y)$ m
by *transfer* (*auto simp*: *fun-eq-iff*)

lemma *restrictpm-add-complements*:
restrictpm X m + *restrictpm* $(-X)$ $m = m$ *restrictpm* $(-X)$ m + *restrictpm* X $m = m$
by (*subst restrictpm-add-disjoint*; *force*)+

2.5 Mapping over a polynomial

lift-definition *map-mpoly* :: $('a :: zero \Rightarrow 'b :: zero) \Rightarrow 'a\ mpoly \Rightarrow 'b\ mpoly$ **is**
$\lambda(f :: 'a \Rightarrow 'b)\ (p :: (nat \Rightarrow_0 nat) \Rightarrow_0 'a).$ *Poly-Mapping.map f p* **.**

lift-definition *mapm-mpoly* :: $((nat \Rightarrow_0 nat) \Rightarrow 'a :: zero \Rightarrow 'b :: zero) \Rightarrow 'a\ mpoly \Rightarrow 'b\ mpoly$ **is**
$\lambda(f :: (nat \Rightarrow_0 nat) \Rightarrow 'a \Rightarrow 'b)\ (p :: (nat \Rightarrow_0 nat) \Rightarrow_0 'a).$
Poly-Mapping.mapp f p **.**

lemma *poly-mapping-map-conv-mapp*: *Poly-Mapping.map f* = *Poly-Mapping.mapp* (λ-. *f*)
by (*auto simp*: *Poly-Mapping.mapp-def Poly-Mapping.map-def map-fun-def o-def fun-eq-iff when-def in-keys-iff cong*: *if-cong*)

lemma *map-mpoly-conv-mapm-mpoly*: *map-mpoly f* = *mapm-mpoly* (λ-. *f*)
by *transfer'* (*auto simp*: *poly-mapping-map-conv-mapp*)

lemma *map-mpoly-comp*: $f\ 0 = 0 \Longrightarrow$ *map-mpoly f* (*map-mpoly g p*) = *map-mpoly* $(f \circ g)$ *p*
by (*transfer'*, *transfer'*) (*auto simp*: *when-def fun-eq-iff*)

lemma *mapp-mapp*:
$(\bigwedge x.\ f\ x\ 0 = 0) \Longrightarrow$ *Poly-Mapping.mapp f* (*Poly-Mapping.mapp g m*) =
Poly-Mapping.mapp $(\lambda x\ y.\ f\ x\ (g\ x\ y))$ *m*

by *transfer′* (*auto simp*: *fun-eq-iff lookup-mapp in-keys-iff*)

lemma *mapm-mpoly-comp*:
$(\bigwedge x.\ f\ x\ 0 = 0) \Longrightarrow$ *mapm-mpoly f* (*mapm-mpoly g p*) = *mapm-mpoly* ($\lambda m\ c.\ f\ m\ (g\ m\ c)$) *p*
by *transfer′* (*simp add*: *mapp-mapp*)

lemma *coeff-map-mpoly*:
coeff (*map-mpoly f p*) *m* = (*if coeff p m* = *0 then 0 else f* (*coeff p m*))
by (*transfer*, *transfer*) *auto*

lemma *coeff-map-mpoly′* [*simp*]: *f 0* = *0* $\Longrightarrow$ *coeff* (*map-mpoly f p*) *m* = *f* (*coeff p m*)
by (*subst coeff-map-mpoly*) *auto*

lemma *coeff-mapm-mpoly*: *coeff* (*mapm-mpoly f p*) *m* = (*if coeff p m* = *0 then 0 else f m* (*coeff p m*))
by (*transfer*, *transfer′*) (*auto simp*: *in-keys-iff*)

lemma *coeff-mapm-mpoly′* [*simp*]: $(\bigwedge m.\ f\ m\ 0 = 0) \Longrightarrow$ *coeff* (*mapm-mpoly f p*) *m* = *f m* (*coeff p m*)
by (*subst coeff-mapm-mpoly*) *auto*

lemma *vars-map-mpoly-subset*: *vars* (*map-mpoly f p*) $\subseteq$ *vars p*
unfolding *vars-def* **by** (*transfer′*, *transfer′*) (*auto simp*: *map-mpoly.rep-eq*)

lemma *coeff-sum* [*simp*]: *coeff* (*sum f A*) *m* = ($\sum x \in A$. *coeff* (*f x*) *m*)
by (*induction A rule*: *infinite-finite-induct*) *auto*

lemma *coeff-Sum-any*: *finite* $\{x.\ f\ x \neq 0\}$ $\Longrightarrow$ *coeff* (*Sum-any f*) *m* = *Sum-any* (λx. *coeff* (*f x*) *m*)
by (*auto simp add*: *Sum-any.expand-set intro*!: *sum.mono-neutral-right*)

lemma *Sum-any-zeroI*: $(\bigwedge x.\ f\ x = 0) \Longrightarrow$ *Sum-any f* = *0*
by (*auto simp*: *Sum-any.expand-set*)

lemma *insertion-Prod-any*:
finite $\{x.\ g\ x \neq 1\}$ $\Longrightarrow$ *insertion f* (*Prod-any g*) = *Prod-any* (λx. *insertion f* (*g x*))
by (*auto simp*: *Prod-any.expand-set insertion-prod intro*!: *prod.mono-neutral-right*)

lemma *insertion-insertion*:
insertion g (*insertion k p*) =
insertion (λx. *insertion g* (*k x*)) (*map-mpoly* (*insertion g*) *p*) (**is** *?lhs* = *?rhs*)
proof −
have *insertion g* (*insertion k p*) =
($\sum x$. *insertion g* (*coeff p x*) $*$ *insertion g* ($\prod i$. *k i* ^ *lookup x i*))
unfolding *insertion-altdef*[*of k p*]
by (*subst insertion-Sum-any*)

(*auto intro*: *finite-subset*[*OF* - *finite-coeff-support*[*of p*]] *simp*: *insertion-mult*)
also have ... = ($\sum$ *x*. *insertion g* (*coeff p x*) $*$ ($\prod$ *i*. *insertion g* (*k i*) ^ *lookup x i*))
proof (*intro Sum-any.cong*)
fix *x* **show** *insertion g* (*coeff p x*) $*$ *insertion g* ($\prod$ *i*. *k i* ^ *lookup x i*) =
insertion g (*coeff p x*) $*$ ($\prod$ *i*. *insertion g* (*k i*) ^ *lookup x i*)
by (*subst insertion-Prod-any*)
(*auto simp*: *insertion-power intro*!: *finite-subset*[*OF* - *finite-lookup*[*of x*]] *Nat.gr0I*)
qed
also have ... = *insertion* (λ*x*. *insertion g* (*k x*)) (*map-mpoly* (*insertion g*) *p*)
unfolding *insertion-altdef*[*of* - *map-mpoly f p* **for** *f*] **by** *auto*
finally show *?thesis* **.**
qed

lemma *insertion-substitute-linear*:
insertion (λ*i*. *c i* $*$ *f i*) *p* =
insertion f (*mapm-mpoly* (λ*m d*. *Prod-any* (λ*i*. *c i* ^ *lookup m i*) $*$ *d*) *p*)
unfolding *insertion-altdef*
proof (*intro Sum-any.cong*, *goal-cases*)
case (*1 m*)
have *coeff* (*mapm-mpoly* (λ*m*. ($*$) ($\prod$ *i*. *c i* ^ *lookup m i*)) *p*) *m* $*$ ($\prod$ *i*. *f i* ^ *lookup m i*) =
MPoly-Type.coeff p m $*$ (($\prod$ *i*. *c i* ^ *lookup m i*) $*$ ($\prod$ *i*. *f i* ^ *lookup m i*))
by (*simp add*: *mult-ac*)
also have ($\prod$ *i*. *c i* ^ *lookup m i*) $*$ ($\prod$ *i*. *f i* ^ *lookup m i*) =
($\prod$ *i*. (*c i* $*$ *f i*) ^ *lookup m i*)
by (*subst Prod-any.distrib* [*symmetric*])
(*auto simp*: *power-mult-distrib intro*!: *finite-subset*[*OF* - *finite-lookup*[*of m*]] *Nat.gr0I*)
finally show *?case* **by** *simp*
qed

lemma *vars-mapm-mpoly-subset*: *vars* (*mapm-mpoly f p*) $\subseteq$ *vars p*
unfolding *vars-def* **using** *keys-mapp-subset*[*of f*] **by** (*auto simp*: *mapm-mpoly.rep-eq*)

lemma *map-mpoly-cong*:
assumes $\bigwedge$*m*. *f* (*coeff p m*) = *g* (*coeff p m*) *p* = *q*
shows *map-mpoly f p* = *map-mpoly g q*
using *assms* **by** (*intro mpoly-eqI*) (*auto simp*: *coeff-map-mpoly*)

2.6 The leading monomial and leading coefficient

The leading monomial of a multivariate polynomial is the one with the largest monomial w. r. t. the monomial ordering induced by the standard variable ordering. The leading coefficient is the coefficient of the leading monomial.

As a convention, the leading monomial of the zero polynomial is defined to be the same as that of any non-constant zero polynomial, i. e. the monomial $X_1^0 \dots X_n^0$.

lift-definition *lead-monom* :: $'a$ :: *zero mpoly* $\Rightarrow$ (*nat* $\Rightarrow_0$ *nat*) **is**
λf :: (*nat* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ $'a$. *Max* (*insert 0* (*keys f*)) **.**

lemma *lead-monom-geI* [*intro*]:
assumes *coeff p m* $\neq$ *0*
shows $m \leq$ *lead-monom p*
using *assms* **by** (*auto simp*: *lead-monom-def coeff-def in-keys-iff*)

lemma *coeff-gt-lead-monom-zero* [*simp*]:
assumes $m >$ *lead-monom p*
shows *coeff p m* = *0*
using *lead-monom-geI*[*of p m*] *assms* **by** *force*

lemma *lead-monom-nonzero-eq*:
assumes $p \neq$ *0*
shows *lead-monom p* = *Max* (*keys* (*mapping-of p*))
using *assms* **by** *transfer* (*simp add*: *max-def*)

lemma *lead-monom-0* [*simp*]: *lead-monom 0* = *0*
by (*simp add*: *lead-monom-def zero-mpoly.rep-eq*)

lemma *lead-monom-1* [*simp*]: *lead-monom 1* = *0*
by (*simp add*: *lead-monom-def one-mpoly.rep-eq*)

lemma *lead-monom-Const* [*simp*]: *lead-monom* (*Const c*) = *0*
by (*simp add*: *lead-monom-def Const.rep-eq* $Const_0$*-def*)

lemma *lead-monom-uminus* [*simp*]: *lead-monom* ($-p$) = *lead-monom p*
by (*simp add*: *lead-monom-def uminus-mpoly.rep-eq*)

lemma *keys-mult-const* [*simp*]:
fixes *c* :: $'a$ :: {*semiring-0*, *semiring-no-zero-divisors*}
assumes $c \neq$ *0*
shows *keys* (*Poly-Mapping.map* (($*$) *c*) *p*) = *keys p*
using *assms* **by** *transfer auto*

lemma *lead-monom-eq-0-iff*: *lead-monom p* = *0* $\longleftrightarrow$ *vars p* = {}
unfolding *vars-def* **by** *transfer'* (*auto simp*: *Max-eq-iff*)

lemma *lead-monom-monom*: *lead-monom* (*monom m c*) = (*if c* = *0 then 0 else m*)
by (*auto simp add*: *lead-monom-def monom.rep-eq* $Const_0$*-def max-def*)

lemma *lead-monom-monom'* [*simp*]: $c \neq$ *0* $\Longrightarrow$ *lead-monom* (*monom m c*) = *m*
by (*simp add*: *lead-monom-monom*)

lemma *lead-monom-numeral* [*simp*]: *lead-monom* (*numeral n*) = *0*

unfolding *monom-numeral*[*symmetric*] **by** (*subst lead-monom-monom*) *auto*

lemma *lead-monom-add*: *lead-monom* $(p + q) \leq$ *max* (*lead-monom p*) (*lead-monom q*)
proof *transfer*
fix $p\ q :: (nat \Rightarrow_0 nat) \Rightarrow_0 'a$
show *Max* (*insert 0* (*keys* $(p + q)$)) $\leq$ *max* (*Max* (*insert 0* (*keys p*))) (*Max* (*insert 0* (*keys q*)))
proof (*rule Max.boundedI*)
fix *m* **assume** *m*: $m \in$ *insert 0* (*keys* $(p + q)$)
thus $m \leq$ *max* (*Max* (*insert 0* (*keys p*))) (*Max* (*insert 0* (*keys q*)))
proof
assume $m \in$ *keys* $(p + q)$
with *keys-add*[*of p q*] **have** $m \in$ *keys p* $\vee$ $m \in$ *keys q*
by (*auto simp*: *in-keys-iff plus-poly-mapping.rep-eq*)
thus *?thesis* **by** (*auto simp*: *le-max-iff-disj*)
qed *auto*
qed *auto*
qed

lemma *lead-monom-diff*: *lead-monom* $(p - q) \leq$ *max* (*lead-monom p*) (*lead-monom q*)
proof *transfer*
fix $p\ q :: (nat \Rightarrow_0 nat) \Rightarrow_0 'a$
show *Max* (*insert 0* (*keys* $(p - q)$)) $\leq$ *max* (*Max* (*insert 0* (*keys p*))) (*Max* (*insert 0* (*keys q*)))
proof (*rule Max.boundedI*)
fix *m* **assume** *m*: $m \in$ *insert 0* (*keys* $(p - q)$)
thus $m \leq$ *max* (*Max* (*insert 0* (*keys p*))) (*Max* (*insert 0* (*keys q*)))
proof
assume $m \in$ *keys* $(p - q)$
with *keys-diff-subset*[*of p q*] **have** $m \in$ *keys p* $\vee$ $m \in$ *keys q* **by** *auto*
thus *?thesis* **by** (*auto simp*: *le-max-iff-disj*)
qed *auto*
qed *auto*
qed

definition *lead-coeff* **where** *lead-coeff p* = *coeff p* (*lead-monom p*)

lemma *vars-empty-iff*: *vars p* = {} $\longleftrightarrow$ *p* = *Const* (*lead-coeff p*)
proof
assume *vars p* = {}
hence [*simp*]: *lead-monom p* = *0*
by (*simp add*: *lead-monom-eq-0-iff*)
have [*simp*]: $mon \neq 0 \longleftrightarrow (mon > (0 :: nat \Rightarrow_0 nat))$ **for** *mon*
by (*auto simp*: *order.strict-iff-order*)
thus *p* = *Const* (*lead-coeff p*)
by (*intro mpoly-eqI*) (*auto simp*: *mpoly-coeff-Const lead-coeff-def*)

next
  assume *p* = *Const* (*lead-coeff p*)
  also have *vars* ... = {} **by** *simp*
  finally show *vars p* = {} **.**
qed

lemma *lead-coeff-0* [*simp*]: *lead-coeff 0* = *0*
  by (*simp add*: *lead-coeff-def*)

lemma *lead-coeff-1* [*simp*]: *lead-coeff 1* = *1*
  by (*simp add*: *lead-coeff-def mpoly-coeff-1*)

lemma *lead-coeff-Const* [*simp*]: *lead-coeff* (*Const c*) = *c*
  by (*simp add*: *lead-coeff-def mpoly-coeff-Const*)

lemma *lead-coeff-monom* [*simp*]: *lead-coeff* (*monom p c*) = *c*
  by (*simp add*: *lead-coeff-def coeff-monom when-def lead-monom-monom*)

lemma *lead-coeff-nonzero* [*simp*]: *p* ≠ *0* ⟹ *lead-coeff p* ≠ *0*
  unfolding *lead-coeff-def lead-monom-def*
  by (*cases keys* (*mapping-of p*) = {}) (*auto simp*: *coeff-def max-def*)

lemma
  fixes *c* :: *'a* :: *semiring-0*
  assumes *c* ∗ *lead-coeff p* ≠ *0*
  shows *lead-monom-smult* [*simp*]: *lead-monom* (*smult c p*) = *lead-monom p*
    and *lead-coeff-smult* [*simp*]: *lead-coeff* (*smult c p*) = *c* ∗ *lead-coeff p*
proof −
  from *assms* **have** ∗: *keys* (*mapping-of p*) ≠ {}
    by *auto*
  from *assms* **have** *coeff* (*MPoly-Type.smult c p*) (*lead-monom p*) ≠ *0*
    by (*simp add*: *lead-coeff-def*)
  hence *smult-nz*: *MPoly-Type.smult c p* ≠ *0* **by** (*auto simp del*: *coeff-smult*)
  with *assms* **have** ∗∗: *keys* (*mapping-of* (*smult c p*)) ≠ {}
    by *simp*

  have *Max* (*keys* (*mapping-of* (*smult c p*))) = *Max* (*keys* (*mapping-of p*))
  proof (*safe intro*!: *antisym Max.coboundedI*)
    have *lookup* (*mapping-of p*) (*Max* (*keys* (*mapping-of p*))) = *lead-coeff p*
      using ∗ **by** (*simp add*: *lead-coeff-def lead-monom-def max-def coeff-def*)
    with *assms* **show** *Max* (*keys* (*mapping-of p*)) ∈ *keys* (*mapping-of* (*smult c p*))
      using ∗ **by** (*auto simp*: *smult.rep-eq intro*!: *in-keys-mapI*)
    from *smult-nz* **have** *lead-coeff* (*smult c p*) ≠ *0*
      by (*intro lead-coeff-nonzero*) *auto*
    hence *coeff p* (*Max* (*keys* (*mapping-of* (*smult c p*)))) ≠ *0*
      using *assms* ∗ ∗∗ **by** (*auto simp*: *lead-coeff-def lead-monom-def max-def*)
    thus *Max* (*keys* (*mapping-of* (*smult c p*))) ∈ *keys* (*mapping-of p*)
      by (*auto simp*: *smult.rep-eq coeff-def in-keys-iff*)
  qed *auto*

with $*$ $**$ **show** *lead-monom* (*smult c p*) = *lead-monom p*
  by (*simp add*: *lead-monom-def max-def*)
thus *lead-coeff* (*smult c p*) = *c* $*$ *lead-coeff p*
  by (*simp add*: *lead-coeff-def*)
qed

lemma *lead-coeff-mult-aux*:
  coeff (*p* $*$ *q*) (*lead-monom p* + *lead-monom q*) = *lead-coeff p* $*$ *lead-coeff q*
proof (*cases p* = *0* ∨ *q* = *0*)
  case *False*
  define *a b* **where** *a* = *lead-monom p* **and** *b* = *lead-monom q*
  have *coeff* (*p* $*$ *q*) (*a* + *b*) = *coeff p a* $*$ *coeff q b*
    unfolding *coeff-mpoly-times*
    by (*rule prod-fun-max*) (*insert False*, *auto simp*: *a-def b-def*)
  thus *?thesis* **by** (*simp add*: *a-def b-def lead-coeff-def*)
qed *auto*

lemma *lead-monom-mult-le*: *lead-monom* (*p* $*$ *q*) ≤ *lead-monom p* + *lead-monom*
q
proof (*cases p* $*$ *q* = *0*)
  case *False*
  show *?thesis*
  proof (*intro leI notI*)
    assume *lead-monom p* + *lead-monom q* < *lead-monom* (*p* $*$ *q*)
    hence *lead-coeff* (*p* $*$ *q*) = *0*
      unfolding *lead-coeff-def coeff-mpoly-times* **by** (*rule prod-fun-gt-max-eq-zero*)
auto
    with *False* **show** *False* **by** *simp*
  qed
qed *auto*

lemma *lead-monom-mult*:
  assumes *lead-coeff p* $*$ *lead-coeff q* ≠ *0*
  shows *lead-monom* (*p* $*$ *q*) = *lead-monom p* + *lead-monom q*
  by (*intro antisym lead-monom-mult-le lead-monom-geI*)
    (*insert assms*, *auto simp*: *lead-coeff-mult-aux*)

lemma *lead-coeff-mult*:
  assumes *lead-coeff p* $*$ *lead-coeff q* ≠ *0*
  shows *lead-coeff* (*p* $*$ *q*) = *lead-coeff p* $*$ *lead-coeff q*
  using *assms* **by** (*simp add*: *lead-monom-mult lead-coeff-mult-aux lead-coeff-def*)

lemma *keys-lead-monom-subset*: *keys* (*lead-monom p*) ⊆ *vars p*
proof (*cases p* = *0*)
  case *False*
  hence *lead-coeff p* ≠ *0* **by** *simp*
  hence *coeff p* (*lead-monom p*) ≠ *0* **unfolding** *lead-coeff-def* **.**
  thus *?thesis* **unfolding** *vars-def* **by** *transfer′* (*auto simp*: *max-def in-keys-iff*)
qed *auto*

lemma
  assumes (∏ i∈A. lead-coeff (f i)) ≠ 0
    shows lead-monom-prod: lead-monom (∏ i∈A. f i) = (∑ i∈A. lead-monom (f i)) (**is** ?th1)
      and lead-coeff-prod: lead-coeff (∏ i∈A. f i) = (∏ i∈A. lead-coeff (f i)) (**is** ?th2)
proof −
  have ?th1 ∧ ?th2 **using** assms
  proof (induction A rule: infinite-finite-induct)
    case (insert x A)
    from insert **have** nz: lead-coeff (f x) ≠ 0 (∏ i∈A. lead-coeff (f i)) ≠ 0 **by** auto
    note IH = insert.IH[OF this(2)]
    from insert **have** nz′: lead-coeff (f x) ∗ lead-coeff (∏ i∈A. f i) ≠ 0
      by (subst IH) auto
    from insert.prems insert.hyps nz nz′ **show** ?case
      by (auto simp: lead-monom-mult lead-coeff-mult IH)
  qed auto
  thus ?th1 ?th2 **by** blast+
qed

lemma lead-monom-sum-le: (⋀x. x ∈ X ⟹ lead-monom (h x) ≤ ub) ⟹ lead-monom (sum h X) ≤ ub
 by (induction X rule: infinite-finite-induct) (auto intro!: order.trans[OF lead-monom-add])

