A verified LLL algorithm*

Jose Divasón  Sebastiaan Joosten  René Thiemann
Akihisa Yamada

February 3, 2018

Abstract

The Lenstra–Lenstra–Lovász basis reduction algorithm, also known as LLL algorithm, is an algorithm to find a basis with short, nearly orthogonal vectors of an integer lattice. Thereby, it can also be seen as an approximation to solve the shortest vector problem (SVP), which is an NP-hard problem, where the approximation quality solely depends on the dimension of the lattice, but not the lattice itself. The algorithm also possesses many applications in diverse fields of computer science, from cryptanalysis to number theory, but it is specially well-known since it was used to implement the first polynomial-time algorithm to factor polynomials. In this work we present the first mechanized soundness proof of the LLL algorithm to compute short vectors in lattices. The formalization follows a textbook by von zur Gathen and Gerhard [1].

Contents

1 Introduction 2
2 List representation 3
3 Missing lemmas 5
4 Norms 59
  4.1 $L_\infty$ Norms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
  4.2 Square Norms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
    4.2.1 Square norms for vectors . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
    4.2.2 Square norm for polynomials . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
  4.3 Relating Norms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
5 Lattice 70

*Supported by FWF (Austrian Science Fund) project Y757. Jose Divasón is partially funded by the Spanish project MTM2017-88804-P.
1 Introduction

The LLL basis reduction algorithm by Lenstra, Lenstra and Lovász [2] is a remarkable algorithm with numerous applications in diverse fields. For instance, it can be used for finding the minimal polynomial of an algebraic number given to a good enough approximation, for finding integer relations, for integer programming and even for breaking knapsack based cryptographic protocols. Its most famous application is a polynomial-time algorithm to factor integer polynomials. Moreover, the LLL algorithm is used as part of the best known polynomial factorization algorithm that is used in today’s computer algebra systems.

In this work we implement it in Isabelle/HOL and fully formalize the correctness of the implementation. The algorithm is parametric by some $\alpha > \frac{4}{3}$, and given $fs$ a list of $m$-linearly independent vectors $f_{s_0}, \ldots, f_{s_{m-1}} \in \mathbb{Z}^n$, it computes a short vector whose norm is at most $\alpha^{m-1}$ larger than the norm of any nonzero vector in the lattice generated by the vectors of the list $fs$. The soundness theorem follows.

Theorem 1 (Soundness of LLL algorithm)

lemma short_vector :
  assumes $\alpha \geq 4/3$
  and lin_indpt_list (RAT fs)
  and short_vector $\alpha \cdot fs = v$
  and length $fs = m$
  and $m \neq 0$
  shows $v \in$ lattice_of $fs$ - $\{0_v, n\}$
  and $h \in$ lattice_of $fs$ - $\{0_v, n\} \rightarrow \|v\|^2 \leq \alpha^{m-1} \cdot \|h\|^2$

To this end, we have performed the following tasks:

- We firstly have to improve some AFP entries, as well as generalize several concepts from the standard library.
- We have to develop a library about norms of vectors and their properties.
• We formalize the Gram–Schmidt orthogonalization procedure, which is a crucial sub-routine of the LLL algorithm. Indeed, we already formalized this procedure in Isabelle as a function `gram_schmidt` when proving the existence of Jordan normal forms [3]. Unfortunately, lemma `gram_schmidt` does not suffice for verifying the LLL algorithm and we have had to extend such a formalization.

• We prove the termination of the algorithm and its soundness.

Regarding the complexity of the LLL algorithm, we did not include a formal statement which would have required an instrumentation of the algorithm by some instruction counter. However, from the termination proof of our Isabelle implementation of the LLL algorithm, one can easily infer a polynomial bound on the number of arithmetic operations. To our knowledge, this is the first formalization of the LLL algorithm in any theorem prover.

2 List representation

theory List-Representation
  imports Main
begin

lemma rev-take-Suc: assumes j: j < length xs
  shows rev (take (Suc j) xs) = xs ! j # rev (take j xs)
proof –
  from j have xs: xs = take j xs @ xs ! j # drop (Suc j) xs by (rule id-take-nth-drop)
  show thesis unfolding arg-cong[OF xs, of λ xs. rev (take (Suc j) xs)]
    by (simp add: min-def)
qed

type-synonym 'a list-repr = 'a list × 'a list

definition list-repr :: nat ⇒ 'a list-repr ⇒ 'a list ⇒ bool where
  list-repr i ba xs = (i ≤ length xs ∧ fst ba = rev (take i xs) ∧ snd ba = drop i xs)

definition of-list-repr :: 'a list-repr ⇒ 'a list where
  of-list-repr ba = (rev (fst ba) @ snd ba)

lemma of-list-repr: list-repr i ba xs ⇒ of-list-repr ba = xs
  unfolding of-list-repr-def list-repr-def by auto

definition get-nth-i :: 'a list-repr ⇒ 'a where
  get-nth-i ba = hd (snd ba)

definition get-nth-im1 :: 'a list-repr ⇒ 'a where
  get-nth-im1 ba = hd (fst ba)
lemma get-nth-i: list-repr i ba xs \implies i < \text{length} \; xs \implies \text{get-nth-i} \; ba = xs ! i

unfolding list-repr-def get-nth-i-def
by (auto simp: hd-drop-conv-nth)

lemma get-nth-im1: list-repr i ba xs \implies i \neq 0 \implies \text{get-nth-im1} \; ba = xs ! (i - 1)

unfolding list-repr-def get-nth-im1-def
by (cases i, auto simp: rev-take-Suc)

definition update-i :: 'a list-repr \Rightarrow 'a \Rightarrow 'a list-repr where
update-i ba x = (fst ba, x # tl (snd ba))

lemma Cons-tl-drop-update: i < \text{length} \; xs \implies x # tl (\text{drop} \; i \; xs) = \text{drop} \; i \; (x[\text{xs}[i := x]])

proof (induct i arbitrary: xs)
  case (0 xs)
  thus \?case by (cases xs, auto)
next
  case (Suc i xs)
  thus \?case by (cases xs, auto)
qed

lemma update-i: list-repr i ba xs \implies i < \text{length} \; xs \implies \text{list-repr} \; i \; (\text{update-i} \; ba \; x) (xs [i := x])

unfolding update-i-def list-repr-def
by (auto simp: Cons-tl-drop-update)

definition update-im1 :: 'a list-repr \Rightarrow 'a \Rightarrow 'a list-repr where
update-im1 ba x = (x # tl (fst ba), snd ba)

lemma update-im1: list-repr i ba xs \implies i \neq 0 \implies \text{list-repr} \; i \; (\text{update-im1} \; ba \; x) (xs [i - 1 := x])

unfolding update-im1-def list-repr-def
by (cases i, auto simp: rev-take-Suc)

lemma tl-drop-Suc: tl \; (\text{drop} \; i \; xs) = \text{drop} \; (\text{Suc} \; i) \; xs

proof (induct i arbitrary: xs)
  case (0 xs) thus \?case by (cases xs, auto)
next
  case (Suc i xs) thus \?case by (cases xs, auto)
qed

definition inc-i :: 'a list-repr \Rightarrow 'a list-repr where
inc-i ba = (case ba of (b, a) \Rightarrow (hd a # b, tl a))

lemma inc-i: list-repr i ba xs \implies i < \text{length} \; xs \implies \text{list-repr} \; (\text{Suc} \; i) \; (\text{inc-i} \; ba) \; xs

unfolding list-repr-def inc-i-def by (cases ba, auto simp: rev-take-Suc hd-drop-cone-nth tl-drop-Suc)
**Definition**

\[ \text{dec-i} :: 'a \text{ list-repr} \Rightarrow 'a \text{ list-repr where} \]
\[ \text{dec-i \, ba} = (\text{case \, ba \, of} \, (b, a) \Rightarrow (\text{tl \, b, \, hd \, b \, # \, a})) \]

**Lemma**

\[ \text{dec-i}: \text{list-repr \, i \, ba \, xs} \implies i \neq 0 \implies \text{list-repr \, (i - 1) \, (dec-i \, ba) \, xs} \]

**Unfolding**

\text{list-repr-def \, dec-i-def}

**By** (cases ba; cases i, auto simp: rev-take-Suc hd-drop-cone-nth Cons-nth-drop-Suc)

**Lemma**

\[ \text{dec-i-Suc}: \text{list-repr \, (Suc \, i) \, ba \, xs} \implies \text{list-repr \, i \, (dec-i \, ba) \, xs} \]

**Using** \[ dec-i[Suc \, i \, ba \, xs] \text{ by auto} \]

**end**

### 3 Missing lemmas

This theory contains many results that are important but not specific for our development. They could be moved to the standard library and some other AFP entries.

**theory** Missing-Lemmas

**imports**

- Berlekamp-Zassenhaus.Sublist-Iteration
- Berlekamp-Zassenhaus.Square-Free-Int-To-Square-Free-GFp
- Algebraic-Numbers.Resultant
- Jordan-Normal-Form.Conjugate
- Jordan-Normal-Form.Missing-VectorSpace
- VS-Connect
- Berlekamp-Zassenhaus.Finite-Field-Factorization-Record-Based
- Berlekamp-Zassenhaus.Berlekamp-Hensel

**begin**

**hide-const(open)** module.smalt up-ring.monom up-ring.coeff

**locale** comp-fun-commute-on =

**fixes** \( f :: 'a \Rightarrow 'a \Rightarrow 'a \text{ and } A::'a \text{ set} \)

**assumes** comp-fun-commute-restrict: \( \forall \, y \in A. \forall \, x \in A. \forall \, z \in A. \, f \, y \, (f \, x \, z) = f \, x \, (f \, y \, z) \)

**and** \( f: A \rightarrow A \rightarrow A \)

**begin**

**lemma** comp-fun-commute-on-UNIV:

**assumes** \( A = (UNIV :: 'a \text{ set}) \)

**shows** comp-fun-commute \( f \)

**unfolding** comp-fun-commute-def

**using** \( \text{assms \, comp-fun-commute-restrict \, f \, by \, auto} \)
lemma fun-left-comm:
  assumes $y \in A$ and $x \in A$ and $z \in A$
  shows $f y (f x z) = f x (f y z)$
  using comp-fun-commute-restrict assms by auto

lemma commute-left-comp:
  assumes $y \in A$ and $x \in A$ and $z \in A$ and $g : A \to A$
  shows $f y (f x (g z)) = f x (f y (g z))$
  using assms by (auto simp add: Pi-def o-assoc comp-fun-commute-restrict)

lemma fold-graph-finite:
  assumes $\text{fold-graph } f z B y$
  shows finite $B$
  using assms by (induct simp-all)

lemma fold-graph-closed:
  assumes $\text{fold-graph } f z B y$
  and $B \subseteq A$ and $z \in A$
  shows $y \in A$
  using assms proof (induct set: fold-graph)
  case emptyI then show ?case by auto
  next
  case (insertI $x$ $B$ $y$)
  then show ?case using insertI $f$ by auto
  qed

lemma fold-graph-insertE-aux:
  $\text{fold-graph } f z B y \Rightarrow a \in B \Rightarrow z \in A$
  $\Rightarrow B \subseteq A$
  $\Rightarrow \exists y'. \ y = f a y' \land \text{fold-graph } f z (B - \{a\}) y' \land y' \in A$
  proof (induct set: fold-graph)
  case emptyI then show ?case by auto
  next
  case (insertI $x$ $B$ $y$)
  then show ?case using insertI $f$ by auto
  qed

proof (exI[of - $y$])
  have $B$: $\text{insert } x B - \{a\} = B$ using True insertI by auto
  have $f x y = f a y$ by (simp add: True)
  moreover have $\text{fold-graph } f z (\text{insert } x B - \{a\}) y$ by (simp add: B insertI)
  moreover have $y \in A$ using insertI fold-graph-closed[of $z$ $B$] by auto
  ultimately show $f x y = f a y \land \text{fold-graph } f z (\text{insert } x B - \{a\}) y \land y \in A$
  by simp
  qed
next
case False
then obtain \( y' \) where \( y = f \ a \ y' \) and \( y': \text{fold-graph} \ f \ z \ (B \ - \ \{a\}) \ y' \) and \( y' \in A \)
  using insertI \( f \) by auto
have \( f \ x \ y = f \ a \ (f \ x \ y') \)
unfolding \( y \)
proof (rule fun-left-comm)
show \( x \in A \) using insertI \( f \) by auto
show \( a \in A \) using insertI \( f \) by auto
show \( y' \in A \) using \( y' \in A \) by auto
qed
moreover have \( \text{fold-graph} \ f \ z \ (\text{insert} \ x \ B \ - \ \{a\}) \ (f \ x \ y') \)
  using \( y' \) and \( \langle x \neq a \ \& \ \& \ x \notin B \rangle \)
by (simp add: \( \text{insert-Diff-if} \ \text{fold-graph-insertI} \))
moreover have \( (f \ x \ y') \in A \) using insertI \( f \ y' \in A \) by auto
ultimately show \( ?\text{thesis} \) using \( y' \in A \) by auto
qed
lemma fold-graph-insertE:
assumes \( \text{fold-graph} \ f \ z \ (\text{insert} \ x \ B) \ v \) and \( x \notin B \) and \( \text{insert} \ x \ B \subseteq A \) and \( z \in A \)
obtains \( y \) where \( v = f \ x \ y \) and \( \text{fold-graph} \ f \ z \ B \ y \)
using assms by (auto dest: fold-graph-insertE-aux [OF - insertI1])

lemma fold-graph-determ: \( \text{fold-graph} \ f \ z \ B \ x \Longrightarrow \text{fold-graph} \ f \ z \ B \ y \Longrightarrow B \subseteq A \)
  \( z \in A \Longrightarrow y = x \)
proof (induct arbitrary: \( y \) set: \( \text{fold-graph} \))
case emptyI
then show \( ?\text{case} \)
  by (meson empty-fold-graphE)
next
case (insertI \( x \) \( B \) \( v \))
from \( \text{fold-graph} \ f \ z \ (\text{insert} \ x \ B) \ v \) and \( x \notin B \) and \( \text{insert} \ x \ B \subseteq A \) and \( z \in A \)
obtain \( y' \) where \( v = f \ x \ y' \) and \( \text{fold-graph} \ f \ z \ B \ y' \)
by (rule fold-graph-insertE)
from \( \text{fold-graph} \ f \ z \ B \ y' \) and \( \text{insert} \ x \ B \subseteq A \) have \( y' = y \) using insertI \( f \) by auto
with \( v = f \ x \ y' \) show \( v = f \ x \ y \)
  by simp
qed
lemma fold-equality: \( \text{fold-graph} \ f \ z \ B \ y \Longrightarrow B \subseteq A \Longrightarrow z \in A \Longrightarrow \text{Finite-Set.fold} \ f \ z \ B = y \)
by (cases finite \( B \))
(auto simp add: Finite-Set.fold-def intro: fold-graph-determ dest: fold-graph-finite)
lemma fold-graph-fold:

7
assumes \( f: \text{finite } B \text{ and } BA: B \subseteq A \text{ and } z: z \in A \)
shows \( \text{fold-graph } f z B \) (Finite-Set.fold \( f z B \))

proof
- have \( \exists x. \text{fold-graph } f z B x \)
  by (rule finite-imp-fold-graph\([OF f]\))
moreover note fold-graph-determ
ultimately have \( \exists! x. \text{fold-graph } f z B x \) using \( f BA z \) by auto
then have fold-graph \( f z B \) (The (fold-graph \( f z B \)))
  by (rule theI')
with assms show \(?thesis\)
  by (simp add: Finite-Set.fold-def)
qed

lemma fold-insert \([simp]\):
assumes \( f: \text{comp-fun-commute-on } f A \) and \( g: \text{comp-fun-commute-on } g A \)
and \( \text{finite } S \)
and cong: \( \forall x. x \in S \implies f x = g x \)
and \( s = t \) and \( S = T \)
and \( S: S \subseteq A \) and \( s: s \in A \)
shows \( \text{Finite-Set.fold } f s S = \text{Finite-Set.fold } g t T \)

proof
- have \( \text{Finite-Set.fold } f s S = \text{Finite-Set.fold } g s S \)
  using \( \text{finite } S; \text{ cong } S A s \)
proof (induct \( S \))
  case empty
  then show \(?case\) by simp
next
  case (insert \( x \ F \))
  interpret \( f: \text{comp-fun-commute-on } f A \) by (fact \( f \))
  interpret \( g: \text{comp-fun-commute-on } g A \) by (fact \( g \))
  show \(?case\) using insert by auto
qed
with assms show \(?thesis\) by simp
qed
context comp-fun-commute-on
begin

lemma comp-fun-Pi: $(\lambda x. f x \sim g x) \in A \rightarrow A \rightarrow A$
proof -
  have $(f x \sim g x)y \in A$ if $y \in A$ and $x: x \in A$ for $x y$
  using $x y$
proof (induct $g x$ arbitrary: $g$)
  case 0
  then show $?case$ by auto
next
  case $(Suc n g)$
  define $h$ where $h z = g z - 1$ for $z$
  have $hyp: (f x \sim h x)y \in A$
    using $h$-def Suc.prems Suc.hyps diff-Suc-1 by metis
  have $g x = Suc (h x)$ unfolding $h$-def
    using Suc.hyps(2) by auto
  then show $?case$ using $f x hyp$ unfolding Pi-def by auto
qed
thus $?thesis$ by (auto simp add: Pi-def)
qed

lemma comp-fun-commute-funpow: comp-fun-commute-on $(\lambda x. f x \sim g x) A$
proof -
  have $f: (f y \sim g y) ((f x \sim g x) z) = (f x \sim g x)((f y \sim g y) z)$
    if $x: x \in A$ and $y: y \in A$ and $z: z \in A$ for $x y z$
proof (cases $x = y$)
  case False
  show $?thesis$
proof (induct $g x$ arbitrary: $g$)
    case $(Suc n g)$
    have $hyp1: (f y \sim g y) (f x k) = f x ((f y \sim g y) k)$ if $k: k \in A$ for $k$
    proof (induct $g y$ arbitrary: $g$)
      case 0
      then show $?case$ by simp
    next
      case $(Suc n g)$
      define $h$ where $h z = g z - 1$ for $z$
      with $Suc$ have $n = h y$
        by simp
      with $Suc$ have $hyp: (f y \sim h y) (f x k) = f x ((f y \sim h y) k)$
        by auto
      from $Suc$ $h$-def have $g: g y = Suc (h y)$
        by simp
      have $((f y \sim h y) k) \in A$ using $y k$ comp-fun-Pi[of $h$] unfolding Pi-def
        by auto
      then show $?case$
by (simp add: comp-assoc g hyp) (auto simp add: o-assoc comp-fun-commute-restrict)

qed

define h where h a = (if a = x then g x - 1 else g a) for a

with Suc have n = h x
  by simp

with Suc have (f y ^^ h y) ((f x ^^ h x) z) = (f x ^^ h x) ((f y ^^ h y) z)
  by auto

with False have Suc2: (f x ^^ h x) ((f y ^^ g y) z) = (f y ^^ g y) ((f x ^^ h x) z)

  using h-def by auto

from Suc h-def have g: g x = Suc (h x)

  hence *: (f y ^^ g y) (f x ((f y ^^ h x) z)) = f x ((f y ^^ g y) ((f x ^^ h x) z))

  using hyp1 by auto

  thus ?case using g Suc2 by auto

  qed simp

  qed simp

  thus ?thesis by (auto simp add: comp-fun-commute-on-def comp-fun-Pi o-def)

  qed


lemma fold-mset-add-mset:
  assumes MA: set-mset M ⊆ A and s: s ∈ A and x: x ∈ A
  shows fold-mset f s (add-mset x M) = f x (fold-mset f s M)

proof -
  interpret mset: comp-fun-commute-on λy. f y ^^ count (add-mset x M) y A
    by (fact comp-fun-commute-funpow)

  interpret mset-union: comp-fun-commute-on λy. f y ^^ count (add-mset x M) y A
    by (fact comp-fun-commute-funpow)

  show ?thesis
    proof (cases x ∈ set-mset M)
      case False
      then have *: count (add-mset x M) x = 1
        by (simp add: not-in-iff)

      have Finite-Set.fold (λy. f y ^^ count (add-mset x M) y) s (set-mset M) =
        Finite-Set.fold (λy. f y ^^ count M y) s (set-mset M)
        by (rule fold-cong[of - A], auto simp add: assms False comp-fun-commute-funpow)

      with False * s MA x show ?thesis
        by (simp add: fold-mset-def del: count-add-mset)

    next
      case True
      let f' = (λxa. f xa ^^ count (add-mset x M) xa)
      let f'2 = (λx. f x ^^ count M x)
      define N where N = set-mset M - {x}
have \( F : \text{Finite-Set.fold} \ ?f \ s \ (\text{insert} \ x \ N) = ?f \ x \ (\text{Finite-Set.fold} \ ?f \ s \ N) \)
by (rule mset-union.fold-insert, auto simp add: assms N-def)

have \( F2 : \text{Finite-Set.fold} \ ?f2 \ s \ (\text{insert} \ x \ N) = ?f2 \ x \ (\text{Finite-Set.fold} \ ?f2 \ s \ N) \)
by (rule mset.fold-insert, auto simp add: assms N-def)

from \( N\text{-def True} \) have \( \ast : \text{set-mset} \ M = \text{insert} \ x \ N \ x \notin N \text{ finite} \ N \text{ by auto} \)
then have \( \text{Finite-Set.fold} \ (\lambda y. f \ y \ "\count (\text{add-mset} \ x \ M) y) \ s \ N = \text{Finite-Set.fold} \ (\lambda y. f \ y \ "\count M y) \ s \ N \)
using \( MA \ N\text{-def} \ s \)
by (auto intro!: fold-cong comp-fun-commute-funpow)

with \( \ast \) show \( \text{thesis by simp add: fold-mset-def del: count-add-mset, unfold F F2, auto}\)
qed
qed
end

context abelian-monoid begin

definition sumlist
where sumlist \( xs \equiv \text{foldr} \ (op \oplus) \ xs \ 0 \)

lemma [simp]:
shows sumlist-Cons: sumlist \((x\#xs)\) = \(x \oplus \text{sumlist} \ xs \)
and sumlist-Nil: sumlist \([]\) = 0
by (simp-all add: sumlist-def)

lemma sumlist-carrier [simp]:
assumes set \( xs \subseteq \text{carrier} \ G \) shows sumlist \( xs \in \text{carrier} \ G \)
using assms by (induct xs, auto)

lemma sumlist-neutral:
assumes set \( xs \subseteq \{0\} \) shows sumlist \( xs \) = 0
proof (insert assms, induct xs)
  case (Cons \( x \ \) \( xs \))
  then have \( x = 0 \) and set \( xs \subseteq \{0\} \) by auto
  with Cons.hyps show \( \)case by auto
qed simp

lemma sumlist-append:
assumes set \( xs \subseteq \text{carrier} \ G \) and set \( ys \subseteq \text{carrier} \ G \)
shows sumlist \((xs \circ ys)\) = sumlist \( xs \oplus \text{sumlist} \ ys \)
proof (insert assms, induct xs arbitrary: \( ys \))
  case (Cons \( x \ \) \( xs \))
  have sumlist \((xs \circ ys)\) = sumlist \( xs \oplus \text{sumlist} \ ys \)
  using Cons.prems by (auto intro: Cons.hyps)
with Cons.prems show ?case by (auto intro!: a-associ[symmetric])
qed auto

lemma sumlist-snoc:
  assumes set xs ⊆ carrier G and x ∈ carrier G
  shows sumlist (xs @ [x]) = sumlist xs ⊕ x
  by (subst sumlist-append, insert assms, auto)

lemma sumlist-as-finsum:
  assumes set xs ⊆ carrier G and distinct xs shows sumlist xs = (∑ x∈set xs. x)
  using assms by (induct xs, auto intro:finsum-insert[symmetric])

lemma sumlist-map-as-finsum:
  assumes f : set xs → carrier G and distinct xs
  shows sumlist (map f xs) = (∑ x∈set xs. f x)
  using assms by (induct xs, auto)

definition summset where summset M ≡ fold-mset (op ⊕) 0 M

lemma summset-empty [simp]: summset {#} = 0 by (simp add: summset-def)

lemma fold-mset-add-carrier: a ∈ carrier G ⇒ set-mset M ⊆ carrier G ⇒
  fold-mset op ⊕ a M ∈ carrier G
proof (induct M arbitrary: a)
  case (add x M)
  thus ?case by
    (subst comp-fun-commute-on.fold-mset-add-mset[of - carrier G], unfold-locales, auto simp: a-lcomm)
qed simp

lemma summset-carrier[intro]: set-mset M ⊆ carrier G ⇒ summset M ∈ carrier G
  unfolding summset-def by (rule fold-mset-add-carrier, auto)

lemma summset-add-mset[simp]:
  assumes a: a ∈ carrier G and MG: set-mset M ⊆ carrier G
  shows summset (add-mset a M) = a ⊕ summset M
  using assms
  by (auto simp add: summset-def)
    (rule comp-fun-commute-on.fold-mset-add-mset, unfold-locales, auto simp add: a-lcomm)

lemma sumlist-as-summset:
  assumes set xs ⊆ carrier G shows sumlist xs = summset (mset xs)
  by (insert assms, induct xs, auto)

lemma sumlist-rev:
  assumes set xs ⊆ carrier G

12
shows \( \text{sumlist } (\text{rev } xs) = \text{sumlist } xs \)
using \( \text{assms} \) by \( \text{(simp add: sumlist-as-sumset)} \)

**lemma** sumlist-as-fold:
assumes \( \text{set } xs \subseteq \text{carrier } G \)
shows \( \text{sumlist } xs = \text{fold } (\text{op } \oplus) \) \( xs 0 \)
by \( \text{(fold sumlist-rev)[OF assms], simp add: sumlist-def foldr-cone-fold)} \)
end

**lemma** (in zero-less-one) zero-le-one [simp]: \( 0 \leq 1 \) by \( \text{(rule less-imp-le, simp)} \)
**subclass** (in zero-less-one) zero-neq-one by \( \text{(unfold-locale, simp add: less-imp-neq)} \)

**class** ordered-semiring-1 = Rings.ordered-semiring-0 + monoid-mult + zero-less-one
begin

**subclass** semiring-1 ..

**lemma** of-nat-ge-zero [intro!]: \( \text{of-nat } n \geq 0 \)
using \( \text{add-right-mono}[of - - 1] \) by \( \text{(induct } n, \text{ auto)} \)

**lemma** zero-le-power [simp]: \( 0 \leq a \Rightarrow 0 \leq a ^ n \)
by \( \text{(induct } n \text{ simp-all)} \)

**lemma** power-mono: \( a \leq b \Rightarrow 0 \leq a \Rightarrow a ^ n \leq b ^ n \)
by \( \text{(induct } n \text{ (auto intro: mult-mono order-trans [of } 0 \text{ a b])}} \)

**lemma** one-le-power [simp]: \( 1 \leq a \Rightarrow 1 \leq a ^ n \)
using \( \text{power-mono [of } 1 \text{ a n]} \) by \( \text{simp} \)

**lemma** power-le-one: \( 0 \leq a \Rightarrow a \leq 1 \Rightarrow a ^ n \leq 1 \)
using \( \text{power-mono [of } a \text{ 1 n]} \) by \( \text{simp} \)

**lemma** power-gt1-lemma:
assumes \( \text{gt1: } 1 < a \)
shows \( 1 < a * a ^ n \)
proof –
from \( \text{gt1} \) have \( 0 \leq a \)
  by \( \text{(fact order-trans [OF zero-le-one less-imp-le])} \)
from \( \text{gt1} \) have \( 1 * 1 < a * 1 \) by \( \text{simp} \)
also from \( \text{gt1} \) have \( \ldots \leq a * a ^ n \)
  by \( \text{(simp only!: mult-mono [0 \leq a] one-le-power order-less-imp-le zero-le-one order-refl)} \)
finally show ?thesis by \( \text{simp} \)
qed
lemma power-gt1: \( 1 < a \Rightarrow 1 < a ^{\mathrm{Suc} \, n} \)
by (simp add: power-gt1-lemma)

lemma one-less-power [simp]: \( 1 < a \Rightarrow 0 < n \Rightarrow 1 < a ^{n} \)
by (cases n) (simp-all add: power-gt1-lemma)

Proof resembles that of power-strict-decreasing.

lemma power-decreasing: \( n \leq N \Rightarrow 0 \leq a \Rightarrow a \leq 1 \Rightarrow a ^{N} \leq a ^{n} \)
proof (induct N)
  case 0
  then show ?case by simp

next
  case (Suc N)
  then show ?case
    apply (auto simp add: le-Suc-eq)
    apply (subgoal_tac a * a ^{\mathrm{Suc} \, N} \leq 1 * a ^{\mathrm{n}})
    apply simp
    apply (rule mult-mono)
    apply auto
    done

qed

Proof again resembles that of power-strict-decreasing.

lemma power-increasing: \( n \leq N \Rightarrow 1 \leq a \Rightarrow a ^{n} \leq a ^{\mathrm{Suc} \, N} \)
proof (induct N)
  case 0
  then show ?case by simp

next
  case (Suc N)
  then show ?case
    apply (auto simp add: le-Suc-eq)
    apply (subgoal-tac a * a ^{\mathrm{n}} \leq a * a ^{\mathrm{N}})
    apply simp
    apply (rule mult-mono)
    apply (auto simp add: order-trans [OF zero-le-one])
    done

qed

lemma power-Suc-le-self: \( 0 \leq a \Rightarrow a \leq 1 \Rightarrow a ^{\mathrm{Suc} \, n} \leq a \)
using power-decreasing [of 1 Suc n a] by simp

end

lemma prod-list-nonneg: \( \:\bigwedge \, x. \, (x :: 'a :: ordered-semiring-1) \in \text{set} \, xs \Rightarrow x \geq 0 \)
\Rightarrow \text{prod-list} \, xs \geq 0
by (induct xs, auto)

subclass (in ordered-idom) ordered-semiring-1 by unfold-locales auto
lemma \textit{log-prod}: assumes $0 \lessdot a \lessdot 1$ \wedge x, x \in X \implies 0 \lessdot f x$

shows $\log a \left( \prod f X \right) = \sum \left( \log a \circ f \right) X$

using \text{assms(3)}

proof (induct X rule: infinite-finite-induct)

case (insert x F)

have $\log a \left( \prod f \left( \text{insert x F} \right) \right) = \log a \left( f x \ast \prod f F \right)$ using insert by simp

also have $\ldots = \log a \left( f x \right) \ast \log a \left( \prod f F \right)$

by (rule \text{log-mult(OF assms(1-2) insert(4) prod-pos), insert insert, auto})

finally show \textit{?case} using insert by auto

qed auto

subclass (in \text{ordered-idom}) \text{zero-less-one} by (\text{unfold-locales, auto})

hide-fact Missing-Ring.zero-less-one

\text{instance} real \\text{::} \text{ordered-semiring-strict} by (\text{intro-classes, auto})

\text{instance} real \\text{::} \text{linordered-idom}.

lemma \textit{less-1-mult}':

fixes a::'a::linordered-semidom

shows $1 \lessdot a \implies 1 \lessdot b \implies 1 \lessdot a \ast b$

by (metis \text{le-less less-1-mult \text{mult.right-neutral}})

lemma \textit{upt-minus-eq-append}: $i \leq j \implies i \lessdot j - k \implies [i..<j] = [i..<j-k] @ [j-k..<j]$

proof (induct k)

case (Suc k)

have hyp: $[i..<j] = [i..<j-k] @ [j-k..<j]$ using Suc.hyps Suc.prems by auto

then show \textit{?case}

by (metis Suc.prems(2) append.simps(1) \text{diff-Suc-less nat-less-le neg0-conv upt-append upt-rec zero-diff})

qed auto

lemma \textit{list-trisect}: $x \lessdot \text{length lst} \implies [\emptyset..<\text{length lst}] = [\emptyset..<x] \oplus [\text{Suc x}..<\text{length lst}]$

by (induct lst, force, rename-tac a lst, case-tac x = \text{length lst}, auto)

lemma \textit{nth-map-out-of-bound}: $i \geq \text{length xs} \implies \text{map f xs} ! i = [] ! (i - \text{length xs})$

by (induct xs arbitrary:i, auto)

lemma \textit{filter-mset-inequality}: $\text{filter-mset f xs} \neq \text{xs} \implies \exists x \in\# \text{ xs}. \neg f x$

by (induct xs, auto)

15
lemma id-imp-bij-betw:
assumes \( f : A \to A \)
and \( ff : \forall a. a \in A \implies f (f a) = a \)
shows bij-betw \( f \) \( A \)
by (intro bij-betwI[OF \( f \) \( f \)], simp-all add: \( ff \))

lemma if-distrib-ap:
(if \( x \to y \) else \( z \)) \( u \) = (if \( x \to y \) \( u \) else \( z \) \( u \)) by auto

lemma range-subsetI:
assumes \( \forall x. f x = g (h x) \)
shows range \( f \) \( \subseteq \) range \( g \)
using assms by auto

lemma Gcd-uminus:
fixes \( A : \) int set
assumes finite \( A \)
shows \( \text{Gcd} A = \text{Gcd} (\text{uminus} \cdot A) \)
using assms by (induct \( A \), auto)

lemma aux-abs-int: fixes \( c : \) int
assumes \( c \neq 0 \)
shows \( |x| \leq |x \cdot c| \)
proof –
  have abs \( x \) = abs \( x \cdot 1 \) by simp
  also have \( \ldots \leq abs x \cdot abs c \)
    by (rule mult-left-mono, insert assms, auto)
  finally show \(?thesis\) unfolding abs-mult by auto
qed

lemma sqrt-int-ceiling-bound: \( 0 \leq x \implies x \leq (\text{sqrt-int-ceiling} x)^2 \)
unfolding sqrt-int-ceiling using le-of-int-ceiling of-int-le-iff sqrt-le-D by fastforce

lemma mod-0-abs-less-imp-0:
fixes \( a : \) int
assumes \( a1 : [a = 0] (\mod m) \)
and \( a2 : \text{abs}(a) < m \)
shows \( a = 0 \)
proof –
  have \( m \geq 0 \) using assms by auto
  thus \(?thesis\)
    using assms unfolding cong-int-def
    using int-mod-pos-eq large-mod-0 zless-imp-add1-zle
    by (metis abs-of-nonneg le-less not-less zabs-less-one-iff zmod-trival-iff)
qed

lemma sum-list-zero:
assumes set \( xs \subseteq \{0\} \)
shows \( \text{sum-list} \) \( xs = 0 \)
using assms by (induct xs, auto)

lemma max-idem [simp]; shows max a a = a by (simp add: max-def)

lemma hom-max:
  assumes a ≤ b ←→ f a ≤ f b
  shows f (max a b) = max (f a) (f b) using assms by (auto simp: max-def)

lemma le-max-self:
  fixes a b :: preorder
  assumes a ≤ b ∨ b ≤ a shows a ≤ max a b and b ≤ max a b
  using assms by (auto simp: max-def)

lemma le-max:
  fixes a b :: preorder
  assumes c ≤ a ∨ c ≤ b and a ≤ b ∨ b ≤ a shows c ≤ max a b
  using assms(1) le-max-self[OF assms(2)] by (auto dest: order-trans)

fun max-list where
  max-list [] = (THE x. False)
| max-list [x] = x
| max-list (x # y # xs) = max x (max-list (y # xs))

declare max-list.simps(1) [simp del]
declare max-list.simps(2-3)[code]

lemma max-list-Cons; max-list (x#xs) = (if xs = [] then x else max x (max-list xs))
  by (cases xs, auto)

lemma max-list-mem: xs ≠ [] ⇒ max-list xs ∈ set xs
  by (induct xs, auto simp: max-list-Cons max-def)

lemma mem-set-imp-le-max-list:
  fixes xs :: 'a :: preorder list
  assumes ∀a b. a ∈ set xs ⇒ b ∈ set xs ⇒ a ≤ b ∨ b ≤ a
  and a ∈ set xs
  shows a ≤ max-list xs
proof (insert assms, induct xs arbitrary:a)
case Nil
  with assms show ?case by auto
next
case (Cons x xs)
show ?case
proof (cases xs = [])
case False
  have x ≤ max-list xs ∨ max-list xs ≤ x
  apply (rule Cons(2)) using max-list-mem[of xs] False by auto


note \( l = \text{le-max-self}[\text{OF this}] \)
from Cons have \( a = x \lor a \in \text{set } xs \) by auto
then show \( \text{thesis} \)
proof (elim disjE)
  assume \( a: a = x \)
  show \( \text{thesis} \) by (unfold a max-list-Cons, auto simp: False intro!: 1)
next
  assume \( a \in \text{set } xs \)
  then have \( a \leq \text{max-list } xs \) by (intro Cons, auto)
with \( 1 \) have \( a \leq x \) (max-list xs) by (auto dest: order-trans)
then have \( a \leq \text{max } x \) (max-list xs)
by (auto simp: False intro!: 1)
qed
qed (insert Cons, auto)

lemma le-max-list:
fixes \( xs :: 'a :: \text{preorder list} \)
assumes \( \text{ord}: \forall a b. a \in \text{set } xs \rightarrow b \in \text{set } xs \rightarrow a \leq b \lor b \leq a \)
and \( \text{ab}: a \leq b \)
and \( \text{b}: b \in \text{set } xs \)
shows \( a \leq \text{max-list } xs \)
proof
  note \( \text{ab} \)
  also have \( b \leq \text{max-list } xs \)
  by (rule mem-set-imp-le-max-list, fact \( \text{ord} \), fact \( \text{b} \))
finally show \( \text{thesis} \).
qed

lemma max-list-le:
fixes \( xs :: 'a :: \text{preorder list} \)
assumes \( a: \forall x. x \in \text{set } xs \rightarrow x \leq a \)
and \( \text{xs}: \text{xs} \neq [] \)
shows \( \text{max-list } xs \leq a \)
using max-list-mem[\text{OF } \text{xs}] a by auto

lemma max-list-as-Greatest:
assumes \( \forall x y. x \in \text{set } xs \rightarrow y \in \text{set } xs \rightarrow x \leq y \lor y \leq x \)
shows \( \text{max-list } xs = (\text{GREATEST } a. a \in \text{set } xs) \)
proof (cases \( \text{xs} = [] \))
case True
  then show \( \text{thesis} \) by (unfold Greatest-def, auto simp: max-list.simps(1))
next
case False
from \( \text{assms} \) have \( 1: x \in \text{set } xs \rightarrow x \leq \text{max-list } xs \) for \( x \)
  by (auto intro: le-max-list)
have \( 2: \text{max-list } xs \in \text{set } xs \) by (fact max-list-mem[\text{OF } False])
have \( \exists x. x \in \text{set } xs \land (\forall y. y \in \text{set } xs \rightarrow y \leq x) \) (is \( \exists !x. \text{P } x \))
proof (intro exI)
  from \( 1 \) \( 2 \)

18
show ?P (max-list xs) by auto

next
fix x assume 3: ?P x
with 1 have x ≤ max-list xs by auto
moreover from 2 3 have max-list xs ≤ x by auto
ultimately show x = max-list xs by auto
qed

note 3 = theI-unique[OF this,symmetric]
from 1 2 show ?thesis
  by (unfold Greatest-def Cons 3, auto)
qed

lemma hom-max-list-commute:
  assumes xs ≠ []
      and ∀x y. x ∈ set xs ⟹ y ∈ set xs ⟹ h (max x y) = max (h x) (h y)
  shows h (max-list xs) = max-list (map h xs)
  by (insert assms, induct xs, auto simp: max-list-Cons max-list-mem)

primrec rev-upt :: nat ⇒ nat ⇒ nat list (([>..]) where
rev-upt-0: [0>..] = [] |
rev-upt-Suc: [(Suc i)>..] = (if i ≥ j then i # [i>..] else [])

lemma rev-upt-rec: [i>..] = (if i > j then [i>..Suc j] @ [j] else [])
  by (induct i, auto)

definition rev-upt-aux :: nat ⇒ nat ⇒ nat list where
rev-upt-aux i j js = [i>.. j] @ js

lemma upt-aux-rec [code]:
  rev-upt-aux i j js = (if j ≥ i then js else rev-upt-aux i (Suc j) (j#js))
  by (induct j, auto simp add: rev-upt-aux-def rev-upt-rec)

lemma rev-upt-code [code]: [i>..] = rev-upt-aux i j []
  by (simp add: rev-upt-aux-def)

lemma upt-rev-upt:
  rev [j>..i] = [i..<j]
  by (induct j, auto)

lemma rev-upt-rev-upt:
  rev [i..<j] = [j>..i]
  by (induct j, auto)

lemma length-rev-upt [simp]: length [i>..] = i - j
  by (induct i) (auto simp add: Suc-diff-le)

lemma nth-rev-upt [simp]: j + k < i ⟹ [i>..] ! k = i - 1 - k
proof
  assume \( jk-i: j + k < i \)
  have \([i>j] = \text{rev} \ [j<i]\) using rev-upt-upt by simp
  also have \( \ldots! k = [j<i]!(\text{length} \ [j<i] - 1 - k) \)
  by (rule nth-rev, insert jk-i, auto)
  also have \( \ldots = [j<i]!(i - j - 1 - k) \) by auto
  also have \( \ldots = j + (i - j - 1 - k) \) by (rule nth-upt, insert jk-i, auto)
  finally show \( \text{thesis} \) using jk-i by auto
qed

lemma nth-map-rev-upt:
  assumes \( i: i < m-n \)
  shows \( (\text{map} \ f \ [m>..n])!i = f\ (m - 1 - i) \)
proof
  have \( (\text{map} \ f \ [m>..n])!i = f\ ([m>..n]!i) \) by (rule nth-map, auto simp add: i)
  also have \( \ldots = f\ (m - 1 - i) \)
  proof (rule arg-cong[of _ _ f], rule nth-rev-upt)
  show \( n + i < m \) using i by linarith
  qed
  finally show \( \text{thesis} \).
qed

lemma coeff-mult-monom:
  coeff \( (p * \text{monom} \ a \ d) \ i = (\text{if} \ d \leq i \ \text{then} \ a * \text{coeff} \ p \ (i - d) \ \text{else} \ 0) \)
  using coeff-monom-mult[of a d p] by (simp add: ac-simps)

lemma smult-sum2: \( \text{smult} \ m \ (\sum i \in S. f\ i) = (\sum i \in S. \text{smult} \ m \ (f\ i)) \)
by (induct S rule: infinite-finite-induct, auto simp add: smult-add-right)

lemma deg-not-zero-imp-not-unit:
  fixes \( f: \text{poly} \)
  assumes \( \text{deg-f: degree} \ f > 0 \)
  shows \( \neg \text{is-unit} \ f \)
proof
  have \( \text{degree} \ (\text{normalize} \ f) > 0 \)
  using deg-f degree-normalize by auto
  hence normalize \( f \neq 1 \)
  by fastforce
  thus \( \neg \text{is-unit} \ f \) using normalize-1-iff by auto
qed
lemma conjugate-square-eq-0 [simp]:
  fixes x :: 'a :: {conjugatable-ring, semiring-no-zero-divisors}
  shows \( x \ast \text{conjugate } x = 0 \iff x = 0 \)
  by simp

lemma conjugate-square-greater-0 [simp]:
  fixes x :: 'a :: {conjugatable-ordered-ring, ring-no-zero-divisors}
  shows \( x \ast \text{conjugate } x > 0 \iff x \neq 0 \)
  using conjugate-square-positive[of x]
  by (auto simp: le_less)

lemma set-rows-carrier:
  assumes A ∈ carrier-mat m n and v ∈ set (rows A)
  shows v ∈ carrier-vec n
  using assms by (auto simp: set-conv-nth)

abbreviation vNil where vNil ≡ vec 0 undefined
definition vCons where vCons a v ≡ vec (Suc (dim-vec v)) (λi. case i of 0 ⇒ a | Suc n ⇒ v $ i)

lemma vec-index-vCons-0 [simp]: vCons a v $ 0 = a
  by (simp add: vCons-def)

lemma vec-index-vCons-Suc [simp]:
  fixes v :: 'a vec
  shows vCons a v $ Suc n = v $ n
proof−
  have 1: vec (Suc d) $ Suc n = vec d (f ∘ Suc) $ n for d and f :: nat ⇒ 'a
    by (transfer, auto simp: mk-vec-def)
  show ?thesis
    apply (auto simp: 1 vCons-def o-def) apply transfer apply (auto simp: mk-vec-def)
    done
qed

lemma vec-index-vCons: vCons a v $ n = (if n = 0 then a else v $ (n - 1))
  by (cases n, auto)

lemma dim-vec-vCons [simp]: dim-vec (vCons a v) = Suc (dim-vec v)
  by (simp add: vCons-def)

lemma vCons-carrier-vec[simp]: vCons a v ∈ carrier-vec (Suc n) ←→ v ∈ carrier-vec n
  by (auto dest!: carrier-vecI intro: carrier-vecI)
lemma vec-Suc: vec (Suc n) f = vCons (f 0) (vec n (f o Suc)) (is $l = ?r$
proof (unfold vec-eq-iff, intro conjI allI impI)
  fix i assume i < dim-vec ?r
  then show $l \ & i = ?r \ & i$ by (cases i, auto)
qed simp

declare Abs-vec-cases[cases del]

lemma vec-cases [case-names vNil vCons, cases type: vec]:
  assumes v = vNil \Longrightarrow thesis and \(\lambda a. v = vCons a w \Longrightarrow thesis\)
  shows thesis
proof (cases dim-vec v)
  case 0 then show thesis by (intro assms)
next
  case (Suc n)
  show thesis
  proof (rule assms)
    show v: v = vCons (v $ 0) (vec n (\lambda i. v $ Suc i)) (is v = ?r)
      proof (rule eq-vecI, unfold dim-vec-vCons dim-vec Suc)
    fix i
    assume i < Suc n
    then show v $ i = ?r $ i by (cases i, auto simp: vCons-def)
  qed simp
  qed

lemma vec-induct [case-names vNil vCons, induct type: vec]:
  assumes P vNil and \(\lambda a. v = vCons a w \Longrightarrow P (vCons a v)\)
  shows P v
proof (induct dim-vec v arbitrary:v)
  case 0 then show ?case by (cases v, auto intro: assms)
next
  case (Suc n) then show ?case by (cases v, auto intro: assms)
qed

lemma carrier-vec-induct [consumes 1, case-names 0 Suc, induct set:carrier-vec]:
  assumes v: v \in carrier-vec n
  and 1: P 0 vNil and 2: \(\lambda n a. v \in carrier-vec n \Longrightarrow P n v \Longrightarrow P (Suc n)\)
  (vCons a v)
  shows P n v
proof (insert v, induct n arbitrary: v)
  case 0 then have v = vec 0 undefined by auto
  with 1 show ?case by auto
next
  case (Suc n) then show ?case by (cases v, auto dest!: carrier-vecD intro:2)
qed

lemma vec-of-list-Cons[simp]: vec-of-list (a#as) = vCons a (vec-of-list as)
by (unfold vCons-def, transfer, auto simp:mk-vec-def split:nat.split)
lemma vec-of-list-Nil[simp]: vec-of-list [] = vNil
  by transfer auto

lemma scalar-prod-vCons[simp]:
  vCons a v · vCons b w = a · b + v · w
apply (unfold scalar-prod-def atLeast0-lessThan-Suc-eq-insert-0 dim-vec-vCons)
apply (subst sum.insert) apply (simp,simp)
apply (subst sum.reindex) apply force
apply simp
done

lemma zero-vec-Suc: 0 v (Suc n) = vCons 0 (0 v n)
by (auto simp: zero-vec-def vec-Suc o-def)

lemma zero-vec-zero[simp]: 0 v 0 = vNil
by auto

lemma vCons-eq-vCons[simp]: vCons a v = vCons b w ←→ a = b ∧ v = w (is ?l ←→ ?r)
proof
  assume ?l
  note arg-cong [OF this]
  from this[of dim-vec] this[of λx. x$0] this[of λx. x$Suc -]
  show ?r by (auto simp: vec-eq-iff)
qed simp

instantiation vec :: (conjugate) conjugate
begin

definition conjugate-vec :: 'a :: conjugate vec ⇒ 'a vec
  where conjugate v = vec (dim-vec v) (λi. conjugate (v $ i))

lemma conjugate-vCons [simp]:
  conjugate (vCons a v) = vCons (conjugate a) (conjugate v)
by (auto simp: vec-Suc conjugate-vec-def)

lemma dim-vec-conjugate[simp]: dim-vec (conjugate v) = dim-vec v
unfolding conjugate-vec-def by auto

lemma carrier-vec-conjugate[simp]: v ∈ carrier-vec n ⇒ conjugate v ∈ carrier-vec n
by (auto intro!: carrier-vecI)

lemma vec-index-conjugate[simp]:
  shows i < dim-vec v ⇒ conjugate v $ i = conjugate (v $ i)
unfolding conjugate-vec-def by auto

instance
proof
  fix v w :: 'a vec
  show conjugate (conjugate v) = v by (induct v, auto simp: conjugate-vec-def)
  let ?v = conjugate v
  let ?w = conjugate w
  show conjugate v = conjugate w $\iff$ v = w
proof (rule iffI)
  assume cvw: ?v = ?w show v = w
  proof (rule)
    have dim-vec ?v = dim-vec ?w using cvw by auto
    then show dim: dim-vec v = dim-vec w by simp
  fix i assume i: i < dim-vec w
  then have conjugate v $\langle i \rangle$ = conjugate w $\langle i \rangle$ using cvw by auto
  then have conjugate (v$\langle i \rangle$) = conjugate (w $\langle i \rangle$) using i dim by auto
  then show v $\langle i \rangle$ = w $\langle i \rangle$ by auto
  qed
  qed auto
  qed
end

lemma conjugate-add-vec:
  fixes v w :: 'a :: conjugatable-ring vec
  assumes dim: v : carrier-vec n w : carrier-vec n
  shows conjugate (v + w) = conjugate v + conjugate w
  by (rule, insert dim, auto simp: conjugate-dist-add)

lemma uminus-conjugate-vec:
  fixes v w :: 'a :: conjugatable-ring vec
  shows $(-\text{conjugate v}) = \text{conjugate (-v)}$
  by (rule, auto simp: conjugate-neg)

lemma conjugate-zero-vec[simp]:
  conjugate (0_v n :: 'a :: conjugatable-ring vec) = 0_v n by auto

lemma conjugate-vec-0[simp]:
  conjugate (vec 0 f) = vec 0 f by auto

lemma sprod-vec-0[simp]:
  v $\cdot$ vec 0 f = 0
  by (auto simp: scalar-prod-def)

lemma conjugate-zero-iff-vec[simp]:
  fixes v :: 'a :: conjugatable-ring vec
  shows conjugate v = 0_v n $\iff$ v = 0_v n
  using conjugate-cancel-iff[of - 0_v n :: 'a vec] by auto

lemma conjugate-smult-vec:
  fixes k :: 'a :: conjugatable-ring
  shows conjugate (k $\cdot$ v) = conjugate k $\cdot$ conjugate v

end
using conjugate-dist-mul by (intro eq-vecI, auto)

lemma conjugate-sprod-vec:
  fixes v w :: 'a :: conjugatable-ring vec
  assumes v : v : carrier-vec n and w : w : carrier-vec n
  shows conjugate (v · w) = conjugate v · conjugate w
proof (insert w v, induct w arbitrary: v rule: carrier-vec-induct)
  case 0 then show ?case by (cases v, auto)
next
  case (Suc n b w)
  then show ?case by (cases v, auto dest: carrier-vecD simp: conjugate-dist-add conjugate-dist-mul)
qed

abbreviation escalar-prod :: 'a vec ⇒ 'a vec ⇒ 'a :: conjugatable-ring (infix ·c 70)
  where op ·c ≡ λv w. v · conjugate w

lemma conjugate-conjugate-sprod[simp]:
  assumes v[simp]: v : carrier-vec n and w[simp]: w : carrier-vec n
  shows conjugate (conjugate v · w) = v ·c w
apply (subst conjugate-sprod-vec[of - n]) by auto

lemma conjugate-vec-sprod-comm:
  fixes v w :: 'a :: {conjugatable-ring, comm-ring} vec
  assumes v : v : carrier-vec n and w : w : carrier-vec n
  shows v ·c w = (conjugate w · v)
unfolding scalar-prod-def using assms by (subst sum-ivl-cong, auto simp: ac-simps)

lemma vec-carrier-vec[simp]: vec n f ∈ carrier-vec m ⟷ n = m
unfolding carrier-vec-def by auto

lemma conjugate-square-ge-0-vec[intro!]:
  fixes v :: 'a :: conjugatable-ordered-ring vec
  shows v ·c v ≥ 0
proof (induct v)
  case vNil
  then show ?case by auto
next
  case (vCons a v)
  then show ?case using conjugate-square-positive[of a] by auto
qed

lemma conjugate-square-eq-0-vec[simp]:
  fixes v :: 'a :: {conjugatable-ordered-ring, semiring-no-zero-divisors} vec
  assumes v ∈ carrier-vec n
  shows v ·c v = 0 ⟷ v = 0_v n
proof (insert assms, induct rule: carrier-vec-induct)
  case 0
  then show ?case by auto
next
  case (Suc n a v)
  then show ?case
    using conjugate-square-positive[of a] conjugate-square-ge-0-vec[of v]
    by (auto simp: le-less add-nonneg-eq-0-iff zero-vec-Suc)
qed

lemma conjugate-square-greater-0-vec[simp]:
  fixes a :: 'a :: {conjugatable-ordered-ring,semiring-no-zero-divisors} vec
  shows v · c v > 0 ←→ v ≠ 0 a n
  using assms by (auto simp: less-le)

lemma vec-conjugate-rat[simp]: (conjugate :: rat vec ⇒ rat vec) = (λx. x)
by force

lemma vec-conjugate-real[simp]: (conjugate :: real vec ⇒ real vec) = (λx. x)
by force

notation transpose-mat (\((-^T)\) [1000])

lemma cols-transpose[simp]: cols A \(^T\) = rows A unfolding cols-def rows-def by auto
lemma rows-transpose[simp]: rows A \(^T\) = cols A unfolding cols-def rows-def by auto
lemma list-of-vec-vec [simp]: list-of-vec (vec n f) = map f [0..<n]
  by (transfer, auto simp: mk-vec-def)

lemma list-of-vec-0 [simp]: list-of-vec (0 a n) = replicate n 0
  by (simp add: zero-vec-def map-replicate-trivial)

lemma diag-mat-map:
  assumes M-carrier: M ∈ carrier-mat n n
  shows diag-mat (map-mat f M) = map f (diag-mat M)
proof –
  have dim-eq: dim-row M = dim-col M using M-carrier by auto
  have m: map-mat f M $$ (i, i) = f (M $$ (i, i)) if i: i < dim-row M for i
    using dim-eq i by auto
  show ?thesis
    by (rule nth-equalityI, insert m, auto simp add: diag-mat-def M-carrier)
qed

lemma mat-of-rows-map [simp]:
  assumes x: set vs ⊆ carrier-vec n
  shows mat-of-rows n (map (map-vec f) vs) = map-mat f (mat-of-rows n vs)
proof –
  have ∀ x∈set vs. dim-vec x = n using x by auto
  then show ?thesis by (auto simp add: mat-eq-iff map-vec-def mat-of-rows-def)
qed

lemma mat-of-cols-map [simp]:
assumes $x$: $\text{set vs} \subseteq \text{carrier-vec n}$
shows $\text{mat-of-cols n} (\text{map (map-vec f) vs}) = \text{map-mat f (mat-of-cols n vs)}$
proof
  have $\forall x \in \text{set vs}. \text{dim-vec x} = n$ using $x$ by auto
  then show $\text{thesis}$ by (auto simp add: mat-eq-iff map-vec-def mat-of-cols-def)
qed

