A verified LLL algorithm*

Ralph Bottesch Jose Divasón Maximilian Haslbeck
Sebastiaan Joosten René Thiemann Akihisa Yamada

March 19, 2025

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 [2].

Contents

1 Introduction **2**

2 Missing lemmas **3**

3 Auxiliary Lemmas and Definitions for Immutable Arrays **21**

4 Norms **22**
4.1 L-∞ Norms 22
4.2 Square Norms 24
4.2.1 Square norms for vectors 24
4.2.2 Square norm for polynomials 24
4.3 Relating Norms 25

5 Optimized Code for Integer-Rational Operations **36**

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

6 Representing Computation Costs as Pairs of Results and Costs **37**

7 List representation **37**

8 Gram-Schmidt **39**
8.1 Explicit Bounds for Size of Numbers that Occur During GSO Algorithm 88
8.2 Gram-Schmidt Implementation for Integer Vectors 94
8.3 Lemmas Summarizing All Bounds During GSO Computation 116

9 The LLL Algorithm **119**
9.1 Core Definitions, Invariants, and Theorems for Basic Version 120
9.2 Basic LLL implementation based on previous results 153
9.3 Integer LLL Implementation which Stores Multiples of the μ-Values 157
9.3.1 Updates of the integer values for Swap, Add, etc. . . . 157
9.3.2 Implementation of LLL via Integer Operations and Arrays 165
9.4 Bound on Number of Arithmetic Operations for Integer Implementation 182
9.5 Explicit Bounds for Size of Numbers that Occur During LLL Algorithm 198
9.5.1 *LLL-bound-invariant* is maintained during execution of *reduce-basis* 205
9.5.2 Bound extracted from *LLL-bound-invariant*. 213

10 Certification of External LLL Invocations **226**
10.1 Checking Results of External LLL Solvers 226
10.2 A Haskell Interface to the FPLLL-Solver 240

1 Introduction

The LLL basis reduction algorithm by Lenstra, Lenstra and Lovász [3] 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 $fs_0, \ldots, fs_{m-1} \in \mathbb{Z}^n$, it computes a short vector whose norm is at most $\alpha^{\frac{m-1}{2}}$ 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* $\alpha\ fs = v$
and *length* $fs = m$
and $m \neq 0$
shows $v \in lattice\_of\ fs - \{0_v\ n\}$
and $h \in lattice\_of\ fs - \{0_v\ n\} \longrightarrow \|v\|^2 \leq \alpha^{m-1} \cdot \|h\|^2$

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 [4]. 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.
- We prove polynomial runtime complexity by showing that there is a polynomial bound on the required number of arithmetic operations. Moreover, we formally prove that the representation size of the numbers that occur during the executation stays polynomial.

To our knowledge, this is the first formalization of the LLL algorithm in any theorem prover.

2 Missing lemmas

This theory contains many results that are important but not specific for our development. They could be moved to the stardard 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
Jordan-Normal-Form.VS-Connect
Berlekamp-Zassenhaus.Finite-Field-Factorization-Record-Based
Berlekamp-Zassenhaus.Berlekamp-Hensel
begin

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

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

subclass *semiring-1* **..**

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

lemma *zero-le-power* [*simp*]: *0* ≤ *a* ⟹ *0* ≤ *a* ^ *n*
by (*induct n*) *simp-all*

lemma *power-mono*: *a* ≤ *b* ⟹ *0* ≤ *a* ⟹ *a* ^ *n* ≤ *b* ^ *n*
by (*induct n*) (*auto intro*: *mult-mono order-trans* [*of 0 a b*])

lemma *one-le-power* [*simp*]: *1* ≤ *a* ⟹ *1* ≤ *a* ^ *n*
using *power-mono* [*of 1 a n*] **by** *simp*

lemma *power-le-one*: *0* ≤ *a* ⟹ *a* ≤ *1* ⟹ *a* ^ *n* ≤ *1*
using *power-mono* [*of a 1 n*] **by** *simp*

lemma *power-gt1-lemma*:
assumes *gt1*: *1* < *a*
shows *1* < *a* ∗ *a* ^ *n*
proof −
from *gt1* **have** *0* ≤ *a*
by (*fact order-trans* [*OF zero-le-one less-imp-le*])
from *gt1* **have** *1* ∗ *1* < *a* ∗ *1* **by** *simp*
also from *gt1* **have** ... ≤ *a* ∗ *a* ^ *n*
by (*simp only*: *mult-mono* ‹*0* ≤ *a*› *one-le-power order-less-imp-le zero-le-one order-refl*)
finally show *?thesis* **by** *simp*
qed

lemma *power-gt1*: $1 < a \Longrightarrow 1 < a \mathbin{\hat{}} Suc\ n$
by (*simp add*: *power-gt1-lemma*)

lemma *one-less-power* [*simp*]: $1 < a \Longrightarrow 0 < n \Longrightarrow 1 < a \mathbin{\hat{}} n$
by (*cases n*) (*simp-all add*: *power-gt1-lemma*)

lemma *power-decreasing*: $n \leq N \Longrightarrow 0 \leq a \Longrightarrow a \leq 1 \Longrightarrow a \mathbin{\hat{}} N \leq a \mathbin{\hat{}} n$
proof (*induction N*)
case (*Suc N*)
then have $a * a\hat{}N \leq 1 * a\hat{}n$ **if** $n \leq N$
using *that* **by** (*intro mult-mono*) *auto*
then show *?case*
using *Suc* **by** (*auto simp add*: *le-Suc-eq*)
qed (*auto*)

lemma *power-increasing*: $n \leq N \Longrightarrow 1 \leq a \Longrightarrow a \mathbin{\hat{}} n \leq a \mathbin{\hat{}} N$
proof (*induction N*)
case (*Suc N*)
then have $1 * a\hat{}n \leq a * a\hat{}N$ **if** $n \leq N$
using *that* **by** (*intro mult-mono*) (*auto simp add*: *order-trans*[*OF zero-le-one*])
then show *?case*
using *Suc* **by** (*auto simp add*: *le-Suc-eq*)
qed (*auto*)

lemma *power-Suc-le-self*: $0 \leq a \Longrightarrow a \leq 1 \Longrightarrow a \mathbin{\hat{}} Suc\ n \leq a$
using *power-decreasing* [*of 1 Suc n a*] **by** *simp*

end

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

subclass (**in** *ordered-idom*) *ordered-semiring-1* **by** *unfold-locales auto*

lemma *log-prod*: **assumes** $0 < a\ a \neq 1\ \bigwedge x.\ x \in X \Longrightarrow 0 < f\ x$
shows *log a* (*prod f X*) = *sum* (*log a o f*) *X*
using *assms*(*3*)
proof (*induct X rule*: *infinite-finite-induct*)
case (*insert x F*)
have *log a* (*prod f* (*insert x F*)) = *log a* (*f x* ∗ *prod f F*) **using** *insert* **by** *simp*
also have ... = *log a* (*f x*) + *log a* (*prod f F*)
by (*meson insert.prems insertCI log-mult-pos prod-pos*)
finally show *?case* **using** *insert* **by** *auto*
qed *auto*

subclass (**in** *ordered-idom*) *zero-less-one* **by** (*unfold-locales*, *auto*)
hide-fact *Missing-Ring.zero-less-one*

instance *real* :: *ordered-semiring-strict* **by** (*intro-classes*, *auto*)
instance *real* :: *linordered-idom*..

lemma *less-1-mult'*:
  fixes *a*::'*a*::*linordered-semidom*
  shows *1* < *a* ⟹ *1* ≤ *b* ⟹ *1* < *a* ∗ *b*
  by (*metis le-less less-1-mult mult.right-neutral*)

lemma *upt-minus-eq-append*: *i*≤*j* ⟹ *i*≤*j*−*k* ⟹ [*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 *?case*
    by (*metis Suc.prems*(*2*) *append.simps*(*1*) *diff-Suc-less nat-less-le neq0-conv upt-append upt-rec zero-diff*)
qed *auto*

lemma *list-trisect*: *x* < *length lst* ⟹ [*0*..<*length lst*] = [*0*..<*x*]@*x*#[*Suc x*..<*length lst*]
  by (*induct lst*, *force*, *rename-tac a lst*, *case-tac x* = *length lst*, *auto*)

lemma *id-imp-bij-betw*:
  assumes *f*: *f* : *A* → *A*
    and *ff*: ⋀*a*. *a* ∈ *A* ⟹ *f* (*f a*) = *a*
  shows *bij-betw f A A*
  by (*intro bij-betwI*[*OF f f*], *simp-all add*: *ff*)

lemma *range-subsetI*:
  assumes ⋀*x*. *f x* = *g* (*h x*) **shows** *range f* ⊆ *range g*
  using *assms* **by** *auto*

lemma *Gcd-uminus*:
  fixes *A*::*int set*
  assumes *finite A*
  shows *Gcd A* = *Gcd* (*uminus* ‘ *A*)
  using *assms*
  by (*induct A*, *auto*)

lemma *aux-abs-int*: **fixes** *c* :: *int*
  assumes *c* ≠ *0*
  shows |*x*| ≤ |*x* ∗ *c*|

proof −
have *abs x* = *abs x* ∗ *1* **by** *simp*
also have . . . ≤ *abs x* ∗ *abs c*
by (*rule mult-left-mono*, *insert assms*, *auto*)
finally show *?thesis* **unfolding** *abs-mult* **by** *auto*
qed

lemma *mod-0-abs-less-imp-0*:
fixes *a*::*int*
assumes *a1*: [*a* = *0*] (*mod m*)
and *a2*: *abs*(*a*)<*m*
shows *a* = *0*
proof −
have *m*≥*0* **using** *assms* **by** *auto*
thus *?thesis*
using *assms* **unfolding** *cong-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-trivial-iff*)
qed

lemma *sum-list-zero*:
assumes *set xs* ⊆ {*0*} **shows** *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* :: *'a* :: *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* :: *'a* :: *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 *1* = *le-max-self*[*OF this*]
from *Cons* **have** *a* = *x* ∨ *a* ∈ *set xs* **by** *auto*
then show *?thesis*
proof (*elim disjE*)
assume *a*: *a* = *x*
show *?thesis* **by** (*unfold a max-list-Cons*, *auto simp*: *False intro*!: *1*)
next
assume *a* ∈ *set xs*
then have *a* ≤ *max-list xs* **by** (*intro Cons*, *auto*)
with *1* **have** *a* ≤ *max x* (*max-list xs*) **by** (*auto dest*: *order-trans*)
then show *?thesis* **by** (*unfold max-list-Cons*, *auto simp*: *False*)
qed
qed (*insert Cons*, *auto*)
qed

