Alon's Combinatorial Nullstellensatz

Arthur F. Ramos David Barros Hulak Ruy J. G. B. de Queiroz

June 25, 2026

Abstract

This entry formalizes Alon's Combinatorial Nullstellensatz for sparse multivariate polynomials over fields. The proof derives the coefficient formula from univariate Lagrange interpolation and uses it to obtain the standard nonvanishing conclusion over finite grids. AI assistance was used for proof engineering. The final definitions, statements, and proofs are checked by Isabelle.

1 Overview

Alon's Combinatorial Nullstellensatz is a polynomial-method theorem about a distinguished coefficient of a multivariate polynomial. In the form formalized here, if all monomials have total degree bounded by $d_1 + \cdots + d_n$ and the coefficient of $X_1^{d_1} \cdots X_n^{d_n}$ is nonzero, then the polynomial has a nonzero value on every product grid whose i-th side has more than d_i elements.

The Isabelle/HOL development represents multivariate polynomials sparsely, as finite-support coefficient functions from exponent lists to field elements. It first proves the necessary univariate Lagrange interpolation identities, then sums polynomial evaluations over a finite grid with the corresponding Lagrange weights. All monomials except the distinguished exponent vector cancel, yielding the coefficient formula and the nonvanishing theorem.

2 Sources

The entry follows Alon's original paper [1]. The formal proof uses only Isabelle/HOL's standard computational algebra library as its mathematical base.

Contents

theory *Combinatorial-Nullstellensatz*
imports *HOL−Computational-Algebra.Polynomial*
begin

3 Alon's Combinatorial Nullstellensatz

Alon's Combinatorial Nullstellensatz [1] is a polynomial method theorem: if the coefficient of a distinguished monomial of total degree $d_1 + \ldots + d_n$ is nonzero, then the polynomial cannot vanish on every point of any grid whose i-th side has more than d_i elements.

The development below proves this statement for sparse multivariate polynomials over arbitrary fields. A polynomial is represented by a finite-support coefficient function from exponent lists to coefficients. This keeps the formalization independent of a particular multivariate polynomial library while still matching the usual mathematical statement.