lemma $\text{vec-of-list-map}$ [simp]: $\text{vec-of-list (map f xs)} = \text{map-vec f (vec-of-list xs)}$
unfolding $\text{map-vec-def}$ by (transfer, auto simp add: mk-vec-def)

lemma $\text{map-vec}$: $\text{map-vec f (vec n g)} = \text{vec n (f o g)}$
by auto

lemma $\text{mat-of-cols-Cons-index-0}$: $i < n \Rightarrow \text{mat-of-cols n (w # ws)} \text{ } \text{ } (i, 0) = w \text{ } \text{ } (i, 0)$
by (unfold $\text{mat-of-cols-def}$, transfer', auto simp: mk-mat-def)

lemma $\text{mat-of-cols-Cons-index-Suc}$: $i < n \Rightarrow \text{mat-of-cols n (w # ws)} \text{ } \text{ } (i, \text{Suc j}) = \text{mat-of-cols n ws} \text{ } \text{ } (i, j)$
by (unfold $\text{mat-of-cols-def}$, transfer, auto simp: mk-mat-def undef-mat-def nth-append nth-map-out-of-bound)

lemma $\text{mat-of-cols-index}$: $i < n \Rightarrow j < \text{length ws} \Rightarrow \text{mat-of-cols n ws} \text{ } \text{ } (i,j) = \text{ws ! j} \text{ } \text{ } (i,j)$
by (unfold $\text{mat-of-cols-def}$, auto)

lemma $\text{transpose-mat-of-cols}$: $(\text{mat-of-cols n vs})^T = \text{mat-of-rows n vs}$
by (auto intro!: eq-matI simp: mat-of-rows-index mat-of-cols-index)

lemma $\text{transpose-mat-of-rows}$: $(\text{mat-of-rows n vs})^T = \text{mat-of-cols n vs}$
by (auto intro!: eq-matI simp: mat-of-rows-index mat-of-cols-index)

lemma $\text{vec-of-poly-0}$ [simp]: $\text{vec-of-poly 0} = 0$ by (auto simp: vec-of-poly-def)

lemma $\text{nth-list-of-vec}$ [simp]:
  assumes $i < \text{dim-vec v}$ shows $\text{list-of-vec v} ! i = v \text{ } \text{ } i$
using $\text{assms}$ by (transfer, auto)

lemma $\text{length-list-of-vec}$ [simp]:
  $\text{length (list-of-vec v)} = \text{dim-vec v}$ by (transfer, auto)

lemma $\text{vec-eq-0-iff}$:
  $v = 0_n \leftrightarrow n = \text{dim-vec v} \land (n = 0 \lor \text{set (list-of-vec v) = \{0\}})$ (is $?l \leftrightarrow {?r}$)
proof
show \( ?l \rightarrow ?r \) by auto
show \( ?r \rightarrow ?l \) by (intro iffI eq-vecI, force simp: set-conv-nth, force)
qed

lemma list-of-vec-vCons[simp]: list-of-vec (vCons a v) = a # list-of-vec v (is ?l = ?r)
proof (intro nth-equalityI allI impI)
  fix i
  assume i < length ?l
  then show \(?l!i = ?r!i\) by (cases i, auto)
qed simp

lemma append-vec-vCons: vCons a v @ v w = vCons a (v @ w) (is ?l = ?r)
proof (unfold vec-eq-iff, intro conjI allI impI)
  fix i
  assume i < dim-vec ?r
  then show \(?l$!i = ?r$!i\) by (cases i; subst index-append-vec, auto)
qed simp

lemma append-vec-vNil: vNil @ v v = v
by (unfold vec-eq-iff, auto)

lemma list-of-vec-append[simp]: list-of-vec (v @ w) = list-of-vec v @ list-of-vec w
by (induct v, auto)

lemma transpose-mat-eq[simp]: \( A^T = B^T \leftrightarrow A = B \)
using transpose-transpose by metis

lemma mat-col-eqI: assumes cols: \( \land i. i < \text{dim-col } B \rightarrow \text{col } A i = \text{col } B i \)
and dims: \( \text{dim-row } A = \text{dim-row } B \land \text{dim-col } A = \text{dim-col } B \)
shows A = B
by(subst transpose-mat-eq[symmetric], rule eq-rowI, insert assms,auto)

lemma upper-triangular-imp-det-eq-0-iff:
fixes A :: 'a :: idom mat
assumes A ∈ carrier-mat n n and upper-triangular A
shows \( \det A = 0 \leftrightarrow 0 \in \text{set } (\text{diag-mat } A) \)
using assms by (auto simp: det-upper-triangular)

lemma upper-triangular-imp-distinct:
fixes u :: 'a :: {zero-neq-one} poly
assumes A: A ∈ carrier-mat n n and tri: upper-triangular A
and diag: 0 /∈ set (diag-mat A)
shows distinct (rows A)
proof
  { fix i and j
    assume eq: rows A ! i = rows A ! j and ij: i < j and jn: j < n
  }
lemma vec-index-vec-of-poly [simp]: i ≤ degree p ⇒ vec-of-poly p $ i = coeff p (degree p − i)
by (simp add: vec-of-poly-def Let-def)

lemma poly-of-vec-vec: poly-of-vec (vec n f) = Poly (rev (map f [0..<n]))
proof (induct n arbitrary: f)
case 0
then show ?case by auto
next
case (Suc n)
have map f [0..<Suc n] = f 0 # map (f o Suc) [0..<n] by (simp add: map-upt-Suc del: upt-Suc)
also have Poly (rev ...) = Poly (rev (map (f o Suc) [0..<n])) + monom (f 0) n
by (simp add: Poly-snoc smult-monom)
also have ... = poly-of-vec (vec n (f o Suc)) + monom (f 0) n
by (fold Suc, simp)
also have ... = poly-of-vec (vec (Suc n) f)
apply (unfold poly-of-vec-def Let-def dim-vec sum-lessThan-Suc)
by (auto simp add: Suc-diff-Suc)
finally show ?case..
qed

lemma sum-list-map-dropWhile0:
assumes f0: f 0 = 0
shows sum-list (map f (dropWhile (op = 0) xs)) = sum-list (map f xs)
by (induct xs, auto simp add: f0)

lemma coeffs-poly-of-vec:
coeffs (poly-of-vec v) = rev (dropWhile (op = 0) (list-of-vec v))
proof -
obtain n f where v: v = vec n f by transfer auto
show ?thesis by (simp add: v poly-of-vec-vec)
qed

lemma poly-of-vec-vCons:
poly-of-vec (vCons a v) = monom a (dim-vec v) + poly-of-vec v (is ?l = ?r)
by (auto intro: poly-eqI simp: coeff-poly-of-vec vec-index-vCons)
lemma poly-of-vec-as-Poly: \( \text{poly-of-vec } v = \text{Poly} (\text{rev } (\text{list-of-vec } v)) \)
by \( \text{(induct } v, \text{ auto simp:poly-of-vec-vCons Poly-snoc ac-simps) \)}

lemma poly-of-vec-add:
assumes \( \text{dim-vec } a = \text{dim-vec } b \)
shows \( \text{poly-of-vec } (a + b) = \text{poly-of-vec } a + \text{poly-of-vec } b \)
using assms
by \( \text{(auto simp add: poly-eq-iff coeff-poly-of-vec) \)}

lemma degree-poly-of-vec-less:
assumes \( 0 < \text{dim-vec } v \) and \( \text{dim-vec } v \leq n \)
shows \( \text{degree } (\text{poly-of-vec } v) < n \)
using degree-poly-of-vec-less assms
by \( \text{(auto dest: less-le-trans) \)}

lemma (in vec-module) poly-of-vec-finsum:
assumes \( f \in X \rightarrow \text{carrier-vec } n \)
shows \( \text{poly-of-vec } (\text{finsum } V f X) = (\sum_{i \in X.} \text{poly-of-vec } (f i)) \)
proof \( \text{(cases finite } X) \)
  case False then show \( ?\text{thesis} \) by auto
next
  case True show \( ?\text{thesis} \) by \( \text{(auto simp: insert-absorb IH) \)}
  qed
qed

definition vec-of-poly-n \( p \ n = \)
\( \text{vec } n \ (\lambda i. \text{if } i < n - \text{degree } p - 1 \text{ then } 0 \text{ else } \text{coeff } p (n - i - 1)) \)
lemma vec-of-poly-as: vec-of-poly-as p (Suc (degree p)) = vec-of-poly p
  by (induct p, auto simp: vec-of-poly-def vec-of-poly-n-def)

lemma vec-of-poly-n-0 [simp]: vec-of-poly-n p 0 = vNil
  by (auto simp: vec-of-poly-n-def)

lemma vec-dim-of-poly [simp]:
  dim-vec (vec-of-poly-n p n) = n
  vec-of-poly-n p n ∈ carrier-vec n
  unfolding vec-of-poly-n-def by auto

lemma dim-vec-of-poly [simp]:
  dim-vec (vec-of-poly f) = degree f + 1
  by (simp add: vec-of-poly-as [symmetric])

lemma vec-index-of-poly-n:
  assumes i < n
  shows vec-of-poly-n p n i =
    (if i < n − Suc (degree p) then 0 else coeff p (n − i − 1))
  using assms by (auto simp: vec-of-poly-n-def Let-def)

lemma vec-of-poly-n-pCons [simp]:
  shows vec-of-poly-n (pCons a p) (Suc n) = vec-of-poly-n p n @ vec-of-list [a]
  (is i?l = i?r)
  proof (unfold vec-eq-iff, intro conjI allI impI)
    show dim-vec i?l = dim-vec i?r by auto
    show i < dim-vec i?r ⇒ i?l $ i = i?r $ i for i
      by (cases n − i, auto simp: coeff-pCons less-Suc-le vec-index-of-poly-n)
  qed

lemma vec-of-poly-pCons:
  shows vec-of-poly (pCons a p) =
    (if p = 0 then vec-of-list [a] else vec-of-poly p @ vec-of-list [a])
  by (cases degree p, auto simp: vec-of-poly-as [symmetric])

lemma list-of-vec-of-poly [simp]:
  list-of-vec (vec-of-poly p) = (if p = 0 then [] else rev (coeffs p))
  by (induct p, auto simp: vec-of-poly-pCons)

lemma poly-of-vec-of-poly-n:
  assumes p: degree p<n
  shows poly-of-vec (vec-of-poly-n p n) = p
  proof
    have vec-of-poly-n p n i $ (n − Suc i) = coeff p i if i: i < n for i
      proof
        have n: n − Suc i < n using i by auto
        have vec-of-poly-n p n i $ (n − Suc i) =
          (if n − Suc i < n − Suc (degree p) then 0 else coeff p (n − (n − Suc i) − 1))
      qed
  qed
using vec-index-of-poly-n[of n, of p].
also have ... = coeff p i using i n le-degree by fastforce
finally show ?thesis.
qed
moreover have coeff p i = 0 if i2: i ≥ n for i
by (rule coeff-eq-0, insert i2 p, simp)
ultimately show ?thesis
using assms
unfolding poly-eq-iff
unfolding coeff-poly-of-vec by auto
qed

lemma vec-of-poly-n0[simp]: vec-of-poly-n 0 n = 0
unfolding vec-of-poly-n-def by auto

lemma vec-of-poly-n-add: vec-of-poly-n (a + b) n = vec-of-poly-n a n + vec-of-poly-n b n
proof (induct n arbitrary: a b)
case 0
then show ?case by auto
next
case (Suc n)
then show ?case by (cases a, cases b, auto)
qed

lemma vec-of-poly-n-poly-of-vec:
assumes n: dim-vec g = n
shows vec-of-poly-n (poly-of-vec g) n = g
proof (auto simp add: poly-of-vec-def vec-of-poly-n-def assms vec-eq-iff Let-def)
have d: degree (∑i<n. monom (g $(n - Suc i)) i) = degree (poly-of-vec g)
unfolding poly-of-vec-def Let-def n by auto
fix i assume i1: i < n - Suc (degree (∑i<n. monom (g $(n - Suc i)) i))
and i2: i < n
have i3: i < n - Suc (degree (poly-of-vec g))
using i1 unfolding d by auto
hence dim-vec g - Suc i > degree (poly-of-vec g)
using n by linarith
then show g $ i = 0 using i1 i2 i3
by (metis (no-types, lifting) Suc-diff-Suc coeff-poly-of-vec diff-Suc-less
diff-diff-cancel leD le-degree less-imp-le-nat n neq0-conv)
next
fix i assume i < n
thus coeff (∑i<n. monom (g $(n - Suc i)) i) (n - Suc i) = g $ i
by (metis (no-types) Suc-diff-Suc coeff-poly-of-vec diff-diff-cancel
diff-less-Suc less-imp-le-nat n not-less-eq poly-of-vec-def)
qed

lemma poly-of-vec-scalar-mult:
assumes degree b<n

32
shows poly-of-vec \((a \cdot_{v} (\text{vec-of-poly-n} b n)) = \text{smult} a b\)
using assms
by (auto simp add: poly-eq-iff coeff-poly-of-vec vec-of-poly-n-def coeff-eq-0)

definition vec-of-poly-rev-shifted where
vec-of-poly-rev-shifted \(p n s j\) \(\equiv\)
vec \(n (\lambda i. \text{if } i \leq j \land j \leq s + i \text{ then coeff } p (s + i - j) \text{ else } 0)\)

lemma vec-of-poly-rev-shifted-dim[simp]: dim-vec (vec-of-poly-rev-shifted \(p n s j\)) = \(n\)
unfolding vec-of-poly-rev-shifted-def by auto

lemma col-sylvester-sub:
assumes \(j: j < m + n\)
shows \(\text{col} (\text{sylvester-mat-sub} m n p q) j = \text{vec-of-poly-rev-shifted} p n m j \cdot_{v} \text{vec-of-poly-rev-shifted} q m n j (\text{is } ?l = ?r)\)
proof
show \(?l \equiv \text{vec-of-poly-rev-shifted-def}\)
apply \((\text{subst index-col})\) using \(i\) apply simp using \(j\) apply simp
apply \((\text{subst sylvester-mat-sub-index})\) using \(i\) apply simp using \(j\) apply simp
apply \((\text{cases } i < n)\) using \(i\) apply force using \(i\)
apply \((\text{auto simp: not-less not-le intro: coeff-eq-0})\)
done
qed

lemma vec-of-poly-rev-shifted-scalar-prod:
fixes \(p v\)
defines \(q \equiv \text{poly-of-vec} v\)
assumes \(m: \text{degree } p \leq m\) and \(n: \text{dim-vec } v = n\)
assumes \(j: j < m + n\)
shows \(\text{vec-of-poly-rev-shifted} p n m (n + m - \text{Suc } j) \cdot v = \text{coeff} (p \ast q) j (\text{is } ?l = ?r)\)
proof
- have \(id1: \bigwedge i. m + i - (n + m - \text{Suc } j) = i + \text{Suc } j - n\)
  using \(j\) by auto
- let \(?g = \lambda i. \text{if } i \leq n + m - \text{Suc } j \land n - \text{Suc } j \leq i \text{ then coeff } p (i + \text{Suc } j - n) \ast v \text{ else } 0\)
  have \(?thesis = ((\sum i = 0..<n. \text{?g } i) = (\sum i \leq j. \text{coeff } p i \ast (\text{if } j - i < n \text{ then } v \$ (n - \text{Suc } (j - i)) \text{ else } 0))) (\text{is -} = (?l = ?r)))\)
  unfolding vec-of-poly-rev-shifted-def coeff-mult m scalar-prod-def n q-def coeff-poly-of-vec
  by (subst sum.cong, insert \(id1\), auto)
also have ...
proof -
have \( \forall r = (\sum_{i \leq j}. (\text{if } j - i < n \text{ then } \text{coeff } p i \times v \$(n - \text{Suc } (j - i)) \text{ else } 0)) \) (is \( = \text{sum } \forall f \))
   by (rule \text{sum.cong}, \text{auto})
also have \( \text{sum } \forall f \{.j\} = \text{sum } \forall f \{\{i. i \leq j \wedge j - i < n\} \cup \{i. i \leq j \wedge \neg j - i < n\}\})
   (is \( = \text{sum } - (\forall R1 \cup \forall R2)) \)
   by (rule \text{sum.cong}, \text{auto})
also have \( \ldots = \text{sum } \forall f \forall R1 + \text{sum } \forall f \forall R2 \)
   by (subst \text{sum.union-disjoint}, \text{auto})
also have \( \text{sum } \forall f \forall R2 = 0 \)
   by (rule \text{sum.neutral}, \text{auto})
also have \( \text{sum } \forall f \forall R1 + 0 = \text{sum } (\lambda i. \text{coeff } p i \times v $(i + n - \text{Suc } j)) \forall R1 \)
   (is \( = \text{sum } \forall f \))
   by (subst \text{sum.cong}, \text{auto simp: ac-simps})
also have \( \ldots = \text{sum } \forall f ((\forall R1 \cap \{.m\}) \cup (\forall R1 - \{.m\})) \)
   (is \( = \text{sum } - (\forall R \cup \forall R')) \)
   by (rule \text{sum.cong}, \text{auto})
also have \( \ldots = \text{sum } \forall f \forall R + \text{sum } \forall f \forall R' \)
   by (subst \text{sum.union-disjoint}, \text{auto})
also have \( \text{sum } \forall f \forall R' = 0 \)
proof -
{|}
   fix \( x \)
   assume \( x > m \)
   with \( m \)
   have \( \forall f x = 0 \) by (subst \text{coeff-eq-0}, \text{auto})
{|}
thus \( \forall \text{thesis} \)
   by (subst \text{sum.neutral}, \text{auto})
qed
finally have \( r = \text{sum } \forall f \forall R \) by \text{simp}

have \( \forall l = \text{sum } \forall g \{\{i. i < n \wedge i \leq n + m - \text{Suc } j \wedge \neg n - \text{Suc } j \leq i\} \cup \{i. i < n \wedge \neg (i \leq n + m - \text{Suc } j \wedge \neg n - \text{Suc } j \leq i)\}\}
   (is \( = \text{sum } - (\forall L1 \cup \forall L2)) \)
   by (rule \text{sum.cong}, \text{auto})
also have \( \ldots = \text{sum } \forall g \forall L1 + \text{sum } \forall g \forall L2 \)
   by (subst \text{sum.union-disjoint}, \text{auto})
also have \( \text{sum } \forall g \forall L2 = 0 \)
   by (rule \text{sum.neutral}, \text{auto})
also have \( \text{sum } \forall g \forall L1 + 0 = \text{sum } (\lambda i. \text{coeff } p (i + \text{Suc } j - n) \times v $(i) \forall L1 \)
   (is \( = \text{sum } \forall G \))
   by (subst \text{sum.cong}, \text{auto})
also have \( \ldots = \text{sum } \forall G ((\forall L1 \cap \{i. i + \text{Suc } j - n \leq m\}) \cup (\forall L1 - \{i. i + \text{Suc } j - n \leq m\})) \)
   (is \( = \text{sum } - (\forall L \cup \forall L')) \)
   by (subst \text{sum.cong}, \text{auto})
also have \( \ldots = \text{sum } \forall G \forall L + \text{sum } \forall G \forall L' \)
   by (subst \text{sum.union-disjoint}, \text{auto})

34
also have \( \sum ?G ?L' = 0 \)
proof -
{  
  fix \( x \)
  assume \( x + \text{Suc } j - n > m \)
  with \( m \)
  have \( ?G x = 0 \) by (subst coeff-eq-0, auto)
}
thus \( \text{thesis} \)
by (subst sum.neutral, auto)
qed
finally have \( l : ?l = \sum ?G ?L \) by simp

let \( ?bij = \lambda i. i + n - \text{Suc } j \)
{  
  fix \( x \)
  assume \( x: j < m + n \) Suc \( x + j \) - \( n \leq m \) \( x < n \) \( n - \text{Suc } j \leq x \)
  define \( y \) where \( y = x + \text{Suc } j - n \)
  from \( x \) have \( x + \text{Suc } j \geq n \) by auto
  with \( x \) have \( xy: x = ?bij y \) unfolding \( y\)-def by auto
  from \( x \) have \( y: y \in ?R \) unfolding \( y\)-def by auto
  have \( x \in ?bij ' ?R \) unfolding \( xy\) using \( y \) by blast
}
  note tedious = this
  show \( \text{thesis} \)
  unfolding \( l \) \( r \)
  by (rule sum.reindex-cong[of \( ?bij \), insert \( j \), auto simp: inj-on-def tedious])
qed
finally show \( \text{thesis} \) by simp
qed

lemma sylvester-sub-poly:
fixes \( p \ q : 'a :: \text{comm-semiring-0 poly} \)
assumes \( m: \text{degree } p \leq m \)
assumes \( n: \text{degree } q \leq n \)
assumes \( v: v \in \text{carrier-vec} (m+n) \)
shows \( \text{poly-of-vec} (\text{vec-first } v n) * p + \text{poly-of-vec} (\text{vec-last } v m) * q \) (is \( ?l = ?r \))
proof (rule poly-eqI)
  fix \( i \)
  let \( ?Tv = (\text{vec-first } v n) * p + \text{vec-last } v m \)
  have \( \text{dim } (\text{vec-first } v n) = n \text{ dim } (\text{vec-last } v m) = m \text{ dim } ?Tv = n + m \)
  using \( v \) by auto
  have if-distrib: \( \bigwedge x y z. (if x \text{ then } y \text{ else } (0 :: 'a)) * z = (if x \text{ then } y \times z \text{ else } 0) \)
  by auto
  show \( \text{coeff } ?l \ i = \text{coeff } ?r \ i \)
  proof (cases \( i < m+n \))
    case False
    hence \( i \text{-mn: } i \geq m+n \)
    and \( i \text{-n: } \bigwedge x. x \leq i \land x < n \iff x < n \)
    qed
and i-m: \( \forall x. x \leq i \land x < m \iff x < m \) by auto

have coeff \(?r i\) =

\[
(\sum_{x < n} \text{vec-first } v \ n \ (n - \text{Suc } x) \ast \text{coeff } p \ (i - x)) +
(\sum_{x < m} \text{vec-last } v \ m \ (m - \text{Suc } x) \ast \text{coeff } q \ (i - x))
\]

(is \(- \ast \text{sum } ?f \ + \ \text{sum } ?g \))

unfolding coeff-add coeff-mult Let-def
unfolding coeff-poly-of-vec dim if-distrib
unfolding atMost-def
apply(subst sum.inter-filter[symmetric],simp)
apply(subst sum.inter-filter[symmetric],simp)
unfolding mem-Collect-eq
unfolding i-n i-m
unfolding lessThan-def by simp
also \{ fix x assume x: x < n

have coeff \(p \ (i-x)\) = 0

apply(rule coeff-eq-0) using i-mn x m by auto

hence \(?f x\) = 0 by auto
\}

hence sum \(?f \{\ldots < n\}\) = 0 by simp
also \{ fix x assume x: x < m

have coeff \(q \ (i-x)\) = 0

apply(rule coeff-eq-0) using i-mn x n by auto

hence \(?g x\) = 0 by auto
\}

hence sum \(?g \{\ldots < m\}\) = 0 by auto
finally have coeff \(?r i\) = 0 by auto
also from False have \(0 = \text{coeff } ?l i\)
finally show \(?\text{thesis by auto}\)

next case True

hence coeff \(?! i\) = ((sylvester-mat-sub \(m \ n \ p \ q\) \(T\) \(, \ v\)) \$ \(n + m - \text{Suc } i\)

unfolding coeff-poly-of-vec dim sum.distrib[symmetric] by auto
also have \(...) = coeff \(p \ast \text{poly-of-vec } (\text{vec-first } v \ n) + q \ast \text{poly-of-vec } (\text{vec-last } v \ m))\ i

apply(subst index-mult-mat-vec) using True apply simp
apply(subst row-transpose) using True apply simp
apply(subst col-sylvester-sub)
using True apply simp
apply(subst vec-first-last-append[of v n m,symmetric]) using v apply(simp add: add.commute)
apply(subst vec-first-cst-vec[cst[of v n m,symmetric]])
apply (rule carrier-vecI,simp)
apply (subst vec-of-poly-rev-shifted-scalar-prod[OF \(m\),simp]) using True apply simp
apply (subst add.commute[of n m])
apply (subst vec-of-poly-rev-shifted-scalar-prod[OF \(n\)])

True apply simp
by simp
also have \(...) =

(\sum_{x \leq i. (if x < n then vec-first v \ n \ (n - \text{Suc } x) else 0) \ast \text{coeff } p \ (i - x)) +
\]
\[
(\sum_{x \leq i} (\text{if } x < m \text{ then vec-last } v m \text{ } (m - \text{Suc } x) \text{ else } 0) \ast \text{coeff } q (i - x))
\]

unfolding coeff-poly-of-vec[of vec-first v n, unfolded dim-vec-first, symmetric]
unfolding coeff-poly-of-vec[of vec-last v m, unfolded dim-vec-last, symmetric]
unfolding coeff-mult[symmetric] by (simp add: mult.commute)
also have \(\ldots = \text{coeff } ?r i\)
unfolding coeff-add coeff-mult Let-def
unfolding coeff-poly-of-vec dim..
finally show \?thesis.

qed

lemma normalize-field [simp]: normalize \((a :: 'a :: \{field, semiring-gcd\})\) = (if \(a = 0\) then 0 else 1)
using unit-factor-normalize by fastforce

lemma content-field [simp]: content \((p :: 'a :: \{field, semiring-gcd\} poly)\) = (if \(p = 0\) then 0 else 1)
by (induct p, auto simp: content-def)

lemma primitive-part-field [simp]: primitive-part \((p :: 'a :: \{field, semiring-gcd\} poly)\) = \(p\)
by (cases \(p = 0\), auto intro!: primitive-part-prim)

lemma primitive-part-dvd: primitive-part \(a\) dvd \(a\)
by (metis content-times-primitive-part dvd-def dvd-refl mult-smult-right)

lemma degree-abs [simp]:
degree \(|p|\) = degree \(p\) by (auto simp: abs-poly-def)

lemma degree-gcd1:
assumes \(a \neq 0\)
shows \(\text{degree } (\text{gcd } a b) \leq \text{degree } a\)
proof -
let \(?g = \text{gcd } a b\)
have \(?g\) dvd-dvd-b: \(?g\) dvd \(a\) by simp
from this obtain \(c\) where \(a-gc: a = \?g * c\) unfolding dvd-def by auto
have \(?g \neq 0\) using \(a-not0\) a-gc by auto
have \(?c: c \neq 0\) using \(a-not0\) a-gc by auto
have \(\text{degree } \?g \leq \text{degree } (\?g * c)\) by (rule degree-mult-right-le[OF \(?c0\)])
also have \(\ldots = \text{degree } a\) using \(a-gc\) by auto
finally show \?thesis .
qed

lemma primitive-part-neg [simp]:
fixes $a :: \text{factorial-ring-gcd poly}$
shows $\text{primitive-part} (-a) = - \text{primitive-part} a$
proof
have $\text{primitive-part} (-a) = \text{primitive-part} (\text{smult} (-1) a)$ by auto
then show $?thesis$ unfolding $\text{primitive-part-smult}$
  by (simp add: $\text{is-unit-unit-factor}$)
qed

lemma $\text{content-uminus}[\text{simp}]$:
fixes $f :: \text{int poly}$
shows $\text{content} (-f) = \text{content} f$
proof
have $-f = - \text{(smult 1 f)}$ by auto
also have $\ldots = \text{smult} (-1) f$ using $\text{smult-minus-left}$ by auto
finally have $\text{content} (-f) = \text{content} (\text{smult} (-1) f)$ by auto
also have $\ldots = \text{normalize} (-1) * \text{content} f$ unfolding $\text{content-smult}$
finally show $?thesis$ by auto
qed

lemma $\text{pseudo-mod-monic}$:
fixes $f \cdot g :: \text{comm-ring-1, semiring-1-no-zero-divisors poly}$
defines $r \equiv \text{pseudo-mod} f g$
assumes $\text{monic-g}$
shows $\exists q. f = g * q + r \land r = 0 \lor \text{degree} r < \text{degree} g$
proof
let $\text{?cg} = \text{coeff} g (\text{degree} g)$
let $\text{?cge} = \text{?cg} ^ \text{(Suc (degree f) - degree g)}$
define $a$ where $a = ?cge$
from $\text{r-def}[\text{unfolded pseudo-mod-def}]$ obtain $q$ where $\text{pdm: pseudo-divmod} f g$
  $(q, r) = (q, r)$
  by (cases pseudo-divmod $f g$) auto
have $g: g \neq 0$ using $\text{monic-g}$ by auto
from $\text{pseudo-divmod}[\text{OF} g \text{ pdm}]$ have $\text{id: smult a f = g * q + r}$ and $r = 0 \lor \text{degree} r < \text{degree} g$
  by (auto simp: $\text{a-def}$)
with $\text{id2}$ show $\exists q. f = g * q + r$
  by auto
qed

lemma $\text{monic-imp-div-mod-int-poly-degree}$:
fixes $p :: \text{comm-ring-1, semiring-1-no-zero-divisors poly}$
assumes $m: \text{monic} a$
shows $\exists q r. p = q * u + r \land (r = 0 \lor \text{degree} r < \text{degree} u)$
using $\text{pseudo-mod-monic}[\text{OF} m]$ using $\text{mult.commute}$ by metis
corollary monic-imp-div-mod-int-poly-degree2:
  fixes p :: 'a::{comm_ring_1,semiring_1-no-zero-divisors} poly
  assumes m: monic u and deg-u: degree u > 0
  shows ∃q r. p = q*u + r ∧ (degree r < degree u)
proof –
  obtain q r where p = q * u + r and r: (r = 0 ∨ degree r < degree u)
    using monic-imp-div-mod-int-poly-degree[OF m, of p] by auto
  moreover have degree r < degree u using deg-u r by auto
  ultimately show ?thesis by auto
  qed

lemma det-identical-columns:
  assumes A: A ∈ carrier-mat n n
     and ij: i ≠ j
     and i: i < n and j: j < n
     and r: col A i = col A j
  shows det A = 0
proof –
  have det A = det Aᵀ using det-transpose[OF A] ...
  also have ... = 0
  proof (rule det-identical-rows[of - n i j])
    show row (transpose-mat A) i = row (transpose-mat A) j
      using A i j r by auto
  qed (auto simp add: assms)
  finally show ?thesis .
  qed

lemma irreducible-uminus [simp]:
  fixes a::'a::idom
  shows irreducible (-a) ↔ irreducible a
  using irreducible-mult-unit-left[of -1::'a] by auto

context poly-mod
begin

lemma dvd-imp-dvdm:
  assumes a dvd b shows a dvdm b
  by (metis assms dvd-def dvdm-def)
lemma dvdm-add:
  assumes a: u dvdm a
  and b: u dvdm b
  shows u dvdm (a+b)
proof
  obtain a' where a = m \cdot a' using a unfolding dvdm-def by auto
  obtain b' where b = m \cdot b' using b unfolding dvdm-def by auto
  have Mp (a + b) = Mp (u \cdot a' + u \cdot b') using a b
    by (metis poly-mod.plus-Mp(1) poly-mod.plus-Mp(2))
  also have ... = Mp (u * (a' + b'))
    by (simp add: distrib-left)
  finally show ?thesis unfolding dvdm-def by auto
qed

lemma monic-dvdm-constant:
  assumes uk: u dvdm [:k:]
  and u1: monic u and u2: degree u > 0
  shows k mod m = 0
proof
  have d1: degree-m [:k:] = degree [:k:]
    by (metis degree-pCons-0 le-zero-eq poly-mod.degree-m-le)
  obtain h where h: \text{Mp} [:k:] = \text{Mp} (u \cdot h)
    using uk unfolding dvdm-def by fast
  have (u \cdot h) = \text{Mp} (u \cdot h) + \text{smult} m \cdot (\text{Dp} (u \cdot h))
    by (simp add: poly-mod.Dp-Mp-eq[of u])
  hence uq: \text{Mp} (u \cdot h) = (u \cdot h) - \text{smult} m \cdot (\text{Dp} (u \cdot h))
    by auto
  have g: g = \text{Mp} g + \text{smult} m \cdot \text{Dp} g
    by (simp add: poly-mod.Dp-Mp-eq[of g])
  also have ... = \text{poly-mod.Mp} m (u \cdot h') + \text{smult} m \cdot (\text{Dp} g) using q by simp

lemma dvdm-imp-div-mod:
  assumes u dvdm g
  shows \exists q r. g = q \cdot u + \text{smult} m \cdot r
proof
  obtain q where q: Mp g = Mp (u \cdot q)
    using assms unfolding dvdm-def by fast
  have (u \cdot q) = Mp (u \cdot q) + \text{smult} m \cdot (\text{Dp} (u \cdot q))
    by (simp add: poly-mod.Dp-Mp-eq[of u])
  hence uq: Mp (u \cdot q) = (u \cdot q) - \text{smult} m \cdot (\text{Dp} (u \cdot q))
    by auto
  have g: g = Mp g + \text{smult} m \cdot (\text{Dp} g)
    by (simp add: poly-mod.Dp-Mp-eq[of g])
  also have ... = \text{poly-mod.Mp} m (u \cdot q') + \text{smult} m \cdot (\text{Dp} g) using q by simp

40
also have \( \ldots = u * q - \text{smult} m (Dp (u * q)) + \text{smult} m (Dp g) \)
unfolding \( uq \) by auto
also have \( \ldots = u * q + \text{smult} m (-Dp (u*q)) + \text{smult} m (Dp g) \) by auto
also have \( \ldots = u * q + \text{smult} m (-Dp (u*q) + Dp g) \)
unfolding \( \text{smult-add-right} \) by auto
finally show \( ?\text{thesis} \) by auto

qed

lemma \( \text{div-mod-imp-dvdm} \):
assumes \( \exists q r. b = q * a + \text{Polynomial.smult m r} \)
shows \( a \text{ dvdm b} \)
proof
from assms obtain \( q r \) where \( b:b = a * q + \text{smult m r} \)
  by (metis \text{mult.commutate})
have \( a: \text{Mp (Polynomial.smult m r)} = 0 \) by auto
show \( ?\text{thesis} \)
proof (unfold \text{dvdm-def}, rule \text{exI}[\text{of - q}])
  have \( \text{Mp (a * q + \text{smult m r})} = \text{Mp (a * q + \text{Mp (smult m r)})} \)
    using \text{plus-Mp(2)[of a*q smult m r]} by auto
  also have \( \ldots = \text{Mp (a*q)} \) by auto
  finally show \( \text{eq-m b (a * q)} \) using \( b \) by auto
qed
qed

corollary \( \text{div-mod-iff-dvdm} \):
shows \( a \text{ dvdm b} = (\exists q r. b = q * a + \text{Polynomial.smult m r}) \)
using \( \text{div-mod-imp-dvdm dvdm-imp-div-mod by blast} \)

lemma \( \text{dvdmE} \):
assumes \( p \text{ dvdm q} \) and \( \forall r. q = m p \text{ Mp r} \Rightarrow \text{thesis} \)
shows \( \text{thesis} \)
using assms by (auto simp: \text{dvdm-def})

lemma \( \text{lead-coeff-monic-mult} \):
fixes \( p :: \text{\{comm-semiring-1,semiring-no-zero-divisors\}} \) \( \text{poly} \)
assumes \( \text{monic p} \)
shows \( \text{lead-coeff (p * q)} = \text{lead-coeff q} \)
using assms by (simp add: \text{lead-coeff-mult})

lemma \( \text{degree-m-mult-eq} \):
assumes \( p: \text{monic p} \) and \( q: \text{lead-coeff q mod m \neq 0 and m1: m > 1} \)
shows \( \text{degree (Mp (p * q))} = \text{degree p + degree q} \)
proof
  have \( \text{lead-coeff (p * q) mod m \neq 0} \)
    using \( q p \) by (auto simp: \text{lead-coeff-monic-mult})
  with \( m1 \) show \( ?\text{thesis} \)
    by (auto simp: \text{degree-m-eq intro!: degree-mult-eq})
qed
lemma dvdm-imp-degree-le:
assumes pq: p dvdm q and p: monic p and q0: Mp q ≠ 0 and m1: m > 1
shows degree p ≤ degree q
proof −
  from q0
  have q: lead-coeff (Mp q) mod m ≠ 0
    by (metis Mp-Mp Mp-coeff leading-coeff-neq-0 M-def)
  from pq obtain r where Mpq: Mp q = Mp (p * Mp r) by (auto elim: dvdmE)
  with p q have lead-coeff (Mp r) mod m ≠ 0
    by (metis Mp-Mp Mp-coeff leading-coeff-0-iff mult-poly-0-right M-def)
  from degree-m-mult-eq[OF p this m1] Mpq
  have degree p ≤ degree-m q by simp
  thus ?thesis using degree-m-le le-trans by blast
qed

lemma dvdm-uminus [simp]:
p dvdm − q ←→ p dvdm q
by (metis add.inverse-inverse dvdm-smult smult-1-left smult-minus-left)

lemma Mp-const-poly: Mp [\,a\,] = [\,a \bmod m\,]
by (simp add: Mp-def M-def Polynomial.map-poly-pCons)

end

case

context poly-mod-2
begin

lemma factorization-m-mem-dvdm: assumes fact: factorization-m f (c,fs)
  and mem: Mp g ∈ # image-mset Mp fs
shows g dvdm f
proof −
  from fact have factorization-m f (Mf (c, fs)) by auto
  then obtain l where f: factorization-m f (l, image-mset Mp fs) by (auto simp: Mf-def)
  from multi-member-split[OF mem] obtain ls where
    fs: image-mset Mp fs = (# Mp g #) + ls by auto
  from f[unfolded fs split factorization-m-def] show g dvdm f
    unfolding dvdm-def
    by (intro ext1[of - smult l (prod-mset ls)], auto simp del: Mp-smult
      simp add: Mp-smult(2)[of - Mp g * prod-mset ls, symmetric], simp)
qed

lemma dvdm-degree: monic u ⟹ u dvdm f ⟹ Mp f ≠ 0 ⟹ degree u ≤ degree f
  using dvdm-imp-degree-le m1 by blast
end

context poly-mod-prime
lemma \textit{pl-dvdm-imp-p-dvdm}:
\begin{itemize}
  \item \textbf{assumes} \(l_0: l \neq 0\)
  \item \textbf{and} \(\text{pl-dvdm}; \text{poly-mod.dvdm} (p^l) a b\)
\end{itemize}
\textbf{shows} \(a \text{ dvdm } b\)
\begin{itemize}
  \item \textbf{proof} –
  \item \textbf{from} \(l_0\) \textbf{have} \(l > 0\) \textbf{by} \text{auto}
  \item \textbf{with} \(m_1\) \textbf{interpret} \(pl: \text{poly-mod-2} p^l\) \textbf{by} \((\text{unfold-locales, auto})\)
  \item \textbf{have} \(p\text{-\textsc{rw}}: p * p ^ (l - 1) = p ^ l\) \textbf{by} \((\text{rule power-minus-simp[symmetric, OF l-gt-0]})\)
  \item \textbf{obtain} \(q r\) \textbf{where} \(b = q * a + \text{smult} (p^l) r\) \textbf{using} \(pl\text{-dvdm-imp-div-mod[OF pl-dvdm]}\) \textbf{by} \text{auto}
  \item \textbf{have} \(\text{smult} (p^l) r = \text{smult} p \text{ (smult} (p ^ (l - 1)) r)\) \textbf{unfolding} \(\text{smult-smult}\)
  \item \textbf{hence} \(b_2\): \(b = q * a + \text{smult} p \text{ (smult} (p ^ (l - 1)) r)\) \textbf{using} \(b\) \textbf{by} \text{auto}
  \item \textbf{show} \(\text{?thesis}\)
  \item \textbf{by} \((\text{rule div-mod-imp-dvdm, rule exI[of - q], rule exI[of - (smult} (p ^ (l - 1)) r)], auto simp add: b2})\)
\end{itemize}
\textbf{qed}

lemma \textit{coprime-exp-mod}: \(\text{coprime} lu p \Rightarrow n \neq 0 \Rightarrow \text{lu mod} p ^ n \neq 0\)
\textbf{using} \(\text{prime}\) \textbf{by} \text{fastforce}

lemma \textit{unique-factorization-m-factor-partition}:
\begin{itemize}
  \item \textbf{assumes} \(l_0: l \neq 0\)
  \item \textbf{and} \(uf: \text{poly-mod.unique-factorization-m} (p^l) f (\text{lead-coeff} f, \text{mset} gs)\)
  \item \textbf{and} \(f: f = f_1 * f_2\)
  \item \textbf{and} \(cop: \text{coprime} (\text{lead-coeff} f) p\)
  \item \textbf{and} \(sf: \text{square-free-m} f\)
  \item \textbf{and} \(part: \text{partition} (\text{AgI.} \text{ gi dvdm} f_1) gs = (gs1, gs2)\)
\end{itemize}
\textbf{shows} \(\text{poly-mod.unique-factorization-m} (p^l) f (\text{lead-coeff} f_1, \text{mset} gs_1)\)
\(\text{poly-mod.unique-factorization-m} (p^l) f_2 (\text{lead-coeff} f_2, \text{mset} gs_2)\)
\begin{itemize}
  \item \textbf{proof} –
  \item \textbf{interpret} \(pl: \text{poly-mod-2} p^l\) \textbf{by} \((\text{standard, insert m1 l0, auto})\)
  \item \textbf{let} \(?I = \text{image-mset} pl Mp\)
  \item \textbf{note} \(Mp\text{-pow [simp] = Mp\text{-Mp-pow-is-Mp[OF l0 m1]}}\)
  \item \textbf{have} \([\text{simp]}]: pl. Mp x dvdm u = (x dvdm u) \text{ for} x u \text{ unfolding dvdm-def using} Mp\text{-pow[of x]}\)
  \item \textbf{by} \((\text{metis poly-mod.mul-Mp(1)})\)
  \item \textbf{have} \(gs\text{-split: set} gs = \text{set} gs_1 \cup \text{set} gs_2\) \textbf{using} \(part\) \textbf{by} \text{auto}
  \item \textbf{from} \(\text{pl.unique-factorization-m-factor[OF prime uf[unfolded f] - - l0 refl, folded f, OF cop sf]}\)
  \item \textbf{obtain} \(hs_1 hs_2\) \textbf{where} \(af': \text{pl.unique-factorization-m} f_1 (\text{lead-coeff} f_1, hs_1)\)
  \item \(\text{pl.unique-factorization-m} f_2 (\text{lead-coeff} f_2, hs_2)\)
  \item \textbf{and} \(gs\text{-hs:} \ ?I \text{ (mset} gs) = hs_1 + hs_2\)
  \item \textbf{unfolding pl.Mf-def split by} \text{auto}
\end{itemize}
have gs-gs: ?I (mset gs) = ?I (mset gs1) + ?I (mset gs2) using part by (auto, induct gs arbitrary: gs1 gs2, auto)

with gs-hs have gs-hs12: ?I (mset gs1) + ?I (mset gs2) = hs1 + hs2 by auto

note pl-dvdm-imp-p-dvdm = pl-dvdm-imp-p-dvdm[OF l0]

note fact = pl.unique-factorization-m-imp-factorization[OF uf]

have gs1: ?I (mset gs1) = \{#x ∈ #?I (mset gs). x dvd f1#\}

using part by (auto, induct gs arbitrary: gs1 gs2, auto)

also have ... = \{#x ∈ #?I (mset gs). x dvd f1#\} + \{#x ∈ #?I (mset gs). x dvd f1#\}

unfolding gs-hs by simp

also have \{#x ∈ #?I (mset gs). x dvd f1#\} = \{}

proof (rule ccontr)

  assume ¬ ?thesis

  then obtain x where x: x ∈ #?I (mset gs). x dvd f1 by fastforce

  from x gs-hs have x ∈ #?I (mset gs) by auto

  with fact[unfolded pl.factorization-m-def]

  have xx: pl.irreducible_d m x monic x by auto

  from square-free-m-prod-imp-coprime-m[OF sf[unfolded f]]

  have cop-h-f: coprime-m f1 f2 by auto

  from pl.factorization-m-mem-dvdm[OF pl.unique-factorization-m-imp-factorization[OF uf'(2)], of x] x

  have pl.dvdm x f2 by auto

  hence x dvd f2 by (rule pl-dvdm-imp-p-dvdm)

  from cop-h-f[unfolded coprime-m-def, rule-format, OF dvd this]

  have x dvd f1 by auto

  from dvdm-imp-degree-le[OF this xx(2) - m1] have degree x = 0 by auto

  with xx show False unfolding pl.irreducible_d m-def by auto

qed

also have \{#x ∈ #?I (mset gs). x dvd f1#\} = hs1

proof (rule ccontr)

  assume ¬ ?thesis

  from filter-mset-inequality[OF this]

  obtain x where x: x ∈ #hs1 and dvd: ¬ x dvd f1 by blast

  from pl.factorization-m-mem-dvdm[OF pl.unique-factorization-m-imp-factorization[OF uf'(1)], of x] x

  have pl.dvdm x f1 by auto

  from pl-dvdm-imp-p-dvdm[OF this] dvd show False by auto

qed

finally have gs-hs1: ?I (mset gs1) = hs1 by simp

with gs-hs12 have ?I (mset gs2) = hs2 by auto

with uf' gs-hs1 have pl.unique-factorization-m f1 (lead-coeff f1, ?I (mset gs1))

  pl.unique-factorization-m f2 (lead-coeff f2, ?I (mset gs2)) by auto

thus pl.unique-factorization-m f1 (lead-coeff f1, mset gs1)

  pl.unique-factorization-m f2 (lead-coeff f2, mset gs2)

unfolding pl.unique-factorization-m-def

  by (auto simp: pl.Mf-def image-mset.compositionality o-def)

qed

end
definition find-indices where find-indices x xs ≡ [i ← [0..<length xs], xs!i = x]

lemma find-indices-nil [simp]:
find-indices x [] = []
by (simp add: find-indices-def)

lemma find-indices-Cons:
find-indices x (y#ys) = (if x = y then Cons 0 else id) (map Suc (find-indices x ys))
apply (unfold find-indices-def length-Cons, subst upt-conv-Cons, simp)
apply (fold map-Suc-upt, auto simp: filter-map o-def) done

lemma find-indices-snoc [simp]:
find-indices x (ys@[y]) = find-indices x ys @ (if x = y then [length ys] else [])
by (unfold find-indices-def, auto intro!: filter-cong simp: nth-append)

lemma mem-set-find-indices [simp]:
i ∈ set (find-indices x xs) ←→ i < length xs ∧ xs!i = x
by (auto simp: find-indices-def)

lemma distinct-find-indices:
distinct (find-indices x xs)
unfolding find-indices-def by simp

context module begin

definition lincomb-list where lincomb-list c vs = sumlist (map (λi. c i ⊙_M vs ! i) [0..<length vs])

lemma lincomb-list-carrier:
assumes set vs ⊆ carrier M and c : {0..<length vs} → carrier R
shows lincomb-list c vs ∈ carrier M
by (insert assms, unfold lincomb-list-def, intro sumlist-carrier, auto intro!: smult-closed)

lemma lincomb-list-nil [simp]: lincomb-list c [] = 0_M
by (simp add: lincomb-list-def)

lemma lincomb-list-cons [simp]:
lincomb-list c (v#vs) = c 0 ⊕_M v ⊕_M lincomb-list (c o Suc) vs
by (unfold lincomb-list-def length-Cons, subst upt-conv-Cons, simp, fold map-Suc-upt, simp add: o-def)

lemma lincomb-list-eq-0:
assumes ⋀i. i < length vs ⇒ c i ⊙_M vs ! i = 0_M
shows lincomb-list c vs = 0_M

end
proof (insert assms, induct vs arbitrary:\(c\))

\[\text{case } (\text{Cons } v \text{ vs})\]

from Cons.prems[of 0] have [simp]: \(c \ 0 \odot_M v = 0_M\) by auto

from Cons.prems[of Suc - Cons.hyps] have lincomb-list \((c \circ Suc) \text{ vs} = 0_M\) by auto

then show \(?case\) by (simp add: o-def)

qed simp

definition \(\text{mk-coeff}\) where \(\text{mk-coeff} \text{ vs } c \text{ v} \equiv R.\text{sumlist} (\text{map } c \text{ (find-indices } v \text{ vs}))\)

lemma \(\text{mk-coeff-carrier}::\)

assumes \(c : \{0..<\text{length vs}\} \to \text{carrier } R\) shows \(\text{mk-coeff vs } c \text{ w} \in \text{carrier } R\)

by (insert assms, auto simp: \(\text{mk-coeff-def find-indices-def intro!} \):\(\text{R.sumlist-carrier elim!} \):\(\text{funcset-mem}\))

lemma \(\text{mk-coeff-Cons}::\)

assumes \(c : \{0..<\text{length } (v \# \text{vs})\} \to \text{carrier } R\) shows \(\text{mk-coeff} (v \# \text{vs}) \ c = (\lambda w. \text{ if } w = v \text{ then } c 0 \text{ else } 0) \oplus \text{mk-coeff vs } (c \circ Suc) \text{ w}\)

proof –

from assms have \(c \circ Suc : \{0..<\text{length vs}\} \to \text{carrier } R\) by auto

from \(\text{mk-coeff-carrier} [\text{OF this}] \text{ assms}\)

show \(?thesis\) by (auto simp add: \(\text{mk-coeff-def find-indices-Cons}\))

qed

lemma \(\text{mk-coeff-0} [\text{simp}]:\)

assumes \(v \notin \text{set vs}\)

shows \(\text{mk-coeff vs } c \text{ v} = 0\)

proof –

have \((\text{find-indices } v \text{ vs}) = []\) using assms unfolding \(\text{find-indices-def}\)

by (simp add: in-set-conv-nth)

thus \(?thesis\) unfolding \(\text{mk-coeff-def}\) by auto

qed

lemma \(\text{lincomb-list-as-lincomb}::\)

assumes vs-M: \(\text{set vs} \subseteq \text{carrier } M\) and \(c : \{0..<\text{length vs}\} \to \text{carrier } R\)

shows \(\text{lincomb-list } c \text{ vs} = \text{lincomb } (\text{mk-coeff vs } c) \text{ (set vs)}\)

proof (insert assms, induct vs arbitrary: \(c\))

\[\text{case } (\text{Cons } v \text{ vs})\]

have \(\text{mk-coeff-Suc-closed: } \text{mk-coeff vs } (c \circ Suc) \ a \in \text{carrier } R\) for \(a\)

apply (rule \(\text{mk-coeff-carrier}\))

using Cons.prems unfolding Pi-def by auto

have \(x\)-in: \(x \in \text{carrier } M\) if \(x : \text{set vs} \text{ for } x\) using Cons.prems \(x\) by auto

show \(?case\) apply (unfold \(\text{mk-coeff-Cons}\) [OF Cons.prems(2)] \(\text{lincomb-list-Cons}\))

apply (subst Cons) using Cons apply (force, force)

proof (cases \(v \in \text{set vs}\), auto simp: insert-absorb)

\[\text{case False}\]

\[\text{let } \text{if } = (\lambda v a. ((\text{if } va = v \text{ then } c 0 \text{ else } 0) \oplus \text{mk-coeff vs } (c \circ Suc) va) \odot_M va)\]

46
have mk-0; mk-coeff vs (c ◦ Suc) v = 0 using False by auto
have [simp]: (c 0 ⊕ 0) = c 0
  using Cons.prems(2) by force
have finsum-rw: (∐_Mva∈insert v (set vs). ?f va) = (?f v) ⊗_M (∐_Mva∈(set vs). ?f va)
proof (rule finsum-insert, auto simp add: False, rule smult-closed, rule R.a-closed)
  fix x
  show mk-coeff vs (c ◦ Suc) x ∈ carrier R
    using mk-coeff-Suc-closed by auto
  show c 0 ⊗_M v ∈ carrier M
    proof
      rule finsum-insert
      auto simp add: False
    qed
  show c 0 ∈ carrier R
    using Cons.prems(2) by fastforce
  show v ∈ carrier M
    using Cons.prems(1) by auto
qed

have finsum-rw2:
  (∐_Mva∈(set vs). ?f va) = (∐_Mva∈set vs. (mk-coeff vs (c ◦ Suc) va) ⊗_M va)
proof (rule finsum-cong2, auto simp add: False)
  fix i
  assume i: i ∈ set vs
  have c ◦ Suc ∈ {0..<length vs} → carrier R using Cons.prems by auto
  then have [simp]: mk-coeff vs (c ◦ Suc) i ∈ carrier R
    using mk-coeff-Suc-closed by auto
  have 0 ⊕ mk-coeff vs (c ◦ Suc) i = mk-coeff vs (c ◦ Suc) i by (rule R.l-zero, simp)
  then show (0 ⊕ mk-coeff vs (c ◦ Suc) i) ⊗_M i = mk-coeff vs (c ◦ Suc) i ⊗_M i
    by auto
  show (0 ⊕ mk-coeff vs (c ◦ Suc) i) ⊗_M i ∈ carrier M
    using Cons.prems(1) by auto
  show c 0 ⊗_M v ⊕_M (∐_M (mk-coeff vs (c ◦ Suc)) (set vs) =
    (∐_M (mk-coeff vs (c ◦ Suc)) (set vs))
  unfolding lincomb-def
  unfolding finsum-rw mk-0
  unfolding finsum-rw2 by auto
next
  case True
  let ?f = λva. ((if va = v then c 0 else 0) ⊕ mk-coeff vs (c ◦ Suc) va) ⊗_M va
  have rw: (c 0 ⊕ mk-coeff vs (c ◦ Suc) v) ⊗_M v
    = (c 0 ⊗_M v) ⊕_M (mk-coeff vs (c ◦ Suc) v) ⊗_M v
  unfolding Cons.prems(1) Cons.prems(2) atLeast0-lessThan-Suc-eq-insert-0
  qed