The leading monomial of a sum where the leading monomial the summands are distinct is simply the maximum of the leading monomials.

lemma lead-monom-sum:
  assumes inj-on (lead-monom ∘ h) X **and** finite X **and** X ≠ {} **and** ⋀x. x ∈ X ⟹ h x ≠ 0
  defines m ≡ Max ((lead-monom ∘ h) ‘ X)
  shows   lead-monom (∑ x∈X. h x) = m
proof (rule antisym)
  show lead-monom (sum h X) ≤ m **unfolding** m-def **using** assms
    by (intro lead-monom-sum-le Max-ge finite-imageI) auto
next
  from assms **have** m ∈ (lead-monom ∘ h) ‘ X
    unfolding m-def **by** (intro Max-in finite-imageI) auto
  then obtain x **where** x: x ∈ X m = lead-monom (h x) **by** auto
  have coeff (∑ x∈X. h x) m = (∑ x∈X. coeff (h x) m)
    by simp
  also have . . . = (∑ x∈{x}. coeff (h x) m)
  proof (intro sum.mono-neutral-right ballI)
    fix y **assume** y: y ∈ X − {x}
    hence (lead-monom ∘ h) y ≤ m
      using assms **unfolding** m-def **by** (intro Max-ge finite-imageI) auto
    moreover have (lead-monom ∘ h) y ≠ (lead-monom ∘ h) x
      using ‹x ∈ X› y inj-onD[OF assms(1), of x y] **by** auto

ultimately have *lead-monom* $(h\ y) < m$
using x **by** *auto*
thus *coeff* $(h\ y)\ m = 0$ **by** *simp*
qed (*insert x assms*, *auto*)
also have $\ldots = $ *coeff* $(h\ x)\ m$ **by** *simp*
also have $\ldots = $ *lead-coeff* $(h\ x)$ **using** x **by** (*simp add*: *lead-coeff-def*)
also have $\ldots \neq 0$ **using** *assms* x **by** *auto*
finally show *lead-monom* (*sum* $h\ X$) $\geq m$ **by** (*intro lead-monom-geI*)
qed

lemma *lead-coeff-eq-0-iff* [*simp*]: *lead-coeff* $p = 0 \longleftrightarrow p = 0$
by (*cases* $p = 0$) *auto*

lemma
fixes f :: - $\Rightarrow$ $'a$ :: *semidom mpoly*
assumes $\bigwedge i.\ i \in A \Longrightarrow f\ i \neq 0$
shows *lead-monom-prod'* [*simp*]: *lead-monom* $(\prod i{\in}A.\ f\ i) = (\sum i{\in}A.$ *lead-monom* $(f\ i))$ (**is** *?th1*)
and *lead-coeff-prod'* [*simp*]: *lead-coeff* $(\prod i{\in}A.\ f\ i) = (\prod i{\in}A.$ *lead-coeff* $(f\ i))$ (**is** *?th2*)
proof −
from *assms* **have** $(\prod i{\in}A.$ *lead-coeff* $(f\ i)) \neq 0$
by (*cases finite A*) *auto*
thus *?th1 ?th2* **by** (*simp-all add*: *lead-monom-prod lead-coeff-prod*)
qed

lemma
fixes p :: $'a$ :: *comm-semiring-1 mpoly*
assumes *lead-coeff* $p \mathbin{\hat{}} n \neq 0$
shows *lead-monom-power*: *lead-monom* $(p \mathbin{\hat{}} n) =$ *of-nat* $n\ *$ *lead-monom* p
and *lead-coeff-power*: *lead-coeff* $(p \mathbin{\hat{}} n) =$ *lead-coeff* $p \mathbin{\hat{}} n$
using *assms lead-monom-prod*[*of* λ-. p {..<n}] *lead-coeff-prod*[*of* λ-. p {..<n}]
by *simp-all*

lemma
fixes p :: $'a$ :: *semidom mpoly*
assumes $p \neq 0$
shows *lead-monom-power'* [*simp*]: *lead-monom* $(p \mathbin{\hat{}} n) =$ *of-nat* $n\ *$ *lead-monom* p
and *lead-coeff-power'* [*simp*]: *lead-coeff* $(p \mathbin{\hat{}} n) =$ *lead-coeff* $p \mathbin{\hat{}} n$
using *assms lead-monom-prod'*[*of* {..<n} λ-. p] *lead-coeff-prod'*[*of* {..<n} λ-. p]
by *simp-all*

2.7 Turning a set of variables into a monomial

Given a finite set $\{X_1, \ldots, X_n\}$ of variables, the following is the monomial $X_1 \ldots X_n$:

lift-definition *monom-of-set* :: *nat set* $\Rightarrow$ (*nat* $\Rightarrow_0$ *nat*) **is**
$\lambda X\ x.$ *if finite* $X \wedge x \in X$ *then* 1 *else* 0

by *auto*

lemma *lookup-monom-of-set*:
Poly-Mapping.lookup (*monom-of-set X*) *i* = (*if finite X* ∧ *i* ∈ *X then 1 else 0*)
by *transfer auto*

lemma *lookup-monom-of-set-1* [*simp*]:
finite X ⟹ *i* ∈ *X* ⟹ *Poly-Mapping.lookup* (*monom-of-set X*) *i* = *1*
and *lookup-monom-of-set-0* [*simp*]:
i ∉ *X* ⟹ *Poly-Mapping.lookup* (*monom-of-set X*) *i* = *0*
by (*simp-all add*: *lookup-monom-of-set*)

lemma *keys-monom-of-set*: *keys* (*monom-of-set X*) = (*if finite X then X else* {})
by *transfer auto*

lemma *keys-monom-of-set-finite* [*simp*]: *finite X* ⟹ *keys* (*monom-of-set X*) = *X*
by (*simp add*: *keys-monom-of-set*)

lemma *monom-of-set-eq-iff* [*simp*]: *finite X* ⟹ *finite Y* ⟹ *monom-of-set X* = *monom-of-set Y* ⟷ *X* = *Y*
by *transfer* (*auto simp*: *fun-eq-iff*)

lemma *monom-of-set-empty* [*simp*]: *monom-of-set* {} = *0*
by *transfer auto*

lemma *monom-of-set-eq-zero-iff* [*simp*]: *monom-of-set X* = *0* ⟷ *infinite X* ∨ *X* = {}
by *transfer* (*auto simp*: *fun-eq-iff*)

lemma *zero-eq-monom-of-set-iff* [*simp*]: *0* = *monom-of-set X* ⟷ *infinite X* ∨ *X* = {}
by *transfer* (*auto simp*: *fun-eq-iff*)

2.8 Permuting the variables of a polynomial

Next, we define the operation of permuting the variables of a monomial and polynomial.

lift-definition *permutep* :: (*'a* ⇒ *'a*) ⇒ (*'a* $\Rightarrow_0$ *'b*) ⇒ (*'a* $\Rightarrow_0$ *'b* :: *zero*) **is**
λ*f p*. *if bij f then p* ∘ *f else p*
proof −
fix *f* :: *'a* ⇒ *'a* **and** *g* :: *'a* ⇒ *'b*
assume ∗: *finite* {*x*. *g x* ≠ *0*}
show *finite* {*x*. (*if bij f then g* ∘ *f else g*) *x* ≠ *0*}
proof (*cases bij f*)
case *True*
with ∗ **have** *finite* (*f* −‘ {*x*. *g x* ≠ *0*})
by (*intro finite-vimageI*) (*auto dest*: *bij-is-inj*)
with *True* **show** *?thesis* **by** *auto*
qed (*use* ∗ **in** *auto*)

qed

lift-definition *mpoly-map-vars* :: $(nat \Rightarrow nat) \Rightarrow$ $'a$ :: *zero mpoly* $\Rightarrow$ $'a$ *mpoly* **is**
$\lambda f\ p.$ *permutep* (*permutep* f) p **.**

lemma *keys-permutep*: *bij* $f \Longrightarrow$ *keys* (*permutep* f m) $= f -`$ *keys* m
by *transfer auto*

lemma *permutep-id''* [*simp*]: *permutep id* = *id*
by *transfer'* (*auto simp*: *fun-eq-iff*)

lemma *permutep-id'''* [*simp*]: *permutep* ($\lambda x.\ x$) = *id*
by *transfer'* (*auto simp*: *fun-eq-iff*)

lemma *permutep-0* [*simp*]: *permutep* f *0* = *0*
by *transfer auto*

lemma *permutep-single*:
bij $f \Longrightarrow$ *permutep* f (*Poly-Mapping.single* a b) = *Poly-Mapping.single* (*inv-into UNIV* f a) b
by *transfer* (*auto simp*: *fun-eq-iff when-def inv-f-f surj-f-inv-f bij-is-inj bij-is-surj*)

lemma *mpoly-map-vars-id* [*simp*]: *mpoly-map-vars id* = *id*
by *transfer auto*

lemma *mpoly-map-vars-id'* [*simp*]: *mpoly-map-vars* ($\lambda x.\ x$) = *id*
by *transfer auto*

lemma *lookup-permutep*:
Poly-Mapping.lookup (*permutep* f m) x = (*if bij* f *then Poly-Mapping.lookup* m (f x) *else Poly-Mapping.lookup* m x)
by *transfer auto*

lemma *inj-permutep* [*intro*]: *inj* (*permutep* (f :: $'a \Rightarrow 'a$) :: - $\Rightarrow$ $'a \Rightarrow_0 'b$ :: *zero*)
unfolding *inj-def*
proof (*transfer*, *safe*)
fix f :: $'a \Rightarrow 'a$ **and** x y :: $'a \Rightarrow 'b$
assume *eq*: (*if bij* f *then* $x \circ f$ *else* x) = (*if bij* f *then* $y \circ f$ *else* y)
show $x = y$
proof (*cases bij* f)
case *True*
show *?thesis*
proof
fix t :: $'a$
from ‹*bij* f› **obtain** s **where** $t = f\ s$
by (*auto dest*!: *bij-is-surj*)
with *eq* **and** *True* **show** $x\ t = y\ t$
by (*auto simp*: *fun-eq-iff*)
qed

qed (*use eq* **in** *auto*)
qed

lemma *surj-permutep* [*intro*]: *surj* (*permutep* (f :: $'a \Rightarrow 'a$) :: - $\Rightarrow$ $'a \Rightarrow_0 'b$ :: *zero*)
unfolding *surj-def*
proof (*transfer*, *safe*)
fix f :: $'a \Rightarrow 'a$ **and** y :: $'a \Rightarrow 'b$
assume *fin*: *finite* $\{t.\ y\ t \neq 0\}$
show $\exists x \in \{f.\ finite\ \{x.\ f\ x \neq 0\}\}.\ y = (if\ bij\ f\ then\ x \circ f\ else\ x)$
proof (*cases bij f*)
case *True*
with *fin* **have** *finite* (*the-inv f* $-`$ $\{t.\ y\ t \neq 0\}$)
by (*intro finite-vimageI*) (*auto simp*: *bij-is-inj bij-betw-the-inv-into*)
moreover have $y \circ$ *the-inv f* $\circ f = y$
using *True* **by** (*simp add*: *fun-eq-iff the-inv-f-f bij-is-inj*)
ultimately show *?thesis* **by** (*intro bexI*[*of* - $y \circ$ *the-inv f*]) (*auto simp*: *True*)
qed (*use fin* **in** *auto*)
qed

lemma *bij-permutep* [*intro*]: *bij* (*permutep f*)
using *inj-permutep*[*of f*] *surj-permutep*[*of f*] **by** (*simp add*: *bij-def*)

lemma *mpoly-map-vars-map-mpoly*:
mpoly-map-vars f (*map-mpoly g p*) = *map-mpoly g* (*mpoly-map-vars f p*)
by (*transfer'*, *transfer'*) (*auto simp*: *fun-eq-iff*)

lemma *coeff-mpoly-map-vars*:
fixes f :: *nat* $\Rightarrow$ *nat* **and** p :: $'a$ :: *zero mpoly*
assumes *bij f*
shows *MPoly-Type.coeff* (*mpoly-map-vars f p*) *mon* =
MPoly-Type.coeff p (*permutep f mon*)
using *assms* **by** *transfer'* (*simp add*: *lookup-permutep bij-permutep*)

lemma *permutep-monom-of-set*:
assumes *bij f*
shows *permutep f* (*monom-of-set A*) = *monom-of-set* (f $-`$ A)
using *assms* **by** *transfer* (*auto simp*: *fun-eq-iff bij-is-inj finite-vimage-iff*)

lemma *permutep-comp*: *bij f* $\Longrightarrow$ *bij g* $\Longrightarrow$ *permutep* ($f \circ g$) = *permutep g* $\circ$ *permutep f*
by *transfer'* (*auto simp*: *fun-eq-iff bij-comp*)

lemma *permutep-comp'*: *bij f* $\Longrightarrow$ *bij g* $\Longrightarrow$ *permutep* ($f \circ g$) *mon* = *permutep g* (*permutep f mon*)
by *transfer* (*auto simp*: *fun-eq-iff bij-comp*)

lemma *mpoly-map-vars-comp*:
bij f $\Longrightarrow$ *bij g* $\Longrightarrow$ *mpoly-map-vars f* (*mpoly-map-vars g p*) = *mpoly-map-vars* ($f \circ g$) p

by *transfer* (*auto simp*: *bij-permutep permutep-comp*)

lemma *permutep-id* [*simp*]: *permutep id mon* = *mon*
  by *transfer auto*

lemma *permutep-id′* [*simp*]: *permutep* ($\lambda x.\ x$) *mon* = *mon*
  by *transfer auto*

lemma *inv-permutep* [*simp*]:
  fixes f :: $'a \Rightarrow 'a$
  assumes *bij f*
  shows *inv-into UNIV* (*permutep f*) = *permutep* (*inv-into UNIV f*)
proof
  fix m :: $'a \Rightarrow_0 'b$
  show *inv-into UNIV* (*permutep f*) *m* = *permutep* (*inv-into UNIV f*) *m*
    using *permutep-comp′*[*of inv-into UNIV f f m*] *assms inj-iff*[*of f*]
    by (*intro inv-f-eq*) (*auto simp*: *bij-imp-bij-inv bij-is-inj*)
qed

lemma *mpoly-map-vars-monom*:
  bij f $\Longrightarrow$ *mpoly-map-vars f* (*monom m c*) = *monom* (*permutep* (*inv-into UNIV f*) *m*) *c*
  by *transfer′* (*simp add*: *permutep-single bij-permutep*)

lemma *vars-mpoly-map-vars*:
  fixes f :: *nat* $\Rightarrow$ *nat* **and** p :: $'a$ :: *zero mpoly*
  assumes *bij f*
  shows *vars* (*mpoly-map-vars f p*) = *f* ‘ *vars p*
  using *assms* **unfolding** *vars-def*
proof *transfer′*
  fix f :: *nat* $\Rightarrow$ *nat* **and** p :: (*nat* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ $'a$
  assume *f*: *bij f*
  have *eq*: *f* (*inv-into UNIV f x*) = *x* **for** *x*
    using *f* **by** (*subst surj-f-inv-f*[*of f*]) (*auto simp*: *bij-is-surj*)
  show $\bigcup$ (*keys* ‘ *keys* (*permutep* (*permutep f*) *p*)) = *f* ‘ $\bigcup$ (*keys* ‘ *keys p*)
  proof *safe*
    fix *m x* **assume** *mx*: *m* $\in$ *keys* (*permutep* (*permutep f*) *p*) *x* $\in$ *keys m*
    from *mx* **have** *permutep f m* $\in$ *keys p*
      by (*auto simp*: *keys-permutep bij-permutep f*)
    with *mx* **have** *f* (*inv-into UNIV f x*) $\in$ *f* ‘ ($\bigcup$*m*$\in$*keys p. keys m*)
      by (*intro imageI*) (*auto intro*!: *bexI*[*of - permutep f m*] *simp*: *keys-permutep f eq*)
    with *eq* **show** *x* $\in$ *f* ‘ ($\bigcup$*m*$\in$*keys p. keys m*) **by** *simp*
  next
    fix *m x* **assume** *mx*: *m* $\in$ *keys p x* $\in$ *keys m*
    from *mx* **have** *permutep id m* $\in$ *keys p* **by** *simp*
    also have *id* = *inv-into UNIV f* $\circ$ *f* **using** *f* **by** (*intro ext*) (*auto simp*: *bij-is-inj inv-f-f*)
    also have *permutep* … *m* = *permutep f* (*permutep* (*inv-into UNIV f*) *m*)

by (*simp add*: *permutep-comp f bij-imp-bij-inv*)
finally have ∗∗: *permutep f* (*permutep* (*inv-into UNIV f*) *m*) ∈ *keys p* **.**
moreover from *f mx* **have** *f x* ∈ *keys* (*permutep* (*inv-into UNIV f*) *m*)
by (*auto simp*: *keys-permutep bij-imp-bij-inv inv-f-f bij-is-inj*)
ultimately show *f x* ∈ ⋃ (*keys* ' *keys* (*permutep* (*permutep f*) *p*)) **using** *f*
by (*auto simp*: *keys-permutep bij-permutep*)
qed
qed

