The Sumcheck Protocol

Azucena Garvia, Christoph Sprenger and Jonathan Bootle

March 17, 2025

Abstract

The sumcheck protocol, first introduced in 1992, is an interactive proof which is a key component of many probabilistic proof systems in computational complexity theory and cryptography, some of which have been deployed. We provide a formally verified security analysis of the sumcheck protocol, following a general and modular approach.

First, we give a general formalization of public-coin interactive proofs. We then define a *generalized sumcheck protocol* for which we axiomatize the underlying mathematical structure and we establish its soundness and completeness. Finally, we prove that these axioms hold for multivariate polynomials, the original setting of the sumcheck protocol. Our modular analysis will facilitate formal verification of sumcheck instances based on different mathematical structures with little effort, by simply proving that these structures satisfy the axioms. Moreover, the analysis will encourage the development and formal verification of future probabilistic proof systems using the sumcheck protocol as a building block.

The paper presenting this formalization is to appear at CSF 2024 under the title "Formal Verification of the Sumcheck Protocol".

Contents

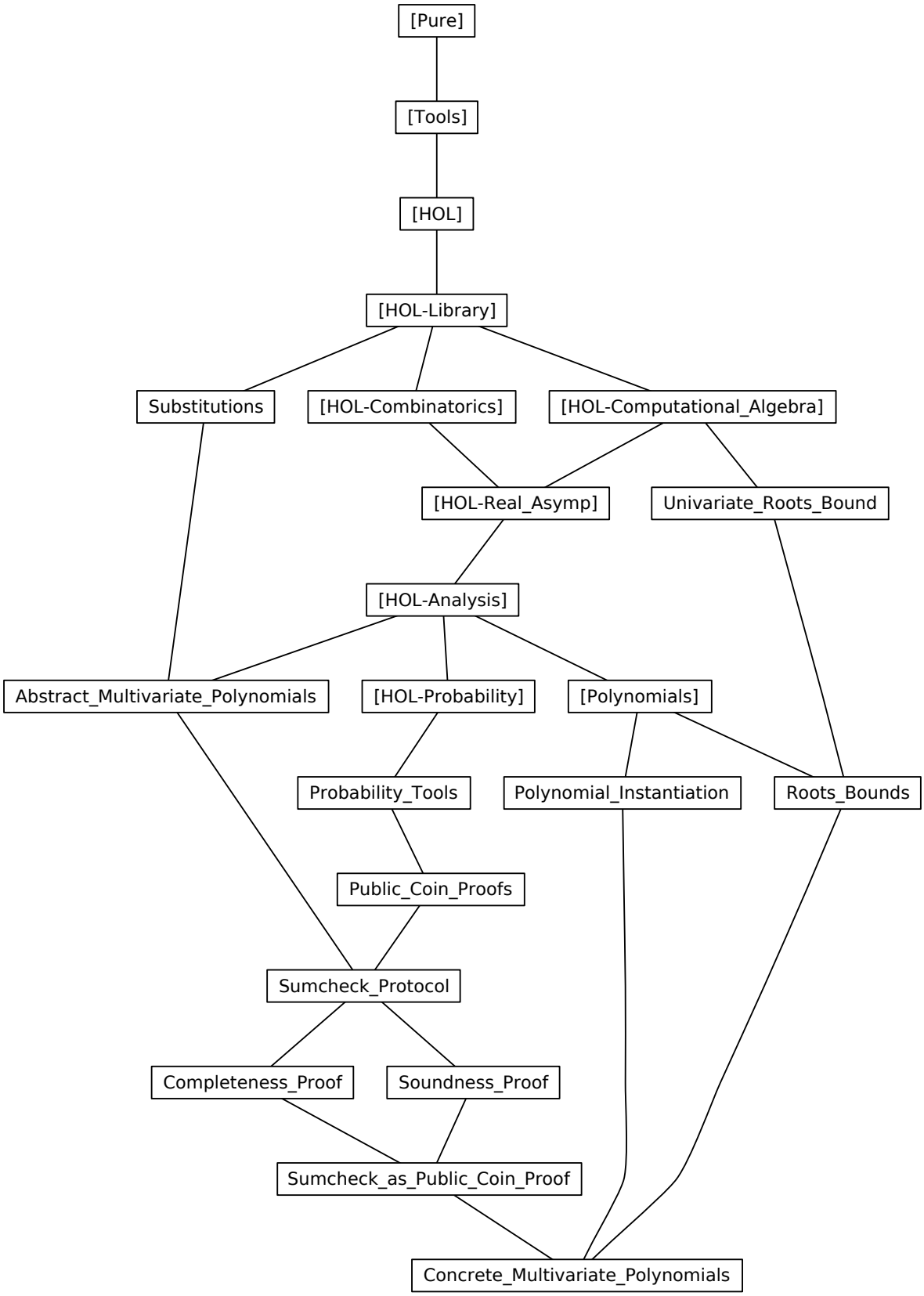

Figure 1: Theory dependencies

1 Auxiliary Lemmas Related to Probability Theory

theory *Probability-Tools*
imports *HOL−Probability.Probability*
begin

1.1 Tuples

definition *tuples* :: ‹*'a set* ⇒ *nat* ⇒ *'a list set*› **where**
‹*tuples S n* = {*xs*. *set xs* ⊆ *S* ∧ *length xs* = *n*}›

lemma *tuplesI*: ‹⟦ *set xs* ⊆ *S*; *length xs* = *n* ⟧ ⟹ *xs* ∈ *tuples S n*›
by (*simp add*: *tuples-def*)

lemma *tuplesE* [*elim*]: ‹⟦ *xs* ∈ *tuples S n*; ⟦ *set xs* ⊆ *S*; *length xs* = *n* ⟧ ⟹ *P* ⟧ ⟹ *P*›
by (*simp add*: *tuples-def*)

lemma *tuples-Zero*: ‹*tuples S 0* = {[]}›
by (*auto simp add*: *tuples-def*)

lemma *tuples-Suc*: ‹*tuples S* (*Suc n*) = (λ(*x*, *xs*). *x* # *xs*) ‘ (*S* × *tuples S n*)›
by (*fastforce simp add*: *tuples-def image-def Suc-length-conv dest*: *sym*)

lemma *tuples-non-empty* [*simp*]: ‹*S* ≠ {} ⟹ *tuples S n* ≠ {}›
by (*induction n*) (*auto simp add*: *tuples-Zero tuples-Suc*)

lemma *tuples-finite* [*simp*]: ‹⟦ *finite* (*S*::*'a set*); *S* ≠ {} ⟧ ⟹ *finite* (*tuples S n* :: *'a list set*)›
by (*auto simp add*: *tuples-def dest*: *finite-lists-length-eq*)

1.2 Congruence and monotonicity

lemma *prob-cong*: — adapted from Joshua
assumes ‹⋀*x*. *x* ∈ *set-pmf M* ⟹ *x* ∈ *A* ⟷ *x* ∈ *B*›
shows ‹*measure-pmf.prob M A* = *measure-pmf.prob M B*›
using *assms*
by (*simp add*: *measure-pmf.finite-measure-eq-AE AE-measure-pmf-iff*)

lemma *prob-mono*:
assumes ‹⋀*x*. *x* ∈ *set-pmf M* ⟹ *x* ∈ *A* ⟹ *x* ∈ *B*›
shows ‹*measure-pmf.prob M A* ≤ *measure-pmf.prob M B*›
using *assms*
by (*simp add*: *measure-pmf.finite-measure-mono-AE AE-measure-pmf-iff*)

1.3 Some simple derived lemmas

lemma *prob-empty*:
assumes ‹*A* = {}›
shows ‹*measure-pmf.prob M A* = *0*›
using *assms*
by (*simp*) — uses *measure-empty*: *Sigma-Algebra.measure ?M* {} = *0*

lemma *prob-pmf-of-set-geq-1*:
assumes *finite S* **and** *S* ≠ {}

shows *measure-pmf.prob* (*pmf-of-set S*) $A \geq 1 \longleftrightarrow S \subseteq A$ **using** *assms*
by (*auto simp add*: *measure-pmf.measure-ge-1-iff measure-pmf.prob-eq-1 AE-measure-pmf-iff*)

1.4 Intersection and union lemmas

lemma *prob-disjoint-union*:
assumes ‹$A \cap B = \{\}$›
shows ‹*measure-pmf.prob M* $(A \cup B)$ = *measure-pmf.prob M A* + *measure-pmf.prob M B*›
using *assms*
by (*fact measure-pmf.finite-measure-Union*[*simplified*])

lemma *prob-finite-Union*:
assumes ‹*disjoint-family-on A I*› ‹*finite I*›
shows ‹*measure-pmf.prob M* $(\bigcup i{\in}I.\ A\ i)$ = $(\sum i{\in}I.$ *measure-pmf.prob M* $(A\ i))$›
using *assms*
by (*intro measure-pmf.finite-measure-finite-Union*) (*simp-all*)

lemma *prob-disjoint-cases*:
assumes ‹$B \cup C = A$› ‹$B \cap C = \{\}$›
shows ‹*measure-pmf.prob M A* = *measure-pmf.prob M B* + *measure-pmf.prob M C*›
proof −
have ‹*measure-pmf.prob M A* = *measure-pmf.prob M* $(B \cup C)$› **using** ‹$B \cup C = A$›
by (*auto intro*: *prob-cong*)
also have ‹... = *measure-pmf.prob M B* + *measure-pmf.prob M C*› **using** ‹$B \cap C = \{\}$›
by (*simp add*: *prob-disjoint-union*)
finally show *?thesis* **.**
qed

lemma *prob-finite-disjoint-cases*:
assumes ‹$(\bigcup i{\in}I.\ B\ i) = A$› ‹*disjoint-family-on B I*› ‹*finite I*›
shows ‹*measure-pmf.prob M A* = $(\sum i{\in}I.$ *measure-pmf.prob M* $(B\ i))$›
proof −
have ‹*measure-pmf.prob M A* = *measure-pmf.prob M* $(\bigcup i{\in}I.\ B\ i)$› **using** *assms*(*1*)
by (*auto intro*: *prob-cong*)
also have ‹... = $(\sum i{\in}I.$ *measure-pmf.prob M* $(B\ i))$› **using** *assms*(*2*,*3*)
by (*intro prob-finite-Union*)
finally show *?thesis* **.**
qed

1.5 Independent probabilities for head and tail of a tuple

lemma *pmf-of-set-Times*: — by Andreas Lochbihler
pmf-of-set $(A \times B)$ = *pair-pmf* (*pmf-of-set A*) (*pmf-of-set B*)
if *finite A finite B* $A \neq \{\}$ $B \neq \{\}$
by(*rule pmf-eqI*)(*auto simp add*: *that pmf-pair indicator-def*)

lemma *prob-tuples-hd-tl-indep*:
assumes ‹$S \neq \{\}$›
shows
‹*measure-pmf.prob* (*pmf-of-set* (*tuples S* (*Suc n*))) $\{(r{::}'a{::}finite)\ \#\ rs \mid r\ rs.\ P\ r \wedge Q\ rs\}$
= *measure-pmf.prob* (*pmf-of-set* ($S{::}'a$ *set*)) $\{r.\ P\ r\}$ ∗
measure-pmf.prob (*pmf-of-set* (*tuples S n*)) $\{rs.\ Q\ rs\}$›

(**is** *?lhs = ?rhs*)
proof − — mostly by Andreas Lochbihler

Step 1: Split the random variable *pmf-of-set* (*tuples S* (*Suc n*)) into two independent (*pair-pmf*) random variables, one producing the head and one producing the tail of the list, and then (#) the two random variables using *map-pmf*.

have ∗: *pmf-of-set* (*tuples S* (*Suc n*))
= *map-pmf* (λ(*x* :: *'a*, *xs*). *x* # *xs*) (*pair-pmf* (*pmf-of-set S*) (*pmf-of-set* (*tuples S n*)))
unfolding *tuples-Suc* **using** ‹*S* ≠ {}›
by (*auto simp add*: *map-pmf-of-set-inj*[*symmetric*] *inj-on-def pmf-of-set-Times*)

Step 2: Transform the event by move the (#) from the random variable into the event. This corresponds to using *distr* on measures.

have *?lhs* = *measure-pmf.prob* (*pair-pmf* (*pmf-of-set S*) (*pmf-of-set* (*tuples S n*)))
((λ(*x* :: *'a*, *xs*). *x* # *xs*) −‘ {*r* # *rs* | *r rs*. *P r* ∧ *Q rs*})
unfolding ∗ *measure-map-pmf* **by** (*rule refl*)

Step 3: Rewrite the event as a pair of events. Then we apply independence of the head from the tail.

also have ((λ(*x*, *xs*). *x* # *xs*) −‘ {*r* # *rs* | *r rs*. *P r* ∧ *Q rs*}) = {*r*. *P r*} × {*rs*. *Q rs*} **by** *auto*
also have *measure-pmf.prob* (*pair-pmf* (*pmf-of-set S*) (*pmf-of-set* (*tuples S n*))) ... =
measure-pmf.prob (*pmf-of-set S*) {*r*. *P r*}
∗ *measure-pmf.prob* (*pmf-of-set* (*tuples S n*)) {*rs*. *Q rs*}
by(*rule measure-pmf-prob-product*) *simp-all*