47
using mk-coeff-Suc-closed smult-l-distr by auto
have rw2: ((mk-coeff vs (c ∘ Suc) v) ⊙ₘ v)
  ⊕ₘ (⨁ₘva∈(set vs − {v}). If va) = (⨁ₘv∈set vs. mk-coeff vs (c ∘ Suc)
  v ⊙ₘ M v)
proof −
  have (⨁ₘva∈(set vs − {v}). If va) = (⨁ₘv∈set vs − {v}. mk-coeff vs (c ∘ Suc) v ⊙ₘ M v)
  by (rule finsum-cong2, unfold Pi-def, auto simp add: mk-coeff-Suc-closed x-in)
moreover have (⨁ₘv∈set vs. mk-coeff vs (c ∘ Suc) v ⊙ₘ M v) = ((mk-coeff
  vs (c ∘ Suc) v) ⊕ₘ M v)
  ⊕ₘ (⨁ₘv∈set vs − {v}. mk-coeff vs (c ∘ Suc) v ⊙ₘ M v)
  by (rule M.add.finprod-split, auto simp add: mk-coeff-Suc-closed True x-in)
ultimately show ?thesis by auto
qed
have lincomb (λa. (if a = v then c 0 else 0) ⊕ₘ mk-coeff vs (c ∘ Suc) a) (set vs)
  = (⨁ₘva∈set vs. If va) unfolding lincomb-def ..
also have ... = (?f v ⊕ₘ (⨁ₘva∈(set vs − {v}). If va)
proof (rule M.add.finprod-split)
  have c0-mkcoeff-in: c 0 ⊕ₘ mk-coeff vs (c ∘ Suc) v ∈ carrier R
  proof (rule R.a-closed)
    show c 0 ∈ carrier R using Cons.prems by auto
    show mk-coeff vs (c ∘ Suc) v ∈ carrier R
      using mk-coeff-Suc-closed by auto
  qed
moreover have (0 ⊕ₘ mk-coeff vs (c ∘ Suc) va) ⊙ₘ va ∈ carrier M
  if va: va ∈ carrier M for va
  by (rule smult-closed[OF - va], rule R.a-closed, auto simp add: mk-coeff-Suc-closed)
ultimately show ?f ∈ set vs ⊆ carrier M using Cons.prems(1) by auto
  show finite (set vs) by simp
  show v ∈ set vs using True by simp
qed
also have ...
  = (c 0 ⊕ₘ mk-coeff vs (c ∘ Suc) v) ⊙ₘ M v
  ⊕ₘ (⨁ₘva∈(set vs − {v}). If va) by auto
also have ...
  = (?c 0 ⊙ₘ M v) ⊕ₘ M (mk-coeff vs (c ∘ Suc) v) ⊙ₘ M v
  ⊕ₘ (⨁ₘva∈(set vs − {v}). If va) unfolding rw by simp
also have ...
  = (c 0 ⊙ₘ M v) ⊕ₘ M (((mk-coeff vs (c ∘ Suc) v) ⊙ₘ M v)
  ⊕ₘ (⨁ₘva∈(set vs − {v}). If va))
proof (rule M.a-assoc)
  show c 0 ⊙ₘ M v ∈ carrier M
    using Cons.prems(1) Cons.prems(2) by auto
  show mk-coeff vs (c ∘ Suc) v ⊙ₘ M v ∈ carrier M
    using Cons.prems(1) mk-coeff-Suc-closed by auto
  show (⨁ₘva∈set vs − {v}. ((if va = v then c 0 else 0)
    ⊕ₘ mk-coeff vs (c ∘ Suc) va) ⊙ₘ M va) ∈ carrier M
    by (rule M.add.finprod-closed) (auto simp add: mk-coeff-Suc-closed x-in)
qed
also have ...
  = c 0 ⊙ₘ M v ⊕ₘ M (⨁ₘv∈set vs. mk-coeff vs (c ∘ Suc) v ⊙ₘ M v)
unfolding rw2 ..
also have ... = c 0 ⊕ M v ⊕ M lincomb (mk-coeff vs (c ◦ Suc)) (set vs)
unfolding lincomb-def ..
finally show c 0 ⊕ M v ⊕ M lincomb (mk-coeff vs (c ◦ Suc)) (set vs)
= lincomb (λa. (if a = v then c 0 else 0) ⊕ mk-coeff vs (c ◦ Suc) a) (set vs)
.. qed
qed simp

definition span-list vs ≡ {lincomb-list c vs | c. c : {0..<length vs} → carrier R}

lemma in-span-listI:
  assumes c : {0..<length vs} → carrier R and v = lincomb-list c vs
  shows v ∈ span-list vs
  using assms by (auto simp: span-list-def)

lemma in-span-listE:
  assumes v ∈ span-list vs
  and ∏ c. c : {0..<length vs} → carrier R → v = lincomb-list c vs → thesis
  shows thesis
  using assms by (auto simp: span-list-def)

lemmas lincomb-insert2 = lincomb-insert[unfolded insert-union][symmetric]

lemma lincomb-zero:
  assumes U: U ⊆ carrier M and a: a : U → {zero R}
  shows lincomb a U = zero M
  using U a
proof (induct U rule: infinite-finite-induct)
case empty show ?case unfolding lincomb-def by auto
next
case (insert u U)
  hence a ∈ insert u U → carrier R using zero-closed by force
  thus ?case using insert by (subst lincomb-insert2; auto)
qed (auto simp: lincomb-def)

end

cell context vec-module
begin

lemma lincomb-list-as-mat-mult:
  assumes ∀ w ∈ set ws. dim-vec w = n
  shows lincomb-list c ws = mat-of-cols n ws *v vec (length ws) c (is ?l ws c =
  ?r ws c)
proof (insert assms, induct ws arbitrary: c)
case Nil
then show \( ?\)case by (auto simp: mult-mat-vec-def scalar-prod-def)

next

  case (\( \text{Cons} \ w \ w s \))

    \{ fix \( i \) assume \( i : i < n \)
       have \( \| (w\#ws) c = c \ 0 \ w + \ \text{mat-of-cols} \ n \ w s \ \text{vec} \ \text{length} \ ws \ \text{c} \circ \ \text{Suc} \)
       by (simp add: Cons o-def)
       also have \( \ldots \) \( \| \ i = \text{index-smult-vec} \)
       using Cons i index-smult-vec
       by (simp add: mat-of-cols-Cons-index-0 mat-of-cols-Cons-index-Suc o-def vec-Suc mult-mat-vec-def row-def length-Cons)
    \}

  with Cons show \( ?\)case by (intro eq-vecI, auto)

qed

lemma lincomb-union2:

  assumes \( A \subseteq \text{carrier-vec} \ n \)
       and \( BA \): \( B \subseteq A \) and \( \text{fin-A} \): \( \text{finite} \ A \)
       and \( f : f \in A \rightarrow \text{UNIV} \) shows \( \text{lincomb} \ f \ A = \text{lincomb} \ f \ (A-B) + \text{lincomb} \ f \ B \)

proof -

  have \( A - B \cup B = A \) using \( BA \) by auto
  hence \( \text{lincomb} \ f \ A = \text{lincomb} \ f \ (A-B) + \text{lincomb} \ f \ B \) by simp
  also have \( \ldots = \text{lincomb} \ f \ (A-B) + \text{lincomb} \ f \ B \)
  by (rule lincomb-union, insert assms, auto intro: finite-subset)

  finally show \( ?\)thesis .

qed

lemma dim-sumlist:

  assumes \( \forall x \in \text{set} \ xs \). \( \text{dim-vec} \ x = n \)

  shows \( \text{dim-vec} \ (M.\text{sumlist} \ xs) = n \) using assms by (induct xs, auto)

lemma sumlist-nth:

  assumes \( \forall x \in \text{set} \ xs \). \( \text{dim-vec} \ x = n \) and \( i \leq n \)

  shows \( M.\text{sumlist} \ xs \ $ i = \text{sum} \ (\lambda j \ (\text{xs} @ [j]) \ $ i) \ \{0..<\text{length} \ xs\} \)

  using assms

proof (induct xs rule: rev-induct)

  case (snoc \( a \) \( \text{xs} \))

  have [simp]: \( x \in \text{carrier-vec} \ n \) if \( x \in \text{set} \ xs \) for \( x \)
  using snoc.prems x unfolding carrier-vec-def by auto

  have [simp]: \( a \in \text{carrier-vec} \ n \)
  using snoc.prems unfolding carrier-vec-def by auto

  have hyp: \( M.\text{sumlist} \ xs \ $ i = (\sum j = 0..<\text{length} \ xs \ . \ xs @ [j]) \ $ i \)
  by (rule snoc.hyps, auto simp add: snoc.prems)

  have \( M.\text{sumlist} \ (xs @ [a]) = M.\text{sumlist} \ xs + M.\text{sumlist} \ [a] \)
  by (rule M.sumlist-append, auto simp add: snoc.prems)

  also have \( \ldots = M.\text{sumlist} \ xs + a \) by auto

  also have \( \ldots \ $ i = (M.\text{sumlist} \ xs \ $ i) + (a \ $ i) \)
  by (rule index-add-vec(1), auto simp add: snoc.prems)

  also have \( \ldots = (\sum j = 0..<\text{length} \ xs \ . \ xs @ [j]) \ $ i) + (a \ $ i) \) unfolding hyp by

50
simp
also have ... = (∑ j = 0..<length (xs @ [a]). (xs @ [a]) ! j $ i)
  by (auto, rule sum.cong, auto simp add: nth-append)
finally show ?case .
qed auto

lemma lincomb-as-lincomb-list-distinct:
  assumes s: set ws ⊆ carrier-vec n and d: distinct ws
  shows lincomb f (set ws) = lincomb-list (λi. f (ws ! i)) ws
proof (insert assms, induct ws)
  case Nil
  then show ?case by auto
next
case (Cons a ws)
  have [simp]: (∀v. v ∈ set ws ⇒ v ∈ carrier-vec n) using Cons.prems(1) by auto
  then have ws: set ws ⊆ carrier-vec n by auto
  have hyp: lincomb f (set (ws)) = lincomb-list (λi. f (ws ! i)) ws
  proof (intro Cons.hyps ws)
    show distinct ws using Cons.prems(2) by auto
  qed
  have (map (λi. f (ws ! i) $ v ws ! i) [0..<length ws]) = (map (λv. f v $ v ws))
    by (simp add: nth-map-conv)
  with ws have sumlist-rw: sumlist (map (λi. f (ws ! i) $ v ws ! i) [0..<length ws])
    = sumlist (map (λv. f v $ v ws))
    by (subst (1 2) sumlist-as-summset, auto)
  have lincomb f (set (a # ws)) = (∑v∈set (a # ws). f v $ v) unfolding lincomb-def ...
    also have ... = (∑v∈ insert a (set ws). f v $ v) by simp
    also have ... = (f a $ v a + (∑v∈ (set ws). f v $ v))
      by (rule finsum-insert, insert Cons.prems, auto)
    also have ... = f a $ v a + lincomb-list (λi. f (ws ! i)) ws using hyp lincomb-def
      by auto
    also have ... = f a $ v a + sumlist (map (λv. f v $ v ws))
      unfolding lincomb-list-def sumlist-rw by auto
    also have ... = sumlist (map (λv. f v $ v ws) (a # ws))
    proof –
      let ?a = (map (λv. f v $ v ws) [a])
      have a: a ∈ carrier-vec n using Cons.prems(1) by auto
      have f a $ v a = sumlist (map (λv. f v $ v ws) [a]) using Cons.prems(1) by auto
        hence f a $ v a + sumlist (map (λv. f v $ v ws))
          = sumlist ?a + sumlist (map (λv. f v $ v ws)) by simp
      also have ... = sumlist (?a @ (map (λv. f v $ v ws)))
        by (rule sumlist-append[symmetric], auto simp add: a)
      finally show ?thesis by auto
    qed
    also have ... = sumlist (map (λi. f (((a # ws) ! i) $ v (a # ws) ! i) [0..<length (a # ws)]))
    proof –
have u: (map (λi. f ((a ≠ ws) ! i)) (a ≠ ws)) = (map (λv. f v v) (a ≠ ws))
proof (rule nth-map-conv)
  show length [0..<length (a ≠ ws)] = length (a ≠ ws) by auto
  show ∀ i < length [0..<length (a ≠ ws)]. f ((a ≠ ws) ! (0..<length (a ≠ ws)) ! i) = f ((a ≠ ws) ! i) by (metis length [0..<length (a ≠ ws)] = length (a ≠ ws): add.left-neutral nth-upl)
qed
show ?thesis unfolding u ..
qed
also have ... = lincomb-list (λi. f ((a ≠ ws) ! i)) (a ≠ ws)
  unfolding lincomb-list-def ..
finally show ?case .
qed

end

locale idom-vec = vec-module f-ty for f-ty :: 'a :: idom itself
begin

lemma lin-dep-cols-imp-det-0':
fixes ws
defines A ≡ mat-of-cols n ws
assumes dimv-ws: ∀ w ∈ set ws. dim-vec w = n
assumes A: A ∈ carrier-mat n n and ld-cols: lin-dep (set (cols A))
shows det A = 0
proof (cases distinct ws)
case False
obtain i j where ij: i ≠ j and c: col A i = col A j and i: i < n and j: j < n
  using False A unfolding A-def
  by (metis dimv-ws distinct-cone-nth carrier-matD(2)
    col-mat-of-cols mat-of-cols-carrier(3) nth-mem carrier-vecI)
show ?thesis by (rule det-identical-columns[OF A ij i j c])
next
case True
have dl[simp]: ∀ x. x ∈ set ws ⇒ x ∈ carrier-vec n using dimv-ws by auto
obtain A' f' v where f'-in: f' ∈ A' → UNIV
  and le-f': lincomb f' A' = 0_v n and f'-v: f' v ≠ 0
  and v-A': v ∈ A' and A'-in-rows: A' ⊆ set (cols A)
  using ld-cols unfolding lin-dep-def by auto
define f where f ≡ λx. if x ∈ A' then 0 else f' x
have f-in: f ∈ (set (cols A)) → UNIV using f'-in by auto
have A'-in-carrier: A' ⊆ carrier-vec n
  by (metis (no-types) A'-in-rows A-def cols-dim carrier-matD(1) mat-of-cols-carrier(1) subset-trans)
have lc-f: lincomb f (set (cols A)) = 0_v n
proof –
have l1: \( \text{lincomb } f \ (\text{set } (\text{cols } A) - A') = 0 \_ \_ n \)
  by (rule lincomb-zero, auto simp add: f-def, insert A cols-dim, blast)
have l2: \( \text{lincomb } f A' = 0 \_ \_ n \) using \( \text{le-f' unfolding f-def using } A'\text{-in-carrier} \) by auto
have lincomb f (set (cols A)) = lincomb f (set (cols A) - A') + lincomb f A'
proof (rule lincomb-union2 )
  show set (cols A) \( \subseteq \) carrier-vec n
    using A cols-dim by blast
  show A' \( \subseteq \) set (cols A)
    using A'-in-rows by blast
  show finite (set (cols A)) by auto
  show \( f \in \text{set } (\text{cols } A) \rightarrow \text{UNIV } by \text{auto} \)
qed
also have ... = 0 \_ \_ n using l1 l2 by auto
finally show \( ?\text{thesis} \).
qed
have v-in: \( v \in (\text{set } (\text{cols } A)) \) using v-A' A'-in-rows by auto
have f v: \( f v \neq 0 \) using f'-v v-A' unfolding f-def by auto
let \( ?c = (\lambda i. f \ (ws ! i)) \)
have lincomb f (set ws) = lincomb-list \( ?c \ ws \)
  by (rule lincomb-as-lincomb-list-distinct[OF - True], auto)
have \( \exists v. \ v \in \text{carrier-vec } n \land v \neq 0 \_ \_ n \land A *_v v = 0 \_ \_ n \)
proof (rule exI[of - vec (length ws) \( ?c \)], rule conjI)
  show vec (length ws) \( ?c \in \text{carrier-vec } n \) using A A-def by auto
  have vec-not0: vec (length ws) \( ?c \neq 0 \_ \_ n \)
    proof 
    obtain i where ws-i: \( (ws ! i) = v \) and i: \( i < \text{length } ws \) using v-in unfolding A-def
      by (metis d1 cols-mat-of-cols in-set-conv-nth subset-eq)
    have vec (length ws) \( ?c \$ i = ?c i \) by (rule index-vec[OF i])
    also have ... = f v using ws-i by simp
    also have ... \neq 0 using f v by simp
    finally show \( ?\text{thesis} \)
      using A A-def i by fastforce
qed
have A *_v vec (length ws) \( ?c = \text{mat-of-cols } n \ ws *_v vec (\text{length } ws) \ ?c \) unfolding A-def ..
also have ... = lincomb-list \( ?c \ ws \) by (rule lincomb-list-as-mat-mult[symmetric, OF dimv-qs])
also have ... = lincomb f (set ws)
  by (rule lincomb-as-lincomb-list-distinct[symmetric, OF - True], auto)
also have ... \neq 0 \_ \_ n
  using \( \text{le-f unfolding A-def using } A \) by (simp add: subset-code(1))
finally show vec (length ws) \( (\lambda i. f \ (ws ! i)) \neq 0 \_ \_ n \land A *_v vec (\text{length } ws) \ (\lambda i. f \ (ws ! i)) = 0 \_ \_ n \)
  using vec-not0 by fast
qed
thus \( ?\text{thesis unfolding det-0-iff-vec-prod-zero[OF A]} \).
qed

lemma lin-dep-cols-imp-det-0:
  assumes A: A ∈ carrier-mat n n and ld: lin-dep (set (cols A))
  shows det A = 0
proof
  have col-rw: (cols (mat-of-cols n (cols A))) = cols A
    using A by auto
  have m: mat-of-cols n (cols A) = A using A by auto
  show ?thesis
    by (rule A lin-dep-cols-imp-det-0[of cols A, unfolded col-rw, unfolded m, OF - A ld])
      (metis A cols-dim carrier-matD(1) subsetCE carrier-vecD)
qed

corollary lin-dep-rows-imp-det-0:
  assumes A: A ∈ carrier-mat n n and ld: lin-dep (set (rows A))
  shows det A = 0
  by (subst det-transpose[OF A, symmetric], rule lin-dep-cols-imp-det-0, auto simp add: ld A)

lemma det-not-0-imp-lin-indpt-rows:
  assumes A: A ∈ carrier-mat n n and det: det A ≠ 0
  shows lin-indpt (set (rows A))
  using det-not-0-imp-lin-indpt-rows OF A det by auto

lemma upper-triangular-imp-lin-indpt-rows:
  assumes A: A ∈ carrier-mat n n
      and tri: upper-triangular A
      and diag: 0 /∈ set (diag-mat A)
  shows lin-indpt (set (rows A))

lemma lincomb-as-lincomb-list:
  fixes ws f
  assumes s: set ws ⊆ carrier-vec n
  shows lincomb f (set ws) = lincomb-list (%i. if ∃ j<i. ws!i = ws!j then 0 else f (ws ! i)) ws
    using assms
proof (induct ws rule: rev-induct)
case (snoc a ws)
  let ?f = %i. if ∃ j<i. ws ! i = ws ! j then 0 else f (ws ! i)
  let ?g = %i. (if ∃ j<i. (ws @ [a]) ! i = (ws @ [a]) ! j then 0 else f ((ws @ [a]) ! i))
    · v (ws @ [a]) ! i
  let ?g2 = (%i. (if ∃ j<i. ws ! i = ws ! j then 0 else f (ws ! i)) · v ws ! i)
  have [simp]: a v ∈ set ws ⟷ v ∈ carrier-vec n using snoc.prems(1) by auto
  then have ws: set ws ⊆ carrier-vec n by auto
have \( \text{hypo} \): \( \text{lincomb} \ f \ (\text{set} \ \text{ws}) = \text{lincomb-list} \ ?f \ \text{ws} \) 
by (intro \text{sno}\text{c.hyps ws})

\text{show} \ ?\text{case}

\text{proof} \ \text{(cases} \ a@\text{set} \ \text{ws})

\text{case} \ True

have \( g\text{-length} : ?g \ (\text{length} \ \text{ws}) = 0@n \ \text{using} \ True \)
by (auto, \text{metis} \ \text{in-set-conv-nth} \ \text{nth-append})

have \( (\text{map} \ ?g \ [0..<\text{length} \ (\text{ws} \ @ \ [a])]) = (\text{map} \ ?g \ [0..<\text{length} \ \text{ws}]) @ [?g \ (\text{length} \ \text{ws})] \)
by auto

also have \( ... = \text{M.sumlist} \ (\text{map} \ ?g2 \ [0..<\text{length} \ \text{ws}]) = \text{M.sumlist} \ (\text{map} \ ?g \ [0..<\text{length} \ \text{ws}]) @ [?g \ [0..<\text{length} \ \text{ws}]) \)
by (rule \text{arg-cong[of - - M.sumlist]}, \text{rule} \ \text{nth-map-conv}, \text{auto} \ \text{simp add: nth-append})

also have \( ... = \text{M.sumlist} \ (\text{map} \ ?g \ [0..<\text{length} \ \text{ws}]) @ [?g \ n] \)
by (\text{metis} \ M.r-zero \ \text{calculation} \ \text{hypo} \ \text{lincomb-closed} \ \text{lincomb-list-def} \ \text{ws})

also have \( ... = \text{M.sumlist} \ (\text{map} \ ?g \ [0..<\text{length} \ \text{ws}]) @ [?g \ n] \)
by (rule \text{M.sumlist-snoc[symmetric]}, \text{auto} \ \text{simp add: nth-append})

finally have \( \text{map-rw}: (\text{map} \ ?g \ [0..<\text{length} \ (\text{ws} \ @ \ [a])]) = (\text{map} \ ?g \ [0..<\text{length} \ \text{ws}]) @ [?g \ [0..<\text{length} \ \text{ws}]) @ [0@v \ n]) \)

have \( \text{lincomb} \ f \ (\text{set} \ \text{ws} @ [a]) = \text{lincomb} \ f \ (\text{set} \ \text{ws}) \ \text{using} \ True \ \text{unfolding} \ \text{lincomb-def})
by (\text{simp add: insert-absorb})

thus \( \text{thesis} \)

\text{unfolding} \ \text{hypo} \ \text{lincomb-list-def} \ \text{map-rw} \ \text{summlist-rw}
by auto

\text{next}

\text{case} \ False

have \( g\text{-length} : ?g \ (\text{length} \ \text{ws}) = f@a \ a \ \text{using} \ False \ by \ (\text{auto} \ \text{simp add: nth-append})

have \( (\text{map} \ ?g \ [0..<\text{length} \ (\text{ws} @ [a])]) = (\text{map} \ ?g \ [0..<\text{length} \ \text{ws}]) @ [?g \ (\text{length} \ \text{ws})] \)
by auto

also have \( ... = (\text{map} \ ?g \ [0..<\text{length} \ \text{ws}]) @ [(f \ a, v) \ a@v \ a] \ \text{using} \ False \ by \ \text{simp}

finally have \( \text{map-rw}: (\text{map} \ ?g \ [0..<\text{length} \ (\text{ws} @ [a])]) = (\text{map} \ ?g \ [0..<\text{length} \ \text{ws}]) @ [(f \ a, v) \ a@v \ a] \)

have \( \text{summlist-rw}: \text{M.sumlist} \ (\text{map} \ ?g2 \ [0..<\text{length} \ \text{ws}]) = \text{M.sumlist} \ (\text{map} \ ?g \ [0..<\text{length} \ \text{ws}]) @ [0@v \ n]) \)
by (rule \text{arg-cong[of - - M.sumlist]}, \text{rule} \ \text{nth-map-conv}, \text{auto} \ \text{simp add: nth-append})

have \( \text{lincomb} \ f \ (\text{set} \ \text{ws} @ [a]) = \text{lincomb} \ f \ (\text{set} \ \text{a} @ [a]) \) by auto
also have \( ... = (\bigoplus v \in \text{set} \ (a \ # \ \text{ws}), f@v \ a \ v) \ \text{unfolding} \ \text{lincomb-def} \ .
also have \( ... = (\bigoplus v \in \text{insert} \ a \ (\text{set} \ \text{ws}), f@v \ a \ v) \ \text{by} \ \text{simp}
also have \( ... = (f \ a, v \ a) + (\bigoplus v \in \text{set} \ \text{ws}, f@v \ a \ v) \)

\text{proof} \ (\text{rule} \ \text{finsum-insert})

\text{show} \ \text{finite} \ (\text{set} \ \text{ws}) \ by \ \text{auto}
show a \notin \text{set ws} \text{ using False by auto}
show (\lambda v. f v \cdot_v v) \in \text{set ws \rightarrow carrier-vec n}
  \text{ using snoc.prems(1) by auto}
show f a \cdot_v a \in carrier-vec n \text{ using snoc.prems by auto}
qed
also have ... = (f a \cdot_v a) + \text{lincomb f (set ws) unfolding lincomb-def ..}
also have ... = (f a \cdot_v a) + \text{lincomb-list ?f ws using hyp by auto}
also have ... = lincomb-list ?f ws + (f a \cdot_v a)
  \text{ using M.add.m-comm lincomb-list-carrier snoc.prems by auto}
also have ... = lincomb-list (\lambda i. \exists j<i. (\text{ws @ [a]} ! i)
  = (\text{ws @ [a]} ! j \text{ then } 0 \text{ else } f ((\text{ws @ [a]} ! i)) (\text{ws @ [a]}))
proof (unfold lincomb-list-def map-rw summlist-rw, rule M.sumlist-snoc[symmetric])
  show set (\map \gamma [\theta..<\text{length ws}]) \subseteq carrier-vec n \text{ using snoc.prems}
  by (auto simp add: nth-append)
show f a \cdot_v a \in carrier-vec n
  \text{ using snoc.prems by auto}
qed
finally show \text{thesis} .
qed
qed auto

lemma span-list-as-span:
assumes set vs \subseteq \text{carrier-vec n}
shows \text{span-list vs = span (set vs)}
  \text{ using assms}
proof (auto simp: span-list-def span-def)
fix f show \exists a. lincomb-list f vs = lincomb a A \land \text{finite A \land A \subseteq set vs}
  \text{ using assms lincomb-list-as-lincomb by auto}
next
fix f::'a vec \Rightarrow 'a and A assume fA: \text{finite A and A: A \subseteq set vs}
have [simp]: x \in carrier-vec n if x: x \in A for x using A x assms by auto
have [simp]: v \in carrier-vec n if v: v \in set vs for v using assms v by auto
have set-vs-Un: ((set vs) - A) \cup A = set vs using A by auto
let \gamma f = (\lambda x. if x \in (set vs) - A then 0 else f x)
have f0: (\bigoplus v \in (set vs) - A. \gamma f v \cdot_v v) = 0_v \text{ n by (rule M.finsum-all0, auto)}

have lincomb f A = lincomb \gamma f A
  \text{ by (auto simp add: lincomb-def intro!: finsum-cong2)}
also have ... = (\bigoplus v \in (set vs) - A. \gamma f v \cdot_v v) + (\bigoplus v \in A. \gamma f v \cdot_v v)
  \text{ unfolding f0 lincomb-def by auto}
also have ... = lincomb \gamma ((set vs) - A) \cup A
  \text{ unfolding lincomb-def}
  \text{ by (rule M.finsum-Un-disjoint[symmetric], auto simp add: fA)}
also have ... = lincomb \gamma (set vs) using set-vs-Un by auto
finally have lincomb f A = lincomb \gamma (set vs) .
with lincomb-as-lincomb-list[OF assms]
show \exists c. lincomb f A = lincomb-list c vs by auto
qed
lemma in-spanI[intro]:
  assumes \( v = \text{lincomb} \ a \ A \text{ finite} \ A \subseteq W \)
  shows \( v \in \text{span} \ W \)
unfolding span-def using assms by auto
lemma in-spanE:
  assumes \( v \in \text{span} \ W \)
  shows \( \exists \ a \ A. \ v = \text{lincomb} \ a \ A \land \text{finite} \ A \land A \subseteq W \)
using assms unfolding span-def by auto

declare in-own-span[intro]

lemma smult-in-span:
  assumes \( W \subseteq \text{carrier-vec} \ n \) and \( \text{insp}: x \in \text{span} \ W \)
  shows \( c \cdot v x \in \text{span} \ W \)
proof
  from in-spanE[OF insp] obtain \( a A \) where \( a: x = \text{lincomb} \ a A \text{ finite} A \subseteq W \)
  by blast
  have \( c \cdot v x = \text{lincomb} \ (\lambda x. c * a x) A \) using \( a(1) \) unfolding lincomb-def a
  apply (subst finsum-smult) using assms a by (auto simp: smult-smult-assoc)
  thus \( c \cdot v x \in \text{span} \ W \) using \( a(2,3) \) by auto
qed

lemma span-subsetI: assumes \( \text{us} \subseteq \text{span} \text{us} \)
  shows \( \text{span} \text{us} \subseteq \text{span} \text{ws} \)
proof
  simp add: assms(1) span-is-submodule span-is-subset subsetI ws
end

context vec-space
begin
sublocale idom-vec.

lemma sumlist-in-span:
  assumes \( \text{W} \subseteq \text{carrier-vec} \ n \)
  shows \( (\forall x. x \in \text{set} \text{xs} \Rightarrow x \in \text{span} \text{W}) \Rightarrow \text{sumlist} \text{xs} \in \text{span} \text{W} \)
proof (induct xs)
  case Nil
  thus \?case using \( \text{W} \) by force
next
  case (Cons x xs)
  from span-is-subset2[OF \( \text{W} \)] Cons(2) have \( \text{x}: x \in \text{carrier-vec} \ n \) set \( \text{x} \subseteq \text{carrier-vec} \ n \) by auto
  from span-addI[OF \( \text{W} \) Cons(2)[of \( \text{x} \)] Cons(1)[OF Cons(2)]]
  have \( x + \text{sumlist} \text{xs} \in \text{span} \text{W} \) by auto
  also have \( x + \text{sumlist} \text{xs} = \text{sumlist} (\{x\} \odot \text{xs}) \)
  by (subst sumlist-append, insert \( \text{x} \), auto)
  finally show \?case by simp
qed

lemma span-span[simp]:
assumes \( W \subseteq \text{carrier-vec } n \)
shows \( \text{span } (\text{span } W) = \text{span } W \)
proof (standard, standard, goal-cases)
case (1 x) with in-spanE obtain a A where a: \( x = \text{lincomb } a \) finite A A \( \subseteq \) span W by blast
from a(3) assms have A\( C : A \subseteq \text{carrier-vec } n \) by auto
show \( ?\text{case unfolding } a(1)[\text{unfolded lincomb-def}] \)
case 1
then show ?case using span-zero by auto
next
case (2 x F)
{ assume F: insert x F \( \subseteq \) span W
  hence a x \( \cdot \), x \( \in \) span W by (intro \text{smult-in-span}[OF assms], auto)
  hence a x \( \cdot \), x + (\( \bigoplus \) \( v \in F. a \ v \cdot v \) \( \in \) span W
    using span-add1 F 2 assms by auto
  hence (\( \bigoplus \) \( v \in \text{insert } x \) F. a \( \cdot \), v \( \in \) span W
    apply (subst M, finsum-insert[OF 2(1,2)]) using F assms by auto
  }
then show ?case by auto
qed
next
case 2
show ?case using assms by (intro in-own-span, auto)
qed

lemma upper-triangular-imp-basis:
assumes A: A \( \in \) carrier-mat n n
  and tri: upper-triangular A
  and diag: 0 \( \notin \) set (diag-mat A)
shows basis (set (rows A))
using upper-triangular-imp-distinct[OF assms]
using upper-triangular-imp-lin-indpt-rows[OF assms] A
by (auto intro: dim-li-is-basis simp: distinct-card dim-is-n set-rows-carrier)
end

lemma (in zero-hom) hom-upper-triangular:
A \( \in \) carrier-mat n n \( \implies \) upper-triangular A \( \implies \) upper-triangular (map-mat hom A)
by (auto simp: upper-triangular-def)
end
4 Norms

In this theory we provide the basic definitions and properties of several norms of vectors and polynomials.

theory Norms
imports Polynomial Adhoc-Overloading
Jordan-Normal-Form.Conjugate
Algebraic-Numbers.Resultant
Missing-Lemmas

begin

4.1 $L^\infty$ Norms

consts linf-norm :: 'a ⇒ 'b (∥(-)∥∞)

definition linf-norm-vec where linf-norm-vec v ≡ max-list (map abs (list-of-vec v) @ [0])

adhoc-overloading linf-norm linf-norm-vec

definition linf-norm-poly where linf-norm-poly f ≡ max-list (map abs (coeffs f) @ [0])

adhoc-overloading linf-norm linf-norm-poly

lemma linf-norm-vec [simp]: ∥vec n f∥∞ = max-list (map abs (f @ [0..<n] @ [0])) by (simp add: linf-norm-vec-def)

lemma linf-norm-vec-vCons [simp]: ∥vCons a v∥∞ = max |a| ∥v∥∞ by (auto simp: linf-norm-vec-def max-list-Cons)

lemma linf-norm-vec-0 [simp]: ∥vec 0 f∥∞ = 0 by (simp add: linf-norm-vec-def)

lemma linf-norm-zero-vec [simp]: ∥0 v n :: 'a :: ordered-ab-group-add-abs vec∥∞ = 0 by (induct n, simp add: zero-vec-def, auto simp: zero-vec-Suc)

lemma linf-norm-vec-ge-0 [intro!]:
  fixes v :: 'a :: ordered-ab-group-add-abs vec
  shows ∥v∥∞ ≥ 0 by (induct v, auto simp: max-def)

lemma linf-norm-vec-eq-0 [simp]:
  fixes v :: 'a :: ordered-ab-group-add-abs vec
  assumes v ∈ carrier-vec n
  shows ∥v∥∞ = 0 ⟷ v = 0 v n
  by (insert assms, induct rule: carrier-vec-induct, auto simp: zero-vec-Suc max-def)

lemma linf-norm-vec-greater-0 [simp]:
  fixes v :: 'a :: ordered-ab-group-add-abs vec
  assumes v ∈ carrier-vec n

59
\[ \|v\|_{\infty} > 0 \iff v \neq 0 \]

**Show** by (insert assms, induct rule: carrier-vec-induct, auto simp: zero-vec-Suc max-def)

**Lemma** *linf-norm-poly-0* [simp]: \( \|0\|_{\infty} = 0 \)

*by (simp add: linf-norm-poly-def)*

**Lemma** *linf-norm-pCons* [simp]:
- **fixes** \( p :: 'a ::= \text{ordered-ab-group-add-vec} \)
- **shows** \( \|pCons a p\|_{\infty} = \max |a| \|p\|_{\infty} \)

*by (cases \( p = 0 \), cases \( a = 0 \), auto simp: linf-norm-poly-def max-list-Cons)*

**Lemma** *linf-norm-poly-ge-0* [intro!]:
- **fixes** \( f :: 'a ::= \text{ordered-ab-group-add-vec} \)
- **shows** \( \|f\|_{\infty} \geq 0 \)

*by (induct \( f \), auto simp: max-def)*

**Lemma** *linf-norm-poly-eq-0* [simp]:
- **fixes** \( f :: 'a ::= \text{ordered-ab-group-add-vec} \)
- **shows** \( \|f\|_{\infty} = 0 \iff f = 0 \)

*by (induct \( f \), auto simp: max-def)*

**Lemma** *linf-norm-poly-greater-0* [simp]:
- **fixes** \( f :: 'a ::= \text{ordered-ab-group-add-vec} \)
- **shows** \( \|f\|_{\infty} > 0 \iff f \neq 0 \)

*by (induct \( f \), auto simp: max-def)*

### 4.2 Square Norms

**Consts** \( \text{sq-norm} :: 'a \Rightarrow 'b (\|\cdot\|^2) \)

**Abbreviation** \( \text{sq-norm-conjugate} x \equiv x \ast \text{conjugate} x \)

**Adhoc-Overloading** \( \text{sq-norm sq-norm-conjugate} \)

#### 4.2.1 Square norms for vectors

We prefer \text{sum_list} over \text{sum} because it is not essentially dependent on commutativity, and easier for proving.

**Definition** \( \text{sq-norm-vec} v \equiv \sum x \leftarrow \text{list-of-vec} v. \|x\|^2 \)

**Adhoc-Overloading** \( \text{sq-norm sq-norm-vec} \)

**Lemma** *sq-norm-vec-vCons*[simp]: \( \|vCons a v\|^2 = \|a\|^2 + \|v\|^2 \)

*by (simp add: sq-norm-vec-def)*

**Lemma** *sq-norm-vec-0*[simp]: \( \|\text{vec } 0\|^2 = 0 \)

*by (simp add: sq-norm-vec-def)*

**Lemma** *sq-norm-vec-as-cscalar-prod*:
- **fixes** \( v :: 'a ::= \text{conjugatable-ring vec} \)
- **shows** \( \|v\|^2 = v \cdot c v \)
by (induct v, simp-all add: sq-norm-vec-def)

lemma sq-norm-zero-vec[simp]: \( \|0\_v\| = 0 \)
  by (simp add: sq-norm-vec-as-cscalar-prod)

lemmas sq-norm-vec-ge-0 [intro!] = conjugate-square-ge-0-vec[folded sq-norm-vec-as-cscalar-prod]

lemmas sq-norm-vec-eq-0 [simp] = conjugate-square-eq-0-vec[folded sq-norm-vec-as-cscalar-prod]

lemmas sq-norm-vec-greater-0 [simp] = conjugate-square-greater-0-vec[folded sq-norm-vec-as-cscalar-prod]

4.2.2 Square norm for polynomials

definition sq-norm-poly where sq-norm-poly p ≡ \( \sum a\_\langle\rangle \_\langle\rangle a\) \( \|a\|^2 \)

adhoc-overloading sq-norm sq-norm-poly

lemma sq-norm-poly-0 [simp]: \( \|0\_p\|^2 = 0 \)
  by (auto simp: sq-norm-poly-def)

lemma sq-norm-poly-pCons [simp]:
  fixes a :: 'a :: conjugatable-ring
  shows \( \|pCons a p\|^2 = \|a\|^2 + \|p\|^2 \)
  by (cases p = 0; cases a = 0, auto simp: sq-norm-poly-def)

lemma sq-norm-poly-ge-0 [intro!]:
  fixes p :: 'a :: conjugatable-ordered-ring poly
  shows \( \|p\|^2 \geq 0 \)
  by (unfold sq-norm-poly-def, rule sum-list-nonneg, auto intro!: conjugate-square-positive)

lemma sq-norm-poly-eq-0 [simp]:
  fixes p :: 'a :: {conjugatable-ordered-ring, ring-no-zero-divisors} poly
  shows \( \|p\|^2 = 0 \longleftrightarrow p = 0 \)
  proof (induct p)
    case IH: (pCons a p)
    show ?case
      proof (cases a = 0)
        case True
        with IH show ?thesis by simp
      next
        case False
        then have \( \|a\|^2 + \|p\|^2 > 0 \) by (intro add-pos-nonneg, auto)
        then show ?thesis by auto
      qed
    qed simp

lemma sq-norm-poly-pos [simp]:
  fixes p :: 'a :: {conjugatable-ordered-ring, ring-no-zero-divisors} poly
  shows \( \|p\|^2 > 0 \longleftrightarrow p \neq 0 \)
by (auto simp: less-le)

**Lemma sq-norm-vec-of-poly [simp]:**

*fixes* \( p :: 'a :: \text{conjugatable-ring poly} \)*

*shows* \( \| \text{vec-of-poly} \ p \|_2^2 = \| \ p \|_2^2 \)*

*apply* (unfold sq-norm-poly-def sq-norm-vec-def)

*apply* (fold sum-mset-sum-list)

*apply* auto.

**Lemma sq-norm-poly-of-vec [simp]:**

*fixes* \( v :: 'a :: \text{conjugatable-ring vec} \)*

*shows* \( \| \text{poly-of-vec} \ v \|_2^2 = \| \ v \|_2^2 \)*

*apply* (unfold sq-norm-poly-def sq-norm-vec-def coeffs-poly-of-vec)

*apply* (fold rev-map)

*apply* (fold sum-mset-sum-list)

*apply* (unfold mset-rev)

*apply* (unfold sum-mset-sum-list)

*by* (auto intro: sum-list-map-dropWhile0)

### 4.3 Relating Norms

A class where ordering around 0 is linear.

**Abbreviation (in ordered-semiring) is-real where is-real \( a \equiv a < 0 \lor a = 0 \lor 0 < a \)**

**Class semiring-real-line = ordered-semiring-strict + ordered-semiring-0 +**

*assumes* \( \text{add-pos-neg-is-real: } a > 0 \implies b < 0 \implies \text{is-real} \ (a + b) \)*

*and* \( \text{mult-neg-neg: } a < 0 \implies b < 0 \implies 0 < a * b \)*

*and* \( \text{pos-pos-linear: } 0 < a \implies 0 < b \implies a < b \lor a = b \lor b < a \)*

*and* \( \text{neg-neg-linear: } a < 0 \implies b < 0 \implies a < b \lor a = b \lor b < a \)**

*begin*

**Lemma add-neg-pos-is-real: \( a < 0 \implies b > 0 \implies \text{is-real} \ (a + b) \)**

*using* \( \text{add-pos-neg-is-real[of b a]} \ by \ (simp add: ac-simps) \)

**Lemma nonneg-linorder-cases [consumes 2, case-names less eq greater]:**

*assumes* \( 0 \leq a \ and \ 0 \leq b \)*

*and* \( a < b \implies \text{thesis} \ a = b \implies \text{thesis} \ a < b \implies \text{thesis} \)*

*shows* \( \text{thesis} \)

*using* \( \text{assms pos-pos-linear} \ by \ (auto simp: le-less) \)

**Lemma nonpos-linorder-cases [consumes 2, case-names less eq greater]:**

*assumes* \( a \leq 0 \ b \leq 0 \)*

*and* \( a < b \implies \text{thesis} \ a = b \implies \text{thesis} \ a < b \implies \text{thesis} \)*

*shows* \( \text{thesis} \)

*using* \( \text{assms neg-neg-linear} \ by \ (auto simp: le-less) \)

**Lemma real-linear:**

*assumes* \( \text{is-real} \ a \ and \ \text{is-real} \ b \)*

*shows* \( a < b \lor a = b \lor b < a \)

62
using pos-pos-linear neg-neg-linear assms by (auto dest: less-trans[of - 0])

lemma real-linorder-cases [consumes 2, case-names less eq greater]:
  assumes real: is-real a is-real b
  and cases: a < b \implies thesis a = b \implies thesis b < a \implies thesis
  shows thesis
  using real-linear[OF real] cases by auto

lemma assumes a: is-real a and b: is-real b
  shows real-add-le-cancel-left-pos: c + a ≤ c + b \iff a ≤ b
  and real-add-less-cancel-left-pos: c + a < c + b \iff a < b
  and real-add-le-cancel-right-pos: a + c ≤ b + c \iff a ≤ b
  and real-add-less-cancel-right-pos: a + c < b + c \iff a < b
  using add-strict-left-mono[of b a c] add-strict-left-mono[of a b c]
  using add-strict-right-mono[of b a c] add-strict-right-mono[of a b c]
  by (atomize(full), cases rule: real-linorder-cases[OF a b], auto)

lemma assumes a: is-real a and b: is-real b and c: 0 < c
  shows real-mult-le-cancel-left-pos: c * a ≤ c * b \iff a ≤ b
  and real-mult-less-cancel-left-pos: c * a < c * b \iff a < b
  and real-mult-le-cancel-right-pos: a * c ≤ b * c \iff a ≤ b
  and real-mult-less-cancel-right-pos: a * c < b * c \iff a < b
  using mult-strict-left-mono[of b a c] mult-strict-left-mono[of a b c] c
  using mult-strict-right-mono[of b a c] mult-strict-right-mono[of a b c] c
  by (atomize(full), cases rule: real-linorder-cases[OF a b], auto)

lemma assumes a: is-real a and b: is-real b
  shows not-le-real: ¬ a ≥ b \iff a < b
  and not-less-real: ¬ a > b \iff a ≤ b
  by (atomize(full), cases rule: real-linorder-cases[OF a b], auto simp: less-imp-le)

lemma real-mult-eq-0-iff:
  assumes a: is-real a and b: is-real b
  shows a * b = 0 \iff a = 0 \lor b = 0
  proof –
    { assume l: a * b = 0 and a ≠ 0 and b ≠ 0
      with a b have a < 0 \lor 0 < a and b < 0 \lor 0 < b by auto
        by (auto simp:l)
    } then show ?thesis by auto
  qed

end

lemma real-pos-mult-max:
fixes \(a\) \(b\) \(c\) :: 'a :: semiring-real-line
assumes \(c > 0\) and \(a\) is-real \(a\) and \(b\) is-real \(b\)
shows \(c \cdot \max a b = \max (c \cdot a) (c \cdot b)\)
by (rule hom-max, simp add: real-mult-le-cancel-left-pos[OF \(a\) \(b\) \(c\)])

class ring-abs-real-line = ordered-ring-abs + semiring-real-line

class semiring-1-real-line = semiring-real-line + monoid-mult + zero-less-one
begin
subclass ordered-semiring-1 by (unfold-locales, auto)

lemma power-both-mono: \(1 \leq a \Longrightarrow m \leq n \Longrightarrow a \leq b \Longrightarrow a \cdot m \leq b \cdot n\)
using power-monot[of \(a\) \(b\) \(n\)] power-increasing[of \(m\) \(n\) \(a\)]
by (auto simp: order.trans[OF zero-le-one])

lemma power-pos:
assumes \(a > 0\) shows \(0 < a \cdot n\)
by (induct \(n\), insert mult-strict-mono[OF \(a\) \(0\)] \(a\) \(0\), auto)

lemma power-neg:
assumes \(a < 0\) shows \(odd n \Longrightarrow a \cdot n < 0\) and \(even n \Longrightarrow a \cdot n > 0\)
by (atomize(full), induct \(n\), insert \(a\) \(0\), auto simp add: mult-pos-neg2 mult-neg-neg)

lemma power-ge-0-iff:
assumes \(a\) is-real \(a\)
shows \(0 \leq a \cdot n \iff 0 \leq a \vee even n\)
using a proof (elim disjE)
assume \(a < 0\)
with power-neg[OF this, of \(n\)] show \(?thesis\) by (cases even \(n\), auto)
next
assume \(0 < a\)
with power-pos[OF this] show \(?thesis\) by auto
next
assume \(a = 0\)
then show \(?thesis\) by (auto simp:power-0-left)
qed

lemma nonneg-power-less:
assumes \(0 \leq a\) and \(0 \leq b\) shows \(a \cdot n < b \cdot n \iff n > 0 \wedge a < b\)
proof (insert assms, induct \(n\) arbitrary: \(a\) \(b\))
case \(0\)
then show \(?case\) by auto
next
case \(Suc n\)
note \(a = :0 \leq a\)
note \(b = :0 \leq b\)
show \(?case\)
proof (cases \(n > 0\))
case True
from a b show thesis
proof (cases rule: nonneg-linorder-cases)
case less
  then show thesis by (auto simp: Suc.hyps[OF a b] True intro!: mult-strict-mono' a b zero-le-power)
next
case eq
  then show thesis by simp
next
case greater
  with Suc.hyps[OF b a] True have b ^ n < a ^ n by auto
  with mult-strict-mono[OF greater this] b greater
  show thesis by auto
qed
qed auto

lemma power-strict-mono:
  shows a < b =⇒ 0 ≤ a =⇒ 0 < n =⇒ a ^ n < b ^ n
  by (subst nonneg-power-less, auto)

lemma nonneg-power-le:
  assumes 0 ≤ a and 0 ≤ b shows a ^ n ≤ b ^ n =⇒ n = 0 ∨ a ≤ b
using assms proof (cases rule: nonneg-linorder-cases)
case less
  with power-strict-mono[OF this, of n] assms show thesis by (cases n, auto)
next
case eq
  then show thesis by auto
next
case greater
  with power-strict-mono[OF this, of n] assms show thesis by (cases n, auto)
qed

end

subclass (in linordered-idom) semiring-1-real-line
  apply unfold-locales
  by (auto simp: mult-strict-left-mono mult-strict-right-mono mult-neg-neg)

class ring-1-abs-real-line = ring-abs-real-line + semiring-1-real-line
begin
subclass ring-1..

lemma abs-cases:
  assumes a = 0 =⇒ thesis and |a| > 0 =⇒ thesis shows thesis
  using assms by auto
lemma abs-linorder-cases[case-names less eq greater]:
assumes \(|a| < |b| \Rightarrow \text{thesis}\) and \(|a| = |b| \Rightarrow \text{thesis}\) and \(|b| < |a| \Rightarrow \text{thesis}\)
shows thesis
apply (cases rule: nonneg-linorder-cases[of \(|a| \ |b|\)])
using assms by auto

lemma [simp]:
shows \(\neg |a| \geq |b| \iff |a| < |b|\)
and \(\neg |a| > |b| \iff |a| \leq |b|\)
by (atomize(full), cases a b rule: abs-linorder-cases, auto simp: less_imp_le)

lemma abs-power-less [simp]: \(|a| \times|b| \iff n > 0 \land |a| < |b|\)
by (subst nonneg-power-less, auto)

lemma abs-power-le [simp]: \(|a| \times|b| \iff n = 0 \lor |a| \leq |b|\)
by (subst nonneg-power-le, auto)

lemma abs-power-pos [simp]: \(|a| > 0 \iff a \neq 0 \land n = 0\)
using power-pos[of \(|a|\)] by (cases n, auto)

lemma abs-power-nonneg [intro!]: \(|a| \geq 0\) by auto

lemma abs-power-eq-0 [simp]: \(|a| \times|b| \iff n = 0 \land n \neq 0\)
apply (induct n, force)
apply (unfold power-Suc)
apply (subst real-mult-eq-0_iff, auto).

end

instance nat :: semiring_1_real_line by (intro_classes, auto)
instance int :: ring_1_abs_real_line..

lemma vec-index-vec-of-list [simp]: vec-of-list xs $ i = xs ! i
by transfer (auto simp: mk_vec_def undef_vec_def dest: empty_nth)

lemma vec-of-list-append: vec-of-list (xs @ ys) = vec-of-list xs @ vec-of-list ys
by (auto simp: nth_append)

lemma linf-norm-vec-of-list:
\(||vec-of-list xs\|_\infty = \text{max-list} (\text{map abs xs} @ [0])\)
by (simp add: linf_norm_vec_def)

lemma linf-norm-vec-as-Greatest:
fixes v :: 'a :: ring_1_abs_real_line vec
shows \(||v\|_\infty = (\text{GREATEST} a. a \in \text{abs v} \set (\text{list-of-vec v}) \cup \{0\})\)
unfolding linf_norm_vec_of_list[of list-of-vec v, simplified]
by (subst max_list_as_Greatest, auto)
lemma vec-of-poly-pCons:
  assumes $f 
eq 0$
  shows $\text{vec-of-poly } (\text{pCons } a \ f) = \text{vec-of-poly } f @ v \ \text{vec-of-list } [a]$
  using assms
  by (auto simp: vec-eq-iff Suc-diff-le)

lemma vec-of-poly-as-vec-of-list:
  assumes $f 
eq 0$
  shows $\text{vec-of-poly } f = \text{vec-of-list } (\text{rev } (\text{coeffs } f))$
proof (insert assms, induct f)
  case 0
  then show ?case by auto
next
  case (pCons a f)
  then show ?case
  by (cases f = 0, auto simp: vec-of-list-append vec-of-poly-pCons)
qed

lemma linf-norm-vec-of-poly [simp]:
  fixes $f :: '\text{ring-1-abs-real-line } poly$
  shows $\|\text{vec-of-poly } f\|_\infty = \|f\|_\infty$
proof (cases f = 0)
  case False
  then show ?thesis
  apply (unfold vec-of-poly-as-vec-of-list linf-norm-vec-of-list linf-norm-poly-def)
  apply (subst (1 2) max-list-as-Greatest, auto).
qed simp

lemma linf-norm-poly-as-Greatest:
  fixes $f :: '\text{ring-1-abs-real-line } poly$
  shows $\|f\|_\infty = (\text{GREATEST } a. \ a \in \text{abs } \text{set} (\text{coeffs } f) \cup \{0\})$
  using linf-norm-vec-as-Greatest [of vec-of-poly f]
  by simp

lemma vec-index-le-linf-norm:
  fixes $v :: '\text{ring-1-abs-real-line } vec$
  assumes $i < \text{dim-vec } v$
  shows $|v$_i$| \leq \|v\|_\infty$
apply (unfold linf-norm-vec-def, rule le-max-list) using assms
apply (auto simp: in-set-conv-nth intro: imageI exI [of - i]),

lemma coeff-le-linf-norm:
  fixes $f :: '\text{ring-1-abs-real-line } poly$
  shows $|\text{coeff } f$_i$| \leq \|f\|_\infty$
  using vec-index-le-linf-norm[of degree f - i vec-of-poly f]
  by (cases i \leq \text{degree } f, auto simp: coeff-eq-0)

class conjugatable-ring-1-abs-real-line = conjugatable-ring + ring-1-abs-real-line +
  power +
assumes sq-norm-as-sq-abs [simp]: \|a\|^2 = |a|^2

begin
subclass conjugatable-ordered-ring by (unfold-locales, simp)
end

instance int :: conjugatable-ring-1-abs-real-line
  by (intro-classes, simp add: numeral-2-eq-2)

instance rat :: conjugatable-ring-1-abs-real-line
  by (intro-classes, simp add: numeral-2-eq-2)

instance real :: conjugatable-ring-1-abs-real-line
  by (intro-classes, simp add: numeral-2-eq-2)

instance complex :: semiring-1-real-line
  apply intro-classes
  by (auto simp: complex-Re-Im-cancel-iff mult-le-cancel-left mult-le-cancel-right mult-neg-neg)

Due to the assumption \(a \leq |a|\) from Groups.thy, complex cannot be ring-1-abs-real-line!

instance complex :: ordered-ab-group-add-abs oops

lemma sq-norm-as-sq-abs [simp]: (sq-norm :: 'a :: conjugatable-ring-1-abs-real-line ⇒ 'a) = power2 ◦ abs
  by auto

lemma sq-norm-vec-le-linf-norm:
  fixes v :: 'a :: {conjugatable-ring-1-abs-real-line} vec
  assumes v ∈ carrier-vec n
  shows \|v\|^2 ≤ of-nat n * \|v\|_\infty^2

proof (insert assms, induct rule: carrier-vec-induct)
  case (Suc n a v)
  have [dest!]: ¬ |a| ≤ \|v\|_\infty ⇒ of-nat n * \|v\|_\infty^2 ≤ of-nat n * |a|^2
    by (rule real-linorder-cases[of |a| \|v\|_\infty], insert Suc, auto simp: less-le intro!: power-mono mult-left-mono)
  from Suc show ?case
    by (auto simp: ring-distrib max-def intro!: add-mono power-mono)
  qed simp

lemma sq-norm-poly-le-linf-norm:
  fixes p :: 'a :: {conjugatable-ring-1-abs-real-line} poly
  shows \|p\|^2 ≤ of-nat (degree p + 1) * \|p\|_\infty^2
  using sq-norm-vec-le-linf-norm[of vec-of-poly p degree p + 1]
  by (auto simp: carrier-dim-vec)

lemma coeff-le-sq-norm:
  fixes f :: 'a :: {conjugatable-ring-1-abs-real-line} poly
  shows |coeff f i|^2 ≤ \|f\|^2
proof (induct f arbitrary: i)
case (pCons a f)
show ?case
proof (cases i)
case (Suc ii)
  note pCons(2) [of ii]
  also have \( \|f\|^2 \leq |a|^2 + \|f\|^2 \) by auto
finally show ?thesis unfolding Suc by auto
qed auto
qed simp

lemma max-norm-witness:
  fixes f :: 'a :: ordered-ring-abs poly
  shows \( \exists i. \|f\|_\infty = |\text{coeff } f \text{ } i| \) by (induct f, auto simp add: max-def intro: exI [of - Suc] exI [of - 0])

lemma max-norm-le-sq-norm:
  fixes f :: 'a :: conjugatable-ring-1-abs-real-line poly
  shows \( \|f\|_\infty^2 \leq \|f\|^2 \)
proof
  from max-norm-witness[of f] obtain i where id: \( \|f\|_\infty = |\text{coeff } f \text{ } i| \) by auto
  show ?thesis unfolding id using coeff-le-sq-norm[of f i] by auto
qed

lemma (in conjugatable-ring) conjugate-minus: conjugate \((x - y)\) = conjugate \(x\) - conjugate \(y\)
by (unfold diff-conv-add-uminus conjugate-dist-add conjugate-neg, rule)

lemma conjugate-1[simp]: (conjugate I :: 'a :: {conjugatable-ring, ring-1}) = 1
proof
  have conjugate I * 1 = (conjugate I :: 'a) by simp
  also have conjugate \_ \_ = 1 by simp
  finally show ?thesis by (unfold conjugate-dist-mul, simp)
qed

lemma conjugate-of-int [simp]:
  (conjugate (of-int x) :: 'a :: {conjugatable-ring,ring-1}) = of-int x
proof (induct x)
case (nonneg n)
  then show ?case by (induct n, auto simp: conjugate-dist-add)
next
case (neg n)
  then show ?case apply (induct n, auto simp: conjugate-minus conjugate-neg)
  by (metis conjugate-1 conjugate-dist-add one-add-one)
qed

lemma sq-norm-of-int: \( \|\text{map-vec of-int } v \|_2^2 \| \)
5 Lattice

This theory implements the mathematical definition of lattice by means of locales and shows it forms a (HOL-Algebra) module.