3.1 Univariate interpolation

The proof starts with a small amount of univariate interpolation. For a finite set S and a point $x \in S$, the following denominator and basis polynomial are the standard Lagrange factors.

```
definition lagrange-denom :: 'a::field set ⇒ 'a ⇒ 'a where
  lagrange-denom S x = (∏ y∈S − {x}. x − y)

definition lagrange-basis :: 'a::field set ⇒ 'a ⇒ 'a poly where
  lagrange-basis S x =
    smult (inverse (lagrange-denom S x))
      (∏ y∈S − {x}. [:− y, 1:])

lemma lagrange-denom-nonzero:
  assumes finite S x ∈ S
  shows lagrange-denom S x ≠ 0
  using assms by (auto simp: lagrange-denom-def)

lemma poly-lagrange-basis:
  assumes finite S x ∈ S z ∈ S
  shows poly (lagrange-basis S x) z = (if z = x then 1 else 0)
proof (cases z = x)
  case True
  have poly (∏ y∈S − {x}. [:− y, 1:]) x = lagrange-denom S x
    by (simp add: lagrange-denom-def poly-prod)
  with assms True lagrange-denom-nonzero[of S x] show ?thesis
    by (simp add: lagrange-basis-def)
next
  case False
  with assms have z ∈ S − {x}
    by simp
  then have poly (∏ y∈S − {x}. [:− y, 1:]) z = 0
    using assms by (simp add: poly-prod)
  with False show ?thesis
    by (simp add: lagrange-basis-def)
qed

lemma degree-lagrange-basis-le:
  assumes finite S x ∈ S
  shows degree (lagrange-basis S x) ≤ card S − 1
proof −
  have degree (∏ y∈S − {x}. [:− y, 1:] :: 'a poly)
      = (∑ y∈S − {x}. degree ([:− y, 1:] :: 'a::field poly))
    using assms by (intro degree-prod-sum-eq) auto
  also have ... ≤ card S − 1
    using assms by auto
  finally show ?thesis
    by (simp add: lagrange-basis-def degree-smult-le dual-order.trans)
qed

lemma lagrange-interpolation:
  fixes P :: 'a::{field,ring-no-zero-divisors} poly
  assumes fin: finite S and deg: degree P < card S
```

```
  shows P = (∑x∈S. smult (poly P x) (lagrange-basis S x))
proof (rule poly-eqI-degree[where A = S])
  fix z
  assume z: z ∈ S
  have poly (∑x∈S. smult (poly P x) (lagrange-basis S x)) z =
      (∑x∈S. poly P x * poly (lagrange-basis S x) z)
    by (simp add: poly-sum)
  also have ... = poly P z
    using fin z
    by (subst sum.remove[of S z]) (auto simp: poly-lagrange-basis eq-commute)
  finally show poly P z = poly (∑x∈S. smult (poly P x) (lagrange-basis S x)) z
    by simp
next
  show card S > degree P
    using deg by simp
next
  have degree (∑x∈S. smult (poly P x) (lagrange-basis S x)) ≤ card S − 1
  proof (intro degree-sum-le fin)
    fix x
    assume x ∈ S
    then show degree (smult (poly P x) (lagrange-basis S x)) ≤ card S − 1
      by (meson degree-lagrange-basis-le degree-smult-le fin order-trans)
  qed
  moreover from deg have card S > 0
    by simp
  ultimately show card S > degree (∑x∈S. smult (poly P x) (lagrange-basis S x))
    by linarith
qed

lemma coeff-lagrange-basis-top:
  assumes fin: finite S and x: x ∈ S
  shows poly.coeff (lagrange-basis S x) (card S − 1) =
    inverse (lagrange-denom S x)
proof −
  have deg-prod: degree (∏y∈S − {x}. [:− y, 1:] :: 'a::field poly) = card S − 1
  proof −
    have degree (∏y∈S − {x}. [:− y, 1:] :: 'a poly) = (∑y∈S − {x}. degree ([:− y, 1:] :: 'a poly))
      using fin by (intro degree-prod-sum-eq) auto
    also have ... = card S − 1
      using fin x by simp
    finally show ?thesis .
  qed
  have poly.coeff (lagrange-basis S x) (card S − 1) =
      inverse (lagrange-denom S x) *
        poly.coeff (∏y∈S − {x}. [:− y, 1:] :: 'a poly) (card S − 1)
    by (simp add: lagrange-basis-def)
  also have ... = inverse (lagrange-denom S x) *
      lead-coeff (∏y∈S − {x}. [:− y, 1:] :: 'a poly)
    by (simp add: deg-prod)
  also have ... = inverse (lagrange-denom S x)
    using fin by (simp add: lead-coeff-prod)
  finally show ?thesis .
qed

lemma lagrange-power-sum:
  fixes S :: 'a::field set
  assumes fin: finite S and card: card S = Suc d
  assumes k: k ≤ d
  shows (∑x∈S. x ^ k / lagrange-denom S x) = (if k = d then 1 else 0)
```

```
proof −
  let ?X = monom (1::'a) k
  have interp: ?X = (∑ x∈S. smult (poly ?X x) (lagrange-basis S x))
    using card k by (intro lagrange-interpolation[OF fin]) (simp add: degree-monom-eq)
  have coeff-basis:
    poly.coeff (lagrange-basis S x) d = inverse (lagrange-denom S x) if x ∈ S for x
    using coeff-lagrange-basis-top[OF fin that] card by simp
  have poly.coeff ?X d =
      poly.coeff (∑ x∈S. smult (poly ?X x) (lagrange-basis S x)) d
    using interp by simp
  also have ... = (∑ x∈S. x ^ k / lagrange-denom S x)
    using fin by (simp add: coeff-sum poly-monom coeff-basis divide-inverse)
  finally show ?thesis
    using k by simp
qed

lemma lagrange-power-sum-list:
  fixes xs :: 'a::field list
  assumes dist: distinct xs and len: length xs = Suc d and k: k ≤ d
  shows sum-list (map (λx. x ^ k / lagrange-denom (set xs) x) xs) =  (if k = d then 1 else 0)
proof −
  have sum-list (map (λx. x ^ k / lagrange-denom (set xs) x) xs) =
      (∑ x∈set xs. x ^ k / lagrange-denom (set xs) x)
    using dist by (rule sum-list-distinct-conv-sum-set)
  also have ... = (if k = d then 1 else 0)
    using dist len k by (intro lagrange-power-sum) (auto simp: distinct-card)
  finally show ?thesis .
qed
```