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

lemma *prob-tuples-fixed-hd*:
‹*measure-pmf.prob* (*pmf-of-set* (*tuples UNIV* (*Suc n*))) {*rs*::*'a list*. *P rs*}
= (∑ *a* ∈ *UNIV*. *measure-pmf.prob* (*pmf-of-set* (*tuples UNIV n*)) {*rs*. *P* (*a* # *rs*)}) / *real*(*CARD*(*'a*::*finite*))›
(**is** *?lhs = ?rhs*)
proof −
{
fix *a*
have ‹*measure-pmf.prob* (*pmf-of-set* (*tuples UNIV* (*Suc n*))) ({*rs*. *P rs*} ∩ {*rs*. *hd rs* = *a*})
= *measure-pmf.prob* (*pmf-of-set* (*tuples UNIV* (*Suc n*))) ({*r*#*rs* | *r rs*. *r* = *a* ∧ *P* (*a*#*rs*)})›
by (*intro prob-cong*) (*auto simp add*: *tuples-Suc*)
also have ‹... = *measure-pmf.prob* (*pmf-of-set* (*UNIV*::*'a set*)) {*r*. *r* = *a*} ∗
measure-pmf.prob (*pmf-of-set* (*tuples UNIV n*)) {*rs*. *P* (*a*#*rs*)}›
by (*intro prob-tuples-hd-tl-indep*) *simp*
also have ‹... = *measure-pmf.prob* (*pmf-of-set* (*tuples UNIV n*)) {*rs*. *P* (*a*#*rs*)} / *real* (*CARD* (*'a*))›
by (*simp add*: *measure-pmf-single*)
finally
have ‹*measure-pmf.prob* (*pmf-of-set* (*tuples UNIV* (*Suc n*))) ({*rs*. *P rs*} ∩ {*rs*. *hd rs* = *a*})
= *measure-pmf.prob* (*pmf-of-set* (*tuples UNIV n*)) {*rs*. *P* (*a*#*rs*)} / *real* (*CARD* (*'a*))› **.**
}
note *A1* = *this*

have ‹*?lhs* = (∑ *a* ∈ *UNIV*. *measure-pmf.prob* (*pmf-of-set* (*tuples UNIV* (*Suc n*))) ({*rs*. *P rs*} ∩ {*rs*. *hd rs* = *a*}))›

by (*intro prob-finite-disjoint-cases*) (*auto simp add*: *disjoint-family-on-def*)
also have ‹... = *?rhs*› **using** *A1*
by (*simp add*: *sum-divide-distrib*)
finally show *?thesis* **.**
qed

end

2 Generic Public-coin Interactive Proofs

theory *Public-Coin-Proofs*
imports *Probability-Tools*
begin

2.1 Generic definition

type-synonym $('i, 'r, 'a, 'resp, 'ps)\ prv = 'i \Rightarrow 'a \Rightarrow 'a\ list \Rightarrow 'r \Rightarrow 'ps \Rightarrow 'resp \times 'ps$

locale *public-coin-proof* =
fixes $ver0 :: 'i \Rightarrow 'vs \Rightarrow bool$
and $ver1 :: 'i \Rightarrow 'resp \Rightarrow 'r \Rightarrow 'a \Rightarrow 'a\ list \Rightarrow 'vs \Rightarrow bool \times 'i \times 'vs$
begin

fun $prove :: 'vs \Rightarrow ('i, 'r, 'a, 'resp, 'ps)\ prv \Rightarrow 'ps \Rightarrow 'i \Rightarrow 'r \Rightarrow ('a \times 'r)\ list \Rightarrow bool$ **where**
$prove\ vs\ prv\ ps\ I\ r\ [] \longleftrightarrow ver0\ I\ vs\ |$
$prove\ vs\ prv\ ps\ I\ r\ ((x, r')\#rm) \longleftrightarrow$
$(let\ (resp, ps') = prv\ I\ x\ (map\ fst\ rm)\ r\ ps\ in$
$let\ (ok, I', vs') = ver1\ I\ resp\ r'\ x\ (map\ fst\ rm)\ vs\ in$
$ok \wedge prove\ vs'\ prv\ ps'\ I'\ r'\ rm)$

The parameters are

- $(ver0, ver1)$ and vs are the verifier and its current state,
- prv and ps are the prover and its current state,
- $I \in S$ is the problem instance,
- r is the verifier's randomness for the current round.
- rs is the (list of) randomness for the remaining rounds, and
- xs is a list of public per-round information/

We assume that rs and xs have the same length.

end

2.2 Generic soundness and completeness

locale *public-coin-proof-security* =
public-coin-proof ver0 ver1
for $ver0 :: 'i \Rightarrow 'vs \Rightarrow bool$
and $ver1 :: 'i \Rightarrow 'resp \Rightarrow 'r \Rightarrow 'a \Rightarrow 'a\ list \Rightarrow 'vs \Rightarrow bool \times 'i \times 'vs +$
fixes $S :: 'i\ set$ — problem specification
and $honest\text{-}pr :: ('i, 'r, 'a, 'resp, 'ps)\ prv$
and $compl\text{-}err :: 'i \Rightarrow real$
and $sound\text{-}err :: 'i \Rightarrow real$
and $compl\text{-}assm :: 'vs \Rightarrow 'ps \Rightarrow 'i \Rightarrow 'a\ list \Rightarrow bool$
and $sound\text{-}assm :: 'vs \Rightarrow 'ps \Rightarrow 'i \Rightarrow 'a\ list \Rightarrow bool$
assumes
completeness:
$[\![I \in S;\ compl\text{-}assm\ vs\ ps\ I\ xs]\!] \Longrightarrow$

measure-pmf.prob
(*pmf-of-set* (*tuples UNIV* (*length xs*)))
{*rs. prove vs honest-pr ps I r* (*zip xs rs*)} ≥ *1 − compl-err I* **and**

soundness:
⟦ *I* ∉ *S*; *sound-assm vs ps I xs* ⟧ ⟹
measure-pmf.prob
(*pmf-of-set* (*tuples UNIV* (*length xs*)))
{*rs. prove vs pr ps I r* (*zip xs rs*)} ≤ *sound-err I*

locale *public-coin-proof-strong-props* =
public-coin-proof ver0 ver1
for *ver0* :: *'i* ⇒ *'vs* ⇒ *bool*
and *ver1* :: *'i* ⇒ *'resp* ⇒ *'r*::*finite* ⇒ *'a* ⇒ *'a list* ⇒ *'vs* ⇒ *bool* × *'i* × *'vs* +
fixes *S* :: *'i set* — problem specification
and *honest-pr* :: (*'i*, *'r*, *'a*, *'resp*, *'ps*) *prv*
and *sound-err* :: *'i* ⇒ *real*
and *compl-assm* :: *'vs* ⇒ *'ps* ⇒ *'i* ⇒ *'a list* ⇒ *bool*
and *sound-assm* :: *'vs* ⇒ *'ps* ⇒ *'i* ⇒ *'a list* ⇒ *bool*
assumes
completeness:
⟦ *I* ∈ *S*; *compl-assm vs ps I* (*map fst rm*) ⟧ ⟹ *prove vs honest-pr ps I r rm* **and**

soundness:
⟦ *I* ∉ *S*; *sound-assm vs ps I xs* ⟧ ⟹
measure-pmf.prob
(*pmf-of-set* (*tuples UNIV* (*length xs*)))
{*rs. prove vs pr ps I r* (*zip xs rs*)} ≤ *sound-err I*
begin

Show that this locale satisfies the weaker assumptions of *public-coin-proof-security*.

sublocale *pc-props*:
public-coin-proof-security ver0 ver1 S honest-pr λ*-. 0 sound-err compl-assm sound-assm*
by (*unfold-locales*)
(*fastforce simp add*: *prob-pmf-of-set-geq-1 tuples-Suc completeness*,
clarsimp simp add: *soundness*)

end

end

3 Substitutions

theory *Substitutions*
imports
Main
HOL−Library.FuncSet
begin

type-synonym $('v, 'a)$ *subst* $= 'v \rightharpoonup 'a$

definition *substs* :: $'v$ *set* $\Rightarrow$ $'a$ *set* $\Rightarrow$ $('v, 'a)$ *subst set* **where**
substs $V\ H = \{\sigma.\ dom\ \sigma = V \wedge ran\ \sigma \subseteq H\}$

Small lemmas about the set of substitutions

lemma *substE* [*elim*]: $[\![\ \sigma \in substs\ V\ H;\ [\![\ dom\ \sigma = V;\ ran\ \sigma \subseteq H\ ]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
by (*simp add*: *substs-def*)

lemma *substs-empty-dom* [*simp*]: *substs* $\{\}\ H = \{Map.empty\}$
by (*auto simp add*: *substs-def*)

lemma *substs-finite*: $[\![\ finite\ V;\ finite\ H\ ]\!] \Longrightarrow finite\ (substs\ V\ H)$
by (*simp add*: *finite-set-of-finite-maps substs-def*)

lemma *substs-nonempty*:
assumes $H \neq \{\}$
shows *substs* $V\ H \neq \{\}$
proof −
obtain h **where** *A1*: $h \in H$ **using** *assms* **by**(*auto*)
obtain ϱ **where** *A2*: $\varrho = (\lambda v.\ if\ v \in V\ then\ Some\ h\ else\ None)$ **by**(*simp*)
have $\varrho \in substs\ V\ H$ **using** *A1 A2* **by**(*auto simp add*: *substs-def ran-def dom-def*)
then show *?thesis* **by**(*auto*)
qed

lemma *subst-dom*: ‹$[\![\ \varrho \in substs\ V\ H;\ x \notin V\ ]\!] \Longrightarrow x \notin dom\ \varrho$›
by(*auto simp add*: *substs-def*)

lemma *subst-add*:
assumes $x \in V$ **and** $\varrho \in substs\ (V - \{x\})\ H$ **and** $a \in H$
shows $\varrho(x \mapsto a) \in substs\ V\ H$
using *assms*
by(*simp add*: *substs-def*)
(*auto simp add*: *dom-def ran-def*)

lemma *subst-im*:
assumes $x \in V$ **and** $\varrho \in substs\ V\ H$
shows *the* $(\varrho\ x) \in H$
using *assms*
by(*auto simp add*: *substs-def dom-def ran-def*)

lemma *subst-restr*:
assumes $x \in V$ **and** $\varrho \in substs\ V\ H$
shows $\varrho\ |`\ (dom\ \varrho - \{x\}) \in substs\ (V - \{x\})\ H$
using *assms*

by(*auto simp add*: *substs-def ran-def dom-def restrict-map-def*)

Bijection between sets of substitutions

lemma *restrict-map-dom*: $\sigma \mid` dom\ \sigma = \sigma$
by (*metis* (*no-types*, *lifting*) *domIff map-le-antisym map-le-def restrict-in restrict-out*)

lemma *bij-betw-set-substs*:
assumes $x \in V$
defines $f \equiv \lambda(a, \sigma::'v \rightharpoonup 'a).\ \sigma(x \mapsto a)$
and $g \equiv \lambda\vartheta::'v \rightharpoonup 'a.\ (the\ (\vartheta\ x), \vartheta \mid` (dom\ \vartheta - \{x\}))$
shows *bij-betw f*
$(H \times substs\ (V - \{x\})\ H)$
$(substs\ V\ H)$
proof (*intro bij-betwI*)
show $f \in H \times substs\ (V - \{x\})\ H \to substs\ V\ H$
using *assms*
by(*auto simp add*: *f-def subst-add*)
next
show $g \in substs\ V\ H \to H \times substs\ (V - \{x\})\ H$
using *assms*
by(*auto simp add*: *g-def subst-im subst-restr*)
next
fix *xa*
assume $xa \in H \times substs\ (V - \{x\})\ H$
then show $g\ (f\ xa) = xa$ **using** *assms*
by (*smt* (*verit*, *ccfv-threshold*) *Diff-insert-absorb SigmaE case-prod-conv domI*
fun-upd-None-restrict fun-upd-same fun-upd-upd mk-disjoint-insert option.sel restrict-map-dom
subst-dom)
next
fix *y*
assume $y \in substs\ V\ H$
then show $f\ (g\ y) = y$ **using** *assms*
by(*auto simp add*: *g-def f-def*)
(*metis domIff fun-upd-restrict map-upd-triv option.exhaust-sel restrict-map-dom substE*)
qed

end

4 Abstract Multivariate Polynomials

theory *Abstract-Multivariate-Polynomials*
imports
Substitutions
HOL−Analysis.Finite-Cartesian-Product
begin