theory Vector-Lattice-Locale
  imports HOL-Library.Multiset Norms Missing-Lemmas
begin

locale vlattice = abelian-group G for G (structure)

fun nat-mult where nat-mult 0 v = 0 | nat-mult (Suc n) v = v ⊕ nat-mult n v

lemma nat-mult-closed [simp]: v ∈ carrier G ⇒ nat-mult n v ∈ carrier G
  by (induct n, auto)

lemma nat-mult-add-distrib1 [simp]:
  assumes v: v ∈ carrier G shows nat-mult (x+y) v = nat-mult x v ⊕ nat-mult y v
  by (induct x, insert v, auto intro!: a-assoc[symmetric])

lemma nat-mult-add-distrib2 [simp]:
  assumes v ∈ carrier G and w ∈ carrier G
  shows nat-mult x (v ⊕ w) = nat-mult x v ⊕ nat-mult x w
proof (induct x)
  case (Suc x)
  have nat-mult (Suc x) (v ⊕ w) = v ⊕ w ⊕ nat-mult x (v ⊕ w) by simp
  also have ... = v ⊕ (w ⊕ nat-mult x v) ⊕ nat-mulx x w
    using assms a-assoc by (auto simp: Suc)
  also have w ⊕ nat-mult x v = nat-mult x v ⊕ w using assms a-comm by auto
  finally show ?case using a-assoc assms by auto
qed simp

definition int-mult (infixl · 70)
where x · y ≡ if x ≥ 0 then nat-mult (Int.nat x) v else ⊕ nat-mult (Int.nat (−x)) v
lemma int-mult-closed [simp]: \( v \in \text{carrier } G \implies x \cdot v \in \text{carrier } G \)
by (unfold int-mult-def, auto)

lemma [simp]: assumes \( v \in \text{carrier } G \)
shows zero-int-mult: \( 0 \cdot v = v \) and one-int-mult: \( 1 \cdot v = v \) and uminus-int-mult:
\( -x \cdot v = \ominus (x \cdot v) \)
using assms by (simp-all add: int-mult-def)

lemma int-mult-add-1:
assumes \( v \in \text{carrier } G \)
shows \( (x + 1) \cdot v = v \oplus x \cdot v \) (is \( ?l = ?r \))
proof (cases \( x - 1 :: \text{int} \) rule: linorder-cases)
case greater
then have \( x \geq 0 \) by auto
then obtain \( n \) where \( x = \text{int } n \) using zero-le-imp-eq-int by auto
have \( ?l = \text{int-mult} (x + 1) \cdot v \) by simp
also have \( ... = ?r \) using \( v \) by (unfold \( x \) int-mult-def nat-int-add, auto)
finally show \( ?thesis \).
next
case equal
with \( v \) show \( ?thesis \) by (auto simp: a-inv-def)
next
case less
then have \( -x - 2 \geq 0 \) by auto
from zero-le-imp-eq-int[OF this] obtain \( n \) where \( -x - 2 = \text{int } n \) by auto
then have \( x = - (\text{int } n + \text{int } 2) \) by auto
have \( ?r = \ominus v \ominus \text{int } n \cdot v \)
using \( v \)
by (unfold \( x \) int-mult-def add, inverse-inverse nat-int-add, simp add: \( x \) [symmetric] minus-add r-neg minus-eq)
also have \( ... = (\ominus (\text{int } 1 + \text{int } n)) \cdot v \)
using \( v \)
by (unfold int-mult-def add, inverse-inverse nat-int-add, simp add: add, inv-mult-group a-comm minus-eq)
also have \( ... = ?l \) by (auto simp: \( x \))
finally show \( ?thesis \) by simp
qed

lemmas int-mult-1-add = int-mult-add-1[folded add.commute[of 1]]

lemma int-mult-minus-1:
assumes \( v \in \text{carrier } G \)
shows \( (x - 1) \cdot v = \ominus v \ominus x \cdot v \) (is \( ?l = ?r \))
proof (cases \( x \) \( 1 :: \text{int} \) rule: linorder-cases)
case less
then have \( -x \geq 0 \) by auto
from zero-le-imp-eq-int[OF this] obtain \( n \) where \( -x = \text{int } n \) by auto
have \( ?l = (- (\text{int } n + \text{int } 1)) \cdot v \) by (simp add: \( x \) [symmetric])
also have \( ... = \ominus v \ominus - \text{int } n \cdot v \)
using v by (unfold int-mult-def add.inverse-inverse nat-int-add, simp add: minus-add)
also have ... = ?r by (fold x, auto)
finally show ?thesis.
next
case equal
with v show ?thesis by (auto simp: l-neg)
next
case greater
then have x - 2 ≥ 0 by auto
from zero-le-imp-eq-int[OF this] obtain n where x - 2 = int n by auto
have ?r = ⊖ v ⊕ (v ⊕ (v ⊕ nat-mult n v))
  by (unfold x int-mult-def add.inverse-inverse nat-int-add, simp)
also have ... = v ⊕ int n · v using v by (simp add: a-assoc[symmetric] l-neg int-mult-def)
also have ... = (int 1 + int n) · v by (unfold int-mult-def minus-minus nat-int-add, simp)
also have ... = ?l by (auto simp: x)
finally show ?thesis by simp
qed

lemma int-mult-add-distrib1:
  assumes v [simp]: v ∈ carrier G
  shows ((x + y) · v = x · v ⊕ y · v)
proof (induct x)
case (nonneg n)
  then show ?case using v by (induct n, auto simp add: ac-simps a-assoc[symmetric] int-mult-1-add)
next
case (neg n)
  show ?case
proof (induct n)
  case 0 show ?case using v by (auto simp add: int-mult-minus-1 minus-eq)
  case IH: (Suc n)
  have (− int (Suc (Suc n)) + y) = (− int (Suc n) + y − 1) by simp
  also have ... · v = ⊖ v ⊕ (− int (Suc n) · v) · v unfolding int-mult-minus-1[OF v] by simp
  also have ... = ⊖ v ⊕ (− int (Suc n) · v ⊕ y · v) by (unfold IH, simp)
  also have ... = (− int (Suc (Suc n)) · v) ⊕ y · v by (auto simp: a-assoc)
  also have ... = (− int (Suc (Suc n)) · v) ⊕ y · v by (auto simp: a-assoc)
  finally show ?case.
qed

lemma int-mult-minus-distrib1:
  assumes v ∈ carrier G
  shows (x - y) · v = x · v ⊕ y · v
using assms by (unfold diff-conv-add-uminus int-mult-add-distrib1, simp add: minus-eq)

lemma int-mult-mult:
  assumes v [simp]: v ∈ carrier G
  shows x · (y · v) = x * y · v
proof (cases x)
  case x: (nonneg n)
  show ?thesis by (unfold x, induct n, auto simp: field-simps int-mult-add-distrib1)
next
  case x: (neg n)
  show ?thesis proof (unfold x, induct n)
    case 0
    then show ?case by simp
  next
    case (Suc n)
    have − int (Suc (Suc n)) · (y · v) = (− int (Suc n) − 1) · (y · v) by simp
    also have ... = □ (y · v) ⊕ − int (Suc n) · (y · v) by (rule int-mult-minus-1, simp)
    also have ... = (− y · v) ⊕ − int (Suc n) · y · v unfolding Suc by simp
    also have ... = − int (Suc (Suc n)) · y · v
    by (subst int-mult-add-distrib1[symmetric], auto simp: left-diff-distrib)
    finally show ?case by (simp add: field-simps)
  qed
qed

lemma int-mult-add-distrib2[simp]:
  assumes v ∈ carrier G and w ∈ carrier G
  shows x · (v ⊕ w) = x · v ⊕ x · w using assms by (auto simp: int-mult-def minus-add)

abbreviation int-ring
  where int-ring ≡ (carrier = UNIV::int set, monoid.mult = op *, one = 1, zero = 0, add = op +)

abbreviation lattice-module
  where lattice-module ≡ (carrier = carrier G, monoid.mult = op ⊗, one = 1, zero = 0, add = op ⊕, module.smult = int-mult)

sublocale module: module int-ring lattice-module
rewrites carrier int-ring = UNIV
  and monoid.mult int-ring = op *
  and one int-ring = 1
  and zero int-ring = 0
  and add int-ring = op +
  and carrier lattice-module = carrier G
  and monoid.mult lattice-module = op ⊗
and one lattice-module = 1
and zero lattice-module = 0
and add lattice-module = op ⊕
and module.smult lattice-module = int-mult
by (unfold-locales,  
 auto simp: field-simps Units-def int-mult-mult l-neg r-neg int-mult-add-distrib1
 intro!: a-assoc a-comm exI[of - ⊖] bezl[of - ⊖ -])

end

end

6 Gram-Schmidt

theory Gram-Schmidt-2
 imports Jordan-Normal-Form.Gram-Schmidt
Jordan-Normal-Form.Show-Matrix
Jordan-Normal-Form.Matrix-Impl
Norms
begin

no-notation Gram-Schmidt.cscalar-prod (infix · 70)

lemma vec-conjugate-connect[simp]: Gram-Schmidt.vec-conjugate = conjugate
 by (auto simp: vec-conjugate-def conjugate-vec-def)

lemma scalar-prod-ge-0: (x :: 'a :: linordered-idom vec) · x ≥ 0
 unfolding scalar-prod-def
 by (rule sum-nonneg, auto)

class trivial-conjugatable-ordered-field =
 conjugatable-ordered-field + linordered-idom +
 assumes conjugate-id [simp]: conjugate x = x

lemma cscalar-prod-is-scalar-prod[simp]: (x :: 'a :: trivial-conjugatable-ordered-field vec) · c y = x · y
 unfolding conjugate-id
 by (rule arg-cong[of - - scalar-prod x], auto)

lemma orthogonal-is-orthogonal[simp]:
 orthogonal (xs :: 'a :: trivial-conjugatable-ordered-field vec list) = orthogonal xs
 unfolding orthogonal-def orthogonal-def by simp

instance rat :: trivial-conjugatable-ordered-field
 by (standard, auto)
instance real :: trivial-conjugatable-ordered-field
  by (standard, auto)

lemma vec-right-zero[simp]:
  (v :: 'a :: monoid-add vec) ∈ carrier-vec n → v + 0 v n = v
  by auto

context vec-module begin

lemma sumlist-dim: assumes (x ∈ set xs) = n
  shows dim vec (sumlist xs) = n
  using sumlist-carrier assms by fastforce

lemma sumlist-vec-index: assumes (x ∈ set xs) = n
  and i < n
  shows sumvec xs $ i = sum-list (map (λ x. x $ i) xs)
  unfolding M.sumlist-def using assms (1)
  proof (induct xs)
    case (Cons a xs)
    hence Cond (x ∈ set xs = n) by auto
    from Cons have (a + foldr op + xs (0 v n) $ i) = (∑ x ∈ xs. x $ i)
    by auto
    also have...
    apply (subst index-add-vec)
    unfolding IH
    using sumlist-dim[OF Cond, unfolded M.sumlist-def] assms by auto
    then show ?case by auto
    next
    case Nil thus ?case using assms by auto
  qed

lemma scalar-prod-left-sum-distrib:
  assumes vs: (∀ v. v ∈ set vvs = n and w: w ∈ carrier-vec n)
  and w: w ∈ carrier-vec n
  shows sumvec vvs · w = sum-list (map (λ v. v · w) vvs)
  using vs
  proof (induct vvs)
    case (Cons v vs)
    from Cons have v: v ∈ carrier-vec n and vs: sumvec vs ∈ carrier-vec n
    by (auto intro!: sumvec-carrier)
    have sumvec (v # vs) · w = sumvec ([v] @ vs) · w by auto
    also have... = (v + sumvec vs) · w
    by (subst sumvec-append, insert Cons v vs, auto)
    also have... = v · w + (sumvec vs · w)
    by (rule add-scalar-prod-distrib[OF v vs w])
    finally show ?case using Cons by auto
  qed (insert w, auto)

definition lattice-of :: 'a vec list ⇒ 'a vec set where
  lattice-of fs = range (λ c. sumvec (map (λ i. of-int (c i) · v fs ! i) [0 ..< length fs]))
lemma in-latticeE: assumes \( f \in \text{lattice-of } fs \) obtains \( c \) where
\[
f = \text{sumlist } (\lambda i. \text{of-int } (c \cdot fs \cdot i)) [0..<\text{length } fs])
\]
using \( \text{assms unfolding lattice-of-def by } \text{auto} \)

lemma in-latticeI: assumes \( f = \text{sumlist } (\lambda i. \text{of-int } (c \cdot fs \cdot i)) [0..<\text{length } fs]) \)
shows \( f \in \text{lattice-of } fs \)
using \( \text{assms unfolding lattice-of-def by } \text{auto} \)

lemma basis-in-latticeI: assumes \( fs: \text{set } fs \subseteq \text{carrier-vec } n \)
and \( f: f \in \text{set } fs \)
shows \( f \in \text{lattice-of } fs \)
proof –
from \( f \) obtain \( i \) where \( f = fs \cdot i \) and \( i: i < \text{length } fs \) unfolding \text{set-conv-nth}
by \text{auto} 
let \( ?c = \lambda j. \text{if } j = i \text{ then } 1 \text{ else } 0 \)
have \text{id: } 0.< \text{length } fs = 0.< i @ [i] @ [\text{Suc } i..< \text{length } fs]
by \text{(rule nth-equalityI, insert } i, \text{auto simp: nth-append, rename-tac } k, \text{case-tac } k = i, \text{auto) }
from \( fs \) have \( fs[\text{intro}]\cdot i. i < \text{length } fs \Rightarrow fs \cdot i \in \text{carrier-vec } n \) unfolding \text{set-conv-nth by } \text{auto} 
have \text{[simp]: } i. i < \text{length } fs \Rightarrow \text{dim-vec } (fs \cdot i) = n \text{ using } fs \text{ by } \text{auto} 
show \text{?thesis unfolding } f
apply \text{(rule in-latticeI[of - } ?c], \text{unfold id map-append, insert } i)
apply \text{(subst sumlist-append,force,force, subst sumlist-append, force, force)}
by \text{(subst sumlist-neutral, force, subst sumlist-neutral, force, auto)} 
qed 

lemma lattice-of-swap: assumes \( fs: \text{set } fs \subseteq \text{carrier-vec } n \)
and \( i: i < \text{length } fs \cdot j < \text{length } fs \cdot i \neq j \)
and \( gs: gs = fs[\cdot i := fs \cdot j, j := fs \cdot i] \)
shows \( \text{lattice-of } gs = \text{lattice-of } fs \)
proof –
let \( i, j \) and \( fs :: \cdot 'a \text{ vec list} \)
assume \( i: i < j < \text{length } fs \) and \( fs: \text{set } fs \subseteq \text{carrier-vec } n \)
fix \( i, j \) and \( fs :: \cdot 'a \text{ vec list} \)
assume \( i: i < j < \text{length } fs \) and \( fs: \text{set } fs \subseteq \text{carrier-vec } n \)
let \( ?gs = fs[\cdot i := fs \cdot j, j := fs \cdot i] \)
let \( ?len = [0..<i] @ [i] @ [\text{Suc } i..<j] @ [j] @ [\text{Suc } j..< \text{length } fs] \)
have \( [0..< \text{length } fs] = [0..< j] @ [j] @ [\text{Suc } j..< \text{length } fs] \text{ using } * \)
by \text{(metis append-Cons append-self-conv2 less-Suc-eq-le less-imp-add-positive}
upt-add-eq-append
upt-conv-Cons zero-less-Suc)
also have \( [0..< j] = [0..< i] @ [i] @ [\text{Suc } i..< j] \text{ using } * \)
by \text{(metis append-Cons append-self-conv2 less-Suc-eq-le less-imp-add-positive}
upt-add-eq-append
upt-conv-Cons zero-less-Suc)
finally have \( \text{len: } [0..< \text{length } fs] = ?len \text{ by } \text{simp} \)
from \( fs \) have \( fs: \cdot i. i < \text{length } fs \Rightarrow fs \cdot i \in \text{carrier-vec } n \) unfolding
set-conv-nth by auto
{
  fix f
  assume f ∈ lattice-of fs
  from in-latticeE[OF this, unfolded len] obtain c where
    f: f = sumlist (map (λi. of-int (c i) · v fs ! i) ?len) by auto
  define sc where sc = (λxs. sumlist (map (λi. of-int (c i) · v fs ! i) xs))
  define d where d = (λk. if k = i then c j else if k = j then c i else c k)
  define sd where sd = (λxs. sumlist (map (λi. of-int (d i) · v gs ! i) xs))
  have isc: set is ⊆ {0 ..< length fs} ⟹ is ∈ carrier-vec n for is
    unfolding sc-def by (intro sumlist-carrier, auto simp: fs)
  let ?a = sc [0..<i] let ?b = sc [i] let ?c = sc [Suc i ..< j] let ?d = sc [j]
  let ?e = sc [Suc j ..< length fs]
    using * by (auto intro: isc)
  have sc-sd: {i, j} ∩ set is ⊆ {} ⟹ is ∈ sd is for is
    unfolding sc-def sd-def by (rule arg-cong[of - - sumlist], rule map-cong,
      auto simp: d-def)
  have f = ?a + (?b + (?c + (?d + ?e)))
    unfolding f map-append sc-def using fs *
    by ((subst sumlist-append, force, force)+, simp)
  also have ... = f a + (?d + (?c + (?b + ?e))) using * by auto
  also have ... = f a + (?d + (?c + (?b + ?e)))
    unfolding f map-append sd-def using fs *
    by ((subst sumlist-append, force, force)+, simp)
  also have ?b = ?D unfolding sd-def sc-def d-def using * by (auto simp: d-def)
  also have ?d = ?B unfolding sd-def sc-def d-def using * by (auto simp: d-def)
  finally have f = ?A + (?B + (?C + (?D + ?E))) .
  also have ... = sumlist (map (λi. of-int (d i) · v gs ! i) ?len)
    unfolding f map-append sd-def using fs *
    by ((subst sumlist-append, force, force)+, simp)
  also have ... = sumlist (map (λi. of-int (d i) · v gs ! i) [0 ..< length ?gs])
    unfolding len[symmetric] by simp
  finally have f = sumlist (map (λi. of-int (d i) · v gs ! i) [0 ..< length ?gs])
    unfolding in-latticeE[OF this] have f ∈ lattice-of ?gs .
} hence lattice-of fs ⊆ lattice-of ?gs by blast
note main = this
{
  fix i j and fs :: 'a vec list
  assume #: i < length fs j < length fs i ≠ j and fs: set fs ⊆ carrier-vec n
  let ?gs = fs[ i := fs ! j, j := fs ! i]
  have lattice-of fs ⊆ lattice-of ?gs

77
proof (cases i < j)
  case True
  from main[OF this *(2) fs] show ?thesis .
next
  case False
  with *(3) have j < i by auto
  from main[OF this *(1) fs]
  have lattice-of fs ⊆ lattice-of (fs[j := fs ! i, i := fs ! j]) .
  also have fs[j := fs ! i, i := fs ! j] = ?gs using *
    by (metis list-update-swap)
  finally show ?thesis .
qed

lemma lattice-of-add: assumes fs: set fs ⊆ carrier-vec n
                  and ij: i < length fs j < length fs i ≠ j
                  and gs: gs = fs[ i := fs ! i + of-int l · v fs ! j]
shows lattice-of gs = lattice-of fs
proof –
  { fix i j l and fs :: 'a vec list
    assume *: i < j j < length fs and fs: set fs ⊆ carrier-vec n
    note * = ij(1) *
    let ?gs = fs[ i := fs ! i + of-int l · v fs ! j]
    let ?len = [0..<i] @ [i] @ [Suc i..<j] @ [j] @ [Suc j..<length fs]
    have [0..< length fs] = [0..< j] @ [j] @ [Suc j..< length fs] using *
      by (metis append-Cons append-self-conv2 less-Suc-eq-le less-imp-add-positive
          upt-add-eq-append
          upt-conv-Cons zero-less-Suc)
    also have [0 ..< j] = [0 ..< i] @ [i] @ [Suc i ..< j] using *
      by (metis append-Cons append-self-conv2 less-Suc-eq-le less-imp-add-positive
          upt-add-eq-append
          upt-conv-Cons zero-less-Suc)
    finally have len: [0..<length fs] = ?len by simp
    from fs have fs: i. i < length fs ⇒ fs ! i ∈ carrier-vec n unfolding
      set-conv-nth by auto
    from fs have fsd: i. i < length fs ⇒ dim-vec (fs ! i) = n by auto
    from fsd[of i] fsd[of j] * have fsd: dim-vec (fs ! i) = n dim-vec (fs ! j) = n
      by auto
  }

78
fix f
assume f ∈ lattice-of fs
from in-latticeE[OF this, unfolded len] obtain c where
  f: f = sumlist (map (λi. of-int (c i) * v fs ! i) ?len) by auto
declare sc where sc = (λ xs. sumlist (map (λi. of-int (c i) * v fs ! i) xs))
declare d where d = (λ k. if k = j then c j − c l else c k)
declare sd where sd = (λ xs. sumlist (map (λi. of-int (d i) * v ?gs ! i) xs))
have isc: set is ⊆ {0 ..< length fs} =⇒ sc is ∈ carrier-vec n for is
  unfolding sd-def using * by (intro sumlist-carrier, auto simp: fs)
have isd: set is ⊆ {0 ..< length fs} =⇒ sd is ∈ carrier-vec n for is
  unfolding sd-def using * by (intro sumlist-carrier, auto simp: fs)
let ?a = sc [0..<i] let ?b = sc [i] let ?c = sc [Suc i < j] let ?d = sc [j]
let ?e = sc [Suc j ..< length fs]

let ?E = sd [Suc j..< length fs]
let ?CC = carrier-vec n
  using * by (auto intro: isc)
  using * by (auto intro: isd)
have sc-sd: {i,j} ∩ set is ⊆ {} =⇒ sc is = sd is for is
  unfolding sd-def using * by (rule arg-cong[of - - sumlist], rule map-cong, auto simp: d-def, case-tac k = i, auto)
  have f = ?a + (?b + (?c + (?d + ?e)))
    unfolding f map-append using fs *
    by ((subst sumlist-append, force, force)+, simp)
also have ... = ?a + ((?b + ?d) + (?c + ?e)) using ae by auto
also have ... = ?A + ((?B + ?d) + (?C + ?E))
  unfolding f map-append using fs *
  by ((auto simp: sc-sd)
also have ?b + ?d = ?B + ?D unfolding sd-def using * by (rule eq-vecI, insert * fsd, auto simp: algebra-simps)
finally have f = ?A + (?B + (?C + (?D + ?E))) using AE by auto
also have ... = sumlist (map (λi. of-int (d i) * v ?gs ! i) ?len)
  unfolding f map-append using fs *
  by ((subst sumlist-append, force, force)+, simp)
also have ... = sumlist (map (λi. of-int (d i) * v ?gs ! i) [0..< length ?gs])
  unfolding len[symmetric] by simp
finally have f = sumlist (map (λi. of-int (d i) * v ?gs ! i) [0..< length ?gs])

from in-latticeE[OF this] have f ∈ lattice-of ?gs .
)
hence lattice-of fs ⊆ lattice-of ?gs by blast

note main = this}
{ fix i j and fs :: 'a vec list
assume *: i < j < length fs and fs: set fs ⊆ carrier-vec n

79
let ?gs = fs[ i := fs ! i + of-int l . v fs ! j ]
define gs where gs = ?gs
from main[OF * fs, of l, folded gs-def]
  have one: lattice-of fs ⊆ lattice-of gs 
  have *: i < j j < length gs set gs ⊆ carrier-vec n using * fs unfolding gs-def
set-conv-nth
  by (auto, rename-tac k, case-tac k = i, (force intro!: add-carrier-vec)+)
from fs have fs: ∧ i. i < length fs ⇒ fs ! i ∈ carrier-vec n unfolding set-conv-nth by auto
from fsd[of i fsd[of j] * have fsd: dim-vec (fs ! i) = n dim-vec (fs ! j) = n
by (auto simp: gs-def)
from main[OF *, of − l]
  have lattice-of-gs ⊆ lattice-of (gs[i := gs ! i + of-int (− l) . v gs ! j]) .
also have gs[i := gs ! i + of-int (− l) . v gs ! j] = fs unfolding gs-def
  by (rule nth-equalityI, auto, insert * fsd, rename-tac k, case-tac k = i, auto)
ultimately have lattice-of-fs = lattice-of ?gs using one unfolding gs-def by auto
}
note main = this
show ?thesis
proof (cases i < j)
case True
  from main[OF this ij(2) fs] show ?thesis unfolding gs by simp
next
case False
  with ij have ji: j < i by auto
  define hs where hs = fs[i := fs ! j, j := fs ! i]
  define ks where ks = hs[j := hs ! j + of-int l . v hs ! i]
  from ij fs have ji': i < length hs set hs ⊆ carrier-vec n unfolding hs-def by auto
  hence ji'': set ks ⊆ carrier-vec n i < length ks j < length ks i ≠ j
    using ji unfolding ks-def set-conv-nth by (auto, rename-tac k, case-tac k = i,
      force, case-tac k = j, (force intro!: add-carrier-vec)+)
  from lattice-of-swaps[OF fs i refl]
  have lattice-of-fs = lattice-of hs unfolding hs-def by auto
also have ... = lattice-of ks
  using main[OF ji'' ij' unfolding ks-def .
also have ... = lattice-of (ks[i := ks ! i, j := ks ! i])
  by (rule sym, rule lattice-of-swaps[OF ij'' refl])
also have ks[i := ks ! i, j := ks ! i] = gs unfolding gs ks-def hs-def
  by (rule nth-equalityI, insert ij, auto,
    rename-tac k, case-tac k = i, force, case-tac k = j, auto)
finally show ?thesis by simp
qed
qed

definition orthogonal-complement W = {x. x ∈ carrier-vec n ∧ (∀ y ∈ W. x · y = 0)}
lemma orthogonal-complement-subset:
  assumes $A \subseteq B$
  shows orthogonal-complement $B \subseteq$ orthogonal-complement $A$
unfolding orthogonal-complement-def using assms by auto

end

countext vec-space
begin

sublocale vec-module - n .

lemma in-orthogonal-complement-span[simp]:
  assumes [intro]:$S \subseteq \text{carrier-vec } n$
  shows orthogonal-complement $(\text{span } S) = \text{orthogonal-complement } S$
proof
  show orthogonal-complement $(\text{span } S) \subseteq \text{orthogonal-complement } S$
    by (fact orthogonal-complement-subset[OF in-own-span[OF assms]])
  { fix $x :: 'a vec$
    fix $a : 'a vec set$
    assume $x$ [intro]:$x \in \text{carrier-vec } n$ and $f$: finite $A$ and $S:A \subseteq S$
    assume i0:$\forall y \in S. \ x \cdot y = 0$
    have $x \cdot \text{lincomb } a \ A = 0$
      unfolding comm-scalar-prod[OF $x$ lincomb-closed[OF subset-trans[OF $S$ assms]]
  } thus ?case using assms $x$ by force
next
  case (2 $f$ $F$)
  { assume i:insert $f$ $F \subseteq S$
    hence $F:F \subseteq S$ and $f$: $f \in S$ by auto
    from $f$ $f$ assms
    have [intro]:$F \subseteq \text{carrier-vec } n$
      and fc[intro]:$f \in \text{carrier-vec } n$
      and [intro]:$x \in F \implies x \in \text{carrier-vec } n$ for $x$ by auto
    have laf:lincomb $a$ $F \cdot x = 0$ using $F$ 2 by auto
    have [simp]:$(\sum u \in F. (a \ u \cdot u) \cdot x) = 0$
      by (insert laf [unfolded lincomb-def], atomize(full), subst finsum-scalar-prod-sum)
    from $f$ i0 have [simp]:$f \cdot x = 0$ by (subst comm-scalar-prod) auto
    from lincomb-closed[OF subset-trans[OF $f$ $i$ assms]]
    have lincomb $a$ (insert $f$ $F$) \cdot x = 0$ unfolding lincomb-def
      apply(subst finsum-scalar-prod-sum,force,force)
      using 2(1,2) small-scalar-prod-distrib[OF $f$ $c$ $x$] by auto
  } thus ?case by auto
qed

thus orthogonal-complement $S \subseteq$ orthogonal-complement $(\text{span } S)$

81
lemma lincomb-list-add-vec-2: assumes us: set us ⊆ carrier-vec n
and x: x = lincomb-list lc (us [i := us ! i + c · v us ! j])
and i: j < length us i < length us i ≠ j
shows x = lincomb-list (lc (j := lc j + lc i * c)) us (is i = ?x)
proof –
  let ?xx = lc j + lc i * c
  let ?i = us ! i
  let ?j = us ! j
  let ?us = us [i := us ! i + c · v us ! j]
  from us have usk: k < length us ⇒ us ! k ∈ carrier-vec n for k by auto
  from usk i have iwj: ?i ∈ carrier-vec n ?j ∈ carrier-vec n by auto
  hence v: c · v ?j ∈ carrier-vec n ?v ∈ carrier-vec n by auto
  with us have us: set ?us ⊆ carrier-vec n unfolding set-conv-nth using i
     by (auto, rename-tac k, case-tac k = i, auto)
  from us have us: ∀ w ∈ set us. dim-vec w = n by auto
  from us have us: ∀ w ∈ set us. dim-vec w = n by auto
  have mset: mset-set {0..<length us} = #i# + #j# + (mset-set {0..<length us} – {i,j})
     by (rule multiset-eqI, insert i, auto, rename-tac x, case-tac x ∈ {0..<length us}, auto)
  define M2 where M2 = M.summset
     {#lc ia · v ?us ! ia. ia ∈# mset-set {0..<length us} – {i,j}}
  define M1 where M1 = M.summset {#(if i = j then ?xx else lc i) · v us ! i. i ∈# mset-set {0..<length us} – {i,j}}
  have M1: M1 ∈ carrier-vec n unfolding M1-def using usk by fastforce
  have M2: M1 = M2 unfolding M2-def M1-def
     by (rule arg-cong[of - M.summset], rule multiset.map-cong0, insert i usk, auto)
  have x1: x = lc j · v ?j + (lc i · v ?i + lc i · v (c · v ?j)) + M1
     unfolding x lincomb-list-def M2 M2-def
     apply (subst sumlist-as-summset, (insert us us v i j), auto simp: set-conv-nth)[1],
     insert i j v us us usk,
     simp add: mset smult-add-distrib-vec[OF ij(1) v(1)]
     by (sub M.summset-add-mset, auto)+
  have x2: ?xx = ?xx · v ?j + (lc i · v ?i + M1)
     unfolding x lincomb-list-def M1-def
     apply (subst sumlist-as-summset, (insert us us v i j), auto simp: set-conv-nth)[1],
     insert i j v us us usk,
     simp add: mset smult-add-distrib-vec[OF ij(1) v(1)]
     by (sub M.summset-add-mset, auto)+
  show ?thesis unfolding x1 x2 using M1 ij
     by (intro eq-vecI, auto simp: field-simps)
qed

lemma lincomb-list-add-vec-1: assumes us: set us ⊆ carrier-vec n
and \( x : x = \text{lincomb-list } lc \ us \)
and \( i : j < \text{length } us \ i < \text{length } us \ i \neq j \)
shows \( x = \text{lincomb-list } (lc \ (j := lc \ j \ - \ lc \ i \ + \ c)) \) (\( us \ [i := us ! i + c \ \cdot \ v \ \text{us} ! j] \))
(\( \text{is} \ - \ ?x \))

proof –
let \( \forall i : us ! i \)
let \( \forall j : us ! j \)
let \( \forall v = \forall i + c \ \cdot \ v \ ?j \)
let \( \forall ws = us [i := us ! i + c \ \cdot \ v \ \text{us} ! j] \)
from \( us \) have \( \text{usk} : k < \text{length } us \ \Longrightarrow \ us ! k \in \text{carrier-vec } n \) for \( k \) by auto
from \( \text{usk} \ i \) have \( \forall i : \exists i \in \text{carrier-vec } n \ \forall j \in \text{carrier-vec } n \) by auto
hence \( v : c \ \cdot \ v \ ?j \in \text{carrier-vec } n \ ?v \in \text{carrier-vec } n \) by auto
with \( us \) have \( \forall us \subseteq \text{carrier-vec } n \) unfolding \( \text{set-conv-nth} \) using \( i \)
by (auto, rename-tac \( k \), case-tac \( k = i \), auto)
from \( us \) have \( \forall w \in \text{set } us \). \text{dim-vec } w = n \) by auto
from \( us \) have \( \forall w \in \text{set } ?ws \). \text{dim-vec } w = n \) by auto

have \( \text{mset} : \text{mset-conv } \{0..<\text{length } us\} = \{\#\} + \{\#\} \) + (\( \text{mset-set } \{0..<\text{length } us\} - \{(i,j)\} \))
by (rule \text{multiset-eqI}, insert \( i \), auto, rename-tac \( x \), case-tac \( x \in \{0..<\text{length } us\} \), auto)

define \( M2 \) where \( M2 = \text{M.summset} \)
\( \{\#\} \in \# \text{mset-conv } \{0..<\text{length } us\} - \{(i,j)\} \)
define \( M1 \) where \( M1 = \text{M.summset} \{\#\} \in \# \text{mset-conv } \{0..<\text{length } us\} - \{(i,j)\} \)

have \( M1 : M1 \in \text{carrier-vec } n \) unfolding \( M1-def \) using \( \text{ask} \) by \text{fastforce}
have \( M2 : M1 = M2 \) unfolding \( M2-def \) \( M1-def \)
by (rule \text{arg-cong}[of - - \text{M.summset}], rule \text{multiset.map-cong0}, insert \( i \) usk, auto)

have \( x1 : x = lc \ j \ \cdot \ v \ ?j \ + \ (lc \ i \ \cdot \ v \ ?i + M1) \)
unfolding \( x \) \text{lincomb-list-def } M1-def

apply (subst \text{sumlist-as-summset}, (insert \( us \) \( i \) \( v \) \( ij \), auto simp: \text{set-conv-nth})[1], insert \( i \) \( ij \) \( v \) \( us \) \( us \) \( usk \),
simp add: \( \text{mset-smult-add-distrib-vec} \{\text{OF } ij(1) \ v(1)\} \))
by (subst \text{M.summset-add-mset}, auto)+

have \( x2 : \exists x = (lc \ j \ - \ lc \ i \ + \ c) \ \cdot \ v \ ?j \ + \ (lc \ i \ \cdot \ v \ ?i + lc \ i \ \cdot \ v \ ?j + M1) \)
unfolding \( x \) \text{lincomb-list-def } M2 M2-def

apply (subst \text{sumlist-as-summset}, (insert \( us \) \( i \) \( v \) \( ij \), auto simp: \text{set-conv-nth})[1], insert \( i \) \( ij \) \( v \) \( us \) \( us \) \( usk \),
simp add: \( \text{mset-smult-add-distrib-vec} \{\text{OF } ij(1) \ v(1)\} \))
by (subst \text{M.summset-add-mset}, auto)+

show ?thesis unfolding \( x1 \) \( x2 \) using \( M1 \) \( ij \)
by (intro \text{eq-vecI}, auto simp: \text{field-simps})

qed

lemma \( \text{add-vec-span} : \text{assumes } us : \text{set } us \subseteq \text{carrier-vec } n \)
and \( i : j < \text{length } us \ i < \text{length } us \ i \neq j \)
shows \( \text{span } (\text{set } us) = \text{span } (\text{set } us [i := \text{us} ! i + c \ \cdot \ v \ \text{us} ! j]) \) (\( \text{is} \ - \ \text{span } (\text{set } \ ?us) \))
proof

let \(?i = \text{us}!i\)
let \(?j = \text{us}!j\)
let \(?v = \text{us}!i + c \cdot \text{us}!j\)
from \(\text{us}!i \in \text{carrier-vec n \ ?j}\) 

have \(\text{us}!i \in \text{carrier-vec n \ by \ auto}\)

hence \(\text{us}!i \in \text{carrier-vec n \ by \ auto}\)

have \(\text{span (set us) = span-list us \ unfolding \ span-list-as-span[OF us]}..\)

also have \(\text{span (set ?ws \ unfolding \ span-list-as-span[OF ws]}..\)

proof

{\{ fix \(x\) 
  assume \(x \in \text{span-list us}\)
  then obtain \(lc\) where \(x = \text{lincomb lc us}\) by (metis in-span-listE)
  from \(\text{lincomb-list-add-vec-1[OF us this i, of c]}\)
  have \(x \in \text{span-list ?ws unfolding \ span-list-def by auto}\)
\}

moreover

{\{ fix \(x\) 
  assume \(x \in \text{span-list ?ws}\)
  then obtain \(lc\) where \(x = \text{lincomb lc ?ws}\) by (metis in-span-listE)
  from \(\text{lincomb-list-add-vec-2[OF us this i]}\)
  have \(x \in \text{span-list us unfolding \ span-list-def by auto}\)
\}

ultimately show \(?thesis by \text{ blast}\)

qed

also have \(\text{span (set ?ws) unfolding \ span-list-as-span[OF ws]}..\)

finally show \(?thesis\).

qed

lemma prod-in-span[intro!]:

assumes \(b \in \text{carrier-vec n \ S \subseteq \text{carrier-vec n \ a = 0 \ \vee \ b \in \text{span S}}\)

shows \(b \cdot v \in \text{span S}\)

proof\(\text{(cases a = 0)}\)

\text{case True}

then show \(?thesis by \text{ (auto simp:mult-0[OF assms{1}]) \span-zero}\)

next

\text{case False with assms have b \in \text{span S by auto}\)

from \(\text{this[THEN in-spanE]}\)

obtain aa A where \(a[introl!]; b = \text{lincomb aa A finite A A \subseteq S by auto}\)

hence \(\text{intro!}; (\lambda v. aa v \cdot v) \in A \rightarrow \text{carrier-vec n \ using \ assms by auto}\)

show \(?thesis proof\)

\text{ show a \cdot v \equiv \text{lincomb (\lambda v. a * aa v) A using a(1) unfolding \ lincomb-def}\)

\text{ smult-smult-assoc[symmetric]}\

\text{ by(subst finsum-smult[symmetric]) force+}

qed \text{ auto}\)

qed
lemma det-nonzero-congruence:
  assumes eq:A * M = B * M and det:det (M::'a mat) ≠ 0
  and M: M ∈ carrier-mat n n and carr:A ∈ carrier-mat n n B ∈ carrier-mat n n
  shows A = B
proof −
  have 1_m n ∈ carrier-mat n n by auto
  from det-non-zero-imp-unit[OF M det]
gauss-jordan-check-invertable[OF M this]
  have gj-fst:(fst (gauss-jordan M (1_m n))) = 1_m n by metis
  define Mi where Mi = snd (gauss-jordan M (1_m n))
  with gj-fst have gj:gauss-jordan M (1_m n) = (1_m n, Mi)
    unfolding fst-def snd-def by (auto split:prod.split)
  from gauss-jordan-compute-inverse(1,3)[OF M gj]
  have Mi: Mi ∈ carrier-mat n n and is1:M * Mi = 1_m n by metis+
  from arg-cong[OF eq, OF λ M. M * Mi]
  show A = B unfolding carr[THEN assoc-mult-mat[OF - M Mi]] is1 carr[THEN right-mult-one-mat].
qed

end

context cof-vec-space
begin

definition lin-indpt-list :: 'a vec list ⇒ bool where
lin-indpt-list fs = (set fs ⊆ carrier-vec n ∧ distinct fs ∧ lin-indpt (set fs))

definition basis-list :: 'a vec list ⇒ bool where
basis-list fs = (set fs ⊆ carrier-vec n ∧ length fs = n ∧ carrier-vec n ⊆ span (set fs))

lemma upper-triangular-imp-lin-indpt-list:
  assumes A: A ∈ carrier-mat n n
    and tri: upper-triangular A
    and diag: 0 ∉ set (diag-mat A)
  shows lin-indpt-list (rows A)
using upper-triangular-imp-distinct[OF assms]
using upper-triangular-imp-lin-indpt-rows[OF assms] A
unfolding lin-indpt-list-def by (auto simp: rows-def)

lemma basis-list-basis; assumes basis-list fs
  shows distinct fs lin-indpt (set fs) basis (set fs)
proof −
  from assms[unfolded basis-list-def]
  have len: length fs = n and C: set fs ⊆ carrier-vec n
    and span: carrier-vec n ⊆ span (set fs) by auto
  show b: basis (set fs)
  proof (rule dim-gen-is-basis[OF finite-set C])

85
show card (set fs) ≤ dim unfolding dim-is-n unfolding len[symmetric] by
(rule card-length)

show span (set fs) = carrier-vec n using (span C) by auto

qed

thus lin-indpt (set fs) unfolding basis-def by auto

show distinct fs

proof (rule contr)

assume ¬ distinct fs

hence card (set fs) < length fs using antisym-convl card-distinct card-length

by auto

also have ... = dim unfolding len dim-is-n ..

finally have card (set fs) < dim by auto

also have ... ≤ card (set fs) using span finite-set[of fs]

using b basis-def gen-ge-dim by auto

finally show False by simp

qed

qed

lemma basis-list-imp-lin-indpt-list: assumes basis-list fs shows lin-indpt-list fs

using basis-list-basis[OF assms] asserts unfolding lin-indpt-list-def basis-list-def

by auto

lemma mat-of-rows-mult-as-finsum:

assumes v ∈ carrier-vec (length lst) \( \land \) i. i < length lst \( \Rightarrow \) lst ! i ∈ carrier-vec n

defines f l ≡ sum (λ i. if l = lst ! i then v $ i else 0) \{0..<length lst\}

shows mat-of-cols-mult-as-finsum: mat-of-cols n lst * v = lincomb f (set lst)

proof –

from assms have \( \forall \) i < length lst. lst ! i ∈ carrier-vec n by blast

note an = all-nth-imp-all-set[OF this]

hence slc: set lst ⊆ carrier-vec n by auto

hence an [simp]: \( \land \) x. x ∈ set lst \( \Rightarrow \) dim-vec x = n by auto

have dl [simp]: dim-vec (lincomb f (set lst)) = n using an by (intro lincomb-dim,auto)

show ?thesis proof

show dim-vec (mat-of-cols n lst * v) = dim-vec (lincomb f (set lst)) using

assms(1,2) by auto

fix i assume ii: i < dim-vec (lincomb f (set lst)) hence i' i < n by auto

with an have fearr:(λv. f v · v) ∈ set lst → carrier-vec n by auto

from i' have (mat-of-cols n lst * v) $ i = row (mat-of-cols n lst) i · v by auto

also have ... = (\( \sum \) ia = 0..<dim-vec v. lst ! ia $ i * v $ ia)

unfolding mat-of-cols-def row-def scalar-prod-def

apply(rule sum.cong[OF refl]) using i an assms(1) by auto

also have ... = (\( \sum \) ia = 0..<length lst. lst ! ia $ i * v $ ia) using assms(1)

by auto

also have ... = (\( \sum \) x ∈ set lst. f x · x $ i)

unfolding f-def sum-distrib-right apply (subst sum.commute)

apply(rule sum.cong[OF refl])

unfolding if-distrib if-distrib-ap mult-zero-left sum.distrib[OF finite-set] by auto

86
also have \( \ldots = (\sum x \in \text{set } \text{lst}. (f x \cdot x) \downarrow i) \)  
apply (rule sum.cong[OF refl], subst index-smult-vec) using i slc by auto
also have \( \ldots = (\bigoplus v \in \text{set } \text{lst}. f v \cdot v) \downarrow i \)  
unfolding finsum-index[OF i' f carr slc] by auto
finally show \((\text{mat-of-cols } n \text{ lst } * v) \downarrow i = \text{lincomb } f (\text{set lst}) \downarrow i\)  
by (auto simp:lincomb-def)
qed


lemma \textit{basis-det-nonzero}:
  assumes \textit{db:basis \{(set G)\} and len:length }G \textit{= n}
  shows \((\text{det } (\text{mat-of-cols } n \text{ G})^T) \neq 0\)
proof
  have \textit{M-car1:mat-of-cols }n \text{ G }\in \text{carrier-mat }n \text{ n} \text{ using }\text{assms by auto}
  hence \textit{M-car:(mat-of-cols }n \text{ G})^T \in \text{carrier-mat }n \text{ n by auto}
  have \textit{lin-indpt (set G)}
    and \textit{inc-2;set G }\subseteq \text{carrier-vec }n
    and \textit{issp:carrier-vec }n = \text{span }\text{(set G)}
    and \textit{RG-in-carr:}\forall i. i < \text{length }G \implies G ! i \in \text{carrier-vec }n
    using \text{assms[unfolded basis-def]} by auto
  hence \textit{basis-list }G \text{ unfolding }basis-list-def \text{ using len by auto}
  from \textit{basis-list-basis[OF this]} have \textit{di:distinct }G \text{ by auto}
  have \textbf{det }((\text{mat-of-cols }n \text{ G})^T) \neq 0 \text{ unfolding det-0-iff-vec-prod-zero[OF M-car]}
proof
  assume \(\exists v. v \in \text{carrier-vec }n \wedge v \neq \theta_v n \wedge (\text{mat-of-cols }n \text{ G})^T * v = \theta_v n\)
  then obtain \(v \text{ where } v:v \in \text{span }\text{(set G)}\)
    \(v \neq \theta_v n \wedge (\text{mat-of-cols }n \text{ G})^T * v = \theta_v n\)
  unfolding \textit{issp} by blast
  from \textit{finite-in-span[OF finite-set inc-2 v(1)]} obtain \(a\)
    where \(a : v = \text{lincomb }a \text{ (set G)}\) by blast
  from \(v(1)\text{[folded issp]}\) obtain \(i\) where \(i:v \downarrow i \neq 0 \wedge i < n\) by fastforce
  hence \(\textit{inG}:G ! i \in \text{set }G\) using \textit{len} by auto
  have \textit{di2: distinct \{0..<\text{length }G\} by auto}
  define \(f \text{ where } f = (\lambda l. \sum i \in \text{set }\{0..<\text{length }G\}. \text{if } l = G ! i \text{ then } v \downarrow i \text{ else } 0)\)
  hence \(f:(G ! \overline{i}) = (\sum i a \leftarrow [0..<n]. \text{if } G ! i = G ! i \text{ then } v \downarrow i \text{ else } 0)\)
  unfolding \textit{f-def sum.distinct-set-conv-list[OF di2] unfolding len} by metis
  from \(v\) have \textit{mat-of-cols }n \text{ G }* v = \theta_v n
  unfolding \textit{transpose-mat-of-cols} by auto
  with \textit{mat-of-cols-mult-as-finsum[OF v(1)[folded issp len] RG-in-carr]}
  have \(f:\text{lincomb }f \text{ (set G) }= \theta_v n \text{ unfolding len f-def by auto}\)
  note \textit{simp} = \textit{list-trisect[OF i(2)[folded len], unfolded len]}
  note \textit{x} = \(i(2)[folded len]\)
  have \textit{[simp]:(\sum x \leftarrow [0..<i]. \text{if } G ! x = G ! i \text{ then } v \downarrow x \text{ else } 0) = 0\}
    by (rule sum-list-0, auto simp: nth-eq-iff-index-eq[OF di less-trans[OF \(OF \text{ di} \cdot x\)] x])
  have \textit{[simp]:(\sum x \leftarrow [Suc i..<n]. \text{if } G ! x = G ! i \text{ then } v \downarrow x \text{ else } 0) = 0\}
    apply (rule sum-list-0) using nth-eq-iff-index-eq[OF di - x] len by auto
  from \(i(1)\) have \(f (G ! \overline{i}) \neq 0\) unfolding \textit{f-f by auto}
from lin-dep-crit[of finite-set subset-refl TrueI inG this f]
  have lin-dep (set G).
  thus False using li by auto

qed
thus det0; det (mat-of-rows G) ≠ 0 by (unfold det-transpose[of M-car1])

qed

lemma lin-indpt-list-add-vec: assumes
  i: j < length us i < length us i ≠ j
  and indep lin-indpt-list us
shows lin-indpt-list (us [i := us ! i + c · v us ! j]) (is lin-indpt-list ?V)

proof –
from indep[unfolded lin-indpt-list-def] have us: set us ⊆ carrier-vec n
  and dist: distinct us and indep: lin-indpt (set us) by auto

let ?E = set us − {us ! i}
let ?us = insert (us ! i) ?E
let ?v = us ! i + c · v us ! j
from us i have usi: us ! i ∈ carrier-vec n us ! i ∈ set us
  and usj: us ! j ∈ carrier-vec n by auto
from usi usj have v: ?v ∈ carrier-vec n by auto

have fin: finite ?E by auto
have id: set us = insert (us ! i) (set us − {us ! i}) using i(2) by auto
from dist i have diff': us ! i ≠ us ! j unfolding distinct-conv-nth by auto
from subset-li-is-li[of OF indep] have indepE: lin-indpt ?E by auto
have Vid: set ?V = insert ?v ?E using set-update-distinct[of dist i(2)] by auto
have E: ?E ⊆ carrier-vec n using us by auto
have V: set ?V ⊆ carrier-vec n using us unfolding Vid by auto
from dist i have diff: us ! i ≠ us ! j unfolding distinct-conv-nth by auto
have vspan: ?v ∈ span ?E by auto

proof
  assume mem: ?v ∈ span ?E
  from diff i have us ! j ∈ ?E by auto
  hence us ! j ∈ span ?E using E by (metis span-mem)
  hence − c · v us ! j ∈ span ?E using smult-in-span[of E] by auto
  from span-add1[of E mem this] have ?v + (− c · v us ! j) ∈ span ?E.
  also have ?v + (− c · v us ! j) = us ! i using usi usj by auto
  finally have mem: us ! i ∈ span ?E.
  from lin-indpt-list-OF this obtain A where lc: us ! i = lincomb lc A and A:

  finite A
  A ⊆ set us − {us ! i}
  by auto

  let ?a = a (us ! i := −1) let ?A = insert (us ! i) A
  from A have fin: finite ?A by auto
  have lc: lincomb ?a A = us ! i unfolding lc
    by (rule lincomb-cong, insert A us lc, auto)
  have lincomb ?a ?A = 0·v n
    by (subst lincomb-insert2[of A(I)], insert A us lc usi diff, auto)
  from not-lindepD[of indep - - - this] A usi
  show False by auto
```

qed

from lin-dep-iff-in-span[OF E indepE v this] vs
have indep1: lin-indpt (set ?V) unfolding Vid by auto
from vmem dist have distinct ?V by (metis distinct-list-update)
with indep1 V show ?thesis unfolding lin-indpt-list-def by auto
qed

lemma scalar-prod-lincomb-orthogonal: assumes ortho: orthogonal gs and gs: set
gs \subseteq carrier-vec n
shows k \leq length gs \implies sumlist (map (\lambda i. g i \cdot \cdot gs ! i) [0 ..< k]) \cdot sumlist
(map (\lambda i. g i \cdot \cdot gs ! i) [0 ..< k])
= sum-list (map (\lambda i. g i \cdot g i \cdot (gs ! i \cdot gs ! i)) [0 ..< k])
proof (induct k)
case (Suc k)
  note ortho = orthogonalD[OF ortho]
  let ?m = length gs
from gs Suc(2) have gsi[simp]: \land i. i \leq k \implies gs ! i \in carrier-vec n by auto
from Suc have kn: k \leq ?m and k < ?m by auto
let ?v1 = sumlist (map (\lambda i. g i \cdot \cdot gs ! i) [0..<k])
let ?v2 = (g k \cdot \cdot gs ! k)
from Suc have id: [0 ..< Suc k] = [0 ..< k] \oplus [k] by simp
have id: sumlist (map (\lambda i. g i \cdot \cdot gs ! i) [0..<Suc k]) = ?v1 + ?v2
  unfolding id map-append
  by (subst sum-list-append, insert Suc(2), auto)
have v1: ?v1 \in carrier-vec n by (rule sumlist-carrier, insert Suc(2), auto)
have v2: ?v2 \in carrier-vec n by (insert Suc(2), auto)
have gsk: gs ! k \in carrier-vec n by simp
have v12: ?v1 + ?v2 \in carrier-vec n using v1 v2 by auto
have 0: i < k \implies (g i \cdot \cdot gs ! i) \cdot (g k \cdot \cdot gs ! k) = 0 for i
  by (subst scalar-prod-smult-distrib[OF - gsk], (insert k, auto))[1], subst smult-scalar-prod-distrib[OF - gsk], (insert k, auto)[1], insert ortho[of i k] k, auto)
have 0: ?v1 \cdot ?v2 = 0
  by (subst scalar-prod-left-sum-distrib[OF - v2], (insert Suc(2), auto)[1], rule sum-list-neutral, insert 0, auto)
show ?case unfolding id
  unfolding scalar-prod-add-distrib[OF v12 v1 v2]
  add-scalar-prod-distrib[OF v1 v2 v1]
  add-scalar-prod-distrib[OF v1 v2 v2]
  scalar-prod-smult-distrib[OF v2 gsk]
  smult-scalar-prod-distrib[OF gsk gsk]
unfolding Suc(1)[OF kn]
  by (simp add: 0 comm-scalar-prod[OF v2 v1])
qed auto

end
```

