Khovanskii's Theorem

Angeliki Koutsoukou-Argyraki and Lawrence C. Paulson

March 17, 2025

Abstract

We formalise the proof of an important theorem in additive combinatorics due to Khovanskii [2, 3], attesting that the cardinality of the set of all sums of n many elements of A, where A is a finite subset of an abelian group, is a polynomial in n for all sufficiently large n. We follow a proof of the theorem due to Nathanson and Ruzsa [4, 5] as presented in the notes "Introduction to Additive Combinatorics" by Timothy Gowers [1] for the University of Cambridge.

Contents

1 Product Operator for Commutative Monoids **3**
1.1 Products over Finite Sets 3
1.2 Results for Abelian Groups 8

2 Khovanskii's Theorem **9**
2.1 Arithmetic operations on lists, pointwise on the elements . . 10
2.2 The pointwise ordering on two equal-length lists of natural numbers 12
2.3 Pointwise minimum and maximum of a set of lists 14
2.4 A locale to fix the finite subset $A \subseteq G$ 14
2.5 Adding one to a list element 17
2.6 The set of all r-tuples that sum to n 17
2.7 Lemma 2.7 in Gowers's notes 21
2.8 The set of minimal elements of a set of r-tuples is finite . . . 23
2.9 Towards Lemma 2.9 in Gowers's notes 25
2.10 Towards the main theorem 30

Acknowledgements The authors were supported by the ERC Advanced Grant ALEXANDRIA (Project 742178) funded by the European Research Council.

1 Product Operator for Commutative Monoids

theory *FiniteProduct*
imports
Jacobson-Basic-Algebra.Group-Theory

begin

1.1 Products over Finite Sets

context *commutative-monoid* **begin**

definition $M\text{-}ify\ x \equiv if\ x \in M\ then\ x\ else\ \mathbf{1}$

definition $fincomp\ f\ A \equiv if\ finite\ A\ then\ Finite\text{-}Set.fold\ (\lambda x\ y.\ f\ x \cdot M\text{-}ify\ y)\ \mathbf{1}\ A$ $else\ \mathbf{1}$

lemma *fincomp-empty* [*simp*]: $fincomp\ f\ \{\} = \mathbf{1}$
by (*simp add*: *fincomp-def*)

lemma *fincomp-infinite*[*simp*]: $infinite\ A \Longrightarrow fincomp\ f\ A = \mathbf{1}$
by (*simp add*: *fincomp-def*)

lemma *left-commute*: $[\![\ a \in M;\ b \in M;\ c \in M\ ]\!] \Longrightarrow b \cdot (a \cdot c) = a \cdot (b \cdot c)$
using *commutative* **by** *force*

lemma *comp-fun-commute-onI*:
assumes $f \in F \to M$
shows $comp\text{-}fun\text{-}commute\text{-}on\ F\ (\lambda x\ y.\ f\ x \cdot M\text{-}ify\ y)$
using *assms*
by (*auto simp add*: *comp-fun-commute-on-def Pi-iff M-ify-def left-commute*)

lemma *fincomp-closed* [*simp*]:
assumes $f \in F \to M$
shows $fincomp\ f\ F \in M$
proof −
interpret $comp\text{-}fun\text{-}commute\text{-}on\ F\ \lambda x\ y.\ f\ x \cdot M\text{-}ify\ y$
by (*simp add*: *assms comp-fun-commute-onI*)
show *?thesis*
unfolding *fincomp-def*
by (*smt* (*verit*, *ccfv-threshold*) *M-ify-def Pi-iff fold-graph-fold assms composition-closed equalityE fold-graph-closed-lemma unit-closed*)
qed

lemma *fincomp-insert* [*simp*]:
assumes F: $finite\ F\ a \notin F$ **and** f: $f \in F \to M\ f\ a \in M$
shows $fincomp\ f\ (insert\ a\ F) = f\ a \cdot fincomp\ f\ F$
proof −
interpret $comp\text{-}fun\text{-}commute\text{-}on\ insert\ a\ F\ \lambda x\ y.\ f\ x \cdot M\text{-}ify\ y$

by (*simp add*: *comp-fun-commute-onI f*)
show *?thesis*
using *assms fincomp-closed commutative-monoid.M-ify-def commutative-monoid-axioms*
by (*fastforce simp add*: *fincomp-def*)
qed

lemma *fincomp-unit-eqI*: $(\bigwedge x.\ x \in A \Longrightarrow f\ x = \mathbf{1}) \Longrightarrow fincomp\ f\ A = \mathbf{1}$
proof (*induct A rule*: *infinite-finite-induct*)
case *empty* **show** *?case* **by** *simp*
next
case (*insert a A*)
have $(\lambda i.\ \mathbf{1}) \in A \to M$ **by** *auto*
with *insert* **show** *?case* **by** *simp*
qed *simp*

lemma *fincomp-unit* [*simp*]: $fincomp\ (\lambda i.\ \mathbf{1})\ A = \mathbf{1}$
by (*simp add*: *fincomp-unit-eqI*)

lemma *funcset-Int-left* [*simp*, *intro*]:
$[\![f \in A \to C;\ f \in B \to C]\!] \Longrightarrow f \in A\ Int\ B \to C$
by *fast*

lemma *funcset-Un-left* [*iff*]:
$(f \in A\ Un\ B \to C) = (f \in A \to C \wedge f \in B \to C)$
by *fast*

lemma *fincomp-Un-Int*:
$[\![finite\ A;\ finite\ B;\ g \in A \to M;\ g \in B \to M]\!] \Longrightarrow$
$fincomp\ g\ (A \cup B) \cdot fincomp\ g\ (A \cap B) =$
$fincomp\ g\ A \cdot fincomp\ g\ B$
— The reversed orientation looks more natural, but LOOPS as a simprule!
proof (*induct set*: *finite*)
case *empty* **then show** *?case* **by** *simp*
next
case (*insert a A*)
then have $g\ a \in M\ g \in A \to M$ **by** *blast+*
with *insert* **show** *?case*
by (*simp add*: *Int-insert-left associative insert-absorb left-commute*)
qed

lemma *fincomp-Un-disjoint*:
$[\![finite\ A;\ finite\ B;\ A \cap B = \{\};\ g \in A \to M;\ g \in B \to M]\!]$
$\Longrightarrow fincomp\ g\ (A \cup B) = fincomp\ g\ A \cdot fincomp\ g\ B$
by (*metis Pi-split-domain fincomp-Un-Int fincomp-closed fincomp-empty right-unit*)

lemma *fincomp-comp*:
$[\![f \in A \to M;\ g \in A \to M]\!] \Longrightarrow fincomp\ (\lambda x.\ f\ x \cdot g\ x)\ A = (fincomp\ f\ A \cdot fincomp\ g\ A)$
proof (*induct A rule*: *infinite-finite-induct*)

case *empty* **show** *?case* **by** *simp*
next
case (*insert a A*)
then have $f\ a \in M$ $g \in A \to M$ $g\ a \in M$ $f \in A \to M$ $(\lambda x.\ f\ x \cdot g\ x) \in A \to M$
by *blast+*
then show *?case*
by (*simp add*: *insert associative left-commute*)
qed *simp*

lemma *fincomp-cong'*:
assumes $A = B$ $g \in B \to M$ $\bigwedge i.\ i \in B \Longrightarrow f\ i = g\ i$
shows *fincomp f A* = *fincomp g B*
proof (*cases finite B*)
case *True*
then have *?thesis*
using *assms*
proof (*induct arbitrary*: *A*)
case *empty* **thus** *?case* **by** *simp*
next
case (*insert x B*)
then have *fincomp f A* = *fincomp f* (*insert x B*) **by** *simp*
also from *insert* **have** ... = $f\ x \cdot$ *fincomp f B*
by (*simp add*: *Pi-iff*)
also from *insert* **have** ... = $g\ x \cdot$ *fincomp g B* **by** *fastforce*
also from *insert* **have** ... = *fincomp g* (*insert x B*)
by (*intro fincomp-insert* [*THEN sym*]) *auto*
finally show *?case* **.**
qed
with *assms* **show** *?thesis* **by** *simp*
next
case *False* **with** *assms* **show** *?thesis* **by** *simp*
qed

lemma *fincomp-cong*:
assumes $A = B$ $g \in B \to M$ $\bigwedge i.\ i \in B$ =*simp*=> $f\ i = g\ i$
shows *fincomp f A* = *fincomp g B*
using *assms* **unfolding** *simp-implies-def* **by** (*blast intro*: *fincomp-cong'*)

Usually, if this rule causes a failed congruence proof error, the reason is that the premise $g \in B \to M$ cannot be shown. Adding *Pi-def* to the simpset is often useful. For this reason, *fincomp-cong* is not added to the simpset by default.

lemma *fincomp-0* [*simp*]:
$f \in \{0::nat\} \to M \Longrightarrow$ *fincomp f* $\{..0\} = f\ 0$
by (*simp add*: *Pi-def*)

lemma *fincomp-0'*: $f \in \{..n\} \to M \Longrightarrow (f\ 0) \cdot$ *fincomp f* $\{Suc\ 0..n\}$ = *fincomp f* $\{..n\}$
by (*metis Pi-split-insert-domain Suc-n-not-le-n atLeastAtMost-iff atLeastAtMost-insertL*

atMost-atLeast0 finite-atLeastAtMost fincomp-insert le0)

lemma *fincomp-Suc* [*simp*]:
  $f \in \{..Suc\ n\} \to M \Longrightarrow fincomp\ f\ \{..Suc\ n\} = (f\ (Suc\ n) \cdot fincomp\ f\ \{..n\})$
  by (*simp add*: *Pi-def atMost-Suc*)

lemma *fincomp-Suc2*:
  $f \in \{..Suc\ n\} \to M \Longrightarrow fincomp\ f\ \{..Suc\ n\} = (fincomp\ (\%i.\ f\ (Suc\ i))\ \{..n\} \cdot f\ 0)$
proof (*induct n*)
  case *0* **thus** *?case* **by** (*simp add*: *Pi-def*)
next
  case *Suc* **thus** *?case*
    by (*simp add*: *associative Pi-def*)
qed

lemma *fincomp-Suc3*:
  assumes $f \in \{..n :: nat\} \to M$
  shows $fincomp\ f\ \{..\ n\} = (f\ n) \cdot fincomp\ f\ \{..<\ n\}$
proof (*cases* $n = 0$)
  case *True* **thus** *?thesis*
    using *assms atMost-Suc* **by** *simp*
next
  case *False*
  then obtain k **where** $n = Suc\ k$
    using *not0-implies-Suc* **by** *blast*
  thus *?thesis*
    using *fincomp-Suc*[*of f k*] *assms atMost-Suc lessThan-Suc-atMost* **by** *simp*
qed

lemma *fincomp-reindex*:
  $f \in (h\ `\ A) \to M \Longrightarrow$
      $inj\text{-}on\ h\ A \Longrightarrow fincomp\ f\ (h\ `\ A) = fincomp\ (\lambda x.\ f\ (h\ x))\ A$
proof (*induct A rule*: *infinite-finite-induct*)
  case (*infinite A*)
  hence $\neg$ *finite* (*h* ‘ *A*)
    using *finite-imageD* **by** *blast*
  with ‹¬ *finite A*› **show** *?case* **by** *simp*
qed (*auto simp add*: *Pi-def*)

lemma *fincomp-const*:
  assumes *a* [*simp*]: $a \in M$
  shows $fincomp\ (\lambda x.\ a)\ A = rec\text{-}nat\ \mathbf{1}\ (\lambda u.\ (\cdot)\ a)\ (card\ A)$
  by (*induct A rule*: *infinite-finite-induct*) *auto*

lemma *fincomp-singleton*:
  assumes *i-in-A*: $i \in A$ **and** *fin-A*: *finite A* **and** *f-Pi*: $f \in A \to M$
  shows $fincomp\ (\lambda j.\ if\ i = j\ then\ f\ j\ else\ \mathbf{1})\ A = f\ i$
  using *i-in-A fincomp-insert* [*of* $A - \{i\}$ *i* $(\lambda j.\ if\ i = j\ then\ f\ j\ else\ \mathbf{1})$]