lemma *permutep-eq-monom-of-set-iff* [*simp*]:
assumes *bij f*
shows *permutep f mon* = *monom-of-set A* ⟷ *mon* = *monom-of-set* (*f* ' *A*)
proof
assume *eq*: *permutep f mon* = *monom-of-set A*
have *permutep* (*inv-into UNIV f*) (*permutep f mon*) = *monom-of-set* (*inv-into UNIV f* −' *A*)
using *assms* **by** (*simp add*: *eq bij-imp-bij-inv assms permutep-monom-of-set*)
also have *inv-into UNIV f* −' *A* = *f* ' *A*
using *assms* **by** (*force simp*: *bij-is-surj image-iff inv-f-f bij-is-inj surj-f-inv-f*)
also have *permutep* (*inv-into UNIV f*) (*permutep f mon*) = *permutep* (*f* ∘ *inv-into UNIV f*) *mon*
using *assms* **by** (*simp add*: *permutep-comp bij-imp-bij-inv*)
also have *f* ∘ *inv-into UNIV f* = *id*
by (*subst surj-iff* [*symmetric*]) (*insert assms*, *auto simp*: *bij-is-surj*)
finally show *mon* = *monom-of-set* (*f* ' *A*) **by** *simp*
qed (*insert assms*, *auto simp*: *permutep-monom-of-set inj-vimage-image-eq bij-is-inj*)

lemma *permutep-monom-of-set-permutes* [*simp*]:
assumes π *permutes A*
shows *permutep* π (*monom-of-set A*) = *monom-of-set A*
using *assms*
by *transfer* (*auto simp*: *if-splits fun-eq-iff permutes-in-image*)

lemma *mpoly-map-vars-0* [*simp*]: *mpoly-map-vars f 0* = *0*
by (*transfer*, *transfer'*) (*simp add*: *o-def*)

lemma *permutep-eq-0-iff* [*simp*]: *permutep f m* = *0* ⟷ *m* = *0*
proof *transfer*
fix *f* :: *'a* ⇒ *'a* **and** *m* :: *'a* ⇒ *'b* **assume** *finite* {*x*. *m x* ≠ *0*}
show ((*if bij f then m* ∘ *f else m*) = (λ*k*. *0*)) = (*m* = (λ*k*. *0*))
proof (*cases bij f*)
case *True*
hence (∀ *x*. *m* (*f x*) = *0*) ⟷ (∀ *x*. *m x* = *0*)
using *bij-iff*[*of f*] **by** *metis*
with *True* **show** *?thesis* **by** (*auto simp*: *fun-eq-iff*)
qed (*auto simp*: *fun-eq-iff*)
qed

lemma *mpoly-map-vars-1* [*simp*]: *mpoly-map-vars f 1* = *1*

by (*transfer*, *transfer'*) (*auto simp*: *o-def fun-eq-iff when-def*)

lemma *permutep-Const*$_0$ [*simp*]: $(\bigwedge x.\ f\ x = 0 \longleftrightarrow x = 0) \Longrightarrow$ *permutep f* (*Const*$_0$ *c*) = *Const*$_0$ *c*
unfolding *Const*$_0$*-def* **by** *transfer'* (*auto simp*: *when-def fun-eq-iff*)

lemma *permutep-add* [*simp*]: *permutep f* (*m1* + *m2*) = *permutep f m1* + *permutep f m2*
unfolding *Const*$_0$*-def* **by** *transfer'* (*auto simp*: *when-def fun-eq-iff*)

lemma *permutep-diff* [*simp*]: *permutep f* (*m1* − *m2*) = *permutep f m1* − *permutep f m2*
unfolding *Const*$_0$*-def* **by** *transfer'* (*auto simp*: *when-def fun-eq-iff*)

lemma *permutep-uminus* [*simp*]: *permutep f* (−*m*) = −*permutep f m*
unfolding *Const*$_0$*-def* **by** *transfer'* (*auto simp*: *when-def fun-eq-iff*)

lemma *permutep-mult* [*simp*]:
$(\bigwedge x\ y.\ f\ (x + y) = f\ x + f\ y) \Longrightarrow$ *permutep f* (*m1* ∗ *m2*) = *permutep f m1* ∗ *permutep f m2*
unfolding *Const*$_0$*-def* **by** *transfer'* (*auto simp*: *when-def fun-eq-iff prod-fun-compose-bij*)

lemma *mpoly-map-vars-Const* [*simp*]: *mpoly-map-vars f* (*Const c*) = *Const c*
by *transfer* (*auto simp*: *o-def fun-eq-iff when-def*)

lemma *mpoly-map-vars-add* [*simp*]: *mpoly-map-vars f* (*p* + *q*) = *mpoly-map-vars f p* + *mpoly-map-vars f q*
by *transfer simp*

lemma *mpoly-map-vars-diff* [*simp*]: *mpoly-map-vars f* (*p* − *q*) = *mpoly-map-vars f p* − *mpoly-map-vars f q*
by *transfer simp*

lemma *mpoly-map-vars-uminus* [*simp*]: *mpoly-map-vars f* (−*p*) = −*mpoly-map-vars f p*
by *transfer simp*

lemma *permutep-smult*:
permutep (*permutep f*) (*Poly-Mapping.map* ((∗) *c*) *p*) =
Poly-Mapping.map ((∗) *c*) (*permutep* (*permutep f*) *p*)
by *transfer'* (*auto split*: *if-splits simp*: *fun-eq-iff*)

lemma *mpoly-map-vars-smult* [*simp*]: *mpoly-map-vars f* (*smult c p*) = *smult c* (*mpoly-map-vars f p*)
by *transfer* (*simp add*: *permutep-smult*)

lemma *mpoly-map-vars-mult* [*simp*]: *mpoly-map-vars f* (*p* ∗ *q*) = *mpoly-map-vars f p* ∗ *mpoly-map-vars f q*
by *transfer simp*

lemma *mpoly-map-vars-sum* [*simp*]: *mpoly-map-vars f* (*sum g A*) = ($\sum$ *x*∈*A*. *mpoly-map-vars f* (*g x*))
  by (*induction A rule*: *infinite-finite-induct*) *auto*

lemma *mpoly-map-vars-prod* [*simp*]: *mpoly-map-vars f* (*prod g A*) = ($\prod$ *x*∈*A*. *mpoly-map-vars f* (*g x*))
  by (*induction A rule*: *infinite-finite-induct*) *auto*

lemma *mpoly-map-vars-eq-0-iff* [*simp*]: *mpoly-map-vars f p* = *0* ⟷ *p* = *0*
  by *transfer auto*

lemma *permutep-eq-iff* [*simp*]: *permutep f p* = *permutep f q* ⟷ *p* = *q*
  by *transfer* (*auto simp*: *comp-bij-eq-iff*)

lemma *mpoly-map-vars-Sum-any* [*simp*]:
  mpoly-map-vars f (*Sum-any g*) = *Sum-any* (λ*x*. *mpoly-map-vars f* (*g x*))
  by (*simp add*: *Sum-any.expand-set*)

lemma *mpoly-map-vars-power* [*simp*]: *mpoly-map-vars f* (*p* ^ *n*) = *mpoly-map-vars f p* ^ *n*
  by (*induction n*) *auto*

lemma *mpoly-map-vars-monom-single* [*simp*]:
  assumes *bij f*
  shows *mpoly-map-vars f* (*monom* (*Poly-Mapping.single i n*) *c*) =
      monom (*Poly-Mapping.single* (*f i*) *n*) *c*
  using *assms* **by** (*simp add*: *mpoly-map-vars-monom permutep-single bij-imp-bij-inv inv-inv-eq*)

lemma *insertion-mpoly-map-vars*:
  assumes *bij f*
  shows *insertion g* (*mpoly-map-vars f p*) = *insertion* (*g* ∘ *f*) *p*
proof −
  have *insertion g* (*mpoly-map-vars f p*) =
      ($\sum$ *m*. *coeff p* (*permutep f m*) ∗ ($\prod$ *i*. *g i* ^ *lookup m i*))
    using *assms* **by** (*simp add*: *insertion-altdef coeff-mpoly-map-vars*)
  also have ... = *Sum-any* (λ*m*. *coeff p* (*permutep f m*) ∗
      Prod-any (λ*i*. *g* (*f i*) ^ *lookup m* (*f i*)))
    by (*intro Sum-any.cong arg-cong*[**where** *?f* = λ*y*. *x* ∗ *y* **for** *x*]
      Prod-any.reindex-cong[*OF assms*]) (*auto simp*: *o-def*)
  also have ... = *Sum-any* (λ*m*. *coeff p m* ∗ ($\prod$ *i*. *g* (*f i*) ^ *lookup m i*))
    by (*intro Sum-any.reindex-cong* [*OF bij-permutep*[*of f*], *symmetric*])
      (*auto simp*: *o-def lookup-permutep assms*)
  also have ... = *insertion* (*g* ∘ *f*) *p*
    by (*simp add*: *insertion-altdef*)
  finally show *?thesis* **.**
qed

lemma *permutep-cong*:
  assumes *f permutes* $(-$*keys p*$)$ *g permutes* $(-$*keys p*$)$ $p = q$
  shows *permutep f p* $=$ *permutep g q*
proof (*intro poly-mapping-eqI*)
  fix $k :: 'a$
  show *lookup* (*permutep f p*) $k =$ *lookup* (*permutep g q*) k
  proof (*cases* $k \in$ *keys p*)
    case *False*
    with *assms* **have** $f\ k \notin$ *keys p* $g\ k \notin$ *keys p*
      using *permutes-in-image*[*of* - $-$*keys p k*] **by** *auto*
    thus *?thesis* **using** *assms* **by** (*auto simp*: *lookup-permutep permutes-bij in-keys-iff*)
  qed (*insert assms*, *auto simp*: *lookup-permutep permutes-bij permutes-not-in*)
qed

lemma *mpoly-map-vars-cong*:
  assumes *f permutes* $(-$*vars p*$)$ *g permutes* $(-$*vars q*$)$ $p = q$
  shows *mpoly-map-vars f p* $=$ *mpoly-map-vars g* ($q :: 'a ::$ *zero mpoly*)
proof (*intro mpoly-eqI*)
  fix *mon* :: *nat* $\Rightarrow_0$ *nat*
  show *coeff* (*mpoly-map-vars f p*) *mon* $=$ *coeff* (*mpoly-map-vars g q*) *mon*
  proof (*cases keys mon* $\subseteq$ *vars p*)
    case *True*
    with *assms* **have** *permutep f mon* $=$ *permutep g mon*
      by (*intro permutep-cong assms*(*1*,*2*)[*THEN permutes-subset*]) *auto*
    thus *?thesis* **using** *assms* **by** (*simp add*: *coeff-mpoly-map-vars permutes-bij*)
  next
    case *False*
    hence $\neg$(*keys mon* $\subseteq f$ ‘ *vars q*) $\neg$(*keys mon* $\subseteq g$ ‘ *vars q*)
      using *assms* **by** (*auto simp*: *subset-iff permutes-not-in*)
    thus *?thesis* **using** *assms*
      by (*subst* (*1 2*) *coeff-notin-vars*)
        (*auto simp*: *coeff-notin-vars vars-mpoly-map-vars permutes-bij*)
  qed
qed

2.9 Symmetric polynomials

A polynomial is symmetric on a set of variables if it is invariant under any permutation of that set.

definition *symmetric-mpoly* :: *nat set* $\Rightarrow$ $'a$ :: *zero mpoly* $\Rightarrow$ *bool* **where**
  symmetric-mpoly A p $= (\forall \pi.\ \pi$ *permutes A* $\longrightarrow$ *mpoly-map-vars* $\pi\ p = p)$

lemma *symmetric-mpoly-empty* [*simp*, *intro*]: *symmetric-mpoly* {} *p*
  by (*simp add*: *symmetric-mpoly-def*)

A polynomial is trivially symmetric on any set of variables that do not occur in it.

lemma *symmetric-mpoly-orthogonal*:

assumes *vars p* $\cap$ *A* = {}
shows *symmetric-mpoly A p*
unfolding *symmetric-mpoly-def*
proof *safe*
fix π **assume** π: π *permutes A*
with *assms* **have** π *x* = *x* **if** *x* $\in$ *vars p* **for** *x*
using *that permutes-not-in*[*of* π *A x*] **by** *auto*
from *assms* **have** *mpoly-map-vars* π *p* = *mpoly-map-vars id p*
by (*intro mpoly-map-vars-cong permutes-subset*[*OF* π] *permutes-id*) *auto*
also have ... = *p* **by** *simp*
finally show *mpoly-map-vars* π *p* = *p* **.**
qed

lemma *symmetric-mpoly-monom* [*intro*]:
assumes *keys m* $\cap$ *A* = {}
shows *symmetric-mpoly A* (*monom m c*)
using *assms vars-monom-subset*[*of m c*] **by** (*intro symmetric-mpoly-orthogonal*) *auto*

lemma *symmetric-mpoly-subset*:
assumes *symmetric-mpoly A p B* $\subseteq$ *A*
shows *symmetric-mpoly B p*
unfolding *symmetric-mpoly-def*
proof *safe*
fix π **assume** π *permutes B*
with *assms* **have** π *permutes A* **using** *permutes-subset* **by** *blast*
with *assms* **show** *mpoly-map-vars* π *p* = *p*
by (*auto simp*: *symmetric-mpoly-def*)
qed

If a polynomial is symmetric over some set of variables, that set must either be a subset of the variables occurring in the polynomial or disjoint from it.

lemma *symmetric-mpoly-imp-orthogonal-or-subset*:
assumes *symmetric-mpoly A p*
shows *vars p* $\cap$ *A* = {} $\lor$ *A* $\subseteq$ *vars p*
proof (*rule ccontr*)
assume $\neg$(*vars p* $\cap$ *A* = {} $\lor$ *A* $\subseteq$ *vars p*)
then obtain *x y* **where** *xy*: *x* $\in$ *vars p* $\cap$ *A y* $\in$ *A* $-$ *vars p* **by** *auto*
define π **where** π = *transpose x y*
from *xy* **have** π: π *permutes A*
unfolding π-*def* **by** (*intro permutes-swap-id*) *auto*
from *xy* **have** *y* $\in$ π ‘ *vars p* **by** (*auto simp*: π-*def transpose-def*)
also from π **have** π ‘ *vars p* = *vars* (*mpoly-map-vars* π *p*)
by (*auto simp*: *vars-mpoly-map-vars permutes-bij*)
also have *mpoly-map-vars* π *p* = *p*
using *assms* π **by** (*simp add*: *symmetric-mpoly-def*)
finally show *False* **using** *xy* **by** *auto*
qed

Symmetric polynomials are closed under ring operations.

lemma *symmetric-mpoly-add* [*intro*]:
symmetric-mpoly A p $\Longrightarrow$ *symmetric-mpoly A q* $\Longrightarrow$ *symmetric-mpoly A* $(p + q)$
unfolding *symmetric-mpoly-def* **by** *simp*

lemma *symmetric-mpoly-diff* [*intro*]:
symmetric-mpoly A p $\Longrightarrow$ *symmetric-mpoly A q* $\Longrightarrow$ *symmetric-mpoly A* $(p - q)$
unfolding *symmetric-mpoly-def* **by** *simp*

lemma *symmetric-mpoly-uminus* [*intro*]: *symmetric-mpoly A p* $\Longrightarrow$ *symmetric-mpoly A* $(-p)$
unfolding *symmetric-mpoly-def* **by** *simp*

lemma *symmetric-mpoly-uminus-iff* [*simp*]: *symmetric-mpoly A* $(-p) \longleftrightarrow$ *symmetric-mpoly A p*
unfolding *symmetric-mpoly-def* **by** *simp*

lemma *symmetric-mpoly-smult* [*intro*]: *symmetric-mpoly A p* $\Longrightarrow$ *symmetric-mpoly A* (*smult c p*)
unfolding *symmetric-mpoly-def* **by** *simp*

lemma *symmetric-mpoly-mult* [*intro*]:
symmetric-mpoly A p $\Longrightarrow$ *symmetric-mpoly A q* $\Longrightarrow$ *symmetric-mpoly A* $(p * q)$
unfolding *symmetric-mpoly-def* **by** *simp*

lemma *symmetric-mpoly-0* [*simp*, *intro*]: *symmetric-mpoly A 0*
and *symmetric-mpoly-1* [*simp*, *intro*]: *symmetric-mpoly A 1*
and *symmetric-mpoly-Const* [*simp*, *intro*]: *symmetric-mpoly A* (*Const c*)
by (*simp-all add*: *symmetric-mpoly-def*)

lemma *symmetric-mpoly-power* [*intro*]:
symmetric-mpoly A p $\Longrightarrow$ *symmetric-mpoly A* $(p \mathbin{\char`\^} n)$
by (*induction n*) (*auto intro*!: *symmetric-mpoly-mult*)

lemma *symmetric-mpoly-sum* [*intro*]:
$(\bigwedge i.\ i \in B \Longrightarrow$ *symmetric-mpoly A* $(f\ i)) \Longrightarrow$ *symmetric-mpoly A* (*sum f B*)
by (*induction B rule*: *infinite-finite-induct*) (*auto intro*!: *symmetric-mpoly-add*)

lemma *symmetric-mpoly-prod* [*intro*]:
$(\bigwedge i.\ i \in B \Longrightarrow$ *symmetric-mpoly A* $(f\ i)) \Longrightarrow$ *symmetric-mpoly A* (*prod f B*)
by (*induction B rule*: *infinite-finite-induct*) (*auto intro*!: *symmetric-mpoly-mult*)

An symmetric sum or product over polynomials yields a symmetric polynomial:

lemma *symmetric-mpoly-symmetric-sum*:
assumes *g permutes X*
assumes $\bigwedge x\ \pi.\ x \in X \Longrightarrow \pi$ *permutes A* $\Longrightarrow$ *mpoly-map-vars* $\pi\ (f\ x) = f\ (g\ x)$
shows *symmetric-mpoly A* $(\sum x \in X.\ f\ x)$
unfolding *symmetric-mpoly-def*

proof *safe*
  fix π **assume** π: π *permutes A*
  have *mpoly-map-vars* π (*sum f X*) = ($\sum x{\in}X$. *mpoly-map-vars* π (*f x*))
    by *simp*
  also have ... = ($\sum x{\in}X$. *f* (*g x*))
    by (*intro sum.cong assms* π *refl*)
  also have ... = ($\sum x{\in}g\text{‘}X$. *f x*)
    using *assms* **by** (*subst sum.reindex*) (*auto simp*: *permutes-inj-on*)
  also have *g* ‘ *X* = *X*
    using *assms* **by** (*simp add*: *permutes-image*)
  finally show *mpoly-map-vars* π (*sum f X*) = *sum f X* **.**
qed

lemma *symmetric-mpoly-symmetric-prod*:
  assumes *g permutes X*
  assumes $\bigwedge x\ \pi.\ x \in X \Longrightarrow \pi$ *permutes A* $\Longrightarrow$ *mpoly-map-vars* π (*f x*) = *f* (*g x*)
  shows *symmetric-mpoly A* ($\prod x{\in}X$. *f x*)
  unfolding *symmetric-mpoly-def*
proof *safe*
  fix π **assume** π: π *permutes A*
  have *mpoly-map-vars* π (*prod f X*) = ($\prod x{\in}X$. *mpoly-map-vars* π (*f x*))
    by *simp*
  also have ... = ($\prod x{\in}X$. *f* (*g x*))
    by (*intro prod.cong assms* π *refl*)
  also have ... = ($\prod x{\in}g\text{‘}X$. *f x*)
    using *assms* **by** (*subst prod.reindex*) (*auto simp*: *permutes-inj-on*)
  also have *g* ‘ *X* = *X*
    using *assms* **by** (*simp add*: *permutes-image*)
  finally show *mpoly-map-vars* π (*prod f X*) = *prod f X* **.**
qed

If p is a polynomial that is symmetric on some subset of variables A, then for the leading monomial of p, the exponents of these variables are decreasing w. r. t. the variable ordering.