89
locale gram-schmidt = cof-vec-space n f-ty
for n :: nat and f-ty :: 'a :: trivial-conjugatable-ordered-field itself
begin

definition Gramian-matrix where
Gramian-matrix G k = (let M = mat k n (λ (i,j). (G ! i) $ j) in M * M^T)

lemma Gramian-matrix-alt-def: k ≤ length G ⇒
Gramian-matrix G k = (let M = mat-of-rows n (take k G) in M * M^T)

unfolding Gramian-matrix-def Let-def
by (rule arg-cong[of - - λ x. x * x^T], unfold mat-of-rows-def, intro eq-matI, auto)

definition Gramian-determinant where
Gramian-determinant G k = det (Gramian-matrix G k)

lemma orthogonal-imp-lin-indpt-list:
assumes ortho: orthogonal gs and gs: set gs ⊆ carrier-vec n
shows lin-indpt-list gs
proof
from orthogonal-distinct[of gs] ortho have dist: distinct gs by simp
show ?thesis unfolding lin-indpt-list-def
proof
fix lc
assume 0: lincomb lc (set gs) = 0_v n (is ?lc = -)
have lc: ?lc ∈ carrier-vec n by (rule lincomb-closed[OF gs])
let ?m = length gs
from 0 have 0 = ?lc · ?lc by simp
also have ?lc = lincomb-list (λi. lc (gs ! i)) gs
unfolding lincomb-as-lincomb-list-distinct[OF gs dist] ..
also have .. = sumlist (map (λi. lc (gs ! i) · v gs ! i) [0..< ?m])
unfolding lincomb-list-def by auto
also have .. · .. = (∑ i← [0..< ?m]. (lc (gs ! i) * lc (gs ! i)) * sq-norm (gs ! i)) (is - = sum-list ?sum)
unfolding scalar-prod-lincomb-orthogonal[OF ortho gs le-refl]
by (auto simp: sq-norm-vec-as-cscalar-prod power2-eq-square)
finally have sum-0: sum-list ?sum = 0 ..
have nonneg: \ x. x ∈ set ?sum ⇒ x ≥ 0
using zero-le-square[of lc (gs ! i) for i] sq-norm-vec-ge-0[of gs ! i for i] by auto

fix x
assume x: x ∈ set gs
then obtain i where i: i < ?m and x = gs ! i unfolding set-conv-nth by auto
hence lc x * lc x * sq-norm x ∈ set ?sum by auto
with sum-list-nonneg-eq-0-iff[of ?sum, OF nonneg] sum-0
have lc x = 0 ∨ sq-norm x = 0 by auto
with orthogonalD[OF ortho, OF i i, folded x]
have lc x = 0 by (auto simp: sq-norm-vec-as-cscalar-prod)
thus $\forall v \in \text{set } gs. \ lc v = 0$ by auto

qed

lemma projection-alt-def:
assumes carr:$(W :: 'a \text{ vec set}) \subseteq \text{carrier-vec } n x \in \text{carrier-vec } n$
and alt1: $y_1 \in W \ x - y_1 \in \text{orthogonal-complement } W$
and alt2: $y_2 \in W \ x - y_2 \in \text{orthogonal-complement } W$
shows $y_1 = y_2$
proof
  have carr: $y_1 \in \text{carrier-vec } n \ y_2 \in \text{carrier-vec } n \ x \in \text{carrier-vec } n - y_1 \in \text{carrier-vec } n$
    0, n \in \text{carrier-vec } n
    using alt1 alt2 carr by auto
  hence $y_1 - y_2 \in \text{carrier-vec } n$ by auto
  note carr = this carr from alt1 have $ya \in W = (x - y_1) \cdot ya = 0$ for $ya$
    unfolding orthogonal-complement-def by blast
  hence $(x - y_1) \cdot y_1 = 0$ using alt2 alt1 by auto
  hence eq1: $y_1 \cdot y_2 = x \cdot y_2 y_1 \cdot y_1 = x \cdot y_1$ using carr minus-scalar-prod-distrib
    by force+
    from this(1) have eq2: $y_2 \cdot y_1 = x \cdot y_2$ using carr comm-scalar-prod by force
    unfolding orthogonal-complement-def by blast
  hence $(x - y_2) \cdot y_1 = 0$ using alt2 alt1 by auto
  hence eq3: $y_2 \cdot y_2 = x \cdot y_2 y_2 \cdot y_1 = x \cdot y_1$ using carr minus-scalar-prod-distrib
    by force+
    with eq2 have eq4: $x \cdot y_1 = x \cdot y_2$ by auto
  have $\|(y_1 - y_2)\|^2 = 0$ unfolding sq-norm-vec-as-cscalar-prod cscalar-prod-is-scalar-prod
    using carr
      apply(subst minus-scalar-prod-distrib) apply force+
    apply(subst (0 0) scalar-prod-minus-distrib) apply force+
    unfolding eq1 eq2 eq3 eq4 by auto
  with sq-norm-vec-eq-0[of $(y_1 - y_2)]$ carr have $y_1 - y_2 = 0$, n by fastforce
  hence $y_1 - y_2 + y_2 = y_2$ using carr by fastforce
  also have $y_1 - y_2 + y_2 = y_1$ using carr by auto
  finally show $y_1 = y_2$.
qed

definition weakly-reduced :: 'a => nat => 'a vec list => bool

where weakly-reduced $\alpha$ k gs = $(\forall i. \ Suc i < k \longrightarrow \ sq-norm (gs ! i) \leq \alpha \ast sq-norm (gs ! (Suc i)))$

definition strictly-reduced :: nat => 'a => 'a vec list => (nat => nat => 'a) => bool

where strictly-reduced N $\alpha$ gs mu = (weakly-reduced $\alpha$ N gs \ 
$(\forall i j. i < N \longrightarrow j < i \longrightarrow abs (mu i j) \leq 1/2))$
definition
  is-projection \( w S v \) = \( (w \in \text{carrier-vec } n \land v - w \in \text{span } S \land (\forall u. u \in S \rightarrow w \cdot u = 0)) \)

definition projection where
  projection \( S \mathfrak{f} \) \( \equiv \) \( \text{SOME } v. \text{is-projection } v S \mathfrak{f} \)

context
  \( \text{fixes } \mathfrak{f}s :: 'a \text{ vec list} \)

begin

fun \( \text{gso and } \mu \text{ where} \)
  \( \text{gso } i = \mathfrak{f}s ! i + \text{sumlist (map } (\lambda j. - \mu i j \cdot v \text{ gso } j) [0 \ldots i]) \)
| \( \mu i j = \begin{cases} \text{if } j < i \text{ then } (\mathfrak{f}s ! i \cdot \text{gso } j)/\text{sq-norm } (\text{gso } j) \text{ else if } i = j \text{ then } 1 \text{ else } 0 \end{cases} \)

declare \( \text{gso.simps[simp del]} \)
declare \( \mu.\text{simps[simp del]} \)

fun \( \text{adjuster-wit :: 'a list } \Rightarrow 'a \text{ vec } \Rightarrow 'a \text{ vec list } \Rightarrow 'a\text{ list } \times 'a \text{ vec} \)
  \( \text{where} \)
  \( \text{adjuster-wit } \mathfrak{w}t\mathfrak{s} \mathfrak{w} \mathfrak{u}s = (\mathfrak{w}t\mathfrak{s}, 0 v n) \)
  | \( \text{adjuster-wit } \mathfrak{w}t\mathfrak{w} \mathfrak{w} \mathfrak{u}s (\mathfrak{u}\#\mathfrak{u}s) = \begin{cases} \text{let } a = (\mathfrak{w} \cdot \mathfrak{u})/\text{sq-norm } \mathfrak{u} \text{ in} \end{cases} \)
    \( \text{case } \text{adjuster-wit } (a \# \text{wits}) \mathfrak{w} \mathfrak{u}s \text{ of } \)
    \( \begin{cases} \text{wit, } -a \cdot v \mathfrak{u} + v \end{cases} \)

fun \( \text{sub2-wit where} \)
  \( \text{sub2-wit } \mathfrak{u}s = (\mathfrak{u}s, \mathfrak{u}s) \)
  | \( \text{sub2-wit } \mathfrak{u}s \mathfrak{w} \mathfrak{w} \mathfrak{u}s = \begin{cases} \text{let } a = \text{aw} + \mathfrak{w} \text{ in} \end{cases} \)
    \( \text{case } \text{sub2-wit } (a \# \mathfrak{w} \mathfrak{u}s) \mathfrak{w} \mathfrak{u}s \text{ of } \)
    \( \begin{cases} \text{wits, } v \mathfrak{v}s \end{cases} \)

definition main :: 'a vec list \( \Rightarrow 'a\text{ list } \times 'a \text{ vec list} \)
  \( \text{where} \)
  \( \text{main } \mathfrak{u}s = \text{sub2-wit } \mathfrak{u}s \mathfrak{u}s \)

lemma \( \text{gso-carrier'}\text{[intro]}: \)
  \( \text{assumes } \bigwedge i. i \leq j \Rightarrow \mathfrak{f}s ! i \in \text{carrier-vec } n \)
  \( \text{shows } \text{gso } j \in \text{carrier-vec } n \)
using \( \text{assms} \)
proof \( (\text{induct } j \text{ rule:nat-less-induct[rule-format]} \) \)
  case \( (1 j) \)
then show \( \text{?case unfolding gso.simps[of } j \text{]} \) by \( (\text{auto intro!:sumlist-carrier add-carrier-vec}) \)
qed

lemma \( \text{adjuster-wit}: \text{assumes res: adjuster-wit } \mathfrak{w} \mathfrak{t} \mathfrak{w} \mathfrak{u}s \mathfrak{w} \mathfrak{u}s = (\mathfrak{w} \mathfrak{t}s', a) \)
  \( \text{and } \mathfrak{w}: \mathfrak{w} \in \text{carrier-vec } n \)
  \( \text{and } \mathfrak{u}s: \bigwedge i. i \leq j \Rightarrow \mathfrak{f}s ! i \in \text{carrier-vec } n \)
  \( \text{and } \mathfrak{u}s-\mathfrak{g}s: \mathfrak{u}s = \text{map } \text{gso } (\text{rev } [0 \ldots j]) \)
  \( \text{and } \mathfrak{w}t\mathfrak{s}: \mathfrak{w} \mathfrak{t}s = \text{map } (\mu i) [j \ldots< n] \)
  \( \text{and } j: j \leq n \) \( j \leq i \)

92
and wi: \( w = fs \uparrow i \)

shows \( \text{adjuster } n \ w u = a \land a \in \text{carrier-vec } n \land \text{wits'} = \text{map } (\mu _i)[0..<n] \land (a = \text{sumlist } (\text{map } (\lambda _j - \mu _i j \cdot v \ gso j)[0..<j])) \)

using \( \text{res } us\ wits j \)

proof (induct us arbitrary; wits wits’ a j)

case (\( \text{Cons } u \ wits\ wits’ \ a \ j \))

note \( \text{us-gs} = \text{Cons}(4) \)

note \( \text{wits} = \text{Cons}(5) \)

note \( \text{jn} = \text{Cons}(6-7) \)

from \( \text{us-gs} \) obtain \( j j \) where \( j = \text{Suc } jj \) by (cases \( j \), auto)

from \( j n \ j \) have \( jj: jj \leq n \ jj < n jj \leq i jj < i \) by auto

have \( \text{vj } [0..<jj] = [0..<jj] \) unfolding \( j \) by simp

have \( \text{jn} : [jj ..< n] = jj \# [jj ..< n] \) using \( jj(2) \) unfolding \( j \) by (rule \( \text{upt-cone-Cons} \))

from \( \text{us-gs} \) unfolding \( jj \) have \( \text{ugs: } u = \text{gso } jj \) and \( \text{us: } us = \text{map } \text{gso } (\text{rev } [0..<jj]) \)

by auto

let \( \?w = w \cdot u \) (\( u \cdot a \))

have \( \text{munij: } ?w = \mu _i jj \) unfolding \( \mu \cdot \text{simp}[\text{of } i jj] \) \( \text{ugs wi sq-norm-vec-as-cscalar-prod} \)

using \( jj \) by auto

have \( \text{wuits: } ?w \# \text{wits } = \text{map } (\mu _i)[jj..<n] \) unfolding \( j n \) \( \text{munij by simp} \)

obtain \( \text{wuits } b \) where \( \text{rec: } \text{adjuster-wit } (\?w \# \text{wits}) \) \( w u s = (\text{wuits,b}) \) by force

from \( \text{Cons}(1) \) \((\text{OF this } \text{Cons}(3) \) \( \text{us wuits } jj(1,3) \) \( \text{unfolded } j \) \( \text{have } \text{IH:} \)

\( \text{adjuster } n \ w u s = b \ wuits = \text{map } (\mu _i)[0..<n] \)

\( b = \text{sumlist } (\text{map } (\lambda _j - \mu _i j \cdot v \ gso j)[0..<jj]) \)

and \( b : b \in \text{carrier-vec } n \) by auto

from \( \text{Cons}(2) \) simplfied unfolded Let-def rec split sq-norm-vec-as-cscalar-prod cscalar-prod-is-scalar-prod]

have \( \text{id: } \text{wits'} = \text{wuits and } a : a = - ?w \cdot v \ u + b \) by auto

have \( \text{1: } \text{adjuster } n \ w (u \# us) = a \) unfolding \( \text{a } \text{IH}(1) [\text{symmetric] by auto} \)

from \( \text{id } \text{IH}(2) \) have \( \text{wits': } \text{wits'} = \text{map } (\mu _i)[0..<n] \) by simp

have \( \text{car\text{r}: } \text{set } (\text{map } (\lambda _j - \mu _i j \cdot v \ gso j)[0..<jj]) \subseteq \text{carrier-vec } n \)

set \( (\text{map } (\lambda _j - \mu _i j \cdot v \ gso j)[0..<jj]) \subseteq \text{carrier-vec } n \) and \( w u : \text{w } u \in \text{carrier-vec } n \)

using \( \text{Cons } j \) by (auto intro; \( \text{gso-carrier'} \))

from \( u b a \) have \( ac : a \in \text{carrier-vec } n \) \( \text{dim-vec } (- ?w \cdot v \ u) = n \) \( \text{dim-vec } b = n \)

\( \text{dim-vec } u = n \) by auto

show \( ?\text{case} \)

apply \( \text{(intro } \text{conj[OF 1] ac ez1 conj wits')} \)

unfolding \( \text{car } \text{r } \text{a } \text{IH } \text{zj munij ugs[symmetric] map-append} \)

apply \( \text{(subst sumlist-append)} \)

using \( \text{Cons.prems } j \) apply force

using \( b u ugs \text{ IH}(3) \) by auto

qed auto

lemma \( \text{sub2-wit}: \)

assumes \( \text{set } us \subseteq \text{carrier-vec } n \) \( \text{set } ws \subseteq \text{carrier-vec } n \) \( \text{length } us + \text{length } ws = m \)

and \( us = \text{map } (\lambda i. \text{fs } i i)[i ..< m] \)

and \( ws = \text{map } \text{gso } (\text{rev } [0..<i]) \)

and \( us : \land _j j < m \Rightarrow \text{fs } i j < \text{carrier-vec } n \)

93
and mn: m \leq n
shows snd (sub2-wit us ws) = vvs \Longrightarrow \text{gram-schmidt-sub2} n us ws = vvs
\land vvs = \text{map} gso [i..<m]
using assms(1-6)
proof (induct ws arbitrary: us vvs i)
case (Cons w ws us vs)
  note us = Cons(3) note wvs = Cons(4)
  note ws-gs = Cons(7)
  from wsf' have i < m i \leq m by (cases i < m, auto)+
  hence i-m: [i..<m] = i # [Suc i..<m] by (metis upt-conv-Cons)
  from (i < m) mn have i < n i \leq n i \leq m by auto
  hence i-n: [i..<n] = i # [Suc i..<n] by (metis upt-conv-Cons)
  from wsf' i-m have wsf: ws = \text{map} (\lambda i. \text{fs} ! i) [Suc i..<m]
    and fiw: \text{fs} ! i = w by auto
  from wvs have w: w \in \text{carrier-vec} n and ws: set ws \subseteq \text{carrier-vec} n by auto
  let ?list = 1 # replicate (n - Suc (length us)) 0
  have map (\mu i) [Suc i..<n] = map (\lambda i. 0) [Suc i..<n]
    by (rule map-cong[OF refl], unfold \mu.\text{simps}[of i], auto)
  moreover have \mu i i = 1 unfolding \mu.\text{simps} by simp
  ultimately have map (\mu i) [i..<n] = 1 # map (\lambda i. 0) [Suc i..<n] unfolding i-n by auto
  also have \ldots = ?list using (i < n) unfolding map-replicate-const by (auto simp: us-gs)
  finally have list: ?list = map (\mu i) [i..<n] by auto
  let ?a = adjuster-wit ?list w us
  obtain vv where a: ?a = (wit,a) by force
  obtain vv where gs: snd (sub2-wit ((a + w) # us) ws) = vv by force
  from adjuster-wit[OF a w Cons(8) us-gs list (i \leq n) - fiw[symmetric]] us wws (i < m)
    have avus: set ((a + w) # us) \subseteq \text{carrier-vec} n
      and aa: adjuster n w us = a a \in \text{carrier-vec} n
      and aaaa: a = list (map (\lambda j. - \mu i j \cdot \text{gso} j) [0..<i])
      and wit: wit = map (\mu i) [0..<n]
    by auto
  have av-gs: a + w = gso i unfolding gso.\text{simps}[of i] fiw aaa[symmetric] using a(2) w by auto
  with us-gs have us-gs': (a + w) # us = map gso (\text{rev} [0..<Suc i]) by auto
  from Cons(1)[OF gs avus ws - wsf gs-us-gs' Cons(8)] Cons(5)
  have IH: gram-schmidt-sub2 n ((a + w) # us) ws = vvs
    and vv: vv = map gso [Suc i..<m] by auto
  from gs a aa IH Cons(5)
  have gs-vs: gram-schmidt-sub2 n us (w # ws) = vs and vs: vs = (a + w) # vv
  using Cons(2)
    by (auto simp add: Let-def snd-def split:prod.splits)
  from vs vv aa-gs have vs: vs = map gso [i..<m] unfolding i-m by auto
  with gs-vs show \?case by auto
qed auto

94
lemma inv-in-span:
  assumes incarr[intro]:U ⊆ carrier-vec n and insp:a ∈ span U
  shows − a ∈ span U
proof −
  from insp[THEN in-spanE] obtain aa A where a:a = lincomb aa A finite A A
  ⊆ U by auto
  with assms have [intro!]: (λv. aa v · v) ∈ A → carrier-vec n by auto
  from a(1) have e1:= a = lincomb (λx. − 1 * aa x) A unfolding smult-smult-assoc[symmetric]
  lincomb-def
  by (subst finsum-smult[symmetric]) force+
  show ?thesis using e1 a span-def by blast
qed

lemma non-span-det-zero:
  assumes len: length G = n
  and nonb:¬ (carrier-vec n ⊆ span (set G))
  and carr:set G ⊆ carrier-vec n
  shows det (mat-of-rows n G) = 0 unfolding det-0-ifc-vec-prod-zero
proof −
  let ?A = (mat-of-rows n G)ᵀ let ?B = 1_m n
  from carr have carr-mat:?A ∈ carrier-mat n n ?B ∈ carrier-mat n n mat-of-rows
  n G ∈ carrier-mat n n
  using len mat-of-rows-carrer(1) by auto
  from carr have g-len: ∃ i. i < length G ⇒ G ! i ∈ carrier-vec n by auto
  from nonb obtain v where v:v ∈ carrier-vec n v ∉ span (set G) by fast
  hence v ≠ 0_v n using span-zero by auto
  obtain B C where gj: gauss-jordan ?A ?B = (B,C) by force
  note gj = carr-mat(1,2) gj
  hence B:B = fst (gauss-jordan ?A ?B) by auto
  from gauss-jordan[OF gj] have BC:B ∈ carrier-mat n n by auto
  from gauss-jordan-transform[OF gj] obtain P where
  P: P ∈ Units (ring-mat TYPE(a) n ?B) B = P * ?A by fast
  hence PC:P ∈ carrier-mat n n unfolding Units-def by (simp add: ring-mat-simps)
  from mat-inverse[OF PC] P obtain PI where mat-inverse P = Some PI by fast
  from mat-inverse(2)[OF PC this]
  have PI:P * PI = 1_m n PI * P = 1_m n PI ∈ carrier-mat n n by auto
  have B ≠ 1_m n proof
    assume B = ?B
    hence ?A * P = ?B unfolding P
    using PC P(1) carr-mat(1) mat-mult-left-right-inverse by blast
    hence ?A * v = v using v by auto
    hence ?A * (P *_v v) = v unfolding assoc-mult-mat-vec[OF carr-mat(1) PC
    v(1)].
    hence v-eq: mat-of-cols n G *_v (P *_v v) = v
    unfolding transpose-mat-of-rows by auto
    have pvc:P *_v v ∈ carrier-vec (length G) using PC v len by auto
    from mat-of-cols-mult-as-finsum[OF pvc g-len,unfolded v-eq] obtain a where
    v = lincomb a (set G) by auto
hence \( v \in \text{span}(\text{set } G) \) by (intro in-span[OF - finite-set subset-refl])
thus False using v by auto
qed

with det-non-zero-imp-unit[OF carr-mat(1)] show ?thesis
  unfolding gauss-jordan-check-invertable[OF carr-mat(1,2)] B det-transpose[OF carr-mat(3)]
  by metis
qed

lemma span-basis-det-zero-iff:
  assumes length G = n set G ⊆ carrier-vec n
  shows carrier-vec n ⊆ \text{span}(\text{set } G) ←→ \text{det}(\text{mat-of-rows } n \text{ G}) \neq 0 (is ?q1)
             carrier-vec n ⊆ \text{span}(\text{set } G) ←→ \text{basis}(\text{set } G) (is ?q2)
             \text{det}(\text{mat-of-rows } n \text{ G}) \neq 0 ←→ \text{basis}(\text{set } G) (is ?q3)

proof –
  have dc: \text{det}(\text{mat-of-rows } n \text{ G}) \neq 0 =⇒ carrier-vec n ⊆ \text{span}(\text{set } G)
      using assms non-span-det-zero by auto
  have cb: carrier-vec n ⊆ \text{span}(\text{set } G) =⇒ \text{basis}(\text{set } G)
      using assms basis-list-basis by (auto simp: basis-list-def)
  have bd: \text{basis}(\text{set } G) =⇒ \text{det}(\text{mat-of-rows } n \text{ G}) \neq 0
      using assms basis-det-nonzero by auto
  show ?q1 ?q2 ?q3 using dc cb bd by metis+
qed

lemma partial-connect: fixes vs
  assumes length fs = m k ≤ m m ≤ n set us ⊆ carrier-vec n snd (main us) = vs
  us = take k fs set fs ⊆ carrier-vec n
  shows gram-schmidt n us = vs
  vs = map gso [0..<k]

proof –
  have [simp]: map (op ! fs) [0..<k] = take k fs using assms(1,2) by (intro nth-equalityI, auto)
  have carr: j < m =⇒ fs ! j ∈ carrier-vec n for j using assms by auto
  from sub2-wit[OF OF - assms(4) - - carr - assms(5)[unfolded main-def], of k 0] assms
  show gram-schmidt n us = vs vs = map gso [0..<k] unfolding gram-schmidt-code
    by auto
qed

lemma adjuster-wit-small:
  (adjuster-wit v a xs) = (x1,x2)
  ⇐⇒ \text{fst}(\text{adjuster-wit } v \text{ a } \text{xs}) = x1 ∧ x2 = adjuster n a xs

proof(induct xs arbitrary: a v x1 x2)
case (Cons a xs)
then show ?case
  by (auto simp:Let-def sq-norm-vec-as-cscalar-prod split:prod.splits)
qed auto
lemma rev-unsimp: \( \text{rev } xs \odot (r \# rs) = \text{rev } (r\#xs) \odot rs \) by (induct xs, auto)

lemma sub2: \( \text{rev } xs \odot \text{snd } (\text{sub2-wit } xs \ us) = \text{rev } (\text{gram-schmidt-sub } n \ xs \ us) \)
proof -
have sub2-wit xs us = \((x1, x2)\) \(\implies\) \(\text{rev } xs \odot x2 = \text{rev } (\text{gram-schmidt-sub } n \ xs \ us)\)
for \(x1\) \(x2\) \(xs\) \(us\)
thus ?thesis
qed

lemma gso-connect: \( \text{snd } (\text{main } us) = \text{gram-schmidt } n \ us \) unfolding main-def
gram-schmidt-def
using sub2[of Nil us] by auto

lemma lin-indpt-list-nonzero:
assumes lin-indpt-list G
shows \(0 \vdash n \notin \text{set } G\)
proof -
from assms[unfolded lin-indpt-list-def] have lin-indpt (set G) by auto
from vs-zero-lin-dep[OF - this] assms[unfolded lin-indpt-list-def] show zero: \(0 \vdash n \notin \text{set } G\) by auto
qed

context
fixes \(m :: \text{nat}\)
begin
definition \(M\) where \(M \equiv \text{mat } m \ m (\lambda (i,j). \mu \ i \ j)\)
end

context
fixes \(vs\)
assumes indep: lin-indpt-list \(fs\)
and len-fs: length \(fs\) = \(m\)
and snd-main: \(\text{snd } (\text{main } fs) = vs\)
begin

lemma fs-carrier[simp]: \(\text{set } fs \subseteq \text{carrier-vec } n\)
and dist: distinct \(fs\)
and lin-indpt: lin-indpt (set \(fs\))
using indep[unfolded lin-indpt-list-def] by auto
lemmas assm = len-fs fs-carrier snd-main

lemma f-carrier[simp]: \(i < m \implies fs ! i \in \text{carrier-vec } n\)

97
using fs-carrier len-fs unfolding set-conv-nth by force

lemma gso-carrier[simp]: \( i < m \implies gso i \in \text{carrier-vec} n \)
using gso-carrier' f-carrier by auto

lemma gso-dim[simp]: \( i < m \implies \dim-vec (gso i) = n \) by auto
lemma f-dim[simp]: \( i < m \implies \dim-vec (fs \! i) = n \) by auto

lemma mn: \( m \leq n \)
proof -
  have n: \( n = \dim \) by (simp add: dim-is-n)
  have m: \( m = \card (set fs) \)
    unfolding len-fs[symmetric]
    using distinct-card[OF dist]
    by simp
  from m n have mn: \( m \leq n \iff \card (set fs) \leq \dim \) by simp
  show ?thesis unfolding mn
    by (rule li-le-dim[OF - fs-carrier lin-indpt], simp)
qed

lemma main-connect: gram-schmidt n fs = vs
vs = map gso [0..<m]
proof -
  have gram-schmidt-sub2 n [] fs = vs \land vs = map gso [0..<m]
    by (rule sub2-wit[OF - assm(2) - - - mn], insert snd-main len-fs, auto simp: intro !: nth-equalityI)
  thus gram-schmidt n fs = vs vs = map gso [0..<m]
    by (auto simp: gram-schmidt-code)
qed

lemma reduced-vs-E: weakly-reduced \( \alpha k vs \implies k \leq m \implies \Suc i < k \implies sq-norm (gso i) \leq \alpha \ast sq-norm (gso (\Suc i)) \)
unfolding weakly-reduced-def main-connect(2) by auto

abbreviation (input) FF where FF \( \equiv \mat-of-rows n fs \)
abbreviation (input) Fs where Fs \( \equiv \mat-of-rows n vs \)

lemma FF-dim[simp]: \( \dim-row FF = m \) \( \dim-col FF = n \) FF \( \in \carrier-mat m n \)
unfolding mat-of-rows-def by (auto simp: assm len-fs)

lemma Fs-dim[simp]: \( \dim-row Fs = m \) \( \dim-col Fs = n \) Fs \( \in \carrier-mat m n \)
unfolding mat-of-rows-def by (auto simp: assm main-connect)

lemma M-dim[simp]: \( \dim-row M = m \) \( \dim-col M = m \) M \( \in \carrier-mat m m \)
unfolding M-def by auto

lemma FF-index[simp]: \( i < m \implies j < n \implies FF ! (i,j) = fs ! i \)$ j
unfolding mat-of-rows-def using assm by auto

lemma M-index[simp]: \( i < m \implies j < m \implies M ! (i,j) = \mu i j \)
unfolding M-def by auto

98
lemma matrix-equality: \( FF = M * F \)

proof

let \(?P = M * F \)

have \( \text{dim: dim-row } FF = m \) \( \text{dim-col } FF = n \) \( \text{dim-row } ?P = m \) \( \text{dim-col } ?P = n \) \( \text{dim-row } M = m \) \( \text{dim-col } M = m \) \( \text{dim-row } Fs = m \) \( \text{dim-col } Fs = n \) 

by (auto simp: assm mat-of-rows-def mat-of-rows-list-def main-connect len-fs)

show \(?thesis \)

proof (rule eq-matI; unfold dim)

fix \( i \) \( j \)

assume \( i : i < m \) and \( j : j < n \)

from \( i \) have split: \([0..<m] = [0..<i] @ [i] @ [Suc i..<m] \)

by (metis append-Cons append-self-conv2 less-Suc-eq-le less-imp-add-positive upt-add-eq-append upt-rec zero-less-Suc)

let \(?prod = \lambda k. \mu \cdot i \cdot k * gso \cdot k \cdot j \)

have \( \text{dim2: dim-vec (col Fs j)} = m \) using \( j \) \( \text{dim} \) by auto

define \( idx \) where \( idx = [0..<i] \)

have \( \text{idx: set idx \subseteq \{0..<i\}} \)

unfolding idx-def using \( i \) by auto

let \(?vec = \text{sumlist (map (\lambda j. - \mu \cdot i \cdot j \cdot v \cdot gso \cdot j \cdot idx)} \)

have \( \text{vec: vec \in carrier-vec } n \) by (rule sumlist-carrier, insert \( idx \) \( gso \cdot carrier \) \( i, \text{auto} \))

hence \( \text{dimv: dim-vec } \text{vec=} n \) by auto

have \(?P \vdash \text{row } M \cdot i \cdot \text{col } Fs \cdot j \) using \( i \cdot j \) by auto

also have \( \ldots = (\sum k = 0..<m. \text{row } M \cdot i \cdot k * \text{col } Fs \cdot j \cdot k) \)

unfolding scalar-prod-def dim2 by auto

also have \( \ldots = (\sum k = 0..<m. \text{?prod } k) \)

by (rule sum.cong[OF refl], insert \( i \cdot j \) \( \text{dim} \), auto simp: mat-of-rows-list-def mat-of-rows-def main-connect(2))

also have \( \ldots = \text{sum-list (map } \text{?prod } \{0..<m\}) \)

by (subst sum-list-distinct-conv-sum-set, auto)

also have \( \ldots = \text{sum-list (map } \text{?prod } \text{idx} + \text{?prod } i + \sum-list (\text{map } \text{?prod} (\{\text{Suc } i..<m\})) \)

unfolding split idx-def by auto

also have \( \text{?prod } i = \text{gso } i \cdot j \) unfolding \( \mu \cdot \text{simps} \) by simp

also have \( \ldots = \text{fs! i } j + \text{sum-list (map (\lambda k. - \mu \cdot i \cdot k * \text{gso } k \cdot j \cdot idx)} \)

unfolding gso.simps[of \( i \) idx-def[ symmetric ]]

by (subst index-add-vec, unfold dimv, rule \( j \), subst sumlist-vec-index[OF 'j], insert \( \text{gso-carrier } i, j \),

auto simp: o-def intro!, sum-list-cong)

also have \( \text{sum-list (map (\lambda k. - \mu \cdot i \cdot k * \text{gso } k \cdot j \cdot idx)} = - \text{sum-list (map (\lambda k. - \mu \cdot i \cdot k * \text{gso } k \cdot j \cdot idx)} \)

by (induct \( \text{idx} \), auto)

also have \( \text{sum-list (map } \text{?prod [Suc } i..<m\}) = 0 \)

by (rule sum-list-neutral, auto simp: \( \mu \cdot \text{simps} \))

finally have \(?P \vdash \text{fs! i } j \) by simp

with \( FF \cdot \text{index}[OF 'i 'j] \)

show \(?P \vdash \text{fs! i } j \) by simp

99
-proof -
  let \( l = \text{sumlist} (\text{map} (\lambda j. \mu j \cdot gso j) [0 ..< \text{Suc} i]) \)
  have \( l \in \text{carrier-vec} n \) by (rule \text{sumlist-carrier}, insert \text{gso-carrier} i, auto)
  hence \( \text{dim} \cdot \text{dim-vec} l = n \) by (rule \text{carrier-vecD})
  show \( \text{thesis} \)
  proof (rule \text{eq-vecI}, unfold \text{dim} \cdot \text{f-dim}[OF i])
  fix \( j \)
  assume \( j : j < n \)
  from \( i \) have \( \text{split}: [0 ..< m] = [0 ..< \text{Suc} i] @ [\text{Suc} i ..< m] \)
  by (metis Suc-lessI append.assoc append-same-eq less-imp-add-positive order-refl)
  let \( ?l = \text{sumlist} (\text{map} (\lambda j. \mu i j \cdot v gso j) [0 ..< \text{Suc} i]) \)
  have \( ?l \in \text{carrier-vec} n \) by (rule \text{sumlist-carrier}, insert \text{gso-carrier} i, auto)
  hence \( \text{dim} \cdot \text{dim-vec} ?l = n \) by (rule \text{carrier-vecD})
  show \( \text{thesis} \)
  proof (rule \text{eq-vecI}, unfold \text{dim} \cdot \text{f-dim}[OF i])
  fix \( j \)
  assume \( j : j < n \)
  from \( i \) have \( \text{split}: [0 ..< m] = [0 ..< \text{Suc} i] @ [\text{Suc} i ..< m] \)
  by (metis Suc-lessI append.assoc append-same-eq less-imp-add-positive order-refl)
  let \( ?prod = \lambda k. \mu i k \cdot gso k \cdot j \)
  have \( \text{fs} ! i \cdot j = \text{FF} (i, j) \) using \( i j \) by simp
  also have \( \ldots = \text{row} M i \cdot \text{col} F s j \cdot \text{FS} \cdot \text{SUC} j k \)
  unfolding \text{scalar-prod-def} by (auto simp: \text{main-connect}(2))
  also have \( \ldots = \text{sum-list} (\text{map} ?prod [0 ..< m]) \)
  unfolding \text{split} by auto
  also have \( \text{set} \cdot \text{vs} = \text{carrier-vec} n \cdot \text{distinct} \cdot \text{vs} \cdot \text{orthogonal} \cdot \text{rev} \cdot \text{vs} \)
  unfolding \text{split} by auto
  also have \( \text{span} \cdot \text{(set} \cdot \text{fs}) = \text{span} \cdot \text{(set} \cdot \text{vs}) \cdot \text{length} \cdot \text{vs} = \text{length} \cdot \text{fs} \)
  from \( \text{main-connect}(1)[\text{unfolded} \cdot \text{gram-schmidt-def}] \) have \( e: \text{gram-schmidt-sub} n \) \[ \]
  \( \text{fs} = \text{rev} \cdot \text{vs} \cdot \text{by auto} \)
  have \( \text{set} \cdot \text{vs} = \text{carrier-vec} n \cdot \text{distinct} \cdot \text{(set} \cdot \text{fs}) \cdot \text{lin-indpt} \cdot \text{(set} \cdot \text{(set} \cdot \text{fs})) \)
  unfolding \text{orthogonal} \cdot \text{by auto}
  from \( \text{cof-vec-space.gram-schmidt-sub-result}[\text{OF} \cdot e \cdot \text{assm}(2) \cdot \text{this}] \)
show set vs ⊆ carrier-vec n distinct vs corthogonal (rev vs)
span (set fs) = span (set vs) length vs = length fs
by auto
qed

lemma gso-inj[intro]:
shows i < m ⇒ inj-on gso {0..<i}
proof
  fix x y assume assms:i < m x ∈ {0..<i} y ∈ {0..<i} gso x = gso y
  have distinct vs x < length vs y < length vs using vs assms assm by auto
  from nth-eq-iff-index-eq[OF this] assms main-connect show x = y by auto
qed

lemmas gram-schmidt = cof-vec-space.gram-schmidt-result[OF fs-carrier dist lin-indpt
main-connect(1)[symmetric]]

lemma partial-span: assumes i: i ≤ m shows span (gso ' {0..<i}) = span (set (take i fs))
proof
  let ?f = λ i. fs ! i
  let ?us = take i fs
  have len: length ?us = i using len-fs i by auto
  from fs-carrier len-fs i have us: set ?us ⊆ carrier-vec n
    by (meson set-take-subset subset-trans)
  obtain vi where main: snd (main ?us) = vi by force
  from dist have dist: distinct ?us by auto
  from lin-indpt have indpt: lin-indpt (set ?us)
    using supset-ld-is-ld[of set ?us, of set (?us @ drop i fs)]
    by (auto simp: set-take-subset)
  from partial-connect[OF len-fs i nn us main refl fs-carrier]
  have gso: vi = gram-schmidt n ?us and vi: vi = map gso [0..<i] by auto
  from cof-vec-space.gram-schmidt-result(1)[OF us dist indpt gso, unfolded vi]
  show ?thesis by auto
qed

lemma partial-span': assumes i: i ≤ m shows span (gso ' {0..<i}) = span ((λ j. fs ! j) ' {0..<i})
unfolding partial-span[OF i]
by (rule arg-cong[of - - span], subst nth-image, insert i len-fs, auto)

lemma det: assumes m: m = n shows det FF = det Fs
unfolding matrix-equality
apply (subt det-mult[OF M-dim(3)], (insert Fs-dim(3) m, auto)[1])
apply (subt det-lower-triangular[OF - M-dim(3)])
by (subt M-index, (auto simp: µ.simps)[3], unfold prod-list-diag-prod, auto simp: µ.simps)

101
lemma orthogonal: $i < m \implies j < m \implies i \neq j \implies gso\ i \cdot gso\ j = 0$
using gram-schmidt(2)[unfolded main-connect corthogonal-def] by auto

lemma same-base: span $\{set\ fs\} = span \{gso\ \{0..<m\}\}$
using gram-schmidt(1)[unfolded - main-connect] by auto

lemma sq-norm-gso-le-f: assumes $i < m$
shows $sq-norm\ (gso\ i) \leq sq-norm\ (fs!i)$
proof –
  have $\{0..<Suc\ i\} = [\{0..<i\} @ [i]]$ by simp
  let $\text{sum} = \text{sumlist}\ (\text{map}\ (\lambda j.\ \mu\ i\ j\ \cdot\ v\ gso\ j)\ [\{0..<i\}])$
  have $\text{sum} \in\ \text{carrier-vec}\ n$ and $\text{gso}:: gso\ i \in\ \text{carrier-vec}\ n$ using i
    by (auto intro!: sumlist-carrier gso-carrier)
  from inl-is-sum-of-mu-gso[OF i, unfolded id]
  have $sq-norm\ (fs!i) = sq-norm\ \text{(sumlist}\ (\text{map}\ (\lambda j.\ \mu\ i\ j\ \cdot\ v\ gso\ j)\ [\{0..<i\}]\ @ [gso\ i]))$ by (simp add: simp+)
  also have $\ldots = sq-norm\ (?\text{sum}\ +\ gso\ i)$
    by (subst sumlist-append, insert gso-carrier i, auto)
  also have $\ldots = (?\text{sum} + gso\ i)\cdot (?\text{sum} + gso\ i)\ by\ (simp\ add:\ sq-norm-vec-as-cscalar-prod)$
  also have $\ldots = sq-norm\ \text{of-vec}\ gso\ [OF\ i,\ unfolded\ id]$ using i
    by (simp add: sq-norm-vec-as-cscalar-prod)
  finally have $sq-norm\ (fs!i) = sq-norm\ \text{of}\ ?\text{sum} + 2\cdot (?\text{sum} + gso\ i) + sq-norm\ (gso\ i)$ by simp
  also have $\ldots \geq 2\cdot (?\text{sum} + gso\ i) + sq-norm\ (gso\ i)$ using sq-norm-vec-ge-0[of ?sum] by simp
  also have $\text{add} \cdot gso\ i = \text{of}\ gso\ i\ by\ (simp\ add:\ sq-norm-vec-as-cscalar-prod)$
  also have $gso\ i\ \cdot\ gso\ i = sq-norm\ (gso\ i)$ by (simp add: sq-norm-vec-as-cscalar-prod)
  also have $gso\ i\ \cdot\ ?\text{sum} = \text{add}\ gso\ i\ using\ gsoi\ gsoi\ by\ (simp\ add:\ sq-norm-vec-as-cscalar-prod)$
  also have $\ldots = 0$
proof (rule sum-list-neutral, goal-cases)
  case (1 x)
    then obtain $j$ where $j < i$ and $x: x = (\mu\ i\ j\ \cdot\ v\ gso\ j)\ \cdot\ gso\ i$ by auto
    from $j\ i$ have $gsoj:: gso\ j \in\ \text{carrier-vec}\ n$ by auto
    have $x = \mu\ i\ j\ \cdot\ gso\ j\ \cdot\ gso\ i$ using gsoi gsoj unfolding x by simp
    also have $gso\ j\ \cdot\ gso\ i = 0$
      by (rule orthogonal, insert $j\ i$, auto)
    finally show $x = 0 \ by\ simp$
qed
finally show $\text{thesis}\ by\ simp$
qed
lemma projection-exist:
  assumes i < m
  shows \( ?A : gso \{ \{0..<i\}\} \)
proof
  let \( ?A = gso \{ \{0..<i\}\} \)
  show isFinite ?A by auto
  have carA[intro]:\( \{\} \subseteq carrier-vec n \) using gso-dim assms by auto
  let \( ?a = \sum \{i \mid \text{if } gso n \text{ then } i \in n \text{ else } 0 \} \)
  have \( d:(\sum \text{ OF assms}) \in carrier-vec n \)
  have [intro]: \( (\lambda v. ?a \cdot v \cdot v) \in gso \{ \{0..<i\}\} \rightarrow carrier-vec n \)
    using gso-carrier assms by auto
  { fix ia assume ia[intro]:ia < n
    have \( (\sum x \in gso \{ \{0..<i\}\} \cdot \text{if } gso n \text{ then } i \in n \text{ else } 0) \cdot x \subseteq ia \)
      using is_finite[OF assms] by auto
    unfolding sum-map comm-monoid-add-class sum_list_nat uminus_sum_list_map o_def
    proof (rule sum-list-cong[OF refl],goal_cases)
      case (1 x) hence \( x: x < m \rightarrow \) using assms by auto
      hence \( d: insert x \text{ set } \{0..<i\} = \{0..<i\} \)
        count \( \text{ mset } \{0..<i\} \cdot x = 1 \) by auto
      hence inj-on gso (insert x (\{0..<i\})) \text{ using gso-inj[OF assms] by auto }
      from inj-on-filter-key-eq[OF assms] unfolding sum_list_nat uminus_sum_list_map o_def
      have \( \text{ if } gso n \text{ then } i \in n \text{ else } 0 \cdot x \in gso n \) \( = \) \( \{0..<i\} \cdot x \in \) using x assms d replicate.simps(2)[of 0]
        by auto
      hence \( (\sum x \in gso \{ \{0..<i\}\} \cdot \text{if } gso n \text{ then } i \in n \text{ else } 0) \cdot x \subseteq \{0..<i\} \)
        unfolding sum_list_nat uminus_sum_list_map o_def
        by auto
      with ia gso-dim x show \( \text{ case apply(subst index-smult-vec) by force+ } \)
        unrefined
      hence \( \sum x \in gso \{ \{0..<i\}\} \cdot \text{if } gso n \text{ then } i \in n \text{ else } 0 = \) \( \mu \cdot x \)
        unfolding sum_list_nat uminus_sum_list_map o_def
        by auto
      with ic local simp
      apply (subst (1 0) finsum-index index-uminus-vec) apply force+
      apply (subst sum_list_nat uminus-vec) by auto
    qed
  }
  hence id: \( \bigoplus v \in gso \{ \{0..<i\}\} \cdot \text{if } gso n \text{ then } i \in n \text{ else } 0 \cdot x \in gso n \) \( = \) \( \sum \text{[OF assms]} \text{ map } \text{ map } \text{ OF assms} \)
    using d assms \( \text{ by intro eq-vecI,auto } \)
  show \( \text{ if } i = gso ?a : A \text{ unfolding lincomb-def gso.simps[of } \cdot \}
    \text{ by (rule eq-vecI, auto) } \)
  qed auto

lemma orthocompl-span:
  assumes \( \forall x. x \in S \implies v \cdot x = 0 \) \( S \subseteq carrier-vec n \) and [intro]: \( v \in carrier-vec n \)
  shows \( v \cdot y = 0 \)
proof –
{fix a A
  assume y = lincomb a A finite A A ⊆ S
  note assms = assms this
  hence [intro!]:lincomb a A ∈ carrier-vec n (λv. a v · v) ∈ A → carrier-vec n by auto
  have ∀x∈A. (a x · v) · v = 0 proof fix x assume x ∈ A note assms = assms this
    hence x:x ∈ S by auto
    with assms have [intro]:x ∈ carrier-vec n by auto
    from assms(1)(OF x) have x · v = 0 by(subst comm-scalar-prod) force+
    thus (a x · v) · v = 0
    apply(subst small-scalar-prod-distrib) by force+
  qed
  hence v · lincomb a A = 0 apply(subst comm-scalar-prod) apply force+
  unfolding lincomb-def
  apply(subst finsum-scalar-prod-sum) by force+
} thus ?thesis using v ∈ span S; unfolding span-def by auto
qed

lemma projection-unique:
  assumes i < m
  \[ \land x, x ∈ gso \cdot \{0..<i\} \quad \Rightarrow \quad v \cdot x = 0 \]
  shows v = gso i
proof -
  from assms have carr-span:span (gso \cdot \{0..<i\}) ⊆ carrier-vec n by(intro span-is-subset2) auto
  from assms have carr: gso \cdot \{0..<i\} ⊆ carrier-vec n by auto
  from assms have eq:fs ! i − (fs ! i − v) = v for v by auto
  from orthocompl-span[OF assms(3) carr assms(2)]
  have y ∈ span (gso \cdot \{0..<i\}) \quad \Rightarrow \quad v \cdot y = 0 \quad for \ y by auto
  hence oc1:fs ! i − (fs ! i − v) ∈ orthogonal-complement (span (gso \cdot \{0..< i\}))
  unfolding eq orthogonal-complement-def using assms by auto
  have x ∈ gso \cdot \{0..<i\} \quad \Rightarrow \quad gso i \cdot x = 0 \quad for \ x using assms orthogonal by auto
  hence y ∈ span (gso \cdot \{0..<i\}) \quad \Rightarrow \quad gso i \cdot y = 0 \quad for \ y
  by (rule orthocompl-span[OF carr gso-carrier[OF assms(1)],rule-format])
  hence oc2:fs ! i − (fs ! i − gso i) ∈ orthogonal-complement (span (gso \cdot \{0..<
  i\}))
  unfolding eq orthogonal-complement-def using assms by auto
  note pe = projection-exist[OF assms(1)]
  note prerec = carr-span f-carrier[OF assms(1)] assms(4) oc1 projection-exist[OF assms(1)] oc2
  have gsoi: gso i ∈ carrier-vec n fs ! i ∈ carrier-vec n by (rule gso-carrier[OF \ i < m], rule f-carrier[OF \ i < m])
  note main = arg-cong[OF projection-alt-def[OF carr-span f-carrier[OF assms(1)]]
  assms(4) oc1 pe oc2]
  of \ λ v. v \& j + fs ! i \& j for j}
show \( v = \text{gso} \, i \)

proof (intro eq-vecI)
  fix \( j \)
  show \( j < \text{dim-vec} (\text{gso} \, i) \implies v \not= j = \text{gso} \, i \, \not= j \)
  using assms(2) gsoi main[of \( j \)] by auto
qed (insert assms(2) gsoi, auto)

lemma gso-projection:
  assumes \( i < m \)
  shows \( \text{gso} \, i = \text{projection} (\text{gso} \cdot \{0..<i\}) \) (fs ! i)

proof (rule some-equality[\text{symmetric},OF - \text{projection-unique}[OF assms]])
  have orthogonal;\( \forall x. \, xa < i \implies \text{gso} \, i \cdot \text{gso} \, xa = 0 \) by (rule orthogonal,insert assms, auto)
qed auto

lemma gso-projection-span:
  assumes \( i < m \)
  shows \( \text{gso} \, i = \text{projection} (\text{span} (\text{gso} \cdot \{0..<i\})) \) (fs ! i)
  and \( \text{is-projection} (\text{gso} \, i) \) (span (gso \cdot \{0..<i\})) (fs ! i)

proof (rule some-equality[\text{symmetric},OF - \text{projection-unique}[OF assms]])
  let \( \exists P \, v = v \in \text{carrier-vec} \, n \land \, \text{fs} ! i \land v \in \text{span} (\text{span} (\text{gso} \cdot \{0..<i\})) \)
  \land (\forall x. \, x \in \text{span} (\text{gso} \cdot \{0..<i\}) \implies v \cdot x = 0)
  have carr: \( \text{gso} \cdot \{0..<i\} \subseteq \text{carrier-vec} \, n \) using assms by auto
  have \( \forall x. \, xa < i \implies \text{gso} \, i \cdot \text{gso} \, xa = 0 \) by (rule orthogonal,insert assms, auto)

  have orthogonal;\( \forall x. \, x \in \text{span} (\text{gso} \cdot \{0..<i\}) \implies \text{gso} \, i \cdot x = 0 \)
  apply (rule orthocompl-span) using assms * by auto
  show \( \exists P \, (\text{gso} \, i) \) \( \exists P \, (\text{gso} \, i) \) unfolding span-span[OF carr]
  using gso-carrier[OF assms] projection-exist[OF assms] orthogonal by auto
  fix \( v \) assume \( p; \exists P \, v \)
  then show \( v \in \text{carrier-vec} \, n \) by auto
  from \( p \) show \( \text{fs} ! i \land v \in \text{span} (\text{gso} \cdot \{0..<i\}) \) unfolding span-span[OF carr]
  by auto
  fix \( xa \) assume \( xa \in \text{gso} \cdot \{0..<i\} \)
  hence \( xa \in \text{span} (\text{gso} \cdot \{0..<i\}) \) using in-own-span[OF carr] by auto
  thus \( v \cdot xa = 0 \) using \( p \) by auto
qed

lemma is-projection-eq:
  assumes \( \text{ispr} : \text{is-projection} \, a \, S \, v \) \, is-projection \, b \, S \, v
  and \( \text{carr} : S \subseteq \text{carrier-vec} \, n \, \text{v} \in \text{carrier-vec} \, n \)
  shows \( a = b \)
proof –
from carr have c2: span S ⊆ carrier-vec n v ∈ carrier-vec n by auto
have a:v = (v - a) using carr ispr by auto
have b:v = (v - b) = b using carr ispr by auto
have (v - a) = (v - b)
  apply (rule projection-alt-def[OF c2])
  using ispr a b unfolding in-orthogonal-complement-span[OF carr(1)]
  unfolding orthogonal-complement-def is-projection-def by auto
hence v - (v - a) = v - (v - b) by metis
thus thesis unfolding a b.
qed

lemma scalar-prod-lincomb-gso: assumes k: k ≤ m
shows sumlist (map (λ i. g i · v gso i) [0..<k]) · sumlist (map (λ i. g i · v gso i) [0..<k])
  = sum-list (map (λ i. g i * g i * (gso i · gso i)) [0..<k])
proof –
  have id1: (map (λ i. g i · v gso i) [0..<k]) · sumlist (map (λ i. g i · v gso i) [0..<k] =
    map (λ i. g i · v gso i) [0..<k] using k by auto
  also have 0 = M · sumlist (map j a · v gso j ·) [0..<j] · c gso j using k
    by (intro sum-list-0[symmetric], auto)
  finally show thesis unfolding sum-list-carrier using k
qed

lemma gso-times-self-is-norm:
assumes j < m shows fs ! · j · gso j = sq-norm-vec (gso j) (is ?lhs = ?rhs)
proof –
  have ?lhs = fs ! · c gso j + 0 by auto
  also have 0 = M · sumlist (map λja. - μ j ja · v gso ja) [0..<j] · c gso j using assms orthogonal
    apply (subst scalar-prod-left-sum-distrib,force)
    by (intro sum-list-0[symmetric],auto)
  finally show thesis unfolding sq-norm-vec-as-cscalar-prod vec-conjugate-rat using assms
    apply (subst (2) gso.simps)
    apply (subst add-scalar-prod-distrib[OF f-carrier M.sumlist-carrier])
    by auto
qed

lemma gram-schmidt-short-vector: assumes in-L: h ∈ lattice-of fs - {0, n}
shows $\exists \ i < m. \ ||h||^2 \geq ||gso \ i||^2$

proof –

from in-L have non-0; $h \neq 0_v n$ by auto

from in-L[unfolded lattice-of-def] obtain lam where

$h \cdot h = \text{sumlist} (\text{map} (\lambda \ i. \ \text{of-int} (\text{lam} \ i) \ \cdot v \ \text{fs} \ i) [0 ..< \text{length} \ h])$

by auto

have in-L: $h = \text{sumlist} (\text{map} (\lambda \ i. \ \text{of-int} (\text{lam} \ i) \ \cdot v \ \text{fs} \ i) [0 ..< m])$ unfolding assm length-map h

by (rule arg-cong[of - sumlist], rule map-cong, auto simp: len-fs)

let $?n = [0 ..< m]$

let $?f = (\lambda i. \ \text{of-int} (\text{lam} \ i) \ \cdot v \ \text{fs} \ i)$

let $?vs = \text{map} ?f \ ?n$

let $?P = \lambda k. k < m \land \text{lam} \ k \neq 0$

define $k$ where $k = (\text{GREATEST} \ kk. \ ?P \ kk)$

{ assumption $\forall \ i < m. \ \text{lam} \ i = 0$

have $?vs = \text{map} (\lambda i. \ 0_v n) \ ?n$

by (rule map-cong, insert f-dim $\ast$, auto)

have $h = 0_v n$ unfolding in-L \ vs

by (rule sumlist-neutral, auto)

with non-0 have False by auto
}

then obtain kk where $?P \ kk$ by auto

from GreatestI-nat[of $?P$, OF this, of m] have Pk: $?P \ k$ unfolding k-def by auto

hence kn: $k < m$ by auto

let $?gso = (\lambda i \ j. \ \mu \ i \ j \ \cdot v \ \text{gso} \ j)$

have $k: k < i \implies i < m \implies \text{lam} \ i = 0$ for $i$

using Greatest-le-nat[of $?P \ i \ m$, folded k-def] by auto

define l where $l = \text{lam} \ k$

from Pk have l: $l \neq 0$ unfolding l-def by auto

define idx where $idx = [0 ..< k]$

have $idex \cdot i < k \implies i < k \land \ i \in set idx \implies i < m$ unfolding idx-def using kn by auto

from Pk have split: $[0 ..< m] = idx \ @ [k] \ @ [\text{Suc} \ k ..< m]$ unfolding idx-def

by (metis append-Cons append-self-conv2 less-Suc-le less-imp-add-positive
upt-add-eq-append
upt-rec zero-less-Suc)

define gg where $gg = \text{sumlist}$

$(\text{map} (\lambda i. \ \text{of-int} (\text{lam} \ i) \ \cdot v \ \text{fs} \ i) \ idex) + \ \text{of-int} \ l \ \cdot v \ \text{sumlist} (\text{map} (\lambda j. \ \mu \ k \ j \ \cdot v \ \text{gso} \ j) \ idex)$

have $h = \text{sumlist} \ \vdash vs$ unfolding in-L ...