Multivariate polynomials, abstractly

locale *multi-variate-polynomial* =
fixes *vars* :: ‹'*p* :: *comm-monoid-add* ⇒ '*v set*›
and *deg* :: ‹'*p* ⇒ *nat*›
and *eval* :: ‹'*p* ⇒ ('*v*, '*a*::*finite*) *subst* ⇒ '*b* :: *comm-monoid-add*›
and *inst* :: ‹'*p* ⇒ ('*v*, '*a*) *subst* ⇒ '*p*›
assumes
— vars
vars-finite: ‹*finite* (*vars p*)› **and**
vars-zero: ‹*vars 0* = {}› **and**
vars-add: ‹*vars* (*p* + *q*) ⊆ *vars p* ∪ *vars q*› **and**
vars-inst: ‹*vars* (*inst p* σ) ⊆ *vars p* − *dom* σ› **and**

— degree
deg-zero: ‹*deg 0* = *0*› **and**
deg-add: ‹*deg* (*p* + *q*) ≤ *max* (*deg p*) (*deg q*)› **and**
deg-inst: ‹*deg* (*inst p* ϱ) ≤ *deg p*› **and**

— eval
eval-zero: ‹*eval 0* σ = *0*› **and**
eval-add: ‹*vars p* ∪ *vars q* ⊆ *dom* σ ⟹ *eval* (*p* + *q*) σ = *eval p* σ + *eval q* σ› **and**
eval-inst: ‹*vars p* ⊆ *dom* σ ∪ *dom* ϱ ⟹ *eval* (*inst p* σ) ϱ = *eval p* (ϱ ++ σ)› **and**

— small number of roots (variant for two polynomials)
roots: ‹*card* {*r*. *deg p* ≤ *d* ∧ *vars p* ⊆ {*x*} ∧ *deg q* ≤ *d* ∧ *vars q* ⊆ {*x*} ∧
p ≠ *q* ∧ *eval p* [*x* ↦ *r*] = *eval q* [*x* ↦ *r*]} ≤ *d*›
begin

lemmas *vars-addD* = *vars-add*[*THEN subsetD*]

4.1 Arity: definition and some lemmas

definition *arity* :: ‹'*p* ⇒ *nat*› **where**
‹*arity p* = *card* (*vars p*)›

lemma *arity-zero*: ‹*arity 0* = *0*›
by (*simp add*: *arity-def vars-zero*)

lemma *arity-add*: ‹*arity* (*p* + *q*) ≤ *arity p* + *arity q*›
proof −
have ‹*card* (*vars* (*p* + *q*)) ≤ *card* (*vars p* ∪ *vars q*)›
by (*intro card-mono*) (*auto simp add*: *vars-add vars-finite*)
also have ‹... ≤ *card* (*vars p*) + *card* (*vars q*)› **by** (*simp add*: *card-Un-le*)
finally show *?thesis* **by** (*simp add*: *arity-def*)
qed

lemma *arity-inst*:
assumes ‹*dom* $\sigma \subseteq$ *vars* p›
shows ‹*arity* (*inst* p σ) $\leq$ *arity* p − *card* (*dom* σ)›
proof −
have ‹*card* (*vars* (*inst* p σ)) $\leq$ *card* (*vars* p − *dom* σ)›
by (*auto simp add*: *vars-finite vars-inst card-mono*)
also have ‹... = *card* (*vars* p) − *card* (*dom* σ)› **using** *assms*
by (*simp add*: *card-Diff-subset finite-subset vars-finite*)
finally show *?thesis* **by** (*simp add*: *arity-def*)
qed

4.2 Lemmas about evaluation, degree, and variables of finite sums

lemma *eval-sum*:
assumes ‹*finite* I› ‹$\bigwedge i.\ i \in I \Longrightarrow$ *vars* (*pp* i) $\subseteq$ *dom* σ›
shows ‹*eval* ($\sum i{\in}I.$ *pp* i) σ = ($\sum i{\in}I.$ *eval* (*pp* i) σ)›
proof −
have ‹*eval* ($\sum i{\in}I.$ *pp* i) σ = ($\sum i{\in}I.$ *eval* (*pp* i) σ) $\wedge$ *vars* ($\sum i{\in}I.$ *pp* i) $\subseteq$ *dom* σ› **using** *assms*
proof (*induction rule*: *finite.induct*)
case *emptyI*
then show *?case* **by** (*simp add*: *eval-zero vars-zero*)
next
case (*insertI A a*)
then show *?case*
by (*auto simp add*: *eval-add vars-add sum.insert-if dest*!: *vars-addD*)
qed
then show *?thesis* **..**
qed

lemma *vars-sum*:
assumes ‹*finite* I›
shows ‹*vars* ($\sum i{\in}I.$ *pp* i) $\subseteq$ ($\bigcup i{\in}I.$ *vars* (*pp* i))›
using *assms*
proof (*induction rule*: *finite.induct*)
case *emptyI*
then show *?case* **by**(*auto simp add*: *vars-zero*)
next
case (*insertI A a*)
then show *?case* **using** *insertI* **by**(*auto simp add*: *sum.insert-if dest*: *vars-addD*)
qed

lemma *deg-sum*:
assumes ‹*finite* I› **and** $I \neq \{\}$
shows ‹*deg* ($\sum i{\in}I.$ *pp* i) $\leq$ *Max* {*deg* (*pp* i) | $i.\ i \in I$}›
using *assms*
proof (*induction rule*: *finite.induct*)
case *emptyI*
then show *?case* **by**(*auto simp add*: *deg-zero*)
next
case (*insertI A a*)
show *?case*
proof(*cases* $A = \{\}$)

assume ‹$A = \{\}$›
then show *?thesis* **by**(*simp*)
next
assume ‹$A \neq \{\}$›
then have $*$: ‹$Max\ \{deg\ (pp\ i)\ |i.\ i \in A\} \leq Max\ \{deg\ (pp\ i)\ |i.\ i = a \vee i \in A\}$› **using** ‹*finite A*›
by (*intro Max-mono*) *auto*
show *?thesis* **using** *insertI* ‹$A \neq \{\}$›
by (*auto 4 4 simp add*: *sum.insert-if intro*: *Max-ge* $*$[*THEN* [*2*] *le-trans*] *deg-add*[*THEN le-trans*])
qed
qed

4.3 Lemmas combining eval, sum, and inst

lemma *eval-sum-inst*:
assumes ‹*vars p* $\subseteq$ *V* $\cup$ *dom* ϱ› ‹*finite V*›
shows ‹*eval* $(\sum \sigma \in substs\ V\ H.\ inst\ p\ \sigma)\ \varrho = (\sum \sigma \in substs\ V\ H.\ eval\ p\ (\varrho\ ++\ \sigma))$›
proof −
have *A1*: ‹$\bigwedge \sigma.\ \sigma \in substs\ V\ H \Longrightarrow vars\ (inst\ p\ \sigma) \subseteq dom\ \varrho$› **using** *assms*(*1*) *vars-inst* **by** *blast*
have *A2*: ‹$\bigwedge \sigma.\ \sigma \in substs\ V\ H \Longrightarrow vars\ p \subseteq dom\ \sigma \cup dom\ \varrho$› **using** *assms*(*1*) **by** (*auto*)

have ‹$eval\ (\sum \sigma \in substs\ V\ H.\ inst\ p\ \sigma)\ \varrho = (\sum \sigma \in substs\ V\ H.\ eval\ (inst\ p\ \sigma)\ \varrho)$› **using** *A1* *assms*(*2*)
by (*simp add*: *eval-sum substs-finite*) — requires *finite H*
also have ‹$... = (\sum \sigma \in substs\ V\ H.\ eval\ p\ (\varrho\ ++\ \sigma))$› **using** *A2*
by (*simp add*: *eval-inst*)
finally show *?thesis* **.**
qed

lemma *eval-sum-inst-commute*:
assumes ‹*vars p* $\subseteq$ *insert x V*› ‹$x \notin V$› ‹*finite V*›
shows ‹$eval\ (\sum \sigma \in substs\ V\ H.\ inst\ p\ \sigma)\ [x \mapsto r]$
$= (\sum \sigma \in substs\ V\ H.\ eval\ (inst\ p\ [x \mapsto r])\ \sigma)$›
proof −
have ‹$eval\ (sum\ (inst\ p)\ (substs\ V\ H))\ [x \mapsto r] =$
$(\sum \sigma \in substs\ V\ H.\ eval\ p\ ([x \mapsto r]\ ++\ \sigma))$› **using** ‹*vars p* $\subseteq$ *insert x V*› ‹*finite V*›
by (*simp add*: *eval-sum-inst*)
also have ‹$... = (\sum \sigma \in substs\ V\ H.\ eval\ p\ (\sigma(x \mapsto r)))$› **using** ‹$x \notin V$›
by (*intro Finite-Cartesian-Product.sum-cong-aux*)
(*auto simp add*: *map-add-comm subst-dom*)
also have ‹$... = (\sum \sigma \in substs\ V\ H.\ eval\ (inst\ p\ [x \mapsto r])\ \sigma)$› **using** ‹*vars p* $\subseteq$ *insert x V*›
by (*intro Finite-Cartesian-Product.sum-cong-aux*)
(*auto simp add*: *eval-inst*)
finally show *?thesis* **.**
qed

4.4 Merging sums over substitutions

lemma *sum-merge*:
assumes ‹$x \notin V$›
shows $(\sum h \in H.\ (\sum \sigma \in substs\ V\ H.\ eval\ p\ ([x \mapsto h]\ ++\ \sigma)))$
$= (\sum \sigma \in substs\ (insert\ x\ V)\ H.\ eval\ p\ \sigma)$
proof −
have $\bigwedge h\ \sigma.\ (h,\sigma) \in H \times substs\ V\ H \Longrightarrow dom\ [x \mapsto h] \cap dom\ \sigma = \{\}$ **using** ‹$x \notin V$›

```
  by(auto simp add: substs-def)
 then have *: ⋀h σ. (h,σ) ∈ H × substs V H ⟹ [x ↦ h] ++ σ = σ(x ↦ h)
  by(auto simp add: map-add-comm)

 have (∑ h∈H. (∑ σ ∈ substs V H. eval p ([x ↦ h] ++ σ))) =
       (∑ (h,σ) ∈ H × substs V H. eval p ([x ↦ h] ++ σ))
  by(auto simp add: sum.cartesian-product)
 also have ... = (∑ (h,σ) ∈ H × substs V H. eval p (σ(x ↦ h))) using *
  by (intro Finite-Cartesian-Product.sum-cong-aux) (auto)
 also have ... =  (∑ σ∈substs (insert x V) H. eval p σ) using ‹x ∉ V›
  by(auto simp add: sum.reindex-bij-betw[OF bij-betw-set-substs,
          where V1 = insert x V and x1 = x and H1 = H and g = λσ. eval p σ, symmetric]
        intro: Finite-Cartesian-Product.sum-cong-aux)
 finally show ?thesis .
qed

end

end
```

5 Sumcheck Protocol

theory *Sumcheck-Protocol*
  imports
    Public-Coin-Proofs
    Abstract-Multivariate-Polynomials
begin

5.1 The sumcheck problem

Type of sumcheck instances

type-synonym $('p, 'a, 'b)$ *sc-inst* $= 'a$ *set* $\times$ $'p \times 'b$

definition (**in** *multi-variate-polynomial*)
  Sumcheck :: $('p, 'a, 'b)$ *sc-inst set* **where**
  Sumcheck $= \{(H, p, v) \mid H\ p\ v.\ v = (\sum \sigma \in substs\ (vars\ p)\ H.\ eval\ p\ \sigma)\}$

5.2 The sumcheck protocol

Type of the prover

type-synonym $('p, 'a, 'b, 'v, 's)$ *prover* $= (('p, 'a, 'b)$ *sc-inst*$, 'a, 'v, 'p, 's)$ *prv*

Here is how the expanded type lools like

$('p, 'a, 'b, 'v, 's)$ *prover*

.

context *multi-variate-polynomial* **begin**

Sumcheck function

fun *sumcheck* :: $('p, 'a, 'b, 'v, 's)$ *prover* $\Rightarrow$ $'s \Rightarrow ('p, 'a, 'b)$ *sc-inst* $\Rightarrow$ $'a \Rightarrow ('v \times 'a)$ *list* $\Rightarrow$ *bool*
where
  sumcheck pr s (H, p, v) *r-prev* $[] \longleftrightarrow v =$ *eval p Map.empty*
| *sumcheck pr s* (H, p, v) *r-prev* $((x, r) \#\ rm) \longleftrightarrow$
    (*let* $(q, s') =$ *pr* (H, p, v) *x* (*map fst rm*) *r-prev s in*
      vars $q \subseteq \{x\} \wedge$ *deg* $q \leq$ *deg* $p \wedge$
      $v = (\sum y \in H.\ eval\ q\ [x \mapsto y]) \wedge$
      sumcheck pr s' $(H,$ *inst* $p\ [x \mapsto r],$ *eval* $q\ [x \mapsto r])$ *r rm*)

Honest prover definition

fun *honest-prover* :: $('p, 'a, 'b, 'v,$ *unit*$)$ *prover* **where**
  honest-prover $(H, p,$ -$)$ *x xs* - - $= (\sum \sigma \in substs\ (set\ xs)\ H.\ inst\ p\ \sigma, ())$

declare *honest-prover.simps* [*simp del*]
lemmas *honest-prover-def* $=$ *honest-prover.simps*

Lemmas on variables and degree of the honest prover.