fin-A f-Pi fincomp-unit [*of* $A - \{i\}$]
fincomp-cong [*of* $A - \{i\}$ $A - \{i\}$ ($\lambda j.$ *if* $i = j$ *then* $f\ j$ *else* $\mathbf{1}$) ($\lambda i.\ \mathbf{1}$)]
unfolding *Pi-def simp-implies-def* **by** (*force simp add*: *insert-absorb*)

lemma *fincomp-singleton-swap*:
assumes *i-in-A*: $i \in A$ **and** *fin-A*: *finite* A **and** *f-Pi*: $f \in A \to M$
shows *fincomp* ($\lambda j.$ *if* $j = i$ *then* $f\ j$ *else* $\mathbf{1}$) $A = f\ i$
using *fincomp-singleton* [*OF assms*] **by** (*simp add*: *eq-commute*)

lemma *fincomp-mono-neutral-cong-left*:
assumes *finite* B
and $A \subseteq B$
and *1*: $\bigwedge i.\ i \in B - A \Longrightarrow h\ i = \mathbf{1}$
and *gh*: $\bigwedge x.\ x \in A \Longrightarrow g\ x = h\ x$
and *h*: $h \in B \to M$
shows *fincomp* $g\ A$ = *fincomp* $h\ B$
proof−
have *eq*: $A \cup (B - A) = B$ **using** ‹$A \subseteq B$› **by** *blast*
have *d*: $A \cap (B - A) = \{\}$ **using** ‹$A \subseteq B$› **by** *blast*
from ‹*finite* B› ‹$A \subseteq B$› **have** *f*: *finite* A *finite* $(B - A)$
by (*auto intro*: *finite-subset*)
have $h \in A \to M$ $h \in B - A \to M$
using *assms* **by** (*auto simp*: *image-subset-iff-funcset*)
moreover have *fincomp* $g\ A$ = *fincomp* $h\ A \cdot$ *fincomp* $h\ (B - A)$
proof −
have *fincomp* $h\ (B - A) = \mathbf{1}$
using *1 fincomp-unit-eqI* **by** *blast*
moreover have *fincomp* $g\ A$ = *fincomp* $h\ A$
using ‹$h \in A \to M$› *fincomp-cong′ gh* **by** *blast*
ultimately show *?thesis*
by (*simp add*: ‹$h \in A \to M$›)
qed
ultimately show *?thesis*
by (*simp add*: *fincomp-Un-disjoint* [*OF f d*, *unfolded eq*])
qed

lemma *fincomp-mono-neutral-cong-right*:
assumes *finite* B
and $A \subseteq B$ $\bigwedge i.\ i \in B - A \Longrightarrow g\ i = \mathbf{1}$ $\bigwedge x.\ x \in A \Longrightarrow g\ x = h\ x$ $g \in B \to M$
shows *fincomp* $g\ B$ = *fincomp* $h\ A$
using *assms* **by** (*auto intro*!: *fincomp-mono-neutral-cong-left* [*symmetric*])

lemma *fincomp-mono-neutral-cong*:
assumes [*simp*]: *finite* B *finite* A
and $*$: $\bigwedge i.\ i \in B - A \Longrightarrow h\ i = \mathbf{1}$ $\bigwedge i.\ i \in A - B \Longrightarrow g\ i = \mathbf{1}$
and *gh*: $\bigwedge x.\ x \in A \cap B \Longrightarrow g\ x = h\ x$
and *g*: $g \in A \to M$
and *h*: $h \in B \to M$
shows *fincomp* $g\ A$ = *fincomp* $h\ B$

proof −
  have *fincomp* g A = *fincomp* g $(A \cap B)$
    by (*rule fincomp-mono-neutral-cong-right*) (*use assms* **in** *auto*)
  also have ... = *fincomp* h $(A \cap B)$
    by (*rule fincomp-cong*) (*use assms* **in** *auto*)
  also have ... = *fincomp* h B
    by (*rule fincomp-mono-neutral-cong-left*) (*use assms* **in** *auto*)
  finally show *?thesis* **.**
qed

lemma *fincomp-UN-disjoint*:
  assumes
    finite I $\bigwedge i.\ i \in I \Longrightarrow$ *finite* $(A\ i)$ *pairwise* $(\lambda i\ j.$ *disjnt* $(A\ i)\ (A\ j))\ I$
    $\bigwedge i\ x.\ i \in I \Longrightarrow x \in A\ i \Longrightarrow g\ x \in M$
  shows *fincomp* g $(\bigcup(A\ `\ I))$ = *fincomp* $(\lambda i.$ *fincomp* g $(A\ i))$ I
  using *assms*
proof (*induction set*: *finite*)
  case *empty*
  then show *?case*
    by *force*
next
  case (*insert i I*)
  then show *?case*
    unfolding *pairwise-def disjnt-def*
    apply *clarsimp*
    apply (*subst fincomp-Un-disjoint*)
        apply (*fastforce intro*!: *funcsetI fincomp-closed*)+
    done
qed

lemma *fincomp-Union-disjoint*:
  $[\![$*finite* C; $\bigwedge A.\ A \in C \Longrightarrow$ *finite* $A \wedge (\forall x{\in}A.\ f\ x \in M)$; *pairwise disjnt* $C]\!] \Longrightarrow$
    fincomp f $(\bigcup C)$ = *fincomp* (*fincomp* f) C
  by (*frule fincomp-UN-disjoint* [*of C id f*]) *auto*

end

1.2 Results for Abelian Groups

context *abelian-group* **begin**

lemma *fincomp-inverse*:
  $f \in A \to G \Longrightarrow$ *fincomp* $(\lambda x.$ *inverse* $(f\ x))$ A = *inverse* (*fincomp* f A)
proof (*induct A rule*: *infinite-finite-induct*)
  case *empty* **show** *?case* **by** *simp*
next
  case (*insert a A*)
  then have $f\ a \in G$ $f \in A \to G$ $(\lambda x.$ *inverse* $(f\ x)) \in A \to G$

by *blast+*
with *insert* **show** *?case*
by (*simp add*: *commutative inverse-composition-commute*)
qed *simp*

Jeremy Avigad. This should be generalized to arbitrary groups, not just Abelian ones, using Lagrange's theorem.

lemma *power-order-eq-one*:
assumes *fin* [*simp*]: *finite G*
and *a* [*simp*]: $a \in G$
shows *rec-nat* **1** $(\lambda u.\ (\cdot)\ a)$ (*card G*) = **1**
proof −
have *rec-G*: *rec-nat* **1** $(\lambda u.\ (\cdot)\ a)$ (*card G*) $\in G$
by (*metis Pi-I′ a fincomp-closed fincomp-const*)
have $\bigwedge x.\ x \in G \Longrightarrow x \in (\cdot)\ a$ ‘ G
by (*metis a composition-closed imageI invertible invertible-inverse-closed invertible-right-inverse2*)
with *a* **have** $(\cdot)\ a$ ‘ $G = G$ **by** *blast*
then have **1** · *fincomp* $(\lambda x.\ x)$ G = *fincomp* $(\lambda x.\ x)$ $((\cdot)\ a$ ‘ $G)$
by *simp*
also have ... = *fincomp* $(\lambda x.\ a \cdot x)$ G
using *fincomp-reindex*
by (*subst* (*2*) *fincomp-reindex* [*symmetric*]) (*auto simp*: *inj-on-def*)
also have ... = *fincomp* $(\lambda x.\ a)$ G · *fincomp* $(\lambda x.\ x)$ G
by (*simp add*: *fincomp-comp*)
also have *fincomp* $(\lambda x.\ a)$ G = *rec-nat* **1** $(\lambda u.\ (\cdot)\ a)$ (*card G*)
by (*simp add*: *fincomp-const*)
finally show *?thesis*
by (*metis commutative fincomp-closed funcset-id invertible invertible-left-cancel rec-G unit-closed*)
qed

end

end

2 Khovanskii's Theorem

We formalise the proof of an important theorem in additive combinatorics due to Khovanskii, attesting that the cardinality of the set of all sums of n many elements of A, where A is a finite subset of an abelian group, is a polynomial in n for all sufficiently large n. We follow a proof due to Nathanson and Ruzsa as presented in the notes "Introduction to Additive Combinatorics" by Timothy Gowers for the University of Cambridge.

theory *Khovanskii*
imports
FiniteProduct

Pluennecke-Ruzsa-Inequality.Pluennecke-Ruzsa-Inequality
Bernoulli.Bernoulli — sums of a fixed power are polynomials
HOL−Analysis.Weierstrass-Theorems — needed for polynomial function
HOL−Library.List-Lenlexorder — lexicographic ordering for the type *nat list*
begin

The sum of the elements of a list

abbreviation $\sigma \equiv$ *sum-list*

Related to the nsets of Ramsey.thy, but simpler

definition *finsets* :: $['a\ set,\ nat] \Rightarrow 'a\ set\ set$
  where *finsets* $A\ n \equiv \{N.\ N \subseteq A \wedge card\ N = n\}$

lemma *card-finsets*: *finite* $N \Longrightarrow card\ (finsets\ N\ k) = card\ N\ choose\ k$
  by (*simp add*: *finsets-def n-subsets*)

lemma *sorted-map-plus-iff* [*simp*]:
  fixes *a*::'*a*::*linordered-cancel-ab-semigroup-add*
  shows *sorted* (*map* ((+) *a*) *xs*) $\longleftrightarrow$ *sorted xs*
  by (*induction xs*) *auto*

lemma *distinct-map-plus-iff* [*simp*]:
  fixes *a*::'*a*::*linordered-cancel-ab-semigroup-add*
  shows *distinct* (*map* ((+) *a*) *xs*) $\longleftrightarrow$ *distinct xs*
  by (*induction xs*) *auto*

2.1 Arithmetic operations on lists, pointwise on the elements

Weak type class properties. Cancellation is difficult to arrange because of complications when lists differ in length.

instantiation *list* :: (*plus*) *plus*
begin
definition *plus-list* $\equiv$ *map2* (+)
instance..
end

lemma *length-plus-list* [*simp*]:
  fixes *xs* :: '*a*::*plus list*
  shows *length* (*xs*+*ys*) = *min* (*length xs*) (*length ys*)
  by (*simp add*: *plus-list-def*)

lemma *plus-Nil* [*simp*]: [] + *xs* = []
  by (*simp add*: *plus-list-def*)

lemma *plus-Cons*: (*y* # *ys*) + (*x* # *xs*) = (*y*+*x*) # (*ys*+*xs*)
  by (*simp add*: *plus-list-def*)

lemma *nth-plus-list* [*simp*]:

fixes *xs* :: *'a::plus list*
assumes *i* < *length xs i* < *length ys*
shows (*xs*+*ys*)!*i* = *xs*!*i* + *ys*!*i*
by (*simp add*: *plus-list-def assms*)

instantiation *list* :: (*minus*) *minus*
begin
definition *minus-list* ≡ *map2* (−)
instance..
end

lemma *length-minus-list* [*simp*]:
fixes *xs* :: *'a::minus list*
shows *length* (*xs*−*ys*) = *min* (*length xs*) (*length ys*)
by (*simp add*: *minus-list-def*)

lemma *minus-Nil* [*simp*]: [] − *xs* = []
by (*simp add*: *minus-list-def*)

lemma *minus-Cons*: (*y* # *ys*) − (*x* # *xs*) = (*y*−*x*) # (*ys*−*xs*)
by (*simp add*: *minus-list-def*)

lemma *nth-minus-list* [*simp*]:
fixes *xs* :: *'a::minus list*
assumes *i* < *length xs i* < *length ys*
shows (*xs*−*ys*)!*i* = *xs*!*i* − *ys*!*i*
by (*simp add*: *minus-list-def assms*)

instance *list* :: (*ab-semigroup-add*) *ab-semigroup-add*
proof
have *map2* (+) (*map2* (+) *xs ys*) *zs* = *map2* (+) *xs* (*map2* (+) *ys zs*) **for** *xs ys zs* :: *'a list*
proof (*induction xs arbitrary*: *ys zs*)
case (*Cons x xs*)
show *?case*
proof (*cases ys*=[] ∨ *zs*=[])
case *False*
then obtain *y ys' z zs'* **where** *ys* = *y*#*ys' zs* = *z* # *zs'*
by (*meson list.exhaust*)
then show *?thesis*
by (*simp add*: *Cons add.assoc*)
qed *auto*
qed *auto*
then show *a* + *b* + *c* = *a* + (*b* + *c*) **for** *a b c* :: *'a list*
by (*auto simp*: *plus-list-def*)
next
have *map2* (+) *xs ys* = *map2* (+) *ys xs* **for** *xs ys* :: *'a list*
proof (*induction xs arbitrary*: *ys*)

case $(Cons\ x\ xs)$
show *?case*
proof $(cases\ ys)$
case $(Cons\ y\ ys')$
then show *?thesis*
by $(simp\ add{:}\ Cons.IH\ add.commute)$
qed *auto*
qed *auto*
then show $a + b = b + a$ **for** $a\ b :: 'a\ list$
by $(auto\ simp{:}\ plus\text{-}list\text{-}def)$
qed