also have $\vdash \vdash \text{sumlist} (\text{map} \ ?f \ idex \ @ [?f \ k]) \ @ \text{map} ?f \ [\text{Suc} \ k ..< m])$

unfolding split by auto

also have $\vdash \vdash \text{sumlist} (\text{map} \ ?f \ idex \ @ [?f \ k]) + \text{sumlist} (\text{map} ?f \ [\text{Suc} \ k ..< m])$

by (rule sumlist-append, auto intro!: f-carrier, insert Pk idex, auto)

also have $\text{sumlist} (\text{map} ?f \ [\text{Suc} \ k ..< m]) = 0_v n$ by (rule sumlist-neutral, auto simp: k)
also have \( \text{sumlist} \ (\text{map} \ ?f \ idx @ [\?f \ k]) = \text{sumlist} \ (\text{map} \ ?f \ idz) + ?f \ k \)
by (subst \text{sumlist-append}, auto intro!: \text{f-carrier}, insert Pk idz, auto)
also have \( fs ! k = \text{sumlist} \ (\text{map} \ (?gso \ k) \ [0..<\text{Suc} \ k]) \) \text{using} \( \text{fi-is-sum-of-mu-gso[OF kn]} \) by simp
also have \( \ldots = \text{sumlist} \ (\text{map} \ (?gso \ k) \ idx @ [gso \ k]) \) \text{by} (simp add: \( \mu \cdot \text{simps[of k \ k]} \) idx-def)
also have \( \ldots = \text{sumlist} \ (\text{map} \ (?gso \ k) \ idx) + gso \ k \)
by (subst \text{sumlist-append}, auto intro!: \text{f-carrier}, insert Pk idx, auto)
also have \( \text{of-int} \ (\text{lam} \ k) \cdot_v \ldots = \text{of-int} \ (\text{lam} \ k) \cdot_v \ (\text{sumlist} \ (\text{map} \ (?gso \ k) \ idx)) \)
+ \( \text{of-int} \ (\text{lam} \ k) \cdot_v \ gso \ k \)
\text{unfolding} \( \text{idx-def} \)
by (rule \text{smul-add-distrib-vec[OF sumlist-carrier]}, auto intro!: gso-carrier, insert kn, auto)
finally have \( h = \text{sumlist} \ (\text{map} \ ?f \ idx) + \)
\( \text{(of-int} \ (\text{lam} \ k) \cdot_v \text{sumlist} \ (\text{map} \ (?gso \ k) \ idx) + \text{of-int} \ (\text{lam} \ k) \cdot_v \ gso \ k) + \)
\( 0 \cdot n \) \text{by simp}
also have \( \ldots = \text{gg + of-int} \ l \cdot_v \ gso \ k \) \text{unfolding} \( \text{gg-def l-def} \)
by (rule \text{eq-vecI}, insert idx kn, auto simp: \text{sumlist-vec-index},
subst \text{index-add-vec}, auto simp: \text{sumlist-dim kn}, subst \text{sumlist-dim, auto})
finally have \( \text{hgg} \cdot h = \text{gg + of-int} \ l \cdot_v \ gso \ k \cdot \)
let \( \mathbb{R} = \{ \text{gg .} \exists \ nu. \ \text{gg = sumlist} \ (\text{map} \ (\lambda \ i. \ nu \ i \cdot_v \ gso \ i) \ idx) \} \)
\{ \text{fix} \ nu \}
\text{have} \( \text{dim-vec} \ (\text{sumlist} \ (\text{map} \ (\lambda \ i. \ nu \ i \cdot_v \ gso \ i) \ idx)) = n \)
by (rule \text{sumlist-dim, insert kn, auto simp: idx-def})
\}
\text{note} \( \text{dim-nu[simp] = this} \)
define \( kk \) \text{where} \( kk = ?k \)
\{ \text{fix} \ v \}
\text{assume} \( v \in \mathbb{R} \)
then obtain \( \text{nu where} \ v: v = \text{sumlist} \ (\text{map} \ (\lambda \ i. \ nu \ i \cdot_v \ gso \ i) \ idx) \) \text{unfolding} \( \text{R-def by auto} \)
\text{have} \( \text{dim-vec} \ v = n \) \text{unfolding} \( \text{gg-def v by simp} \)
\}
\text{note} \( \text{dim-R = this} \)
\{ \text{fix} \ v1 \ v2 \}
\text{assume} \( v1 \in \mathbb{R} v2 \in \mathbb{R} \)
then obtain \( \text{nu1 nu2 where} \ v1: v1 = \text{sumlist} \ (\text{map} \ (\lambda \ i. \ nu1 \ i \cdot_v \ gso \ i) \ idx) \)
and 
\( v2: v2 = \text{sumlist} \ (\text{map} \ (\lambda \ i. \ nu2 \ i \cdot_v \ gso \ i) \ idx) \)
\text{unfolding} \( \text{R-def by auto} \)
\text{have} \( v1 + v2 \in \mathbb{R} \) \text{unfolding} \( \text{R-def} \)
by (standard, rule \text{exI[of - \lambda \ i. \ nu1 \ i + nu2 \ i]}, \text{unfold} \ v1 \ v2, \text{rule eq-vecI},
(subst \text{sumlist-vec-index, insert idz, auto intro!: gso-carrier simp: o-def}+, \text{unfold} \ \text{sum-list-addf[symmetric], induct idz, auto simp: algebra-simps})
\}
\text{note} \( \text{add-R = this} \)
\text{have} \( \text{gg \in} \mathbb{R} \) \text{unfolding} \( \text{gg-def} \)
\text{proof (rule} \text{add-R})
show of-int l  \cdot v sumlist (map (\lambda j. \mu k j  \cdot v gso j) idx) \in R

unfolding R-def
by (standard, rule exI[of - \lambda i. of-int l * \mu k i], rule eq-vecI,
(subst sumlist-vec-index, insert idx, auto intro!: gso-carrier simp: o-def)+,
induct idx, auto simp: algebra-simps)

show sumlist (map \?f idx) \in R using idx

proof (induct idx)

case Nil
show \?case by (simp add: R-def, intro exI[of - \lambda i. 0], rule eq-vecI,
(subst sumlist-vec-index, insert idx, auto intro!: gso-carrier simp: o-def)+,
induct idx, auto)

next
case (Cons i idxs)
have sumlist (map \?f (i # idxs)) = sumlist ([\?f i] @ map \?f idxs) by simp
also have \ldots = \?f i + sumlist (map \?f idxs)
by (subst sumlist-append, insert Cons(3), auto intro!: f-carrier)
finally have id: sumlist (map \?f (i # idxs)) = \?f i + sumlist (map \?f idxs)

show \?case unfolding id

proof (rule add-R[OF Cons(1)[OF Cons(2-3)]]
from Cons(2-3) have i: i < m i < k by auto
hence idx-split: idx = [0 ..< Suc i] @ [Suc i ..< k] unfolding idx-def
by (metis Suc-lessI append-Nil2 less-imp-add-positive upt-add-eq-append upt-rec zero-le)
{
fix j
assume j: j < n

define idxs where idxs = [0 ..< Suc i]
let \?f = \lambda x. ((if x < Suc i then of-int (lam i) * \mu i x else 0)  \cdot v gso x) $ j
have (\sum x \leftarrow idxs. \?f x) = (\sum x \leftarrow [0 ..< Suc i]. \?f x) + (\sum x \leftarrow [Suc i ..< k]. \?f x)

unfolding idx-split by auto
also have (\sum x \leftarrow [Suc i ..< k]. \?f x) = 0 by (rule sum-list-neutral, insert j kn, auto)
also have (\sum x \leftarrow [0 ..< Suc i]. \?f x) = (\sum x \leftarrow idxs. of-int (lam i) * (\mu i x \cdot v, gso x) $ j)

unfolding idxs-def by (rule arg-cong[of - sum-list], rule map-cong[OF refl],
subst index-smult-vec, insert j i kn, auto)
also have \ldots = of-int (lam i) * ((\sum x \leftarrow [0..<Suc i]. (\mu i x \cdot v gso x) $ j))

unfolding idxs-def[symmetric] by (induct idxs, auto simp: algebra-simps)
finally have (\sum x \leftarrow idxs. \?f x) = of-int (lam i) * ((\sum x \leftarrow [0..<Suc i]. (\mu i x \cdot v, gso x) $ j))
by simp
}

note main = this

show \?fi \in R unfolding ft-is-sum-of-mu-gso[OF i(1)] R-def
apply (standard, rule exI[of - \lambda j. if j < Suc i then of-int (lam i) * \mu i j else 0], rule eq-vecI)

109
apply (subst sumlist-vec-index, insert idx i, auto intro!: gso-carrier sumlist-dim simp: o-def)
  apply ( subst index-smult-vec, subst sumlist-dim, auto)
  apply ( subst sumlist-vec-index, auto, insert idx i main, auto simp: o-def)
  done
  qed auto
  qed
  qed
then obtain nu where gg: gg = sumlist (map (λ i. nu i ·v gso i) idx) unfolding R-def by auto
  let ?? = sumlist (map (λ i. nu i ·v gso i) idx) + of-int l ·v gso k
  define hh where hh = (λ i. (if i < k then nu i else of-int l))
  let ?hh = sumlist (map (λ i. hh i ·v gso i) [0..< Suc k])
  have ffhh: ?hh = sumlist (map (λ i. hh i ·v gso i) [0..< k] @ [hh k ·v gso k])
  by simp
  also have ... = sumlist (map (λ i. hh i ·v gso i) [0..< k]) + sumlist [hh k ·v gso k]
  by (rule sumlist-append, insert kn, auto)
  also have sumlist [hh k ·v gso k] = hh k ·v gso k using kn by auto
  also have ... = of-int l ·v gso k unfolding hh-def by auto
  also have map (λ i. hh i ·v gso i) [0..< k] = map (λ i. nu i ·v gso i) [0..< k]
  by (rule map-cong, auto simp: hh-def)
finally have ffhh: ?? = ?? by (simp add: idx-def)
from hgg[unfolded gg]
  have h: h = ?? by auto
  have gso k · gso k ≤ 1 *(gso k · gso k) by simp
  also have ... ≤ of-int (l + l) * (gso k · gso k)
  proof (rule mult-right-mono)
  from l have l + l ≥ 1 by (meson eq-iff int-one-le-iff-zero-less mult-le-0-iff not-le)
    thus 1 ≤ (of-int (l + l) :: 'a) by presburger
  show 0 ≤ gso k · gso k by (rule scalar-prod-ge-0)
  qed
  also have ... = 0 + of-int (l + l) * (gso k · gso k) by simp
  also have ... ≤ sum-list (map (λ i. (nu i * nu i) * (gso i · gso i)) idx) + of-int
(l + l) * (gso k · gso k)
  by (rule add-right-mono, rule sum-list-nonneg, auto, rule mult-nonneg-nonneg, auto simp: scalar-prod-ge-0)
  also have map (λ i. (nu i * nu i) * (gso i · gso i)) idx = map (λ i. hh i * hh i
* (gso i · gso i)) [0..<k]
  unfolding idx-def by (rule map-cong, auto simp: hh-def)
  also have of-int (l + l) = hh k · hh k unfolding hh-def by auto
  also have (∑ i∈[0..<k]. hh i * hh i * (gso i · gso i)) + hh k · hh k * (gso k ·
gso k) = (∑ i∈[0..<Suc k]. hh i * hh i * (gso i · gso i)) by simp
  also have ... = ?? · ?? by (rule sym, rule scalar-prod-lincomb-gso, insert kn, auto)
  also have ... = ?? · ?? by (simp add: ffhh)
  also have ... = h · h unfolding h ..
finally show \( ? \)thesis using \( kn \) unfolding \( sq\text{-}norm\text{-}vec\text{-}as\text{-}cscalar\text{-}prod \) by auto

qed

lemma \( fs0\text{-}gso0 \): \( 0 < m \Rightarrow fs ! 0 = gso 0 \)

unfolding \( gso\text{-}simps[0] \) using \( f\text{-}dim[0] \) \( \text{arg\text{-}cong[OF \ assm(3), \ of \ hd]} \) \( \text{len}\text{-}fs \)

by (cases \( fs \), auto simp add: upt-rec)

lemma weakly-reduced-imp-short-vector: assumes weakly-reduced \( \alpha \) \( m \) \( vs \)

and \( \text{in}\text{-}L: \( h \in \text{lattice}\text{-}of \) \( fs \) \( \{0 \cdot n\} \) \( \text{and} \) \( \alpha\text{-pos}\alpha \geq 1 \)

shows \( fs \neq [] \land sq\text{-}norm \) \( (fs ! 0) \leq \alpha^\cdot(m-1) \ast sq\text{-}norm h \)

proof —

from \( \text{gram}\text{-}schmidt\text{-}short\text{-}vector[OF \ in\text{-}L] \) obtain \( i \) where

\( i: i < m \) \( \text{and} \) \( le: sq\text{-}norm \) \( (gso \ i) \leq sq\text{-}norm h \) by auto

have small: \( sq\text{-}norm \) \( (fs ! 0) \leq \alpha^\cdot i \ast sq\text{-}norm \) \( (gso \ i) \) using \( i \)

proof (induct \( i \))

case \( 0 \)

show \( ? \)case unfolding \( fs0\text{-}gso0[OF \ 0] \) by auto

next

case (Suc \( i \))

hence \( sq\text{-}norm \) \( (fs ! 0) \leq \alpha^\cdot i \ast sq\text{-}norm \) \( (gso \ i) \) by auto

also have \( \ldots \leq \alpha^\cdot i \ast (\alpha \ast (sq\text{-}norm \) \( (gso \ (Suc \ i)))) \)

using \( \text{reduced}\text{-}vs\text{-}E[OF \ assms(1) \ \text{le\text{-}refl \ Suc(2)]} \) \( \alpha\text{-pos} \) by auto

finally show \( ? \)case unfolding \( \text{class}\text{-}semiring}\text{-}nat\text{-}pow\text{-}Suc[of \ \alpha \ i] \) by auto

qed

also have \( \ldots \leq \alpha^\cdot(m-1) \ast sq\text{-}norm h \)

by (rule \( \text{mult}\text{-}mono[OF \ power\text{-}increasing \ le], \ insert \ i \ \alpha\text{-pos}, \ auto) \)

finally show \( ? \)thesis using \( i \ \text{assm} \) by (cases \( fs \), auto)

qed

lemma sq-norm-pos: assumes \( j: j < m \)

shows \( sq\text{-}norm \) \( (vs ! j) > 0 \)

proof —

have \( len\text{-}vs: length \) \( vs = m \) using \( \text{main}\text{-}connect(2) \) by simp

have \( \text{corthogonal} \text{ vs by} (\text{rule gram}\text{-}schmidt) \)

from \( \text{corthogonalD[OF this, \ unfolded \ len\text{-}vs, \ OF \ j \ j]} \)

have \( sq\text{-}norm \) \( (vs ! j) \neq 0 \) by (simp add: \( sq\text{-}norm\text{-}vec\text{-}as\text{-}cscalar\text{-}prod \))

moreover have \( sq\text{-}norm \) \( (vs ! j) \geq 0 \) by auto

ultimately show \( 0 < sq\text{-}norm \) \( (vs ! j) \) by auto

qed

lemma Gramian-determinant: assumes \( k: k \leq m \)

shows \( \text{Gramian}\text{-}determinant} \) \( fs \ k = (\prod \ j<k. \ sq\text{-}norm \) \( (vs \ ! j)) \)

Gramian-determinant \( fs \ k > 0 \)

proof —

define \( Gk \) where \( Gk = \text{mat} \ k \ n \ (\lambda (i,j). \ fs ! i \& j) \)

have \( Gk: Gk \in \text{carrier}\text{-}mat \ k \ n \) unfolding \( Gk\text{-}def \) by auto

define \( Mk \) where \( Mk = \text{mat} \ k \ (\lambda (i,j). \ \mu i j) \)

111
have $Mk : i < k \implies j < k \implies Mk \$(i.j) = \mu \ i \ j \ \text{for} \ i \ j$

unfolding $Mk$-def using $k$ by auto

have $Mk : Mk \in \text{carrier-mat } k \times k$ and [simp]: $\text{dim-row } Mk = k \ \text{dim-col } Mk = k$

unfolding $Mk$-def by auto

have $\det Mk = \text{prod-list } (\text{diag-mat } Mk)$

by (rule $\det$-lower-triangular[OF - $Mk$], auto simp: $Mk$-$\mu$ $\mu$.sims)

also have $\ldots = 1$

by (rule $\text{prod-list-neutral}$, auto simp: $\text{diag-mat-def } Mk$-$\mu$. $\mu$.sims)

finally have $\det Mk : \det Mk = 1$.

define $Gsk$ where $Gsk = \text{mat } k \times n \ (\lambda \ (i.j). \ vs \ ! i \ $ j)$

have $Gsk : Gsk \in \text{carrier-mat } k \times n$ unfolding $Gsk$-def by auto

have $Gsk : Gsk^T \in \text{carrier-mat } n \times k$ using $Gsk$ by auto

have $\text{len-vs} : \text{length } vs = m$ using $\text{main-connect}(2)$ by simp

let $?Rn = \text{carrier-vec } n$

have $\text{id} : Gk = Mk * Gsk$

proof (rule eq-mat$I$

from $Gk \ Mk Gsk$

have $\text{dim} : \text{dim-row } Gk = k \ \text{dim-row } (Mk * Gsk) = k \ \text{dim-col } Gk = n \ \text{dim-col } (Mk * Gsk) = n$ by auto

from $\text{dim}$ show $\text{dim-row } Gk = \text{dim-row } (Mk * Gsk) \ \text{dim-col } Gk = \text{dim-col } (Mk * Gsk)$ by auto

fix $i \ j$

assume $i < \text{dim-row } (Mk * Gsk) \ j < \text{dim-col } (Mk * Gsk)$

hence $ij : i < k \ j < n$ and $i : i < m$ using $\text{dim } k$ by auto

have $Gi : \text{fs} ! i \in ?Rn \ \text{using } i \ \text{by simp}$

have $(Gk \$(i.j) = \text{fs} ! i \ $ j$)$ unfolding $Gk$-def using $i \ j \ k \ Gi$ by auto

also have $\ldots = \text{FF}$ $\$(i.j) \ \text{using } i \ \text{by simp}$

also have $\text{FF} = M * Fs$ by (rule matrix-equality)

also have $\ldots \$(i.j) = \text{row } M \ i \ \cdot \ \text{col } Fs \ j$

by (rule $\text{index-mult-mat}(1)$, insert $i \ ij$, auto simp: $\text{mat-of-rows-list-def}$)

also have $\text{row } M \ i = \text{vec } m \ (\lambda \ j. \ if \ j < k \ then \ Mk \$(i.j) \ else \ 0)$

(is $\ = \text{vec } m \ ?Mk$)

unfolding $Mk$-def using $ij$ i

by (auto simp: $\text{mat-of-rows-list-def}$ $\mu$.sims)

also have $\text{col } Fs \ j = \text{vec } m \ (\lambda \ i \ j'. \ if \ i' < k \ then \ Gsk \$(i'j) \ \text{else } (Fs \$$(i'j)))$

(is $\ = \text{vec } m \ ?Gsk$)

unfolding $Gsk$-def using $ij$ i len-vs by (auto simp: $\text{mat-of-rows-def}$)

also have $\text{vec } m \ \ ?Mk \ \cdot \ \text{vec } m \ ?Gsk = (\sum (i \in \{0 ..< k\}. \ ?Mk \ i \ ?Gsk \ i))$

unfolding scalar-prod-def by auto

also have $\ldots = (\sum (i \in \{0 ..< k\} \cup \ k ..< m). \ ?Mk \ i \ ?Gsk \ i)$

by (rule $\text{sum.cong}$, insert $k$, auto)

also have $\ldots = (\sum (i \in \{0 ..< k\} \cup \ k ..< m). \ ?Mk \ i \ ?Gsk \ i) + (\sum (i \in \{k ..< m\}. \ ?Mk \ i \ ?Gsk \ i)$

(is $\ = \text{vec } m \ ?Gsk$)

unfolding $Gsk$-def using $ij$ i len-vs by (auto simp: $\text{mat-of-rows-def}$)

also have $\text{vec } m \ ?Mk \ \cdot \ \text{vec } m \ ?Gsk = (\sum (i \in \{0 ..< k\}. \ ?Mk \ i \ ?Gsk \ i))$

unfolding scalar-prod-def by auto

also have $\ldots = (\sum (i \in \{k ..< m\}. \ ?Mk \ i \ ?Gsk \ i) + (\sum (i \in \{k ..< m\}. \ ?Mk \ i \ ?Gsk \ i)$

by (rule $\text{sum.union-disjoint}$, auto)

also have $\sum (i \in \{k ..< m\}. \ ?Mk \ i \ ?Gsk \ i) = 0$

by (rule $\text{sum.neutral}$, auto)

also have $\sum (i \in \{0 ..< k\}. \ ?Mk \ i \ ?Gsk \ i) = (\sum (i' \in \{0 ..< k\}. \ ?Mk \ i \ ?Gsk \ i) = (\sum (i' \in \{0 ..< k\}. \ ?Mk \ i \ ?Gsk \ i))$
by (rule sum.cong, auto)
also have ... = row Mk i \cdot col Gsk j unfolding scalar-prod-def using ij
by (auto simp: Gsk-def Mk-def)
also have ... = (Mk * Gsk) \$\$ (i, j) using ij Mk Gsk by simp
finally show Gk \$\$ (i, j) = (Mk * Gsk) \$\$ (i, j) by simp
qed
have cong: \(\bigwedge a b c d. a = b \implies c = d \implies a * c = b * d\) by auto
have Gramian-determinant fs k = det (Gk * Gk^T)
  unfolding Gramian-determinant-def Gramian-matrix-def Let-def
by (rule arg-cong[of \$\$ det\$\$, rule cong, insert k, auto simp: Gk-def assms(3))
also have Gk^T = Gsk^T * Mk^T (is \$\$ = ?TGsk * ?TMk) unfolding id
by (rule transpose-mult[OF Mk Gsk])
also have Gk = Mk * Gsk by fact
also have ... * (?TGsk * ?TMk) = Mk * (Gsk * (?TGsk * ?TMk))
  by (rule assoc-mult-mat[OF Mk Gsk, of - k], insert Gsk Mk, auto)
also have det ... = det Mk * det (Gsk * (?TGsk * ?TMk))
  by (rule det-mult[OF Mk], insert Gsk Mk, auto)
also have ... = det (Gsk * (?TGsk * ?TMk)) using detMk by simp
also have Gsk * (?TGsk * ?TMk) = (Gsk * ?TGsk) * ?TMk
  by (rule assoc-mult-mat[symmetric, OF Gsk], insert Gsk Mk, auto)
also have det ... = det (Gsk * ?TGsk) * det ?TMk
  by (rule det-mult, insert Gsk Mk, auto)
also have ... = det (Gsk * ?TGsk) using detMk det-transpose[OF Mk] by simp
also have Gsk * ?TGsk = mat k k (λ (i,j). if i = j then sq-norm (vs ! j) else 0) (is \$\$ = ?M)
proof (rule eq-multI)
  show dim-row (Gsk * ?TGsk) = dim-row ?M unfolding Gsk-def by auto
  show dim-col (Gsk * ?TGsk) = dim-col ?M unfolding Gsk-def by auto
fix i j
assume i < dim-row ?M j < dim-col ?M
hence ij: i < k j < k and ijn: i < m j < m using k by auto
{
  fix i
  assume i < k
  hence i < m using k by auto
  hence Gs: vs ! i ∈ ?Rn using len-vs vs(1) by auto
  have row Gsk i = vs ! i unfolding row-def Gsk-def
    by (rule eq-vecI, insert Gs (i < k), auto)
} note row = this
have (Gsk * ?TGsk) \$\$ (i,j) = row Gsk i \cdot row Gsk j using ij Gsk by auto
also have ... = vs ! i \cdot c vs ! j using row ij by simp
also have ... = (if i = j then sq-norm (vs ! j) else 0)
proof (cases i = j)
  assume i = j
  thus ?thesis by (simp add: sq-norm-vec-as-cscalar-prod)
next
  assume i ≠ j
  have corthogonal vs by (rule gram-schmidt)
  from (i ≠ j) corthogonalD[OF this, unfolded len-vs, OF ijn]
show \(?thesis\) by auto

qed

also have \(\ldots = ?M \$(i,j)\) using \(ij\) by simp

finally show \((Gsk \ast ?TGsk) \$(i,j) = ?M \$(i,j)\).

qed

also have \(\det ?M = \prod\text{-}\text{list} (\text{diag}\text{-}\text{mat} ?M)\)
  by (rule \text{det-upper-triangular}, auto)

also have \(\text{diag}\text{-}\text{mat} ?M = \text{map} (\lambda j. \text{sq-norm} (\text{vs} ! j)) [0 ..< k]\) unfolding \text{diag}\text{-}\text{mat-def} by auto

also have \(\ldots > 0\)
  by (rule \text{prod-pos}, intro \text{ballI} \text{sq-norm-pos}, insert \(k\), auto)

finally show \(0 < \text{Gramian-determinant fs k}\) by auto

qed simp

lemma \(\text{prod}\text{-}\text{list-le-mono}\): fixes \(us :: \{\text{linordered-nonzero-semiring, ordered-ring}\}\)

list
  assumes \(\text{length } us = \text{length } vs\)
  and \(\bigwedge i. i < \text{length } vs \implies 0 \leq us ! i \wedge us ! i \leq vs ! i\)
  shows \(0 \leq \text{prod}\text{-}\text{list } us \wedge \text{prod}\text{-}\text{list } us \leq \text{prod}\text{-}\text{list } vs\)
  using \text{assms}

proof (induction \(us vs\) rule: \text{list-induct2})

  case (Cons \(u us v vs\))

  have \(0 \leq \text{prod}\text{-}\text{list } us \wedge \text{prod}\text{-}\text{list } us \leq \text{prod}\text{-}\text{list } vs\)
    by (rule \text{Cons.IH}, \text{insert} \text{Cons.prems[of Suc i for i]}, auto)

  moreover have \(0 \leq u \wedge u \leq v\) using \text{Cons.prems[of 0]} by auto

  ultimately show \(?\text{case}\) by (auto intro: \text{mult-mono})

qed simp

lemma \(\text{lattice}\text{-}\text{of}\text{-}\text{of}\text{-}\text{int}\): assumes \(G: \text{set } F \subseteq \text{carrier}\text{-}\text{vec } n\)
  and \(f \in \text{vec}\text{-}\text{module.\text{lattice}\text{-}\text{of}\ n } F\)
  shows \(\text{map}\text{-}\text{vec rat}\text{-}\text{of}\text{-}\text{int } f \in \text{vec}\text{-}\text{module.\text{lattice}\text{-}\text{of} } n\) (\text{map} (\text{map}\text{-}\text{vec of-int} ) F)\)
  (\(is = ?f\) \in \text{vec}\text{-}\text{module.\text{lattice}\text{-}\text{of}\ n} - ?F\)

proof –

  let \(?sl = \text{abelian-monoid.\text{sumlist}} (\text{module}\text{-}\text{vec \text{TYPE}'a::semiring-1} ) n\)
  note \(d = \text{vec}\text{-}\text{module.\text{lattice}\text{-}\text{of-def}}\)

  note \(d = \text{vec}\text{-}\text{module.\text{lattice}\text{-}\text{of-def}}\)

  note \(\text{sumlist}\text{-}\text{vec-index} = \text{vec}\text{-}\text{module.\text{sumlist}\text{-}\text{vec-index}}\)

  from \(G\) have \(Gi: \bigwedge i. i < \text{length } F \implies F ! i \in \text{carrier}\text{-}\text{vec } n\) by auto

  from \(Gi\) have \(Gid: \bigwedge i. i < \text{length } F \implies \text{dim}\text{-}\text{vec } (F ! i) = n\) by auto

  from \text{assms}(2)[\text{unfolded } d]\]

  obtain \(c\) where

    \(ffe: f = ?sl (\text{map } (\lambda i. \text{of-int } (c i) \cdot_{F} ! i) [0..<\text{length } F])\)
    (\(is = ?g\)) by auto
have \( \mathcal{F} = \text{sl} \left( \text{map} \left( \lambda i. \text{of-int} \ (\text{c i}) \cdot \mathcal{F} \right) \left[ 0..<\text{length} \ ?\mathcal{F} \right] \right) \ (\text{is} = \ ?\text{gg}) \)
proof
  have \( d1\left[\text{simp}\right] \): \( \text{dim-vec} \ ?g = n \) by (subst \( \text{dim} \), auto simp: \( \text{Gi} \))
  have \( d2\left[\text{simp}\right] \): \( \text{dim-vec} \ ?gg = n \) unfolding \( \text{length-map} \) by (subst \( \text{vec-module} \cdot \text{sumlist-dim} \), auto simp: \( \text{Gi} \ G \))
  show \( \text{?thesis} \) unfolding \( \text{ffc} \ \text{length-map} \) apply (rule eq-vecI)
  apply (insert \( d1 \ d2 \), auto)
  apply (subst \( 1 \ 2 \ \text{sumlist-vec-index} \), auto simp: \( \text{Gi} \ G \))
  apply (unfold of-int-hom \( \cdot \text{hom-sum-list} \))
  apply (intro arg-cong)
  by (auto simp: \( \text{G} \ \text{Gi} \), (subst \( \text{index-smult-vec} \), simp add: \( \text{Gid} \)) +,
      subst \( \text{index-map-vec} \), auto simp: \( \text{Gid} \))
qed
thus \( \mathcal{F} \in \text{vec-module} \cdot \text{lattice-of} \ ?n \ ?\mathcal{F} \) unfolding \( \text{d} \) by auto
qed

lemma Hadamard’s inequality:
fixes \( A :: \text{real mat} \)
assumes \( A : A \in \text{carrier-mat} \ n \ n \)
shows \( \text{abs} \ (\text{det} \ A) \le \sqrt{\text{prod-list} \ (\text{map} \ \text{sq-norm} \ (\text{rows} \ A))} \)
proof
  interpret \( \text{gso} : \text{gram-schmidt} \ n \ \text{TYPE}(\text{real}) \).
  let \( ?us = \text{map} \ (\text{row} \ A) \left[ 0..<n \right] \)
  have \( \text{len} : \text{length} \ ?us = n \) by simp
  have \( \text{us} : \text{set} \ ?us \subseteq \text{carrier-vec} \ n \) using \( A \) by auto
  obtain \( \text{vs} \) where \( \text{main} : \text{snd} \left( \text{gso} . \text{main} ?us \right) = \text{vs} \) by force
  show \( \text{?thesis} \) proof
    (cases \( \text{carrier-vec} \ n \subseteq \text{gso} . \text{span} \ (\text{set} \ ?us) \))
    case True
    with \( \text{us} \) \( \text{len} \) have \( \text{basis} : \text{gso} . \text{basis-list} \ ?us \) unfolding \( \text{gso} . \text{basis-list-def} \) by auto
    note \( \text{conn} = \text{gso} . \text{basis-list-imp-lin-indpt-list}[\text{OF} \ \text{basis}] \) \( \text{len} \) \( \text{main} \)
    note \( \text{gram} = \text{gso} . \text{gram-schmidt}[\text{OF} \ \text{conn}] \)
    note \( \text{main} = \text{gso} . \text{main-connect}[\text{OF} \ \text{conn}] \)
    from \( \text{main} \) have \( \text{len-vs} : \text{length} \ \text{vs} = n \) by simp
    have \( \text{last} : 0 \le \text{prod-list} \ (\text{map} \ \text{sq-norm} \ \text{vs}) \ \wedge \text{prod-list} \ (\text{map} \ \text{sq-norm} \ \text{vs}) \le \text{prod-list} \ (\text{map} \ \text{sq-norm} \ ?us) \)
      proof (rule \( \text{prod-list-le-mono} \), force simp: \( \text{main}(2) \), unfold \( \text{length-map} \) \( \text{length-upt} \))
        fix \( i \)
        assume \( i < n - 0 \)
        hence \( i : i < n \) by simp
        have \( \text{vsi} : \text{map} \ \text{sq-norm} \ \text{vs} \ ! \ i = \text{sq-norm} \ (\text{vs} ! i) \) using \( \text{main}(2) \) \( i \) by simp
        have \( \text{usi} : \text{map} \ \text{sq-norm} \ ?us \ ! \ i = \text{sq-norm} \ (\text{row} \ A \ i) \) using \( i \) by simp
        have \( \text{zero} : 0 \le \text{sq-norm} \ (\text{vs} ! i) \) by auto
        have \( \text{le} : \text{sq-norm} \ (\text{vs} ! i) \le \text{sq-norm} \ (\text{row} \ A \ i) \) using \( \text{gso} . \text{sq-norm-gso-le-f}[\text{OF} \ \text{main} \ i] \)
      qed
  qed
proof
  interpret \( \text{gso} : \text{gram-schmidt} \ n \ \text{TYPE}(\text{real}) \).
  let \( ?us = \text{map} \ (\text{row} \ A) \left[ 0..<n \right] \)
  have \( \text{len} : \text{length} \ ?us = n \) by simp
  have \( \text{us} : \text{set} \ ?us \subseteq \text{carrier-vec} \ n \) using \( A \) by auto
  obtain \( \text{vs} \) where \( \text{main} : \text{snd} \left( \text{gso} . \text{main} ?us \right) = \text{vs} \) by force
  show \( \text{?thesis} \) proof
    (cases \( \text{carrier-vec} \ n \subseteq \text{gso} . \text{span} \ (\text{set} \ ?us) \))
    case True
    with \( \text{us} \) \( \text{len} \) have \( \text{basis} : \text{gso} . \text{basis-list} \ ?us \) unfolding \( \text{gso} . \text{basis-list-def} \) by auto
    note \( \text{conn} = \text{gso} . \text{basis-list-imp-lin-indpt-list}[\text{OF} \ \text{basis}] \) \( \text{len} \) \( \text{main} \)
    note \( \text{gram} = \text{gso} . \text{gram-schmidt}[\text{OF} \ \text{conn}] \)
    note \( \text{main} = \text{gso} . \text{main-connect}[\text{OF} \ \text{conn}] \)
    from \( \text{main} \) have \( \text{len-vs} : \text{length} \ \text{vs} = n \) by simp
    have \( \text{last} : 0 \le \text{prod-list} \ (\text{map} \ \text{sq-norm} \ \text{vs}) \ \wedge \text{prod-list} \ (\text{map} \ \text{sq-norm} \ \text{vs}) \le \text{prod-list} \ (\text{map} \ \text{sq-norm} \ ?us) \)
      proof (rule \( \text{prod-list-le-mono} \), force simp: \( \text{main}(2) \), unfold \( \text{length-map} \) \( \text{length-upt} \))
        fix \( i \)
        assume \( i < n - 0 \)
        hence \( i : i < n \) by simp
        have \( \text{vsi} : \text{map} \ \text{sq-norm} \ \text{vs} \ ! \ i = \text{sq-norm} \ (\text{vs} ! i) \) using \( \text{main}(2) \) \( i \) by simp
        have \( \text{usi} : \text{map} \ \text{sq-norm} \ ?us \ ! \ i = \text{sq-norm} \ (\text{row} \ A \ i) \) using \( i \) by simp
        have \( \text{zero} : 0 \le \text{sq-norm} \ (\text{vs} ! i) \) by auto
        have \( \text{le} : \text{sq-norm} \ (\text{vs} ! i) \le \text{sq-norm} \ (\text{row} \ A \ i) \) using \( \text{gso} . \text{sq-norm-gso-le-f}[\text{OF} \ \text{main} \ i] \)
      qed
  qed
qed
conn i]
   unfolding main\(^{(2)}\) using \(i\) by simp
   show \(\theta \leq\) map sq-norm vs \(!\) \(i\wedge\) map sq-norm vs \(!\) \(i\leq\) map sq-norm ?us \(!\)
   unfolding vsi usi using zero le by auto
   qed

have Fs: gso.Fs ?us \(\in\) carrier-mat n n by auto
have A-Fs: \(A = gso.Fs ?us\)
   by (rule eq-matI, subst gso.FF-index[of conn], insert A, auto)

   hence \(\text{abs (det } A) = \text{abs (det (gso.Fs ?us))}\)
   by simp

also have \(\ldots = \text{abs (sqrt (det (gso.Fs ?us) \ast det (gso.Fs ?us))})\)
also have \(\text{det (gso.Fs ?us) \ast det (gso.Fs ?us)} = \text{det (gso.Fs ?us) \ast det (gso.Fs ?us)}\)
   unfolding det-transpose[of Fs ..]
also have \(\ldots = \text{det (gso.Fs ?us} \ast (gso.Fs ?us)^T\)
   by (subt det-mult[of Fs], insert Fs, auto)
also have \(\ldots = \text{gso.Gramian-determinant} ?us n\)
   by (rule arg-cong[of - - det], rule arg-cong2[of - - - - op *], insert A, auto)
also have \(\ldots = \prod\{\|vs ! j\|^2\} \quad \text{unfolding gso.Gramian-determinant[of Fs conn le-refl]}\)
   by (rule prod.cong, auto)
also have \(\ldots = \text{prod-list} (map (\lambda i. \text{sq-norm} \(\{vs ! i\}) [0 ..< n])\)
   by (subt prod.distinct-set-conv-list, auto)
also have \(\text{map (\lambda i. \text{sq-norm} \(\{vs ! i\}) [0 ..< n]\) = map sq-norm vs}\)
   using len-vs by (intro nth-equalityI, auto)
also have \(\text{abs (sqrt (prod-list \ldots ))} \leq \text{sqrt (prod-list (map sq-norm ?us))}\)
   using last by simp
also have \(\?us = \text{rows } A\)
   unfolding rows-def using A by simp
finally show \(\?\text{thesis} .\)

next
   case False
   from mat-of-rows-rows[unfolded rows-def,of A] A gram-schmidt.non-span-det-zero[of Fs conn le-refl]

   have zero: det A = 0 by auto
   have ge: prod-list (map sq-norm (rows A)) \(\geq\) 0
      by (rule prod-list-nonneg, auto simp: sq-norm-vec-ge-0)
   show \(\?\text{thesis}\)
   unfolding zero using ge by simp
   qed

qed

definition gram-schmidt-wit = gram-schmidt.main
lemmas gram-schmidt-wit = gram-schmidt.weakly-reduced-imp-short-vector[folded gram-schmidt-wit-def]
declare gram-schmidt.adjuster-wit.simps[code]
declare gram-schmidt.sub2-wit.simps[code]
declare gram-schmidt.main-def[code]
definition gram-schmidt-int :: nat ⇒ int vec list ⇒ rat list list × rat vec list
where
gram-schmidt-int n us = gram-schmidt-wit n (map (map-vec of-int) us)

lemma snd-gram-schmidt-int : snd (gram-schmidt-int n us) = gram-schmidt n (map (map-vec of-int) us)
  unfolding gram-schmidt-int-def gram-schmidt-wit-def gram-schmidt.gso-connect by metis

fun adjuster-triv :: nat ⇒ 'a :: trivial-conjugatable-ordered-field vec ⇒ ('a vec × 'a) list ⇒ 'a vec
where adjuster-triv n w [] = 0v n
  | adjuster-triv n w ((u,nu)#us) = -(w · u)/ nu · v u + adjuster-triv n w us

fun gram-schmidt-sub-triv
where gram-schmidt-sub-triv n us [] = us
  | gram-schmidt-sub-triv n us (w # ws) = (let u = adjuster-triv n w us + w in gram-schmidt-sub-triv n ((u, sq-norm u) # us) ws)

definition gram-schmidt-triv :: nat ⇒ 'a :: trivial-conjugatable-ordered-field vec list ⇒ ('a vec × 'a) list
where gram-schmidt-triv n ws = rev (gram-schmidt-sub-triv n [] ws)

lemma adjuster-triv: adjuster-triv n w (map (λ x. (x,sq-norm x)) us) = adjuster n w us
  by (induct us, auto simp: sq-norm-vec-as-cscalar-prod)

lemma gram-schmidt-sub-triv: gram-schmidt-sub-triv n ((map (λ x. (x,sq-norm x)) us)) ws =
  map (λ x. (x, sq-norm x)) (gram-schmidt-sub n us ws)
  by (rule sym, induct ws arbitrary: us, auto simp: adjuster-triv o-def Let-def)

lemma gram-schmidt-triv[simp]: gram-schmidt-triv n ws = map (λ x. (x,sq-norm x)) (gram-schmidt n ws)
  unfolding gram-schmidt-def gram-schmidt-triv-def rev-map[symmetric]
  by (auto simp: gram-schmidt-sub-triv[symmetric])

definition gram-schmidt-int :: nat ⇒ int vec list ⇒ rat list list × rat vec list
where
gram-schmidt-int n us = gram-schmidt-wit n (map (map-vec of-int) us)

lemma snd-gram-schmidt-int : snd (gram-schmidt-int n us) = gram-schmidt n (map (map-vec of-int) us)
  unfolding gram-schmidt-int-def gram-schmidt-wit-def gram-schmidt.gso-connect by metis

fun adjuster-triv :: nat ⇒ 'a :: trivial-conjugatable-ordered-field vec ⇒ ('a vec × 'a) list ⇒ 'a vec
where adjuster-triv n w [] = 0v n
  | adjuster-triv n w ((u,nu)#us) = -(w · u)/ nu · v u + adjuster-triv n w us

fun gram-schmidt-sub-triv
where gram-schmidt-sub-triv n us [] = us
  | gram-schmidt-sub-triv n us (w # ws) = (let u = adjuster-triv n w us + w in gram-schmidt-sub-triv n ((u, sq-norm u) # us) ws)

definition gram-schmidt-triv :: nat ⇒ 'a :: trivial-conjugatable-ordered-field vec list ⇒ ('a vec × 'a) list
where gram-schmidt-triv n ws = rev (gram-schmidt-sub-triv n [] ws)

lemma adjuster-triv: adjuster-triv n w (map (λ x. (x,sq-norm x)) us) = adjuster n w us
  by (induct us, auto simp: sq-norm-vec-as-cscalar-prod)

lemma gram-schmidt-sub-triv: gram-schmidt-sub-triv n ((map (λ x. (x,sq-norm x)) us)) ws =
  map (λ x. (x, sq-norm x)) (gram-schmidt-sub n us ws)
  by (rule sym, induct ws arbitrary: us, auto simp: adjuster-triv o-def Let-def)

lemma gram-schmidt-triv[simp]: gram-schmidt-triv n ws = map (λ x. (x,sq-norm x)) (gram-schmidt n ws)
  unfolding gram-schmidt-def gram-schmidt-triv-def rev-map[symmetric]
  by (auto simp: gram-schmidt-sub-triv[symmetric])

end

7 The LLL algorithm

This theory provides an implementation and a soundness proof of the LLL algorithm to compute a ”short” vector in a lattice.

theory LLL
  imports
    Gram-Schmidt-2
7.1 Implementation of the LLL algorithm

definition floor-ceil where floor-ceil \( x = \text{floor}\ (x + 1/2) \)

definition scalar-prod-int-rat :: int vec \Rightarrow rat vec \Rightarrow rat (\text{infix} \cdot i 70) where
\( x \cdot i y = (y \cdot \text{map-vec\ rat-of-int\ } x) \)

type-synonym f-repr = int vec list-repr

type-synonym g-repr = (rat vec \times rat) list-repr

definition g-i :: g-repr \Rightarrow rat vec where g-i Gr = (fst (get-nth-i Gr))
definition sqnorm-g-i :: g-repr \Rightarrow rat where sqnorm-g-i Gr = (snd (get-nth-i Gr))
definition g-im1 :: g-repr \Rightarrow rat vec where g-im1 Gr = (fst (get-nth-im1 Gr))
definition sqnorm-g-im1 :: g-repr \Rightarrow rat where sqnorm-g-im1 Gr = (snd (get-nth-im1 Gr))
definition \( \mu \)-i-im1 :: f-repr \Rightarrow g-repr \Rightarrow rat where
\( \mu \)-i-im1 Fr Gr = (get-nth-i Fr \cdot i g-im1 Gr) / sqnorm-g-im1 Gr

definition \( \mu \)-ij :: int vec \Rightarrow rat vec \times rat \Rightarrow rat where
\( \mu \)-ij fi gj-norm = (case gj-norm of (gj, norm-gj) \Rightarrow (fi \cdot i gj) / norm-gj)

type-synonym state = nat \times f-repr \times g-repr

fun basis-reduction-add-row-main :: state \Rightarrow int vec \Rightarrow rat \Rightarrow state \times int where
basis-reduction-add-row-main (i,F,G) fj mu = (let
\( c = \text{floor-ceil\ } mu \)
in if \( c = 0 \) then
\((i,F,G), c)\)
else
let
\( \bar{f}_i = \text{get-nth-i } F - (c \cdot i fj)\);
\( F' = \text{update-i F } \bar{f}_i \)
in \((i,F',G), c)\)

definition basis-reduction-add-row-i-im1 :: state \Rightarrow state \times rat where
basis-reduction-add-row-i-im1 state = (case state of (\cdot,F,G) \Rightarrow let
  \mu = \mu-i-im1 F G;
  f_j = get-nth-im1 F
  in map-prod id (\lambda c. \mu = rat-of-int c) (basis-reduction-add-row-main state f_j mu))

definition increase-i :: state \Rightarrow state where
increase-i state = (case state of (i, F, G) \Rightarrow (Suc i, inc-i F, inc-i G))

fun basis-reduction-swap :: state \Rightarrow rat \Rightarrow state where
basis-reduction-swap (i,F,G) mu = (let
  g_i = g-i G;
  g-im1 = g-im1 G;
  f_i = get-nth-i F;
  f-im1 = get-nth-im1 F;
  new-g-im1 = g_i + \mu \cdot g-im1;
  norm-g-im1 = sq-norm new-g-im1;
  new-gi = g-im1 - (f-im1 \cdot new-g-im1 / norm-g-im1) \cdot new-g-im1;
  norm-gi = sq-norm new-gi;
  G' = dec-i (update-im1 (update-i G (new-gi,norm-gi)) (new-g-im1,norm-g-im1));
  F' = dec-i (update-im1 (update-i F f-im1) f_i)
in (i - 1, F', G'))

definition basis-reduction-step :: rat \Rightarrow state \Rightarrow state where
basis-reduction-step \alpha state = (if fst state = 0 then increase-i state
  else case basis-reduction-add-row-i-im1 state of
  (state', \mu) \Rightarrow
    case state' of (i, F, G) \Rightarrow
    if sq-norm-g-im1 G > \alpha \cdot sq-norm-g-i G
    then basis-reduction-swap state' mu
    else increase-i state'
  )

partial-function (tailrec) basis-reduction-main :: rat \Rightarrow nat \Rightarrow state \Rightarrow state where
  [code]: basis-reduction-main \alpha m state = (case state of (i,F,G) \Rightarrow
  if i < m
  then basis-reduction-main \alpha m (basis-reduction-step \alpha state)
  else state)

definition basis-reduction-part-1 :: nat \Rightarrow rat \Rightarrow int vec list \Rightarrow state where
basis-reduction-part-1 n \alpha F = (let m = length F;
  G = gram-schmidt-triv n (map (map-vec of-int) F);
  Fr = ([], F);
  Gr = ([], G)
in basis-reduction-main \alpha m (0, Fr, Gr))

definition weakly-reduce-basis :: nat \Rightarrow rat \Rightarrow int vec list \Rightarrow int vec list \times rat vec list where
weakly-reduce-basis \( n \alpha F = (\lambda state. ((\text{of-list-repr o fst o snd}) state)) \)
\((\text{basis-reduction-part-1} n \alpha F)\)

definition short-vector :: \( rat \Rightarrow int vec list \Rightarrow int vec \)
where
\(\text{short-vector} \alpha F = (hd o fst) (\text{weakly-reduce-basis} (\text{dim-vec} (hd F)) \alpha F)\)

fun basis-reduction-add-row-i-all-main :: \( state \Rightarrow int vec list \Rightarrow (rat vec \times rat) \)
\(\Rightarrow state \)
where
\(\text{basis-reduction-add-row-i-all-main} state (\text{Cons fj fjs}) (\text{Cons gj gjs}) = (\text{case state of} (i,F,G) \Rightarrow \]
\(\text{let} fi = \text{get-nth-i} F \in \]
\(\text{basis-reduction-add-row-i-all-main} (\text{fst} (\text{basis-reduction-add-row-main} state fj \(\mu-ij fi gj\))) fjs gjs\)
\(| \text{basis-reduction-add-row-i-all-main} state - - = state\)

definition basis-reduction-add-row-i-all :: \( state \Rightarrow state \)
where
\(\text{basis-reduction-add-row-i-all} state = (\text{case state of} (i,F,G) \Rightarrow \]
\(\text{let} fjs = \text{fst} F;
\text{gjs} = \text{fst} G\)
\(in\ \text{basis-reduction-add-row-i-all-main} state fjs gjs\)

fun basis-reduction-part-2-main :: \( state \Rightarrow state \)
where
\(\text{basis-reduction-part-2-main} (i,F,G) = (if i = 0 \text{ then} (i,F,G) \text{ else} \]
\(\text{case} \text{basis-reduction-add-row-i-all} (i - 1, dec-i F, dec-i G) \text{ of} (-, F', G') \Rightarrow \text{basis-reduction-part-2-main} (i - 1, F', G')\)

definition basis-reduction-part-2 :: \( nat \Rightarrow rat \Rightarrow int vec list \Rightarrow state \)
where
\(\text{basis-reduction-part-2} n \alpha F = \text{basis-reduction-part-2-main} \)
\((\text{basis-reduction-part-1} n \alpha F)\)

definition strictly-reduce-basis :: \( nat \Rightarrow rat \Rightarrow int vec list \Rightarrow int vec list \times rat vec list \)
where
\(\text{strictly-reduce-basis} n \alpha F = (\lambda state. ((\text{of-list-repr o fst o snd}) state)) \)
\((\text{basis-reduction-part-2} n \alpha F)\)

7.2 LLL algorithm is sound

lemma floor-ceil: \(|x - \text{rat-of-int} (\text{floor-ceil} \ x)| \leq \text{inverse} 2\)
unfolding floor-ceil-def by (metis (no-types, hide-lams) abs-divide abs-neg-one round-def
div-by-1 div-minus-right inverse-eq-divide minus-diff-eq of-int-round-ubs-le)

lemma \(\mu-i-im1\)-code[code-unfold]:
\(\mu-i-im1 F G = \mu-ij (get-nth-i F) (get-nth-im1 G)\)
unfolding \(\mu-i-im1\)-def g-im1-def sqnorm-g-im1-def \mu-ij-def
by (auto split: prod.splits)
lemma scalar-prod-int-rat[simp]: dim-vec x = dim-vec y \implies x \cdot y = map-vec of-int x \cdot y

unfolding scalar-prod-int-rat-def by (intro comm-scalar-prod[of - dim-vec x], auto intro: carrier-vecI)

definition int-times-rat :: int \Rightarrow rat \Rightarrow rat
where int-times-rat i x = of-int i \ast x

lemma scalar-prod-int-rat-code[code]: v \cdot i w = (\sum i = 0..<dim-vec v. int-times-rat (v \$ i) (w \$ i))

unfolding scalar-prod-int-rat-def Let-def scalar-prod-def int-times-rat-def by (rule sum.cong, auto)

lemma int-times-rat-code[code abstract]: quotient-of (int-times-rat i x) = (case quotient-of x of (n, d) \Rightarrow Rat.normalize (i \ast n, d))

unfolding int-times-rat-def rat-times-code by auto

locale LLL =
fixes n :: nat and m :: nat
begin

sublocale vec-module TYPE(int) n.

sublocale gs: gram-schmidt n TYPE(rat).

abbreviation RAT where RAT = map (map-vec rat-of-int)
abbreviation SRAT where SRAT xs = set (RAT xs)
abbreviation Rn where Rn = carrier-vec n :: rat vec set

definition GSO where GSO F = gs.gso (RAT F)

definition g-repr :: nat \Rightarrow g-repr \Rightarrow int vec list \Rightarrow bool
where g-repr i G F = (i \leq m \land list-repr i G (map (\lambda x. (x, sq-norm x)) (map (GSO F) [0..<m])))

abbreviation (input) f-repr = list-repr

lemma g-i-GSO: g-repr i G F \Rightarrow i < m \Rightarrow g-i G = GSO F i

unfolding g-repr-def g-i-def by (cases F, auto simp: get-nth-i)

lemma sqnorm-g-i-GSO: g-repr i G F \Rightarrow i < m \Rightarrow sqnorm-g-i G = sq-norm (GSO F i)

unfolding g-repr-def sqnorm-g-i-def by (cases G, auto simp: get-nth-i)

lemma g-im1-GSO: g-repr i G F \Rightarrow i \neq 0 \Rightarrow g-im1 G = GSO F (i - 1)

unfolding g-repr-def g-im1-def by (cases G, cases i, auto simp: get-nth-im1)

lemma sqnorm-g-im1-GSO: g-repr i G F \Rightarrow i \neq 0 \Rightarrow sqnorm-g-im1 G =
lemma inc-i-gso: assumes i < m g-repr i G F
shows g-repr (Suc i) (inc-i G) F
  using assms unfolding g-repr-def by (auto simp: inc-i)

lemma μ-i-im1: assumes gr: f-repr i Fr F and gso: g-repr i G F
  and n:m = length F and i:i ≠ 0 i < m
  and dim: F ! i ∈ carrier-vec n gs.gso (RAT F) (i - 1) ∈ carrier-vec n
shows μ-i-im1 Fr G = gs.μ (RAT F) i (i - 1)
  unfolding g-repr-def μ-i-im1-def gs.μ.simps
  get-nth-i[OF gr,unfolded length-map,OF i(2)[unfolded n]]
  of-list-repr[OF gr] g-im1-GSO[OF gso i(1)]
  sqnorm-g-im1-GSO[OF gso i(1)] GSO-def
  using i n dim by auto

context fixes L :: int vec set and α :: rat
begin

definition LLL-invariant :: state ⇒ int vec list ⇒ rat vec list ⇒ bool where
  LLL-invariant state F G = (case state of (i,Fr,Gr) ⇒
    snd (gram-schmidt-int n F) = G ∧
    gs.lin-indpt-list (RAT F) ∧
    lattice-of F = L ∧
    gs.weakly-reduced α i G ∧
    i ≤ m ∧
    length F = m ∧
    f-repr i Fr F ∧
    g-repr i Gr F
  )

lemma LLL-invD: assumes LLL-invariant (i,Fr,Gr) F G
  shows F = of-list-repr Fr
    snd (gram-schmidt-int n F) = G
    set F ⊆ carrier-vec n
    length F = m
    lattice-of F = L
    gs.weakly-reduced α i G
    i ≤ m
    f-repr i Fr F
    g-repr i Gr F
    gs.lin-indpt-list (RAT F)
  using assms gs.lin-indpt-list-def of-list-repr[of i Fr F] unfolding LLL-invariant-def
  split by auto

lemma LLL-invI: assumes
  f-repr i Fr F
  g-repr i Gr F
\[ \text{snd (gram-schmidt-int \( n \) \( F \))} = G \]

\[ \text{lattice-of} \ F = L \]

\[ \text{gs.weakly-reduced} \ \alpha \ \iota \ G \]

\[ i \leq m \]

\[ \text{length} \ F = m \]

\[ \text{gs.lin-indpt-list} \ (\text{RAT} \ F) \]

\[ \text{shows} \ \text{LLL-invariant} \ (i, Fr, Gr) \ F \ G \]

\[ \text{unfolding} \ \text{LLL-invariant-def} \ \text{Let-def} \ \text{split using} \ \text{assms of-list-repr[OF assms(1)]} \]

\[ \text{by auto} \]

\[ \text{lemma} \ \text{gram-schmidt-int-connect:} \ \text{fixes} \ F :: \int \text{vec list} \]

\[ \text{assumes} \ \text{gs.lin-indpt-list} \ (\text{RAT} \ F) \ \text{snd (gram-schmidt-int \( n \) \( F \))} = G \ \text{length} \ F = m \]

\[ \text{shows} \ G = \text{map} \ (\text{gs.gso} \ (\text{RAT} \ F)) \ [0..<m] \]

\[ \text{proof} - \]

\[ \text{from} \ \text{assms} \ \text{have} \ \text{gsw:} \ \text{snd (gram-schmidt-wit \( n \) (RAT F))} = G \]

\[ \text{by (auto simp: gram-schmidt-int-def)} \]

\[ \text{from gram-schmidt.main-connect[OF assms(1)] - gsw[unfolded gram-schmidt-wit-def], of m]} \ \text{assms(3)} \]

\[ \text{show} \ G = \text{map} \ (\text{gs.gso} \ (\text{RAT} \ F)) \ [0..<m] \ \text{by auto} \]

\[ \text{qed} \]

\[ \text{lemma} \ \text{LLL-connect:} \ \text{fixes} \ F :: \int \text{vec list} \]

\[ \text{assumes} \ \text{inv:} \ \text{LLL-invariant} \ (i, Fr, Gr) \ F \ G \]

\[ \text{shows} \ G = \text{map} \ (\text{gs.gso} \ (\text{RAT} \ F)) \ [0..<m] \]

\[ \text{using} \ \text{gram-schmidt-int-connect[of F G] LLL-invD[OF inv]} \ \text{by auto} \]

\[ \text{lemma} \ \text{gs-gs-identical:} \ \text{assumes} \ \land \ i. \ i \leq x \ \Longrightarrow \ f1 ! i = f2 ! i \]

\[ \text{shows} \ \text{gs.gso f1 x = gs.gso f2 x} \]

\[ \text{using} \ \text{assms} \]

\[ \text{proof} (\text{induct x rule: nat-less-induct[rule-format]}) \]

\[ \text{case (1 \( x \))} \]

\[ \text{hence} \ \text{fg:} \text{op + (f1 ! x) = op + (f2 ! x)} \ \text{by auto} \]

\[ \text{show ?case} \]

\[ \text{apply(subst (1 2) gs.gso.simps) unfolding gs.\( \mu \)..simps} \]

\[ \text{apply(rule cong[OF fg cong[OF refl[of gs.sumlist]]])} \]

\[ \text{using 1 by auto} \]

\[ \text{qed} \]

\[ \text{lemma} \ \text{gs-\( \mu \)-identical:} \ \text{assumes} \ \land \ k. \ j < i \ \Longrightarrow \ k \leq j \ \Longrightarrow \ f1 ! k = f2 ! k \]

\[ \text{and} \ j < i \ \Longrightarrow \ f1 ! i = f2 ! i \]

\[ \text{shows} \ \text{gs.\( \mu \) f1 i j = gs.\( \mu \) f2 i j} \]

\[ \text{proof} - \]

\[ \text{from gs-gs-identical[of j f1 f2] assms have:} \ j < i \ \Longrightarrow \ \text{gs.gso f1 j = gs.gso f2 j} \]

\[ \text{by auto} \]

\[ \text{show ?thesis unfolding gs.\( \mu \)..simps using assms id by auto} \]

\[ \text{qed} \]