lemma *honest-prover-vars*:
  assumes *vars* $p \subseteq$ *insert x V finite V H* $\neq \{\}$ *finite H*
  shows *vars* $(\sum \sigma \in substs\ V\ H.\ inst\ p\ \sigma) \subseteq \{x\}$
proof −

```
  have *: ⋀σ. σ ∈ substs V H ⟹ vars (inst p σ) ⊆ {x} using assms
    by (metis (no-types, lifting) Diff-eq-empty-iff Diff-insert subset-iff substE vars-inst)

  have vars (sum (inst p) (substs V H)) ⊆ (⋃σ∈substs V H. vars (inst p σ))
    using ‹finite V› ‹finite H›
    by (auto simp add: vars-sum substs-finite)
  also have ... ⊆ {x} using ‹H ≠ {}› *
    by (auto simp add: substs-nonempty vars-finite substs-finite)
  finally show ?thesis .
qed

lemma honest-prover-deg:
  assumes H ≠ {} finite V
  shows deg (∑σ∈substs V H. inst p σ) ≤ deg p
proof −
  have deg (∑σ∈substs V H. inst p σ) ≤ Max {deg (inst p σ) |σ. σ ∈ substs V H}
    by(auto simp add: substs-finite substs-nonempty deg-sum assms)
  also have ... ≤ deg p
    by(auto simp add: substs-finite substs-nonempty deg-inst assms)
  finally show ?thesis .
qed
```

5.3 The sumcheck protocol as a public-coin proof instance

Define verifier functions

```
fun sc-ver0 :: ('p, 'a, 'b) sc-inst ⇒ unit ⇒ bool where
   sc-ver0 (H, p, v) () ⟷ v = eval p Map.empty

fun sc-ver1 ::
  ('p, 'a, 'b) sc-inst ⇒ 'p ⇒ 'a ⇒ 'v ⇒ 'v list ⇒ unit ⇒ bool × ('p, 'a, 'b) sc-inst × unit
where
   sc-ver1 (H, p, v) q r x - () = (
      vars q ⊆ {x} ∧ deg q ≤ deg p ∧ v = (∑ y ∈ H. eval q [x ↦ y]),
      (H, inst p [x ↦ r], eval q [x ↦ r]),
      ()
   )
```

sublocale *sc*: *public-coin-proof sc-ver0 sc-ver1* .

Equivalence of *sumcheck* with public-coin proofs instance

```
lemma prove-sc-eq-sumcheck:
  ‹sc.prove () pr ps (H, p, v) r rm = sumcheck pr ps (H, p, v) r rm›
proof (induction () pr ps (H, p, v) r rm arbitrary: p v rule: sc.prove.induct)
  case (1 vs prv ps r)
  then show ?case by (simp)
next
  case (2 vs prv ps r r' rs x xs)
  then show ?case by (simp split:prod.split)
qed


end
```

end

6 Completeness Proof for the Sumcheck Protocol

theory *Completeness-Proof*
imports
Sumcheck-Protocol
begin

context *multi-variate-polynomial* **begin**

Completeness proof

theorem *completeness-inductive*:
assumes
‹$v = (\sum \sigma \in$ *substs* (*set* (*map fst rm*)) H. *eval p* σ)›
‹*vars p* $\subseteq$ *set* (*map fst rm*)›
‹*distinct* (*map fst rm*)›
‹$H \neq \{\}$›
shows
sumcheck honest-prover u (H, *p*, *v*) *r-prev rm*
using *assms*
proof(*induction honest-prover u* (H, *p*, *v*) *r-prev rm arbitrary*: *H p v rule*: *sumcheck.induct*)
case (*1 s H p v r-prev*)
then show *?case* **by**(*simp*)
next
case (*2 s H p v r-prev x r rm*)

note *IH* = *2*(*1*) — induction hypothesis

let *?V* = *set* (*map fst rm*)
let *?q* = ($\sum \sigma \in$ *substs ?V H*. *inst p* σ)

have ‹*vars p* $\subseteq$ *insert x ?V*› ‹$x \notin$ *?V*› ‹*distinct* (*map fst rm*)›
using *2*(*3*−*4*) **by**(*auto*)

— evaluation check
have ‹($\sum \sigma \in$ *substs* (*insert x ?V*) H. *eval p* σ) = ($\sum h \in H$. *eval ?q* [$x \mapsto h$])›
proof −
have ($\sum \sigma \in$ *substs* (*insert x ?V*) H. *eval p* σ) =
($\sum h \in H$. ($\sum \sigma \in$ *substs ?V H*. *eval p* ([$x \mapsto h$] ++ σ)))
using ‹$x \notin$ *?V*›
by(*auto simp add*: *sum-merge*)
also have ... = ($\sum h \in H$. *eval ?q* [$x \mapsto h$])
using ‹*vars p* $\subseteq$ *insert x ?V*›
by(*auto simp add*: *eval-sum-inst*)
finally show *?thesis* **.**
qed
moreover
— recursive check
have ‹*sumcheck honest-prover* () (H, *inst p* [$x \mapsto r$], *eval ?q* [$x \mapsto r$]) *r rm*›
proof −
have ‹*vars* (*inst p* [$x \mapsto r$]) $\subseteq$ *?V*›
using ‹*vars p* $\subseteq$ *insert x ?V*› *vars-inst* **by** *fastforce*
moreover
have *eval ?q* [$x \mapsto r$] = ($\sum \sigma \in$ *substs ?V H*. *eval* (*inst p* [$x \mapsto r$]) σ)

using ‹*vars p* ⊆ *insert x ?V*› ‹*x* ∉ *set* (*map fst rm*)›
by (*auto simp add*: *eval-sum-inst-commute*)
ultimately
show *?thesis* **using** *IH* ‹*distinct* (*map fst rm*)› ‹*H* ≠ {}›
by (*simp add*: *honest-prover-def*)
qed
ultimately show *?case* **using** *2*(*2*−*3*,*5*)
by (*simp add*: *honest-prover-def honest-prover-vars honest-prover-deg*)
qed

corollary *completeness*:
assumes
‹(*H*, *p*, *v*) ∈ *Sumcheck*›
‹*vars p* = *set* (*map fst rm*)›
‹*distinct* (*map fst rm*)›
‹*H* ≠ {}›
shows
sumcheck honest-prover u (*H*, *p*, *v*) *r rm*
using *assms*
by (*auto simp add*: *Sumcheck-def intro*: *completeness-inductive*)

end

end

7 Soundness Proof for the Sumcheck Protocol

theory *Soundness-Proof*
imports
Probability-Tools
Sumcheck-Protocol
begin

context *multi-variate-polynomial* **begin**

— Helper lemma: Proves that the probability of two different polynomials evaluating to the same value is small.
lemma *prob-roots*:
assumes *deg q2* $\leq$ *deg p* **and** *vars q2* $\subseteq$ $\{x\}$
shows *measure-pmf.prob* (*pmf-of-set UNIV*)
$\{r.\ deg\ q1 \leq deg\ p \wedge vars\ q1 \subseteq \{x\} \wedge q1 \neq q2 \wedge eval\ q1\ [x \mapsto r] = eval\ q2\ [x \mapsto r]\}$
$\leq$ *real* (*deg p*) / *real CARD*('*a*)
proof −
have *card* $\{r.\ deg\ q1 \leq deg\ p \wedge vars\ q1 \subseteq \{x\} \wedge$
$q1 \neq q2 \wedge eval\ q1\ [x \mapsto r] = eval\ q2\ [x \mapsto r]\} =$
card $\{r.\ deg\ q1 \leq deg\ p \wedge vars\ q1 \subseteq \{x\} \wedge$
$deg\ q2 \leq deg\ p \wedge vars\ q2 \subseteq \{x\} \wedge$
$q1 \neq q2 \wedge eval\ q1\ [x \mapsto r] = eval\ q2\ [x \mapsto r]\}$ **using** *assms* **by**(*auto*)
also have ... $\leq$ *deg p* **by**(*auto simp add*: *roots*)
finally show *?thesis* **by**(*auto simp add*: *measure-pmf-of-set divide-right-mono*)
qed

Soundness proof

theorem *soundness-inductive*:
assumes
vars p $\subseteq$ *set vs* **and**
deg p $\leq$ *d* **and**
distinct vs **and**
$H \neq \{\}$
shows
measure-pmf.prob
(*pmf-of-set* (*tuples UNIV* (*length vs*)))
$\{rs.$
sumcheck pr s (*H*, *p*, *v*) *r* (*zip vs rs*) $\wedge$
$v \neq (\sum \sigma \in substs\ (set\ vs)\ H.\ eval\ p\ \sigma)\}$
$\leq$ *real* (*length vs*) $*$ *real d* / *real* (*CARD*('*a*))
using *assms*
proof(*induction vs arbitrary*: *s p v r*)
case *Nil*
show *?case*
by(*simp*)
next
case (*Cons x vs*)

— abbreviations
let *?prob* = *measure-pmf.prob* (*pmf-of-set* (*tuples UNIV* (*Suc* (*length vs*))))
let *?reduced-prob* = *measure-pmf.prob* (*pmf-of-set* (*tuples UNIV* (*length vs*)))

let *?q* = $\sum \sigma \in$ *substs* (*set vs*) *H*. *inst p* σ — honest polynomial q

let *?pr-q* = *fst* (*pr* (*H*, *p*, *v*) *x vs r s*) — polynomial q from prover
let *?pr-s′* = *snd* (*pr* (*H*, *p*, *v*) *x vs r s*) — prover's next state

— some useful derived facts
have ‹*vars* (*p*) ⊆ *insert x* (*set vs*)› ‹*x* ∉ *set vs*› ‹*distinct vs*›
using ‹*vars* (*p*) ≤ *set* (*x* # *vs*)› ‹*distinct* (*x* # *vs*)› **by** *auto*

have *P0*:
‹*?prob* {*r1*#*rs* | *r1 rs*.
deg (*?pr-q*) ≤ *deg* (*p*) ∧ *vars* (*?pr-q*) ⊆ {*x*} ∧
v = ($\sum$ *a*∈*H*. *eval* (*?pr-q*) [*x* ↦ *a*]) ∧
sumcheck pr (*?pr-s′*) (*H*, *inst* (*p*) [*x* ↦ *r1*], *eval* (*?pr-q*) [*x* ↦ *r1*]) *r1* (*zip vs rs*) ∧
v ≠ ($\sum \sigma \in$ *substs* (*insert x* (*set vs*)) *H*. *eval* (*p*) σ) ∧ *?pr-q* = *?q*} = 0›
proof −
have ($\sum$ *a*∈*H*. *eval* (*?q*) [*x* ↦ *a*]) =
($\sum$ *a*∈*H*. $\sum \sigma \in$ *substs* (*set vs*) *H*. *eval* (*p*) ([*x* ↦ *a*] ++ σ))
using ‹*vars* (*p*) ⊆ *insert x* (*set vs*)› ‹*x* ∉ *set vs*›
by(*auto simp add*: *eval-sum-inst*)
moreover
have ($\sum$ *a*∈*H*. $\sum \sigma \in$ *substs* (*set vs*) *H*. *eval* (*p*) ([*x* ↦ *a*] ++ σ)) =
($\sum \sigma \in$ *substs* (*insert x* (*set vs*)) *H*. *eval* (*p*) σ) **using** ‹*x* ∉ *set vs*›
by(*auto simp add*: *sum-merge*)
ultimately
have {*r1*#*rs* | *r1 rs*.
v = ($\sum$ *a*∈*H*. *eval* (*?pr-q*) [*x* ↦ *a*]) ∧
v ≠ ($\sum \sigma \in$ *substs* (*insert x* (*set vs*)) *H*. *eval* (*p*) σ) ∧ *?pr-q* = *?q*} = {}
by(*auto*)
then show *?thesis*
by (*intro prob-empty*) (*auto 4 4*)
qed