2.2 The pointwise ordering on two equal-length lists of natural numbers

Gowers uses the usual symbol ($\leq$) for his pointwise ordering. In our development, $\leq$ is the lexicographic ordering and $\trianglelefteq$ is the pointwise ordering.

definition *pointwise-le* $:: [nat\ list,\ nat\ list] \Rightarrow bool$ (**infixr** ‹$\trianglelefteq$› *50*)
where $x \trianglelefteq y \equiv list\text{-}all2\ (\leq)\ x\ y$

definition *pointwise-less* $:: [nat\ list,\ nat\ list] \Rightarrow bool$ (**infixr** ‹$\triangleleft$› *50*)
where $x \triangleleft y \equiv x \trianglelefteq y \wedge x \neq y$

lemma *pointwise-le-iff-nth*:
$x \trianglelefteq y \longleftrightarrow length\ x = length\ y \wedge (\forall\, i < length\ x.\ x!i \leq y!i)$
by $(simp\ add{:}\ list\text{-}all2\text{-}conv\text{-}all\text{-}nth\ pointwise\text{-}le\text{-}def)$

lemma *pointwise-le-iff*:
$x \trianglelefteq y \longleftrightarrow length\ x = length\ y \wedge (\forall\,(i,j) \in set\ (zip\ x\ y).\ i{\leq}j)$
by $(simp\ add{:}\ list\text{-}all2\text{-}iff\ pointwise\text{-}le\text{-}def)$

lemma *pointwise-append-le-iff* $[simp]$: $u@x \trianglelefteq u@y \longleftrightarrow x \trianglelefteq y$
by $(auto\ simp{:}\ pointwise\text{-}le\text{-}iff\text{-}nth\ nth\text{-}append)$

lemma *pointwise-le-refl* $[iff]$: $x \trianglelefteq x$
by $(simp\ add{:}\ list.rel\text{-}refl\ pointwise\text{-}le\text{-}def)$

lemma *pointwise-le-antisym*: $[\![x \trianglelefteq y;\ y \trianglelefteq x]\!] \Longrightarrow x{=}y$
by $(metis\ antisym\ list\text{-}all2\text{-}antisym\ pointwise\text{-}le\text{-}def)$

lemma *pointwise-le-trans*: $[\![x \trianglelefteq y;\ y \trianglelefteq z]\!] \Longrightarrow x \trianglelefteq z$
by $(smt\ (verit,\ del\text{-}insts)\ le\text{-}trans\ list\text{-}all2\text{-}trans\ pointwise\text{-}le\text{-}def)$

lemma *pointwise-le-Nil* $[simp]$: $Nil \trianglelefteq x \longleftrightarrow x = Nil$
by $(simp\ add{:}\ pointwise\text{-}le\text{-}def)$

lemma *pointwise-le-Nil2* $[simp]$: $x \trianglelefteq Nil \longleftrightarrow x = Nil$
by $(simp\ add{:}\ pointwise\text{-}le\text{-}def)$

lemma *pointwise-le-iff-less-equal*: $x \trianglelefteq y \longleftrightarrow x \triangleleft y \vee x = y$
  using *pointwise-less-def* **by** *blast*

lemma *pointwise-less-iff*:
  $x \triangleleft y \longleftrightarrow x \trianglelefteq y \wedge (\exists (i,j) \in set\ (zip\ x\ y).\ i<j)$
  using *list-eq-iff-zip-eq pointwise-le-iff pointwise-less-def* **by** *fastforce*

lemma *pointwise-less-iff2*: $x \triangleleft y \longleftrightarrow x \trianglelefteq y \wedge (\exists k < length\ x.\ x!k < y\ !\ k)$
  unfolding *pointwise-less-def pointwise-le-iff-nth*
  by (*fastforce intro*!: *nth-equalityI*)

lemma *pointwise-less-Nil* [*simp*]: $\neg$ *Nil* $\triangleleft$ *x*
  by (*simp add*: *pointwise-less-def*)

lemma *pointwise-less-Nil2* [*simp*]: $\neg$ *x* $\triangleleft$ *Nil*
  by (*simp add*: *pointwise-less-def*)

lemma *zero-pointwise-le-iff* [*simp*]: $replicate\ r\ 0 \trianglelefteq x \longleftrightarrow length\ x = r$
  by (*auto simp*: *pointwise-le-iff-nth*)

lemma *pointwise-le-imp-σ*:
  assumes $xs \trianglelefteq ys$ **shows** $\sigma\ xs \le \sigma\ ys$
  using *assms*
proof (*induction ys arbitrary*: *xs*)
  case *Nil*
  then show *?case*
    by (*simp add*: *pointwise-le-iff*)
next
  case (*Cons y ys*)
  then obtain *x xs'* **where** $x \le y$ $xs = x\#xs'$ $xs' \trianglelefteq ys$
    by (*auto simp*: *pointwise-le-def list-all2-Cons2*)
  then show *?case*
    by (*simp add*: *Cons.IH add-le-mono*)
qed

lemma *sum-list-plus*:
  fixes *xs* :: *'a*::*comm-monoid-add list*
  assumes $length\ xs = length\ ys$ **shows** $\sigma\ (xs + ys) = \sigma\ xs + \sigma\ ys$
  using *assms* **by** (*simp add*: *plus-list-def case-prod-unfold sum-list-addf*)

lemma *sum-list-minus*:
  assumes $xs \trianglelefteq ys$ **shows** $\sigma\ (ys - xs) = \sigma\ ys - \sigma\ xs$
  using *assms*
proof (*induction ys arbitrary*: *xs*)
  case (*Cons y ys*)
  then obtain *x xs'* **where** $x \le y$ $xs = x\#xs'$ $xs' \trianglelefteq ys$
    by (*auto simp*: *pointwise-le-def list-all2-Cons2*)
  then show *?case*
    using *pointwise-le-imp-σ* **by** (*auto simp*: *Cons minus-Cons*)

qed (*auto simp*: *in-set-conv-nth*)

2.3 Pointwise minimum and maximum of a set of lists

definition *min-pointwise* :: $[nat, nat\ list\ set] \Rightarrow nat\ list$
where $min\text{-}pointwise \equiv \lambda r\ U.\ map\ (\lambda i.\ Min\ ((\lambda u.\ u!i)\ `\ U))\ [0..<r]$

lemma *min-pointwise-le*: $[\![u \in U; finite\ U]\!] \Longrightarrow min\text{-}pointwise\ (length\ u)\ U \trianglelefteq u$
by (*simp add*: *min-pointwise-def pointwise-le-iff-nth*)

lemma *min-pointwise-ge-iff*:
assumes $finite\ U\ U \neq \{\}\ \bigwedge u.\ u \in U \Longrightarrow length\ u = r\ length\ x = r$
shows $x \trianglelefteq min\text{-}pointwise\ r\ U \longleftrightarrow (\forall u \in U.\ x \trianglelefteq u)$
by (*auto simp*: *min-pointwise-def pointwise-le-iff-nth assms*)

definition *max-pointwise* :: $[nat, nat\ list\ set] \Rightarrow nat\ list$
where $max\text{-}pointwise \equiv \lambda r\ U.\ map\ (\lambda i.\ Max\ ((\lambda u.\ u!i)\ `\ U))\ [0..<r]$

lemma *max-pointwise-ge*: $[\![u \in U; finite\ U]\!] \Longrightarrow u \trianglelefteq max\text{-}pointwise\ (length\ u)\ U$
by (*simp add*: *max-pointwise-def pointwise-le-iff-nth*)

lemma *max-pointwise-le-iff*:
assumes $finite\ U\ U \neq \{\}\ \bigwedge u.\ u \in U \Longrightarrow length\ u = r\ length\ x = r$
shows $max\text{-}pointwise\ r\ U \trianglelefteq x \longleftrightarrow (\forall u \in U.\ u \trianglelefteq x)$
by (*auto simp*: *max-pointwise-def pointwise-le-iff-nth assms*)

lemma *max-pointwise-mono*:
assumes $X' \subseteq X\ finite\ X\ X' \neq \{\}$
shows $max\text{-}pointwise\ r\ X' \trianglelefteq max\text{-}pointwise\ r\ X$
using *assms* **by** (*simp add*: *max-pointwise-def pointwise-le-iff-nth Max-mono image-mono*)

lemma *pointwise-le-plus*: $[\![xs \trianglelefteq ys; length\ ys \le length\ zs]\!] \Longrightarrow xs \trianglelefteq ys+zs$
proof (*induction xs arbitrary*: *ys zs*)
case (*Cons x xs*)
then obtain $y\ ys'\ z\ zs'$ **where** $ys = y\#ys'\ zs = z\#zs'$
unfolding *pointwise-le-iff* **by** (*metis Suc-le-length-iff le-refl length-Cons*)
with *Cons* **show** *?case*
by (*auto simp*: *plus-list-def pointwise-le-def*)
qed (*simp add*: *pointwise-le-iff*)

lemma *pairwise-minus-cancel*: $[\![z \trianglelefteq x;\ z \trianglelefteq y;\ x - z = y - z]\!] \Longrightarrow x = y$
unfolding *pointwise-le-iff-nth* **by** (*metis eq-diff-iff nth-equalityI nth-minus-list*)

2.4 A locale to fix the finite subset $A \subseteq G$

locale *Khovanskii* = *additive-abelian-group* +
fixes $A :: 'a\ set$
assumes *AsubG*: $A \subseteq G$ **and** *finA*: *finite A*

begin

finite products of a group element

definition *Gmult* :: $'a \Rightarrow nat \Rightarrow 'a$
where $Gmult\ a\ n \equiv (((\oplus)a)\ \widehat{\ }\widehat{\ }\ n)\ \mathbf{0}$

lemma *Gmult-0* [*simp*]: $Gmult\ a\ 0 = \mathbf{0}$
by (*simp add*: *Gmult-def*)

lemma *Gmult-1* [*simp*]: $a \in G \Longrightarrow Gmult\ a\ (Suc\ 0) = a$
by (*simp add*: *Gmult-def*)

lemma *Gmult-Suc* [*simp*]: $Gmult\ a\ (Suc\ n) = a \oplus Gmult\ a\ n$
by (*simp add*: *Gmult-def*)

lemma *Gmult-in-G* [*simp,intro*]: $a \in G \Longrightarrow Gmult\ a\ n \in G$
by (*induction n*) *auto*

lemma *Gmult-add-add*:
assumes $a \in G$
shows $Gmult\ a\ (m+n) = Gmult\ a\ m \oplus Gmult\ a\ n$
by (*induction m*) (*use assms local.associative* **in** *fastforce*)+

lemma *Gmult-add-diff*:
assumes $a \in G$
shows $Gmult\ a\ (n+k) \ominus Gmult\ a\ n = Gmult\ a\ k$
by (*metis Gmult-add-add Gmult-in-G assms commutative inverse-closed invertible invertible-left-inverse2*)

lemma *Gmult-diff*:
assumes $a \in G\ n \le m$
shows $Gmult\ a\ m \ominus Gmult\ a\ n = Gmult\ a\ (m-n)$
by (*metis Gmult-add-diff assms le-add-diff-inverse*)

Mapping elements of A to their numeric subscript

abbreviation $idx \equiv$ *to-nat-on A*

The elements of A in order

definition *aA* :: *'a list*
where $aA \equiv$ *map* (*from-nat-into A*) [*0..<card A*]

definition α :: $nat\ list \Rightarrow 'a$
where $\alpha \equiv \lambda x.\ fincomp\ (\lambda i.\ Gmult\ (aA!i)\ (x!i))\ \{..<card\ A\}$