theorem *lookup-lead-monom-decreasing*:
  assumes *symmetric-mpoly A p*
  defines $m \equiv$ *lead-monom p*
  assumes $i \in A$ $j \in A$ $i \leq j$
  shows *lookup m i* $\geq$ *lookup m j*
proof (*cases p* = *0*)
  case [*simp*]: *False*
  show *?thesis*
  proof (*intro leI notI*)
    assume *less*: *lookup m i* < *lookup m j*
    define π **where** π = *transpose i j*
    from *assms* **have** π: π *permutes A*
      unfolding π*-def* **by** (*intro permutes-swap-id*) *auto*
    have [*simp*]: $\pi \circ \pi = id$ $\pi\ i = j$ $\pi\ j = i$ $\bigwedge k.\ k \neq i \Longrightarrow k \neq j \Longrightarrow \pi\ k = k$
      by (*auto simp*: π*-def Fun.swap-def fun-eq-iff*)

have $0 \neq$ *lead-coeff p* **by** *simp*
also have *lead-coeff p* = *MPoly-Type.coeff* (*mpoly-map-vars* π *p*) (*permutep* π *m*)
using π **by** (*simp add*: *lead-coeff-def m-def coeff-mpoly-map-vars permutes-bij permutep-comp'* [*symmetric*])
also have *mpoly-map-vars* π *p* = *p*
using π *assms* **by** (*simp add*: *symmetric-mpoly-def*)
finally have *permutep* π *m* $\leq$ *m* **by** (*auto simp*: *m-def*)

moreover have *lookup m i* < *lookup* (*permutep* π *m*) *i*
and ($\forall$ *k*<*i*. *lookup m k* = *lookup* (*permutep* π *m*) *k*)
using *assms* π *less* **by** (*auto simp*: *lookup-permutep permutes-bij*)
hence *m* < *permutep* π *m*
by (*auto simp*: *less-poly-mapping-def less-fun-def*)
ultimately show *False* **by** *simp*
qed
qed (*auto simp*: *m-def*)

2.10 The elementary symmetric polynomials

The k-th elementary symmetric polynomial for a finite set of variables A, with k ranging between 1 and $|A|$, is the sum of the product of all subsets of A with cardinality k:

lift-definition *sym-mpoly-aux* :: *nat set* $\Rightarrow$ *nat* $\Rightarrow$ (*nat* $\Rightarrow_0$ *nat*) $\Rightarrow_0$ *'a* :: {*zero-neq-one*} **is**
λ*X k mon*. *if finite X* $\wedge$ ($\exists$ *Y*. *Y* $\subseteq$ *X* $\wedge$ *card Y* = *k* $\wedge$ *mon* = *monom-of-set Y*) *then 1 else 0*
proof −
fix *k* :: *nat* **and** *X* :: *nat set*
show *finite* {*x*. (*if finite X* $\wedge$ ($\exists$ *Y*$\subseteq$*X*. *card Y* = *k* $\wedge$ *x* = *monom-of-set Y*) *then 1 else 0*) $\neq$
(*0*::*'a*)} (**is** *finite ?A*)
proof (*cases finite X*)
case *True*
have *?A* $\subseteq$ *monom-of-set* ‘ *Pow X* **by** *auto*
moreover from *True* **have** *finite* (*monom-of-set* ‘ *Pow X*) **by** *simp*
ultimately show *?thesis* **by** (*rule finite-subset*)
qed *auto*
qed

lemma *lookup-sym-mpoly-aux*:
Poly-Mapping.lookup (*sym-mpoly-aux X k*) *mon* =
(*if finite X* $\wedge$ ($\exists$ *Y*. *Y* $\subseteq$ *X* $\wedge$ *card Y* = *k* $\wedge$ *mon* = *monom-of-set Y*) *then 1 else 0*)
by *transfer' simp*

lemma *lookup-sym-mpoly-aux-monom-of-set* [*simp*]:
assumes *finite X Y* $\subseteq$ *X card Y* = *k*

shows *Poly-Mapping.lookup* (*sym-mpoly-aux X k*) (*monom-of-set Y*) = *1*
using *assms* **by** (*auto simp*: *lookup-sym-mpoly-aux*)

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

lift-definition *sym-mpoly* :: *nat set* ⇒ *nat* ⇒ *′a* :: {*zero-neq-one*} *mpoly* **is**
sym-mpoly-aux **.**

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

lemma *coeff-sym-mpoly*:
MPoly-Type.coeff (*sym-mpoly X k*) *mon* =
(*if finite X* ∧ (∃ *Y*. *Y* ⊆ *X* ∧ *card Y* = *k* ∧ *mon* = *monom-of-set Y*) *then 1*
else 0)
by *transfer′* (*simp add*: *lookup-sym-mpoly-aux*)

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

lemma *sym-mpoly-altdef*: *sym-mpoly A k* = (∑ *X* | *X* ⊆ *A* ∧ *card X* = *k*. *monom*
(*monom-of-set X*) *1*)
proof (*cases finite A*)
case *False*
hence ∗: *infinite* {*X*. *X* ⊆ *A* ∧ *infinite X*}
by (*rule infinite-infinite-subsets*)
have *infinite* {*X*. *X* ⊆ *A* ∧ *card X* = *0*}
by (*rule infinite-super*[*OF* - ∗]) *auto*
moreover have ∗∗: *infinite* {*X*. *X* ⊆ *A* ∧ *finite X* ∧ *card X* = *k*} **if** *k* ≠ *0*
using *that infinite-card-subsets*[*of A k*] *False* **by** *auto*
have *infinite* {*X*. *X* ⊆ *A* ∧ *card X* = *k*} **if** *k* ≠ *0*
by (*rule infinite-super*[*OF* - ∗∗[*OF that*]]) *auto*
ultimately show *?thesis* **using** *False*
by (*cases k* = *0*) (*simp-all add*: *sym-mpoly-infinite*)
next
case *True*
show *?thesis*
proof (*intro mpoly-eqI*, *goal-cases*)
case (*1 m*)
show *?case*
proof (*cases* ∃ *X*. *X* ⊆ *A* ∧ *card X* = *k* ∧ *m* = *monom-of-set X*)
case *False*
thus *?thesis* **by** (*auto simp*: *coeff-sym-mpoly coeff-sum coeff-monom*)
next
case *True*
then obtain *X* **where** *X*: *X* ⊆ *A card X* = *k m* = *monom-of-set X*
by *blast*
have *coeff* (∑ *X* | *X* ⊆ *A* ∧ *card X* = *k*.

monom (*monom-of-set X*) *1*) $m = (\sum X{\in}\{X\}.\ 1)$ **unfolding** *coeff-sum*
proof (*intro sum.mono-neutral-cong-right ballI*)
fix *Y* **assume** *Y*: $Y \in \{X.\ X \subseteq A \wedge card\ X = k\} - \{X\}$
hence $X = Y$ **if** *monom-of-set X* = *monom-of-set Y*
using *that finite-subset*[*OF X*(*1*)] *finite-subset*[*of Y A*] ‹*finite A*› **by** *auto*
thus *coeff* (*monom* (*monom-of-set Y*) *1*) *m* = *0*
using *X Y* **by** (*auto simp*: *coeff-monom when-def*)
qed (*insert X* ‹*finite A*›, *auto simp*: *coeff-monom*)
thus *?thesis* **using** ‹*finite A*› **by** (*auto simp*: *coeff-sym-mpoly coeff-sum coeff-monom*)
qed
qed
qed

lemma *coeff-sym-mpoly-monom-of-set* [*simp*]:
assumes *finite X Y* ⊆ *X card Y* = *k*
shows *MPoly-Type.coeff* (*sym-mpoly X k*) (*monom-of-set Y*) = *1*
using *assms* **by** (*auto simp*: *coeff-sym-mpoly*)

lemma *coeff-sym-mpoly-0*: *coeff* (*sym-mpoly X k*) *0* = (*if finite X* ∧ *k* = *0 then 1 else 0*)
proof −
consider *finite X k* = *0* | *finite X k* ≠ *0* | *infinite X* **by** *blast*
thus *?thesis*
proof *cases*
assume *finite X k* = *0*
hence *coeff* (*sym-mpoly X k*) (*monom-of-set* {}) = *1*
by (*subst coeff-sym-mpoly-monom-of-set*) *auto*
thus *?thesis* **unfolding** *monom-of-set-empty* **using** ‹*finite X*› ‹*k* = *0*› **by** *simp*
next
assume *finite X k* ≠ *0*
hence ¬(∃ *Y*. *finite Y* ∧ *Y* ⊆ *X* ∧ *card Y* = *k* ∧ *monom-of-set Y* = *0*)
by *auto*
thus *?thesis* **using** ‹*k* ≠ *0*›
by (*auto simp*: *coeff-sym-mpoly*)
next
assume *infinite X*
thus *?thesis* **by** (*simp add*: *coeff-sym-mpoly*)
qed
qed

lemma *symmetric-sym-mpoly* [*intro*]:
assumes *A* ⊆ *B*
shows *symmetric-mpoly A* (*sym-mpoly B k* :: *'a* :: *zero-neq-one mpoly*)
unfolding *symmetric-mpoly-def*
proof (*safe intro*!: *mpoly-eqI*)
fix *π* **and** *mon* :: *nat* $\Rightarrow_0$ *nat* **assume** *π*: *π permutes A*
from *π* **have** *π'*: *π permutes B* **by** (*rule permutes-subset*) *fact*
from *π* **have** *MPoly-Type.coeff* (*mpoly-map-vars π* (*sym-mpoly B k* :: *'a mpoly*))

mon =
MPoly-Type.coeff (*sym-mpoly B k* :: *'a mpoly*) (*permutep* π *mon*)
by (*simp add*: *coeff-mpoly-map-vars permutes-bij*)
also have ... = *1* $\longleftrightarrow$ *MPoly-Type.coeff* (*sym-mpoly B k* :: *'a mpoly*) *mon* = *1*
(**is** *?lhs* = *1* $\longleftrightarrow$ *?rhs* = *1*)
proof
assume *?rhs* = *1*
then obtain *Y* **where** *finite B* **and** *Y*: *Y* $\subseteq$ *B card Y* = *k mon* = *monom-of-set Y*
by (*auto simp*: *coeff-sym-mpoly split*: *if-splits*)
with π' **have** π $-$' *Y* $\subseteq$ *B card* (π $-$' *Y*) = *k permutep* π *mon* = *monom-of-set* (π $-$' *Y*)
by (*auto simp*: *permutes-in-image card-vimage-inj permutep-monom-of-set permutes-bij permutes-inj permutes-surj*)
thus *?lhs* = *1* **using** ‹*finite B*› **by** (*auto simp*: *coeff-sym-mpoly*)
next
assume *?lhs* = *1*
then obtain *Y* **where** *finite B* **and** *Y*: *Y* $\subseteq$ *B card Y* = *k permutep* π *mon* = *monom-of-set Y*
by (*auto simp*: *coeff-sym-mpoly split*: *if-splits*)
from *Y*(*1*) **have** *inj-on* π *Y* **using** *inj-on-subset*[*of* π *UNIV Y*] π'
by (*auto simp*: *permutes-inj*)
with *Y* π' **have** π ' *Y* $\subseteq$ *B card* (π ' *Y*) = *k mon* = *monom-of-set* (π ' *Y*)
by (*auto simp*: *permutes-in-image card-image permutep-monom-of-set permutes-bij permutes-inj permutes-surj*)
thus *?rhs* = *1* **using** ‹*finite B*› **by** (*auto simp*: *coeff-sym-mpoly*)
qed
hence *?lhs* = *?rhs*
by (*auto simp*: *coeff-sym-mpoly split*: *if-splits*)
finally show *MPoly-Type.coeff* (*mpoly-map-vars* π (*sym-mpoly B k* :: *'a mpoly*)) *mon* =
MPoly-Type.coeff (*sym-mpoly B k* :: *'a mpoly*) *mon* **.**
qed

lemma *insertion-sym-mpoly*:
assumes *finite X*
shows *insertion f* (*sym-mpoly X k*) = ($\sum$ *Y* | *Y* $\subseteq$ *X* $\wedge$ *card Y* = *k*. *prod f Y*)
using *assms*
proof (*transfer*, *transfer*)
fix *f* :: *nat* $\Rightarrow$ *'a* **and** *k* :: *nat* **and** *X* :: *nat set*
assume *X*: *finite X*
have *insertion-fun f* (λ*mon*.
if finite X $\wedge$ ($\exists$ *Y*$\subseteq$*X*. *card Y* = *k* $\wedge$ *mon* = *monom-of-set Y*) *then 1 else 0*) =
($\sum$ *m*. ($\prod$ *v*. *f v* ^ *poly-mapping.lookup m v*) *when* ($\exists$ *Y*$\subseteq$*X*. *card Y* = *k* $\wedge$ *m* = *monom-of-set Y*))
by (*auto simp add*: *insertion-fun-def X when-def intro*!: *Sum-any.cong*)
also have ... = ($\sum$ *m* | $\exists$ *Y*$\in$*Pow X*. *card Y* = *k* $\wedge$ *m* = *monom-of-set Y*. ($\prod$ *v*. *f v* ^ *poly-mapping.lookup m v*) *when* ($\exists$ *Y*$\subseteq$*X*. *card Y* = *k* $\wedge$ *m* = *monom-of-set*

Y))
  by (*rule Sum-any.expand-superset*) (*use X* **in** *auto*)
 also have ... $=$ ($\sum$ *m* | $\exists$ *Y*$\in$*Pow X*. *card Y* $=$ *k* $\wedge$ *m* $=$ *monom-of-set Y*. ($\prod$ *v*. *f v* ^ *poly-mapping.lookup m v*))
  by (*intro sum.cong*) (*auto simp*: *when-def*)
 also have ... $=$ ($\sum$ *Y* | *Y* $\subseteq$ *X* $\wedge$ *card Y* $=$ *k*. ($\prod$ *v*. *f v* ^ *poly-mapping.lookup* (*monom-of-set Y*) *v*))
  by (*rule sum.reindex-bij-witness*[*of - monom-of-set keys*]) (*auto simp*: *finite-subset*[*OF - X*])
 also have ... $=$ ($\sum$ *Y* | *Y* $\subseteq$ *X* $\wedge$ *card Y* $=$ *k*. $\prod$ *v*$\in$*Y*. *f v*)
 proof (*intro sum.cong when-cong refl*, *goal-cases*)
  case (*1 Y*)
  hence *finite Y* **by** (*auto dest*: *finite-subset*[*OF - X*])
  with *1* **have** ($\prod$ *v*. *f v* ^ *poly-mapping.lookup* (*monom-of-set Y*) *v*) $=$
         ($\prod$ *v*::*nat*. *if v* $\in$ *Y then f v else 1*)
   by (*intro Prod-any.cong*) (*auto simp*: *lookup-monom-of-set*)
  also have ... $=$ ($\prod$ *v*$\in$*Y*. *f v*)
   by (*rule Prod-any.conditionalize* [*symmetric*]) *fact*+
  finally show *?case* **.**
 qed
 finally show *insertion-fun f*
        (λ*mon*. *if finite X* $\wedge$ ($\exists$ *Y*$\subseteq$*X*. *card Y* $=$ *k* $\wedge$ *mon* $=$ *monom-of-set Y*) *then 1 else 0*) $=$
       ($\sum$ *Y* | *Y* $\subseteq$ *X* $\wedge$ *card Y* $=$ *k*. *prod f Y*) **.**
qed

lemma *sym-mpoly-nz* [*simp*]:
 assumes *finite A k* $\leq$ *card A*
 shows *sym-mpoly A k* $\neq$ (*0* :: *'a* :: *zero-neq-one mpoly*)
proof −
 from *assms* **obtain** *B* **where** *B*: *B* $\subseteq$ *A card B* $=$ *k*
  using *ex-subset-of-card* **by** *blast*
 with *assms* **have** *coeff* (*sym-mpoly A k* :: *'a mpoly*) (*monom-of-set B*) $=$ *1*
  by (*intro coeff-sym-mpoly-monom-of-set*)
 thus *?thesis* **by** *auto*
qed

lemma *coeff-sym-mpoly-0-or-1*: *coeff* (*sym-mpoly A k*) *m* $\in$ {*0*, *1*}
 by (*transfer*, *transfer*) *auto*

lemma *lead-coeff-sym-mpoly* [*simp*]:
 assumes *finite A k* $\leq$ *card A*
 shows *lead-coeff* (*sym-mpoly A k*) $=$ *1*
proof −
 from *assms* **have** *lead-coeff* (*sym-mpoly A k*) $\neq$ *0* **by** *simp*
 thus *?thesis* **using** *coeff-sym-mpoly-0-or-1*[*of A k lead-monom* (*sym-mpoly A k*)]
  unfolding *lead-coeff-def* **by** *blast*
qed

```
lemma lead-monom-sym-mpoly:
  assumes sorted xs distinct xs k ≤ length xs
  shows   lead-monom (sym-mpoly (set xs) k :: 'a :: zero-neq-one mpoly) =
            monom-of-set (set (take k xs)) (is lead-monom ?p = -)
proof −
  let ?m = lead-monom ?p
  have sym: symmetric-mpoly (set xs) (sym-mpoly (set xs) k)
    by (intro symmetric-sym-mpoly) auto
  from assms have [simp]: card (set xs) = length xs
    by (subst distinct-card) auto
  from assms have lead-coeff ?p = 1
    by (subst lead-coeff-sym-mpoly) auto
  then obtain X where X: X ⊆ set xs card X = k ?m = monom-of-set X
   unfolding lead-coeff-def by (subst (asm) coeff-sym-mpoly) (auto split: if-splits)
  define ys where ys = map (λx. if x ∈ X then 1 else 0 :: nat) xs
  have [simp]: length ys = length xs by (simp add: ys-def)

  have ys-altdef: ys = map (lookup ?m) xs
    unfolding ys-def using X finite-subset[OF X(1)]
    by (intro map-cong) (auto simp: lookup-monom-of-set)

  define i where i = Min (insert (length xs) {i. i < length xs ∧ ys ! i = 0})
  have i ≤ length xs by (auto simp: i-def)
  have in-X: xs ! j ∈ X if j < i for j
    using that unfolding i-def by (auto simp: ys-def)
  have not-in-X: xs ! j ∉ X if i ≤ j j < length xs for j
  proof −
    have ne: {i. i < length xs ∧ ys ! i = 0} ≠ {}
    proof
      assume [simp]: {i. i < length xs ∧ ys ! i = 0} = {}
      from that show False by (simp add: i-def)
    qed
    hence Min {i. i < length xs ∧ ys ! i = 0} ∈ {i. i < length xs ∧ ys ! i = 0}
      using that by (intro Min-in) auto
    also have Min {i. i < length xs ∧ ys ! i = 0} = i
      unfolding i-def using ne by (subst Min-insert) (auto simp: min-def)
    finally have i: ys ! i = 0 i < length xs by simp-all

    have lookup ?m (xs ! j) ≤ lookup ?m (xs ! i) using that assms
      by (intro lookup-lead-monom-decreasing[OF sym])
         (auto intro!: sorted-nth-mono simp: set-conv-nth)
    also have ... = 0 using i by (simp add: ys-altdef)
     finally show ?thesis using that X finite-subset[OF X(1)] by (auto simp:
lookup-monom-of-set)
  qed

  from X have k = card X
    by simp
  also have X = (λi. xs ! i) ` {i. i < length xs ∧ xs ! i ∈ X}
```

```
    using X by (auto simp: set-conv-nth)
  also have card ... = (∑ i | i < length xs ∧ xs ! i ∈ X. 1)
    using assms by (subst card-image) (auto intro!: inj-on-nth)
  also have ... = (∑ i | i < length xs. if xs ! i ∈ X then 1 else 0)
    by (intro sum.mono-neutral-cong-left) auto
  also have ... = sum-list ys
    by (auto simp: sum-list-sum-nth ys-def intro!: sum.cong)
  also have ys = take i ys @ drop i ys by simp
  also have sum-list ... = sum-list (take i ys) + sum-list (drop i ys)
    by (subst sum-list-append) auto
  also have take i ys = replicate i 1 using ‹i ≤ length xs› in-X
    by (intro replicate-eqI) (auto simp: ys-def set-conv-nth)
  also have sum-list ... = i by simp
  also have drop i ys = replicate (length ys − i) 0 using ‹i ≤ length xs› not-in-X
    by (intro replicate-eqI) (auto simp: ys-def set-conv-nth)
  also have sum-list ... = 0 by simp
  finally have i = k by simp

  have X = set (filter (λx. x ∈ X) xs)
    using X by auto
  also have xs = take i xs @ drop i xs by simp
  also note filter-append
  also have filter (λx. x ∈ X) (take i xs) = take i xs
    using in-X by (intro filter-True) (auto simp: set-conv-nth)
  also have filter (λx. x ∈ X) (drop i xs) = []
    using not-in-X by (intro filter-False) (auto simp: set-conv-nth)
  finally have X = set (take i xs) by simp
  with ‹i = k› and X show ?thesis by simp
qed
```