3.2 Sparse multivariate polynomials

Monomials are indexed by exponent lists. The value of $[e_1, \ldots, e_n]$ at $[x_1, \ldots, x_n]$ is $x_1 \hat{} e_1 * \ldots * x_n \hat{} e_n$; mismatched lengths evaluate to zero. The predicate *sparse-poly* records finite support and a fixed arity.

```
fun monomial-value :: nat list ⇒ 'a::comm-semiring-1 list ⇒ 'a where
  monomial-value [] [] = 1
| monomial-value (e # es) (x # xs) = x ^ e * monomial-value es xs
| monomial-value - - = 0

fun grid-weight :: 'a::field list list ⇒ 'a list ⇒ 'a where
  grid-weight [] [] = 1
| grid-weight (S # Ss) (x # xs) = lagrange-denom (set S) x * grid-weight Ss xs
| grid-weight - - = 1

definition support :: (nat list ⇒ 'a::zero) ⇒ nat list set where
  support p = {m. p m ≠ 0}

definition sparse-poly :: nat ⇒ (nat list ⇒ 'a::zero) ⇒ bool where
  sparse-poly n p ⟷ finite (support p) ∧ (∀ m∈support p. length m = n)

definition total-degree-le :: (nat list ⇒ 'a::zero) ⇒ nat ⇒ bool where
  total-degree-le p d ⟷ (∀ m∈support p. sum-list m ≤ d)

definition eval-sparse-poly :: (nat list ⇒ 'a::comm-semiring-1) ⇒ 'a list ⇒ 'a where
  eval-sparse-poly p xs = (∑ m∈support p. p m * monomial-value m xs)

lemma product-lists-set-Cons:
  set (product-lists (xs # xss)) = (λ(x, ys). x # ys) ' (set xs × set (product-lists xss))
```

by *auto*

lemma *sum-list-concat*:
sum-list (*concat xss*) = *sum-list* (*map sum-list xss*)
by (*induction xss*) *simp-all*

lemma *sum-list-map-zero*:
fixes *f* :: *'a* ⇒ *'b*::*monoid-add*
assumes ⋀*x*. *x* ∈ *set xs* ⟹ *f x* = *0*
shows *sum-list* (*map f xs*) = *0*
using *assms* **by** (*induction xs*) *auto*

lemma *sum-list-product-lists-Cons*:
sum-list (*map f* (*product-lists* (*xs* # *xss*))) =
sum-list (*map* (λ*x*. *sum-list* (*map* (λ*ys*. *f* (*x* # *ys*)) (*product-lists xss*))) *xs*)
by (*simp add*: *sum-list-concat map-concat comp-def*)

lemma *grid-weight-Cons*:
grid-weight (*S* # *Ss*) (*x* # *xs*) = *lagrange-denom* (*set S*) *x* ∗ *grid-weight Ss xs*
by *simp*

lemma *monomial-value-Cons*:
monomial-value (*e* # *es*) (*x* # *xs*) = *x* ^ *e* ∗ *monomial-value es xs*
by *simp*