The underlying assumption is $length\ y = length\ x$

definition *useless*:: $nat\ list \Rightarrow bool$
where $useless\ x \equiv \exists\, y < x.\ \sigma\ y = \sigma\ x \wedge \alpha\ y = \alpha\ x \wedge length\ y = length\ x$

abbreviation $useful\ x \equiv \neg\ useless\ x$

lemma *alpha-replicate-0* [*simp*]: α (*replicate* (*card A*) *0*) = **0**
by (*auto simp*: α*-def intro*: *fincomp-unit-eqI*)

lemma *idx-less-cardA*:
assumes $a \in A$ **shows** *idx* $a <$ *card A*
by (*metis assms bij-betw-def finA imageI lessThan-iff to-nat-on-finite*)

lemma *aA-idx-eq* [*simp*]:
assumes $a \in A$ **shows** *aA* ! (*idx a*) = *a*
by (*simp add*: *aA-def assms countable-finite finA idx-less-cardA*)

lemma *set-aA*: *set aA* = *A*
using *bij-betw-from-nat-into-finite* [*OF finA*]
by (*simp add*: *aA-def atLeast0LessThan bij-betw-def*)

lemma *nth-aA-in-G* [*simp*]: $i <$ *card A* $\Longrightarrow$ *aA*!$i \in G$
using *AsubG aA-def set-aA* **by** *auto*

lemma *alpha-in-G* [*iff*]: $\alpha\ x \in G$
using *nth-aA-in-G fincomp-closed* **by** (*simp add*: α*-def*)

lemma *Gmult-in-PiG* [*simp*]: ($\lambda i.$ *Gmult* (*aA*!i) ($f\ i$)) $\in$ {..<*card A*} $\to G$
by *simp*

lemma *alpha-plus*:
assumes *length x* = *card A* *length y* = *card A*
shows $\alpha\ (x + y) = \alpha\ x \oplus \alpha\ y$
proof −
have $\alpha\ (x + y)$ = *fincomp* ($\lambda i.$ *Gmult* (*aA*!i) (*map2* (+) *x y*!i)) {..<*card A*}
by (*simp add*: α*-def plus-list-def*)
also have ... = *fincomp* ($\lambda i.$ *Gmult* (*aA*!i) (x!i + y!i)) {..<*card A*}
by (*intro fincomp-cong'*; *simp add*: *assms*)
also have ... = *fincomp* ($\lambda i.$ *Gmult* (*aA*!i) (x!i) $\oplus$ *Gmult* (*aA*!i) (y!i)) {..<*card A*}
by (*intro fincomp-cong'*; *simp add*: *Gmult-add-add*)
also have ... = $\alpha\ x \oplus \alpha\ y$
by (*simp add*: α*-def fincomp-comp*)
finally show *?thesis* **.**
qed

lemma *alpha-minus*:
assumes $y \trianglelefteq x$ *length y* = *card A*
shows $\alpha\ (x - y) = \alpha\ x \ominus \alpha\ y$
proof −
have $\alpha\ (x - y)$ = *fincomp* ($\lambda i.$ *Gmult* (*aA*!i) (*map2* (−) *x y*!i)) {..<*card A*}
by (*simp add*: α*-def minus-list-def*)
also have ... = *fincomp* ($\lambda i.$ *Gmult* (*aA*!i) (x!i − y!i)) {..<*card A*}
using *assms* **by** (*intro fincomp-cong'*) (*auto simp*: *pointwise-le-iff*)

also have ... = *fincomp* (λi. *Gmult* (*aA*!*i*) (*x*!*i*) $\ominus$ *Gmult* (*aA*!*i*) (*y*!*i*)) {..<*card* *A*}
  using *assms*
  by (*intro fincomp-cong'*) (*simp add*: *pointwise-le-iff-nth Gmult-diff*)+
also have ... = α *x* $\ominus$ α *y*
  by (*simp add*: α-*def fincomp-comp fincomp-inverse*)
finally show *?thesis* **.**
qed

2.5 Adding one to a list element

definition *list-incr* :: *nat* $\Rightarrow$ *nat list* $\Rightarrow$ *nat list*
  where *list-incr i x* $\equiv$ *x*[*i* := *Suc* (*x*!*i*)]

lemma *list-incr-Nil* [*simp*]: *list-incr i* [] = []
  by (*simp add*: *list-incr-def*)

lemma *list-incr-Cons* [*simp*]: *list-incr* (*Suc i*) (*k*#*ks*) = *k* # *list-incr i ks*
  by (*simp add*: *list-incr-def*)

lemma *sum-list-incr* [*simp*]: *i* < *length x* $\Longrightarrow$ σ (*list-incr i x*) = *Suc* (σ *x*)
  by (*auto simp*: *list-incr-def sum-list-update*)

lemma *length-list-incr* [*simp*]: *length* (*list-incr i x*) = *length x*
  by (*auto simp*: *list-incr-def*)

lemma *nth-le-list-incr*: *i* < *card A* $\Longrightarrow$ *x*!*i* $\leq$ *list-incr* (*idx a*) *x*!*i*
  unfolding *list-incr-def*
  by (*metis Suc-leD linorder-not-less list-update-beyond nth-list-update-eq nth-list-update-neq order-refl*)

lemma *list-incr-nth-diff*: *i* < *length x* $\Longrightarrow$ *list-incr j x*!*i* − *x*!*i* = (*if i* = *j then 1 else 0*)
  by (*simp add*: *list-incr-def*)

2.6 The set of all r-tuples that sum to n

definition *length-sum-set* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat list set*
  where *length-sum-set r n* $\equiv$ {*x*. *length x* = *r* $\wedge$ σ *x* = *n*}

lemma *length-sum-set-Nil* [*simp*]: *length-sum-set 0 n* = (*if n*=*0 then* {[]} *else* {})
  by (*auto simp*: *length-sum-set-def*)

lemma *length-sum-set-Suc* [*simp*]: *k*#*ks* $\in$ *length-sum-set* (*Suc r*) *n* $\longleftrightarrow$ ($\exists$ *m*. *ks* $\in$ *length-sum-set r m* $\wedge$ *n* = *m*+*k*)
  by (*auto simp*: *length-sum-set-def*)

lemma *length-sum-set-Suc-eqpoll*: *length-sum-set* (*Suc r*) *n* $\approx$ *Sigma* {..*n*} (λi. *length-sum-set r* (*n*−*i*)) (**is** *?L* $\approx$ *?R*)
  unfolding *eqpoll-def*

proof
  let *?f* = ($\lambda l.$ (*hd l*, *tl l*))
  show *bij-betw ?f ?L ?R*
  proof (*intro bij-betw-imageI*)
    show *inj-on ?f ?L*
      by (*force simp*: *inj-on-def length-sum-set-def intro*: *list.expand*)
    show *?f ‘ ?L = ?R*
      by (*force simp*: *length-sum-set-def length-Suc-conv*)
  qed
qed

lemma *finite-length-sum-set*: *finite* (*length-sum-set r n*)
proof (*induction r arbitrary*: *n*)
  case *0*
  then show *?case*
    by (*auto simp*: *length-sum-set-def*)
next
  case (*Suc r*)
  then show *?case*
    using *length-sum-set-Suc-eqpoll eqpoll-finite-iff* **by** *blast*
qed

lemma *card-length-sum-set*: *card* (*length-sum-set* (*Suc r*) *n*) = ($\sum i\le n.$ *card* (*length-sum-set* *r* ($n-i$)))
proof −
  have *card* (*length-sum-set* (*Suc r*) *n*) = *card* (*Sigma* {..*n*} ($\lambda i.$ *length-sum-set r* ($n-i$)))
    by (*metis eqpoll-finite-iff eqpoll-iff-card finite-length-sum-set length-sum-set-Suc-eqpoll*)
  also have ... = ($\sum i\le n.$ *card* (*length-sum-set r* ($n-i$)))
    by (*simp add*: *finite-length-sum-set*)
  finally show *?thesis* **.**
qed

lemma *sum-up-index-split'*:
  assumes $N \le n$ **shows** ($\sum i\le n.\ f\ i$) = ($\sum i\le n-N.\ f\ i$) + ($\sum i=Suc\ (n-N)..n.\ f\ i$)
  by (*metis assms diff-add sum-up-index-split*)

lemma *sum-invert*: $N \le n \Longrightarrow (\sum i = Suc\ (n - N)..n.\ f\ (n - i)) = (\sum j<N.\ f\ j)$
proof (*induction N*)
  case (*Suc N*)
  then show *?case*
    apply (*auto simp*: *Suc-diff-Suc*)
    by (*metis sum.atLeast-Suc-atMost Suc-leD add.commute diff-diff-cancel diff-le-self*)
qed *auto*

lemma *real-polynomial-function-length-sum-set*:
  $\exists p.$ *real-polynomial-function p* $\wedge$ ($\forall n>0.$ *real* (*card* (*length-sum-set r n*)) = *p*

(*real n*))
proof (*induction r*)
  case *0*
  have $\forall n>0.$ *real* (*card* (*length-sum-set 0 n*)) = *0*
    by *auto*
  then show *?case*
    by *blast*
next
  case (*Suc r*)
  then obtain *p* **where** *poly*: *real-polynomial-function p*
    and *p*: $\bigwedge n.\ n>0 \Longrightarrow$ *real* (*card* (*length-sum-set r n*)) = *p* (*real n*)
    by *blast*
  then obtain *a n* **where** *p-eq*: $p = (\lambda x. \sum i\le n.\ a\ i * x \mathbin{\hat{}} i)$
    using *real-polynomial-function-iff-sum* **by** *auto*
  define *q* **where** $q \equiv \lambda x. \sum j\le n.\ a\ j * ((bernpoly\ (Suc\ j)\ (1 + x) - bernpoly\ (Suc\ j)\ 0)$
$/ (1 + real\ j) - 0 \mathbin{\hat{}} j)$
  have *rp-q*: *real-polynomial-function q*
    by (*fastforce simp*: *bernpoly-def p-eq q-def*)
  have *q-eq*: $(\sum x\le k-1.\ p\ (k-x)) = q\ k$ **if** $k>0$ **for** *k*::*nat*
  proof −
    have $(\sum x\le k-1.\ p\ (k-x)) = (\sum j\le n.\ a\ j * ((\sum x\le k.\ real\ x \mathbin{\hat{}} j) - 0\hat{}j))$
      using *that*
      by (*simp add*: *p-eq sum.swap*
        flip: *sum-distrib-left of-nat-diff sum-diff-split*[**where** $f=\lambda i.\ real\ i \mathbin{\hat{}}$ -])
    also have ... = *q k*
      by (*simp add*: *sum-of-powers add.commute q-def*)
    finally show *?thesis* **.**
  qed
  define *p'* **where** $p' \equiv \lambda x.\ q\ x + real\ (card\ (length\text{-}sum\text{-}set\ r\ 0))$
  have *real-polynomial-function p'*
    using *rp-q* **by** (*force simp*: *p'-def*)
  moreover have $(\sum x\le n - Suc\ 0.\ p\ (real\ (n - x))) +$
        real (*card* (*length-sum-set r 0*)) = *p'* (*real n*) **if** $n>0$ **for** *n*
    using *that q-eq* **by** (*auto simp*: *p'-def*)
  ultimately show *?case*
    unfolding *card-length-sum-set*
    by (*force simp*: *sum-up-index-split'* [*of 1*] *p sum-invert*)
qed

lemma *all-zeroes-replicate*: *length-sum-set r 0* = {*replicate r 0*}
  by (*auto simp*: *length-sum-set-def replicate-eqI*)

lemma *length-sum-set-Suc-eq-UN*: *length-sum-set r* (*Suc n*) = ($\bigcup i<r.$ *list-incr i* ‘ *length-sum-set r n*)
proof −
  have $\exists i<r.\ x \in$ *list-incr i* ‘ *length-sum-set r n*
    if $\sigma\ x = Suc\ n$ **and** *r* = *length x* **for** *x*
  proof −