lemma *le-max-list*:
fixes *xs* :: 'a :: *preorder list*
assumes *ord*: ⋀*a b*. *a* ∈ *set xs* ⟹ *b* ∈ *set xs* ⟹ *a* ≤ *b* ∨ *b* ≤ *a*
and *ab*: *a* ≤ *b*
and *b*: *b* ∈ *set xs*
shows *a* ≤ *max-list xs*
proof−
note *ab*
also have *b* ≤ *max-list xs*
by (*rule mem-set-imp-le-max-list*, *fact ord*, *fact b*)
finally show *?thesis***.**

qed

lemma *max-list-le*:
fixes *xs* :: *'a* :: *preorder list*
assumes *a*: $\bigwedge x.\ x \in set\ xs \Longrightarrow x \leq a$
and *xs*: $xs \neq []$
shows $max\text{-}list\ xs \leq a$
using *max-list-mem*[*OF xs*] *a* **by** *auto*

lemma *max-list-as-Greatest*:
assumes $\bigwedge x\ y.\ x \in set\ xs \Longrightarrow y \in set\ xs \Longrightarrow x \leq y \vee y \leq x$
shows $max\text{-}list\ xs = (GREATEST\ a.\ a \in set\ xs)$
proof (*cases* $xs = []$)
case *True*
then show *?thesis* **by** (*unfold Greatest-def*, *auto simp*: *max-list.simps*(*1*))
next
case *False*
from *assms* **have** *1*: $x \in set\ xs \Longrightarrow x \leq max\text{-}list\ xs$ **for** *x*
by (*auto intro*: *le-max-list*)
have *2*: $max\text{-}list\ xs \in set\ xs$ **by** (*fact max-list-mem*[*OF False*])
have $\exists!x.\ x \in set\ xs \wedge (\forall y.\ y \in set\ xs \longrightarrow y \leq x)$ (**is** $\exists!x.\ ?P\ x$)
proof (*intro ex1I*)
from *1 2*
show *?P* (*max-list xs*) **by** *auto*
next
fix *x* **assume** *3*: *?P x*
with *1* **have** $x \leq max\text{-}list\ xs$ **by** *auto*
moreover from *2 3* **have** $max\text{-}list\ xs \leq x$ **by** *auto*
ultimately show $x = max\text{-}list\ xs$ **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 $xs \neq []$
and $\bigwedge x\ y.\ x \in set\ xs \Longrightarrow y \in set\ xs \Longrightarrow h\ (max\ x\ y) = max\ (h\ x)\ (h\ y)$
shows $h\ (max\text{-}list\ xs) = max\text{-}list\ (map\ h\ xs)$
by (*insert assms*, *induct xs*, *auto simp*: *max-list-Cons max-list-mem*)

primrec *rev-upt* :: *nat* ⇒ *nat* ⇒ *nat list* (‹(*1*[->..-])›) **where**
rev-upt-0: [*0*>..*j*] = [] |
rev-upt-Suc: [(*Suc i*)>..*j*] = (*if i* ≥ *j then i* # [*i*>..*j*] *else* [])

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

definition *rev-upt-aux* :: *nat* ⇒ *nat* ⇒ *nat list* ⇒ *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*>..*j*] = *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-upt*:
rev [*i*..<*j*] = [*j*>..*i*]
by (*induct j*, *auto*)

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

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

lemma *nth-map-rev-upt*:
assumes *i*: *i* < *m*−*n*
shows (*map f* [*m*>..*n*]) ! *i* = *f* (*m* − *1* − *i*)
proof −
have (*map f* [*m*>..*n*]) ! *i* = *f* ([*m*>..*n*] ! *i*) **by** (*rule nth-map*, *auto simp add*: *i*)
also have ... = *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 *?thesis* **.**
qed

lemma *coeff-mult-monom*:
coeff (*p* ∗ *monom a d*) *i* = (*if d* ≤ *i then a* ∗ *coeff p* (*i* − *d*) *else 0*)
using *coeff-monom-mult*[*of a d p*] **by** (*simp add*: *ac-simps*)

lemma *vec-of-poly-0* [*simp*]: *vec-of-poly 0 = 0$_v$ 1* **by** (*auto simp*: *vec-of-poly-def*)

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* ∘ *Suc*) [*0*..<*n*] **by** (*simp add*: *map-upt-Suc del*: *upt-Suc*)
also have *Poly* (*rev* ...) = *Poly* (*rev* (*map* (*f* ∘ *Suc*) [*0*..<*n*])) + *monom* (*f 0*) *n*
by (*simp add*: *Poly-snoc smult-monom*)
also have ... = *poly-of-vec* (*vec n* (*f* ∘ *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* ((=) *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* ((=) *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*: *poly-of-vec v = Poly* (*rev* (*list-of-vec v*))
by (*induct v*, *auto simp*:*poly-of-vec-vCons Poly-snoc ac-simps*)

lemma *poly-of-vec-add*:
assumes *dim-vec a = dim-vec b*

shows *poly-of-vec* $(a + b)$ = *poly-of-vec* a + *poly-of-vec* b
using *assms*
by (*auto simp add*: *poly-eq-iff coeff-poly-of-vec*)

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

lemma (**in** *vec-module*) *poly-of-vec-finsum*:
assumes $f \in X \to$ *carrier-vec* n
shows *poly-of-vec* (*finsum* V f X) = $(\sum i \in X.$ *poly-of-vec* $(f\ i))$
proof (*cases finite* X)
case *False* **then show** *?thesis* **by** *auto*
next
case *True* **show** *?thesis*
proof (*insert True assms*, *induct* X *rule*: *finite-induct*)
case *IH*: (*insert* a X)
have [*simp*]: $f\ x \in$ *carrier-vec* n **if** x: $x \in X$ **for** x
using x *IH.prems* **unfolding** *Pi-def* **by** *auto*
have [*simp*]: $f\ a \in$ *carrier-vec* n **using** *IH.prems* **unfolding** *Pi-def* **by** *auto*
have [*simp*]: *dim-vec* (*finsum* V f X) = n **by** *simp*
have [*simp*]: *dim-vec* $(f\ a) = n$ **by** *simp*
show *?case*
proof (*cases* $a \in X$)
case *True* **then show** *?thesis* **by** (*auto simp*: *insert-absorb IH*)
next
case *False*
then have (*finsum* V f (*insert* a X)) = $f\ a$ + (*finsum* V f X)
by (*auto intro*: *finsum-insert IH*)
also have *poly-of-vec* ... = *poly-of-vec* $(f\ a)$ + *poly-of-vec* (*finsum* V f X)
by (*rule poly-of-vec-add*, *simp*)
also have ... = $(\sum i \in$*insert* a $X.$ *poly-of-vec* $(f\ i))$
using *IH False* **by** (*subst sum.insert*, *auto*)
finally show *?thesis* **.**
qed
qed *auto*
qed

definition *vec-of-poly-n* p n =
vec n $(\lambda i.$ *if* $i < n -$ *degree* $p - 1$ *then* 0 *else coeff* p $(n - i - 1))$

lemma *vec-of-poly-as*: *vec-of-poly-n* 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-vec-of-poly-n* [*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* @$_v$ *vec-of-list* [*a*]
(**is** *?l* = *?r*)
proof (*unfold vec-eq-iff*, *intro conjI allI impI*)
show *dim-vec ?l* = *dim-vec ?r* **by** *auto*
show *i* < *dim-vec ?r* ⟹ *?l* $ *i* = *?r* $ *i* **for** *i*
by (*cases n* − *i*, *auto simp*: *coeff-pCons less-Suc-eq-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* @$_v$ *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* [*0*] *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* $ (*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* $ (*n* − *Suc i*) =
(*if n* − *Suc i* < *n* − *Suc* (*degree p*) *then 0 else coeff p* (*n* − (*n* − *Suc i*) − *1*))
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_v *n*
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* ($\sum$ *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* ($\sum$ *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* ($\sum$ *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*
shows *poly-of-vec* (*a* ·$_v$ (*vec-of-poly-n b n*)) = *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 ≡

vec n (λ*i*. *if i* ≤ *j* ∧ *j* ≤ *s* + *i then coeff p* (*s* + *i* − *j*) *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 *col* (*sylvester-mat-sub m n p q*) *j* =
vec-of-poly-rev-shifted p n m j ⊛$_v$ *vec-of-poly-rev-shifted q m n j* (**is** *?l* = *?r*)
proof
show *dim-vec ?l* = *dim-vec ?r* **by** *simp*
fix *i* **assume** *i* < *dim-vec ?r* **then have** *i*: *i* < *m*+*n* **by** *auto*
show *?l* $ *i* = *?r* $ *i*
unfolding *vec-of-poly-rev-shifted-def*
apply (*subst index-col*) **using** *i* **apply** *simp* **using** *j* **apply** *simp*
apply (*subst sylvester-mat-sub-index*) **using** *i* **apply** *simp* **using** *j* **apply** *simp*
apply (*cases i* < *n*) **using** *i* **apply** *force* **using** *i*
apply (*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* ≡ *poly-of-vec v*
assumes *m*: *degree p* ≤ *m* **and** *n*: *dim-vec v* = *n*
assumes *j*: *j* < *m*+*n*
shows *vec-of-poly-rev-shifted p n m* (*n*+*m*−*Suc j*) ⋅ *v* = *coeff* (*p* ∗ *q*) *j* (**is** *?l* = *?r*)
proof −
have *id1*: ⋀ *i*. *m* + *i* − (*n* + *m* − *Suc j*) = *i* + *Suc j* − *n*
using *j* **by** *auto*
let *?g* = λ *i*. *if i* ≤ *n* + *m* − *Suc j* ∧ *n* − *Suc j* ≤ *i then coeff p* (*i* + *Suc j* − *n*) ∗ *v* $ *i else 0*
have *?thesis* = ((∑ *i* = *0*..<*n*. *?g i*) =
(∑ *i*≤*j*. *coeff p i* ∗ (*if j* − *i* < *n then v* $ (*n* − *Suc* (*j* − *i*)) *else 0*))) (**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 *?r* = (∑ *i*≤*j*. (*if j* − *i* < *n then coeff p i* ∗ *v* $ (*n* − *Suc* (*j* − *i*)) *else 0*)) (**is** - = *sum ?f* -)
by (*rule sum.cong*, *auto*)
also have *sum ?f* {..*j*} = *sum ?f* ({*i*. *i* ≤ *j* ∧ *j* − *i* < *n*} ∪ {*i*. *i* ≤ *j* ∧ ¬ *j* − *i* < *n*})
(**is** - = *sum* - (*?R1* ∪ *?R2*))
by (*rule sum.cong*, *auto*)