definition *grid-monom-sum* :: *'a*::*field list list* ⇒ *nat list* ⇒ *'a* **where**
grid-monom-sum Xss es =
sum-list (*map* (λ*xs*. *monomial-value es xs* / *grid-weight Xss xs*) (*product-lists Xss*))

lemma *grid-monom-sum-Cons*:
assumes *dist*: *distinct Xs*
shows *grid-monom-sum* (*Xs* # *Xss*) (*e* # *es*) =
sum-list (*map* (λ*x*. *x* ^ *e* / *lagrange-denom* (*set Xs*) *x*) *Xs*) ∗
grid-monom-sum Xss es
proof −
have *denom-nz*: *lagrange-denom* (*set Xs*) *x* ≠ *0* **if** *x* ∈ *set Xs* **for** *x*
using *that* **by** (*intro lagrange-denom-nonzero*) *auto*
have *inner*:
sum-list (*map* (λ*xs*. *monomial-value* (*e* # *es*) (*x* # *xs*) /
grid-weight (*Xs* # *Xss*) (*x* # *xs*)) (*product-lists Xss*)) =
(*x* ^ *e* / *lagrange-denom* (*set Xs*) *x*) ∗
sum-list (*map* (λ*xs*. *monomial-value es xs* / *grid-weight Xss xs*)
(*product-lists Xss*))
if *x* ∈ *set Xs* **for** *x*
proof −
have *term-eq*:
⋀*xs*. *monomial-value* (*e* # *es*) (*x* # *xs*) / *grid-weight* (*Xs* # *Xss*) (*x* # *xs*) =
(*x* ^ *e* / *lagrange-denom* (*set Xs*) *x*) ∗
(*monomial-value es xs* / *grid-weight Xss xs*)
using *denom-nz*[*OF that*]
by (*simp add*: *divide-inverse ac-simps*)
have *sum-list* (*map* (λ*xs*. *monomial-value* (*e* # *es*) (*x* # *xs*) /
grid-weight (*Xs* # *Xss*) (*x* # *xs*)) (*product-lists Xss*)) =
sum-list (*map* (λ*xs*. (*x* ^ *e* / *lagrange-denom* (*set Xs*) *x*) ∗
(*monomial-value es xs* / *grid-weight Xss xs*)) (*product-lists Xss*))
by (*simp add*: *term-eq*)
also have ... =
(*x* ^ *e* / *lagrange-denom* (*set Xs*) *x*) ∗
sum-list (*map* (λ*xs*. *monomial-value es xs* / *grid-weight Xss xs*) (*product-lists Xss*))

```
    by (rule sum-list-const-mult)
  finally show ?thesis .
 qed
 have mapped:
  map (λx. sum-list (map (λxs. monomial-value (e # es) (x # xs) /
      grid-weight (Xs # Xss) (x # xs)) (product-lists Xss))) Xs =
   map (λx. (x ^ e / lagrange-denom (set Xs) x) *
      sum-list (map (λxs. monomial-value es xs / grid-weight Xss xs)
       (product-lists Xss))) Xs
  using inner by auto
 have grid-monom-sum (Xs # Xss) (e # es) =
    sum-list (map (λx. sum-list (map (λxs. monomial-value (e # es) (x # xs) /
      grid-weight (Xs # Xss) (x # xs)) (product-lists Xss))) Xs)
  unfolding grid-monom-sum-def by (rule sum-list-product-lists-Cons)
 also have ... =
    sum-list (map (λx. x ^ e / lagrange-denom (set Xs) x) Xs) *
      grid-monom-sum Xss es
  using grid-monom-sum-def[of Xss es] mapped
       sum-list-mult-const[of - grid-monom-sum Xss es Xs]
  by presburger
 finally show ?thesis .
qed

lemma grid-monom-sum-Nil [simp]:
 grid-monom-sum [] [] = 1
 by (simp add: grid-monom-sum-def)

lemma monomial-value-wrong-length:
 length es ≠ length xs ⟹ monomial-value es xs = 0
proof (induction es arbitrary: xs)
 case Nil
 then show ?case