{ — left-hand-side case where we use the roots assumption
have ‹*?prob* {*r1*#*rs* | *r1 rs*.
deg (*?pr-q*) ≤ *deg* (*p*) ∧ *vars* (*?pr-q*) ⊆ {*x*} ∧
v = ($\sum$ *a*∈*H*. *eval* (*?pr-q*) [*x* ↦ *a*]) ∧
sumcheck pr (*?pr-s′*) (*H*, *inst* (*p*) [*x* ↦ *r1*], *eval* (*?pr-q*) [*x* ↦ *r1*]) *r1* (*zip vs rs*) ∧
?pr-q ≠ *?q* ∧ *eval* (*?pr-q*) [*x* ↦ *r1*] = *eval* (*?q*) [*x* ↦ *r1*]} ≤
?prob {*r1*#*rs* | *r1 rs*.
deg (*?pr-q*) ≤ *deg* (*p*) ∧ *vars* (*?pr-q*) ⊆ {*x*} ∧
?pr-q ≠ *?q* ∧ *eval* (*?pr-q*) [*x* ↦ *r1*] = *eval* (*?q*) [*x* ↦ *r1*]}›
by (*intro prob-mono*) (*auto 4 4*)

also have ‹... =
measure-pmf.prob (*pmf-of-set UNIV*) {*r1*.
deg (*?pr-q*) ≤ *deg* (*p*) ∧ *vars* (*?pr-q*) ⊆ {*x*} ∧
?pr-q ≠ *?q* ∧ *eval* (*?pr-q*) [*x* ↦ *r1*] = *eval* (*?q*) [*x* ↦ *r1*]}›
by (*auto simp add*: *prob-tuples-hd-tl-indep*[**where** *Q* = λ*rs*. *True*, *simplified*])

also have ‹... ≤ *real* (*deg* (*p*)) / *real CARD*('*a*)›
using ‹*vars* (*p*) ⊆ *insert x* (*set vs*)› ‹*H* ≠ {}›
by(*auto simp add*: *prob-roots honest-prover-deg honest-prover-vars*)

```
  also have ‹... ≤ real d / real CARD('a)› using ‹deg (p) ≤ d›
    by (simp add: divide-right-mono)

  finally
  have ‹?prob {r1#rs | r1 rs.
                  deg (?pr-q) ≤ deg (p) ∧ vars (?pr-q) ⊆ {x} ∧
                  v = (∑ a∈H. eval (?pr-q) [x ↦ a]) ∧
                 sumcheck pr (?pr-s') (H, inst (p) [x ↦ r1], eval (?pr-q) [x ↦ r1]) r1 (zip vs rs) ∧
                  ?pr-q ≠ ?q ∧ eval (?pr-q) [x ↦ r1] = eval (?q) [x ↦ r1]}
        ≤ real d / real CARD('a)› .
}
note RP-left = this

{
  have *: ‹⋀α. eval (?q) [x ↦ α] = (∑σ ∈ substs (set vs) H. eval (inst (p) [x ↦ α]) σ)›
    using ‹vars (p) ⊆ insert x (set vs)› ‹x ∉ set vs›
    by(auto simp add: eval-sum-inst-commute)

  have ‹⋀α. vars (inst (p) [x ↦ α]) ⊆ set vs› using vars-inst ‹vars (p) ⊆ insert x (set vs)›
    by fastforce
  have ‹⋀α. deg (inst (p) [x ↦ α]) ≤ d› using deg-inst ‹deg (p) ≤ d›
    using le-trans by blast

  — right-hand-side case where we apply the induction hypothesis
  have ‹?prob {r1#rs | r1 rs.
                  deg (?pr-q) ≤ deg (p) ∧ vars (?pr-q) ⊆ {x} ∧
                  v = (∑ a∈H. eval (?pr-q) [x ↦ a]) ∧
                 sumcheck pr (?pr-s') (H, inst (p) [x ↦ r1], eval (?pr-q) [x ↦ r1]) r1 (zip vs rs) ∧
                  ?pr-q ≠ ?q ∧ eval (?pr-q) [x ↦ r1] ≠ eval (?q) [x ↦ r1]}
      ≤ ?prob {r1#rs | r1 rs.
                 sumcheck pr (?pr-s') (H, inst (p) [x ↦ r1], eval (?pr-q) [x ↦ r1]) r1 (zip vs rs) ∧
                  eval (?pr-q) [x ↦ r1] ≠ (∑σ ∈ substs (set vs) H. eval (inst (p) [x ↦ r1]) σ)}›
    by(intro prob-mono) (auto simp add: *)

  — fix r1
  also have ‹... = (∑α ∈ UNIV.
                 ?reduced-prob {rs.
                  sumcheck pr (?pr-s') (H, inst (p) [x ↦ α], eval (?pr-q) [x ↦ α]) α (zip vs rs) ∧
                  eval (?pr-q) [x ↦ α] ≠ (∑σ ∈ substs (set vs) H. eval (inst (p) [x ↦ α]) σ)})
              / real(CARD('a))›
    by(auto simp add: prob-tuples-fixed-hd)

  — apply the induction hypothesis
  also have ‹... ≤ (∑α ∈ (UNIV::'a set). real (length vs) * real d / real CARD('a))
                / real(CARD('a))›
    using ‹⋀α. vars (inst (p) [x ↦ α]) ⊆ set vs›
          ‹⋀α. deg (inst (p) [x ↦ α]) ≤ d›
          ‹distinct vs› ‹H ≠ {}›
    by (intro divide-right-mono sum-mono Cons.IH) (auto)

  also have ‹... = real (length vs) * real d / real CARD('a)›
    by fastforce
```

finally
have ‹*?prob* {*r1*#*rs* | *r1 rs*.
deg (*?pr-q*) ≤ *deg* (*p*) ∧ *vars* (*?pr-q*) ⊆ {*x*} ∧
v = (∑ *a*∈*H*. *eval* (*?pr-q*) [*x* ↦ *a*]) ∧
sumcheck pr (*?pr-s'*) (*H*, *inst* (*p*) [*x* ↦ *r1*], *eval* (*?pr-q*) [*x* ↦ *r1*]) *r1* (*zip vs rs*) ∧
?pr-q ≠ *?q* ∧ *eval* (*?pr-q*) [*x* ↦ *r1*] ≠ *eval* (*?q*) [*x* ↦ *r1*]}
≤ *real* (*length vs*) ∗ *real d* / *real CARD*('*a*)› **.**
}
note *RP-right* = *this*

— main equational reasoning proof
have ‹*?prob* {*rs*.
sumcheck pr s (*H*, *p*, *v*) *r* (*zip* (*x* # *vs*) *rs*) ∧
v ≠ (∑ σ ∈ *substs* (*insert x* (*set vs*)) *H*. *eval p* σ)}
= *?prob* {*r1*#*rs* | *r1 rs*. *sumcheck pr s* (*H*, *p*, *v*) *r* (*zip* (*x* # *vs*) (*r1*#*rs*))
∧ *v* ≠ (∑ σ ∈ *substs* (*insert x* (*set vs*)) *H*. *eval p* σ)}›
by(*intro prob-cong*) (*auto simp add*: *tuples-Suc*)

— unfold sumcheck
also have ‹... = *?prob* {*r1*#*rs* | *r1 rs*.
(*let* (*q*, *s'*) = *pr* (*H*, *p*, *v*) *x vs r s in*
deg q ≤ *deg p* ∧ *vars q* ⊆ {*x*} ∧
v = (∑ *a* ∈ *H*. *eval q* [*x* ↦ *a*]) ∧
sumcheck pr s' (*H*, *inst p* [*x* ↦ *r1*], *eval q* [*x* ↦ *r1*]) *r1* (*zip vs rs*)) ∧
v ≠ (∑ σ ∈ *substs* (*insert x* (*set vs*)) *H*. *eval p* σ)}›
by(*intro prob-cong*) (*auto del*: *subsetI*)

also have ‹... = *?prob* {*r1*#*rs* | *r1 rs* .
deg ?pr-q ≤ *deg p* ∧ *vars ?pr-q* ⊆ {*x*} ∧
v = (∑ *a*∈*H*. *eval ?pr-q* [*x* ↦ *a*]) ∧
sumcheck pr ?pr-s' (*H*, *inst p* [*x* ↦ *r1*], *eval ?pr-q* [*x* ↦ *r1*]) *r1* (*zip vs rs*) ∧
v ≠ (∑ σ ∈ *substs* (*insert x* (*set vs*)) *H*. *eval p* σ)}›
by (*intro prob-cong*) (*auto del*: *subsetI*)

— case split on whether prover's polynomial q equals honest one
also have ‹... = *?prob* {*r1*#*rs* | *r1 rs*.
deg (*?pr-q*) ≤ *deg* (*p*) ∧ *vars* (*?pr-q*) ⊆ {*x*} ∧
v = (∑ *a*∈*H*. *eval* (*?pr-q*) [*x* ↦ *a*]) ∧
sumcheck pr (*?pr-s'*) (*H*, *inst* (*p*) [*x* ↦ *r1*], *eval* (*?pr-q*) [*x* ↦ *r1*]) *r1* (*zip vs rs*) ∧
v ≠ (∑ σ ∈ *substs* (*insert x* (*set vs*)) *H*. *eval* (*p*) σ) ∧ *?pr-q* = *?q*}
+ *?prob* {*r1*#*rs* | *r1 rs*.
deg (*?pr-q*) ≤ *deg* (*p*) ∧ *vars* (*?pr-q*) ⊆ {*x*} ∧
v = (∑ *a*∈*H*. *eval* (*?pr-q*) [*x* ↦ *a*]) ∧
sumcheck pr (*?pr-s'*) (*H*, *inst* (*p*) [*x* ↦ *r1*], *eval* (*?pr-q*) [*x* ↦ *r1*]) *r1* (*zip vs rs*) ∧
v ≠ (∑ σ ∈ *substs* (*insert x* (*set vs*)) *H*. *eval* (*p*) σ) ∧ *?pr-q* ≠ *?q*}›
by(*intro prob-disjoint-cases*) *auto*

— first probability is 0
also have ‹... = *?prob* {*r1*#*rs* | *r1 rs*.
deg (*?pr-q*) ≤ *deg* (*p*) ∧ *vars* (*?pr-q*) ⊆ {*x*} ∧
v = (∑ *a*∈*H*. *eval* (*?pr-q*) [*x* ↦ *a*]) ∧
sumcheck pr (*?pr-s'*) (*H*, *inst* (*p*) [*x* ↦ *r1*], *eval* (*?pr-q*) [*x* ↦ *r1*]) *r1* (*zip vs rs*) ∧

```
                v ≠ (∑ σ ∈ substs (insert x (set vs)) H. eval (p) σ) ∧ ?pr-q ≠ ?q}›
    by (simp add: P0)

  — dropped condition
  also have ‹... ≤ ?prob {r1#rs | r1 rs.
                deg (?pr-q) ≤ deg (p) ∧ vars (?pr-q) ⊆ {x} ∧
                v = (∑ a∈H. eval (?pr-q) [x ↦ a]) ∧
                sumcheck pr (?pr-s') (H, inst (p) [x ↦ r1], eval (?pr-q) [x ↦ r1]) r1 (zip vs rs) ∧
                ?pr-q ≠ ?q}›
    by(intro prob-mono) (auto)

  also have ‹... =
        ?prob {r1#rs | r1 rs.
                  deg (?pr-q) ≤ deg (p) ∧ vars (?pr-q) ⊆ {x} ∧
                  v = (∑ a∈H. eval (?pr-q) [x ↦ a]) ∧
                 sumcheck pr (?pr-s') (H, inst (p) [x ↦ r1], eval (?pr-q) [x ↦ r1]) r1 (zip vs rs) ∧
                  ?pr-q ≠ ?q ∧ eval (?pr-q) [x ↦ r1] = eval (?q) [x ↦ r1]} +
        ?prob {r1#rs | r1 rs.
                  deg (?pr-q) ≤ deg (p) ∧ vars (?pr-q) ⊆ {x} ∧
                  v = (∑ a∈H. eval (?pr-q) [x ↦ a])  ∧
                 sumcheck pr (?pr-s') (H, inst (p) [x ↦ r1], eval (?pr-q) [x ↦ r1]) r1 (zip vs rs) ∧
                  ?pr-q ≠ ?q ∧ eval (?pr-q) [x ↦ r1] ≠ eval (?q) [x ↦ r1]}›
    by(intro prob-disjoint-cases) (auto)

  also have ‹... ≤ real d / real CARD('a) +
              (real (length vs) * real d) / real CARD('a)›
    by (intro add-mono RP-left RP-right)

  also have ‹... = (1 + real (length vs)) * real d / real CARD('a)›
    by (metis add-divide-distrib mult-Suc of-nat-Suc of-nat-add of-nat-mult)

  finally show ?case by simp
qed


corollary soundness:
  assumes
    (H, p, v) ∉ Sumcheck
    vars p = set vs and
    distinct vs and
    H ≠ {}
  shows
    measure-pmf.prob
      (pmf-of-set (tuples UNIV (arity p)))
      {rs. sumcheck pr s (H, p, v) r (zip vs rs)}
    ≤ real (arity p) * real (deg p) / real (CARD('a))
  using assms
proof −
  have *: ‹arity p = length vs› using ‹vars p = set vs› ‹distinct vs›
    by (simp add: arity-def distinct-card)

  have measure-pmf.prob (pmf-of-set (tuples UNIV (arity p)))
        {rs. sumcheck pr s (H, p, v) r (zip vs rs)} =
```

```
      measure-pmf.prob (pmf-of-set (tuples UNIV (arity p)))
        {rs. sumcheck pr s (H, p, v) r (zip vs rs) ∧ (H, p, v) ∉ Sumcheck}
  using ‹(H, p, v) ∉ Sumcheck›
    by (intro prob-cong) (auto)
  also have ... ≤ real (arity p) * real (deg p) / real (CARD('a)) using assms(2−) *
    by (auto simp add: Sumcheck-def intro!: soundness-inductive)
  finally show ?thesis by simp
qed


end

end
```