also have ... = *sum ?f ?R1 + sum ?f ?R2*
by (*subst sum.union-disjoint*, *auto*)
also have *sum ?f ?R2 = 0*
by (*rule sum.neutral*, *auto*)
also have *sum ?f ?R1 + 0 = sum* (λ *i. coeff p i * v $ (i + n − Suc j))* *?R1*
(**is** - = *sum ?F* -)
by (*subst sum.cong*, *auto simp*: *ac-simps*)
also have ... = *sum ?F* ((*?R1* ∩ {..*m*}) ∪ (*?R1* − {..*m*}))
(**is** - = *sum* - (*?R* ∪ *?R′*))
by (*rule sum.cong*, *auto*)
also have ... = *sum ?F ?R + sum ?F ?R′*
by (*subst sum.union-disjoint*, *auto*)
also have *sum ?F ?R′ = 0*
proof −
{
fix *x*
assume *x > m*
with *m*
have *?F x = 0* **by** (*subst coeff-eq-0*, *auto*)
}
thus *?thesis*
by (*subst sum.neutral*, *auto*)
qed
finally have *r*: *?r = sum ?F ?R* **by** *simp*

have *?l = sum ?g* ({*i. i < n* ∧ *i* ≤ *n + m − Suc j* ∧ *n − Suc j* ≤ *i*}
∪ {*i. i < n* ∧ ¬ (*i* ≤ *n + m − Suc j* ∧ *n − Suc j* ≤ *i*)})
(**is** - = *sum* - (*?L1* ∪ *?L2*))
by (*rule sum.cong*, *auto*)
also have ... = *sum ?g ?L1 + sum ?g ?L2*
by (*subst sum.union-disjoint*, *auto*)
also have *sum ?g ?L2 = 0*
by (*rule sum.neutral*, *auto*)
also have *sum ?g ?L1 + 0 = sum* (λ *i. coeff p* (*i + Suc j − n*) * *v $ i*) *?L1*
(**is** - = *sum ?G* -)
by (*subst sum.cong*, *auto*)
also have ... = *sum ?G* (*?L1* ∩ {*i. i + Suc j − n* ≤ *m*} ∪ (*?L1* − {*i. i + Suc j − n* ≤ *m*}))
(**is** - = *sum* - (*?L* ∪ *?L′*))
by (*subst sum.cong*, *auto*)
also have ... = *sum ?G ?L + sum ?G ?L′*
by (*subst sum.union-disjoint*, *auto*)
also have *sum ?G ?L′ = 0*
proof −
{
fix *x*
assume *x + Suc j − n > m*
with *m*
have *?G x = 0* **by** (*subst coeff-eq-0*, *auto*)

```
    }
    thus ?thesis
      by (subst sum.neutral, auto)
  qed
  finally have l: ?l = sum ?G ?L by simp

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

lemma sylvester-sub-poly:
 fixes p q :: 'a :: comm-semiring-0 poly
 assumes m: degree p ≤ m
 assumes n: degree q ≤ n
 assumes v: v ∈ carrier-vec (m+n)
 shows poly-of-vec ((sylvester-mat-sub m n p q)^T *_v v) =
   poly-of-vec (vec-first v n) * p + poly-of-vec (vec-last v m) * q (is ?l = ?r)
proof (rule poly-eqI)
 fix i
 let ?Tv = (sylvester-mat-sub m n p q)^T *_v v
 have dim: dim-vec (vec-first v n) = n dim-vec (vec-last v m) = m dim-vec ?Tv
= n + m
   using v by auto
 have if-distrib: ⋀ x y z. (if x then y else (0 :: 'a)) * z = (if x then y * z else 0)
   by auto
 show coeff ?l i = coeff ?r i
 proof (cases i < m+n)
   case False
     hence i-mn: i ≥ m+n
       and i-n: ⋀x. x ≤ i ∧ x < n ⟷ x < n
       and i-m: ⋀x. x ≤ i ∧ x < m ⟷ x < m by auto
     have coeff ?r i =
           (∑ x < n. vec-first v n $ (n − Suc x) * coeff p (i − x)) +
           (∑ x < m. vec-last v m $ (m − Suc x) * coeff q (i − x))
       (is - = sum ?f - + 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* $\{..<n\} = 0$ **by** *auto*
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* $\{..<m\} = 0$ **by** *auto*
finally have *coeff ?r i* $= 0$ **by** *auto*
also from *False* **have** $0 =$ *coeff ?l i*
unfolding *coeff-poly-of-vec dim sum.distrib*[*symmetric*] **by** *auto*
finally show *?thesis* **by** *auto*
next case *True*
hence *coeff ?l i* $= ((sylvester\text{-}mat\text{-}sub\ m\ n\ p\ q)^T *_v v)\ \$\ (n + m - Suc\ i)$
unfolding *coeff-poly-of-vec dim sum.distrib*[*symmetric*] **by** *auto*
also have ... $=$ *coeff* $(p * poly\text{-}of\text{-}vec\ (vec\text{-}first\ v\ n) + q * poly\text{-}of\text{-}vec\ (vec\text{-}last\ v\ m))\ i$
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 scalar-prod-append*)
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*]) **apply** *simp* **using** *True* **apply** *simp*
by *simp*
also have ... $=$
$(\sum x{\le}i.\ (if\ x < n\ then\ vec\text{-}first\ v\ n\ \$\ (n - Suc\ x)\ else\ 0) * coeff\ p\ (i - x))$
$+$
$(\sum x{\le}i.\ (if\ x < m\ then\ vec\text{-}last\ v\ m\ \$\ (m - Suc\ x)\ else\ 0) * coeff\ q\ (i - x))$
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 ... $=$ *coeff ?r i*
unfolding *coeff-add coeff-mult Let-def*
unfolding *coeff-poly-of-vec dim*..

finally show *?thesis*.
qed
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-not0*: *a* ≠ *0*
shows *degree* (*gcd a b*) ≤ *degree a*
proof −
let *?g* = *gcd a b*
have *gcd-dvd-b*: *?g dvd a* **by** *simp*
from *this* **obtain** *c* **where** *a-gc*: *a* = *?g* ∗ *c* **unfolding** *dvd-def* **by** *auto*
have *g-not0*: *?g* ≠*0* **using** *a-not0 a-gc* **by** *auto*
have *c0*: *c* ≠ *0* **using** *a-not0 a-gc* **by** *auto*
have *degree ?g* ≤ *degree* (*?g* ∗ *c*) **by** (*rule degree-mult-right-le*[*OF c0*])
also have ... = *degree a* **using** *a-gc* **by** *auto*
finally show *?thesis* .
qed

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

lemma *content-uminus*[*simp*]:
fixes *f*::*int poly*
shows *content* (−*f*) = *content f*
proof −
have −*f* = − (*smult 1 f*) **by** *auto*
also have ... = *smult* (−*1*) *f* **using** *smult-minus-left* **by** *auto*
finally have *content* (−*f*) = *content* (*smult* (−*1*) *f*) **by** *auto*
also have ... = *normalize* (− *1*) ∗ *content f* **unfolding** *content-smult* **..**
finally show *?thesis* **by** *auto*
qed

lemma *pseudo-mod-monic*:
fixes *f g* :: *'a*::{*comm-ring-1*,*semiring-1-no-zero-divisors*} *poly*
defines *r* ≡ *pseudo-mod f g*
assumes *monic-g*: *monic g*
shows ∃ *q*. *f* = *g* ∗ *q* + *r r* = *0* ∨ *degree r* < *degree g*
proof −
let *?cg* = *coeff g* (*degree g*)
let *?cge* = *?cg* ^ (*Suc* (*degree f*) − *degree g*)
define *a* **where** *a* = *?cge*
from *r-def*[*unfolded pseudo-mod-def*] **obtain** *q* **where** *pdm*: *pseudo-divmod f g* = (*q*, *r*)
by (*cases pseudo-divmod f g*) *auto*
have *g*: *g* ≠ *0* **using** *monic-g* **by** *auto*
from *pseudo-divmod*[*OF g pdm*] **have** *id*: *smult a f* = *g* ∗ *q* + *r* **and** *r* = *0* ∨ *degree r* < *degree g*
by (*auto simp*: *a-def*)
have *a1*: *a* = *1* **unfolding** *a-def* **using** *monic-g* **by** *auto*
hence *id2*: *f* = *g* ∗ *q* + *r* **using** *id* **by** *auto*
show *r* = *0* ∨ *degree r* < *degree g* **by** *fact*
from *g* **have** *a* ≠ *0*
by (*auto simp*: *a-def*)
with *id2* **show** ∃ *q*. *f* = *g* ∗ *q* + *r*
by *auto*
qed

lemma *monic-imp-div-mod-int-poly-degree*:
fixes *p* :: *'a*::{*comm-ring-1*,*semiring-1-no-zero-divisors*} *poly*
assumes *m*: *monic u*
shows ∃ *q r*. *p* = *q*∗*u* + *r* ∧ (*r* = *0* ∨ *degree r* < *degree u*)
using *pseudo-mod-monic*[*OF m*] **using** *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 (**in** *zero-hom*) *hom-upper-triangular*:
$A \in$ *carrier-mat n n* $\Longrightarrow$ *upper-triangular A* $\Longrightarrow$ *upper-triangular* (*map-mat hom A*)
by (*auto simp*: *upper-triangular-def*)

end

3 Auxiliary Lemmas and Definitions for Immutable Arrays

We define some definitions on immutable arrays, and modify the simplication rules so that IArrays will mainly operate pointwise, and not as lists. To be more precise, IArray.of-fun will become the main constructor.

theory *More-IArray*
imports *HOL−Library.IArray*
begin

definition *iarray-update* :: *'a iarray* $\Rightarrow$ *nat* $\Rightarrow$ *'a* $\Rightarrow$ *'a iarray* **where**
iarray-update a i x = *IArray.of-fun* (λ *j*. *if j* = *i then x else a* !! *j*) (*IArray.length a*)

lemma *iarray-cong*: *n* = *m* $\Longrightarrow$ ($\bigwedge$ *i*. *i* $<$ *m* $\Longrightarrow$ *f i* = *g i*) $\Longrightarrow$ *IArray.of-fun f n* = *IArray.of-fun g m*
by *auto*

lemma *iarray-cong'*: ($\bigwedge$ *i*. *i* $<$ *n* $\Longrightarrow$ *f i* = *g i*) $\Longrightarrow$ *IArray.of-fun f n* = *IArray.of-fun g n*
by (*rule iarray-cong*, *auto*)

lemma *iarray-update-length*[*simp*]: *IArray.length* (*iarray-update a i x*) = *IArray.length a*
unfolding *iarray-update-def* **by** *simp*

lemma *iarray-length-of-fun*[*simp*]: *IArray.length* (*IArray.of-fun f n*) = *n* **by** *simp*

lemma *iarray-update-of-fun*[*simp*]: *iarray-update* (*IArray.of-fun f n*) *i x* = *IArray.of-fun* (*f* (*i* := *x*)) *n*
unfolding *iarray-update-def iarray-length-of-fun*
by (*rule iarray-cong*, *auto*)

fun *iarray-append* **where** *iarray-append* (*IArray xs*) *x* = *IArray* (*xs* @ [*x*])

lemma *iarray-append-code*[*code*]: *iarray-append xs x* = *IArray* (*IArray.list-of xs* @ [*x*])
by (*cases xs*, *auto*)

lemma *iarray-append-of-fun*[*simp*]: *iarray-append* (*IArray.of-fun f n*) *x* = *IArray.of-fun* (*f* (*n* := *x*)) (*Suc n*)
by *auto*

declare *iarray-append.simps*[*simp del*]

lemma *iarray-of-fun-sub*[*simp*]: *i* < *n* ⟹ *IArray.of-fun f n* !! *i* = *f i*
by *auto*

lemma *IArray-of-fun-conv*: *IArray xs* = *IArray.of-fun* (λ *i*. *xs* ! *i*) (*length xs*)
by (*auto intro*!: *nth-equalityI*)

declare *IArray.of-fun-def*[*simp del*]
declare *IArray.sub-def*[*simp del*]