have $x \neq$ *replicate r 0*
using *that* **by** (*metis sum-list-replicate Zero-not-Suc mult-zero-right*)
then obtain *i* **where** *i*: $i < r$ $x!i \neq 0$
by (*metis* ‹r = *length x*› *in-set-conv-nth replicate-eqI*)
with *that* **have** $x[i := x!i - 1] \in$ *length-sum-set r n*
by (*simp add*: *sum-list-update length-sum-set-def*)
with *i that* **show** *?thesis*
unfolding *list-incr-def* **by** *force*
qed
then show *?thesis*
by (*auto simp*: *length-sum-set-def Bex-def*)
qed

lemma *alpha-list-incr*:
assumes $a \in A$ $x \in$ *length-sum-set* (*card A*) *n*
shows α (*list-incr* (*idx a*) *x*) = $a \oplus \alpha$ *x*
proof −
have *lenx*: *length x* = *card A*
using *assms length-sum-set-def* **by** *blast*
have α (*list-incr* (*idx a*) *x*) $\ominus$ α *x* = *fincomp* ($\lambda i.$ *Gmult* (*aA*!*i*) (*list-incr* (*idx a*) *x*!*i*) $\ominus$ *Gmult* (*aA*!*i*) (*x*!*i*)) {..<*card A*}
by (*simp add*: α*-def fincomp-comp fincomp-inverse*)
also have ... = *fincomp* ($\lambda i.$ *Gmult* (*aA*!*i*) (*list-incr* (*idx a*) *x*!*i* − *x*!*i*)) {..<*card A*}
by (*intro fincomp-cong*; *simp add*: *Gmult-diff nth-le-list-incr*)
also have ... = *fincomp* ($\lambda i.$ *if i* = *idx a then* (*aA*!*i*) *else* **0**) {..<*card A*}
by (*intro fincomp-cong*′; *simp add*: *list-incr-nth-diff lenx*)
also have ... = *a*
using *assms* **by** (*simp add*: *fincomp-singleton-swap idx-less-cardA*)
finally have α (*list-incr* (*idx a*) *x*) $\ominus$ α *x* = *a* **.**
then show *?thesis*
by (*metis alpha-in-G associative inverse-closed invertible invertible-left-inverse right-unit*)
qed

lemma *sumset-iterated-enum*:
defines $r \equiv$ *card A*
shows *sumset-iterated A n* = α ‘ *length-sum-set r n*
proof (*induction n*)
case *0*
then show *?case*
by (*simp add*: *all-zeroes-replicate r-def*)
next
case (*Suc n*)
have *eq*: {..<*r*} = *idx* ‘ *A*
by (*metis bij-betw-def finA r-def to-nat-on-finite*)
have *sumset-iterated A* (*Suc n*) = ($\bigcup a \in A.$ ($\lambda i.$ $a \oplus \alpha$ *i*) ‘ *length-sum-set r n*)
using *AsubG* **by** (*auto simp*: *Suc sumset*)
also have ... = ($\bigcup a \in A.$ ($\lambda i.$ α (*list-incr* (*idx a*) *i*)) ‘ *length-sum-set r n*)

by (*simp add*: *alpha-list-incr r-def*)
also have ... = α ‘ *length-sum-set r* (*Suc n*)
by (*simp add*: *image-UN image-comp length-sum-set-Suc-eq-UN eq*)
finally show *?case* .
qed

2.7 Lemma 2.7 in Gowers's notes

The following lemma corresponds to a key fact about the cardinality of the set of all sums of n many elements of A, stated before Gowers's Lemma 2.7.

lemma *card-sumset-iterated-length-sum-set-useful*:
defines $r \equiv$ *card A*
shows *card*(*sumset-iterated A n*) = *card* (*length-sum-set r n* $\cap$ {*x*. *useful x*})
(**is** *card ?L* = *card ?R*)
proof −
have α *x* $\in$ α ‘ (*length-sum-set r n* $\cap$ {*x*. *useful x*})
if *x* $\in$ *length-sum-set r n* **for** *x*
proof −
define *y* **where** *y* $\equiv$ *LEAST y*. *y* $\in$ *length-sum-set r n* $\wedge$ α *y* = α *x*
have *y*: *y* $\in$ *length-sum-set* (*card A*) *n* $\wedge$ α *y* = α *x*
by (*metis* (*mono-tags*, *lifting*) *LeastI r-def y-def that*)
moreover
have *useful y*
proof (*clarsimp simp*: *useless-def*)
show *False*
if σ *z* = σ *y length z* = *length y* **and** *z* < *y* α *z* = α *y* **for** *z*
using *that Least-le length-sum-set-def not-less-Least r-def y y-def* **by** *fastforce*
qed
ultimately show *?thesis*
unfolding *image-iff length-sum-set-def r-def* **by** (*smt* (*verit*) *Int-Collect*)
qed
then have *sumset-iterated A n* = α ‘ (*length-sum-set r n* $\cap$ {*x*. *useful x*})
by (*auto simp*: *sumset-iterated-enum length-sum-set-def r-def*)
moreover have *inj-on* α (*length-sum-set r n* $\cap$ {*x*. *useful x*})
apply (*simp add*: *image-iff length-sum-set-def r-def inj-on-def useless-def Ball-def*)
by (*metis linorder-less-linear*)
ultimately show *?thesis*
by (*simp add*: *card-image length-sum-set-def*)
qed

The following lemma corresponds to Lemma 2.7 in Gowers's notes.

lemma *useless-leq-useless*:
defines $r \equiv$ *card A*
assumes *useless x* **and** *x* $\trianglelefteq$ *y* **and** *length x* = *r*
shows *useless y*
proof −
have *leny*: *length y* = *r*
using *pointwise-le-iff assms* **by** *auto*

obtain x' **where** $x' < x$ **and** $\sigma x'$: $\sigma\ x' = \sigma\ x$ **and** $\alpha x'$: $\alpha\ x' = \alpha\ x$ **and** *lenx'*: *length* x' = *length* x
using *assms useless-def* **by** *blast*
obtain i **where** $i <$ *card* A **and** *xi*: $x'!i < x!i$ **and** *takex'*: *take* $i\ x'$ = *take* $i\ x$
using ‹$x' < x$› *lenx' assms* **by** (*auto simp*: *list-less-def lenlex-def elim*!: *lex-take-index*)
define y' **where** $y' \equiv y + x' - x$
have *leny'*: *length* y' = *length* y
using *assms lenx' pointwise-le-iff* **by** (*simp add*: *y'-def*)
have $x!i \le y!i$
using ‹$x \trianglelefteq y$› ‹$i <$ *card* A› *assms* **by** (*simp add*: *pointwise-le-iff-nth*)
then have $y'!i < y!i$
using ‹$i <$ *card* A› *assms lenx' xi pointwise-le-iff* **by** (*simp add*: *y'-def plus-list-def minus-list-def*)
moreover have *take* $i\ y'$ = *take* $i\ y$
proof (*intro nth-equalityI*)
show *length* (*take* $i\ y'$) = *length* (*take* $i\ y$)
by (*simp add*: *leny'*)
show $\bigwedge k.\ k <$ *length* (*take* $i\ y'$) $\Longrightarrow$ *take* $i\ y'\ !\ k$ = *take* $i\ y!k$
using *takex'* **by** (*simp add*: *y'-def plus-list-def minus-list-def take-map take-zip*)
qed
ultimately have $y' < y$
using *leny'* ‹$i <$ *card* A› *assms pointwise-le-iff*
by (*auto simp*: *list-less-def lenlex-def lexord-lex lexord-take-index-conv*)
moreover have $\sigma\ y' = \sigma\ y$
using *assms*
by (*simp add*: $\sigma x'$ *lenx' leny pointwise-le-plus sum-list-minus sum-list-plus y'-def*)
moreover have $\alpha\ y' = \alpha\ y$
using *assms lenx'* $\alpha x'$ *leny*
by (*fastforce simp*: *y'-def pointwise-le-plus alpha-minus alpha-plus local.associative*)
ultimately show *?thesis*
using *leny' useless-def* **by** *blast*
qed

inductive-set *minimal-elements* **for** U
where $[\![x \in U;\ \bigwedge y.\ y \in U \Longrightarrow \neg\ y \lhd x]\!] \Longrightarrow x \in$ *minimal-elements* U

lemma *pointwise-less-imp-*σ:
assumes $xs \lhd ys$ **shows** $\sigma\ xs < \sigma\ ys$
proof −
have *eq*: *length* ys = *length* xs **and** $xs \trianglelefteq ys$
using *assms* **by** (*auto simp*: *pointwise-le-iff pointwise-less-iff*)
have $\forall k <$ *length* $xs.\ xs!k \le ys!k$
using ‹$xs \trianglelefteq ys$› *list-all2-nthD pointwise-le-def* **by** *auto*
moreover have $\exists k <$ *length* $xs.\ xs!k < ys!k$
using *assms pointwise-less-iff2* **by** *force*
ultimately show *?thesis*

by (*force simp*: *eq sum-list-sum-nth intro*: *sum-strict-mono-ex1*)
qed

lemma *wf-measure-σ*: *wf* (*inv-image less-than σ*)
by *blast*

lemma *WFP*: *wfP* (◁)
by (*auto simp*: *wfp-def pointwise-less-imp-σ intro*: *wf-subset* [*OF wf-measure-σ*])

The following is a direct corollary of the above lemma, i.e. a corollary of Lemma 2.7 in Gowers's notes.

corollary *useless-iff*:
assumes *length x* = *card A*
shows *useless x* ⟷ (∃ *x'* ∈ *minimal-elements* (*Collect useless*). *x'* ⊴ *x*) (**is** -=*?R*)
proof
assume *useless x*
obtain *z* **where** *z*: *useless z z* ⊴ *x* **and** *zmin*: ⋀*y*. *y* ◁ *z* ⟹ *y* ⊴ *x* ⟹ *useful y*
using *wfE-min* [*to-pred*, **where** *Q* = {*z*. *useless z* ∧ *z* ⊴ *x*}, *OF WFP*]
by (*metis* (*no-types*, *lifting*) ‹*useless x*› *mem-Collect-eq pointwise-le-refl*)
then show *?R*
by (*smt* (*verit*) *mem-Collect-eq minimal-elements.intros pointwise-le-trans pointwise-less-def*)
next
assume *?R*
with *useless-leq-useless minimal-elements.cases* **show** *useless x*
by (*metis assms mem-Collect-eq pointwise-le-iff*)
qed

2.8 The set of minimal elements of a set of r-tuples is finite

The following general finiteness claim corresponds to Lemma 2.8 in Gowers's notes and is key to the main proof.

lemma *minimal-elements-set-tuples-finite*:
assumes *Ur*: ⋀*x*. *x* ∈ *U* ⟹ *length x* = *r*
shows *finite* (*minimal-elements U*)
using *assms*
proof (*induction r arbitrary*: *U*)
case *0*
then have *U* ⊆ {[]}
by *auto*
then show *?case*
by (*metis finite.simps minimal-elements.cases finite-subset subset-eq*)
next
case (*Suc r*)
show *?case*
proof (*cases U*={})
case *True*
with *Suc.IH* **show** *?thesis* **by** *blast*

```
  next
    case False
    then obtain u where u: u ∈ U and zmin: ⋀y. y ◁ u ⟹ y ∉ U
      using wfE-min [to-pred, where Q = U, OF WFP] by blast
    define V where V = {v ∈ U. ¬ u ⊴ v}
    define VF where VF ≡ λi t. {v ∈ V. v!i = t}
    have [simp]: length v = Suc r if v ∈ VF i t for v i t
      using that by (simp add: Suc.prems VF-def V-def)
    have ∗: ∃ i≤r. v!i < u!i if v ∈ V for v
      using that u Suc.prems
      by (force simp: V-def pointwise-le-iff-nth not-le less-Suc-eq-le)
   with u have minimal-elements U ≤ insert u (⋃ i≤r. ⋃ t < u!i. minimal-elements (VF i t))
      by (force simp: VF-def V-def minimal-elements.simps pointwise-less-def)
    moreover
    have finite (minimal-elements (VF i t)) if i≤r t < u!i for i t
    proof −
     define delete where delete ≡ λv::nat list. take i v @ drop (Suc i) v — deletion of i
      have len-delete[simp]: length (delete u) = r if u ∈ VF i t for u
        using Suc.prems VF-def V-def ‹i ≤ r› delete-def that by auto
      have nth-delete: delete u!k = (if k<i then u!k else u!Suc k) if u ∈ VF i t k<r for u k
        using that by (simp add: delete-def nth-append)
      have delete-le-iff [simp]: delete u ⊴ delete v ⟷ u ⊴ v if u ∈ VF i t v ∈ VF i t for u v
      proof
        assume delete u ⊴ delete v
         then have ∀ j. (j < i ⟶ u!j ≤ v!j) ∧ (j < r ⟶ i ≤ j ⟶ u!Suc j ≤ v!Suc j)
          using that ‹i ≤ r›
             by (force simp: pointwise-le-iff-nth nth-delete split: if-split-asm cong: conj-cong)
        then show u ⊴ v
          using that ‹i ≤ r›
          apply (simp add: pointwise-le-iff-nth VF-def)
          by (metis eq-iff le-Suc-eq less-Suc-eq-0-disj linorder-not-less)
      next
        assume u ⊴ v then show delete u ⊴ delete v
          using that by (simp add: pointwise-le-iff-nth nth-delete)
      qed
      then have delete-eq-iff: delete u = delete v ⟷ u = v if u ∈ VF i t v ∈ VF i t for u v
        by (metis that pointwise-le-antisym pointwise-le-refl)
      have delete-less-iff: delete u ◁ delete v ⟷ u ◁ v if u ∈ VF i t v ∈ VF i t for u v
        by (metis delete-le-iff pointwise-le-antisym pointwise-less-def that)
      have length (delete v) = r if v ∈ V for v
        using id-take-nth-drop Suc.prems V-def ‹i ≤ r› delete-def that by auto
```