8 Sumcheck Protocol as Public-coin Proof

theory *Sumcheck-as-Public-Coin-Proof*
imports
Completeness-Proof
Soundness-Proof
begin

context *multi-variate-polynomial* **begin**

8.1 Property-related definitions

fun *sc-sound-err* :: ('*p*, '*a*, '*b*) *sc-inst* ⇒ *real* **where**
sc-sound-err (*H*, *p*, *v*) = *real* (*arity p*) ∗ *real* (*deg p*) / *real* (*CARD*('*a*))

fun *sc-compl-assm* **where**
sc-compl-assm vs ps (*H*, *p*, *v*) *xs* ⟷
set xs = *vars p* ∧ *distinct xs* ∧ *H* ≠ {}

fun *sc-sound-assm* **where**
sc-sound-assm vs ps (*H*, *p*, *v*) *xs* ⟷
set xs = *vars p* ∧ *distinct xs* ∧ *H* ≠ {}

8.2 Public coin proof locale interpretation

sublocale
scp: *public-coin-proof-strong-props*
sc-ver0 sc-ver1 Sumcheck honest-prover sc-sound-err sc-compl-assm sc-sound-assm
proof
fix *I* :: ('*p*, '*a*, '*b*) *sc-inst* **and**
vs :: *unit* **and** *ps* :: *unit* **and**
rm :: ('*v* × '*a*) *list* **and** *r* :: '*a*
assume *I* ∈ *Sumcheck* **and** *sc-compl-assm vs ps I* (*map fst rm*)
then show *sc.prove vs honest-prover ps I r rm*
by (*cases I*) (*simp add*: *prove-sc-eq-sumcheck completeness*)
next
fix *I* :: ('*p*, '*a*, '*b*) *sc-inst* **and**
vs :: *unit* **and** *ps* :: '*ps* **and**
r :: '*a* **and** *rs* :: '*a list* **and** *xs* :: '*v list* **and** *pr*
assume *I* ∉ *Sumcheck* **and** *sc-sound-assm vs ps I xs*
then show
measure-pmf.prob
(*pmf-of-set* (*tuples UNIV* (*length xs*)))
{*rs*. *sc.prove vs pr ps I r* (*zip xs rs*)}
≤ *sc-sound-err I*
proof (*cases I*)
case (*fields H p v*)
have *length xs* = *arity p* **using** ‹*sc-sound-assm vs ps I xs*› *fields*
by (*auto simp add*: *arity-def distinct-card dest*: *sym*)
then show *?thesis* **using** ‹*I* ∉ *Sumcheck*› ‹*sc-sound-assm vs ps I xs*› *fields*
by (*auto simp add*: *prove-sc-eq-sumcheck intro*: *soundness*)
qed

qed

end — context *multi-variate-polynomial*

end

9 Instantiation for Multivariate Polynomials

theory *Polynomial-Instantiation*
imports
Polynomials.More-MPoly-Type
begin

NOTE: if considered to be useful enough, the definitions and lemmas in this theory could be moved to the theory *Polynomials.More-MPoly-Type*.

Define instantiation of mpoly's. The conditions $(\neq)$ *1* and $(\neq)$ *0* in the sets being multiplied or added over are needed to prove the correspondence with evaluation: a full instance corresponds to an evaluation (see lemma below).

9.1 Instantiation of monomials

type-synonym $('a, 'b)$ *subst* $= 'a \rightharpoonup 'b$

lift-definition
inst-monom-coeff :: ‹$(nat \Rightarrow_0 nat) \Rightarrow (nat, 'a)$ *subst* $\Rightarrow$ $'a$::*comm-semiring-1*›
is ‹$\lambda m\ \sigma.\ (\prod v \mid v \in dom\ \sigma \wedge the\ (\sigma\ v)\ \widehat{}\ m\ v \neq 1.\ the\ (\sigma\ v)\ \widehat{}\ m\ v)$› .

lift-definition
inst-monom-resid :: ‹$(nat \Rightarrow_0 nat) \Rightarrow (nat, 'a)$ *subst* $\Rightarrow (nat \Rightarrow_0 nat)$›
is ‹$(\lambda m\ \sigma\ v.\ m\ v$ *when* $v \notin dom\ \sigma)$›
by (*metis* (*mono-tags*, *lifting*) *finite-subset mem-Collect-eq subsetI zero-when*)

lemmas *inst-monom-defs* = *inst-monom-coeff-def inst-monom-resid-def*

lemma *lookup-inst-monom-resid*:
shows ‹*lookup* (*inst-monom-resid m* σ) v = (*if* $v \in dom\ \sigma$ *then 0 else lookup m* v)›
by *transfer simp*

9.2 Instantiation of polynomials

definition
inst-fun :: ‹$((nat \Rightarrow_0 nat) \Rightarrow 'a) \Rightarrow (nat, 'a)$ *subst* $\Rightarrow (nat \Rightarrow_0 nat) \Rightarrow$ $'a$::*comm-semiring-1*› **where**
‹*inst-fun* $p\ \sigma = (\lambda m.\ (\sum m' \mid$ *inst-monom-resid* $m'\ \sigma = m \wedge p\ m' *$ *inst-monom-coeff* $m'\ \sigma \neq 0.$
$p\ m' *$ *inst-monom-coeff* $m'\ \sigma))$›

lemma *finite-inst-fun-keys*:
assumes ‹*finite* $\{m.\ p\ m \neq 0\}$›
shows ‹*finite* $\{m.\ (\sum m' \mid$ *inst-monom-resid* $m'\ \sigma = m \wedge p\ m' \neq 0 \wedge$ *inst-monom-coeff* $m'\ \sigma \neq 0.$
$p\ m' *$ *inst-monom-coeff* $m'\ \sigma) \neq 0\}$›
proof −
from ‹*finite* $\{m.\ p\ m \neq 0\}$›
have ‹*finite* $((\lambda m'.$ *inst-monom-resid* $m'\ \sigma)$ ‘$\{x.\ p\ x \neq 0\})$› **by** *auto*
moreover
have ‹$\{m.\ (\sum m' \mid$ *inst-monom-resid* $m'\ \sigma = m \wedge p\ m' \neq 0 \wedge$ *inst-monom-coeff* $m'\ \sigma \neq 0.$
$p\ m' *$ *inst-monom-coeff* $m'\ \sigma) \neq 0\}$
$\subseteq (\lambda m'.$ *inst-monom-resid* $m'\ \sigma)$ ‘$\{m.\ p\ m \neq 0\}$›
by (*auto elim*: *sum.not-neutral-contains-not-neutral*)
ultimately show *?thesis*

by (*auto elim*: *finite-subset*)
qed

lemma *finite-inst-fun-keys-ext*:
assumes ‹*finite* {*m*. *p m* ≠ *0*}›
shows *finite* {*a*. (∑ *m'* | *inst-monom-resid m' σ* = *a* ∧ *p m'* ≠ *0* ∧ *inst-monom-coeff m' σ* ≠ *0*.
p m' ∗ *inst-monom-coeff m' σ* ∗ (∏ *aa*. *the* (*ϱ aa*) ⌢ *lookup* (*inst-monom-resid m' σ*) *aa*)) ≠ *0*}
proof −
from ‹*finite* {*m*. *p m* ≠ *0*}›
have ‹*finite* ((λ*m'*. *inst-monom-resid m' σ*)‘{*x*. *p x* ≠ *0*})› **by** *auto*
moreover
have ‹{*m*. (∑ *m'* | *inst-monom-resid m' σ* = *m* ∧ *p m'* ≠ *0* ∧ *inst-monom-coeff m' σ* ≠ *0*.
p m' ∗ *inst-monom-coeff m' σ* ∗
(∏ *aa*. *the* (*ϱ aa*) ⌢ *lookup* (*inst-monom-resid m' σ*) *aa*)) ≠ *0*}
⊆ (λ*m'*. *inst-monom-resid m' σ*)‘{*m*. *p m* ≠ *0*}›
by (*auto elim*: *sum.not-neutral-contains-not-neutral*)
ultimately show *?thesis*
by (*auto elim*: *finite-subset*)
qed

lift-definition
inst-aux :: ‹((*nat* ⇒₀ *nat*) ⇒₀ *'a*) ⇒ (*nat*, *'a*) *subst* ⇒ (*nat* ⇒₀ *nat*) ⇒₀ *'a*::*semidom*›
is *inst-fun*
by (*auto simp add*: *inst-fun-def intro*: *finite-inst-fun-keys*)

lift-definition *inst* :: ‹*'a mpoly* ⇒ (*nat*, *'a*::*semidom*) *subst* ⇒ *'a mpoly*›
is *inst-aux* **.**

lemmas *inst-defs* = *inst-def inst-aux-def inst-fun-def*

9.3 Full instantiation corresponds to evaluation

lemma *dom-Some*: ‹*dom* (*Some o f*) = *UNIV*›
by (*simp add*: *dom-def*)

lemma *inst-full-eq-insertion*:
fixes *p* :: ‹(*'a*::*semidom*) *mpoly*› **and** *σ* :: ‹*nat* ⇒ *'a*›
shows ‹*inst p* (*Some o σ*) = *Const* (*insertion σ p*)›
proof *transfer*
fix *p* :: ‹(*nat* ⇒₀ *nat*) ⇒₀ *'a*› **and** *σ* :: ‹*nat* ⇒ *'a*›
show ‹*inst-aux p* (*Some o σ*) = *Const*₀ (*insertion-aux σ p*)›
unfolding *poly-mapping-eq-iff*
apply (*simp add*: *Const*₀*-def inst-aux.rep-eq inst-fun-def inst-monom-defs*
Poly-Mapping.single.rep-eq insertion-aux.rep-eq insertion-fun-def)
apply (*rule ext*)
subgoal for *m*
by (*cases m* = *0*)
(*simp-all add*: *Sum-any.expand-set Prod-any.expand-set dom-Some*)
done
qed

end

10 Roots Bound for Univariate Polynomials

theory *Univariate-Roots-Bound*
imports
HOL−Computational-Algebra.Polynomial
begin

NOTE: if considered to be useful enough, the lemmas in this theory could be moved to the theory *HOL−Computational-Algebra.Polynomial*.

10.1 Basic lemmas

lemma *finite-non-zero-coeffs*: ‹*finite* {*n*. *poly.coeff p n* ≠ *0*}›
using *MOST-coeff-eq-0 eventually-cofinite*
by *fastforce*

Univariate degree in terms of *Max*

lemma *poly-degree-eq-Max-non-zero-coeffs*:
degree p = *Max* (*insert 0* {*n*. *poly.coeff p n* ≠ *0*})
by (*intro le-antisym degree-le*) (*auto simp add*: *finite-non-zero-coeffs le-degree*)

10.2 Univariate roots bound

The number of roots of a product of polynomials is bounded by the sum of the number of roots of each.

lemma *card-poly-mult-roots*:
fixes *p* :: *'a*::{*comm-ring-1*,*ring-no-zero-divisors*} *poly*
and *q* :: *'a*::{*comm-ring-1*,*ring-no-zero-divisors*} *poly*
assumes *p* ≠ *0* **and** *q* ≠ *0*
shows *card* {*x*. *poly p x* ∗ *poly q x* = *0*} ≤ *card* {*x*. *poly p x* = *0*} + *card* {*x*. *poly q x* = *0*}
proof −
have *card* {*x* . *poly p x* ∗ *poly q x* = *0*} ≤ *card* ({*x* . *poly p x* = *0*} ∪ {*x* . *poly q x* = *0*})
by (*auto simp add*: *poly-roots-finite assms intro*!: *card-mono*)
also have . . . ≤ *card* {*x* . *poly p x* = *0*} + *card* {*x* . *poly q x* = *0*}
by(*auto simp add*: *Finite-Set.card-Un-le*)
finally show *?thesis* **.**
qed

A univariate polynomial p has at most deg p roots.

lemma *univariate-roots-bound*:
fixes *p* :: ‹*'a*::*idom poly*›
assumes ‹*p* ≠ *0*›
shows ‹*card* {*x*. *poly p x* = *0*} ≤ *degree p*›
using *assms*
proof (*induction degree p arbitrary*: *p rule*: *nat-less-induct*)
case *1*
then show *?case*
proof(*cases* ∃ *r*. *poly p r* = *0*)
case *True* — A root exists

— Get root r of polynomial and write *p* = [:− *r*, *1*:] ^ *order r p* ∗ *q* for some *q*.

```
  then obtain r where poly p r = 0 by(auto)
  let ?xr = [:- r, 1:] ^ order r p
  obtain q where p = ?xr * q using order-decomp ‹p ≠ 0› by(auto)

  — Useful facts about q and [:- r, 1:]^order r p
  have q ≠ 0 using ‹p = ?xr * q› ‹p ≠ 0› by(auto)
  have ?xr ≠ 0 by(simp)
  have degree ?xr > 0 using ‹?xr ≠ 0› ‹p ≠ 0› ‹poly p r = 0›
    by (simp add: degree-power-eq order-root)
  have degree q < degree p
    using ‹?xr ≠ 0› ‹q ≠ 0› ‹p = ?xr * q› ‹degree ?xr > 0›
        degree-mult-eq[where p = ?xr and q = q]
    by (simp)
  have x-roots: card {r. poly ?xr r = 0} = 1 using ‹p ≠ 0› ‹poly p r = 0›
    by(simp add: order-root)
  have q-roots: card {r. poly q r = 0} ≤ degree q using ‹q ≠ 0› ‹degree q < degree p› 1
    by (simp)

  — Final bound
  have card {r . poly p r = 0} ≤ degree p
    using ‹p = ?xr * q› ‹q ≠ 0› ‹?xr ≠ 0› ‹degree q < degree p›
        poly-mult[where p = ?xr and q = q]
        card-poly-mult-roots[where p = ?xr and q = q]
        x-roots q-roots
    by (simp)
  then show ?thesis .
 next
  case False — No root exists
  then show ?thesis by simp
 qed
qed


end
```

11 Roots Bound for Multivariate Polynomials of Arity at Most One

theory *Roots-Bounds*
imports
Polynomials.MPoly-Type-Univariate
Univariate-Roots-Bound
begin

NOTE: if considered to be useful enough, the lemmas in this theory could be moved to the theory *Polynomials.MPoly-Type-Univariate*.