lemmas *iarray-simps* = *iarray-update-of-fun iarray-append-of-fun IArray-of-fun-conv iarray-of-fun-sub*

end

4 Norms

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

theory *Norms*
imports *HOL−Computational-Algebra.Polynomial*
Jordan-Normal-Form.Conjugate
Algebraic-Numbers.Resultant
Missing-Lemmas
begin

4.1 L-∞ Norms

consts *linf-norm* :: *'a* ⇒ *'b* (‹‖(-)‖$_\infty$›)

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*: $\|vec\ n\ f\|_\infty = max\text{-}list\ (map\ (abs \circ f)\ [0..<n]\ @\ [0])$
by (*simp add*: *linf-norm-vec-def*)

lemma *linf-norm-vec-vCons*[*simp*]: $\|vCons\ a\ v\|_\infty = max\ |a|\ \|v\|_\infty$
by (*auto simp*: *linf-norm-vec-def max-list-Cons*)

lemma *linf-norm-vec-0* [*simp*]: $\|vec\ 0\ f\|_\infty = 0$ **by** (*simp add*: *linf-norm-vec-def*)

lemma *linf-norm-zero-vec* [*simp*]: $\|0_v\ n :: 'a :: ordered\text{-}ab\text{-}group\text{-}add\text{-}abs\ vec\|_\infty = 0$
by (*induct n*, *simp add*: *zero-vec-def*, *auto simp*: *zero-vec-Suc*)

lemma *linf-norm-vec-ge-0* [*intro*!]:
fixes $v :: 'a :: ordered\text{-}ab\text{-}group\text{-}add\text{-}abs\ vec$
shows $\|v\|_\infty \geq 0$
by (*induct v*, *auto simp*: *max-def*)

lemma *linf-norm-vec-eq-0* [*simp*]:
fixes $v :: 'a :: ordered\text{-}ab\text{-}group\text{-}add\text{-}abs\ vec$
assumes $v \in carrier\text{-}vec\ n$
shows $\|v\|_\infty = 0 \longleftrightarrow v = 0_v\ n$
by (*insert assms*, *induct rule*: *carrier-vec-induct*, *auto simp*: *zero-vec-Suc max-def*)

lemma *linf-norm-vec-greater-0* [*simp*]:
fixes $v :: 'a :: ordered\text{-}ab\text{-}group\text{-}add\text{-}abs\ vec$
assumes $v \in carrier\text{-}vec\ n$
shows $\|v\|_\infty > 0 \longleftrightarrow v \neq 0_v\ n$
by (*insert assms*, *induct rule*: *carrier-vec-induct*, *auto simp*: *zero-vec-Suc max-def*)

lemma *linf-norm-poly-0* [*simp*]: $\|0::\text{-}\ poly\|_\infty = 0$
by (*simp add*: *linf-norm-poly-def*)

lemma *linf-norm-pCons* [*simp*]:
fixes $p :: 'a :: ordered\text{-}ab\text{-}group\text{-}add\text{-}abs\ poly$
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 $f :: 'a :: ordered\text{-}ab\text{-}group\text{-}add\text{-}abs\ poly$
shows $\|f\|_\infty \geq 0$
by (*induct f*, *auto simp*: *max-def*)

lemma *linf-norm-poly-eq-0* [*simp*]:
fixes $f :: 'a :: ordered\text{-}ab\text{-}group\text{-}add\text{-}abs\ poly$
shows $\|f\|_\infty = 0 \longleftrightarrow f = 0$
by (*induct f*, *auto simp*: *max-def*)

lemma *linf-norm-poly-greater-0* [*simp*]:

fixes $f :: 'a :: ordered\text{-}ab\text{-}group\text{-}add\text{-}abs\ poly$
shows $\|f\|_\infty > 0 \longleftrightarrow f \neq 0$
by (*induct f*, *auto simp*: *max-def*)

4.2 Square Norms

consts $sq\text{-}norm :: 'a \Rightarrow 'b\ (\langle\|(\text{-})\|^2\rangle)$

abbreviation $sq\text{-}norm\text{-}conjugate\ x \equiv x * conjugate\ x$
adhoc-overloading $sq\text{-}norm \rightleftharpoons sq\text{-}norm\text{-}conjugate$

4.2.1 Square norms for vectors

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

definition $sq\text{-}norm\text{-}vec\ v \equiv \sum x \leftarrow list\text{-}of\text{-}vec\ v.\ \|x\|^2$
adhoc-overloading $sq\text{-}norm \rightleftharpoons sq\text{-}norm\text{-}vec$

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*]: $\|vec\ 0\ f\|^2 = 0$
by (*simp add*: *sq-norm-vec-def*)

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

lemma *sq-norm-zero-vec*[*simp*]: $\|0_v\ n :: 'a :: conjugatable\text{-}ring\ vec\|^2 = 0$
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 $sq\text{-}norm\text{-}poly$ **where** $sq\text{-}norm\text{-}poly\ p \equiv \sum a \leftarrow coeffs\ p.\ \|a\|^2$

adhoc-overloading $sq\text{-}norm \rightleftharpoons sq\text{-}norm\text{-}poly$

lemma *sq-norm-poly-0* [*simp*]: $\|0::\text{-}poly\|^2 = 0$
by (*auto simp*: *sq-norm-poly-def*)

lemma *sq-norm-poly-pCons* [*simp*]:
fixes $a :: 'a :: conjugatable\text{-}ring$

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 $p :: 'a :: conjugatable\text{-}ordered\text{-}ring\ poly$
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 $p :: 'a :: \{conjugatable\text{-}ordered\text{-}ring, ring\text{-}no\text{-}zero\text{-}divisors\}\ poly$
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 $p :: 'a :: \{conjugatable\text{-}ordered\text{-}ring, ring\text{-}no\text{-}zero\text{-}divisors\}\ poly$
shows $\|p\|^2 > 0 \longleftrightarrow p \neq 0$
by (*auto simp*: *less-le*)

lemma *sq-norm-vec-of-poly* [*simp*]:
fixes $p :: 'a :: conjugatable\text{-}ring\ poly$
shows $\|vec\text{-}of\text{-}poly\ p\|^2 = \|p\|^2$
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 $v :: 'a :: conjugatable\text{-}ring\ vec$
shows $\|poly\text{-}of\text{-}vec\ v\|^2 = \|v\|^2$
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* ≡ *a* < *0* ∨ *a* = *0* ∨ *0* < *a*

class *semiring-real-line* = *ordered-semiring-strict* + *ordered-semiring-0* +
  assumes *add-pos-neg-is-real*: *a* > *0* ⟹ *b* < *0* ⟹ *is-real* (*a* + *b*)
    and *mult-neg-neg*: *a* < *0* ⟹ *b* < *0* ⟹ *0* < *a* ∗ *b*
    and *pos-pos-linear*: *0* < *a* ⟹ *0* < *b* ⟹ *a* < *b* ∨ *a* = *b* ∨ *b* < *a*
    and *neg-neg-linear*: *a* < *0* ⟹ *b* < *0* ⟹ *a* < *b* ∨ *a* = *b* ∨ *b* < *a*
begin

lemma *add-neg-pos-is-real*: *a* < *0* ⟹ *b* > *0* ⟹ *is-real* (*a* + *b*)
  using *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* ≤ *a* **and** *0* ≤ *b*
    and *a* < *b* ⟹ *thesis a* = *b* ⟹ *thesis b* < *a* ⟹ *thesis*
  shows *thesis*
  using *assms pos-pos-linear* **by** (*auto simp*: *le-less*)

lemma *nonpos-linorder-cases* [*consumes 2*, *case-names less eq greater*]:
  assumes *a* ≤ *0 b* ≤ *0*
    and *a* < *b* ⟹ *thesis a* = *b* ⟹ *thesis b* < *a* ⟹ *thesis*
  shows *thesis*
  using *assms neg-neg-linear* **by** (*auto simp*: *le-less*)

lemma *real-linear*:
  assumes *is-real a* **and** *is-real b* **shows** *a* < *b* ∨ *a* = *b* ∨ *b* < *a*
  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* ⟹ *thesis a* = *b* ⟹ *thesis b* < *a* ⟹ *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* ⟷ *a* ≤ *b*
    and *real-add-less-cancel-left-pos*: *c* + *a* < *c* + *b* ⟷ *a* < *b*
    and *real-add-le-cancel-right-pos*: *a* + *c* ≤ *b* + *c* ⟷ *a* ≤ *b*
    and *real-add-less-cancel-right-pos*: *a* + *c* < *b* + *c* ⟷ *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* ⟷ *a* ≤ *b*
    and *real-mult-less-cancel-left-pos*: *c* ∗ *a* < *c* ∗ *b* ⟷ *a* < *b*

and *real-mult-le-cancel-right-pos*: $a * c \leq b * c \longleftrightarrow a \leq b$
and *real-mult-less-cancel-right-pos*: $a * c < b * c \longleftrightarrow a < b$
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*: $\neg\ a \geq b \longleftrightarrow a < b$
and *not-less-real*: $\neg\ a > b \longleftrightarrow a \leq b$
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 $a * b = 0 \longleftrightarrow a = 0 \vee b = 0$
proof−
{ assume *l*: $a * b = 0$ **and** $a \neq 0$ **and** $b \neq 0$
with *a b* **have** $a < 0 \vee 0 < a$ **and** $b < 0 \vee 0 < b$ **by** *auto*
then have *False* **using** *mult-pos-pos*[*of a b*] *mult-pos-neg*[*of a b*] *mult-neg-pos*[*of a b*] *mult-neg-neg*[*of a b*]
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*: $c > 0$ **and** *a*: *is-real a* **and** *b*: *is-real b*
shows $c * max\ a\ b = max\ (c * a)\ (c * b)$
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*: $1 \leq a \Longrightarrow m \leq n \Longrightarrow a \leq b \Longrightarrow a \hat{}\ m \leq b \hat{}\ n$
using *power-mono*[*of a b n*] *power-increasing*[*of m n a*]
by (*auto simp*: *order.trans*[*OF zero-le-one*])

lemma *power-pos*:
assumes *a0*: $0 < a$ **shows** $0 < a \hat{}\ n$
by (*induct n*, *insert mult-strict-mono*[*OF a0*] *a0*, *auto*)

lemma *power-neg*:
assumes *a0*: $a < 0$ **shows** *odd* $n \Longrightarrow a \hat{}\ n < 0$ **and** *even* $n \Longrightarrow a \hat{}\ n > 0$

by (*atomize*(*full*), *induct n*, *insert a0*, *auto simp add*: *mult-pos-neg2 mult-neg-neg*)

lemma *power-ge-0-iff*:
assumes *a*: *is-real a*
shows $0 \leq a \mathbin{\hat{}} n \longleftrightarrow 0 \leq a \vee$ *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** $a \mathbin{\hat{}} n < b \mathbin{\hat{}} n \longleftrightarrow n > 0 \wedge a < b$
proof (*insert assms*, *induct n arbitrary*: *a b*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc n*)
note $a = \langle 0 \leq a \rangle$
note $b = \langle 0 \leq b \rangle$
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** $b \mathbin{\hat{}} n < a \mathbin{\hat{}} n$ **by** *auto*
with *mult-strict-mono′*[*OF greater this*] *b greater*
show *?thesis* **by** *auto*
qed
qed *auto*
qed

lemma *power-strict-mono*:
shows $a < b \Longrightarrow 0 \leq a \Longrightarrow 0 < n \Longrightarrow a \mathbin{\hat{}} n < b \mathbin{\hat{}} n$
by (*subst nonneg-power-less*, *auto*)