  by (cases xs) simp-all
next
 case (Cons e es xs)
 then show ?case
  by (cases xs) auto
qed

lemma grid-monom-sum-wrong-length:
 assumes length es ≠ length Xss
  shows grid-monom-sum Xss es = 0
proof −
 have zero: monomial-value es xs = 0 if xs ∈ set (product-lists Xss) for xs
  by (metis in-set-product-lists-length assms monomial-value-wrong-length that)
 show ?thesis
  unfolding grid-monom-sum-def
  by (rule sum-list-map-zero) (metis divide-eq-0-iff zero)
qed

lemma grid-monom-sum-delta:
 fixes Xss :: 'a::field list list
 assumes grids: list-all2 (λXs d. distinct Xs ∧ length Xs = Suc d) Xss ds
 assumes len: length es = length ds
 assumes deg: sum-list es ≤ sum-list ds
 shows grid-monom-sum Xss es = (if es = ds then 1 else 0)
 using grids len deg
proof (induction Xss arbitrary: ds es)
 case Nil
```

```
  then show ?case
    by (cases ds; cases es) simp-all
next
  case (Cons X Xss ds es)
  then obtain d ds' where ds: ds = d # ds'
    by (cases ds) auto
  then obtain e es' where es: es = e # es'
    using Cons.prems by (cases es) auto
  from Cons.prems ds es have X: distinct X length X = Suc d
    by auto
  from Cons.prems ds es have grids-tail:
    list-all2 (λXs d. distinct Xs ∧ length Xs = Suc d) Xss ds'
    and len-tail: length es' = length ds'
    and deg-all: e + sum-list es' ≤ d + sum-list ds'
    by auto
  show ?case
  proof (cases e d rule: linorder-cases)
    case less
    have head-delta: sum-list (map (λx. x ^ e / lagrange-denom (set X) x) X) =
        (if e = d then 1 else 0)
      using X less by (intro lagrange-power-sum-list) auto
    then have head: sum-list (map (λx. x ^ e / lagrange-denom (set X) x) X) = 0
      using less by simp
    show ?thesis
      using ds es less head X by (simp add: grid-monom-sum-Cons)
  next
    case equal
    have head-delta: sum-list (map (λx. x ^ e / lagrange-denom (set X) x) X) =
        (if e = d then 1 else 0)
      using X equal by (intro lagrange-power-sum-list) auto
    then have head: sum-list (map (λx. x ^ e / lagrange-denom (set X) x) X) = 1
      using equal by simp
    have tail-deg: sum-list es' ≤ sum-list ds'
      using deg-all equal by simp
    have tail: grid-monom-sum Xss es' = (if es' = ds' then 1 else 0)
      by (rule Cons.IH) (use grids-tail len-tail tail-deg in auto)
    show ?thesis
      using ds es equal head tail X by (simp add: grid-monom-sum-Cons)
  next
    case greater
    have tail-deg: sum-list es' ≤ sum-list ds'
      using deg-all greater by simp
    have tail-ne: es' ≠ ds'
    proof
      assume es' = ds'
      with deg-all greater show False
        by simp
    qed
    have tail: grid-monom-sum Xss es' = 0
      using Cons.IH[OF grids-tail len-tail tail-deg] tail-ne by auto
    show ?thesis
      using ds es greater tail X by (simp add: grid-monom-sum-Cons)
  qed
qed
```