2.11 Induction on the leading monomial

We show that the monomial ordering for a fixed set of variables is well-founded, so we can perform induction on the leading monomial of a polynomial.

```
definition monom-less-on where
  monom-less-on A = {(m1, m2). m1 < m2 ∧ keys m1 ⊆ A ∧ keys m2 ⊆ A}

lemma wf-monom-less-on:
  assumes finite A
  shows   wf (monom-less-on A :: ((nat ⇒₀ 'b :: {zero, wellorder}) × -) set)
proof (rule wf-subset)
  define n where n = Suc (Max (insert 0 A))
  have less-n: k < n if k ∈ A for k
    using that assms by (auto simp: n-def less-Suc-eq-le Max-ge-iff)

  define f :: (nat ⇒₀ 'b) ⇒ 'b list where f = (λm. map (lookup m) [0..<n])

  show wf (inv-image (lexn {(x,y). x < y} n) f)
```

by (*intro wf-inv-image wf-lexn wellorder-class.wf*)
show *monom-less-on A* ⊆ *inv-image* (*lexn* {(*x*, *y*). *x* < *y*} *n*) *f*
proof *safe*
fix *m1 m2* :: *nat* $\Rightarrow_0$ *'b* **assume** (*m1*, *m2*) ∈ *monom-less-on A*
hence *m12*: *m1* < *m2 keys m1* ⊆ *A keys m2* ⊆ *A*
by (*auto simp*: *monom-less-on-def*)
then obtain *k* **where** *k*: *lookup m1 k* < *lookup m2 k* ∀ *i*<*k*. *lookup m1 i* = *lookup m2 i*
by (*auto simp*: *less-poly-mapping-def less-fun-def*)
have ¬(*lookup m1 k* = *0* ∧ *lookup m2 k* = *0*)
proof (*intro notI*)
assume *lookup m1 k* = *0* ∧ *lookup m2 k* = *0*
hence [*simp*]: *lookup m1 k* = *0 lookup m2 k* = *0* **by** *blast+*
from *k*(*1*) **show** *False* **by** *simp*
qed
hence *k* ∈ *A* **using** *m12* **by** (*auto simp*: *in-keys-iff*)
hence *k* < *n* **by** (*simp add*: *less-n*)

define *as* **where** *as* = *map* (*lookup m1*) [*0*..<*k*]
define *bs1* **where** *bs1* = *map* (*lookup m1*) [*Suc k*..<*n*]
define *bs2* **where** *bs2* = *map* (*lookup m2*) [*Suc k*..<*n*]
have *decomp*: [*0*..<*n*] = [*0*..<*k*] @ [*k*] @ *drop* (*Suc k*) [*0*..<*n*]
using ‹*k* < *n*› **by** (*simp flip*: *upt-conv-Cons upt-add-eq-append'*)
have [*simp*]: *length as* = *k length bs1* = *n* − *Suc k length bs2* = *n* − *Suc k*
by (*simp-all add*: *as-def bs1-def bs2-def*)

have *f m1* = *as* @ [*lookup m1 k*] @ *bs1* **unfolding** *f-def*
by (*subst decomp*) (*simp add*: *as-def bs1-def*)
moreover have *f m2* = *as* @ [*lookup m2 k*] @ *bs2* **unfolding** *f-def*
using *k* **by** (*subst decomp*) (*simp add*: *as-def bs2-def*)
ultimately show (*m1*, *m2*) ∈ *inv-image* (*lexn* {(*x*,*y*). *x* < *y*} *n*) *f*
using *k*(*1*) ‹*k* < *n*› **unfolding** *lexn-conv* **by** *fastforce*
qed
qed

lemma *lead-monom-induct* [*consumes 2*, *case-names less*]:
fixes *p* :: *'a* :: *zero mpoly*
assumes *fin*: *finite A* **and** *vars*: *vars p* ⊆ *A*
assumes *IH*: ⋀*p*. *vars p* ⊆ *A* ⟹
(⋀*p'*. *vars p'* ⊆ *A* ⟹ *lead-monom p'* < *lead-monom p* ⟹ *P p'*)
⟹ *P p*
shows *P p*
using *assms*(*2*)
proof (*induct m* ≡ *lead-monom p arbitrary*: *p rule*: *wf-induct-rule*[*OF wf-monom-less-on*[*OF fin*]])
case (*1 p*)
show *?case*
proof (*rule IH*)
fix *p'* :: *'a mpoly* **assume** ∗: *vars p'* ⊆ *A lead-monom p'* < *lead-monom p*

show $P\ p'$
by (*rule 1*) (*insert* ∗ *1.prems keys-lead-monom-subset*, *auto simp*: *monom-less-on-def*)
qed (*insert 1*, *auto*)
qed

lemma *lead-monom-induct′* [*case-names less*]:
fixes p :: $'a$ :: *zero mpoly*
assumes *IH*: $\bigwedge p.\ (\bigwedge p'.\ vars\ p' \subseteq vars\ p \Longrightarrow lead\text{-}monom\ p' < lead\text{-}monom\ p \Longrightarrow P\ p') \Longrightarrow P\ p$
shows $P\ p$
proof −
have *finite* (*vars* p) *vars* $p \subseteq$ *vars* p **by** (*auto simp*: *vars-finite*)
thus *?thesis*
by (*induction rule*: *lead-monom-induct*) (*use IH* **in** *blast*)
qed

2.12 The fundamental theorem of symmetric polynomials

lemma *lead-coeff-sym-mpoly-powerprod*:
assumes *finite A* $\bigwedge x.\ x \in X \Longrightarrow f\ x \in \{1..card\ A\}$
shows *lead-coeff* $(\prod x{\in}X.\ sym\text{-}mpoly\ A\ (f\ (x{::}'a))\ \widehat{}\ g\ x) = 1$
proof −
have *eq*: *lead-coeff* $(sym\text{-}mpoly\ A\ (f\ x)\ \widehat{}\ g\ x :: 'b\ mpoly) = 1$ **if** $x \in X$ **for** x
using *that assms* **by** (*subst lead-coeff-power*) (*auto simp*: *lead-coeff-sym-mpoly assms*)
hence $(\prod x{\in}X.\ lead\text{-}coeff\ (sym\text{-}mpoly\ A\ (f\ x)\ \widehat{}\ g\ x :: 'b\ mpoly)) = (\prod x{\in}X.\ 1)$
by (*intro prod.cong eq refl*)
also have $\ldots = 1$ **by** *simp*
finally have *eq′*: $(\prod x{\in}X.\ lead\text{-}coeff\ (sym\text{-}mpoly\ A\ (f\ x)\ \widehat{}\ g\ x :: 'b\ mpoly)) = 1$
.
show *?thesis* **by** (*subst lead-coeff-prod*) (*auto simp*: *eq eq′*)
qed

context
fixes A :: *nat set* **and** *xs n f* **and** *decr* :: $'a$ :: *comm-ring-1 mpoly* $\Rightarrow$ *bool*
defines $xs \equiv$ *sorted-list-of-set A*
defines $n \equiv$ *card A*
defines $f \equiv (\lambda i.\ if\ i < n\ then\ xs\ !\ i\ else\ 0)$
defines $decr \equiv (\lambda p.\ \forall i{\in}A.\ \forall j{\in}A.\ i \le j \longrightarrow$
$lookup\ (lead\text{-}monom\ p)\ i \ge lookup\ (lead\text{-}monom\ p)\ j)$
begin

The computation of the witness for the fundamental theorem works like this: Given some polynomial p (that is assumed to be symmetric in the variables in A), we inspect its leading monomial, which is of the form $cX_1^{i_1} \ldots X_n i_n$ where the $A = \{X_1, \ldots, X_n\}$, c contains only variables not in A, and the sequence i_j is decreasing. The latter holds because p is symmetric.

Now, we form the polynomial $q := ce_1^{i_1-i_2} e_2^{i_2-i_3} \ldots e_n^{i_n}$, which has the same leading term as p. Then $p - q$ has a smaller leading monomial, so by induction, we can assume it to be of the required form and obtain a witness for $p - q$.

Now, we only need to add $cY_1^{i_1-i_2} \dots Y_n^{i_n}$ to that witness and we obtain a witness for p.

definition *fund-sym-step-coeff* :: *'a mpoly* ⇒ *'a mpoly* **where**
fund-sym-step-coeff p = *monom* (*restrictpm* (−*A*) (*lead-monom p*)) (*lead-coeff p*)

definition *fund-sym-step-monom* :: *'a mpoly* ⇒ (*nat* ⇒$_0$ *nat*) **where**
fund-sym-step-monom p = (
let g = (λ*i*. *if i* < *n then lookup* (*lead-monom p*) (*f i*) *else 0*)
in (∑ *i*<*n*. *Poly-Mapping.single* (*Suc i*) (*g i* − *g* (*Suc i*))))

definition *fund-sym-step-poly* :: *'a mpoly* ⇒ *'a mpoly* **where**
fund-sym-step-poly p = (
let g = (λ*i*. *if i* < *n then lookup* (*lead-monom p*) (*f i*) *else 0*)
in fund-sym-step-coeff p ∗ (∏ *i*<*n*. *sym-mpoly A* (*Suc i*) ^ (*g i* − *g* (*Suc i*))))

The following function computes the witness, with the convention that it returns a constant polynomial if the input was not symmetric:

function (*domintros*) *fund-sym-poly-wit* :: *'a* :: *comm-ring-1 mpoly* ⇒ *'a mpoly mpoly* **where**
fund-sym-poly-wit p =
(*if* ¬*symmetric-mpoly A p* ∨ *lead-monom p* = *0* ∨ *vars p* ∩ *A* = {} *then Const p else*
fund-sym-poly-wit (*p* − *fund-sym-step-poly p*) +
monom (*fund-sym-step-monom p*) (*fund-sym-step-coeff p*))
by *auto*

lemma *coeff-fund-sym-step-coeff*: *coeff* (*fund-sym-step-coeff p*) *m* ∈ {*lead-coeff p*, *0*}
by (*auto simp*: *fund-sym-step-coeff-def coeff-monom when-def*)

lemma *vars-fund-sym-step-coeff*: *vars* (*fund-sym-step-coeff p*) ⊆ *vars p* − *A*
unfolding *fund-sym-step-coeff-def* **using** *keys-lead-monom-subset*[*of p*]
by (*intro order.trans*[*OF vars-monom-subset*]) *auto*

lemma *keys-fund-sym-step-monom*: *keys* (*fund-sym-step-monom p*) ⊆ {*1..n*}
unfolding *fund-sym-step-monom-def Let-def*
by (*intro order.trans*[*OF keys-sum*] *UN-least*, *subst keys-single*) *auto*

lemma *coeff-fund-sym-step-poly*:
assumes *C*: ∀ *m*. *coeff p m* ∈ *C* **and** *ring-closed C*
shows *coeff* (*fund-sym-step-poly p*) *m* ∈ *C*
proof −
interpret *ring-closed C* **by** *fact*
have ∗: ⋀*m*. *coeff* (*p* ^ *x*) *m* ∈ *C* **if** ⋀*m*. *coeff p m* ∈ *C* **for** *p x*
using *that* **by** (*induction x*)
(*auto simp*: *coeff-mpoly-times mpoly-coeff-1 intro*!: *prod-fun-closed*)

have $**$: $\bigwedge m.$ *coeff* (*prod f X*) $m \in C$ **if** $\bigwedge i\ m.\ i \in X \Longrightarrow$ *coeff* (f i) $m \in C$
for X **and** f :: *nat* $\Rightarrow$ -
using *that* **by** (*induction X rule*: *infinite-finite-induct*)
(*auto simp*: *coeff-mpoly-times mpoly-coeff-1 intro*!: *prod-fun-closed*)
show *?thesis* **using** C
unfolding *fund-sym-step-poly-def Let-def fund-sym-step-coeff-def coeff-mpoly-times*
by (*intro prod-fun-closed*)
(*auto simp*: *coeff-monom when-def lead-coeff-def coeff-sym-mpoly intro*!: $*$ $**$)
qed

We now show various relevant properties of the subtracted polynomial:

1. Its leading term is the same as that of the input polynomial.
2. It contains now new variables.
3. It is symmetric in the variables in A.

lemma *fund-sym-step-poly*:
shows *finite* $A \Longrightarrow p \neq 0 \Longrightarrow$ *decr* $p \Longrightarrow$ *lead-monom* (*fund-sym-step-poly* p) $=$ *lead-monom* p
and *finite* $A \Longrightarrow p \neq 0 \Longrightarrow$ *decr* $p \Longrightarrow$ *lead-coeff* (*fund-sym-step-poly* p) $=$ *lead-coeff* p
and *finite* $A \Longrightarrow p \neq 0 \Longrightarrow$ *decr* $p \Longrightarrow$ *fund-sym-step-poly* p $=$
fund-sym-step-coeff p $*$ ($\prod x.$ *sym-mpoly* A x $\hat{}$ *lookup* (*fund-sym-step-monom* p) x)
and *vars* (*fund-sym-step-poly* p) $\subseteq$ *vars* $p \cup A$
and *symmetric-mpoly* A (*fund-sym-step-poly* p)
proof $-$
define g **where** $g = (\lambda i.$ *if* $i < n$ *then lookup* (*lead-monom* p) (f i) *else 0*)
define q **where** $q = (\prod i{<}n.$ *sym-mpoly* A (*Suc* i) $\hat{}$ (g i $-$ g (*Suc* i)) :: *'a mpoly*)
define c **where** $c =$ *monom* (*restrictpm* ($-A$) (*lead-monom* p)) (*lead-coeff* p)
have [*simp*]: *fund-sym-step-poly* $p = c * q$
by (*simp add*: *fund-sym-step-poly-def fund-sym-step-coeff-def c-def q-def f-def g-def*)

have *vars* ($c * q$) $\subseteq$ *vars* $p \cup A$
using *keys-lead-monom-subset*[*of* p]
vars-monom-subset[*of restrictpm* ($-A$) (*lead-monom* p) *lead-coeff* p]
unfolding *c-def q-def*
by (*intro order.trans*[*OF vars-mult*] *order.trans*[*OF vars-prod*] *order.trans*[*OF vars-power*]
Un-least UN-least order.trans[*OF vars-sym-mpoly-subset*]) *auto*
thus *vars* (*fund-sym-step-poly* p) $\subseteq$ *vars* $p \cup A$
by *simp*
have *symmetric-mpoly* A ($c * q$) **unfolding** *c-def q-def*
by (*intro symmetric-mpoly-mult symmetric-mpoly-monom symmetric-mpoly-prod*
symmetric-mpoly-power symmetric-sym-mpoly) *auto*

thus *symmetric-mpoly A* (*fund-sym-step-poly p*) **by** *simp*

assume *finite*: *finite A* **and** [*simp*]: $p \neq 0$ **and** *decr p*
have *set xs* = *A distinct xs* **and** [*simp*]: *length xs* = *n*
using *finite* **by** (*auto simp*: *xs-def n-def*)
have [*simp*]: *lead-coeff c* = *lead-coeff p lead-monom c* = *restrictpm* (− *A*) (*lead-monom p*)
by (*simp-all add*: *c-def lead-monom-monom*)
hence *f-range* [*simp*]: $f\ i \in A$ **if** $i < n$ **for** *i*
using *that* ‹*set xs* = *A*› **by** (*auto simp*: *f-def set-conv-nth*)
have *sorted xs* **by** (*simp add*: *xs-def*)
hence *f-mono*: $f\ i \leq f\ j$ **if** $i \leq j\ j < n$ **for** *i j* **using** *that*
by (*auto simp*: *f-def n-def intro*: *sorted-nth-mono*)
hence *g-mono*: $g\ i \geq g\ j$ **if** $i \leq j$ **for** *i j*
unfolding *g-def* **using** *that* **using** ‹*decr p*› **by** (*auto simp*: *decr-def*)

have ∗: ($\prod$ *i*<*n*. *lead-coeff* (*sym-mpoly A* (*Suc i*) ^ (*g i* − *g* (*Suc i*)) :: ′*a mpoly*)) =
($\prod$ *i*<*card A*. *1*)
using ‹*finite A*› **by** (*intro prod.cong*) (*auto simp*: *n-def lead-coeff-power*)
hence *lead-coeff q* = ($\prod$ *i*<*n*. *lead-coeff* (*sym-mpoly A* (*Suc i*) ^ (*g i* − *g* (*Suc i*)) :: ′*a mpoly*))
by (*simp add*: *lead-coeff-prod lead-coeff-power n-def q-def*)
also have ... = ($\prod$ *i*<*n*. *1*)
using ‹*finite A*› **by** (*intro prod.cong*) (*auto simp*: *lead-coeff-power n-def*)
finally have [*simp*]: *lead-coeff q* = *1* **by** *simp*

have *lead-monom q* = ($\sum$ *i*<*n*. *lead-monom* (*sym-mpoly A* (*Suc i*) ^ (*g i* − *g* (*Suc i*)) :: ′*a mpoly*))
using ∗ **by** (*simp add*: *q-def lead-monom-prod lead-coeff-power n-def*)
also have ... = ($\sum$ *i*<*n*. *of-nat* (*g i* − *g* (*Suc i*)) ∗ *lead-monom* (*sym-mpoly A* (*Suc i*) :: ′*a mpoly*))
using ‹*finite A*› **by** (*intro sum.cong*) (*auto simp*: *lead-monom-power n-def*)
also have ... = ($\sum$ *i*<*n*. *of-nat* (*g i* − *g* (*Suc i*)) ∗ *monom-of-set* (*set* (*take* (*Suc i*) *xs*)))
proof (*intro sum.cong refl*, *goal-cases*)
case (*1 i*)
have *lead-monom* (*sym-mpoly A* (*Suc i*) :: ′*a mpoly*) =
lead-monom (*sym-mpoly* (*set xs*) (*Suc i*) :: ′*a mpoly*)
by (*simp add*: ‹*set xs* = *A*›)
also from *1* **have** ... = *monom-of-set* (*set* (*take* (*Suc i*) *xs*))
by (*subst lead-monom-sym-mpoly*) (*auto simp*: *xs-def n-def*)
finally show *?case* **by** *simp*
qed
finally have *lead-monom-q*:
lead-monom q = ($\sum$ *i*<*n*. *of-nat* (*g i* − *g* (*Suc i*)) ∗ *monom-of-set* (*set* (*take* (*Suc i*) *xs*))) **.**