lemma *nonneg-power-le*:
assumes $0 \leq a$ **and** $0 \leq b$ **shows** $a\hat{}n \leq b\hat{}n \longleftrightarrow n = 0 \vee a \leq b$
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 $a = 0 \Longrightarrow$ *thesis* **and** $|a| > 0 \Longrightarrow$ *thesis* **shows** *thesis*
using *assms* **by** *auto*

lemma *abs-linorder-cases*[*case-names less eq greater*]:
assumes $|a| < |b| \Longrightarrow$ *thesis* **and** $|a| = |b| \Longrightarrow$ *thesis* **and** $|b| < |a| \Longrightarrow$ *thesis*
shows *thesis*
apply (*cases rule*: *nonneg-linorder-cases*[*of* $|a|$ $|b|$])
using *assms* **by** *auto*

lemma [*simp*]:
shows *not-le-abs-abs*: $\neg\ |a| \geq |b| \longleftrightarrow |a| < |b|$
and *not-less-abs-abs*: $\neg\ |a| > |b| \longleftrightarrow |a| \leq |b|$
by (*atomize*(*full*), *cases a b rule*: *abs-linorder-cases*, *auto simp*: *less-imp-le*)

lemma *abs-power-less* [*simp*]: $|a|\hat{}n < |b|\hat{}n \longleftrightarrow n > 0 \wedge |a| < |b|$
by (*subst nonneg-power-less*, *auto*)

lemma *abs-power-le* [*simp*]: $|a|\hat{}n \leq |b|\hat{}n \longleftrightarrow n = 0 \vee |a| \leq |b|$
by (*subst nonneg-power-le*, *auto*)

lemma *abs-power-pos* [*simp*]: $|a|\hat{}n > 0 \longleftrightarrow a \neq 0 \vee n = 0$
using *power-pos*[*of* $|a|$] **by** (*cases n*, *auto*)

lemma *abs-power-nonneg* [*intro*!]: $|a|\hat{}n \geq 0$ **by** *auto*

lemma *abs-power-eq-0* [*simp*]: $|a| \hat{} n = 0 \longleftrightarrow a = 0 \wedge n \neq 0$
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* $@_v$ *vec-of-list ys*
by (*auto simp*: *nth-append*)

lemma *linf-norm-vec-of-list*:
$\|$*vec-of-list xs*$\|_\infty$ = *max-list* (*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 $\|v\|_\infty$ = (*GREATEST a*. *a* ∈ *abs* ‘ *set* (*list-of-vec v*) ∪ {*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* ≠ *0*
shows *vec-of-poly* (*pCons a f*) = *vec-of-poly f* $@_v$ *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* ≠ *0*
shows *vec-of-poly f* = *vec-of-list* (*rev* (*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* :: *'a* :: *ring-1-abs-real-line poly*
shows $\|$*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 :: $'a$ :: *ring-1-abs-real-line poly*
shows $\|f\|_\infty$ = (*GREATEST a*. $a \in$ *abs* ' *set* (*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 :: $'a$ :: *ring-1-abs-real-line vec*
assumes $i <$ *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 :: $'a$ :: *ring-1-abs-real-line poly*
shows $|coeff\ f\ i| \leq \|f\|_\infty$
using *vec-index-le-linf-norm*[*of degree f − i vec-of-poly f*]
by (*cases i* $\leq$ *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-eq-iff mult-le-cancel-left mult-le-cancel-right mult-neg-neg less-complex-def less-eq-complex-def*)

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 $\|v\|^2 \leq$ *of-nat n* $*$ $\|v\|_\infty{}^2$
proof (*insert assms*, *induct rule*: *carrier-vec-induct*)
case (*Suc n a v*)
have [*dest*!]: ¬ |*a*| ≤ $\|v\|_\infty \Longrightarrow$ *of-nat n* $*$ $\|v\|_\infty{}^2 \leq$ *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-distribs 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 $\|p\|^2 \leq$ *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*$|^2 \leq \|f\|^2$
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 ∃ *i*. $\|f\|_\infty = |$*coeff f 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*: $\|f\|_\infty = |$*coeff f 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 1* :: $'a$ :: {*conjugatable-ring*, *ring-1*}) = *1*
proof−
have *conjugate 1* ∗ *1* = (*conjugate 1* :: $'a$) **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*: $\|$*map-vec of-int v* :: $'a$ :: {*conjugatable-ring*,*ring-1*} *vec*$\|^2$
= *of-int* $\|v\|^2$
unfolding *sq-norm-vec-as-cscalar-prod scalar-prod-def*
unfolding *hom-distribs*
by (*rule sum.cong*, *auto*)

definition *norm1 p* = *sum-list* (*map abs* (*coeffs p*))

lemma *norm1-ge-0*: *norm1* (*f* :: $'a$ :: {*abs*,*ordered-semiring-0*,*ordered-ab-group-add-abs*}*poly*)
≥ *0*
unfolding *norm1-def* **by** (*rule sum-list-nonneg*, *auto*)

lemma *norm2-norm1-main-equality*: **fixes** *f* :: *nat* ⇒ $'a$ :: *linordered-idom*
shows ($\sum$ *i* = *0*..<*n*. |*f i*|)^2 = ($\sum$ *i* = *0*..<*n*. *f i* ∗ *f i*)
+ ($\sum$ *i* = *0*..<*n*. $\sum$ *j* = *0*..<*n*. *if i* = *j then 0 else* |*f i*| ∗ |*f j*|)
proof (*induct n*)
case (*Suc n*)
have *id*: {*0* ..< *Suc n*} = *insert n* {*0* ..< *n*} **by** *auto*
have *id*: *sum f* {*0* ..< *Suc n*} = *f n* + *sum f* {*0* ..< *n*} **for** *f* :: *nat* ⇒ $'a$
unfolding *id* **by** (*rule sum.insert*, *auto*)
show *?case* **unfolding** *id power2-sum* **unfolding** *Suc*
by (*auto simp*: *power2-eq-square sum-distrib-left sum.distrib ac-simps*)
qed *auto*

lemma *norm2-norm1-main-inequality*: **fixes** $f :: nat \Rightarrow 'a :: linordered\text{-}idom$
shows $(\sum i = 0..<n.\ f\ i * f\ i) \leq (\sum i = 0..<n.\ |f\ i|)^2$
unfolding *norm2-norm1-main-equality*
by (*auto intro*!: *sum-nonneg*)

lemma *norm2-le-norm1-int*: $\|f :: int\ poly\|^2 \leq (norm1\ f)\hat{}2$
proof −
define F **where** $F = (!)\ (coeffs\ f)$
define n **where** $n = length\ (coeffs\ f)$
have *1*: $\|f\|^2 = (\sum i = 0..<n.\ F\ i * F\ i)$
unfolding *norm1-def sq-norm-poly-def sum-list-sum-nth F-def n-def*
by (*subst sum.cong*, *auto simp*: *power2-eq-square*)
have *2*: $norm1\ f = (\sum i = 0..<n.\ |F\ i|)$
unfolding *norm1-def sq-norm-poly-def sum-list-sum-nth F-def n-def*
by (*subst sum.cong*, *auto*)
show *?thesis* **unfolding** *1 2* **by** (*rule norm2-norm1-main-inequality*)
qed

lemma *sq-norm-smult-vec*: $sq\text{-}norm\ ((c :: 'a :: \{conjugatable\text{-}ring, comm\text{-}semiring\text{-}0\}) \cdot_v v) = (c * conjugate\ c) * sq\text{-}norm\ v$
unfolding *sq-norm-vec-as-cscalar-prod*
by (*subst scalar-prod-smult-left*, *force*, *unfold conjugate-smult-vec*,
subst scalar-prod-smult-right, *force*, *simp add*: *ac-simps*)

lemma *vec-le-sq-norm*:
fixes $v :: 'a :: conjugatable\text{-}ring\text{-}1\text{-}abs\text{-}real\text{-}line\ vec$
assumes $v \in carrier\text{-}vec\ n\ i < n$
shows $|v\ \$\ i|^2 \leq \|v\|^2$
using *assms* **proof** (*induction v arbitrary*: *i*)
case (*Suc n a v i*)
note *IH* = *Suc*
show *?case*
proof (*cases i*)
case (*Suc ii*)
then show *?thesis*
using *IH IH*(*2*)[*of ii*] *le-add-same-cancel2 order-trans* **by** *fastforce*
qed *auto*
qed *auto*

class *trivial-conjugatable* =
conjugate +
assumes *conjugate-id* [*simp*]: $conjugate\ x = x$

class *trivial-conjugatable-ordered-field* =
conjugatable-ordered-field + *trivial-conjugatable*

class *trivial-conjugatable-linordered-field* =
trivial-conjugatable-ordered-field + *linordered-field*

begin
subclass *conjugatable-ring-1-abs-real-line*
by (*standard*) (*auto simp add*: *semiring-normalization-rules*)
end

instance *rat* :: *trivial-conjugatable-linordered-field*
by (*standard*, *auto*)

instance *real* :: *trivial-conjugatable-linordered-field*
by (*standard*, *auto*)

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

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 *scalar-prod-Cauchy*:
fixes *u v*::*'a* :: {*trivial-conjugatable-linordered-field*} *Matrix.vec*
assumes *u* ∈ *carrier-vec n v* ∈ *carrier-vec n*
shows (*u* ⋅ *v*)^2 ≤ ‖*u*‖^2 ∗ ‖*v*‖^2
proof −
{ assume *v-0*: *v* ≠ 0_v *n*
have *0* ≤ (*u* − *r* ⋅$_v$ *v*) ⋅ (*u* − *r* ⋅$_v$ *v*) **for** *r*
by (*simp add*: *scalar-prod-ge-0*)
also have (*u* − *r* ⋅$_v$ *v*) ⋅ (*u* − *r* ⋅$_v$ *v*) = *u* ⋅ *u* − *r* ∗ (*u* ⋅ *v*) − *r* ∗ (*u* ⋅ *v*) + *r* ∗ *r* ∗ (*v* ⋅ *v*) **for** *r*::*'a*
proof −
have (*u* − *r* ⋅$_v$ *v*) ⋅ (*u* − *r* ⋅$_v$ *v*) = (*u* − *r* ⋅$_v$ *v*) ⋅ *u* − (*u* − *r* ⋅$_v$ *v*) ⋅ (*r* ⋅$_v$ *v*)
using *assms* **by** (*subst scalar-prod-minus-distrib*) *auto*
also have ... = *u* ⋅ *u* − (*r* ⋅$_v$ *v*) ⋅ *u* − *r* ∗ ((*u* − *r* ⋅$_v$ *v*) ⋅ *v*)
using *assms* **by** (*subst minus-scalar-prod-distrib*) *auto*
also have ... = *u* ⋅ *u* − *r* ∗ (*v* ⋅ *u*) − *r* ∗ (*u* ⋅ *v* − *r* ∗ (*v* ⋅ *v*))
using *assms* **by** (*subst minus-scalar-prod-distrib*) *auto*
also have ... = *u* ⋅ *u* − *r* ∗ (*u* ⋅ *v*) − *r* ∗ (*u* ⋅ *v*) + *r* ∗ *r* ∗ (*v* ⋅ *v*)
using *assms comm-scalar-prod* **by** (*auto simp add*: *field-simps*)
finally show *?thesis*
by *simp*
qed
also have *u* ⋅ *u* − *r* ∗ (*u* ⋅ *v*) − *r* ∗ (*u* ⋅ *v*) + *r* ∗ *r* ∗ (*v* ⋅ *v*) = *sq-norm u* − (*u* ⋅ *v*)^2 / *sq-norm v*
if *r* = (*u* ⋅ *v*) / (*v* ⋅ *v*) **for** *r*
unfolding *that* **by** (*auto simp add*: *sq-norm-vec-as-cscalar-prod power2-eq-square*)
finally have *0* ≤ ‖*u*‖^2 − (*u* ⋅ *v*)^2 / ‖*v*‖^2
by *auto*

then have $(u \cdot v)^2 \ / \ \|v\|^2 \leq \|u\|^2$
by *auto*
then have $(u \cdot v)^2 \leq \|u\|^2 * \|v\|^2$