then have *finite* (*minimal-elements* (*delete* ‘ *V*))
by (*metis* (*mono-tags*, *lifting*) *Suc.IH image-iff*)
moreover have *inj-on delete* (*minimal-elements* (*VF i t*))
by (*simp add*: *delete-eq-iff inj-on-def minimal-elements.simps*)
moreover have *delete* ‘ (*minimal-elements* (*VF i t*)) ⊆ *minimal-elements* (*delete* ‘ (*VF i t*))
by (*auto simp*: *delete-less-iff minimal-elements.simps*)
ultimately show *?thesis*
by (*metis* (*mono-tags*, *lifting*) *Suc.IH image-iff inj-on-finite len-delete*)
qed
ultimately show *?thesis*
by (*force elim*: *finite-subset*)
qed
qed

2.9 Towards Lemma 2.9 in Gowers's notes

Increasing sequences

fun *augmentum* :: *nat list* ⇒ *nat list*
where *augmentum* [] = []
| *augmentum* (*n*#*ns*) = *n* # *map* ((+)*n*) (*augmentum ns*)

definition *dementum*:: *nat list* ⇒ *nat list*
where *dementum xs* ≡ *xs* − (*0*#*xs*)

lemma *dementum-Nil* [*simp*]: *dementum* [] = []
by (*simp add*: *dementum-def*)

lemma *zero-notin-augmentum* [*simp*]: *0* ∉ *set ns* ⟹ *0* ∉ *set* (*augmentum ns*)
by (*induction ns*) *auto*

lemma *length-augmentum* [*simp*]:*length* (*augmentum xs*) = *length xs*
by (*induction xs*) *auto*

lemma *sorted-augmentum* [*simp*]: *0* ∉ *set ns* ⟹ *sorted* (*augmentum ns*)
by (*induction ns*) *auto*

lemma *distinct-augmentum* [*simp*]: *0* ∉ *set ns* ⟹ *distinct* (*augmentum ns*)
by (*induction ns*) (*simp-all add*: *image-iff*)

lemma *augmentum-subset-sum-list*: *set* (*augmentum ns*) ⊆ {..σ *ns*}
by (*induction ns*) *auto*

lemma *sum-list-augmentum*: σ *ns* ∈ *set* (*augmentum ns*) ⟷ *length ns* > *0*
by (*induction ns*) *auto*

lemma *length-dementum* [*simp*]: *length* (*dementum xs*) = *length xs*
by (*simp add*: *dementum-def*)

lemma *sorted-imp-pointwise*:
  assumes *sorted* $(xs@[n])$
  shows $0 \# xs \trianglelefteq xs @ [n]$
  using *assms*
  by (*simp add*: *pointwise-le-iff-nth nth-Cons′ nth-append sorted-append sorted-wrt-append sorted-wrt-nth-less*)

lemma *sum-list-dementum*:
  assumes *sorted* $(xs@[n])$
  shows σ (*dementum* $(xs@[n])) = n$
proof −
  have *dementum* $(xs@[n]) = (xs@[n]) - (0 \# xs)$
    by (*rule nth-equalityI*; *simp add*: *nth-append dementum-def nth-Cons′*)
  then show *?thesis*
    by (*simp add*: *sum-list-minus sorted-imp-pointwise assms*)
qed

lemma *augmentum-cancel*: *map* $((+)k)$ (*augmentum ns*) − $(k \#$ *map* $((+)k)$ (*augmentum ns*)) = *ns*
proof (*induction ns arbitrary*: *k*)
  case *Nil*
  then show *?case*
    by *simp*
next
  case (*Cons n ns*)
  have $(+)\ k \circ (+)\ n = (+)\ (k+n)$ **by** *auto*
  then show *?case*
    by (*simp add*: *minus-Cons Cons*)
qed

lemma *dementum-augmentum* [*simp*]:
  assumes $0 \notin$ *set ns*
  shows (*dementum* ∘ *sorted-list-of-set*) ((*set* ∘ *augmentum*) *ns*) = *ns* (**is** *?L ns* = -)
  using *assms augmentum-cancel* [*of 0*]
  by (*simp add*: *dementum-def map-idI sorted-list-of-set.idem-if-sorted-distinct*)

lemma *dementum-nonzero*:
  assumes *ns*: *sorted-wrt* $(<)$ *ns* **and** *0*: $0 \notin$ *set ns*
  shows $0 \notin$ *set* (*dementum ns*)
  unfolding *dementum-def minus-list-def*
  using *sorted-wrt-nth-less* [*OF ns*] *0*
  by (*auto simp*: *in-set-conv-nth image-iff set-zip nth-Cons′ dest*: *leD*)

lemma *nth-augmentum* [*simp*]: $i <$ *length ns* $\Longrightarrow$ *augmentum ns*$!i = (\sum j{\le}i.\ ns!j)$
proof (*induction ns arbitrary*: *i*)
  case *Nil*
  then show *?case*
    by *simp*

```
next
  case (Cons a ns)
  show ?case
  proof (cases i=0)
    case False
    then have augmentum (a # ns)!i = a + sum ((!) ns) {..i−1}
      using Cons.IH Cons.prems by auto
    also have ... = a + (∑j∈{0<..i}. ns!(j−1))
      using sum.reindex [of Suc {..i − Suc 0} λj. ns!(j−1), symmetric] False
       by (simp add: image-Suc-atMost atLeastSucAtMost-greaterThanAtMost del:
sum.cl-ivl-Suc)
    also have ... = (∑j = 0..i. if j=0 then a else ns!(j−1))
      by (simp add: sum.head)
    also have ... = sum ((!) (a # ns)) {..i}
      by (simp add: nth-Cons' atMost-atLeast0)
    finally show ?thesis .
  qed auto
qed

lemma augmentum-dementum [simp]:
  assumes 0 ∉ set ns sorted ns
  shows augmentum (dementum ns) = ns
proof (rule nth-equalityI)
  fix i
  assume i < length (augmentum (dementum ns))
  then have i: i < length ns
    by simp
  show augmentum (dementum ns)!i = ns!i
  proof (cases i=0)
    case True
    then show ?thesis
      using nth-augmentum dementum-def i by auto
  next
    case False
    have ns-le: ⋀j. ⟦0 < j; j ≤ i⟧ ⟹ ns ! (j − Suc 0) ≤ ns ! j
      using ‹sorted ns› i by (simp add: sorted-iff-nth-mono)
     have augmentum (dementum ns)!i = (∑j≤i. ns!j − (if j = 0 then 0 else
ns!(j−1)))
      using i by (simp add: dementum-def nth-Cons')
    also have ... = (∑j=0..i. if j = 0 then ns!0 else ns!j − ns!(j−1))
      by (smt (verit, del-insts) diff-zero sum.cong atMost-atLeast0)
    also have ... = ns!0 + (∑j∈{0<..i}. ns!j − ns!(j−1))
      by (simp add: sum.head)
    also have ... = ns!0 + ((∑j∈{0<..i}. ns!j) − (∑j∈{0<..i}. ns!(j−1)))
      by (auto simp: ns-le intro: sum-subtractf-nat)
    also have ... = ns!0 + (∑j∈{0<..i}. ns!j) − (∑j∈{0<..i}. ns!(j−1))
    proof −
      have (∑j∈{0<..i}. ns ! (j − 1)) ≤ sum ((!) ns) {0<..i}
        by (metis One-nat-def greaterThanAtMost-iff ns-le sum-mono)
```

then show *?thesis* **by** *simp*
qed
also have ... = $ns!0$ + ($\sum j{\in}\{0{<}..i\}$. $ns!j$) − ($\sum j{\le}i{-}Suc\ 0$. $ns!j$)
using *sum.reindex* [*of Suc* {..*i* − *Suc 0*} λj. *ns*!(*j*−*1*), *symmetric*] *False*
by (*simp add*: *image-Suc-atMost atLeastSucAtMost-greaterThanAtMost*)
also have ... = ($\sum j{=}0..i$. $ns!j$) − ($\sum j{\le}i{-}Suc\ 0$. $ns!j$)
by (*simp add*: *sum.head* [*of 0 i*])
also have ... = ($\sum j{=}0..i{-}Suc\ 0$. $ns!j$) + $ns!i$ − ($\sum j{\le}i{-}Suc\ 0$. $ns!j$)
by (*metis False Suc-pred less-Suc0 not-less-eq sum.atLeast0-atMost-Suc*)
also have ... = *ns*!*i*
by (*simp add*: *atLeast0AtMost*)
finally show *augmentum* (*dementum ns*)!*i* = *ns*!*i* **.**
qed
qed *auto*

The following lemma corresponds to Lemma 2.9 in Gowers's notes. The proof involves introducing bijective maps between r-tuples that fulfill certain properties/r-tuples and subsets of naturals, so as to show the cardinality claim.

lemma *bound-sum-list-card*:
assumes $r > 0$ **and** *n*: $n \ge \sigma\ x'$ **and** *lenx'*: *length* $x' = r$
defines $S \equiv \{x.\ x' \trianglelefteq x \wedge \sigma\ x = n\}$
shows *card S* = ($n - \sigma\ x' + r - 1$) *choose* ($r{-}1$)
proof−
define *m* **where** $m \equiv n - \sigma\ x'$
define *f* **where** $f \equiv \lambda x{::}nat\ list.\ x - x'$
have *f*: *bij-betw f S* (*length-sum-set r m*)
proof (*intro bij-betw-imageI*)
show *inj-on f S*
using *pairwise-minus-cancel* **by** (*force simp*: *S-def f-def inj-on-def*)
have $\bigwedge x.\ x \in S \Longrightarrow f\ x \in$ *length-sum-set r m*
by (*simp add*: *S-def f-def length-sum-set-def lenx' m-def pointwise-le-iff sum-list-minus*)
moreover have $x \in f$ ‘ S **if** $x \in$ *length-sum-set r m* **for** *x*
proof
have *x*[*simp*]: *length* $x = r$ $\sigma\ x = m$
using *that* **by** (*auto simp*: *length-sum-set-def*)
have $x = x' + x - x'$
by (*rule nth-equalityI*; *simp add*: *lenx'*)
then show $x = f\ (x' + x)$
unfolding *f-def* **by** *fastforce*
have $x' \trianglelefteq x' + x$
by (*simp add*: *lenx' pointwise-le-plus*)
moreover have $\sigma\ (x' + x) = n$
by (*simp add*: *lenx' m-def n sum-list-plus*)
ultimately show $x' + x \in S$
using *S-def* **by** *blast*
qed
ultimately show f ‘ S = *length-sum-set r m* **by** *auto*