The next lemma is the coefficient formula specialized to the sparse representation: the weighted sum of all grid evaluations extracts exactly the coefficient of the target exponent list.

```
lemma sum-list-sum:
  fixes f :: 'b ⇒ 'c ⇒ 'a::comm-monoid-add
```

```
  assumes finite A
  shows sum-list (map (λx. ∑ a∈A. f x a) xs) =
    (∑ a∈A. sum-list (map (λx. f x a) xs))
  using assms by (induction xs) (simp-all add: sum.distrib)

lemma eval-sparse-poly-grid-sum:
  fixes p :: nat list ⇒ 'a::field
  assumes sp: sparse-poly (length ds) p
  assumes grids: list-all2 (λXs d. distinct Xs ∧ length Xs = Suc d) Xss ds
  assumes deg: total-degree-le p (sum-list ds)
  shows sum-list (map (λxs. eval-sparse-poly p xs / grid-weight Xss xs) (product-lists Xss)) =
    p ds
proof −
  have fin: finite (support p)
    using sp by (simp add: sparse-poly-def)
  have len-Xss: length Xss = length ds
    using grids by (simp add: list-all2-lengthD)
  have sum-list (map (λxs. eval-sparse-poly p xs / grid-weight Xss xs) (product-lists Xss)) =
      sum-list (map (λxs. ∑ m∈support p.
        p m * (monomial-value m xs / grid-weight Xss xs)) (product-lists Xss))
    by (simp add: eval-sparse-poly-def sum-divide-distrib sum-distrib-left mult.assoc)
  also have ... =
      (∑ m∈support p.
        sum-list (map (λxs. p m * (monomial-value m xs / grid-weight Xss xs))
          (product-lists Xss)))
    using fin by (simp add: sum-list-sum)
  also have ... = (∑ m∈support p. p m * grid-monom-sum Xss m)
    by (intro sum.cong refl)
      (simp add: grid-monom-sum-def sum-list-const-mult divide-inverse ac-simps)
  also have ... = p ds
  proof −
    have delta: grid-monom-sum Xss m = (if m = ds then 1 else 0) if m ∈ support p for m
    proof −
      have length m = length ds
        using sp that by (simp add: sparse-poly-def)
      moreover have sum-list m ≤ sum-list ds
        using deg that by (simp add: total-degree-le-def)
      ultimately show ?thesis
        by (rule grid-monom-sum-delta[OF grids])
    qed
    show ?thesis
    proof (cases ds ∈ support p)
      case True
      have (∑ m∈support p. p m * grid-monom-sum Xss m) =
          p ds * grid-monom-sum Xss ds
        using fin True delta by (subst sum.remove[of support p ds]) auto
      also have ... = p ds
        using delta True by simp
      finally show ?thesis .
    next
      case False
      then have p0: p ds = 0
        by (simp add: support-def)
      have (∑ m∈support p. p m * grid-monom-sum Xss m) = 0
        using fin False delta by (intro sum.neutral) auto
      with p0 show ?thesis
        by simp
    qed
  qed
```

```
  finally show ?thesis .
qed

theorem combinatorial-nullstellensatz-exact-lists:
  fixes p :: nat list ⇒ 'a::field
  assumes sp: sparse-poly (length ds) p
  assumes deg: total-degree-le p (sum-list ds)
  assumes coeff: p ds ≠ 0
  assumes grids: list-all2 (λXs d. distinct Xs ∧ length Xs = Suc d) Xss ds
  shows ∃ xs∈set (product-lists Xss). eval-sparse-poly p xs ≠ 0
proof (rule ccontr)
  assume ¬ ?thesis
  then have zero: ⋀xs. xs ∈ set (product-lists Xss) ⟹ eval-sparse-poly p xs = 0
    by auto
  have sum-list (map (λxs. eval-sparse-poly p xs / grid-weight Xss xs) (product-lists Xss)) = 0
    by (rule sum-list-map-zero) (metis divide-eq-0-iff zero)
  moreover have sum-list (map (λxs. eval-sparse-poly p xs / grid-weight Xss xs) (product-lists Xss)) =
      p ds
    by (rule eval-sparse-poly-grid-sum[OF sp grids deg])
  ultimately show False
    using coeff by simp
qed
```