using *pos-divide-le-eq*[*of* $\|v\|^2$] *v-0 assms* **by** (*auto*)
}
then show *?thesis*
by (*fastforce simp add*: *assms*)
qed

end

5 Optimized Code for Integer-Rational Operations

theory *Int-Rat-Operations*
imports
Sqrt-Babylonian.Sqrt-Babylonian-Auxiliary
Norms
begin

definition *int-times-rat* :: *int* ⇒ *rat* ⇒ *rat* **where** *int-times-rat i x* = *of-int i* ∗ *x*

declare *int-times-rat-def*[*simp*]

lemma *int-times-rat-code*[*code abstract*]: *quotient-of* (*int-times-rat i x*) =
(*case quotient-of x of* (*n*,*d*) ⇒ *Rat.normalize* (*i* ∗ *n*, *d*))
unfolding *int-times-rat-def rat-times-code* **by** *auto*

definition *square-rat* :: *rat* ⇒ *rat* **where** [*simp*]: *square-rat x* = *x* ∗ *x*

lemma *quotient-of-square*: **assumes** *quotient-of x* = (*a*,*b*)
shows *quotient-of* (*x* ∗ *x*) = (*a* ∗ *a*, *b* ∗ *b*)
proof −
have *b0*: *b* > *0 b* ≠ *0* **using** *quotient-of-denom-pos*[*OF assms*] **by** *auto*
hence *b*: (*b* ∗ *b* > *0*) = *True* **by** *auto*
show *?thesis*
unfolding *rat-times-code assms Let-def split Rat.normalize-def fst-conv snd-conv b if-True*
using *quotient-of-coprime*[*OF assms*] *b0* **by** *simp*
qed

lemma *square-rat-code*[*code abstract*]: *quotient-of* (*square-rat x*) = (*case quotient-of x of* (*n*,*d*)
⇒ (*n* ∗ *n*, *d* ∗ *d*)) **using** *quotient-of-square*[*of x*] **unfolding** *square-rat-def*
by (*cases quotient-of x*, *auto*)

definition *scalar-prod-int-rat* :: *int vec* ⇒ *rat vec* ⇒ *rat* (**infix** ‹·*i*› *70*) **where**
x ·*i y* = (*y* · *map-vec rat-of-int x*)

lemma *scalar-prod-int-rat-code*[*code*]: $v \cdot_i w = (\sum i = 0..<\textit{dim-vec}\ v.\ \textit{int-times-rat}\ (v\ \$\ i)\ (w\ \$\ i))$
unfolding *scalar-prod-int-rat-def Let-def scalar-prod-def int-times-rat-def*
by (*rule sum.cong*, *auto*)

lemma *scalar-prod-int-rat*[*simp*]: *dim-vec* x = *dim-vec* $y \Longrightarrow x \cdot_i y$ = *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 *sq-norm-vec-rat* :: *rat vec* ⇒ *rat* **where** [*simp*]: *sq-norm-vec-rat* x = *sq-norm-vec* x

lemma *sq-norm-vec-rat-code*[*code*]: *sq-norm-vec-rat* $x = (\sum x \leftarrow \textit{list-of-vec}\ x.\ \textit{square-rat}\ x)$
unfolding *sq-norm-vec-rat-def sq-norm-vec-def square-rat-def* **by** *auto*

end

6 Representing Computation Costs as Pairs of Results and Costs

theory *Cost*
imports *Main*
begin

type-synonym $'a$ *cost* = $'a \times$ *nat*

definition *cost* :: $'a$ *cost* ⇒ *nat* **where** *cost* = *snd*
definition *result* :: $'a$ *cost* ⇒ $'a$ **where** *result* = *fst*

lemma *cost-simps*: *cost* (a,c) = c *result* (a,c) = a
unfolding *cost-def result-def* **by** *auto*

lemma *result-costD*: **assumes** *result f-c* = f
cost f-c ≤ b
f-c = (a,c)
shows $a = f\ c \le b$ **using** *assms* **by** (*auto simp*: *cost-simps*)

lemma *result-costD'*: **assumes** *result f-c* = $f \wedge$ *cost f-c* ≤ b
f-c = (a,c)
shows $a = f\ c \le b$ **using** *assms* **by** (*auto simp*: *cost-simps*)

end

7 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* $\lambda\ xs.\ rev\ (take\ (Suc\ j)\ xs)$]
by (*simp add*: *min-def*)
qed

type-synonym $'a\ list\text{-}repr = 'a\ list \times 'a\ list$

definition *list-repr* :: $nat \Rightarrow 'a\ list\text{-}repr \Rightarrow 'a\ list \Rightarrow bool$ **where**
$list\text{-}repr\ i\ ba\ xs = (i \leq length\ xs \wedge fst\ ba = rev\ (take\ i\ xs) \wedge snd\ ba = drop\ i\ xs)$

definition *of-list-repr* :: $'a\ list\text{-}repr \Rightarrow 'a\ list$ **where**
$of\text{-}list\text{-}repr\ ba = (rev\ (fst\ ba)\ @\ snd\ ba)$

lemma *of-list-repr*: $list\text{-}repr\ i\ ba\ xs \Longrightarrow of\text{-}list\text{-}repr\ ba = xs$
unfolding *of-list-repr-def list-repr-def* **by** *auto*

definition *get-nth-i* :: $'a\ list\text{-}repr \Rightarrow 'a$ **where**
$get\text{-}nth\text{-}i\ ba = hd\ (snd\ ba)$

definition *get-nth-im1* :: $'a\ list\text{-}repr \Rightarrow 'a$ **where**
$get\text{-}nth\text{-}im1\ ba = hd\ (fst\ ba)$

lemma *get-nth-i*: $list\text{-}repr\ i\ ba\ xs \Longrightarrow i < length\ xs \Longrightarrow get\text{-}nth\text{-}i\ ba = xs\ !\ i$
unfolding *list-repr-def get-nth-i-def*
by (*auto simp*: *hd-drop-conv-nth*)

lemma *get-nth-im1*: $list\text{-}repr\ i\ ba\ xs \Longrightarrow i \neq 0 \Longrightarrow get\text{-}nth\text{-}im1\ ba = xs\ !\ (i - 1)$
unfolding *list-repr-def get-nth-im1-def*
by (*cases i*, *auto simp*: *rev-take-Suc*)

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

lemma *Cons-tl-drop-update*: $i < length\ xs \Longrightarrow x\ \#\ tl\ (drop\ i\ xs) = drop\ i\ (xs[i := x])$
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* $\Longrightarrow$ *i* < *length xs* $\Longrightarrow$ *list-repr i* (*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* $\Longrightarrow$ *i* $\neq$ *0* $\Longrightarrow$ *list-repr i* (*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* (*drop i xs*) = *drop* (*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* $\Longrightarrow$ *i* < *length xs* $\Longrightarrow$ *list-repr* (*Suc i*) (*inc-i ba*) *xs*
unfolding *list-repr-def inc-i-def* **by** (*cases ba*, *auto simp*: *rev-take-Suc hd-drop-conv-nth tl-drop-Suc*)

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

lemma *dec-i*: *list-repr i ba xs* $\Longrightarrow$ *i* $\neq$ *0* $\Longrightarrow$ *list-repr* (*i* − *1*) (*dec-i ba*) *xs*
unfolding *list-repr-def dec-i-def*
by (*cases ba*; *cases i*, *auto simp*: *rev-take-Suc hd-drop-conv-nth Cons-nth-drop-Suc*)

lemma *dec-i-Suc*: *list-repr* (*Suc i*) *ba xs* $\Longrightarrow$ *list-repr i* (*dec-i ba*) *xs*
using *dec-i*[*of Suc i ba xs*] **by** *auto*

end

8 Gram-Schmidt

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

Int-Rat-Operations
begin

unbundle *no m-inv-syntax*

lemma *rev-unsimp*: *rev xs* @ (*r* # *rs*) = *rev* (*r*#*xs*) @ *rs* **by** *simp*

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

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*])
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 ccontr)
    assume ¬ distinct fs
    hence card (set fs) < length fs using antisym-conv1 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] assms unfolding lin-indpt-list-def basis-list-def
by auto

lemma basis-det-nonzero:
  assumes db:basis (set G) and len:length G = n
  shows det (mat-of-rows n G) ≠ 0
proof −
  have M-car1:mat-of-rows n G ∈ carrier-mat n n using assms by auto
  hence M-car:(mat-of-rows n G)^T ∈ carrier-mat n n by auto
  have li:lin-indpt (set G)
   and inc-2:set G ⊆ carrier-vec n
   and issp:carrier-vec n = span (set G)
   and RG-in-carr:⋀i. i < length G ⟹ G ! i ∈ carrier-vec n
    using assms[unfolded basis-def] by auto
  hence basis-list G unfolding basis-list-def using len by auto
  from basis-list-basis[OF this] have di:distinct G by auto
 have det ((mat-of-rows n G)^T) ≠ 0 unfolding det-0-iff-vec-prod-zero[OF M-car]

  proof
    assume ∃ v. v ∈ carrier-vec n ∧ v ≠ 0_v n ∧ (mat-of-rows n G)^T *_v v = 0_v n
    then obtain v where v:v ∈ span (set G)
                        v ≠ 0_v n (mat-of-rows n G)^T *_v v = 0_v n
      unfolding issp by blast
    from finite-in-span[OF finite-set inc-2 v(1)] obtain a
      where aA: v = lincomb a (set G) by blast
    from v(1,2)[folded issp] obtain i where i:v $ i ≠ 0 i < n by fastforce
    hence inG:G ! i ∈ set G using len by auto
    have di2: distinct [0..<length G] by auto
    define f where f = (λl. ∑ i ∈ set [0..<length G]. if l = G ! i then v $ i else
0)
    hence f':f (G ! i) = (∑ ia←[0..<n]. if G ! ia = G ! i then v $ ia else 0)
      unfolding f-def sum.distinct-set-conv-list[OF di2] unfolding len by metis
    from v have mat-of-cols n G *_v v = 0_v n
```

```
    unfolding transpose-mat-of-rows by auto
   with mat-of-cols-mult-as-finsum[OF v(1)[folded issp len] RG-in-carr]
   have f:lincomb f (set G) = 0_v n unfolding len f-def by auto
   note [simp] = list-trisect[OF i(2)[folded len],unfolded len]
   note x = i(2)[folded len]
   have [simp]:(∑ x←[0..<i]. if G ! x = G ! i then v $ x else 0) = 0
    by (rule sum-list-0,auto simp: nth-eq-iff-index-eq[OF di less-trans[OF - x] x])
   have [simp]:(∑ x←[Suc i..<n]. if G ! x = G ! i then v $ x 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 ! i) ≠ 0 unfolding 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 n 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 ∉ ?E 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 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 v unfolding Vid by auto
  from dist i have diff: us ! i ≠ us ! j unfolding distinct-conv-nth by auto
  have vspan: ?v ∉ span ?E
  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 in-spanE[OF this] obtain a A where lc: us ! i = lincomb a A and A:
finite A
```