qed
define *g* **where** *g ≡ λx::nat list. map Suc x*
define *g′* **where** *g′ ≡ λx::nat list. x − replicate (length x) 1*
define *T* **where** *T ≡ length-sum-set r (m+r) ∩ lists(−{0})*
have *g*: *bij-betw g (length-sum-set r m) T*
proof (*intro bij-betw-imageI*)
show *inj-on g (length-sum-set r m)*
by (*auto simp*: *g-def inj-on-def*)
have *⋀x. x ∈ length-sum-set r m ⟹ g x ∈ T*
by (*auto simp*: *g-def length-sum-set-def sum-list-Suc T-def*)
moreover have *x ∈ g ‘ length-sum-set r m* **if** *x ∈ T* **for** *x*
proof
have [*simp*]: *length x = r*
using *length-sum-set-def that T-def* **by** *auto*
have *r1-x*: *replicate r (Suc 0) ⊴ x*
using *that* **unfolding** *T-def pointwise-le-iff-nth*
by (*simp add*: *lists-def in-listsp-conv-set Suc-leI*)
show *x = g (g′ x)*
using *that* **by** (*intro nth-equalityI*) (*auto simp*: *g-def g′-def T-def*)
show *g′ x ∈ length-sum-set r m*
using *that T-def* **by** (*simp add*: *g′-def r1-x sum-list-minus length-sum-set-def sum-list-replicate*)
qed
ultimately show *g ‘ (length-sum-set r m) = T* **by** *auto*
qed
define *U* **where** *U ≡ (insert (m+r)) ‘ finsets {0<..<m+r} (r−1)*
have *h*: *bij-betw (set ∘ augmentum) T U*
proof (*intro bij-betw-imageI*)
show *inj-on ((set ∘ augmentum)) T*
unfolding *inj-on-def T-def*
by (*metis ComplD IntE dementum-augmentum in-listsD insertI1*)
have (*set ∘ augmentum*) *t ∈ U* **if** *t ∈ T* **for** *t*
proof −
have *t*: *length t = r σ t = m+r 0 ∉ set t*
using *that* **by** (*force simp*: *T-def length-sum-set-def*)+
then have *mrt*: *m + r ∈ set (augmentum t)*
by (*metis ‹r>0› sum-list-augmentum*)
then have *set (augmentum t) = insert (m + r) (set (augmentum t) − {m + r})*
by *blast*
moreover have *set (augmentum t) − {m + r} ∈ finsets {0<..<m + r} (r − Suc 0)*
apply (*auto simp*: *finsets-def mrt distinct-card t*)
by (*metis atMost-iff augmentum-subset-sum-list le-eq-less-or-eq subsetD t(2)*)
ultimately show *?thesis*
by (*metis One-nat-def U-def comp-apply imageI*)
qed
moreover have *u ∈ (set ∘ augmentum) ‘ T* **if** *u ∈ U* **for** *u*
proof

from *that*
obtain *N* **where** *u*: *u* = *insert* (*m* + *r*) *N* **and** *Nsub*: $N \subseteq \{0<..<m + r\}$
and [*simp*]: *card N* = *r* − *Suc 0*
by (*auto simp*: *U-def finsets-def*)
have [*simp*]: *0* $\notin$ *N m+r* $\notin$ *N finite N*
using *finite-subset Nsub* **by** *auto*
have [*simp*]: *card u* = *r*
using *Nsub* ‹*r*>*0*› **by** (*auto simp*: *u card-insert-if*)
have *ssN*: *sorted* (*sorted-list-of-set N* @ [*m* + *r*])
using *Nsub* **by** (*simp add*: *less-imp-le-nat sorted-wrt-append subset-eq*)
have *so-u-N*: *sorted-list-of-set u* = *insort* (*m*+*r*) (*sorted-list-of-set N*)
by (*simp add*: *u*)
also have . . . = *sorted-list-of-set N* @ [*m*+*r*]
using *Nsub* **by** (*force intro*: *sorted-insort-is-snoc*)
finally have *so-u*: *sorted-list-of-set u* = *sorted-list-of-set N* @ [*m*+*r*] **.**
have *0*: *0* $\notin$ *set* (*sorted-list-of-set u*)
by (*simp add*: ‹*r*>*0*› *set-insort-key so-u-N*)
show *u* = (*set* $\circ$ *augmentum*) ((*dementum* $\circ$ *sorted-list-of-set*)*u*)
using *0 so-u ssN u* **by** *force*
have *sortd-wrt-u*: *sorted-wrt* (<) (*sorted-list-of-set u*)
by *simp*
show (*dementum* $\circ$ *sorted-list-of-set*) *u* $\in$ *T*
apply (*simp add*: *T-def length-sum-set-def*)
using *sum-list-dementum* [*OF ssN*] *sortd-wrt-u 0* **by** (*force simp*: *so-u dementum-nonzero*)+
qed
ultimately show (*set* $\circ$ *augmentum*) ‘ *T* = *U* **by** *auto*
qed
obtain φ **where** *bij-betw* φ *S U*
by (*meson bij-betw-trans f g h*)
moreover have *card U* = $(n - \sigma\ x' + r{-}1)$ *choose* (*r*−*1*)
proof −
have *inj-on* (*insert* (*m* + *r*)) (*finsets* $\{0<..<m + r\}$ (*r* − *Suc 0*))
by (*simp add*: *inj-on-def finsets-def subset-iff*) (*meson insert-ident order-less-le*)
then have *card U* = *card* (*finsets* $\{0<..<m + r\}$ (*r* − *1*))
unfolding *U-def* **by** (*simp add*: *card-image*)
also have . . . = $(n - \sigma\ x' + r{-}1)$ *choose* (*r*−*1*)
by (*simp add*: *card-finsets m-def*)
finally show *?thesis* **.**
qed
ultimately show *?thesis*
by (*metis bij-betw-same-card*)
qed

2.10 Towards the main theorem

lemma *extend-tuple*:
assumes σ *xs* $\le$ *n length xs* $\neq$ *0*
obtains *ys* **where** σ *ys* = *n xs* $\trianglelefteq$ *ys*

proof −
  obtain x xs' **where** xs: $xs = x\#xs'$
    using *assms list.exhaust* **by** *auto*
  define y **where** $y \equiv x + n - \sigma\ xs$
  show *thesis*
  proof
    show $\sigma\ (y\#xs') = n$
      using *assms xs y-def* **by** *auto*
    show $xs \trianglelefteq y\#xs'$
      using *assms y-def pointwise-le-def xs* **by** *auto*
  qed
qed

lemma *extend-preserving*:
  assumes $\sigma\ xs \leq n$ $length\ xs > 1$ $i < length\ xs$
  obtains ys **where** $\sigma\ ys = n$ $xs \trianglelefteq ys$ $ys!i = xs!i$
proof −
  define j **where** $j \equiv Suc\ i\ mod\ length\ xs$
  define $xs1$ **where** $xs1 = take\ j\ xs$
  define $xs2$ **where** $xs2 = drop\ (Suc\ j)\ xs$
  define x **where** $x = xs!j$
  have xs: $xs = xs1\ @\ [x]\ @\ xs2$
    using *assms*
    apply (*simp add*: *Cons-nth-drop-Suc assms x-def xs1-def xs2-def j-def*)
    by (*meson Suc-lessD id-take-nth-drop mod-less-divisor*)
  define y **where** $y \equiv x + n - \sigma\ xs$
  define ys **where** $ys \equiv xs1\ @\ [y]\ @\ xs2$
  have $x \leq y$
    using *assms y-def* **by** *linarith*
  show *thesis*
  proof
    show $\sigma\ ys = n$
      using *assms*(*1*) *xs y-def ys-def* **by** *auto*
    show $xs \trianglelefteq ys$
      using *xs ys-def* ‹$x \leq y$› *pointwise-append-le-iff pointwise-le-def* **by** *fastforce*
    have $length\ xs1 \neq i$
      using *assms* **by** (*simp add*: *xs1-def j-def min-def mod-Suc*)
    then show $ys!i = xs!i$
      by (*auto simp*: *ys-def xs nth-append nth-Cons'*)
  qed
qed

The proof of the main theorem will make use of the inclusion-exclusion formula, in addition to the previously shown results.

theorem *Khovanskii*:
  assumes $card\ A > 1$
  defines $f \equiv \lambda n.\ card(sumset\text{-}iterated\ A\ n)$
  obtains N p **where** $real\text{-}polynomial\text{-}function\ p$ $\bigwedge n.\ n \geq N \Longrightarrow real\ (f\ n) = p\ (real\ n)$

proof $-$
define r **where** $r \equiv \mathit{card}\ A$
define C **where** $C \equiv \lambda n\ x'.\ \{x.\ x' \trianglelefteq x \wedge \sigma\ x = n\}$
define X **where** $X \equiv \mathit{minimal\text{-}elements}\ \{x.\ \mathit{useless}\ x \wedge \mathit{length}\ x = r\}$
have $r > \mathit{1}\ r \neq \mathit{0}$
using *assms r-def* **by** *auto*
have *Csub*: $C\ n\ x' \subseteq \mathit{length\text{-}sum\text{-}set}\ (\mathit{length}\ x')\ n$ **for** $n\ x'$
by (*auto simp*: *C-def length-sum-set-def pointwise-le-iff*)
then have *finC*: $\mathit{finite}\ (C\ n\ x')$ **for** $n\ x'$
by (*meson finite-length-sum-set finite-subset*)
have *finite X*
using *minimal-elements-set-tuples-finite X-def* **by** *force*
then have *max-X*: $\bigwedge x'.\ x' \in X \Longrightarrow\ \sigma\ x' \leq \sigma\ (\mathit{max\text{-}pointwise}\ r\ X)$
using *X-def max-pointwise-ge minimal-elements.simps pointwise-le-imp-*σ **by** *force*
let $?z0 = \mathit{replicate}\ r\ \mathit{0}$
have *Cn0*: $C\ n\ ?z0 = \mathit{length\text{-}sum\text{-}set}\ r\ n$ **for** n
by (*auto simp*: *C-def length-sum-set-def*)
then obtain *p0* **where** *pf-p0*: *real-polynomial-function p0* **and** *p0*: $\bigwedge n.\ n>0 \Longrightarrow p0\ (\mathit{real}\ n) = \mathit{real}\ (\mathit{card}\ (C\ n\ ?z0))$
by (*metis real-polynomial-function-length-sum-set*)
obtain q **where** *pf-q*: *real-polynomial-function q* **and** q: $\bigwedge x.\ q\ x = x\ \mathit{gchoose}\ (r-\mathit{1})$
using *real-polynomial-function-gchoose* **by** *metis*
define p **where** $p \equiv \lambda x{::}\mathit{real}.\ p0\ x - (\sum Y \mid Y \subseteq X \wedge Y \neq \{\}.\ (-\ \mathit{1})\ \hat{}\ (\mathit{card}\ Y + \mathit{1}) * q((x - \mathit{real}(\sigma\ (\mathit{max\text{-}pointwise}\ r\ Y)) + \mathit{real}\ r - \mathit{1})))$
show *thesis*
proof
note $\mathit{pf\text{-}q}' = \mathit{real\text{-}polynomial\text{-}function\text{-}compose}$ [*OF - pf-q*, *unfolded o-def*]
note *pf-intros* = *real-polynomial-function-sum real-polynomial-function-diff real-polynomial-function.intros*
show *real-polynomial-function p*
unfolding *p-def* **using** ‹*finite X*› **by** (*intro pf-p0* $\mathit{pf\text{-}q}'$ *pf-intros* | *force*)+
next
fix n
assume $n \geq \mathit{max}\ \mathit{1}\ (\sigma\ (\mathit{max\text{-}pointwise}\ r\ X))$
then have *nlarge*: $n \geq \sigma\ (\mathit{max\text{-}pointwise}\ r\ X)$ **and** $n > \mathit{0}$
by *auto*
define U **where** $U \equiv \lambda n.\ \mathit{length\text{-}sum\text{-}set}\ r\ n \cap \{x.\ \mathit{useful}\ x\}$
have *2*: $(\mathit{length\text{-}sum\text{-}set}\ r\ n \cap \{x.\ \mathit{useless}\ x\}) = (\bigcup x'{\in}X.\ C\ n\ x')$
unfolding *C-def X-def length-sum-set-def r-def*
using *useless-leq-useless* **by** (*force simp*: *minimal-elements.simps pointwise-le-iff useless-iff*)
define *SUM1* **where** $\mathit{SUM1} \equiv \sum I \mid I \subseteq C\ n\ \text{`}\ X \wedge I \neq \{\}.\ (-\ \mathit{1})\ \hat{}\ (\mathit{card}\ I + \mathit{1}) * \mathit{int}\ (\mathit{card}\ (\bigcap I))$
define *SUM2* **where** $\mathit{SUM2} \equiv \sum Y \mid Y \subseteq X \wedge Y \neq \{\}.\ (-\ \mathit{1})\ \hat{}\ (\mathit{card}\ Y + \mathit{1}) * \mathit{int}\ (\mathit{card}\ (\bigcap (C\ n\ \text{`}\ Y)))$
have *SUM1-card*: $\mathit{card}(\mathit{length\text{-}sum\text{-}set}\ r\ n \cap \{x.\ \mathit{useless}\ x\}) = \mathit{nat}\ \mathit{SUM1}$
unfolding *SUM1-def 2* **using** ‹*finite X*› *finC* **by** (*intro card-UNION*; *force*)