11.1 Lemmas connecting univariate and multivariate polynomials

11.1.1 Basic lemmas

lemma *mpoly-to-poly-zero-iff*:
fixes p::$'a$::*comm-monoid-add mpoly*
assumes ‹*vars* $p \subseteq \{v\}$›
shows *mpoly-to-poly* v $p = 0 \longleftrightarrow p = 0$
by (*metis assms mpoly-to-poly-inverse poly-to-mpoly0 poly-to-mpoly-inverse*)

lemma *keys-monom-subset-vars*:
fixes p::$'a$::*zero mpoly*
assumes ‹$m \in$ *keys* (*mapping-of* p)›
shows *keys* $m \subseteq$ *vars* p
using *assms*
by (*auto simp add*: *vars-def*)

lemma *sum-lookup-keys-eq-lookup*:
fixes p::$'a$::*zero mpoly*
assumes ‹$m \in$ *keys* (*mapping-of* p)› **and** ‹*vars* $p \subseteq \{v\}$›
shows *sum* (*lookup* m) (*keys* m) = *lookup* m v
using *assms*
by (*auto simp add*: *subset-singleton-iff dest*!: *keys-monom-subset-vars*)

11.1.2 Total degree corresponds to degree for polynomials of arity at most one

lemma *poly-degree-eq-mpoly-degree*:
fixes p::$'a$::*comm-monoid-add mpoly*
assumes ‹*vars* $p \subseteq \{v\}$›
shows *degree* (*mpoly-to-poly* v p) = *MPoly-Type.degree* p v
using *assms*
proof −
have ∗: $\bigwedge n$. *MPoly-Type.coeff* p (*Poly-Mapping.single* v n) $\neq 0$
$\longleftrightarrow (\exists\, m{\in}$*keys* (*mapping-of* p)$.\ n =$ *lookup* m v)
by (*metis* (*no-types*, *opaque-lifting*) *Diff-eq-empty-iff Diff-insert add-0 keys-eq-empty*
keys-monom-subset-vars lookup-single-eq remove-key-keys remove-key-sum
singleton-insert-inj-eq′ *coeff-keys*[*symmetric*] *assms*)

have *degree* (*mpoly-to-poly* v p)
= *Max* (*insert 0* {n. *MPoly-Type.coeff* p (*Poly-Mapping.single* v n) $\neq 0$})
by (*simp add*: *poly-degree-eq-Max-non-zero-coeffs*)

also have ... = *MPoly-Type.degree p v*
by (*simp add*: *degree.rep-eq image-def* ∗)
finally show *?thesis* **.**
qed

lemma *mpoly-degree-eq-total-degree*:
fixes *p*::*'a*::*zero mpoly*
assumes ‹*vars p* ⊆ {*v*}›
shows *MPoly-Type.degree p v* = *total-degree p*
using *assms*
by (*auto simp add*: *MPoly-Type.degree-def total-degree-def sum-lookup-keys-eq-lookup*)

corollary *poly-degree-eq-total-degree*:
fixes *p*::*'a*::*comm-monoid-add mpoly*
assumes ‹*vars p* ⊆ {*v*}›
shows *degree* (*mpoly-to-poly v p*) = *total-degree p*
using *assms*
by (*simp add*: *poly-degree-eq-mpoly-degree mpoly-degree-eq-total-degree*)

11.2 Roots bound for univariate polynomials of type *'a mpoly*

lemma *univariate-mpoly-roots-bound*:
fixes *p*::*'a*::*idom mpoly*
assumes ‹*vars p* ⊆ {*v*}› ‹*p* ≠ *0*›
shows ‹*card* {*x*. *insertion* (λ*v*. *x*) *p* = *0*} ≤ *total-degree p*›
using *assms univariate-roots-bound*[*of mpoly-to-poly v p*, *unfolded poly-eq-insertion*[*OF* ‹*vars p* ⊆ {*v*}›]]
by (*auto simp add*: *poly-degree-eq-total-degree mpoly-to-poly-zero-iff*)

end

12 Multivariate Polynomials: Instance

theory *Concrete-Multivariate-Polynomials*
imports
../Generalized-Sumcheck-Protocol/Sumcheck-as-Public-Coin-Proof
Polynomial-Instantiation
Roots-Bounds
begin

declare *total-degree-zero* [*simp del*]

12.1 Auxiliary lemmas

lemma *card-indep-bound*:
assumes $P \Longrightarrow card\ \{x.\ Q\ x\} \leq d$
shows $card\ \{x.\ P \wedge Q\ x\} \leq d$
using *assms*
by (*cases P*) *auto*

lemma *sum-point-neq-zero* [*simp*]: $(\sum x' \mid x' = x \wedge f\ x' \neq 0.\ f\ x') = f\ x$
proof −
have ‹$(\sum x' \mid x' = x \wedge f\ x' \neq 0.\ f\ x') = (\sum x' \mid x' = x \wedge f\ x \neq 0.\ f\ x')$›
by (*intro sum.cong*) *auto*
also have ‹$\ldots = f\ x$›
by (*cases* $f\ x = 0$) (*simp-all*)
finally show *?thesis* **.**
qed

12.2 Proving the assumptions of the locale

12.2.1 Variables

lemma *vars-zero*: ‹*vars 0* = {}›
by (*simp add*: *vars-def zero-mpoly.rep-eq*)

lemma *vars-inst*: ‹*vars* (*inst p σ*) ⊆ *vars p* − *dom σ*›
by (*auto simp add*: *vars-def inst-defs keys-def MPoly-inverse*
finite-inst-fun-keys lookup-inst-monom-resid
elim!: *sum.not-neutral-contains-not-neutral split*: *if-splits*)

— Lemmas for to translate the roots bound to the format of the locale assumption.

lemma *vars-minus*: ‹*vars p* = *vars* (−*p*)›
by(*simp add*: *vars-def uminus-mpoly.rep-eq*)

lemma *vars-subtr*:
fixes *p q* :: ‹*'a*::*comm-ring mpoly*›
shows ‹*vars* (*p* − *q*) ⊆ *vars p* ∪ *vars q*›
by(*simp add*: *vars-add*[**where** *?p1.0* = *p* **and** *?p2.0* = −*q*, *simplified*] *vars-minus*[**where** *p* = *q*])

12.2.2 Degree

abbreviation *deg* :: ‹(*'a*::*zero*) *mpoly* ⇒ *nat*› **where**

‹*deg p* ≡ *total-degree p*›

— We show the assumptions *multi-variate-polynomial.deg-zero*, *multi-variate-polynomial.deg-add* and *multi-variate-polynomial.deg-inst*.

lemma *deg-zero*: ‹*deg 0 = 0*› **by** (*fact total-degree-zero*)

lemma *deg-add*: ‹*deg* (*p* + *q*) ≤ *max* (*deg p*) (*deg q*)›
proof −
have ‹*deg* (*p* + *q*) = *Max* (*insert 0* ((λ*x*. *sum* (*lookup x*) (*keys x*)) ‘ *keys* (*mapping-of p* + *mapping-of q*)))›
by (*simp add*: *total-degree.rep-eq plus-mpoly.rep-eq*)
also have ‹... ≤ *Max* (*insert 0* ((λ*x*. *sum* (*lookup x*) (*keys x*)) ‘ (*keys* (*mapping-of p*) ∪ *keys* (*mapping-of q*))))›
by (*intro Max-mono Set.insert-mono image-mono Poly-Mapping.keys-add*) (*auto*)
also have ‹... = *Max* ((*insert 0* ((λ*x*. *sum* (*lookup x*) (*keys x*)) ‘ *keys* (*mapping-of p*)))
∪ (*insert 0* ((λ*x*. *sum* (*lookup x*) (*keys x*)) ‘ *keys* (*mapping-of q*))))›
by (*simp add*: *image-Un*)
also have ‹... = *max* (*Max* (*insert 0* ((λ*x*. *sum* (*lookup x*) (*keys x*)) ‘ *keys* (*mapping-of p*))))
(*Max* (*insert 0* ((λ*x*. *sum* (*lookup x*) (*keys x*)) ‘ *keys* (*mapping-of q*))))›
by (*intro Max-Un*) (*auto*)
also have ‹... = *max* (*deg p*) (*deg q*)›
by (*simp add*: *total-degree.rep-eq*)
finally show *?thesis* **.**
qed

lemma *deg-inst*: ‹*deg* (*inst p σ*) ≤ *deg p*›
proof (*transfer*)
fix *p* :: ‹(*nat* ⇒$_0$ *nat*) ⇒$_0$ *'a*› **and** *σ* :: (*nat*, *'a*) *subst*
show ‹*Max* (*insert 0* ((λ*m*. *sum* (*lookup m*) (*keys m*)) ‘ *keys* (*inst-aux p σ*)))
≤ *Max* (*insert 0* ((λ*m*. *sum* (*lookup m*) (*keys m*)) ‘ *keys p*))›
by (*auto simp add*: *keys-def inst-defs finite-inst-fun-keys lookup-inst-monom-resid*
elim!: *sum.not-neutral-contains-not-neutral*)
(*fastforce simp add*: *Max-ge-iff intro*!: *disjI2 intro*: *sum-mono2*)
qed

— Lemmas for translating the roots bound to the format of the locale assumption.

lemma *deg-minus*: ‹*deg p* = *deg* (−*p*)›
by(*auto simp add*: *total-degree-def uminus-mpoly.rep-eq*)

lemma *deg-subtr*:
fixes *p q* :: ‹*'a*::*comm-ring mpoly*›
shows ‹*deg* (*p* − *q*) ≤ *max* (*deg p*) (*deg* (*q*))›
by(*auto simp add*: *deg-add*[**where** *p* = *p* **and** *q* = −*q*, *simplified*] *deg-minus*[**where** *p* = *q*])

12.2.3 Evaluation

abbreviation *eval* :: ‹*'a mpoly* ⇒ (*nat*, *'a*) *subst* ⇒ (*'a*::*comm-semiring-1*)› **where**
‹*eval p σ* ≡ *insertion* (*the o σ*) *p*›

— We show the assumptions *multi-variate-polynomial.eval-zero*, *multi-variate-polynomial.eval-add* and *multi-variate-polynomial.eval-inst*.

lemma *eval-zero*: ‹*eval 0 σ = 0*›
  by (*fact insertion-zero*)

lemma *eval-add*: ‹*vars p ∪ vars q ⊆ dom σ ⟹ eval (p + q) σ = eval p σ + eval q σ*›
  by (*intro insertion-add*)