have *lead-monom* (*c* ∗ *q*) = *lead-monom c* + *lead-monom q*

by (*simp add*: *lead-monom-mult*)
also have ... = *lead-monom p* (**is** *?S* = -)
proof (*intro poly-mapping-eqI*)
fix *i* :: *nat*
show *lookup* (*lead-monom c* + *lead-monom q*) *i* = *lookup* (*lead-monom p*) *i*
proof (*cases* $i \in A$)
case *False*
hence *lookup* (*lead-monom c* + *lead-monom q*) *i* = *lookup* (*lead-monom p*) *i* +
($\sum j<n.$ (*g j* − *g* (*Suc j*)) ∗ *lookup* (*monom-of-set* (*set* (*take* (*Suc j*) *xs*))) *i*)
(**is** - = - + *?S*) **by** (*simp add*: *lookup-add lead-monom-q lookup-sum*)
also from *False* **have** *?S* = *0*
by (*intro sum.neutral*) (*auto simp*: *lookup-monom-of-set* ‹*set xs* = *A*› *dest*!: *in-set-takeD*)
finally show *?thesis* **by** *simp*
next
case *True*
with ‹*set xs* = *A*› **obtain** *m* **where** *m*: *i* = *xs* ! *m m* < *n*
by (*auto simp*: *set-conv-nth*)
have *lookup* (*lead-monom c* + *lead-monom q*) *i* =
($\sum j<n.$ (*g j* − *g* (*Suc j*)) ∗ *lookup* (*monom-of-set* (*set* (*take* (*Suc j*) *xs*))) *i*)
using *True* **by** (*simp add*: *lookup-add lookup-sum lead-monom-q*)
also have ... = ($\sum j \mid j < n \wedge i \in$ *set* (*take* (*Suc j*) *xs*). *g j* − *g* (*Suc j*))
by (*intro sum.mono-neutral-cong-right*) *auto*
also have $\{j.\ j < n \wedge i \in$ *set* (*take* (*Suc j*) *xs*)$\}$ = $\{m..<n\}$
using *m* ‹*distinct xs*› **by** (*force simp*: *set-conv-nth nth-eq-iff-index-eq*)
also have ($\sum j\in\ldots$ *g j* − *g* (*Suc j*)) = ($\sum j\in\ldots$ *g j*) − ($\sum j\in\ldots$ *g* (*Suc j*))
by (*subst sum-subtractf-nat*) (*auto intro*!: *g-mono*)
also have ($\sum j\in\{m..<n\}.$ *g* (*Suc j*)) = ($\sum j\in\{m<..n\}.$ *g j*)
by (*intro sum.reindex-bij-witness*[*of* - $\lambda j.\ j - 1$ *Suc*]) *auto*
also have ... = ($\sum j\in\{m<..<n\}.$ *g j*)
by (*intro sum.mono-neutral-right*) (*auto simp*: *g-def*)
also have ($\sum j\in\{m..<n\}.$ *g j*) − ... = ($\sum j\in\{m..<n\}-\{m<..<n\}.$ *g j*)
by (*intro sum-diff-nat* [*symmetric*]) *auto*
also have $\{m..<n\}-\{m<..<n\} = \{m\}$ **using** *m* **by** *auto*
also have ($\sum j\in\ldots$ *g j*) = *lookup* (*lead-monom p*) *i*
using *m* **by** (*auto simp*: *g-def not-less le-Suc-eq f-def*)
finally show *?thesis* **.**
qed
qed
finally show *lead-monom* (*fund-sym-step-poly p*) = *lead-monom p* **by** *simp*
show *lead-coeff* (*fund-sym-step-poly p*) = *lead-coeff p*
by (*simp add*: *lead-coeff-mult*)

have ∗: *lookup* (*fund-sym-step-monom p*) *k* = (*if* $k \in \{1..n\}$ *then g* (*k* − *1*) − *g k else 0*) **for** *k*

proof −
have *lookup* (*fund-sym-step-monom p*) *k* =
($\sum x \in$(*if* $k \in \{1..n\}$ *then* $\{k - 1\}$ *else* $\{\}$). g $(k - 1) - g$ k)
unfolding *fund-sym-step-monom-def lookup-sum Let-def*
by (*intro sum.mono-neutral-cong-right*)
(*auto simp*: *g-def lookup-single when-def split*: *if-splits*)
thus *?thesis* **by** *simp*
qed
hence ($\prod x$. *sym-mpoly A x* ^ *lookup* (*fund-sym-step-monom p*) *x* :: *'a mpoly*) =
($\prod x \in \{1..n\}$. *sym-mpoly A x* ^ *lookup* (*fund-sym-step-monom p*) *x*)
by (*intro Prod-any.expand-superset*) *auto*
also have ... = ($\prod x < n$. *sym-mpoly A* (*Suc x*) ^ *lookup* (*fund-sym-step-monom p*) (*Suc x*))
by (*intro prod.reindex-bij-witness*[*of* - *Suc* λi. $i - 1$]) *auto*
also have ... = *q*
unfolding *q-def* **by** (*intro prod.cong*) (*auto simp*: ∗)
finally show *fund-sym-step-poly p* =
fund-sym-step-coeff p ∗ ($\prod x$. *sym-mpoly A x* ^ *lookup* (*fund-sym-step-monom p*) *x*)
by (*simp add*: *c-def q-def f-def g-def fund-sym-step-monom-def fund-sym-step-coeff-def*)
qed

If the input is well-formed, a single step of the procedure always decreases the leading monomial.

lemma *lead-monom-fund-sym-step-poly-less*:
assumes *finite A* **and** *lead-monom p* ≠ *0* **and** *decr p*
shows *lead-monom* (*p* − *fund-sym-step-poly p*) < *lead-monom p*
proof (*cases p* = *fund-sym-step-poly p*)
case *True*
thus *?thesis* **using** *assms* **by** (*auto simp*: *order.strict-iff-order*)
next
case *False*
from *assms* **have** [*simp*]: *p* ≠ *0* **by** *auto*
let *?q* = *fund-sym-step-poly p* **and** *?m* = *lead-monom p*
have *coeff* (*p* − *?q*) *?m* = *0*
using *fund-sym-step-poly*[*of p*] *assms* **by** (*simp add*: *lead-coeff-def*)
moreover have *lead-coeff* (*p* − *?q*) ≠ *0* **using** *False* **by** *auto*
ultimately have *lead-monom* (*p* − *?q*) ≠ *?m*
unfolding *lead-coeff-def* **by** *auto*
moreover have *lead-monom* (*p* − *?q*) ≤ *?m*
using *fund-sym-step-poly*[*of p*] *assms*
by (*intro order.trans*[*OF lead-monom-diff*] *max.boundedI*) *auto*
ultimately show *?thesis* **by** (*auto simp*: *order.strict-iff-order*)
qed

Finally, we prove that the witness is indeed well-defined for all inputs.

lemma *fund-sym-poly-wit-dom-aux*:
assumes *finite B vars p* ⊆ *B A* ⊆ *B*
shows *fund-sym-poly-wit-dom p*

using *assms*(*1*−*3*)
proof (*induction p rule*: *lead-monom-induct*)
case (*less p*)
have [*simp*]: *finite A* **by** (*rule finite-subset*[*of* - *B*]) *fact+*
show *?case*
proof (*cases lead-monom p* = *0* ∨ ¬*symmetric-mpoly A p*)
case *False*
hence [*simp*]: *p* ≠ *0* **by** *auto*
note *decr* = *lookup-lead-monom-decreasing*[*of A p*]
have *vars* (*p* − *fund-sym-step-poly p*) ⊆ *B*
using *fund-sym-step-poly*[*of p*] *decr False less.prems less.hyps* ‹*A* ⊆ *B*›
by (*intro order.trans*[*OF vars-diff*]) *auto*
hence *fund-sym-poly-wit-dom* (*p* − *local.fund-sym-step-poly p*)
using *False less.prems less.hyps decr*
by (*intro less.IH fund-sym-step-poly symmetric-mpoly-diff lead-monom-fund-sym-step-poly-less*) (*auto simp*: *decr-def*)
thus *?thesis* **using** *fund-sym-poly-wit.domintros* **by** *blast*
qed (*auto intro*: *fund-sym-poly-wit.domintros*)
qed

lemma *fund-sym-poly-wit-dom* [*intro*]: *fund-sym-poly-wit-dom p*
proof −
consider ¬*symmetric-mpoly A p* | *vars p* ∩ *A* = {} | *symmetric-mpoly A p A* ⊆ *vars p*
using *symmetric-mpoly-imp-orthogonal-or-subset*[*of A p*] **by** *blast*
thus *?thesis*
proof *cases*
assume *symmetric-mpoly A p A* ⊆ *vars p*
thus *?thesis* **using** *fund-sym-poly-wit-dom-aux*[*of vars p p*] **by** (*auto simp*: *vars-finite*)
qed (*auto intro*: *fund-sym-poly-wit.domintros*)
qed

termination *fund-sym-poly-wit*
by (*intro allI fund-sym-poly-wit-dom*)

Next, we prove that our witness indeed fulfils all the properties stated by the fundamental theorem:

1. If the original polynomial was in $R[X_1, \ldots, X_n, \ldots, X_m]$ where the X_1 to X_n are the symmetric variables, then the witness is a polynomial in $R[X_{n+1}, \ldots, X_m][Y_1, \ldots, Y_n]$. This means that its coefficients are polynomials in the variables of the original polynomial, minus the symmetric ones, and the (new and independent) variables of the witness polynomial range from 1 to n.

2. Substituting the i-th symmetric polynomial $e_i(X_1, \ldots, X_n)$ for the Y_i variable for every i yields the original polynomial.

3. The coefficient ring R need not be the entire type; if the coefficients of the original polynomial are in some subring, then the coefficients of the coefficients of the witness also do.

lemma *fund-sym-poly-wit-coeffs-aux*:
  assumes *finite B vars p* ⊆ *B symmetric-mpoly A p A* ⊆ *B*
  shows *vars* (*coeff* (*fund-sym-poly-wit p*) *m*) ⊆ *B* − *A*
  using *assms*
proof (*induction p rule*: *fund-sym-poly-wit.induct*)
  case (*1 p*)
  show *?case*
  proof (*cases lead-monom p* = *0* ∨ *vars p* ∩ *A* = {})
    case *False*
    have *vars* (*p* − *fund-sym-step-poly p*) ⊆ *B*
      using *1.prems fund-sym-step-poly*[*of p*] **by** (*intro order.trans*[*OF vars-diff*]) *auto*
    with *1 False* **have** *vars* (*coeff* (*fund-sym-poly-wit* (*p* − *fund-sym-step-poly p*)) *m*) ⊆ *B* − *A*
      by (*intro 1 symmetric-mpoly-diff fund-sym-step-poly*) *auto*
    hence *vars* (*coeff* (*fund-sym-poly-wit* (*p* − *fund-sym-step-poly p*) +
              monom (*fund-sym-step-monom p*) (*fund-sym-step-coeff p*)) *m*) ⊆ *B* − *A*
     unfolding *coeff-add coeff-monom* **using** *vars-fund-sym-step-coeff*[*of p*] *1.prems*
     by (*intro order.trans*[*OF vars-add*] *Un-least order.trans*[*OF vars-monom-subset*])
        (*auto simp*: *when-def*)
    thus *?thesis* **using** *1.prems False* **unfolding** *fund-sym-poly-wit.simps*[*of p*] **by** *simp*
  qed (*insert 1.prems*,
      auto simp: *fund-sym-poly-wit.simps*[*of p*] *mpoly-coeff-Const lead-monom-eq-0-iff*)
qed

lemma *fund-sym-poly-wit-coeffs*:
  assumes *symmetric-mpoly A p*
  shows *vars* (*coeff* (*fund-sym-poly-wit p*) *m*) ⊆ *vars p* − *A*
proof (*cases A* ⊆ *vars p*)
  case *True*
  with *fund-sym-poly-wit-coeffs-aux*[*of vars p p m*] *assms*
    show *?thesis* **by** (*auto simp*: *vars-finite*)
next
  case *False*
  hence *vars p* ∩ *A* = {}
    using *symmetric-mpoly-imp-orthogonal-or-subset*[*OF assms*] **by** *auto*
  thus *?thesis* **by** (*auto simp*: *fund-sym-poly-wit.simps*[*of p*] *mpoly-coeff-Const*)
qed

lemma *fund-sym-poly-wit-vars*: *vars* (*fund-sym-poly-wit p*) ⊆ {*1..n*}
proof (*cases symmetric-mpoly A p* ∧ *A* ⊆ *vars p*)
  case *True*
  define *B* **where** *B* = *vars p*

have *finite B vars p* ⊆ *B symmetric-mpoly A p A* ⊆ *B*
  using *True* **unfolding** *B-def* **by** (*auto simp*: *vars-finite*)
thus *?thesis*
proof (*induction p rule*: *fund-sym-poly-wit.induct*)
  case (*1 p*)
  show *?case*
  proof (*cases lead-monom p* = *0* ∨ *vars p* ∩ *A* = {})
    case *False*
    have *vars* (*p* − *fund-sym-step-poly p*) ⊆ *B*
      using *1.prems fund-sym-step-poly*[*of p*] **by** (*intro order.trans*[*OF vars-diff*])
auto
    hence *vars* (*local.fund-sym-poly-wit* (*p* − *local.fund-sym-step-poly p*)) ⊆ {*1..n*}
      using *False 1.prems*
    by (*intro 1 symmetric-mpoly-diff fund-sym-step-poly*) (*auto simp*: *lead-monom-eq-0-iff*)
    hence *vars* (*fund-sym-poly-wit* (*p* − *fund-sym-step-poly p*) +
        monom (*fund-sym-step-monom p*) (*local.fund-sym-step-coeff p*)) ⊆ {*1..n*}
    by (*intro order.trans*[*OF vars-add*] *Un-least order.trans*[*OF vars-monom-subset*]
          keys-fund-sym-step-monom) *auto*
    thus *?thesis* **using** *1.prems False* **unfolding** *fund-sym-poly-wit.simps*[*of p*]
by *simp*
  qed (*insert 1.prems*,
    auto simp: *fund-sym-poly-wit.simps*[*of p*] *mpoly-coeff-Const lead-monom-eq-0-iff*)
qed
next
  case *False*
  then consider ¬*symmetric-mpoly A p* | *symmetric-mpoly A p vars p* ∩ *A* = {}
    using *symmetric-mpoly-imp-orthogonal-or-subset*[*of A p*] **by** *auto*
  thus *?thesis*
    by *cases* (*auto simp*: *fund-sym-poly-wit.simps*[*of p*])
qed

lemma *fund-sym-poly-wit-insertion-aux*:
  assumes *finite B vars p* ⊆ *B symmetric-mpoly A p A* ⊆ *B*
  shows *insertion* (*sym-mpoly A*) (*fund-sym-poly-wit p*) = *p*
  using *assms*
proof (*induction p rule*: *fund-sym-poly-wit.induct*)
  case (*1 p*)
  from *1.prems* **have** *decr p*
    using *lookup-lead-monom-decreasing*[*of A p*] **by** (*auto simp*: *decr-def*)
  show *?case*
  proof (*cases lead-monom p* = *0* ∨ *vars p* ∩ *A* = {})
    case *False*
    have *vars* (*p* − *fund-sym-step-poly p*) ⊆ *B*
      using *1.prems fund-sym-step-poly*[*of p*] **by** (*intro order.trans*[*OF vars-diff*])
auto
    hence *insertion* (*sym-mpoly A*) (*fund-sym-poly-wit* (*p* − *fund-sym-step-poly p*))
=
        p − *fund-sym-step-poly p* **using** *1 False*
    by (*intro 1 symmetric-mpoly-diff fund-sym-step-poly*) *auto*

moreover have *fund-sym-step-poly p* =
fund-sym-step-coeff p ∗ (∏ *x. sym-mpoly A x* ^ *lookup* (*fund-sym-step-monom p*) *x*)
using *1.prems finite-subset*[*of A B*] *False* ‹*decr p*› **by** (*intro fund-sym-step-poly*) *auto*
ultimately show *?thesis*
unfolding *fund-sym-poly-wit.simps*[*of p*] **by** (*auto simp*: *insertion-add*)
qed (*auto simp*: *fund-sym-poly-wit.simps*[*of p*])
qed

lemma *fund-sym-poly-wit-insertion*:
assumes *symmetric-mpoly A p*
shows *insertion* (*sym-mpoly A*) (*fund-sym-poly-wit p*) = *p*
proof (*cases A* ⊆ *vars p*)
case *False*
hence *vars p* ∩ *A* = {}
using *symmetric-mpoly-imp-orthogonal-or-subset*[*OF assms*] **by** *auto*
thus *?thesis*
by (*auto simp*: *fund-sym-poly-wit.simps*[*of p*])
next
case *True*
with *fund-sym-poly-wit-insertion-aux*[*of vars p p*] *assms* **show** *?thesis*
by (*auto simp*: *vars-finite*)
qed

lemma *fund-sym-poly-wit-coeff*:
assumes ∀ *m. coeff p m* ∈ *C ring-closed C*
shows ∀ *m m′. coeff* (*coeff* (*fund-sym-poly-wit p*) *m*) *m′* ∈ *C*
using *assms*(*1*)
proof (*induction p rule*: *fund-sym-poly-wit.induct*)
case (*1 p*)
interpret *ring-closed C* **by** *fact*
show *?case*
proof (*cases* ¬*symmetric-mpoly A p* ∨ *lead-monom p* = *0* ∨ *vars p* ∩ *A* = {})
case *True*
thus *?thesis* **using** *1.prems*
by (*auto simp*: *fund-sym-poly-wit.simps*[*of p*] *mpoly-coeff-Const*)
next
case *False*
have ∗: ∀ *m m′. coeff* (*coeff* (*fund-sym-poly-wit* (*p* − *fund-sym-step-poly p*)) *m*) *m′* ∈ *C*
using *False 1.prems assms coeff-fund-sym-step-poly* [*of p*] **by** (*intro 1*) *auto*
show *?thesis*
proof (*intro allI*, *goal-cases*)
case (*1 m m′*)
thus *?case* **using** ∗ *False coeff-fund-sym-step-coeff*[*of p m′*] *1.prems*
by (*auto simp*: *fund-sym-poly-wit.simps*[*of p*] *coeff-monom lead-coeff-def when-def*)
qed

qed
qed

2.13 Uniqueness

Next, we show that the polynomial representation of a symmetric polynomial in terms of the elementary symmetric polynomials not only exists, but is unique.