have *SUM1* $\geq$ *0*
unfolding *SUM1-def* **using** *card-UNION-nonneg finC* ‹*finite X*› **by** *auto*
have *C-empty-iff*: $C\ n\ x' = \{\} \longleftrightarrow \sigma\ x' > n$ **if** *length* $x' \neq 0$ **for** x'
by (*simp add*: *set-eq-iff C-def*) (*meson extend-tuple linorder-not-le pointwise-le-imp-σ that*)
have *C-eq-1*: $C\ n\ x' = \{[n]\}$ **if** $\sigma\ x' \leq n$ *length* $x' = 1$ **for** x'
using *that* **by** (*auto simp*: *C-def length-Suc-conv pointwise-le-def elim*!: *list.rel-cases*)
have *n-ge-X*: $\sigma\ x \leq n$ **if** $x \in X$ **for** x
by (*meson le-trans max-X nlarge that*)
have *len-X-r*: $x \in X \Longrightarrow$ *length* $x = r$ **for** x
by (*auto simp*: *X-def minimal-elements.simps*)

have *min-pointwise* $r\ (C\ n\ x') = x'$ **if** $r > 1$ $x' \in X$ **for** x'
proof (*rule pointwise-le-antisym*)
have [*simp*]: *length* $x' = r$ $\sigma\ x' \leq n$
using *X-def minimal-elements.cases that*(*2*) *n-ge-X* **by** *auto*
have [*simp*]: *length* (*min-pointwise* $r\ (C\ n\ x')$) $= r$
by (*simp add*: *min-pointwise-def*)
show *min-pointwise* $r\ (C\ n\ x') \trianglelefteq x'$
proof (*clarsimp simp add*: *pointwise-le-iff-nth*)
fix i
assume $i < r$
then obtain y **where** $\sigma\ y = n \wedge x' \trianglelefteq y \wedge y!i \leq x'!i$
by (*metis extend-preserving* ‹$1 < r$› ‹*length* $x' = r$› ‹$x' \in X$› *order.refl n-ge-X*)
then have $\exists\, y{\in}C\ n\ x'.\ y!i \leq x'!i$
using *C-def* **by** *blast*
with ‹$i < r$› **show** *min-pointwise* $r\ (C\ n\ x')!i \leq x'!i$
by (*simp add*: *min-pointwise-def Min-le-iff finC C-empty-iff leD*)
qed
have $x' \trianglelefteq$ *min-pointwise* $r\ (C\ n\ x')$ **if** $\sigma\ x' \leq n$ *length* $x' = r$ **for** x'
by (*smt* (*verit*, *del-insts*) *C-def C-empty-iff* ‹$r \neq 0$› *finC leD mem-Collect-eq min-pointwise-ge-iff pointwise-le-iff that*)
then show $x' \trianglelefteq$ *min-pointwise* $r\ (C\ n\ x')$
using *X-def minimal-elements.cases that* **by** *force*
qed
then have *inj-C*: *inj-on* $(C\ n)\ X$
by (*smt* (*verit*, *best*) *inj-onI mem-Collect-eq* ‹$r{>}1$›)
have *inj-on-imageC*: *inj-on* (*image* $(C\ n)$) (*Pow* $X - \{\{\}\}$)
by (*simp add*: *inj-C inj-on-diff inj-on-image-Pow*)

have *Pow* $(C\ n\ \text{`}\ X) - \{\{\}\} \subseteq$ (*image* $(C\ n)$) ‘ (*Pow* $X - \{\{\}\}$)
by (*metis Pow-empty image-Pow-surj image-diff-subset image-empty*)
then have (*image* $(C\ n)$) ‘ (*Pow* $X - \{\{\}\}$) $=$ *Pow* $(C\ n\ \text{`}\ X) - \{\{\}\}$
by *blast*
then have *SUM1* $=$ *sum* ($\lambda I.\ (-\ 1)$ ^ (*card* $I + 1$) $*$ *int* (*card* ($\bigcap I$))) ((*image* $(C\ n)$) ‘ (*Pow* $X - \{\{\}\}$))
unfolding *SUM1-def* **by** (*auto intro*: *sum.cong*)

also have ... = *sum* $((\lambda I.\ (-\ 1)\ \widehat{}\ (card\ I + 1) * int\ (card\ (\bigcap\ I))) \circ (image\ (C\ n)))\ (Pow\ X - \{\{\}\})$
by (*simp add*: *sum.reindex inj-on-imageC*)
also have ... = *SUM2*
unfolding *SUM2-def* **using** *subset-inj-on* [*OF inj-C*] **by** (*force simp*: *card-image intro*: *sum.cong*)
finally have *SUM1* = *SUM2* **.**

have *length-sum-set r n* = (*length-sum-set r n* $\cap$ {*x*. *useful x*}) $\cup$ (*length-sum-set r n* $\cap$ {*x*. *useless x*})
by *auto*
then have *card* (*length-sum-set r n*) =
card (*length-sum-set r n* $\cap$ {*x*. *useful x*}) +
card (*length-sum-set r n* $\cap$ *Collect useless*)
by (*simp add*: *finite-length-sum-set disjnt-iff flip*: *card-Un-disjnt*)
moreover have *C n ?z0* = *length-sum-set r n*
by (*auto simp*: *C-def length-sum-set-def*)
ultimately have *card* (*C n ?z0*) = *card* (*U n*) + *nat SUM2*
by (*simp add*: *U-def flip*: ‹*SUM1* = *SUM2*› *SUM1-card*)
then have *SUM2-le*: *nat SUM2* $\le$ *card* (*C n ?z0*)
by *arith*
have σ*-max-pointwise-le*: $\bigwedge Y.\ [\![Y \subseteq X;\ Y \neq \{\}]\!] \Longrightarrow \sigma\ (max\text{-}pointwise\ r\ Y) \le n$
by (*meson* ‹*finite X*› *le-trans max-pointwise-mono nlarge pointwise-le-imp-*σ)

have *card-C-max*: *card* (*C n* (*max-pointwise r Y*)) =
($n - \sigma$ (*max-pointwise r Y*) + r − *Suc 0 choose* (r − *Suc 0*))
if $Y \subseteq X$ $Y \neq \{\}$ **for** Y
proof −
have [*simp*]: *length* (*max-pointwise r Y*) = *r*
by (*simp add*: *max-pointwise-def*)
then show *?thesis*
using ‹$r \neq 0$› *that C-def* **by** (*simp add*: *bound-sum-list-card* [*of r*] σ*-max-pointwise-le*)
qed

define *SUM3* **where** $SUM3 \equiv (\sum Y \mid Y \subseteq X \wedge Y \neq \{\}.$
$-\ ((-\ 1)\ \widehat{}\ (card\ Y) * ((n - \sigma\ (max\text{-}pointwise\ r\ Y) + r - 1\ choose\ (r - 1)))))$
have $\bigcap (C\ n\ `\ Y) = C\ n\ (max\text{-}pointwise\ r\ Y)$ **if** $Y \subseteq X$ $Y \neq \{\}$ **for** Y
proof
show $\bigcap\ (C\ n\ `\ Y) \subseteq C\ n\ (max\text{-}pointwise\ r\ Y)$
unfolding *C-def*
proof *clarsimp*
fix *x*
assume $\forall y \in Y.\ y \trianglelefteq x \wedge \sigma\ x = n$
moreover have *finite Y*
using ‹*finite X*› *infinite-super that* **by** *blast*
moreover have $\bigwedge u.\ u \in Y \Longrightarrow length\ u = r$

using *len-X-r that* **by** *blast*
ultimately show *max-pointwise r Y* $\trianglelefteq$ *x* $\wedge$ σ *x* = *n*
by (*smt* (*verit*, *del-insts*) *all-not-in-conv max-pointwise-le-iff pointwise-le-iff-nth that*(*2*))
qed
next
show *C n* (*max-pointwise r Y*) $\subseteq$ $\bigcap$ (*C n* ' *Y*)
apply (*clarsimp simp*: *C-def*)
by (*metis* ‹*finite X*› *finite-subset len-X-r max-pointwise-ge pointwise-le-trans subsetD that*(*1*))
qed
then have *SUM2* = *SUM3*
by (*simp add*: *SUM2-def SUM3-def card-C-max*)
have *U n* = *C n ?z0* − (*length-sum-set r n* $\cap$ {*x*. *useless x*})
by (*auto simp*: *U-def C-def length-sum-set-def*)
then have *card* (*U n*) = *card* (*C n ?z0*) − *card*(*length-sum-set r n* $\cap$ {*x*. *useless x*})
using *finite-length-sum-set*
by (*simp add*: *C-def Collect-mono-iff inf.coboundedI1 length-sum-set-def flip*: *card-Diff-subset*)
then have *card-U-eq-diff*: *card* (*U n*) = *card* (*C n ?z0*) − *nat SUM1*
using *SUM1-card* **by** *presburger*
have *SUM3* $\geq$ *0*
using ‹*0* $\leq$ *SUM1*› ‹*SUM1* = *SUM2*› ‹*SUM2* = *SUM3*› **by** *blast*
have $**$: $\bigwedge$*Y*. ⟦*Y* $\subseteq$ *X*; *Y* $\neq$ {}⟧ $\Longrightarrow$ *Suc* (σ (*max-pointwise r Y*)) $\leq$ *n* + *r*
by (*metis* ‹*1* < *r*› σ*-max-pointwise-le add.commute add-le-mono less-or-eq-imp-le plus-1-eq-Suc*)
have *real* (*f n*) = *card* (*U n*)
unfolding *f-def r-def U-def length-sum-set-def*
using *card-sumset-iterated-length-sum-set-useful length-sum-set-def* **by** *presburger*
also have ... = *card* (*C n ?z0*) − *nat SUM3*
using *card-U-eq-diff* ‹*SUM1* = *SUM2*› ‹*SUM2* = *SUM3*› **by** *presburger*
also have ... = *real* (*card* (*C n* (*replicate r 0*))) − *real* (*nat SUM3*)
using *SUM2-le* ‹*SUM2* = *SUM3*› *of-nat-diff* **by** *blast*
also have ... = *p* (*real n*)
using ‹*1* < *r*› ‹*n*>*0*›
apply (*simp add*: *p-def p0 q* ‹*SUM3* $\geq$ *0*›)
apply (*simp add*: *SUM3-def binomial-gbinomial of-nat-diff* σ*-max-pointwise-le algebra-simps* $**$)
done
finally show *real* (*f n*) = *p* (*real n*) **.**
qed
qed

end

end

References

[1] W. T. Gowers. Introduction to additive combinatorics. Lecture notes, University of Cambridge, 2022.

[2] A. G. Khovanskii. Newton polyhedron, Hilbert polynomial, and sums of finite sets. *Functional Analysis and Its Applications*, 26(4):276–281, 1992.

[3] A. G. Khovanskii. Sums of finite sets, orbits of commutative semigroups, and Hilbert functions. *Functional Analysis and Its Applications*, 29(2):102–112, 1995.

[4] M. B. Nathanson and I. Z. Ruzsa. Polynomial growth of sumsets in abelian semigroups. *Journal de Théorie des Nombres de Bordeaux*, 14(2):553–560, 2002.

[5] I. Z. Ruzsa. Sumsets and structure. Lecture notes, Institute of Mathematics, Budapest.