123
lemma \( g\text{-}i \): assumes \( \text{inv} \): LLL-invariant \((i, Fr, Gr)\) \( F G \) and \( i < m \)
shows \( g\text{-}i Gr = G ! i \)
\( \text{sqnorm}\text{-}g\text{-}i Gr = \text{sq-norm} (G ! i) \)

proof –

\( \text{note conn} = \text{LLL-connect}[OF \text{inv}] \)
\( \text{note inv} = \text{LLL-invD}[OF \text{inv}] \)
\( \text{note conn} = \text{conn}[\text{folded \ inv}(1)] \)
from \( \text{inv} i \) have \( \Gr \): g-repr \( i \) \( Gr F \)
\( \text{and len}: \text{length} F = m \) by \( \text{auto} \)
from \( g\text{-}i\text{-GSO}[OF \text{Gr} i] \) \( \text{sqnorm}\text{-}g\text{-}i\text{-GSO}[OF \text{Gr} i] \)
have \( g\text{-}i \text{Gr} = \text{GSO} F (i - 1) \wedge \text{sqnorm}\text{-}g\text{-}i \text{Gr} = \text{sq-norm} (\text{GSO} F (i - 1)) \) by \( \text{simp} \)
also have \( \text{GSO} F (i - 1) = G ! (i - 1) \) unfolding \( \text{GSO-def conn using} \ i \) \( \text{len by simp} \)
finally show \( g\text{-}i \text{Gr} = G ! (i - 1) \)
\( \text{sqnorm}\text{-}g\text{-}i \text{Gr} = \text{sq-norm} (G ! (i - 1)) \) by \( \text{auto} \)
qed

lemma \( g\text{-}im1 \): assumes \( \text{inv} \): LLL-invariant \((i, Fr, Gr)\) \( F G \) and \( i < m \ i \neq 0 \)
shows \( g\text{-}im1 Gr = G ! (i - 1) \)
\( \text{sqnorm}\text{-}g\text{-}im1 Gr = \text{sq-norm} (G ! (i - 1)) \)

proof –

\( \text{note conn} = \text{LLL-connect}[OF \text{inv}] \)
\( \text{note inv} = \text{LLL-invD}[OF \text{inv}] \)
\( \text{note conn} = \text{conn}[\text{folded \ inv}(1)] \)
from \( \text{inv} i \) have \( \Gr \): g-repr \( i \) \( Gr F \)
\( \text{and len}: \text{length} F = m \) by \( \text{auto} \)
from \( g\text{-}im1\text{-GSO}[OF \text{Gr} i(2)] \) \( \text{sqnorm}\text{-}g\text{-}im1\text{-GSO}[OF \text{Gr} i(2)] \)
have \( g\text{-}im1 \text{Gr} = \text{GSO} F (i - 1) \wedge \text{sqnorm}\text{-}g\text{-}im1 \text{Gr} = \text{sq-norm} (\text{GSO} F (i - 1)) \) by \( \text{simp} \)
also have \( \text{GSO} F (i - 1) = G ! (i - 1) \) unfolding \( \text{GSO-def conn using} \ i \) \( \text{len by simp} \)
finally show \( g\text{-}im1 \text{Gr} = G ! (i - 1) \)
\( \text{sqnorm}\text{-}g\text{-}im1 \text{Gr} = \text{sq-norm} (G ! (i - 1)) \) by \( \text{auto} \)
qed

definition reduction where \( \text{reduction} = \frac{4 + \alpha}{4 * \alpha} \)

definition \( \text{dk} :: \text{nat} \Rightarrow \text{int} \text{ vec list} \Rightarrow \text{int} \) \( \text{where} \ \text{dk} k \text{ fs} = (\text{gs.Gramian-determinant} \text{ fs} k) \)

definition \( D :: \text{int} \text{ vec list} \Rightarrow \text{nat} \) \( \text{where} \ D \text{ fs} = \text{nat} (\prod i < m. \text{dk} i \text{ fs}) \)

definition \( \text{logD} :: \text{int} \text{ vec list} \Rightarrow \text{nat} \) \( \text{where} \ \text{logD} \text{ fs} = (\text{if} \ \alpha = 4/3 \ \text{then} \ (D \text{ fs}) \ \text{else} \ \text{nat} (\text{floor} (\log (1 / \text{of-rat \ reduction}) (D \text{ fs})))) \)

definition \( \text{LLL-measure} :: \text{state} \Rightarrow \text{nat} \) \( \text{where} \ \text{LLL-measure} \text{ state} = (\text{case} \ \text{state} \ \text{of} \ (i, \text{fs}, \text{gs}) \Rightarrow 2 * \text{logD} (\text{of-list-repr} \text{ fs}) + m - i) \)
lemma Gramian-determinant: assumes LLL-invariant (i,Fr,Gr) F G
and k: k ≤ m
shows of-int (gs.Gramian-determinant F k) = (∏ j<k. sq-norm (G ! j))
gs.Gramian-determinant F k > 0

proof –
let ?f = (λi. of-int-hom.vec-hom (F ! i))
note LLL = LLL-connect[OF assms(1)]
note LLLD = LLL-invD[OF assms(1)]
let ?F = map of-int-hom.vec-hom F

from LLL have lenGs: length G = m by auto
from LLLD(2-)[unfolded gram-schmidt-int-def gram-schmidt-wit-def]
have main: snd (gs.main ?F) = G and len: length ?F = m and F: set ?F ⊆ Rn
and indep: gs.lin-indpt-list ?F by (auto intro: nth-equalityI)
note conn = indep len main
have Fi: ∀ i. i < m ⇒ F ! i ∈ carrier-vec n using len F unfolding set-conv-nth
by auto
have det: gs.Gramian-determinant ?F k = (∏ j<k. ∥G ! j∥^2) (0 :: rat) <
gs.Gramian-determinant ?F k
using gs.Gramian-determinant[OF conn k] by auto
have hom: gs.Gramian-determinant ?F k = of-int (gs.Gramian-determinant F k)

unfolding gs.Gramian-determinant-def of-int-hom.hom-det[symmetric]
proof (rule arg-cong[of _ - _ det])
have cong: ∀ a b c d. a = b ⇒ c = d ⇒ a * c = b * d by auto
show gs.Gramian-matrix ?F k = map-mat of-int (gs.Gramian-matrix F k)
unfolding gs.Gramian-matrix-def Let-def
proof (subst of-int-hom.mat-mat-mul[of - k n - k], (auto)[2], rule cong)
show id: mat k n (λ (i,j). ?F ! i $ j) = map-mat of-int (mat k n (λ (i,j). F ! i $ j)) (is ?L = map-mat - ?R)
proof (rule eq-matI, goal-cases)
case (1 i j)
hence ij: i < k j < n i < length F dim-vec (F ! i) = n using len k Fi[of i] by auto
show ?case using ij by simp
qed auto
show ?L^T = map-mat of-int ?R^T unfolding id by (rule eq-matI, auto)
qed

show of-int (gs.Gramian-determinant F k) = (∏ j<k. sq-norm (G ! j))
gs.Gramian-determinant F k > (0 :: int) using det[unfolded hom] by auto

lemma LLL-dk-pos [intro]: assumes inv: LLL-invariant state F G
and k: k ≤ m
shows dk k F > 0
proof –
obtain i Gr gso where trip: state = (i, Gr, gso) by (cases state, auto)
\[ \begin{align*}
\text{proof} &
\begin{aligned}
&\text{shows} \ LLL\text{-invariant} \\
&\text{proof} - \\
&\text{have} (\prod j < m. \ dk \ j F) > 0 \\
&\text{by} \ \text{rule prod-pos, insert LLL-dk-pos}[OF \ inv], \ \text{auto} \\
&\text{thus} \ ?\text{thesis unfolding} \ D\text{-def} \ \text{by auto}
\end{aligned}
\end{align*}
\]

\text{qed}

\text{lemma} \ LLL-D-pos: \ \text{assumes} \ inv: \ LLL\text{-invariant state} \ F \ G \\
\text{shows} \ D \ F > 0 \\
\text{proof} - \\
\text{note} \ inv = \ LLL\text{-invD}[OF \ LLL] \\
\text{from} \ inv \ \text{have} \ Gr: \ g\text{-repr} \ i \ Gr \ F \ \text{and} \ Fr: \ f\text{-repr} \ i \ Fr \ F \\
\text{and} \ red: \ gs\text{-weakly-reduced} \ \alpha \ i \ G \ \text{by} \ \text{auto} \\
\text{from} \ inv \ i \ \text{inc-i-gso}[OF \ i \ Gr] \ \text{inc-i}[OF \ Fr] \\
\text{have} \ Gr': \ g\text{-repr} \ (Suc \ i) \ (\text{inc-i} \ Gr) \ F \ \text{and} \ Fr': \ f\text{-repr} \ (Suc \ i) \ (\text{inc-i} \ Fr) \ F \\
\text{by} \ \text{auto} \\
\text{from} \ red \ red-i \ \text{have} \ red: \ gs\text{-weakly-reduced} \ \alpha \ (Suc \ i) \ G \\
\text{unfolding} \ gs\text{-weakly-reduced-def} \\
\text{by} \ \text{(intro all impI, rename-tac ii, case-tac Suc ii = i, auto)} \\
\text{show} \ ?\text{thesis unfolding} \ \text{increase-i-def split} \\
\text{by} \ \text{(rule LLL-invF}[OF \ Fr' \ Gr'], \ \text{insert inv red} \ i, \ \text{auto)}
\end{align*}
\]

\text{qed}

\text{lemma} \ basis-reduction-add-row-main: \ \text{assumes} \ Linv: \ LLL\text{-invariant} \ (i,Fr,Gr) \ F \ G \\
\text{and} \ i: \ i < m \ \text{and} \ j: \ j < i \\
\text{and} \ res: \ basis\text{-reduction-add-row-main} \ (i,Fr,Gr) \ fj \ mu = ((i',Fr',Gr'), \ c) \\
\text{and} \ fj: \ fj = F ! j \\
\text{and} \ mu: \ mu = gs.\mu \ (RAT \ F) \ i \ j \\
\text{shows} \ \exists v. \ LLL\text{-invariant} \ (i',Fr',Gr') \ (F[i := v]) \ G \ \wedge \ i' = i \ \wedge \ Gr' = Gr \ \wedge \ abs \\
\text{by} \ \text{(rename fj fj, unfold sym fj)} \\
\text{proof} - \\
\text{define} \ M \ \text{where} \ M = map \ (\lambda i. \ map \ (gs.\mu \ (RAT \ F) \ i) \ [0..<m]) \ [0..<m] \\
\text{note} \ inv = \ LLL\text{-invD}[OF \ Linv] \\
\text{note} \ Gr = inv(1) \\
\text{have} \ ji: \ j \leq i \ j < m \ \text{and} \ j\strict: \ j < i \\
\text{and} \ add: \ set \ F \subseteq \ carrier-vec \ n \ i < \ length \ F \ j < \ length \ F \ i \neq j \\
\text{and} \ len: \ length \ F = m \ \text{and} \ red: \ gs\text{-weakly-reduced} \ \alpha \ i \ G
\]
and gs·snd (gram-schmidt-int n F) = G
and Fr·f-repr i Fr F
and Gr·g-repr i Gr F
and indep·gs·lin-indpt-list (RAT F)
using inv i j by auto
let ?R = rat-of-int
let ?RV = map-vec ?R
from add[unfolded set-conv-nth]
have Fij·F ! i ∈ carrier-vec n F ! j ∈ carrier-vec n by auto
let ?x = F ! i − c · v F ! j
define F1 where F1 = F[i := ?x]
let ?g = gs·gso (RAT F)
from add[unfolded set-conv-nth]
have Fi·⋀ i . i < length (RAT F) =⇒ (RAT F) ! i ∈ carrier-vec n by auto
with len j i
have gs-carr·?g j ∈ carrier-vec n
  ?g i ∈ carrier-vec n
  ∩ i . i < j =⇒ ?g i ∈ carrier-vec n
  ∩ j . j < i =⇒ ?g j ∈ carrier-vec n
  by (intro gs·gso-carrier′, force)+
have RAT-F1·RAT F1 = (RAT F)[i := (RAT F) ! i − ?R c · v (RAT F) ! j]
unfolding F1-def
proof (rule nth-equalityI[rule-format], goal-cases)
case 2 k
  show ?case
  proof (cases k = i)
    case False
    thus ?thesis using 2 by auto
  next
  case True
  hence ?thesis = (?RV (F ! i − c · v F ! j) =
    ?RV (F ! i) − ?R c · v ?RV (F ! j))
  using 2 add by auto
  also have ... by (rule eq-vecI, insert Fij, auto)
  finally show ?thesis by simp
qed
qed auto

hence RAT-F1·i·RAT F1 ! i = (RAT F) ! i − ?R c · v (RAT F) ! j (is - = -
?mui)
using i len by auto
have uminus·F ! i − c · v F ! j = F ! i + −c · v F ! j
  by (subst minus-add-uminus-vec, insert Fij, auto)
obtain G1 where gs′·snd (gram-schmidt-int n F1) = G1 by force
have F1-F·lattice-of F1 = lattice-of F unfolding F1-def uminus
  by (rule lattice-of-add[OF add, of − c], auto)
from len have len′·length (RAT F) = m by auto
from add have add′·set (map ?RV F) ⊆ carrier-vec n by auto
from add len
have k < length F =⇒ ¬ k ≠ i =⇒ F1 ! k ∈ carrier-vec n for k
unfolding $F1$-def
by (metis (no-types, lifting) nth-list-update nth-mem subset-eq carrier-dim-vec
index-minus-vec(2)
index-smul-vec(2))
hence $k < \text{length } F \implies F1 \land k \in \text{carrier-vec } n \text{ for } k$

unfolding $F1$-def using add len by (cases $k \neq i$, auto)
with len have $F1$: set $F1 \subseteq \text{carrier-vec } n \text{ length } F1 = m$
unfolding $F1$-def by (auto simp: set-cov-nth)
hence $F1$': length $(RAT F1) = m$ $SRAT F1 \subseteq \text{Rn by auto}$
from indep have dist: distinct $(RAT F)$ by (auto simp: gs.lin-indpt-list-def)
have $Fij'$: $(RAT F) ! i \in \text{Rn} (RAT F) ! j \in \text{Rn}$ using add\[\text{[unfolded set-cov-nth]}
\text{i \ j < m \ len by auto}
have $uminus'$: $(RAT F) ! i - \text{rat-of-int } c \cdot_v (RAT F) ! j = (RAT F) ! i + -$
\text{rat-of-int } c \cdot_v (RAT F) ! j$
by (subst minus-add-uminus-vec|where $n = n$, insert $Fij'$, auto)
have span-$F-F1$: gs.span $(SRAT F) = gs.span (SRAT F1)$
unfolding RAT-$F1$ $uminus'$
by (rule gs.add-vec-span, insert len add, auto)
have $\star$: \text{of-int-hom.vec-hom} $(F ! i) + - \text{rat-of-int } c \cdot_v (RAT F) ! j$
\text{ = of-int-hom.vec-hom} $(F ! i - c \cdot_v F ! j)$
by (rule eq-vecI, insert $Fij \text{ len } i \ j$, auto)
from $i \ j \text{ len}$ have $j < \text{length } (RAT F) i < \text{length } (RAT F) i \neq j$ by auto
from gs.lin-indpt-list-add-vec[of this indep, of - of-int $c$]
have $gs.lin-indpt-list ($(RAT F) [i := (RAT F) ! i + - of-int c \cdot_v (RAT F) ! j])$
(is gs.lin-indpt-list $?!F1$).
also have $?!F1 = RAT F1$ unfolding $F1$-def using $i \ text{ len } Fij'$ $\star$
by (auto simp: map-update)
finally have indep-$F1$: gs.lin-indpt-list $(RAT F1)$.
note $\text{conn1} = \text{indep len'} gs[\text{unfolded gram-schmidt-int-def gram-schmidt-wit-def}]$
note $\text{conn2} = \text{indep-F1 F1}'(1) gs[\text{unfolded gram-schmidt-int-def gram-schmidt-wit-def}]$
from gs.main-connect|OF conn1| gs.main-connect|OF conn2|
have $G$-def: $G = \text{map } ?g [0..< m] G = \text{gram-schmidt } n (RAT F)$
and $G1$-def: $G1 = \text{map } (gs.gso (RAT F1)) [0..< m] G1 = \text{gram-schmidt } n$
$(RAT F1)$
by (auto simp: o-def)
from gs.gram-schmidt|OF conn1| gs.gram-schmidt|OF conn2| F1 len
have span-$G$-$G1$: gs.span $(set G) = gs.span (set G1)$
and len$G$: length $G = m$
and $G1$: $i < \text{length } G \implies G ! i \in \text{Rn}$
and $G1i$: $i < \text{length } G1 \implies G1 ! i \in \text{Rn}$ for $i$
by auto
have eq: $x \neq i \implies RAT F1 ! x = (RAT F) ! x$ for $x$
unfolding RAT-$F1$ by auto
hence eq-part: $x < i \implies gs.gso (RAT F1) x = gs.gso (RAT F) x$ for $x$
by (intro gs-gs-identical, insert len, auto)
have $G$: $i < m \implies (RAT F) ! i \in \text{Rn}$
by (insert add len', auto)
note carr1[intro] = this[OF i] this[OF ji(2)]
from $G1[\text{unfolded gs.main-connect|OF conn1}] G1i[\text{unfolded gs.main-connect|OF
\[
\text{conn2}]
\]

\[\text{have } x < m \implies \exists g \in Rn \quad x < m \implies gs.gso (RAT F1) x \in Rn \quad x < m \implies \text{dim-vec} (gs.gso (RAT F1) x) = n \quad x < m \implies \text{dim-vec} (gs.gso (RAT F) x) = n
\]

\[\text{for } x \text{ by (auto simp:o-def)}
\]

\[\text{hence carr2[introl]: } \exists g \in Rn \text{ gs.gso (RAT F1) } i \in Rn
\]

\[\text{?g} : \{0..<i\} \subseteq Rn
\]

\[\text{?g} : \{0..<\text{Suc } i\} \subseteq Rn \text{ using } i \text{ by auto}
\]

\[\text{have } F1-RV: \exists \text{RV} (F1 ! i) = \text{RAT F1 ! i using } i \text{ F1 by auto}
\]

\[\text{have } F-RV: \exists \text{RV} (F ! i) = (\text{RAT F} ! i \text{ using } i \text{ len by auto)
}\]

\[\text{have } x < i \implies gs.gso (RAT F1) x = gs.gso (RAT F) x \text{ for } x \text{ using eq by (rule gs-gs-identical, auto)
}\]

\[\text{hence span-G1-G: gs.span (gs.gso (RAT F1) ! \{0..<i\}) = gs.span (gs.gso (RAT F)) ! \{0..<i\}) (is } ?ls = ?rs
\]

\[\text{apply(intro cong\[\text{OF refl[of gs.span]],rule image-cong\[\text{OF refl]} \text{ using eq by auto)
}\]

\[\text{have } (\text{RAT F1}) ! i = gs.gso (RAT F1) i = ((\text{RAT F} ! i) = gs.gso (RAT F1) i) - ?mui
\]

\[\text{unfolding } \text{RAT-F1-i using carr1 carr2 by (intro eq-vec1, auto)
}\]

\[\text{hence m1: } ((\text{RAT F}) ! i = gs.gso (RAT F1) i) = ?mui \in ?rs
\]

\[\text{using } \text{gram-schmidt.projection-exist\[\text{OF conn2 i]
}\]

\[\text{unfolding } \text{span-G1-G by auto
}\]

\[\text{from } j < i \text{ have } Gj-mem: (\text{RAT F} ! j \in (\lambda x. ((\text{RAT F} ! x)) ) ! \{0..<i\} \text{ by auto
}\]

\[\text{have id1: set (take i (map of-int-hom vec-hom F)) = (\lambda x. of-int-hom vec-hom (F ! x)) \{0..<i\)
\]

\[\text{using } i \leq m \text{ len by (subst nth-image[symmetric], force+)
}\]

\[\text{have } (\text{RAT F}) ! j \in \?rs \iff (\text{RAT F}) ! j \in \text{gs.span ((\lambda x. ?RV (F ! x)) \{0..<i\})
\]

\[\text{unfolding } \text{gs.partial-span\[\text{OF conn1 } i \leq m] \text{ id1 ..
}\]

\[\text{also have } (\lambda x. ?RV (F ! x)) \{0..<i\} = (\lambda x. ((\text{RAT F} ! x)) \{0..<i\} \text{ using i < m len by force
}\]

\[\text{also have } (\text{RAT F}) ! j \in \?rs
\]

\[\text{by (rule gs.span-mem[OF - Gj-mem], insert } i < m \text{ G, auto)
}\]

\[\text{finally have } (\text{RAT F}) ! j \in \?rs .
\]

\[\text{hence m2: } ?mui \in \?rs
\]

\[\text{apply(intro gs.prod-in-span) by force+}
\]

\[\text{have ineq:((RAT F) ! i = gs.gso (RAT F1) i) + ?mui - ?mui = ((RAT F) ! i = gs.gso (RAT F1) i)
\]

\[\text{using carr1 carr2 by (intro eq-vec1, auto)
}\]

\[\text{have cong’: } A \Rightarrow A \subseteq B \Rightarrow B \subseteq C \text{ for } A B :: \text{ a vec and } C \text{ by auto}
\]

\[\text{have *: gs.gso (RAT F) \{0..<i\} \subseteq Rn by auto
\]

\[\text{have in-span: } (\text{RAT F}) ! i = gs.gso (RAT F1) i \in \?rs
\]

\[\text{by (rule cong\[\text{OF eq-vec1 gs.span-add1[OF * in1 in2,unfolded ineq]], insert carr1 carr2, auto)
\]

\{
\[
\begin{align*}
\text{fix } x \& \text{ assume } x: x < i \text{ hence } x < m \& i \neq x \text{ using } i \text{ by auto} \\
\text{from } \text{gram-schmidt.orthogonal}[\text{OF conn2}, \text{OF i this}] \text{ this} \\
\text{have } gs.gso (\text{RAT F1}) i \cdot gs.gso (\text{RAT F1}) x = 0 \text{ by auto} \\
\end{align*}
\]

\(\text{hence G1-G: gs.gso (RAT F1) i = gs.gso (RAT F) i}\)

apply\(\text{intro gram-schmidt.projection-unique[OF conn1 i gs.gso-carrier[OF conn2 i]]}\)

using \(\text{in-span by (auto simp: eq-part[symmetric])}\)

have \(\text{eq-fs:x < m \Rightarrow gs.gso (RAT F1) x = gs.gso (RAT F) x}\)

for \(x\) proof\(\text{induct x rule:nat-less-induct[rule-format]}\)

\(\text{case (1 x)}\)

\(\text{hence ind: m < x \Rightarrow gs.gso (RAT F1) m = gs.gso (RAT F) m}\)

for \(m\) by auto

\{ assume \(x > i\)

\(\text{hence ?case apply(subst (1 2) gs.gso.simps) unfolding gs.\mu.simps}\)

apply\(\text{rule cong[OF - cong[OF refl[of gs.sumlist]]]}\)

using \(\text{ind eq by auto}\)

\}\ note \(\text{eq-rest = this}\)

show \(?\text{case by (rule linorder-class.linorder-cases[of x i],insert G1-G eq-part eq-rest,auto)}\)

\(\text{qed}\)

with \(\text{G-def G1-def cof-vec-space.gram-schmidt-result}\)

have \(\text{Hs:G1 = G by (auto simp:o-def)}\)

\(\text{hence red: gs.weakly-reduced } \alpha \text{ i G1 using red by auto}\)

let \(?\text{Mi = M ! i ! j}\)

let \(?\text{x'} = \text{get-nth-i Fr = floor-ceil } ?\text{Mi } \alpha F ! j\)

\(\text{define Fr1 where Fr1 = update-i Fr } ?x'\)

have \(\text{Hr: update-i Fr (} ?x') = Fr1 \text{ unfolding Fr1-def by simp}\)

have \(\text{Gjn: dim-vec (} F ! j\) = n using \(\text{Fi}(2) \text{ carrier-vecD by blast}\)

\(\text{define E where E = addrow-mat m (} - ?R c \text{ ) i j}\)

\(\text{define M' where M' = gs.M (RAT F) m}\)

\(\text{define N' where N' = gs.M (RAT F1) m}\)

have \(\text{E: E } \in \text{ carrier-mat m m unfolding E-def by simp}\)

have \(\text{M: M'} \in \text{ carrier-mat m m unfolding gram-schmidt.M-def M'-def by auto}\)

have \(\text{N: N'} \in \text{ carrier-mat m m unfolding gram-schmidt.M-def N'-def by auto}\)

let \(?\text{GsM = gs.Fs G}\)

have \(\text{Gs: } ?\text{GsM } \in \text{ carrier-mat m n using G-def by auto}\)

\(\text{hence GsT: } ?\text{GsM} T \in \text{ carrier-mat n m by auto}\)

have \(\text{Gnn: gs.Fs (RAT F) } \in \text{ carrier-mat m n unfolding mat-of-rows-def using len by auto}\)

have \(\text{gs.Fs (RAT F1) = addrow (} - ?R c \text{ ) i j (gs.Fs (RAT F))}\)

\(\text{unfolding RAT-F1 by (rule eq-matI, insert Gjn ji(2), auto simp: len-of-rows-def)}\)

also have \(\ldots = E * gs.Fs (RAT F) \text{ unfolding E-def}\)

by \(\text{rule addrow-mat, insert j i, auto simp: mat-of-rows-def len}\)

finally have \(\text{HEG: gs.Fs (RAT F1) = E * gs.Fs (RAT F)}\)

have \((E * M') * gs.Fs G = E * (M' * gs.Fs G)\)

by \(\text{rule assoc-mult-mat[OF E M Gs]}\)

also have \(\text{M' * ?GsM = gs.Fs (RAT F) using gs.matrix-equality[OF conn1]}\)

\(\text{M'-def by simp}\)
also have $E \ast \ldots = g_s.Fs$ (RAT F1) unfolding HEG ..
also have $\ldots = N' \ast g_s.Fs$ G1 using g.s.matrix-equality[OF conn2] $N'$-def by simp
also have $g_s.Fs$ G1 = $?GsM$ unfolding Hs ..
finally have $\langle E \ast M' \rangle ^* \ast ?GsM = N' \ast ?GsM$ ,
from arg-cong[OF this, of $\lambda x. x \ast ?GsM^T$] E M N
have EMN: $\langle E \ast M' \rangle ^* \ast \langle ?GsM \ast ?GsM^T \rangle = N' \ast \langle ?GsM \ast ?GsM^T \rangle$
by (subst (1 2) assoc-mult-map[OF - Gs GsT, of - m, symmetric], auto)
have det $?GsM \ast ?GsM^T = g_s.Gramian-determinant G m
unfolding g_s.Gramian-determinant-def
by (subst g_s.Gramian-matrix-alt-def, auto simp: Let-def G-def)
also have $\ldots > 0$
by (rule g_s.Gramian-determinant(2))[OF g_s.orthogonal-imp-lin-indpt-list $\langle$ length $G = m$],
insert g_s.gram-schmidt(2-)[OF conn1], auto)
finally have det $?GsM \ast ?GsM^T \neq 0$ by simp
from vec-space.det-nonzero-congruence[OF EMN this - - N] Gs E M
have EMN: $E \ast M' = N'$ by auto

have Mij: $mu = M ! i ! j$ unfolding M-def mu using $i < m$ ji(2) by auto
from res[unfolded Mij] have $c, c = \text{floor-ceil } (M ! i ! j)$
by (auto simp: Let-def split: if-splits)
have x: $?x = $?x' by (subst get-nth-i[OF Fr], insert add, auto simp: c Mij)
{
  assume $c0: c = 0$
  have Fr1 = update-i Fr (F ! i) unfolding Fr1-def x[symmetric] $c0$
    using $F ! i \in \text{carrier-vec n} \ (F ! j \in \text{carrier-vec n})$ by auto
  from update-i[OF Fr, of $F ! i$], folded this] i len
  have list-repr i Fr1 $\langle F[i := (F ! i)] \rangle$ by auto
  also have $F[i := F ! i] = F$
    by (rule nth-equalityI, force, intro allI, rename-tac j, insert i len, case-tac j
    = i, auto)
  finally have f-repr i Fr1 F ,
    with Fr have Fr1 = Fr unfolding list-repr-def by (cases Fr, cases Fr1, auto)
  } note $c0 = this$
from res[unfolded basis-reduction-add-rov-main.simps Let-def fi Mij Hr]
have res: $i' = i Fr' = Fr1 Gr' = Gr$ using Mij $c0 \ i \ len$
by (auto simp: j split: if-splits)
{
  from Gr[unfolded g-repr-def] i
  have list-repr i Gr $\langle$ map $\langle$Ax. $\langle$x, $\|x\|^2\rangle\rangle$ (map (GSO F) $[0..<m]$)]
    (is list-repr - $\langle$ map $\langle$GSO F $\rangle$ $\langle$is$\rangle$)\rangle]
    by auto
  also have map (GSO F) $?is = map (GSO F1) $?is (is $?l = $?r)
  proof (rule nth-equalityI, force, unfold length-map, intro allI impI, goal-cases)
    case (1 ii)
    hence $ii: ii < m$ using i by auto
  from i have id: $?is ! ii = ii$
    by (metis add.commute add.right-neutral diff-zero length-upt nth-upt)
have \( GSO F \) \( ii = GSO F1 \) \( ii \) using \( \text{arg-cong}[OF Hs[unfolded G1-def G-def]] \), of \( \lambda x. x \) \( ! \) \( ii \) \( ! \)

unfolding \( \text{GSO-def} \) by auto
thus \( \text{case}\) unfolding \( \text{nth-map}[OF 1] \) \( id \).

qed
finally have \( \text{list-repr i Gr (map \( \varphi f (\text{map (GSO F1) [0..<m]))}) \).}

hence \( \text{gsso}^\varphi: \text{g-repr i Gr} \ F1 \text{ unfolding g-repr-def using i by simp} \)

have \( \text{repr'}: \text{f-repr i Fr1 F1 unfolding F1-def Fr1-def x map-update} \)
by (rule update-i[OF Fr], insert add, auto)

have \( \text{LLL-invariant (i, Fr1, Gr) F1 G unfolding Hs[symmetric]} \)

apply (rule LLL-invI[OF repr' gso'' G1-def(2)][folded snd-gram-schmidt-int,symmetric]

- red \( \text{inv(7)})\)

by (insert \( \text{F1 F1-F inv(5) indep-F1, auto}) \)

}\ note \( \text{inv-gso = this} \)
\{
fix \( \text{ia}\) assume \( \text{ia} \leq j\) hence \( \text{ia} < i\) using \( \text{ji j by auto}\)

hence \( \text{(RAT F1)}!\) \( \text{ia} = (\text{RAT F})!\) \( \text{ia}\)

using \( \text{F1-def i len by auto} \)
\}

hence \( \text{fs-eq:gs.gso (RAT F1) j = gs.gso (RAT F) j}\)
by (intro gs-gs-identical, auto)

have \( \text{dim:dim-vec a = dim-vec b \Rightarrow ?RV (a + b) = ?RV a + ?RV b for a b}\)
by auto
from \( \text{gs.gso-times-self-is-norm[OF conn1 ji(2)]}\)

have \( \text{gs-norm:(RAT F) ! j \cdot gs.gso (RAT F) j = \|gs.gso (RAT F) j\|^2}\)
by auto

have \( \text{fc:floor-ceil (0::rat)} = 0\) unfolding \( \text{floor-ceil-def by linarith}\)
\{
\assume \( \text{sq-norm-vec (gs.gso (RAT F) j)} = 0\)
\hence \( \text{gs.gso (RAT F) j = 0 \_ n using gs-carr(1) sq-norm-vec-eq-0 len by force}\)
\hence \( \text{c = 0}\) unfolding \( \text{c M-def gs.µ.simps using j i fc by auto}\)
\}

note \( \text{zero = this}\)

\from \( \text{ij < i } \) have \( \text{if-True: (if j < i then t else e)} = \text{t for t e by simp}\)

have \( \text{id1 : ?RV (F ! j) = (RAT F) ! j using ji len by auto}\)

have \( \text{id: (RAT F1) ! i = (RAT F1) ! i - ?RV c · ν \ ?RV (F ! j) unfolding F1-def}\)
using \( \text{i len Fi j by auto}\)

have \( \text{mudiff: mu = (RAT F) ! i \cdot gs.gso (RAT F) j / \|gs.gso (RAT F) j\|^2}\)
\( \text{unfolding mu gs.µ.simps if-True id using ji(2) by auto}\)

have \( \text{mudiff:mu = of-int c = gs.µ (RAT F1) i j}\)

unfolding \( \text{mudiff unfolding gs.µ.simps fs-eq if-True id}\)
\apply \( \text{subst minus-scalar-prod-distrib, (insert Fi j gs-carr, auto)[3]}\)
\apply \( \text{(subst scalar-prod-smult-left, (insert Fi j gs-carr, auto)[1]}\)
\apply \( \text{(unfold id1 gs-norm)}\)
using \( \text{zero divide-diff-eq-iff by fastforce}\)
\{ 
\fix \( \text{i' j'}\)
assume \( \text{ij: i' < m j' < m and choice: i' \neq i \lor j < j'}\)

have \( \text{gs.µ (RAT F1) i' j'}\)
\( = N \_ ñ (i' j') using ji F1 unfolding N'-def gs.µ-def by auto\)
also have \ldots = \text{addrow} (- \ ?R c) i j M' \Longrightarrow (i',j') \text{ unfolding \ EMN[symmetric]}

\text{E-def}

\begin{itemize}
\item by (\text{subst \ addrow-mat}[OF \ M], \ insert \ ji, \ auto)
\item also have \ldots = (if \ i = i' \ then \ - \ ?R c * M' \Longrightarrow (j, j') + M' \Longrightarrow (i, j') \ else \ M' \Longrightarrow (i', j'))
\item also have \ldots = \text{addrow} M, \ auto
\end{itemize}

also have \ldots = M' \Longrightarrow (i', j')
proof (cases \ i = i')
\begin{itemize}
\item case True
\item with choice have jj: j < j' by auto
\item have M' \Longrightarrow (j, j') = gs.\mu (RAT F) j j'
\item using ij ji len unfolding M'-def gs.M-def by auto
\item also have \ldots = 0 unfolding gs.\mu.simps using jj by auto
\item finally show \ ?thesis using True by auto
\item qed auto
\item also have \ldots = gs.\mu (RAT F) i' j'
\item using ij len unfolding M'-def gs.M-def by auto
\item also note calculation
\end{itemize}
\} note \ mu-change = this
\begin{itemize}
\item have abs (\mu - of-int c) \leq inverse 2 unfolding res j Mij c
\item by (rule floor-ceil)
\item thus \ ?thesis using mu-change inv-gso mudiff unfolding res j F1-def by auto
\item qed
\end{itemize}

\textbf{lemma} basis-reduction-add-row-i-im1: assumes \text{Linvo} \: LLL-invariant \: (i,Fr,Gr) \: F \: G
and \: i: i < m \: and \: i0: \: i \neq 0
and \: res: \: \text{basis-reduction-add-row-i-im1} \: ((i',Fr',Gr'), \: \mu )
shows \: \exists \: i'. \: LLL-invariant \: (i',Fr',Gr') \: F' \: G \: \land \: i' = i \: \land \: Gr' = Gr \: \land \: abs \: \mu \leq inverse 2 \: \land
\mu = gs.\mu (RAT F') i (i-1)
proof -
\begin{itemize}
\item note inv = LLL-invD[OF Linv]
\item from \: i \: have \: im1: \: i - 1 < m by auto
\item from \: i0 \: have \: im1': \: i - 1 < i \: by \: auto
\item from \: inv \: have \: G: \: length \: F = m \: by \: auto
\item have \: fst: \: get-nth-im1 \: Fr = \: F ! (i - 1) \: unfolding \: get-nth-im1[OF \ inv(5) \ i0]
\item using \: \mu \: im1 \: G
\item by \: auto
\item have \: Gi: \: F ! i \: \in \: \text{carrier-vec} \: n \: \text{using} \: i \: \text{inv(3,4)} \: \text{by} \: \text{auto}
\item have \: gs-gs: \: gs.gso (RAT F) (i - 1) \in Rn
\item by \: (rule gs.gso-carrier', \: insert \: i \: \text{inv(3,4)}, \: \text{auto})
\item note \: res = res[unfolded \: basis-reduction-add-row-i-im1-def \: Let-def \: split \: \text{fst}]
\item from \: res \: obtain \: c \: \text{where}
\item res': \: \text{basis-reduction-add-row-main} \: ((i, \: Fr, \: Gr) \: (F \: (i - 1)) \: (\mu - i-im1 \: Fr \: Gr))
\item = (\mu'i-im1 \: Fr \: Gr', \: c)
\item (is \: \text{?res = -}) \: \text{by} \: \text{(cases \: \text{?res, \: auto})}
\item from \: res[unfolded \: res'] \: \text{have} \: mu: \: mu = \: \mu - i-im1 \: Fr \: Gr \: \text{- of-int} \: c \: \text{by} \: \text{auto}
\item have \: id: \: \mu - i-im1 \: Fr \: Gr = \: gs.\mu (RAT F) i (i - 1)
\end{itemize}
proof

- shows \( \exists F \ G \)

lemma basis-reduction-add-row-i-all: fixes Gr assumes Linv: LLL-invariant \((i,Fr,Gr)\) 

| \( \mu \)-im1 Fr Gr - of-int c | \( \leq \) inverse 2 

thus \(?\)thesis unfolding ma by auto

qed

|\mu \)-im1 Fr Gr - of-int c = gs.\( \mu \) (RAT \( G' \)) \( i \) \((i - 1)\) by auto
have \( G_i : F \! \mid i \in \text{carrier-vec} \ n \) using \( \text{inv}(3,4) \) i by auto

have \( \text{gs-gs} : \text{gram-schmidt.gso} \ n \ (\text{RAT} F) \) ii \( \in \text{R}n \) by (rule \( \text{gram-schmidt.gso-carrier}' \), insert \( \text{inv}(3,4) \) i ii, auto)

from \( \text{get-nth-i}(\text{OF inv}(\text{8})) \) inv(\( \text{4} \)) i \( G_i \) \( \text{gs-gs} \)

have pair: \( \mu_{ij} \ (\text{get-nth-i} \ F_i) \ ?\text{fsn} = \)$

unfolding \( \mu_{ij-def} \) split \( \text{gs-\mu}.\text{simps} \) ii if-True id by auto

from \( \text{basis-reduction-add-row-main}(\text{OF Suc}(\text{4})) \) ii(\( \text{1} \)) main refl pair

obtain \( v \) where

\( \text{Linv: LLL-invariant} \ (i, \ F_{ij}', \ Gr) \ (F[i := v]) \ G \)

and id: \( i'' = i \) \( G_{ii}'' = G_r \)

and small: \( \exists \text{small} \ (F[i := v]) \ ii \)

and \( \text{id-'}: \ A \ i' j' : i' < m \implies j' < m \implies (i' \neq i \lor ii < j') \implies \)

\( \text{gs-\mu} \ (\text{RAT} \ (F[i := v])) \ i' j' = \)

\( \text{gs-\mu} \ (\text{RAT} \ F) \ i' j' \) by auto

let \( ?G = F[i := v] \)

from \( \text{inv Suc}(\text{3}) \) have lt: \( ii < \text{length} \ F \) by auto

have (rev (map \( (\lambda x. \ (x, \ ||x||^2)) \) \ (map \( (\text{gs.gso} \ (\text{RAT} F)) \ [\emptyset..<ii]) \)))

= \( (?\text{fsn} \ # \ \text{rev} \ (\text{map} \ (\lambda x. \ (x, \ ||x||^2)) \ (\text{map} \ (\text{gs.gso} \ (\text{RAT} F)) \ [\emptyset..<ii]) \))) \)

by auto

from id Suc(\( \text{2} \)) unfolded rev-take-Suc(\( \text{OF lt} \)) this basis-reduction-add-row-i-all-main.simps

split Let-def main

have \( \text{basis-reduction-add-row-i-all-main}' ((i, \ F_{i}'', \ Gr)) \ (\text{rev} \ (\text{take} \ ii \ F)) \)

(\( \text{rev} \ (\text{map} \ (\lambda x. \ (x, \ ||x||^2)) \) \ (\text{map} \ (\text{gs.gso} \ (\text{RAT} F)) \ [\emptyset..<ii]) \) =

\( (i'', \ F_{i}'', \ Gr') \) by auto

also have \( \text{rev} \ (\text{take} \ ii \ F) = \text{rev} \ (\text{take} \ ii \ ?G) \) using ii by auto

also have \( \text{map} \ (\text{gs.gso} \ (\text{RAT} F)) \ [\emptyset..<ii] = \text{map} \ (\text{gs.gso} \ (\text{RAT} ?G)) \ [\emptyset..<ii] \)

by (rule nth-equality1, auto intro: gs-gs-identical, insert inv(\( \text{4} \) i ii, auto)

finally have res: \( \text{basis-reduction-add-row-i-all-main}' ((i, \ F_{i}'', \ Gr)) \ (\text{rev} \ (\text{take} \ ii \ ?G)) \)

(\( \text{rev} \ (\text{map} \ (\lambda x. \ (x, \ ||x||^2)) \) \ (\text{map} \ (\text{gs.gso} \ (\text{RAT} ?G)) \ [\emptyset..<ii]) \) =

\( (i'', \ F_{i}'', \ Gr') \) by auto

from ii have ii- le: \( ii \leq i \) by auto

have small: \( \forall j \geq ii. \ j < i \implies \text{small} \ ?G \ j \)

proof (intro allI impI)

fix \( j \)

assume *: \( ii \leq j j < i \)

show \( \text{small} \ ?G \ j \)

proof (cases \( j = ii \))

case True

with small show \( \text{thesis} \) by auto

next

case False

with * Suc(\( \text{5} \)) rule-format, of \( j \)

have small: \( \text{small} \ F \ j \) by auto

with \( \text{id-'}\text{rule-format, OF \( i, \ of \) \( j \) \( \times \) False} \)

show \( \text{thesis} \) by auto

qed
qed
from Suc(1)[OF res ii-le Linv small] obtain G' where
  Linv: LLL-invariant (i, Fr', Gr) G' G
  and i': i' = i
  and gso': Gr' = Gr
  and small: (∀ j<i. ?small G' j)
  and id: \( \forall i' j'. i' < m \implies j' < m \implies i' \not= i \implies \) 
  gs.\( \mu \) (RAT G') i' j' = 
  gs.\( \mu \) (RAT ?G) i' j' by blast
show ?case
proof (intro exI conjI, rule Linv, rule i', rule gso', rule small, intro allI impI, goal-cases)
  case (1 i' j')
  show ?case unfolding id [OF 1]
  by (rule id' [rule-format], insert 1 i', auto)
qed
qed auto

lemma basis-reduction-part-2-main: fixes Gr assumes Linv: LLL-invariant state F G
  and n: fst state = m
  and res: basis-reduction-part-2-main state = (i, Fr, Gr)
  shows \( \exists F'. LLL\text{-invariant} (i, Fr, Gr) F' G \land i = 0 \land \) 
  gs.strictly-reduced m \( \alpha \) G (gs.\( \mu \) (RAT F'))
proof -
from n obtain i1 Fr1 Gr1 where state: state = (i1, Fr1, Gr1)
  and i1: i1 = m i1 \leq m
  by (cases state, auto)
note Linv = Linv[unfolded state]
from LLL-invD[OF Linv] i1
  have weak: gs.weakly-reduced m \( \alpha \) m G by auto
let \( ?small = \lambda F. i. \forall j. j < i \implies abs (gs.\( \mu \) (RAT F) i j) \leq 1/2 \)
  have small: \( \forall i. i1 \leq i \implies i < m \implies ?small F i \) unfolding i1 by auto
from res[unfolded state] small Linv (i1 \leq m)
show ?thesis
proof (induct i1 arbitrary: F Fr1 Gr1)
  case (0 F Fr Gr)
  thus ?thesis using weak by (auto intro!: exI[of - F] simp: gs.strictly-reduced-def o-def)
next
  case (Suc i1 F Fr1 Gr1)
from Suc(4)[OF res ii-le Linv small] unfolding LLL-invariant-def by auto
hence 1: list-repr (Suc i1) Fr1 F
using of-list-repr by blast
have Linv: LLL-invariant (i1, dec-i Fr1, dec-i Gr1) F G
  using Suc(4) unfolding LLL-invariant-def split

136
by (auto simp: g-repr-def dec-i Suc gs.weakly-reduced-def *)

obtain i2 Fr2 Gr2 where
call: basis-reduction-add-row-i-all (i1, dec-i Fr1, dec-i Gr1) = (i2, Fr2, Gr2)

(is ?call = -) by (cases ?call, auto)
from Suc(3–) have i1: i1 < m i1 ≤ m by auto
from basis-reduction-add-row-i-all[OF Linv iI(1) call]

obtain F' where
Linv: LLL-invariant (i1, Fr2, dec-i Gr1) F' G
and i2: i2 = i1
and gso2: Gr2 = dec-i Gr1
and small: ?small F' i1
and id: ∃ i' j'. i' < m =⇒ j' < m =⇒ i' ≠ i1 =⇒
  gs.µ (RAT F') i' j' =
gs.µ (RAT F) i' j' by auto
from Suc(2)[unfolded basis-reduction-part-2-main.simps[of Suc iI]] call i2 gso2

have res: basis-reduction-part-2-main (i1, Fr2, dec-i Gr1) = (i, Fr, Gr) by auto

show ?case
proof (rule Suc(1)[OF res - Linv iI(2)], intro allI impI, goal-cases)
case (1 i j)
  thus ?case using small[rule-format, of j] Suc(3)[rule-format, of i j] id[of i j] i1 by (cases i = i1, auto simp: o-def)
qed
qed

context

assumes α: α ≥ 4/3
begin

lemma α0: α > 0 α ≠ 0
  using α by auto

lemma reduction: 0 < reduction reduction ≤ 1
  α > 4/3 =⇒ reduction < 1
  α = 4/3 =⇒ reduction = 1
  using α unfolding reduction-def by auto

lemma dk-swap-unchanged: assumes len: length F1 = m
  and i0: i ≠ 0 and i: i < m and ki: k ≠ i and km: k ≤ m
  and swap: F2 = F1[i := F1 ! (i − 1), i − 1 := F1 ! i]
  shows dk k F1 = dk k F2
proof
  let F1-M = mat k n (λ(i, y). F1 ! i $ y)
  let F2-M = mat k n (λ(i, y). F2 ! i $ y)
  have ∃ P. P ∈ carrier-mat k k ∧ det P ∈ {-1, 1} ∧ F2-M = P * F1-M
  proof cases
    assume ki: k < i

137
hence $H$: $?F2-M = ?F1-M$ unfolding swap

by (intro eq-matI, auto)

let $?P = 1_m k$

have $?P \in carrier-mat k k$ det $?P \in \{-1, 1\}$ $?F2-M = ?P * ?F1-M$ unfolding $H$

by auto

thus $thesis$ by blast

next

assume $\neg k < i$

with $ki$ have $ki: k > i$ by auto

let $?P = swaprows-mat k i (i - 1)$

from $i0$ $ki$ have $neq: i \neq i - 1$ and $kmi: i - 1 < k$ by auto

have $*: i?P \in carrier-mat k k$ det $?P \in \{-1, 1\}$ using det-swaprows-mat[OF $ki kmi neq$]

by auto

from $i len$ have $iH: i < length F1 i - 1 < length F1$ by auto

have $?P * ?F1-M = swaprows i (i - 1) ?F1-M$

by (subst swaprows-mat[OF - ki kmi], auto)

also have $\ldots = ?F2-M$ unfolding swap

by (intro eq-matI, rename-tac ii jj,

case-tac ii = i, (insert iH, simp add: nth-list-update)[1],

case-tac ii = i - 1, insert iH neq ki, auto simp: nth-list-update)

finally show $thesis$ using $*$ by metis

qed

then obtain $P$ where $P: P \in carrier-mat k k$ and $detP: det P \in \{-1, 1\}$ and

$H': ?F2-M = P * ?F1-M$ by auto

have dk k $F2 = det (gs.Gramian-matrix F2 k)$

unfolding dk-def gs.Gramian-determinant-def by simp

also have $\ldots = det (?F2-M * ?F2-M^T)$ unfolding gs.Gramian-matrix-def

Let-def by simp

also have $?F2-M * ?F2-M^T = ?F2-M * (?F1-M^T * P^T)$ unfolding $H'$

by (subst transpose-mult[OF $P$], auto)

also have $\ldots = P * (?F1-M * (?F1-M^T * P^T))$ unfolding $H'$

by (subst assoc-mult-mat[OF $P$], auto)

also have $det \ldots = det P * det (?F1-M * (?F1-M^T * P^T))$

by (rule det-mult[OF $P$], insert $P$, auto)

also have $?F1-M * (?F1-M^T * P^T) = (?F1-M * ?F1-M^T) * P^T$

by (subst assoc-mult-mat, insert $P$, auto)

also have $det \ldots = det (?F1-M * ?F1-M^T) * det P$

by (subst det-mult, insert $P$, auto simp: det-transpose)

also have $det (?F1-M * ?F1-M^T) = det (gs.Gramian-matrix F1 k)$ unfolding gs.Gramian-matrix-def Let-def by simp

also have $\ldots = dk k F1$

unfolding dk-def gs.Gramian-determinant-def by simp

finally have $dk k F2 = (det P * det P) * dk k F1$ by simp

also have $det P * det P = 1$ using detP by auto

finally show $dk k F1 = dk k F2$ by simp

qed

lemma basis-reduction-step: assumes inv: LLL-invariant $(i, Fr, Gr) F G$

138
and $i : i < m$

and res: basis-reduction-step $\alpha (i, Fr, Gr) =$ state

shows $(\exists F' G'. \text{LLL-invariant state } F' G')$

and LLL-measure state < LLL-measure $(i, Fr, Gr)$

proof (atomize(full), cases $i = 0$)

case $i0$: False

note res = res[unfolded basis-reduction-step-def split]

obtain $i1 Fr1 Gr1 mu$ where
    $i1$: basis-reduction-add-row-i-im1 $(i, Fr, Gr) = ((i1, Fr1, Gr1), mu)$ (is $?b = -$)
    by (cases $?b$, auto)

from basis-reduction-add-row-i-im1[OF inv i $i0$ $i1$] obtain $F1$
    where $Line'': \text{LLL-invariant } (i, Fr1, Gr1) F1 G$ and $ii: i1 = i$
    and $m12$: $|mu| \leq \text{inverse } 2$
    and $mu$: $mu = gs.\mu$ (RAT $F1$) $i$ ($i - 1$) by auto

note $dk = dk-def$

note $Gd = \text{Gramian-determinant}(1)$

note $Gd12 = Gd[\text{OF inv}] Gd[\text{OF Linv}]$

have $dk-eq: k \leq m \implies dk k F = dk k F1$ for $k$
    unfolding $dk$ using $Gd12[of k]$ by auto

have $D-eq: D F = D F1$ unfolding $D-def$
    by (rule arg-cong[of $-$ nat], rule prod.cong, insert $dk-eq$, auto)

hence $logD-eq: logD F = logD F1$ unfolding $logD-def$ by simp

note $inv = \text{LLL-invD}[OF inv]$

note $inv' = \text{LLL-invD}[OF Linv]$

from $inv$ have $repr: f-repr i Fr F$ by auto

note res = res[unfolded basis-reduction-step-def this split il id Let-def]

let $?x' = G ! (i - 1)$ let $?y' = G ! i$

let $\ell$: $\text{sq-norm-g-im1 Gr1}$ let $?y' = \text{sq-norm-g-i Gr1}$

let $\ell$: $\alpha \ast \text{sq-norm} ?y' < \text{sq-norm} ?x$

let $\ell$: $\alpha \ast ?y' < ?x'$

from $inv'$ have $red: gs.weakly-reduced \alpha i G$
    and $repr: f-repr i Fr1 F1$ and $gs: \text{snd (gram-schmidt-int n F1)} = G$
    and $len$: $\text{length F1} = m$ and $HC$: $\text{set F1} \subseteq \text{carrier-vec n}$
    and $gso$: $g-repr i Gr1 F1$ and $L$: $\text{lattice-of F1} = L$

using $i$ by auto

from g-i[OF Linv' $i$] have $y$: $?y' = \text{sq-norm} ?y$ by auto

from g-im1[OF Linv' $i0$] have $x$: $?x' = \text{sq-norm} ?x$ by auto

hence cond: $\ell$: cond' = ?cond using $y$ by auto

from $i0$ have $(i = 0) =$ False by auto

note res = res[unfolded cond fst-conv this if-False]

show $(\exists H. \exists (LLL-invariant state H)) \land \text{LLL-measure state < LLL-measure} (i, Fr, Gr)$

proof (cases $?cond$)

case False
    from len inc-i[OF repr] repr i have $repr': f-repr (Suc i) (\text{inc-i Fr1}) F1$ by auto

    from of-list-repr[OF repr'] have $Hr': \text{of-list-repr (inc-i Fr1)} = F1$ by auto

from False have $le: \text{sq-norm} (G ! (i - 1)) \leq \alpha \ast \text{sq-norm} (G ! i)$ by force
unfolding gs.weakly-reduced-def
proof (intro allI impI)
  fix k
  assume ki: Suc k < Suc i
  show sq-norm (G ! k) ≤ α * sq-norm (G ! Suc k)
  proof (cases Suc k < i)
    case True
    from red[unfolded gs.weakly-reduced-def, rule-format, OF True] show ?thesis
  next
    case False
    with i0 ki have id: k = i - 1 Suc k = i by auto
    with le show ?thesis by auto
  qed
  qed from res False ii
  have state: state = increase-i (Suc i, Fr1, Gr1) by auto
  have invS: LLL-invariant state F1 G unfolding state
    by (rule increase-i[OF Line' i], insert False, auto)
  obtain Fr' Gr' where state: state = (Suc i, Fr', Gr') using state
    by (cases state, auto simp: increase-i-def)
  have LLL-measure state < LLL-measure (Suc i, Fr, Gr) unfolding LLL-measure-def
    logD-eq split state LLL-invD[1][OF invS[unfolded state], symmetric] inv(1)[symmetric]
    using i by simp
  thus ?thesis using invS by blast
next
  case True
  from i0 inv' True i swap: set F1 ⊆ carrier-vec n i < length F1 i - 1 <
  unfolding LLL-invariant-def Let-def by auto
  define F2 where F2 = F1[i := F1 ! (i - 1), i - 1 := F1 ! i]
  define Fr2 where Fr2 = dec-i (update-im1 (update-i Fr1 (get-nth-im1 Fr1))
    (get-nth-i Fr1))
  from dec-i[OF update-im1[OF update-i[OF repr]], of get-nth-im1 Fr1 get-nth-i
    Fr1, folded Fr2-def] swap(2) i0
  have list-repr (i - 1) Fr2 (F1[i := get-nth-im1 Fr1, i - 1 := get-nth-i Fr1])
    by auto
  hence repr' : f-repr (i - 1) Fr2 F2 unfolding F2-def
    using get-nth-im1[OF repr] get-nth-i[OF repr] i0 swap(2) by (auto simp: map-update)
  note Hr' = of-list-repr[OF repr']
  obtain G2 where gH': snd (gram-schmidt-int n F2) = G2 by force
  let ?gso' = let gi = g-i Gr1;
    gim1 = g-im1 Gr1;
    fim1 = get-nth-im1 Fr1;
    new-gim1 = gi + mu ·v gim1;
    norm-gim1 = sq-norm new-gim1;
    new-gi = gim1 - (fim1 ·i new-gim1 / norm-gim1) ·v new-gim1;
norm-gi = sq-norm new-gi
in dec-i (update-im1 (update-i Gr1 (new-gi,norm-gi)) (new-gim1,norm-gim1))

define Gr2 where gso': Gr2 = ?gso'
have span': gs.span (SRAT F1) = gs.span (SRAT F2)
by (rule arg-cong[of .. gs.span], unfold F2-def, insert swap, auto)
from res gH' Fr2-def Hr' gso' True ii
have state: state = (i - 1, Fr2, Gr2) by (auto simp: Let-def)
have lF2: lattice-of F2 = lattice-of F1 unfolding F2-def
by (rule lattice-of-swap[OF swap refl])
have len': length F2 = m using inv' unfolding F2-def by auto
have F2: set F2 ⊆ carrier-vec n using swap unfolding F2-def set-cone-nth
by (auto, rename-tac k, case-tac k = i, force, case-tac k = i - 1, auto)
let ?rv = map-vec rat-of-int
from inv'(10) have indepH: gs.lin-indpt-list (RAT F1) .
from i l len have i < length (RAT F1) i - 1 < length (RAT F1) by auto
with distinct-swap[OF this] len have distinct (RAT F2) = distinct (RAT F1)
unfolding F2-def
by (auto simp: map-update)
with len' F2 span' indepH have indepH': gs.lin-indpt-list (RAT F2) unfolding F2-def using i l
by (auto simp: gs.lin-indpt-list-def)
note conn1 = indepH inv'(2) len
note conn2 = indepH' gH' len'
from gram-schmidt-int-connect[OF conn1]
have Gs-fs: G = map (gs.gso (RAT F1)) [0..<m] .
from gram-schmidt-int-connect[OF conn2]
have G2-F2: G2 = map (gs.gso (RAT F2)) [0..<m] .
have F2-F1: k < i - 1 \Longrightarrow F2 ! k = F1 ! k for k unfolding F2-def by auto
{
fix k
assume ki: k < i - 1
with i have kn: k < m by simp
have G2 ! k = gs.gso (RAT F2) k unfolding G2-F2 using kn by auto
also have ... = gs.gso (RAT F1) k
by (rule gs-gs-identical, insert ki kn len, auto simp: F2-def)
also have ... = G ! k unfolding Gs-fs using kn by auto
finally have G2 ! k = G ! k .
} note G2-G = this
have take-eq: take (Suc i - 1 - 1) F2 = take (Suc i - 1 - 1) F1
by (intro nth-equalityI, insert len len' i swap(2-), auto intro!: F2-F1)
from inv' have gs.weakly-reduced α i G by auto
hence gs.weakly-reduced α (i - 1) G unfolding gs.weakly-reduced-def by auto
hence red: gs.weakly-reduced α (i - 1) G2
  unfolding gs.weakly-reduced-def using G2-G by auto
from inv' have L: lattice-of F2 = L unfolding lF2 by auto
have ii: g-repr (i - 1) = g-repr ((Suc i - 1) - 1) using True by auto
have i1n: i - 1 < m using i by auto
let ?R = rat-of-int