The key property here is that products of powers of elementary symmetric polynomials uniquely determine the exponent vectors, i. e. if $e_1, \ldots, e_n$ are the elementary symmetric polynomials, $a = (a_1, \ldots, a_n)$ and $b = (b_1, \ldots, b_n)$ are vectors of natural numbers, then:

$$e_1^{a_1} \ldots e_n^{a_n} = e_1^{b_1} \ldots e_n^{b_n} \longleftrightarrow a = b$$

We show this now.

lemma *lead-monom-sym-mpoly-prod*:
assumes *finite A*
shows *lead-monom* ($\prod$ *i = 1..n. sym-mpoly A i ^ h i :: 'a mpoly*) =
($\sum$ *i = 1..n. of-nat* (*h i*) ∗ *lead-monom* (*sym-mpoly A i :: 'a mpoly*))
proof −
have ($\prod$ *i=1..n. lead-coeff* (*sym-mpoly A i ^ h i :: 'a mpoly*)) = *1*
using *assms* **unfolding** *n-def* **by** (*intro prod.neutral allI*) (*auto simp*: *lead-coeff-power*)
hence *lead-monom* ($\prod$ *i=1..n. sym-mpoly A i ^ h i :: 'a mpoly*) =
($\sum$ *i=1..n. lead-monom* (*sym-mpoly A i ^ h i :: 'a mpoly*))
by (*subst lead-monom-prod*) *auto*
also have ... = ($\sum$ *i=1..n. of-nat* (*h i*) ∗ *lead-monom* (*sym-mpoly A i :: 'a mpoly*))
by (*intro sum.cong refl*, *subst lead-monom-power*)
(*auto simp*: *lead-coeff-power assms n-def*)
finally show *?thesis* **.**
qed

lemma *lead-monom-sym-mpoly-prod-notin*:
assumes *finite A k* ∉ *A*
shows *lookup* (*lead-monom* ($\prod$ *i=1..n. sym-mpoly A i ^ h i :: 'a mpoly*)) *k* = *0*
proof −
have *xs*: *set xs* = *A distinct xs sorted xs* **and** [*simp*]: *length xs* = *n*
using *assms* **by** (*auto simp*: *xs-def n-def*)
have *lead-monom* ($\prod$ *i = 1..n. sym-mpoly A i ^ h i :: 'a mpoly*) =
($\sum$ *i = 1..n. of-nat* (*h i*) ∗ *lead-monom* (*sym-mpoly* (*set xs*) *i :: 'a mpoly*))
by (*subst lead-monom-sym-mpoly-prod*) (*use xs assms* **in** *auto*)
also have *lookup ... k* = *0* **unfolding** *lookup-sum*
by (*intro sum.neutral ballI*, *subst lead-monom-sym-mpoly*)
(*insert xs assms*, *auto simp*: *xs lead-monom-sym-mpoly lookup-monom-of-set set-conv-nth*)
finally show *?thesis* **.**
qed

lemma *lead-monom-sym-mpoly-prod-in*:
  assumes *finite A k* $<$ *n*
  shows *lookup* (*lead-monom* ($\prod$ *i*=*1*..*n*. *sym-mpoly A i* ^ *h i* :: '*a mpoly*)) (*xs* ! *k*) =
  ($\sum$ *i*=*k*+*1*..*n*. *h i*)
proof −
  have *xs*: *set xs* = *A distinct xs sorted xs* **and** [*simp*]: *length xs* = *n*
    using *assms* **by** (*auto simp*: *xs-def n-def*)
  have *lead-monom* ($\prod$ *i* = *1*..*n*. *sym-mpoly A i* ^ *h i* :: '*a mpoly*) =
      ($\sum$ *i* = *1*..*n*. *of-nat* (*h i*) ∗ *lead-monom* (*sym-mpoly* (*set xs*) *i* :: '*a mpoly*))
    by (*subst lead-monom-sym-mpoly-prod*) (*use xs assms* **in** *simp-all*)
  also have ... = ($\sum$ *i*=*1*..*n*. *of-nat* (*h i*) ∗ *monom-of-set* (*set* (*take i xs*)))
    using *xs* **by** (*intro sum.cong refl*, *subst lead-monom-sym-mpoly*) *auto*
  also have *lookup* ... (*xs* ! *k*) = ($\sum$ *i* | *i* ∈ {*1*..*n*} ∧ *xs* ! *k* ∈ *set* (*take i xs*). *h i*)
   unfolding *lookup-sum lookup-monom-of-set* **by** (*intro sum.mono-neutral-cong-right*) *auto*
  also have {*i*. *i* ∈ {*1*..*n*} ∧ *xs* ! *k* ∈ *set* (*take i xs*)} = {*k*+*1*..*n*}
  proof (*intro equalityI subsetI*)
    fix *i* **assume** *i*: *i* ∈ {*k*+*1*..*n*}
    hence *take i xs* ! *k* = *xs* ! *k k* < *n k* < *i* **using** *assms*
      by *auto*
    with *i* **show** *i* ∈ {*i*. *i* ∈ {*1*..*n*} ∧ *xs* ! *k* ∈ *set* (*take i xs*)}
      by (*force simp*: *set-conv-nth*)
  qed (*insert assms xs*, *auto simp*: *set-conv-nth Suc-le-eq nth-eq-iff-index-eq*)
  finally show *?thesis* **.**
qed

lemma *lead-monom-sym-poly-powerprod-inj*:
  assumes *lead-monom* ($\prod$ *i*. *sym-mpoly A i* ^ *lookup m1 i* :: '*a mpoly*) =
        lead-monom ($\prod$ *i*. *sym-mpoly A i* ^ *lookup m2 i* :: '*a mpoly*)
  assumes *finite A keys m1* ⊆ {*1*..*n*} *keys m2* ⊆ {*1*..*n*}
  shows *m1* = *m2*
proof (*rule poly-mapping-eqI*)
  fix *k* :: *nat*
  have *xs*: *set xs* = *A distinct xs sorted xs* **and** [*simp*]: *length xs* = *n*
    using *assms* **by** (*auto simp*: *xs-def n-def*)

  from *assms*(*3*,*4*) **have** ∗: *i* ∈ {*1*..*n*} **if** *lookup m1 i* ≠ *0* ∨ *lookup m2 i* ≠ *0* **for** *i*
    using *that* **by** (*auto simp*: *subset-iff in-keys-iff*)
  have ∗∗: ($\prod$ *i*. *sym-mpoly A i* ^ *lookup m i* :: '*a mpoly*) =
        ($\prod$ *i*=*1*..*n*. *sym-mpoly A i* ^ *lookup m i* :: '*a mpoly*) **if** *m* ∈ {*m1*, *m2*} **for** *m*
     using *that* ∗ **by** (*intro Prod-any.expand-superset subsetI* ∗) (*auto intro*!: *Nat.gr0I*)
  have ∗∗∗: *lead-monom* ($\prod$ *i*=*1*..*n*. *sym-mpoly A i* ^ *lookup m1 i* :: '*a mpoly*) =
        lead-monom ($\prod$ *i*=*1*..*n*. *sym-mpoly A i* ^ *lookup m2 i* :: '*a mpoly*)

using *assms* **by** (*simp add*: ∗∗)

have *sum-eq*: *sum* (*lookup m1*) {*Suc k..n*} = *sum* (*lookup m2*) {*Suc k..n*} **if** *k* < *n* **for** *k*
using *arg-cong*[*OF* ∗∗∗, *of* λ*m. lookup m* (*xs* ! *k*)] ‹*finite A*› *that*
by (*subst* (*asm*) (*1 2*) *lead-monom-sym-mpoly-prod-in*) *auto*

show *lookup m1 k* = *lookup m2 k*
proof (*cases k* ∈ {*1..n*})
case *False*
hence *lookup m1 k* = *0 lookup m2 k* = *0* **using** *assms* **by** (*auto simp*: *in-keys-iff*)
thus *?thesis* **by** *simp*
next
case *True*
thus *?thesis*
proof (*induction n* − *k arbitrary*: *k rule*: *less-induct*)
case (*less l*)
have *sum* (*lookup m1*) {*Suc* (*l* − *1*)..*n*} = *sum* (*lookup m2*) {*Suc* (*l* − *1*)..*n*}
using *less* **by** (*intro sum-eq*) *auto*
also have {*Suc* (*l* − *1*)..*n*} = *insert l* {*Suc l..n*}
using *less* **by** *auto*
also have *sum* (*lookup m1*) ... = *lookup m1 l* + (∑ *i*=*Suc l..n. lookup m1 i*)
by (*subst sum.insert*) *auto*
also have (∑ *i*=*Suc l..n. lookup m1 i*) = (∑ *i*=*Suc l..n. lookup m2 i*)
by (*intro sum.cong less*) *auto*
also have *sum* (*lookup m2*) (*insert l* {*Suc l..n*}) = *lookup m2 l* + (∑ *i*=*Suc l..n. lookup m2 i*)
by (*subst sum.insert*) *auto*
finally show *lookup m1 l* = *lookup m2 l* **by** *simp*
qed
qed
qed

We now show uniqueness by first showing that the zero polynomial has a unique representation. We fix some polynomial p with $p(e_1, \ldots, e_n) = 0$ and then show, by contradiction, that $p = 0$.

We have

$$p(e_1, \ldots, e_n) = \sum c_{a_1,\ldots,a_n} e_1^{a_1} \ldots e_n^{a_n}$$

and due to the injectivity of products of powers of elementary symmetric polynomials, the leading term of that sum is precisely the leading term of the summand with the biggest leading monomial, since summands cannot cancel each other.

However, we also know that $p(e_1, \ldots, e_n) = 0$, so it follows that all summands must have leading term 0, and it is then easy to see that they must all be identically 0.

lemma *sym-mpoly-representation-unique-aux*:
fixes *p* :: *'a mpoly mpoly*

```
  assumes finite A insertion (sym-mpoly A) p = 0
         ⋀m. vars (coeff p m) ∩ A = {} vars p ⊆ {1..n}
  shows   p = 0
proof (rule ccontr)
  assume p: p ≠ 0
  have xs: set xs = A distinct xs sorted xs and [simp]: length xs = n
    using assms by (auto simp: xs-def n-def)
  define h where h = (λm. coeff p m ∗ (∏ i. sym-mpoly A i ^ lookup m i))
  define M where M = {m. coeff p m ≠ 0}
  define maxm where maxm = Max ((lead-monom ∘ h) ‘ M)
  have finite M
    by (auto intro!: finite-subset[OF - finite-coeff-support[of p]] simp: h-def M-def)
  have keys-subset: keys m ⊆ {1..n} if coeff p m ≠ 0 for m
    using that assms coeff-notin-vars[of m p] by blast

  have lead-coeff: lead-coeff (h m) = lead-coeff (coeff p m) (is ?th1)
   and lead-monom: lead-monom (h m) = lead-monom (coeff p m) +
                    lead-monom (∏ i. sym-mpoly A i ^ lookup m i :: 'a mpoly) (is
?th2)
   if [simp]: coeff p m ≠ 0 for m
  proof −
    have (∏ i. sym-mpoly A i ^ lookup m i :: 'a mpoly) =
          (∏ i | lookup m i ≠ 0. sym-mpoly A i ^ lookup m i :: 'a mpoly)
      by (intro Prod-any.expand-superset) (auto intro!: Nat.gr0I)
    also have lead-coeff ... = 1
      using assms keys-subset[of m]
       by (intro lead-coeff-sym-mpoly-powerprod) (auto simp: in-keys-iff subset-iff
n-def)
   finally have eq: lead-coeff (∏ i. sym-mpoly A i ^ lookup m i :: 'a mpoly) = 1 .
    thus ?th1 unfolding h-def using ‹coeff p m ≠ 0› by (subst lead-coeff-mult)
auto
    show ?th2 unfolding h-def by (subst lead-monom-mult) (auto simp: eq)
  qed

  have insertion (sym-mpoly A) p = (∑ m∈M. h m)
     unfolding insertion-altdef h-def M-def by (intro Sum-any.expand-superset)
auto
  also have lead-monom ... = maxm
    unfolding maxm-def
  proof (rule lead-monom-sum)
    from p obtain m where coeff p m ≠ 0
      using mpoly-eqI[of p 0] by auto
    hence m ∈ M
      using ‹coeff p m ≠ 0› lead-coeff[of m] by (auto simp: M-def)
    thus M ≠ {} by auto
  next
    have restrict-lead-monom:
           restrictpm A (lead-monom (h m)) =
             lead-monom (∏ i. sym-mpoly A i ^ lookup m i :: 'a mpoly)
```

if [*simp*]: *coeff p m* $\neq$ *0* **for** *m*
proof −
have *restrictpm A* (*lead-monom* (*h m*)) =
restrictpm A (*lead-monom* (*coeff p m*)) +
restrictpm A (*lead-monom* ($\prod$ *i. sym-mpoly A i* ^ *lookup m i* :: ′*a mpoly*))
by (*auto simp*: *lead-monom restrictpm-add*)
also have *restrictpm A* (*lead-monom* (*coeff p m*)) = *0*
using *assms* **by** (*intro restrictpm-orthogonal order.trans*[*OF keys-lead-monom-subset*]) *auto*
also have *restrictpm A* (*lead-monom* ($\prod$ *i. sym-mpoly A i* ^ *lookup m i* :: ′*a mpoly*)) =
lead-monom ($\prod$ *i. sym-mpoly A i* ^ *lookup m i* :: ′*a mpoly*)
by (*intro restrictpm-id order.trans*[*OF keys-lead-monom-subset*]
order.trans[*OF vars-Prod-any*] *UN-least order.trans*[*OF vars-power*]
vars-sym-mpoly-subset)
finally show *?thesis* **by** *simp*
qed
show *inj-on* (*lead-monom* ∘ *h*) *M*
proof
fix *m1 m2* **assume** *m12*: *m1* ∈ *M m2* ∈ *M* (*lead-monom* ∘ *h*) *m1* = (*lead-monom* ∘ *h*) *m2*
hence [*simp*]: *coeff p m1* $\neq$ *0 coeff p m2* $\neq$ *0* **by** (*auto simp*: *M-def h-def*)
have *restrictpm A* (*lead-monom* (*h m1*)) = *restrictpm A* (*lead-monom* (*h m2*))
using *m12* **by** *simp*
hence *lead-monom* ($\prod$ *i. sym-mpoly A i* ^ *lookup m1 i* :: ′*a mpoly*) =
lead-monom ($\prod$ *i. sym-mpoly A i* ^ *lookup m2 i* :: ′*a mpoly*)
by (*simp add*: *restrict-lead-monom*)
thus *m1* = *m2*
by (*rule lead-monom-sym-poly-powerprod-inj*)
(*use* ‹*finite A*› *keys-subset*[*of m1*] *keys-subset*[*of m2*] **in** *auto*)
qed
next
fix *m* **assume** *m* ∈ *M*
hence *lead-coeff* (*h m*) = *lead-coeff* (*coeff p m*)
by (*simp add*: *lead-coeff M-def*)
with ‹*m* ∈ *M*› **show** *h m* $\neq$ *0* **by** (*auto simp*: *M-def*)
qed *fact*+
finally have *maxm* = *0* **by** (*simp add*: *assms*)

have *only-zero*: *m* = *0* **if** *m* ∈ *M* **for** *m*
proof −
from *that* **have** *nz* [*simp*]: *coeff p m* $\neq$ *0* **by** (*auto simp*: *M-def h-def*)
from *that* **have** (*lead-monom* ∘ *h*) *m* ≤ *maxm*
using ‹*finite M*› **unfolding** *maxm-def* **by** (*intro Max-ge imageI finite-imageI*)
with ‹*maxm* = *0*› **have** [*simp*]: *lead-monom* (*h m*) = *0* **by** *simp*
have *lookup-nzD*: *k* ∈ {*1..n*} **if** *lookup m k* $\neq$ *0* **for** *k*
using *keys-subset*[*of m*] *that* **by** (*auto simp*: *in-keys-iff subset-iff*)

have *lead-monom* (*coeff p m*) + *0* ≤ *lead-monom* (*h m*)
unfolding *lead-monom*[*OF nz*] **by** (*intro add-left-mono*) *auto*
also have ... = *0* **by** *simp*
finally have *lead-monom-0*: *lead-monom* (*coeff p m*) = *0* **by** *simp*

have *sum* (*lookup m*) {*1..n*} = *0*
proof (*rule ccontr*)
assume *sum* (*lookup m*) {*1..n*} ≠ *0*
hence *sum* (*lookup m*) {*1..n*} > *0* **by** *presburger*
have *0* ≠ *lead-coeff* (*MPoly-Type.coeff p m*)
by *auto*
also have *lead-coeff* (*MPoly-Type.coeff p m*) = *lead-coeff* (*h m*)
by (*simp add*: *lead-coeff*)
also have *lead-coeff* (*h m*) = *coeff* (*h m*) *0*
by (*simp add*: *lead-coeff-def*)
also have ... = *coeff* (*coeff p m*) *0* ∗ *coeff* (∏*i*. *sym-mpoly A i* ^ *lookup m i*) *0*
by (*simp add*: *h-def mpoly-coeff-times-0*)
also have (∏*i*. *sym-mpoly A i* ^ *lookup m i*) = (∏*i*=*1..n*. *sym-mpoly A i* ^ *lookup m i*)
by (*intro Prod-any.expand-superset subsetI lookup-nzD*) (*auto intro*!: *Nat.gr0I*)
also have *coeff* ... *0* = (∏*i*=*1..n*. *0* ^ *lookup m i*)
unfolding *mpoly-coeff-prod-0 mpoly-coeff-power-0*
by (*intro prod.cong*) (*auto simp*: *coeff-sym-mpoly-0*)
also have ... = *0* ^ (∑*i*=*1..n*. *lookup m i*)
by (*simp add*: *power-sum*)
also have ... = *0*
using *zero-power*[*OF* ‹*sum* (*lookup m*) {*1..n*} > *0*›] **by** *simp*
finally show *False* **by** *auto*
qed
hence *lookup m k* = *0* **for** *k*
using *keys-subset*[*of m*] **by** (*cases k* ∈ {*1..n*}) (*auto simp*: *in-keys-iff*)
thus *m* = *0* **by** (*intro poly-mapping-eqI*) *auto*
qed

have *0* = *insertion* (*sym-mpoly A*) *p*
using *assms* **by** *simp*
also have *insertion* (*sym-mpoly A*) *p* = (∑*m*∈*M*. *h m*)
by *fact*
also have ... = (∑*m*∈{*0*}. *h m*)
using *only-zero* **by** (*intro sum.mono-neutral-left*) (*auto simp*: *h-def M-def*)
also have ... = *coeff p 0*
by (*simp add*: *h-def*)
finally have *0* ∉ *M* **by** (*auto simp*: *M-def*)
with *only-zero* **have** *M* = {} **by** *auto*
hence *p* = *0* **by** (*intro mpoly-eqI*) (*auto simp*: *M-def*)
with ‹*p* ≠ *0*› **show** *False* **by** *contradiction*
qed

The general uniqueness theorem now follows easily. This essentially shows that the substitution $Y_i \mapsto e_i(X_1, \ldots, X_n)$ is an isomorphism between the ring $R[Y_1, \ldots, Y_n]$ and the ring $R[X_1, \ldots, X_n]^{S_n}$ of symmetric polynomials.

theorem *sym-mpoly-representation-unique*:
  fixes *p* :: *'a mpoly mpoly*
  assumes *finite A*
    insertion (*sym-mpoly A*) *p* = *insertion* (*sym-mpoly A*) *q*
    $\bigwedge$*m. vars* (*coeff p m*) ∩ *A* = {} $\bigwedge$*m. vars* (*coeff q m*) ∩ *A* = {}
    vars p ⊆ {*1..n*} *vars q* ⊆ {*1..n*}
  shows *p* = *q*
proof −
  have *p* − *q* = *0*
  proof (*rule sym-mpoly-representation-unique-aux*)
    fix *m* **show** *vars* (*coeff* (*p* − *q*) *m*) ∩ *A* = {}
      using *vars-diff*[*of coeff p m coeff q m*] *assms*(*3*,*4*)[*of m*] **by** *auto*
  qed (*insert assms vars-diff*[*of p q*], *auto simp*: *insertion-diff*)
  thus *?thesis* **by** *simp*
qed

theorem *eq-fund-sym-poly-witI*:
  fixes *p* :: *'a mpoly* **and** *q* :: *'a mpoly mpoly*
  assumes *finite A symmetric-mpoly A p*
    insertion (*sym-mpoly A*) *q* = *p*
    $\bigwedge$*m. vars* (*coeff q m*) ∩ *A* = {}
    vars q ⊆ {*1..n*}
  shows *q* = *fund-sym-poly-wit p*
  using *fund-sym-poly-wit-insertion*[*of p*] *fund-sym-poly-wit-vars*[*of p*]
    fund-sym-poly-wit-coeffs[*of p*]
  by (*intro sym-mpoly-representation-unique*)
    (*insert assms*, *auto simp*: *fund-sym-poly-wit-insertion*)

2.14 A recursive characterisation of symmetry

In a similar spirit to the proof of the fundamental theorem, we obtain a nice recursive and executable characterisation of symmetry.