$A \subseteq$ *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*(*1*)], *insert A us lc usi diff*, *auto*)
from *not-lindepD*[*OF indep - - - this*] *A usi*
show *False* **by** *auto*
qed
hence *vmem*: *?v* $\notin$ *?E* **using** *span-mem*[*OF E*, *of ?v*] **by** *auto*
from *lin-dep-iff-in-span*[*OF E indepE v this*] *vspan*
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* $\Longrightarrow$ *sumlist* (*map* (λ *i*. *g i* $\cdot_v$ *gs* ! *i*) [*0* ..< *k*]) $\cdot$ *sumlist* (*map* (λ *i*. *h i* $\cdot_v$ *gs* ! *i*) [*0* ..< *k*])
= *sum-list* (*map* (λ *i*. *g i* $*$ *h i* $*$ (*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*]: $\bigwedge$ *i*. *i* $\leq$ *k* $\Longrightarrow$ *gs* ! *i* $\in$ *carrier-vec n* **by** *auto*
from *Suc* **have** *kn*: *k* $\leq$ *?m* **and** *k*: *k* < *?m* **by** *auto*
let *?v1* = *sumlist* (*map* (λ*i*. *g i* $\cdot_v$ *gs* ! *i*) [*0*..<*k*])
let *?v2* = (*g k* $\cdot_v$ *gs* ! *k*)
let *?w1* = *sumlist* (*map* (λ*i*. *h i* $\cdot_v$ *gs* ! *i*) [*0*..<*k*])
let *?w2* = (*h k* $\cdot_v$ *gs* ! *k*)
from *Suc* **have** *id*: [*0* ..< *Suc k*] = [*0* ..< *k*] @ [*k*] **by** *simp*
have *id*: *sumlist* (*map* (λ*i*. *g i* $\cdot_v$ *gs* ! *i*) [*0*..<*Suc k*]) = *?v1* + *?v2*
sumlist (*map* (λ*i*. *h i* $\cdot_v$ *gs* ! *i*) [*0*..<*Suc k*]) = *?w1* + *?w2*
unfolding *id map-append*
by (*subst sumlist-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 *w1*: *?w1* $\in$ *carrier-vec n* **by** (*rule sumlist-carrier*, *insert Suc*(*2*), *auto*)
have *w2*: *?w2* $\in$ *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 *w12*: *?w1* + *?w2* $\in$ *carrier-vec n* **using** *w1 w2* **by** *auto*
have *0*: $\bigwedge$ *g h*. *i* < *k* $\Longrightarrow$ (*g* $\cdot_v$ *gs* ! *i*) $\cdot$ (*h* $\cdot_v$ *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 1: ?v1 · ?w2 = 0
    by (subst scalar-prod-left-sum-distrib[OF - w2], (insert Suc(2), auto)[1], rule
sum-list-neutral,
        insert 0, auto)
  have 2: ?v2 · ?w1 = 0 unfolding comm-scalar-prod[OF v2 w1]
    apply (subst scalar-prod-left-sum-distrib[OF - v2])
     apply ((insert gs, force)[1])
    apply (rule sum-list-neutral)
    by (insert 0, auto)
  show ?case unfolding id
    unfolding scalar-prod-add-distrib[OF v12 w1 w2]
      add-scalar-prod-distrib[OF v1 v2 w1]
      add-scalar-prod-distrib[OF v1 v2 w2]
      scalar-prod-smult-distrib[OF w2 gsk]
      smult-scalar-prod-distrib[OF gsk gsk]
    unfolding Suc(1)[OF kn]
    by (simp add: 1 2 comm-scalar-prod[OF v2 w1])
qed auto
end