— evaluation and instantiation

lemma *eval-inst*: ‹*eval (inst p σ) ϱ = eval p (ϱ ++ σ)*›
proof (*transfer*, *transfer*)
  fix *p* :: ‹(*nat* ⇒₀ *nat*) ⇒ *'a*› **and** *σ ϱ* :: ‹(*nat*, *'a*) *subst*›
  assume *fin*: ‹*finite* {*m*. *p m* ≠ *0*}›
  show ‹*insertion-fun* (*the* ∘ *ϱ*) (*inst-fun p σ*) = *insertion-fun* (*the* ∘ *ϱ* ++ *σ*) *p*›
  proof −
    let *?mon* = ‹*λσ m v*. *the* (*σ v*) ^ *lookup m v*›

    have ‹*x* ^ *lookup m v* ≠ *1* ⟹ *v* ∈ *keys m*› **for** *x* :: *'a* **and** *v* **and** *m* :: *nat* ⇒₀ *nat*
      using *zero-less-iff-neq-zero* **by** (*fastforce simp add*: *in-keys-iff*)
    then have *exp-fin*: ‹*finite* {*v*. *P v* ∧ *f v* ^ *lookup m v* ≠ *1*}›
      for *f* :: *nat* ⇒ *'a* **and** *m* :: *nat* ⇒₀ *nat* **and** *P* :: *nat* ⇒ *bool*
      by (*auto intro*: *finite-subset*[**where** *B*=*keys m*])

    note *fin-simps* [*simp*] = *fin this this*[**where** *P1*=*λ*-. *True*, *simplified*]
    note *map-add-simps* [*simp*] = *map-add-dom-app-simps*(*1*,*3*)

    have ‹*insertion-fun* (*the* ∘ *ϱ*) (*inst-fun p σ*) =
      (∑ *m*. (∑ *m'* | *inst-monom-resid m' σ = m* ∧ *p m'* ≠ *0* ∧ *inst-monom-coeff m' σ* ≠ *0*.
          p m' ∗ *inst-monom-coeff m' σ*) ∗ (∏ *v*. *?mon ϱ m v*))›
      by (*simp add*: *insertion-fun-def inst-fun-def*)

    also have ‹... =
      (∑ *m*. (∑ *m'* | *inst-monom-resid m' σ = m* ∧ *p m'* ≠ *0* ∧ *inst-monom-coeff m' σ* ≠ *0*.
          p m' ∗ *inst-monom-coeff m' σ* ∗ (∏ *v*. *?mon ϱ m v*)))›
      by (*intro Sum-any.cong*) (*simp add*: *sum-distrib-right*)

    also have ‹... =
      (∑ *m*. (∑ *m'* | *inst-monom-resid m' σ = m* ∧ *p m'* ≠ *0* ∧ *inst-monom-coeff m' σ* ≠ *0*.
          p m' ∗ *inst-monom-coeff m' σ* ∗ (∏ *v* | *v* ∉ *dom σ* ∧ *?mon ϱ m' v* ≠ *1*. *?mon ϱ m' v*)))›
      by (*intro Sum-any.cong sum.cong*)
        (*auto simp add*: *lookup-inst-monom-resid Prod-any.expand-set intro*: *arg-cong*)

    also have ‹... =
      (∑ *m*. (∑ *m'* | *inst-monom-resid m' σ = m* ∧ *p m'* ≠ *0* ∧
            (∏ *v* | *v* ∈ *dom σ* ∧ *?mon σ m' v* ≠ *1*. *?mon σ m' v*) ≠ *0*.
          p m' ∗
          ((∏ *v* | *v* ∈ *dom σ* ∧ *?mon σ m' v* ≠ *1*. *?mon* (*ϱ* ++ *σ*) *m' v*) ∗
           (∏ *v* | *v* ∉ *dom σ* ∧ *?mon ϱ m' v* ≠ *1*. *?mon* (*ϱ* ++ *σ*) *m' v*))))›
      by (*simp add*: *inst-monom-coeff-def mult.assoc*)

    also have ‹... =

$(\sum m.\ (\sum m' \mid \mathit{inst\text{-}monom\text{-}resid}\ m'\ \sigma = m \wedge p\ m' \neq 0 \wedge$
$(\prod v \mid v \in \mathit{dom}\ \sigma \wedge \mathit{?mon}\ \sigma\ m'\ v \neq 1.\ \mathit{?mon}\ \sigma\ m'\ v) \neq 0 \wedge$
$(\prod v \mid v \notin \mathit{dom}\ \sigma \wedge \mathit{?mon}\ \varrho\ m'\ v \neq 1.\ \mathit{?mon}\ \varrho\ m'\ v) \neq 0.$
$p\ m' *$
$((\prod v \mid v \in \mathit{dom}\ \sigma \wedge \mathit{?mon}\ \sigma\ m'\ v \neq 1.\ \mathit{?mon}\ (\varrho +\!\!+ \sigma)\ m'\ v) *$
$(\prod v \mid v \notin \mathit{dom}\ \sigma \wedge \mathit{?mon}\ \varrho\ m'\ v \neq 1.\ \mathit{?mon}\ (\varrho +\!\!+ \sigma)\ m'\ v))))\rangle$
by (*intro Sum-any.cong sum.mono-neutral-right*) (*auto*)

also have ‹... =
$(\sum m.\ (\sum m' \mid \mathit{inst\text{-}monom\text{-}resid}\ m'\ \sigma = m \wedge p\ m' \neq 0 \wedge$
$(\prod v \mid v \in \mathit{dom}\ \sigma \wedge \mathit{?mon}\ \sigma\ m'\ v \neq 1.\ \mathit{?mon}\ \sigma\ m'\ v) \neq 0 \wedge$
$(\prod v \mid v \notin \mathit{dom}\ \sigma \wedge \mathit{?mon}\ \varrho\ m'\ v \neq 1.\ \mathit{?mon}\ \varrho\ m'\ v) \neq 0.$
$p\ m' *$
$(\prod v \mid v \in \mathit{dom}\ \sigma \wedge \mathit{?mon}\ \sigma\ m'\ v \neq 1 \vee v \notin \mathit{dom}\ \sigma \wedge \mathit{?mon}\ \varrho\ m'\ v \neq 1.\ \mathit{?mon}\ (\varrho +\!\!+ \sigma)\ m'\ v)))\rangle$
by (*subst prod.union-disjoint*[*symmetric*])
(*auto intro*!: *Sum-any.cong sum.cong prod.cong intro*: *arg-cong*)

also have ‹... =
$(\sum m.\ (\sum m' \mid \mathit{inst\text{-}monom\text{-}resid}\ m'\ \sigma = m \wedge p\ m' \neq 0 \wedge$
$(\prod v \mid v \in \mathit{dom}\ \sigma \wedge \mathit{?mon}\ \sigma\ m'\ v \neq 1.\ \mathit{?mon}\ \sigma\ m'\ v) \neq 0 \wedge$
$(\prod v \mid v \notin \mathit{dom}\ \sigma \wedge \mathit{?mon}\ \varrho\ m'\ v \neq 1.\ \mathit{?mon}\ \varrho\ m'\ v) \neq 0.$
$p\ m' * (\prod v.\ \mathit{?mon}\ (\varrho +\!\!+ \sigma)\ m'\ v)))\rangle$
apply (*intro Sum-any.cong sum.cong arg-cong*[**where** f=(∗) x **for** x], *simp*)
apply (*simp add*: *Prod-any.expand-set*)
apply (*intro prod.cong*, *simp-all*)
by (*metis* (*no-types*, *opaque-lifting*) *map-add-dom-app-simps*(*1*,*3*))

also have ‹... =
$(\sum m.\ (\sum m' \mid \mathit{inst\text{-}monom\text{-}resid}\ m'\ \sigma = m \wedge p\ m' \neq 0 \wedge (\prod v.\ \mathit{?mon}\ (\varrho +\!\!+ \sigma)\ m'\ v) \neq 0.$
$p\ m' * (\prod v.\ \mathit{?mon}\ (\varrho +\!\!+ \sigma)\ m'\ v)))\rangle$
apply (*intro Sum-any.cong sum.mono-neutral-right*, *simp-all*)
apply (*safe*, *simp-all*)
— fixme: cannot get auto/fastforce to do instantiations below
subgoal for $m\ v\ z$
by (*auto dest*: *Prod-any-not-zero*[*rotated*, **where** a=v])
subgoal for $m'\ v$
by (*auto simp add*: *domIff dest*: *Prod-any-not-zero*[*rotated*, **where** a=v])
done

also have ‹... =
$(\sum m.\ (\mathit{sum}$
$(\lambda m'.\ p\ m' * (\prod v.\ \mathit{?mon}\ (\varrho +\!\!+ \sigma)\ m'\ v))$
$\{m' \in \{m'.\ p\ m' \neq 0 \wedge (\prod v.\ \mathit{?mon}\ (\varrho +\!\!+ \sigma)\ m'\ v) \neq 0\}.$
$\mathit{inst\text{-}monom\text{-}resid}\ m'\ \sigma = m\}))\rangle$
by (*intro Sum-any.cong sum.cong*) (*auto*)

also have ‹... =
$(\sum m \in (\lambda m'.\ \mathit{inst\text{-}monom\text{-}resid}\ m'\ \sigma)\ `\ \{m'.\ p\ m' \neq 0 \wedge (\prod v.\ \mathit{the}\ ((\varrho +\!\!+ \sigma)\ v)\ \widehat{}\ \mathit{lookup}\ m'\ v) \neq 0\}.$
$(\mathit{sum}$
$(\lambda m'.\ p\ m' * (\prod v.\ \mathit{?mon}\ (\varrho +\!\!+ \sigma)\ m'\ v))$
$\{m' \in \{m'.\ p\ m' \neq 0 \wedge (\prod v.\ \mathit{?mon}\ (\varrho +\!\!+ \sigma)\ m'\ v) \neq 0\}.$

```
                inst-monom-resid m' σ = m}))›
    by (intro Sum-any.expand-superset) (auto elim: sum.not-neutral-contains-not-neutral)

  also have ‹... = (∑ m. p m * (∏ v. ?mon (ϱ ++ σ) m v))›
    by (subst Sum-any.expand-set, subst sum.group) (auto)

  also have ‹... = insertion-fun (the ∘ ϱ ++ σ) p›
    by (simp add: insertion-fun-def)
  finally show ?thesis .
 qed
qed
```

— Lemmas for translating the roots bound to the format of the locale assumption.

```
lemma eval-minus:
 fixes p :: ‹'a::comm-ring-1 mpoly›
 shows ‹eval (−p) σ = − eval p σ›
 using sum-negf[where f = λa . (lookup (mapping-of p) a * (∏ aa. the (σ aa) ^ lookup a aa))]
 by(auto simp add: uminus-mpoly.rep-eq insertion-def insertion-aux-def insertion-fun-def)
   (smt (verit) Collect-cong Sum-any.expand-set add.inverse-neutral neg-equal-iff-equal)

lemma eval-subtr:
 fixes p q :: ‹'a::comm-ring-1 mpoly›
 assumes ‹vars p ⊆ dom σ› ‹vars q ⊆ dom σ›
 shows ‹eval (p − q) σ = eval p σ − eval q σ›
 using assms
 by(auto simp add: vars-minus[where p = q] eval-minus[where p = q]
                   eval-add[where p = p and q = −q, simplified])
```

12.2.4 Roots assumption

```
lemma univariate-eval-as-insertion:
 fixes p::‹'a::comm-ring-1 mpoly› and r
 assumes vars p ⊆ {x}
 shows eval p [x ↦ r] = insertion (λx. r) p
 using assms
 by (intro insertion-irrelevant-vars) auto

lemma univariate-mpoly-roots-bound-eval:   — variant using eval
 fixes p::'a::idom mpoly
 assumes ‹vars p ⊆ {x}› ‹p ≠ 0›
 shows ‹card {r. eval p [x ↦ r] = 0} ≤ deg p›
 using assms
 by (simp add: univariate-eval-as-insertion univariate-mpoly-roots-bound)

lemma mpoly-roots:
 fixes p q :: ‹'a::idom mpoly› and d x
 shows ‹card {r. deg p ≤ d ∧ vars p ⊆ {x} ∧ deg q ≤ d ∧ vars q ⊆ {x} ∧
              p ≠ q ∧ eval p [x ↦ r] = eval q [x ↦ r]} ≤ d›
proof (intro card-indep-bound)
 assume ‹deg p ≤ d› ‹vars p ⊆ {x}› ‹deg q ≤ d› ‹vars q ⊆ {x}› ‹p ≠ q›
 show ‹card {r. eval p [x ↦ r] = eval q [x ↦ r]} ≤ d›
 proof −
```

```
    have ‹card {r. eval p [x ↦ r] = eval q [x ↦ r]} = card {r. eval (p − q) [x ↦ r] = 0}›
      using ‹vars p ⊆ {x}› ‹vars q ⊆ {x}› by (simp add: eval-subtr)
    also have ‹... ≤  deg (p − q)›
      using ‹vars p ⊆ {x}› ‹vars q ⊆ {x}› ‹p ≠ q›
      by (intro univariate-mpoly-roots-bound-eval subset-trans[OF vars-subtr]) (auto)
    also have ‹... ≤  d› using ‹deg p ≤ d› ‹deg q ≤ d›
      by (intro le-trans[OF deg-subtr]) (simp)
    finally show ?thesis .
  qed
qed
```

12.3 Locale interpretation

Finally, collect all relevant lemmas and instantiate the abstract polynomials locale.

```
lemmas multi-variate-polynomial-lemmas =
  vars-finite vars-zero vars-add vars-inst
  deg-zero deg-add deg-inst
  eval-zero eval-add eval-inst
  mpoly-roots

interpretation mpoly:
  multi-variate-polynomial vars deg :: 'a::{finite, idom} mpoly ⇒ nat eval inst
  by (unfold-locales) (auto simp add: multi-variate-polynomial-lemmas)
```

Here are the main results, spezialized for type *'a mpoly*. The completeness theorem for this type is

```
⟦(?H, ?p, ?v) ∈ mpoly.Sumcheck; vars ?p = set (map fst ?rm);
 distinct (map fst ?rm); ?H ≠ {}⟧
⟹ mpoly.sumcheck mpoly.honest-prover ?u (?H, ?p, ?v) ?r ?rm
```

and the soundness theorem reads

```
⟦(?H, ?p, ?v) ∉ mpoly.Sumcheck; vars ?p = set ?vs; distinct ?vs; ?H ≠ {}⟧
⟹ measure-pmf.prob (pmf-of-set (tuples UNIV (mpoly.arity ?p)))
      {rs. mpoly.sumcheck ?pr ?s (?H, ?p, ?v) ?r (zip ?vs rs)}
     ≤ real (mpoly.arity ?p) * real (deg ?p) / real CARD(?'a)
```

.

```
end
```