function (*domintros*) *check-symmetric-mpoly* **where**
  check-symmetric-mpoly p ⟷
    (*vars p* ∩ *A* = {} ∨
     A ⊆ *vars p* ∧ *decr p* ∧ *check-symmetric-mpoly* (*p* − *fund-sym-step-poly p*))
  by *auto*

lemma *check-symmetric-mpoly-dom-aux*:
  assumes *finite B vars p* ⊆ *B A* ⊆ *B*
  shows *check-symmetric-mpoly-dom p*
  using *assms*(*1*−*3*)
proof (*induction p rule*: *lead-monom-induct*)
  case (*less p*)
  have [*simp*]: *finite A* **by** (*rule finite-subset*[*of - B*]) *fact+*

  show *?case*
  proof (*cases lead-monom p* = *0* ∨ ¬*decr p*)
    case *False*
    hence [*simp*]: *p* ≠ *0* **by** *auto*
    have *vars* (*p* − *fund-sym-step-poly p*) ⊆ *B*
      using *fund-sym-step-poly*[*of p*] *False less.prems less.hyps* ‹*A* ⊆ *B*›
      by (*intro order.trans*[*OF vars-diff*]) *auto*
    hence *check-symmetric-mpoly-dom* (*p* − *local.fund-sym-step-poly p*)
      using *False less.prems less.hyps*
      by (*intro less.IH fund-sym-step-poly symmetric-mpoly-diff*
              lead-monom-fund-sym-step-poly-less) (*auto simp*: *decr-def*)
    thus *?thesis* **using** *check-symmetric-mpoly.domintros* **by** *blast*
  qed (*auto intro*: *check-symmetric-mpoly.domintros simp*: *lead-monom-eq-0-iff*)
qed

lemma *check-symmetric-mpoly-dom* [*intro*]: *check-symmetric-mpoly-dom p*
proof −
  show *?thesis*
  proof (*cases A* ⊆ *vars p*)
    assume *A* ⊆ *vars p*
    thus *?thesis* **using** *check-symmetric-mpoly-dom-aux*[*of vars p p*] **by** (*auto simp*:
vars-finite)
  qed (*auto intro*: *check-symmetric-mpoly.domintros*)
qed

termination *check-symmetric-mpoly*
  by (*intro allI check-symmetric-mpoly-dom*)

lemmas [*simp del*] = *check-symmetric-mpoly.simps*

lemma *check-symmetric-mpoly-correct*: *check-symmetric-mpoly p* ⟷ *symmetric-mpoly*
A p
proof (*induction p rule*: *check-symmetric-mpoly.induct*)
  case (*1 p*)
  have *symmetric-mpoly A* (*p* − *fund-sym-step-poly p*) ⟷ *symmetric-mpoly A p*
(**is** *?lhs* = *?rhs*)
  proof
    assume *?rhs*
    thus *?lhs* **by** (*intro symmetric-mpoly-diff fund-sym-step-poly*)
  next
    assume *?lhs*
    hence *symmetric-mpoly A* (*p* − *fund-sym-step-poly p* + *fund-sym-step-poly p*)
      by (*intro symmetric-mpoly-add fund-sym-step-poly*)
    thus *?rhs* **by** *simp*
  qed
  moreover have *decr p* **if** *symmetric-mpoly A p*
    using *lookup-lead-monom-decreasing*[*of A p*] *that* **by** (*auto simp*: *decr-def*)
  ultimately show *check-symmetric-mpoly p* ⟷ *symmetric-mpoly A p*
    using *1 symmetric-mpoly-imp-orthogonal-or-subset*[*of A p*]

by (*auto simp*: *Let-def check-symmetric-mpoly.simps*[*of p*] *intro*: *symmetric-mpoly-orthogonal*)
qed

end

2.15 Symmetric functions of roots of a univariate polynomial

Consider a factored polynomial

$$p(X) = c_n X^n + c_{n-1} X^{n-1} + \ldots + c_1 X + c_0 = (X - x_1) \ldots (X - x_n)\ .$$

where c_n is a unit.

Then any symmetric polynomial expression $q(x_1, \ldots, x_n)$ in the roots x_i can be written as a polynomial expression $q'(c_0, \ldots, c_{n-1})$ in the c_i.

Moreover, if the coefficients of q and the inverse of c_n all lie in some subring, the coefficients of q' do as well.

context
  fixes C :: *'b* :: *comm-ring-1 set*
    and A :: *nat set*
    and *root* :: *nat* ⇒ *'a* :: *comm-ring-1*
    and l :: *'a* ⇒ *'b*
    and q :: *'b mpoly*
    and n :: *nat*
  defines $n \equiv$ *card A*
  assumes *C*: *ring-closed C* ∀ *m*. *coeff q m* ∈ *C*
  assumes *l*: *ring-homomorphism l*
  assumes *finite*: *finite A*
  assumes *sym*: *symmetric-mpoly A q* **and** *vars*: *vars q* ⊆ *A*
begin

interpretation *ring-closed C* **by** *fact*
interpretation *ring-homomorphism l* **by** *fact*

theorem *symmetric-poly-of-roots-conv-poly-of-coeffs*:
  assumes *c*: *cinv* ∗ *l c* = *1 cinv* ∈ *C*
  assumes *p* = *Polynomial.smult c* (∏ *i*∈*A*. [:−*root i*, *1*:])
  obtains *q'* **where** *vars q'* ⊆ {*0*..<*n*}
        and ⋀*m*. *coeff q' m* ∈ *C*
        and *insertion* (*l* ∘ *poly.coeff p*) *q'* = *insertion* (*l* ∘ *root*) *q*
proof −
  define *q'* **where** *q'* = *fund-sym-poly-wit A q*
  define *q''* **where** *q''* =
    mapm-mpoly (λ*m x*. (∏ *i*. (*cinv* ∗ *l* (− *1*) ^ *i*) ^ *lookup m i*) ∗ *insertion* (λ-.
0) *x*) *q'*
  define *reindex* **where** *reindex* = (λ*i*. *if i* ≤ *n then n* − *i else i*)
  have *bij reindex*
    by (*intro bij-betwI*[*of reindex - - reindex*]) (*auto simp*: *reindex-def*)
  have *vars q'* ⊆ {*1*..*n*} **unfolding** *q'-def n-def* **by** (*intro fund-sym-poly-wit-vars*)

hence *vars* $q'' \subseteq \{1..n\}$
unfolding *q''-def* **using** *vars-mapm-mpoly-subset* **by** *auto*

have *insertion* ($l \circ$ *root*) (*insertion* (*sym-mpoly A*) q') =
insertion (λn. *insertion* ($l \circ$ *root*) (*sym-mpoly A n*))
(*map-mpoly* (*insertion* ($l \circ$ *root*)) q')
by (*rule insertion-insertion*)
also have *insertion* (*sym-mpoly A*) $q' = q$
unfolding *q'-def* **by** (*intro fund-sym-poly-wit-insertion sym*)
also have *insertion* (λi. *insertion* ($l \circ$ *root*) (*sym-mpoly A i*))
(*map-mpoly* (*insertion* ($l \circ$ *root*)) q') =
insertion (λi. *cinv* $*$ *l* (($-$ *1*) ^ *i*) $*$ *l* (*poly.coeff p* ($n - i$)))
(*map-mpoly* (*insertion* ($l \circ$ *root*)) q')
proof (*intro insertion-irrelevant-vars*, *goal-cases*)
case (*1 i*)
hence $i \in$ *vars* q' **using** *vars-map-mpoly-subset* **by** *auto*
also have ... $\subseteq \{1..n\}$ **unfolding** *q'-def n-def*
by (*intro fund-sym-poly-wit-vars*)
finally have *i*: $i \in \{1..n\}$ **.**
have *insertion* ($l \circ$ *root*) (*sym-mpoly A i*) =
l ($\sum Y \mid Y \subseteq A \wedge$ *card* $Y = i$. *prod root Y*)
using ‹*finite A*› **by** (*simp add*: *insertion-sym-mpoly*)
also have ... = *cinv* $*$ *l* (*c* $*$ ($\sum Y \mid Y \subseteq A \wedge$ *card* $Y = i$. *prod root Y*))
unfolding *mult mult.assoc*[*symmetric*] ‹*cinv* $*$ *l c* = *1*› **by** *simp*
also have *c* $*$ ($\sum Y \mid Y \subseteq A \wedge$ *card* $Y = i$. *prod root Y*) = (($-$*1*) ^ *i* $*$
poly.coeff p ($n - i$))
using *coeff-poly-from-roots*[*of A n* $-$ *i root*] *i assms finite*
by (*auto simp*: *n-def minus-one-power-iff*)
finally show *?case* **by** (*simp add*: *o-def*)
qed
also have *map-mpoly* (*insertion* ($l \circ$ *root*)) q' = *map-mpoly* (*insertion* (λ-. *0*))
q'
using *fund-sym-poly-wit-coeffs*[*OF sym*] *vars*
by (*intro map-mpoly-cong insertion-irrelevant-vars*) (*auto simp*: *q'-def*)
also have *insertion* (λi. *cinv* $*$ *l* (($-$ *1*) ^ *i*) $*$ *l* (*poly.coeff p* ($n - i$))) ... =
insertion (λi. *l* (*poly.coeff p* ($n - i$))) q''
unfolding *insertion-substitute-linear map-mpoly-conv-mapm-mpoly q''-def*
by (*subst mapm-mpoly-comp*) *auto*
also have ... = *insertion* ($l \circ$ *poly.coeff p*) (*mpoly-map-vars reindex* q'')
using ‹*bij reindex*› **and** ‹*vars* $q'' \subseteq \{1..n\}$›
by (*subst insertion-mpoly-map-vars*)
(*auto simp*: *o-def reindex-def intro*!: *insertion-irrelevant-vars*)
finally have *insertion* ($l \circ$ *root*) *q* =
insertion ($l \circ$ *poly.coeff p*) (*mpoly-map-vars reindex* q'') **.**

moreover have *coeff* (*mpoly-map-vars reindex* q'') $m \in C$ **for** *m*
unfolding *q''-def q'-def* **using** ‹*bij reindex*› *fund-sym-poly-wit-coeff*[*of q C A*]
C ‹*cinv* $\in C$›
by (*auto simp*: *coeff-mpoly-map-vars*

intro!: *mult-closed Prod-any-closed power-closed Sum-any-closed*)
moreover have *vars* (*mpoly-map-vars reindex* q'') $\subseteq$ {*0*..<*n*}
using ‹*bij reindex*› **and** ‹*vars* q'' $\subseteq$ {*1*..*n*}›
by (*subst vars-mpoly-map-vars*) (*auto simp*: *reindex-def subset-iff*)+
ultimately show *?thesis* **using** *that*[*of mpoly-map-vars reindex* q''] **by** *auto*
qed

corollary *symmetric-poly-of-roots-conv-poly-of-coeffs-monic*:
assumes p = ($\prod$ *i*∈*A*. [:−*root i*, *1*:])
obtains q' **where** *vars* q' $\subseteq$ {*0*..<*n*}
and $\bigwedge$*m*. *coeff* q' *m* ∈ *C*
and *insertion* (*l* ∘ *poly.coeff p*) q' = *insertion* (*l* ∘ *root*) *q*
proof −
obtain q' **where** *vars* q' $\subseteq$ {*0*..<*n*}
and $\bigwedge$*m*. *coeff* q' *m* ∈ *C*
and *insertion* (*l* ∘ *poly.coeff p*) q' = *insertion* (*l* ∘ *root*) *q*
by (*rule symmetric-poly-of-roots-conv-poly-of-coeffs*[*of 1 1 p*])
(*use assms* **in** *auto*)
thus *?thesis* **by** (*intro that*[*of q'*]) *auto*
qed

As a corollary, we obtain the following: Let R, S be rings with $R \subseteq S$. Consider a polynomial $p \in R[X]$ whose leading coefficient c is a unit in R and that has a full set of roots $x_1, \ldots, x_n \in S$, i. e. $p(X) = c(X - x_1) \ldots (X - x_n)$. Let $q \in R[X_1, \ldots, X_n]$ be some symmetric polynomial expression in the roots. Then $q(x_1, \ldots, x_n) \in R$.

A typical use case is $R = \mathbb{Q}$ and $S = \mathbb{C}$, i. e. any symmetric polynomial expression with rational coefficients in the roots of a rational polynomial is again rational. Similarly, any symmetric polynomial expression with integer coefficients in the roots of a monic integer polynomial is agan an integer.

This is remarkable, since the roots themselves are usually not rational (possibly not even real). This particular fact is a key ingredient used in the standard proof that π is transcendental.

corollary *symmetric-poly-of-roots-in-subring*:
assumes *cinv* ∗ *l c* = *1 cinv* ∈ *C*
assumes p = *Polynomial.smult c* ($\prod$ *i*∈*A*. [:−*root i*, *1*:])
assumes ∀ *i*. *l* (*poly.coeff p i*) ∈ *C*
shows *insertion* (λ*x*. *l* (*root x*)) *q* ∈ *C*
proof −
obtain q'
where q': *vars* q' $\subseteq$ {*0*..<*n*} $\bigwedge$*m*. *coeff* q' *m* ∈ *C*
insertion (*l* ∘ *poly.coeff p*) q' = *insertion* (*l* ∘ *root*) *q*
by (*rule symmetric-poly-of-roots-conv-poly-of-coeffs*[*of cinv c p*])
(*use assms* **in** *simp-all*)
have *insertion* (*l* ∘ *poly.coeff p*) q' ∈ *C* **using** *C assms* **unfolding** *insertion-altdef*
by (*intro Sum-any-closed mult-closed* q' *Prod-any-closed power-closed*) *auto*
also have *insertion* (*l* ∘ *poly.coeff p*) q' = *insertion* (*l* ∘ *root*) *q* **by** *fact*

finally show *?thesis* **by** (*simp add*: *o-def*)
qed

corollary *symmetric-poly-of-roots-in-subring-monic*:
  assumes $p = (\prod i{\in}A.\ [:-root\ i,\ 1:])$
  assumes $\forall\, i.\ l\ (poly.coeff\ p\ i) \in C$
  shows $insertion\ (\lambda x.\ l\ (root\ x))\ q \in C$
proof −
  interpret *ring-closed C* **by** *fact*
  interpret *ring-homomorphism l* **by** *fact*
  show *?thesis*
    by (*rule symmetric-poly-of-roots-in-subring*[*of 1 1 p*]) (*use assms* **in** *auto*)
qed

end

end

3 Executable Operations for Symmetric Polynomials

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

Lastly, we shall provide some code equations to get executable code for operations related to symmetric polynomials, including, most notably, the fundamental theorem of symmetric polynomials and the recursive symmetry check.

lemma *Ball-subset-right*:
  assumes $T \subseteq S\ \forall\, x{\in}S-T.\ P\ x$
  shows $(\forall\, x{\in}S.\ P\ x) = (\forall\, x{\in}T.\ P\ x)$
  using *assms* **by** *auto*

lemma *compute-less-pp*[*code*]:
  $xs < (ys :: 'a :: linorder \Rightarrow_0 'b :: \{zero,\ linorder\}) \longleftrightarrow$
    $(\exists\, i{\in}keys\ xs \cup keys\ ys.\ lookup\ xs\ i < lookup\ ys\ i\ \wedge$
    $(\forall\, j{\in}keys\ xs \cup keys\ ys.\ j < i \longrightarrow lookup\ xs\ j = lookup\ ys\ j))$
proof *transfer*
  fix $f\ g :: 'a \Rightarrow 'b$
  let $?dom = \{i.\ f\ i \neq 0\} \cup \{i.\ g\ i \neq 0\}$
  have $less\text{-}fun\ f\ g \longleftrightarrow (\exists\, k.\ f\ k < g\ k \wedge (\forall\, k'{<}k.\ f\ k' = g\ k'))$
    unfolding *less-fun-def* **..**
  also have $\ldots \longleftrightarrow (\exists\, i.\ f\ i < g\ i \wedge (i \in ?dom \wedge (\forall\, j{\in}?dom.\ j < i \longrightarrow f\ j = g\ j)))$
  proof (*intro iff-exI conj-cong refl*)
    fix *k* **assume** $f\ k < g\ k$

hence *k*: *k* ∈ *?dom* **by** *auto*
have (∀ *k'*<*k*. *f k'* = *g k'*) = (∀ *k'*∈{..<*k*}. *f k'* = *g k'*)
by *auto*
also have ... ⟷ (∀ *j*∈({*k*. *f k* ≠ *0*} ∪ {*k*. *g k* ≠ *0*}) ∩ {..<*k*}. *f j* = *g j*)
by (*intro Ball-subset-right*) *auto*
also have ... ⟷ (∀ *j*∈({*k*. *f k* ≠ *0*} ∪ {*k*. *g k* ≠ *0*}). *j* < *k* ⟶ *f j* = *g j*)
by *auto*
finally show (∀ *k'*<*k*. *f k'* = *g k'*) ⟷ *k* ∈ *?dom* ∧ (∀ *j*∈*?dom*. *j* < *k* ⟶ *f j* = *g j*)
using *k* **by** *simp*
qed
also have ... ⟷ (∃ *i*∈*?dom*. *f i* < *g i* ∧ (∀ *j*∈*?dom*. *j* < *i* ⟶ *f j* = *g j*))
by (*simp add*: *Bex-def conj-ac*)
finally show *less-fun f g* ⟷ (∃ *i*∈*?dom*. *f i* < *g i* ∧ (∀ *j*∈*?dom*. *j* < *i* ⟶ *f j* = *g j*)) **.**
qed

lemma *compute-le-pp*[*code*]:
xs ≤ *ys* ⟷ *xs* = *ys* ∨ *xs* < (*ys* :: - $\Rightarrow_0$ -)
by (*auto simp*: *order.order-iff-strict*)

lemma *vars-code* [*code*]:
vars (*MPoly p*) = (⋃ *m*∈*keys p*. *keys m*)
unfolding *vars-def* **by** *transfer' simp*

lemma *mpoly-coeff-code* [*code*]: *coeff* (*MPoly p*) = *lookup p*
by *transfer' simp*

lemma *sym-mpoly-code* [*code*]:
sym-mpoly (*set xs*) *k* = (∑ *X*∈*Set.filter* (λ*X*. *card X* = *k*) (*Pow* (*set xs*)). *monom* (*monom-of-set X*) *1*)
by (*simp add*: *sym-mpoly-altdef Set.filter-def*)

lemma *monom-of-set-code* [*code*]:
monom-of-set (*set xs*) = *Pm-fmap* (*fmap-of-list* (*map* (λ*x*. (*x*, *1*)) *xs*))
(**is** *?lhs* = *?rhs*)
proof (*intro poly-mapping-eqI*)
fix *k*
show *lookup ?lhs k* = *lookup ?rhs k*
by (*induction xs*) (*auto simp*: *lookup-monom-of-set fmlookup-default-def*)
qed

lemma *restrictpm-code* [*code*]:
restrictpm A (*Pm-fmap m*) = *Pm-fmap* (*fmrestrict-set A m*)
by (*intro poly-mapping-eqI*) (*auto simp*: *lookup-restrictpm fmlookup-default-def*)

lemmas [*code*] = *check-symmetric-mpoly-correct* [*symmetric*]

notepad

```
begin
  define X Y Z :: int mpoly where X = Var 1 Y = Var 2 Z = Var 3
  define e1 e2 :: int mpoly mpoly where e1 = Var 1 e2 = Var 2
  have sym-mpoly {1, 2, 3} 2 = X * Y + X * Z + Y * Z
    unfolding X-Y-Z-def by eval
  have symmetric-mpoly {1, 2} (X ^ 3 + Y ^ 3)
    unfolding X-Y-Z-def by eval
  have fund-sym-poly-wit {1, 2} (X ^ 3 + Y ^ 3) = e1 ^ 3 − 3 * e1 * e2
    unfolding X-Y-Z-def e1-e2-def by eval
end

end
```

References

[1] B. Blum-Smith and S. Coskey. The fundamental theorem on symmetric polynomials: History's first whiff of Galois theory. 48, 01 2013.