Finally, the standard "more than d_i points" formulation follows by selecting $d_i + 1$ points from each side of the grid.

```
definition exact-grid-sublists :: 'a list list ⇒ nat list ⇒ 'a list list where
  exact-grid-sublists Xss ds = map (λ(Xs, d). take (Suc d) Xs) (zip Xss ds)

lemma exact-grid-sublists-all2:
  assumes list-all2 (λXs d. distinct Xs ∧ length Xs > d) Xss ds
  shows list-all2 (λXs d. distinct Xs ∧ length Xs = Suc d)
    (exact-grid-sublists Xss ds) ds
  using assms
proof (induction Xss arbitrary: ds)
  case Nil
  then show ?case
    by (cases ds) (simp-all add: exact-grid-sublists-def)
next
  case (Cons X Xss ds)
  then obtain d ds' where ds: ds = d # ds'
    by (cases ds) auto
  with Cons.prems have distinct (take (Suc d) X) length (take (Suc d) X) = Suc d
    by auto
  moreover have list-all2 (λXs d. distinct Xs ∧ length Xs = Suc d)
      (exact-grid-sublists Xss ds') ds'
    using Cons.IH Cons.prems ds by auto
  ultimately show ?case
    by (simp add: exact-grid-sublists-def ds)
qed

lemma exact-grid-sublists-subset:
  assumes list-all2 (λXs d. length Xs > d) Xss ds
  shows set (product-lists (exact-grid-sublists Xss ds)) ⊆ set (product-lists Xss)
  using assms
proof (induction Xss arbitrary: ds)
  case Nil
  then show ?case
    by (cases ds) (simp-all add: exact-grid-sublists-def)
next
```

```
  case (Cons X Xss ds)
  then obtain d ds' where ds: ds = d # ds'
    by (cases ds) auto
  have tail: set (product-lists (exact-grid-sublists Xss ds')) ⊆ set (product-lists Xss)
    using Cons.IH Cons.prems ds by auto
  show ?case
  proof
    fix xs
    assume xs ∈ set (product-lists (exact-grid-sublists (X # Xss) ds))
    then obtain x ys where xs: xs = x # ys
      and x: x ∈ set (take (Suc d) X)
      and ys: ys ∈ set (product-lists (exact-grid-sublists Xss ds'))
      by (auto simp: exact-grid-sublists-def ds)
    from x have x ∈ set X
      by (rule in-set-takeD)
    moreover from tail ys have ys ∈ set (product-lists Xss)
      by auto
    ultimately show xs ∈ set (product-lists (X # Xss))
      using xs by auto
  qed
qed

theorem combinatorial-nullstellensatz-lists:
  fixes p :: nat list ⇒ 'a::field
  assumes sp: sparse-poly (length ds) p
  assumes deg: total-degree-le p (sum-list ds)
  assumes coeff: p ds ≠ 0
  assumes grids: list-all2 (λXs d. distinct Xs ∧ length Xs > d) Xss ds
  shows ∃xs∈set (product-lists Xss). eval-sparse-poly p xs ≠ 0
proof −
  have exact: list-all2 (λXs d. distinct Xs ∧ length Xs = Suc d)
      (exact-grid-sublists Xss ds) ds
    by (rule exact-grid-sublists-all2[OF grids])
  obtain xs where xs: xs ∈ set (product-lists (exact-grid-sublists Xss ds))
    and nz: eval-sparse-poly p xs ≠ 0
    using combinatorial-nullstellensatz-exact-lists[OF sp deg coeff exact] by blast
  have lengths: list-all2 (λXs d. length Xs > d) Xss ds
    by (rule list-all2-mono[OF grids]) auto
  have xs ∈ set (product-lists Xss)
    using exact-grid-sublists-subset[OF lengths] xs by auto
  with nz show ?thesis
    by blast
qed

end
```

References

[1] N. Alon. Combinatorial nullstellensatz. *Combinatorics, Probability and Computing*, 8(1–2):7–29, 1999. DOI: https://doi.org/10.1017/S0963548398003411.