141
let ?RV = map-vec ?R
let ?f1 = λ i. RAT F1 \ i
let ?f2 = λ i. RAT F2 \ i
have hgt:F1 ! (i - 1) = F2 ! i take (i-1) F1 = take (i-1) F2
  \ ?f2 (i - 1) = ?f1 ?f2 i = ?f1 (i - 1)

unfolding F2-def using i len \i0 by auto
let ?g2 = gs.gso (RAT F2)
let ?g1 = gs.gso (RAT F1)
let ?mu1 = gs.µ (RAT F1)
let ?mu2 = gs.µ (RAT F2)
from \gH [unfolded gram-schmidt-int-def gram-schmidt-wit-def] indepH' len'
have connH':
gs.lin-indpt-list (RAT F2) length (RAT F2) = m snd (gs.main (RAT F2))

G2
by (auto intro: nth-equalityI)
from \gS [unfolded gram-schmidt-int-def gram-schmidt-wit-def] indepH len
have connH:
gs.lin-indpt-list (RAT F1) length (RAT F1) = m snd (gs.main (RAT F1))

G
by (auto intro: nth-equalityI)
have gs: \ j. j < m \Rightarrow ?g1 j ∈ Rn using gs.gso-carrier[OF connH] .
have \ j. j < m \Rightarrow ?f1 j ∈ Rn using gs.f-carrier[OF connH] .
let \fs1 = \?f1 \{0..< (i - 1)}
have G: \fs1 \subseteq Rn using g i by auto
let \gs1 = ?g1 \{0..< (i - 1)}
have G': ?gs1 \subseteq Rn using gs i by auto
let \S = gs.span \fs1
let \S' = gs.span ?gs1
have \S' = ?S
by (rule gs.partial-span[OF connH], insert i, auto)
have gs.ls.is-projection (?g2 (i - 1)) (gs.span (?g2 \{0..< (i - 1)})) (?f2 (i - 1))
by (rule gs.gso-projection-span(2)[OF connH' i - 1 < m])
also have \f2 (i - 1) = \?f1 i unfolding F2-def using len i by auto
also have gs.span (?g2 \{0..< (i - 1)} ) = gs.span (\f2 \{0..< (i - 1)} )
by (rule gs.partial-span[OF connH'], insert i, auto)
also have \f2 \{0..< (i - 1)} = \fs1
by (rule image-cong[OF refl], insert len i, auto simp: F2-def)
finally have claim1: gs.ls.is-projection (?g2 (i - 1)) \S (\f1 i) .
have \f1 i = gs.sumlist (map (λj. ?mu1 i j .v. ?g1 j) [0..< i] @ [?g1 i])
unfolding gs.µ ls.is-sum-of-mu-gso[OF connH i < m] by (simp add: gs.µ.simps)
also have ... = gs.sumlist (map (λj. ?mu1 i j .v. ?g1 j) [0..< i]) + ?g1 i
by (subst gs.sumlist-append, insert i gs, auto)
finally have claim2: \f1 i = gs.sumlist (map (λj. ?mu1 i j .v. ?g1 j) [0..< i])
+ ?g1 i (is = ?sum + ...).
have sum: \sum ∈ Rn by (rule gs.sumlist-carrier, insert gs i, auto)
from gs.span-closed[OF G] have \S \subseteq Rn by auto
from gs i have gs i \ j. j < i - 1 \Rightarrow ?g1 j ∈ Rn and gsi: ?g1 (i - 1) ∈ Rn by auto

142
have \([0 ..< i] = [0 ..< \text{Suc} \ (i - 1)]\) using \(i0\) by simp
also have \(\ldots = [0 ..< i - 1] @ [i - 1]\) by simp
finally have list: \([0 ..< i] = [0 ..< i - 1] @ [i - 1]\).

have \(g2-im1: \{g2 \ (i - 1) = g1 \ i + \text{?mu-f1} \ (i - 1), v \ g1 \ (i - 1) \ (\text{is -} = - + \ ?\mu-f1)\}

proof (rule gs.is-projection-eq[OF connH \(\text{claim}1\) - \(S\) \(g\)[OF \(i\)])

show \(\text{gs.is-projection} (\{\text{?g1} \ i + \ ?\mu-f1\} : \{?\}\) unfolding gs.is-projection-def

proof (intro conjI allI impI)

let \(\text{sum'} = \text{gs.sumlist} (\text{map} (\lambda j. \ ?\mu1 \ i \ j \ v \ g1 \ j) [0 ..< i - 1])\)

have \(\?\text{sum'} \in \text{Rn by (rule gs.sumlist-carrier}, \text{insert gs i, auto})\)

show \(\text{inRn}: (?\text{g1} \ i + ?\text{mu-f1}) \in \text{Rn using gs}[OF \ i] \text{gsi i by auto}\)

have \(\text{carr}: \?\text{sum} \in \text{Rn} \ ?\text{g1} \ i \in \text{Rn} \ ?\text{mu-f1} \in \text{Rn} \ ?\text{sum'} \in \text{Rn using sum'}\)

sum gs[OF \ i] gsi i by auto

have \(\?\text{f1} i - (\?\text{g1} \ i + ?\mu-f1) = (\?\text{sum} + ?\text{g1} \ i) - (\?\text{g1} \ i + ?\mu-f1)\)

unfolding claim2 by simp

also have \(\ldots = \?\text{sum} - ?\mu-f1\) unfolding carr by auto

also have \(\?\text{sum} = \text{gs.sumlist} (\text{map} (\lambda j. \ ?\mu1 \ i \ j \ v \ g1 \ j) [0 ..< i - 1]) \at [\ ?\mu-f1]\)

unfolding list by simp

also have \(\ldots = \?\text{sum'} + \ ?\mu-f1\)

by (subst \(\text{gs.sumlist-append, insert gs'}\) gsi, auto)

also have \(\ldots = ?\mu-f1 = ?\text{sum'} \text{gzi by auto}\)

finally have id: \(\?\text{f1} i - (?\text{g1} \ i + \ ?\mu-f1) = ?\text{sum'}\).

show \(\?\text{f1} i - (?\text{g1} \ i + ?\mu-f1) \in \text{gs.span} \ ?\text{S}\) unfolding id \(\text{gs.span-span}[OF \ G]\)

proof (rule gs.sumlist-in-span[OF \ G])

fix \(v\)

assume \(v \in \text{set} (\text{map} (\lambda j. \ ?\mu1 \ i \ j \ v \ g1 \ j) [0 ..< i - 1])\)

then obtain \(j\) where \(j: j < i - 1\) and \(v: v = ?\mu1 \ i \ j \ v \ g1 \ j\) by auto

show \(v \in ?\text{S}\) unfolding \(v\)

by (rule gs.smult-in-span[OF \ G], unfold \(\text{S} \text{S'})[\text{symmetric}]\), rule gs.span-mem, insert gs i j, auto)

qed

fix \(x\)

assume \(x \in ?\text{S}\)

hence \(x: x \in ?\text{S'}\) using \(\text{S'}\) by simp

show \((?\text{g1} \ i + ?\mu-f1) \cdot x = 0\)

proof (rule gs.orthocompl-span[OF connH - \(G'\) inRn \(x\)])

fix \(x\)

assume \(x \in ?\text{gsi}\)

then obtain \(j\) where \(j: j < i - 1\) and \(x-id: x = ?g1 \ j\) by auto

from \(j \ i \ x\)-id \(gs[j]\) have \(x: x \in \text{Rn by auto}\)

\{ fix \(k\)

assume \(k: k > j \ k < m\)

have \(?g1 \ k \cdot x = 0\) unfolding \(x\)-id

by (rule gs.orthogonal[OF connH], insert k, auto)
\}

from \(\text{this}[of i] \: \text{this}[of i - 1] \ j \ i\)
have main: \( \mathcal{g}_1 i \cdot x = 0 \) \( \mathcal{g}_1 (i - 1) \cdot x = 0 \) by auto

have \( (\mathcal{g}_1 i + ?\mu-f1) \cdot x = \mathcal{g}_1 i \cdot x + ?\mu-f1 \cdot x \)
by (rule add-scalar-prod-distrib[OF gs[OF i] - x], insert gsi, auto)
also have \( \ldots = 0 \) using main
by (subst smallt-scalar-prod-distrib[OF gsi x], auto)

finally show \( (?\mathcal{g}_1 i + ?\mu-f1) \cdot x = 0 \).
qed
qed
qed

{ 
fix \( k \)
assume \( kn: k < m \)
and \( ki: k \neq i \) \( k \neq i - 1 \)

have \( ?\mathcal{g}_2 k = gs\_projection (gs\_span (\mathcal{g}_2 \cdot \{0..<k\})) (\mathcal{f}_2 k) \)
by (rule gs\_gs-projection-span[OF connH' kn])
also have \( gs\_span (\mathcal{g}_2 \cdot \{0..<k\}) = gs\_span (\mathcal{f}_2 \cdot \{0..<k\}) \)
by (rule gs\_partial-span[OF connH k], insert kn, auto)
also have \( ?\mathcal{f}_2 \cdot \{0..<k\} = ?\mathcal{f}_1 \cdot \{0..<k\} \)
proof (cases \( k \leq i \))
case True hence \( k < i - 1 \) using \( ki \) by auto

then show \( \?\text{thesis apply(intro image-cong) unfolding F2-def using len} \)
i by auto
next
case False

have \( ?\mathcal{f}_2 \cdot \{0..<k\} = Fun\_swap i (i - 1) ?\mathcal{f}_1 \cdot \{0..<k\} \)

unfolding Fun\_swap-def F2-def a-def using len i
by (intro image-cong, insert len kn, force+)
also have \( \ldots = ?\mathcal{f}_1 \cdot \{0..<k\} \)
apply (rule swap\_image-eq) using False by auto
finally show \( \?\text{thesis} \).
qed
also have \( gs\_span \ldots = gs\_span (\mathcal{g}_1 \cdot \{0..<k\}) \)
by (rule sym, rule gs\_partial-span[OF connH k], insert kn, auto)
also have \( ?\mathcal{f}_2 k = ?\mathcal{f}_1 k \) using \( ki \) \( kn \) len unfolding F2-def by auto
also have \( gs\_projection (gs\_span (\mathcal{g}_1 \cdot \{0..<k\})) \ldots = \mathcal{g}_1 k \)
by (subst gs\_gs-projection-span[OF connH kn], auto)
finally have \( ?\mathcal{g}_2 k = ?\mathcal{g}_1 k \).
}

note g2-g1-identical = this

{ 
fix \( jj \)
assume \( jj: jj < i - 1 \)

hence \( id1: jj < i - 1 \iff True \) \( jj < i \iff True \) by auto

have \( id2: ?\mathcal{g}_2 jj = ?\mathcal{g}_1 jj \) by (subst g2-g1-identical, insert jj i, auto)

have \( ?\mu2 i jj = ?\mu1 (i - 1) \) \( jj \)
unfolding gs\_\( \mu \)\_dists id1 id2 if-True using len i i0 by (auto simp: F2-def)
}

note mu'\_mu-i = this

let \( ?\mathcal{g}_2\_\text{im1} = ?\mathcal{g}_2 (i - 1) \)

have \( g2\_\text{im1-Rn:} \ ?g2\_\text{im1} \in Rn \) using i by (auto intro!: gs\_gs-carrier[OF connH k])

144
\{ 
let \( ?\mu_2-f2 = \lambda i. - ?\mu_2 i \cdot g2 \cdot j \)
let \( ?\sum = gs\text{s}um\text{list (map (}\lambda j. - ?\mu_1 (i - 1) j \cdot v \cdot g1 j) [0 \ldots < i - 1]) \)
have mhs: \( ?\mu_2-f2 (i - 1) \in \text{Rn using i by (auto intro!: gs.gso-carrier[OF connH])} \)
  have \( ?\sum \in \text{Rn by (rule gs.suml\text{ist-carrier, insert gs i, auto)} \)
  have \( \text{gim1: } ?f1 (i - 1) \in \text{Rn using g i by auto} \)
  have \( ?g2 i = ?f2 i + gs\text{s}um\text{list (map } ?\mu_2-f2 [0 \ldots < i - 1] \text{ @ } [?\mu_2-f2 (i - 1)]) \)
    unfolding gs.gso.simps[of - i] list by simp
also have \( ?f2 i = ?f1 (i - 1) \text{ unfolding } F2\_de\text{f using len i 0 by auto} \)
also have \( \text{map } ?\mu_2-f2 [0 \ldots < i - 1] = \text{map (}\lambda j. - ?\mu_1 (i - 1) j \cdot v \cdot g1 j) [0 \ldots i - 1] \)
  by (rule map-cong[OF refl], subst g2-g1-identical, insert i, auto simp: mu\_\text{\prime}-mu-i)
also have \( gs\text{s}um\text{list (... @ } [?\mu_2-f2 (i - 1)]) = ?\sum + ?\mu_2-f2 (i - 1) \)
  by (subst gs.suml\text{ist-append, insert gs i mhs, auto)}
also have \( ?f1 (i - 1) + \ldots = (??f1 (i - 1) + ?\sum + ?\mu_2-f2 (i - 1) \)
  using gim1 \( \text{sum } \text{mhs by auto} \)
also have \( ?f1 (i - 1) + ?\sum = ?g1 (i - 1) \text{ unfolding gs.gso.simps[of - i} \)
  by simp
also have \( ?\mu_2-f2 (i - 1) = - (?f2 i \cdot g2-im1 / sq-norm \cdot g2-im1) \cdot v \cdot g2-im1 \text{ unfolding gs.gso.simps using i0 by simp} \)
also have \( \ldots \in - ((f2 i \cdot \text{g2-im1 / sq-norm } \cdot g2-im1) \cdot v \cdot g2-im1) \text{ by auto} \)
also have \( ?g1 (i - 1) + \ldots = ?g1 (i - 1) - ((?f2 i \cdot \text{g2-im1 / sq-norm g2-im1}) \cdot v \cdot g2-im1) \)
  by (rule sym, rule minus-add-uminus-vec[of - n], insert gsi g2-im1-Rn, auto)
also have \( ?f2 i = ?f1 (i - 1) \text{ by fact} \)
  finally have \( ?g2 i = ?g1 (i - 1) - (?f1 (i - 1) \cdot g2 (i - 1) / sq-norm (??g2 (i - 1)]) \cdot v \cdot g2 (i - 1) . \)
  } 
note g2-i = this
\}

from i\in have i\text{\prime}n': \( i - 1 \leq m \text{ by simp} \)
  have upd-im1: list-repr i ba xs \( \Rightarrow \) ys = (xs \([i - 1 := x]\)) \( \Rightarrow \) list-repr i (update-im1 ba x) ys
    for ba xs x ys using update-im1[of i ba xs] i0 by force
from gso[unfolded g-repr-def]
  have gsoH: list-repr i Gr1 (map (\( \lambda x. (x, \|x\|^2) \)) (map (GSO F1) [0..<m]))
by auto

let \( ?g2\_\text{\prime}-im1 = \text{get-nth-im1 } F\text{r1} \)
let \( ?g2\_\text{\prime}-im1 = g-i Gr1 + \text{mu} \cdot v \cdot g-i Gr1 \)
let \( \text{fnorm-im1 = sq-norm } \cdot g2\_\text{\prime}-im1 \)
let \( ?g2\_\text{\prime}-i = g\_im1 Gr1 - (\text{?f1-im1 } \cdot i \cdot g2\_\text{\prime}-im1 / \text{sq-norm-im1}) \cdot v \cdot g2\_\text{\prime}-im1 \)
define \( g2\_\text{\prime}-i \text{ where } g2\_\text{\prime}-i = ?g2\_\text{\prime}-i \)
define \( g2\_\text{\prime}-im1 \text{ where } g2\_\text{\prime}-im1 = g2\_\text{\prime}-im1 \)

have \( ?g2 (i - 1) = ?g1 i + (?mu1 i (i - 1) \cdot v \cdot g1 (i - 1) \text{ by fact} \)
  also have \( ?g1 i = g-i Gr1 \text{ unfolding } g-i[OF Linv' i] \text{ Gs-fs o-def using i} \)
by simp
also have \( ?g1 (i - 1) = g-im1 Gr1 \text{ unfolding } g-im1[OF Linv' i i0] \text{ Gs-fs} \)
\( o-def \) using \( i \) by simp

finally have \( g2im1: \, ?g2 \, (i - 1) = g2' - im1 \)

unfolding \( m \) \( g2' - im1 - def \) by blast

have \( ?f - im1 \in carrier-vec \, n \) using \( inv'(3-4) \, (i - 1 < m) \) unfolding 
get-nth-im1[\( OF \, inv'(8) \, i0 \)]

by auto

hence \( dim: \, dim-vec \, ?f - im1 \, = \, n \) unfolding \( g2' - im1 - def \)[symmetric]
\( g2im1 \)[symmetric]

using \( \langle \, g2 - im1 \in \, Rn \, \rangle \) by auto

have \( ?g2 \, i = \, ?g1 \, (i - 1) - \, (?f1 \, (i - 1) \cdot \, ?g2 \, (i - 1) / \, sq-norm \, (\, ?g2 \, (i - 1) \rangle \) \)

by (rule \( g2 - i \))

also have \( ?g2 \, (i - 1) = g2' - im1 \) by (simp add: \( g2im1 \)[symmetric])

also have \( ?g1 \, (i - 1) = g - im1 \, G1 \) by fact

also have \( ?f1 \, (i - 1) = map - vec \, of - int \, ?f - im1 \)

unfolding \( get-nth-im1[\, OF \, repr \, i0 \] \) \( o-def \) using \( \, len \, i \) by simp

finally have \( g2i: \, ?g2 \, i = \, g2' - i \) using \( dim \) unfolding \( g2' - i - def \) \( g2' - im1 - def \)

by simp

have \( map \, (\, \lambda x. \, (\, \parallel x \parallel ^2)\, ) \) \( (\, map \, (\, GSO \, F1 \, ) \, [0..<m]\)\)

\[ i = \, (g2' - i, \, sq-norm \, g2' - i), \, i - 1 = \, (g2' - im1, \, sq-norm \, g2' - im1) \]

unfolding \( map - update \) by auto

also have \( GSOH: \, map \, (\, GSO \, F1 \, ) \, [0..<m] = \, map \, ?g1 \, [0..<m] \)

by (rule \( map - cong[\, OF \, refl]\), \, auto \, simp: \(\, GSO - def \, len \, intro!: \, g2 - g2 - identical\) )

also have \( id: \, \ldots \, [\, i := \, g2' - i, \, i - 1 := \, g2' - im1 \] = \, map \, ?g2 \, [0..<m] \) (is \( ?G \)

\( = \, ?G2 \) )

proof -

\{ 

fix \( k \)

assume \( k: \, k < \, m \)

consider \( (\, ki\, ) \, k = \, i \, \mid \, (\, im1\, ) \, k = \, i - 1 \, \mid \, (\, other\, ) \, k \notin \, \{i - 1, \, i\} \) by auto

hence \( ?G\, k \, k = \, ?g2 \, k \)

proof (cases)

case \( other \)

hence \( ?G, \, \, k \, k = \, ?g1 \, k \) using \( k \) by simp

also have \( \ldots \, = \, ?g2 \, k \, k \) using \( g2 - g1 - identical[\, OF \, k\, ] \, other \, by \, auto \)

finally show \( ?thesis \).

next

case \( ki \)

have \( ?g2 \, i = \, g2' - i \) unfolding \( g2i \, \ldots \)

also have \( \ldots \, = \, ?G, \, k \, k \) using \( \, i \, len \, i0 \, ki \) by simp

finally show \( ?thesis \) unfolding \( ki \) by simp

next

case \( im1 \)

hence \( ?G, \, k \, k = \, g2' - im1 \) using \( i \, len \) by simp

also have \( \ldots \, = \, ?g2 \, (i - 1) \) unfolding \( g2im1 \, \ldots \)

finally show \( ?thesis \) unfolding \( im1 \) by simp

qed
also have \( g_2^k = ?G_2^k ! k \) using \( k \) by simp 

finally have \( ?G_2 ! k = ?G_2 ! k \) by simp 

} note main = this 

show \( \text{thesis} \)

by (rule nth-equalityI, force, insert main, auto)

qed

also have \( \ldots = \text{map} \ GSO \ F_2 [\ldots m] \)

by (rule map-cong[OF refl], auto simp: GSO-def len' intro!: gs-gs-identical)

finally

have \( \text{map} \ (\lambda x. (x, \| x \|^2)) \ (\text{map} \ GSO \ F_2 [\ldots m]) \)

by (rule add-scalar-prod-distrib, insert gs i, auto)

also have \( ?g_1 + ?\mu-f1 \cdot ?g_1 \)

by (rule add-scalar-prod-distrib, insert gs i, auto)

also have \( \langle i, i \rangle = 0 \) using orthogonalD[OF gs.gram-schmidt(2)[OF connH], of i i-1] i len i0

unfolding Gs-fs 

by (auto simp: o-def)

also have \( ?\mu-f1 \cdot ?\mu-f1 = ?m_1 \cdot ?m_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot \ldots \)

unfolding Gs-fs 

by (auto simp: o-def)

also have \( ?m_1 \cdot ?m_1 = ?m_1 \cdot ?m_1 \cdot ?m_1 \cdot ?m_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot \ldots \)

by (rule scalar-prod-smult-right, insert gs[OF i] gs[OF i i-1 < m], auto)

also have \( ?g_1 \cdot ?g_1 = 0 \)

unfolding Gs-fs 

by (auto simp: o-def)

also have \( ?\mu-f1 \cdot ?\mu-f1 = ?m_1 \cdot ?m_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot ?g_1 \cdot \ldots \)

by (rule scalar-prod-smult-right, insert gs[OF i] gs[OF i i-1 < m], auto)
by (rule scalar-prod-smult-left, insert gs[OF i] gs[OF \( i - 1 < m \)], auto)
also have \( ?g1 (i - 1) \cdot ?g1 (i - 1) = \text{sq-norm} \ (\text{sq-norm} \ ?g1 (i - 1)) \)
by (simp add: sq-norm-vec-as-cscalar-prod)
finally have \( \text{sq-norm} (\ ?g2 (i - 1)) = \text{sq-norm} (\ ?g1 i) + (\ ?mu1 i (i - 1) \cdot \ ?mu1 i (i - 1)) \cdot \text{sq-norm} (\ ?g1 (i - 1)) \)
by (simp add: ac-simps o-def
also have \( \ldots < \frac{1}{\alpha} \cdot (\text{sq-norm} \ (\ ?g1 (i - 1))) + (\frac{1}{2} \cdot \frac{1}{2}) \cdot (\text{sq-norm} \ (\ ?g1 (i - 1))) \)
proof (rule add-less-le-mono[OF - mult-mono])
from True[unfolded mult.commute[of \( \alpha \) Gs-fs],
THEN linordered-field-class.mult-imp-less-div-pos[OF \( \alpha0(1) \)]]
show \( \text{sq-norm} (\ ?g1 i) < \frac{1}{\alpha} \cdot (\text{sq-norm} \ ?g1 (i - 1)) \)
unfolding Gs-fs o-def using len i by auto
from m12 have \( \text{abs} (\ ?mu1 i (i - 1)) \leq \frac{1}{2} \) unfolding mu by auto
have \( ?mu1 i (i - 1) \cdot \ ?mu1 i (i - 1) \leq \text{abs} (\ ?mu1 i (i - 1)) \cdot \text{abs} (\ ?mu1 i (i - 1)) \) by auto
also have \( \ldots \leq \frac{1}{2} \cdot \frac{1}{2} \) using mult-mono[OF \( \text{abs} \ ?mu1 \)] by auto
finally show \( ?mu1 i (i - 1) \cdot ?mu1 i (i - 1) \leq \frac{1}{2} \) unfolding by auto
qed auto
also have \( \ldots = \text{reduction} \cdot \text{sq-norm} (\ ?g1 (i - 1)) \) unfolding reduction-def

using \( \alpha0 \) by (simp add: ring-distrib add-divide-distrib)
finally have \( \text{sq-norm} (\ ?g2 (i - 1)) < \text{reduction} \cdot \text{sq-norm} (\ ?g1 (i - 1)) \).
} note g-reduction = this
have \( \text{norm-pos} \ j < m \implies \text{sq-norm} (\ ?g2 j) > 0 \) for \( j \)
using gs sq-norm-pos[OF \( \text{connH} \ of \ ?j \)] unfolding G2-F2 o-def by simp

\{ fix \( k \)
assume \( k: k = i \)
hence \( kn: k \leq m \) using \( i \) by auto
from \( Gd[OF \text{newInV}, \text{folded dk-def, folded state, OF kn}] \)
have \( \text{R} (\ ?k \ ?k F2) = \{ \?j < i. \text{sq-norm} (\text{G2} \ ?j) \} \) by auto
also have \( \ldots = \text{prod} (\ ?j. \text{sq-norm} (\ ?g2 ?j)) (\{ \?0 ..< \?i-1 \} \cup \{ \?i-1 \}) \)
by (rule sym, rule prod.cong, \( \text{insert} \ ?i0, \text{auto} \?i1 \), insert \( G2-F2 i, \text{auto} \) simp: o-def)
also have \( \ldots = \text{sq-norm} (\ ?g2 (i - 1)) \cdot \text{prod} (\ ?j. \text{sq-norm} (\ ?g2 ?j)) (\{ \?0 ..< \?i-1 \}) \)
by simp
also have \( \ldots \leq (\text{reduction} \cdot \text{sq-norm} (\ ?g1 (i - 1))) \cdot \text{prod} (\ ?j. \text{sq-norm} (\ ?g2 ?j)) (\{ \?0 ..< \?i-1 \}) \)
by (rule mult-strict-right-mono[OF \( \text{g-reduction prod-pos} \), insert \( \text{norm-pos} i, \text{auto} \))
also have \( \text{prod} (\ ?j. \text{sq-norm} (\ ?g2 ?j)) (\{ \?0 ..< \?i-1 \}) = \text{prod} (\ ?j. \text{sq-norm} (\ ?g1 ?j)) (\{ \?0 ..< \?i-1 \}) \)
by (rule \( \text{prod.cong[OF refl]} \), subst \( \text{g2-g1-identical} \), insert \( i, \text{auto} \))
also have \( (\text{reduction} \cdot \text{sq-norm} (\ ?g1 (i - 1))) \cdot \text{prod} (\ ?j. \text{sq-norm} (\ ?g1 ?j)) (\{ \?0 ..< \?i-1 \} \cup \{ \?i-1 \}) \) by simp

148
also have \( \prod \text{sq-norm} (g_1 j) \) \( \{0 ..< i - 1\} \cup \{i - 1\} = (\prod j < i. \text{sq-norm} (g_1 j)) \)

by (rule \( \text{prod.cong} \), insert \( i_0 \), auto)

also have \( \ldots = \R (dk k F1) \) unfolding \( \text{dk-def Gd[OF Linv' kn]} \) unfolding \( k \)

by (rule \( \text{prod.cong[OF refl]} \), insert \( i \), auto simp: \( \text{Gs-fs o-def} \))

also have \( \ldots = \R (dk k F) \) unfolding \( \text{dk-eq[OF kn]} \) by simp

finally have \( dk k F2 < \text{real-of-rat reduction} \ast dk k F \)

using \( \text{real-of-int} \) by simp

also have \( \R (dk k F2) \) unfolding \( \text{prod-pos} \)

by (rule \( \text{prod-pos} \))

fix \( k \)

assume \( kn: k \leq m \) and \( ki: k \neq i \)

from \( \text{dk-swap-unchanged[OF len i0 i ki kn F2-def]} \) \( \text{dk-eq[OF kn]} \)

have \( dk k F = dk k F2 \) by simp

} note \( dk-i = \text{this[OF refl]} \)

\}

have \( \text{prod-pos}:0 < \{ (\prod i < m. \text{dk i F2}) \) apply (rule \( \text{prod-pos} \))

using \( \text{LLL-dk-pos[OF newInv, folded state, of k]} \) by auto

have \( \text{prodpos2}:0 < (\prod i < m. \text{dk i F2}) \) apply (rule \( \text{prod-pos} \))

using \( \text{LLL-dk-pos[OF newInv, folded state] \text{pos by auto}} \)

have \( \text{prod-nonneg}:0 \leq (\prod x \in \{0 ..< m\} - \{i\}. \text{real-of-int} (dk x F2)) \) apply (rule \( \text{prod-nonneg} \))

using \( \text{LLL-dk-pos[OF newInv, folded state] \text{pos by auto}} \)

have \( \text{prodpos2}:0 < (\prod i < m. \text{dk i F2}) \) apply (rule \( \text{prod-pos} \))

using \( \text{LLL-dk-pos[OF \text{assms}(1)] \text{by auto}} \)

have \( D F2 = \text{real-of-int} (\prod i < m. \text{dk i F2}) \) unfolding \( \text{D-def using prodpos} \)

by simp

also have \( (\prod i < m. \text{dk i F2}) = (\prod j \in \{0 ..< m\} - \{i\} \cup \{i\}. \text{dk j F2}) \)

by (rule \( \text{prod.cong, insert i, auto} \))

also have \( \text{real-of-int} \ldots = \text{real-of-int} (\prod j \in \{0 ..< m\} - \{i\}. \text{dk j F2}) \ast \text{real-of-int} (\text{dk j F2}) \)

by (subst \( \text{prod.union-disjoint, auto} \))

also have \( \ldots < (\prod j \in \{0 ..< m\} - \{i\}. \text{dk j F2}) \ast (\text{of-rat reduction} \ast \text{dk i F}) \)

by (rule \( \text{mul-strict-left-mono[OF dk-i,insert prod-pos',auto]} \))

also have \( (\prod j \in \{0 ..< m\} - \{i\}. \text{dk j F2}) = (\prod j \in \{0 ..< m\} - \{i\}. \text{dk j F}) \)

by (rule \( \text{prod.cong, insert dk, auto} \))

also have \( \ldots \ast (\text{of-rat reduction} \ast \text{dk i F}) \)

by (subt \( \text{prod.union-disjoint, auto} \))

also have \( (\prod j \in \{0 ..< m\} - \{i\} \cup \{i\}. \text{dk j F}) = (\prod j < m. \text{dk j F}) \)

by (subst \( \text{prod.cong, insert i, auto} \))

finally have \( D: D F2 < \text{real-of-rat reduction} \ast D F \)

unfolding \( \text{D-def using prodpos2 by auto} \)

have \( \text{logD}: \text{logD F2} < \text{logD F} \)
proof (cases $\alpha = 4/3$)

case True

show ?thesis using D unfolding reduction(4)[OF True] logD-def unfolding

True by simp

next

case False

hence False': $\alpha = 4/3 \iff False$ by simp

from False α have $\alpha > 4/3$ by simp

with reduction have reduction1: reduction $< 1$ by simp

let ?new = real (D F2)

let ?old = real (D F)

let ?log = log (1/of-rat reduction)

note pos = LLL-D-pos[OF newInv[folded state] LLL-D-pos[OF assms(1)]

from reduction have real-of-rat reduction $> 0$ by auto

hence gediv: $1$/real-of-rat reduction $>$ 0 by auto

have $(1/of-rat reduction) \ast ?new \leq ((1/of-rat reduction) \ast of-rat reduction)$

unfolding mult.assoc real-mul-le-cancel-iff2[of gediv]

using D by simp

also have $(1/of-rat reduction) \ast of-rat reduction = 1$ using reduction by auto

finally have $(1/of-rat reduction) \ast ?new \leq ?old$ by auto

hence $?log ((1/of-rat reduction) \ast ?new) \leq ?log ?old$

by (subth log-le-cancel-iff, auto simp: pos reduction1 reduction)

hence floor $?log ((1/of-rat reduction) \ast ?new) \leq$ floor $?log ?old$

by (rule floor-mono)

hence nat (floor $?log ((1/of-rat reduction) \ast ?new)) \leq$ nat (floor $?log ?old)$

by simp

also have .. = logD F unfolding logD-def False' by simp

also have $?log ((1/of-rat reduction) \ast ?new) = 1 + ?log ?new$

by (subth log-mul, insert reduction1 reduction, auto simp: pos )

also have floor $(1 + ?log ?new) = 1 +$ floor $?log ?new)$ by simp

also have nat $(1 +$ floor $?log ?new)) = 1 +$ nat (floor $?log ?new))$

by (subth nat-add-distrib, insert pos reduction1 , auto)

also have nat $(floor ?log ?new)) = logD F2 unfolding logD-def False' by simp

finally show logD F2 $<$ logD F by simp

qed

hence LLL-measure state $<$ LLL-measure $(i, Fr, Gr)$ unfolding LLL-measure-def

state split

inv(1)[symmetric] of-list-repr[OF repr] using i logD by simp

thus ?thesis using newInv unfolding state by auto

qed

next

case i0: True

from res i0 have state: state = increase-i $(i, Fr, Gr)$ unfolding basis-reduction-step-def

by auto

with increase-i[OF inv i] i0

have inv': LLL-invariant state F G by auto
from LLL-invD[OF inv] have Gr: of-list-repr Fr = F by simp
from LLL-invD[OF inv"[unfolded increase-i-def state split]]
have Gr": of-list-repr (inc-i Fr) = F by simp
have id: of-list-repr (inc-i Fr) = of-list-repr Fr by (simp add: Gr Gr")
have dec: LLL-measure state < LLL-measure (i, Fr, Gr) using i unfolding state id
unfolding LLL-measure-def by (simp add: increase-i-def id)
show (\exists H. Ex (LLL-invariant state H)) \land LLL-measure state < LLL-measure (i, Fr, Gr)
  by (intro conjI exI dec, rule inv")
qed

lemma D-approx: assumes LLL-invariant (i, Fr, Gr) F G
shows D F ≤ nat (\prod i less m. (\prod j less i. \|F ! j\|^2))
proof –
  note inv = LLL-invD[OF assms]
  note conn = LLL-connect[OF assms]
  note main = inv(2)[unfolded gram-schmidt-int-def gram-schmidt-wit-def]
  have rat-of-int (\prod i less m. dk i F) = (\prod i less m. rat-of-int (dk i F)) by simp
  also have \ldots = (\prod i less m. (\prod j less i. \|G ! j\|^2)) unfolding dk-def
    by (rule prod.cong, auto simp: Gramian-determinant[OF assms])
  also have \ldots = (\prod i less m. (\prod j less i. \|gs.gso (RAT F) ! j\|^2))
    by (intro prod.cong arg-cong[of - sq-norm-vec], insert conn, auto)
  also have \ldots ≤ (\prod i less m. (\prod j less i. \|F ! j\|^2)) unfolding D-def of-int-le-iff
    by (intro prod-mono ball ? conjI prod-nonneg, insert gs.sqrt-norm-sq-norm-ge-0)
qed

lemma LLL-measure-approx: assumes inv: LLL-invariant (i, Fr, Gr) F G
and α > 4/3
shows LLL-measure (i, Fr, Gr) ≤ m + 2 * m *
(\sum i less m. log ((4 * of-rat α) / (4 + of-rat α)) (of-int \|F ! i\|^2))
proof –
  have id: 1 / real-of-rat reduction = (4 * of-rat α) / (4 + of-rat α)
    unfolding reduction-def of-rat-divide of-rat-add of-rat-mult by simp
define b where b = (1 / real-of-rat reduction)
  have b1: b > 1 using reduction(3)[OF assms(2)] reduction(1) unfolding b-def
    by auto
from LLL-pos[OF inv] have D1: real (D F) ≥ 1 by auto
note incD = LLL-invD[OF inv]
from incD
have F: set F ⊆ carrier-vec n and len: length F = m by auto
from gs.lin-indpt-list-nonzero[OF incD(10)]
have θv, n ≤ set (RAT F) by auto
hence \( \theta_n n \notin \text{set } F \) using \( F \) by force

hence \( \theta: \land i. i < m \implies \text{sq-norm}(F ! i) \neq 0 \) using \( F \) sq-norm-vec-eq-0[of \( F \) ! i] for i n for i

unfolding set-conv-nth len by force

have 1: \( i < m \implies \text{sq-norm}(F ! i) \geq 1 \) for i using \( 0 \text{[of i]} \) sq-norm-vec-ge-0[of \( F ! i \)] by simp

from D-approx[OF inv]

have \( DF \leq \text{nat}(\prod i. j. \|F ! j\|^2) \) by auto

also have \( \ldots \leq \text{nat}(\prod i. j. \|F ! j\|^2) \)

proof (intro nat-mono prod-mono ballI conjI prod-nonneg)

fix i

assume i: \( i \in \{..<m\} \)

hence id: \( \{..<m\} = \{..<i\} \cup \{i..<m\} \) by auto

have \( (\prod j<i. \|F ! j\|^2) = (\prod j<i. \|F ! j\|^2) \times 1 \) by simp

also have \( \ldots \leq (\prod j<i. \|F ! j\|^2) \times (\prod j=i..<m. \|F ! j\|^2) \)

by (rule mult-left-mono[OF prod-ge-1 prod-nonneg], insert 1, auto)

also have \( \ldots = (\prod j<m. \|F ! j\|^2) \) unfolding id

by (subst prod.union-disjoint, auto)

finally show \( (\prod j<i. \|F ! j\|^2) \leq (\prod j<m. \|F ! j\|^2) \).

qed auto

also have \( (\prod i. j<m. \|F ! j\|^2) = (\prod i<m. \|F ! i\|^2) \times m \)

unfolding prod-constant by simp

finally have \( D: DF \leq \text{nat}(\prod i<m. \|F ! i\|^2) \times m \) (is - \( \leq \text{nat } e \))

have \( e: \emptyset e \geq 0 \) by (intro zero-le-power prod-nonneg, auto)

let \( \&\text{prod} = (\sum i<m. \text{real-of-int}(\|F ! i\|^2)) \)

let \( \&\text{sum} = (\sum i<m. \text{log b}(\text{of-int}(\|F ! i\|^2))) \)

have \( \&\text{prod0}: \&\text{prod} \geq 1 \) by (rule prod-pos, insert 1, force)

have \( \&\text{prod1}: \&\text{prod} \geq 1 \) by (rule prod-ge-1, insert 1, force)

from D have \( \text{real}(DF) \leq \text{real}(\text{nat } e) \) by blast

also have \( \ldots = \text{of-int } e \) using e by simp

also have \( \ldots = \&\text{prod} \times m \) by simp

finally have \( \text{log b}(\text{real}(DF)) \leq \text{log b}(\&\text{prod} \times m) \)

by (subst log-le-cancel-iff[OF b1], insert D1, auto)

have \( \text{real}(\log DF) = \text{real}(\text{nat}\{\text{log b}(\text{real}(DF))\}) \)

unfolding logD-def b-def using assms by auto

also have \( \ldots \leq \text{log b}(\text{real}(DF)) \) using b1 D1 by auto

also have \( \ldots \leq \text{log b}(\&\text{prod} \times m) \) by fact

also have \( \ldots = m \times \text{log b}\&\text{prod} \) unfolding log-nat-power[OF prod0] by simp

also have \( \ldots = m \times \&\text{sum} \)

by (subst log-prod, insert 1 b1, force+)

finally have \( \text{main}: \log D F \leq m \times \&\text{sum} \).

have \( \text{real}(\text{LLL-measure}(i, Fr, Gr)) = \text{real}(\text{2}\times\log D F + m - i) \)

unfolding LLL-measure-def split invD(1) by simp

also have \( \ldots \leq 2 \times \text{real}(\log DF) + m \) using invD by simp

also have \( \ldots \leq 2 \times m + m \) using main by auto

finally show \( \text{thesis} \) unfolding b-def id by simp

qed
lemma basis-reduction-main: fixes $F$ $G$ assumes LLL-invariant $L$ $\alpha$ state $F$ $G$
and basis-reduction-main $\alpha$ m state = state'$
and $\alpha \geq 4/3$
shows $\exists F'$ $G'$. LLL-invariant $L$ $\alpha$ state' $F'$ $G'$ ∧ fst state' = m
proof (cases $m = 0$)
case True
  from assms(2)[unfolded True basis-reduction-main.simps[of - 0 state]]
  have state': state' = state by auto
  obtain $i$ Fr Gr where state: state = $(i, Fr, Gr)$ by (cases state, auto)
  from LLL-invD[OF assms(1)[unfolded state]] True have $i = 0$ by auto
  show ?thesis using assms(1) unfolding state' state $i$ True by auto
next
case ne: False
  note [simp] = basis-reduction-main.simps
  show ?thesis using assms(1−2)
proof (induct state arbitrary: $F$ $G$ rule: wf-induct[OF wf-measure[of LLL-measure $\alpha$]])
case (1 state $F$ $G$)
  note inv = 1(2)
  note IH = 1(1)[rule-format]
  note res = 1(3)
  obtain $i$ Fr1 Gr1 where state: state = $(i, Fr1, Gr1)$ by (cases state, auto)
  note inv = inv[unfolded state]
  note res = res[unfolded state]
  show ?case
  proof (cases $i < m$)
    case True
    with inv have $i < m$ unfolding LLL-invariant-def by auto
    obtain state'" where $b$: basis-reduction-step $\alpha$ $(i, Fr1, Gr1) = state'"$ by auto
    from res True $b$
    have res: basis-reduction-main $\alpha$ m state" = state' by simp
    note bsr = basis-reduction-step[OF $\alpha$ inv $i$ $b$]
    from bsr(1) obtain $F'$ $G'$ where inv: LLL-invariant $L$ $\alpha$ state" $F'$ $G'$ by auto
    from bsr(2) have (state" ,state) ∈ measure (LLL-measure $\alpha$) by (auto simp: state)
    from IH[OF this inv] $b$ res state show ?thesis by auto
next
case False
  define $G1$ where $Gr$: $G1 = of-list-repr Fr1$
  note inv = inv[unfolded LLL-invariant-def split $Gr$[symmetric] Let-def]
  from False res have state': state' = $(i, Fr1, Gr1)$ by simp
  from False inv have $i = m$ unfolding LLL-invariant-def by auto
  show ?thesis using 1(2) unfolding state' state $i$ by auto
qed
qed
qed

context fixes α :: rat and F
assumes α: α ≥ 4 / 3
and lin-dep: gs.lin-indpt-list (RAT F)
and len: length F = m
begin

lemma basis-reduction-part-1: assumes basis-reduction-part-1 n α F = state
shows ∃ F' G'. LLL-invariant (lattice-of F) α state F' G' ∧ fst state = m

proof –
let ?F = RAT F
define Fr0::f-repr where Fr0 = ([], F)
have FrF: RAT (snd Fr0) = ?F unfolding Fr0-def by auto
from lin-dep
have F: set F ⊆ carrier-vec n
unfolding gs.lin-indpt-list-def by auto
have repr: f-repr 0 Fr0 F unfolding list-repr-def Fr0-def by auto
obtain G where gs: snd (gram-schmidt-int n F) = G (is snd ?gs = G) by force
from gram-schmidt.mn[OF lin-dep - gs]

have mn: m ≤ n by auto
have G':length (RAT F) = m m ≤ m m ≤ n
set ?F ⊆ carrier-vec n using F len mn by auto
define Gr0 where Gr0 = gram-schmidt-triv n (RAT F)
let ?Gr0 = ([], Gr0)
have RAT-carr: set (RAT F) ⊆ Rn using F len by auto
have take: RAT F = take m (RAT F) using len by auto
from gram-schmidt.partial-connect[OF G']
gs[unfolded gram-schmidt-int-def gram-schmidt-wit-def] take RAT-carr
have gso-init:Gr0 = map (λ x. (x, sq-norm x)) (map (GSO F) [0..<m])
unfolding Gr0-def FrF GSO-def gram-schmidt-triv using len by auto
from gram-schmidt-int-connect[OF lin-dep gs len]
have gso0: g-repr 0 ?Gr0 F unfolding gso-init g-repr-def list-repr-def gs by auto
have inv: LLL-invariant (lattice-of F) α (0, Fr0, ?Gr0) F G
by (rule LLL-invI[OF repr gso0 gs refl - - - lin-dep], auto simp:gs.weakly-reduced-def len)
obtain i Fr1 Gr1 where br:state = (i, Fr1, Gr1) by(cases state,auto)

note * = assms[unfolded basis-reduction-part-1-def o-def Let-def folded Gr0-def Fr0-def,unfolded len]
from basis-reduction-main[OF inv * α]
show ?thesis by auto
qed

shows ∃ F' G'. LLL-invariant (lattice-of F) α state F' G' ∧
fst state = 0 ∧ gs.strictly-reduced m α G' (gs.µ (RAT F'))
proof

obtain \(i\, Fr\, Gr\) where state: state = \((i, Fr, Gr)\) by (cases state, auto)

obtain state' where 1: basis-reduction-part-1 \(n\, \alpha\) \(F = \text{state}'\) by auto

from res[unfolded basis-reduction-part-2-def 1 state]

have 2: basis-reduction-part-2-main state' = \((i, Fr, Gr)\) by auto

from basis-reduction-part-1[OF 1] obtain \(H, Hs\)

where Linv: LLL-invariant (lattice-of \(F\)) \(\alpha\) state' \(H, Hs\)

and \(n\): \(\text{fst state}' = m\) by auto

from basis-reduction-part-2-main[OF Linv n 2] state

show ?thesis by auto

qed

lemma weakly-reduce-basis: assumes res: weakly-reduce-basis \(n\, \alpha\) \(F = (F', G')\)

shows lattice-of \(F\) = lattice-of \(F'\) (is ?g1)

gs.weakly-reduced \(\alpha\) \(m\) \(G'\) (is ?g2)

\(G' = \text{gram-schmidt} \, n\) (RAT \(F'\)) (is ?g3)

gs.lin-indpt-list (RAT \(F'\)) (is ?g4)

length \(F'\) = \(m\) (is ?g5)

proof

obtain \(i\, Fr\, Gr\) where 1: basis-reduction-part-1 \(n\, \alpha\) \(F = (i, Fr, Gr)\) (is ?main

= ?)

by (cases ?main) auto

from basis-reduction-part-1[OF 1] obtain \(F1\, G1\)

where Linv: LLL-invariant (lattice-of \(F\)) \(\alpha\) \((i, Fr, Gr)\) \(F1\, G1\)

and \(i\)-n: \(i = m\) by auto

from res[unfolded weakly-reduce-basis-def 1] have \(R: \text{of-list-repr} \, Fr\)

and \(Rs\): \(G' = \text{map} \, \text{fst} \, (\text{of-list-repr} \, Gr)\) by auto

note inv = LLL-invD[OF Linv]

from inv(1) \(R\) have RH: \(F' = F1\) unfolding of-list-repr-def by auto

with inv have Hs: \(G1 = \text{snd} \, (\text{gram-schmidt-int} \, n \, F')\) by auto

from inv(9)[unfolded g-repr-def]

have list-repr i Gr (map (\(\lambda x. \, (x, \|x\|^2)\)) (map (GSO \(F1\) \([0..<m]\)))) by auto

from Rs[unfolded of-list-repr[OF this]] have \(Rs\): \(G' = \text{map} \, (\text{GSO} \, F1) \, [0..<m]\)

by (auto simp: o-def)

also have \(\ldots = G1\) unfolding LLL-connect[OF Linv] unfolding GSO-def by simp

finally have RsHs: \(G' = G1\) by auto

from RsHs inv(4,5,6,10) Rs R Hs RH i-n show ?g1 ?g2 ?g3 ?g4 ?g5 by (auto simp: snd-grm-schmidt-int)

qed

lemma strictly-reduce-basis: fixes \(F'\, G'\) assumes res: strictly-reduce-basis \(n\, \alpha\) \(F = (F', G')\)

shows lattice-of \(F\) = lattice-of \(F'\) (is ?g1)

\(gs.\text{strictly-reduced} \, m\) \(\alpha\) \(G'\) (gs.\mu (RAT \(F'\))) (is ?g2)

\(G' = \text{gram-schmidt} \, n\) (RAT \(F'\)) (is ?g3)

\(gs.\text{lin-indpt-list} \, (RAT \(F'\))\) (is ?g4)

length \(F'\) = \(m\) (is ?g5)

proof
obtain \( i \ Fr G \) where 2: basis-reduction-part-2 \( n \ \alpha \ F = (i, \ Fr, G) \) (is ?main = \( \gamma \)) by (cases ?main) auto
from basis-reduction-part-2[OF 2] obtain \( F \) 1 \( G \)
where Linv: LLL-invariant (lattice-of \( F \) \( \alpha \) (i, \( Fr, G \)) \( F \) 1 \( G \))
and red: gs.strictly-reduced \( m \ \alpha \ G \) \( 1 \) \( (gs.\mu (RAT F)) \)
and \( \alpha \) 0: \( i = 0 \) by auto
from res[unfolded strictly-reduce-basis-def 2] have \( R \): \( F' = \) of-list-repr \( Fr \)
and \( R \): \( G' = \) map fst (of-list-repr \( Gr \)) by auto
note inv = LLL-invD[OF Linv]
from inv(1) \( R \) have \( RH: F' = F \) 1 unfolding of-list-repr-def by auto
with inv have Hs: \( G \) 1 = snd (gram-schmidt-int \( n \) \( F' \)) by auto
from inv(9)[unfolded g-repr-def]
have list-repr i Gr (map (\( \lambda x. (x, \| x \|^2) \)) \( map (GSO F1) [0 \ldots < m] \)) by auto
from Rs[unfolded of-list-repr[OF this]] have Rs: \( G' = \) map \( (GSO F1) [0 \ldots < m] \)
by (auto simp: o-def)
also have \( \ldots = G \) 1 unfolding LLL-connect[OF Linv] unfolding GSO-def by simp
finally have RsHs: \( G' = G \) 1 by auto
from RsHs inv(4,5,10) \( R \) \( H \) \( Rs \) \( RH \) red show ?g1 ?g2 ?g3 ?g4 ?g5 by (auto simp: snd-gram-schmidt-int)
qed

lemma short-vector: assumes short-vector \( \alpha \ F = v \)
and \( m0: m \neq 0 \)
shows \( v \in \) carrier-vec \( n \)
\( v \in \) lattice-of \( F = \{0_v, n\} \)
\( h \in \) lattice-of \( F = \{0_v, n\} \implies \) rat-of-int (sq-norm \( v \)) \( \leq \) \( \alpha \) \( (m - 1) \) * rat-of-int (sq-norm \( h \))
\( v \neq 0_v \)
proof –
let \( ?L = \) lattice-of \( F \)
have a1: \( \alpha \geq 1 \) using \( \alpha \) by auto
obtain \( F \) 1 \( G \) where weak: weakly-reduce-basis \( n \ \alpha \ F = (F1, \ G1) \) by force
from weakly-reduce-basis[OF weak] len have \( L: \) lattice-of \( F1 = ?L \)
and red: gs.weakly-reduced \( m \) \( G \)
and Gs: \( G \) 1 = gram-schmidt \( n \) (RAT \( F1 \))
and basis: gs.lin-indpt-list (RAT \( F1 \))
and lenH: length \( F1 = m \)
and H: set \( F1 \subseteq \) carrier-vec \( n \)
by (auto simp: gs.lin-indpt-list-def)
from lin-dep have \( G: \) set \( F \subseteq \) carrier-vec \( n \) unfolding gs.lin-indpt-list-def by auto
with \( m0 \) \( len \)
have dim-vec (hd \( F \)) = \( n \) by (cases \( F \), auto)
note res = assms[unfolded short-vector-def this weak]
from res \( m0 \) \( lenH \)
have \( v: v = F1 \) ! 0 by (cases \( F1 \), auto)
from gs.main-connect[OF basis refl] \( Gs \)
have gs: snd (gs.main (RAT \( F1 \))) = \( G1 \) by auto
let ?r = rat-of-int
let ?rv = map-vec ?r
let ?F = RAT F1
let ?h = ?rv h
{ assume h:h ∈ ?L \ - \ \{ 0_v \ n \} \ (is \ ?h-req)
  from h[folded L] have h: h ∈ lattice-of F1 h \neq \ 0_v \ n \ by \ auto
  { assume f: ?h = 0_v n
   have ?h = ?rv (0_v n) unfolding f by (intro eq-vec1, auto)
   hence h = 0_v n
   with h have False by simp
  } hence h0: ?h \neq \ 0_v \ n \ by \ auto
  with lattice-of-of-int[OF H h(1)]
  have ?h ∈ gs.lattice-of ?F \ - \ \{ 0_v \ n \} \ by \ auto
}
from gs.weakly-reduced-imp-short-vector[OF basis - gs red this a1] lenH
show h ∈ ?L \ - \ \{ 0_v \ n \} \ implies \ ?r (sq-norm v) \leq \ \alpha ^ m \ - \ 1 \ \ast \ ?r (sq-norm h)
  unfolding L v by (auto simp: sq-norm-of-int)
from m0 H lenH show vn: v ∈ carrier-vec n unfolding v by (cases F1, auto)
have vl: v ∈ ?L unfolding L[symmetric] v using m0 H lenH
  by (intro basis-in-latticeI, cases F1, auto)
  { assume v = 0_v n
   hence hd ?F = 0_v n unfolding v using m0 lenH by (cases F1, auto)
   with gs.lin-indpt-list-nonzero[OF basis] have False using m0 lenH by (cases F1, auto)
  }
  with vl show v: v ∈ ?L \ - \ \{ 0_v \ n \} \ by \ auto
  have jn:0_v j ∈ carrier-vec n \ implies \ j = n \ unfolding zero-vec-def carrier-vec-def
  by auto
  with v vn show v \neq 0_v j by auto
qed
end
end
end

References