locale gram-schmidt = cof-vec-space n f-ty
  for n :: nat and f-ty :: 'a :: {trivial-conjugatable-linordered-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 Gramian-determinant-0 [simp]: Gramian-determinant G 0 = 1
  unfolding Gramian-determinant-def Gramian-matrix-def Let-def
  by (simp add: times-mat-def)

lemma orthogonal-imp-lin-indpt-list:
  assumes ortho: orthogonal gs and gs: set gs ⊆ carrier-vec n
  shows lin-indpt-list gs
proof −
  from corthogonal-distinct[of gs] ortho have dist: distinct gs by simp
  show ?thesis unfolding lin-indpt-list-def
  proof (intro conjI gs dist finite-lin-indpt2 finite-set)
    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: 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 ∀ v∈set gs. lc v = 0 by auto
  qed
qed

lemma orthocompl-span:
  assumes ⋀ x. x ∈ S ⟹ v · x = 0 S ⊆ carrier-vec n and [intro]: v ∈ carrier-vec
n
  and y ∈ span S
  shows v · 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 v) ∈ A → carrier-vec n
by auto
   have ∀ x∈A. (a x ·v x) · 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 x) · v = 0
       apply(subst smult-scalar-prod-distrib) by force+
   qed
```

hence $v \cdot lincomb\ a\ A = 0$ **apply**(*subst comm-scalar-prod*) **apply** *force+* **unfolding** *lincomb-def*
apply(*subst finsum-scalar-prod-sum*) **by** *force+*
}
thus *?thesis* **using** ‹$y \in span\ S$› **unfolding** *span-def* **by** *auto*
qed

lemma *orthogonal-sumlist*:
assumes *ortho*: $\bigwedge x.\ x \in set\ S \Longrightarrow v \cdot x = 0$ **and** *S*: $set\ S \subseteq carrier\text{-}vec\ n$ **and** *v*: $v \in carrier\text{-}vec\ n$
shows $v \cdot sumlist\ S = 0$
by (*rule orthocompl-span*[*OF ortho S v sumlist-in-span*[*OF S span-mem*[*OF S*]]])

lemma *oc-projection-alt-def*:
assumes *carr*:$(W::'a\ vec\ set) \subseteq carrier\text{-}vec\ n\ x \in carrier\text{-}vec\ n$
and *alt1*:$y1 \in W\ x - y1 \in orthogonal\text{-}complement\ W$
and *alt2*:$y2 \in W\ x - y2 \in orthogonal\text{-}complement\ W$
shows $y1 = y2$
proof −
have *carr*:$y1 \in carrier\text{-}vec\ n\ y2 \in carrier\text{-}vec\ n\ x \in carrier\text{-}vec\ n - y1 \in carrier\text{-}vec\ n$
$0_v\ n \in carrier\text{-}vec\ n$
using *alt1 alt2 carr* **by** *auto*
hence $y1 - y2 \in carrier\text{-}vec\ n$ **by** *auto*
note $carr = this\ carr$
from *alt1* **have** $ya{\in}W \Longrightarrow (x - y1) \cdot ya = 0$ **for** *ya*
unfolding *orthogonal-complement-def* **by** *blast*
hence $(x - y1) \cdot y2 = 0\ (x - y1) \cdot y1 = 0$ **using** *alt2 alt1* **by** *auto*
hence *eq1*:$y1 \cdot y2 = x \cdot y2\ y1 \cdot y1 = x \cdot y1$ **using** *carr minus-scalar-prod-distrib* **by** *force+*
from *this*(*1*) **have** *eq2*:$y2 \cdot y1 = x \cdot y2$ **using** *carr comm-scalar-prod* **by** *force*
from *alt2* **have** $ya{\in}W \Longrightarrow (x - y2) \cdot ya = 0$ **for** *ya*
unfolding *orthogonal-complement-def* **by** *blast*
hence $(x - y2) \cdot y1 = 0\ (x - y2) \cdot y2 = 0$ **using** *alt2 alt1* **by** *auto*
hence *eq3*:$y2 \cdot y2 = x \cdot y2\ y2 \cdot y1 = x \cdot y1$ **using** *carr minus-scalar-prod-distrib* **by** *force+*
with *eq2* **have** *eq4*:$x \cdot y1 = x \cdot y2$ **by** *auto*
have $\|(y1 - y2)\|^2 = 0$ **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* $(y1 - y2)$] *carr* **have** $y1 - y2 = 0_v\ n$ **by** *fastforce*
hence $y1 - y2 + y2 = y2$ **using** *carr* **by** *fastforce*
also have $y1 - y2 + y2 = y1$ **using** *carr* **by** *auto*
finally show $y1 = y2$ **.**
qed

definition

is-oc-projection w S v = (*w* ∈ *carrier-vec n* ∧ *v* − *w* ∈ *span S* ∧ (∀ *u*. *u* ∈ *S* ⟶ *w* ⋅ *u* = *0*))

lemma *is-oc-projection-sq-norm*: **assumes** *is-oc-projection w S v*
and *S*: *S* ⊆ *carrier-vec n*
and *v*: *v* ∈ *carrier-vec n*
shows *sq-norm w* ≤ *sq-norm v*
proof −
from *assms*[*unfolded is-oc-projection-def*]
have *w*: *w* ∈ *carrier-vec n*
and *vw*: *v* − *w* ∈ *span S* **and** *ortho*: ⋀ *u*. *u* ∈ *S* ⟹ *w* ⋅ *u* = *0* **by** *auto*
have *sq-norm v* = *sq-norm* ((*v* − *w*) + *w*) **using** *v w*
by (*intro arg-cong*[*of - - sq-norm-vec*], *auto*)
also have ... = ((*v* − *w*) + *w*) ⋅ ((*v* − *w*) + *w*) **unfolding** *sq-norm-vec-as-cscalar-prod*
by *simp*
also have ... = (*v* − *w*) ⋅ ((*v* − *w*) + *w*) + *w* ⋅ ((*v* − *w*) + *w*)
by (*rule add-scalar-prod-distrib*, *insert v w*, *auto*)
also have ... = ((*v* − *w*) ⋅ (*v* − *w*) + (*v* − *w*) ⋅ *w*) + (*w* ⋅ (*v* − *w*) + *w* ⋅ *w*)
by (*subst* (*1 2*) *scalar-prod-add-distrib*, *insert v w*, *auto*)
also have ... = *sq-norm* (*v* − *w*) + *2* ∗ (*w* ⋅ (*v* − *w*)) + *sq-norm w*
unfolding *sq-norm-vec-as-cscalar-prod* **using** *v w* **by** (*auto simp*: *comm-scalar-prod*[*of w - v* − *w*])
also have ... ≥ *2* ∗ (*w* ⋅ (*v* − *w*)) + *sq-norm w* **using** *sq-norm-vec-ge-0*[*of v* − *w*] **by** *auto*
also have *w* ⋅ (*v* − *w*) = *0* **using** *orthocompl-span*[*OF ortho S w vw*] **by** *auto*
finally show *?thesis* **by** *auto*
qed

definition *oc-projection* **where**
oc-projection S fi ≡ (*SOME v*. *is-oc-projection v S fi*)

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*!]:(λ*v*. *aa v* ⋅$_v$ *v*) ∈ *A* → *carrier-vec n* **by** *auto*
from *a*(*1*) **have** *e1*:− *a* = *lincomb* (λ *x*. − *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-iff-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-carrier*(*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*(*2*) *carr-mat*(*1*) *mat-mult-left-right-inverse* **by** *blast*
hence *?A* ∗ *P* ∗$_v$ *v* = *v* **using** *v* **by** *auto*
hence *?A* ∗$_v$ (*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* ∈ *span* (*set G*) **by** (*intro in-spanI*[*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* ⊆ *span* (*set G*) ⟷ *det* (*mat-of-rows n G*) ≠ *0* (**is** *?q1*)
carrier-vec n ⊆ *span* (*set G*) ⟷ *basis* (*set G*) (**is** *?q2*)
det (*mat-of-rows n G*) ≠ *0* ⟷ *basis* (*set G*) (**is** *?q3*)
proof −
have *dc*:*det* (*mat-of-rows n G*) ≠ *0* ⟹ *carrier-vec n* ⊆ *span* (*set G*)
using *assms non-span-det-zero* **by** *auto*
have *cb*:*carrier-vec n* ⊆ *span* (*set G*) ⟹ *basis* (*set G*) **using** *assms basis-list-basis*

```
    by (auto simp: basis-list-def)
  have bd:basis (set G) ⟹ det (mat-of-rows n G) ≠ 0 using assms basis-det-nonzero
by auto
  show ?q1 ?q2 ?q3 using dc cb bd by metis+
qed

lemma lin-indpt-list-nonzero:
  assumes lin-indpt-list G
  shows 0_v n ∉ 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_v
n ∉ set G by auto
qed

lemma is-oc-projection-eq:
  assumes ispr:is-oc-projection a S v is-oc-projection b S v
    and carr: S ⊆ carrier-vec n v ∈ 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) = a using carr ispr by auto
  have b:v − (v − b) = b using carr ispr by auto
  have (v − a) = (v − b)
    apply(rule oc-projection-alt-def[OF c2])
    using ispr a b unfolding in-orthogonal-complement-span[OF carr(1)]
    unfolding orthogonal-complement-def is-oc-projection-def by auto
  hence v − (v − a) = v − (v − b) by metis
  thus ?thesis unfolding a b.
qed



fun adjuster-wit :: 'a list ⇒ 'a vec ⇒ 'a vec list ⇒ 'a list × 'a vec
  where adjuster-wit wits w [] = (wits, 0_v n)
  | adjuster-wit wits w (u#us) = (let a = (w • u)/ sq-norm u in
          case adjuster-wit (a # wits) w us of (wit, v)
        ⇒ (wit, −a ·_v u + v))

fun sub2-wit where
    sub2-wit us [] = ([], [])
  | sub2-wit us (w # ws) =
     (case adjuster-wit [] w us of (wit,aw) ⇒ let u = aw + w in
      case sub2-wit (u # us) ws of (wits, vvs) ⇒ (wit # wits, u # vvs))

definition main :: 'a vec list ⇒ 'a list list × 'a vec list where
  main us = sub2-wit [] us
end
```

locale *gram-schmidt-fs* =
  fixes n :: *nat* **and** *fs* :: $'a$ :: {*trivial-conjugatable-linordered-field*} *vec list*
begin

sublocale *gram-schmidt n TYPE*($'a$) **.**

fun *gso* **and** μ **where**
  gso i = *fs* ! *i* + *sumlist* (*map* (λ *j*. − μ *i j* ·$_v$ *gso j*) [*0* ..< *i*])
| μ *i j* = (*if j* < *i then* (*fs* ! *i* • *gso j*)/ *sq-norm* (*gso j*) *else if i* = *j then 1 else 0*)

declare *gso.simps*[*simp del*]
declare μ*.simps*[*simp del*]

lemma *gso-carrier'*[*intro*]:
  assumes $\bigwedge$ *i*. *i* ≤ *j* ⟹ *fs* ! *i* ∈ *carrier-vec n*
  shows *gso j* ∈ *carrier-vec n*
using *assms* **proof**(*induct j rule*:*nat-less-induct*[*rule-format*])
  case (*1 j*)
 then show *?case* **unfolding** *gso.simps*[*of j*] **by** (*auto intro*!:*sumlist-carrier add-carrier-vec*)
qed

lemma *adjuster-wit*: **assumes** *res*: *adjuster-wit wits w us* = (*wits'*,*a*)
  and *w*: *w* ∈ *carrier-vec n*
    and *us*: $\bigwedge$ *i*. *i* ≤ *j* ⟹ *fs* ! *i* ∈ *carrier-vec n*
    and *us-gs*: *us* = *map gso* (*rev* [*0* ..< *j*])
    and *wits*: *wits* = *map* (μ *i*) [*j* ..< *i*]
    and *j*: *j* ≤ *n j* ≤ *i*
    and *wi*: *w* = *fs* ! *i*
  shows *adjuster n w us* = *a* ∧ *a* ∈ *carrier-vec n* ∧ *wits'* = *map* (μ *i*) [*0* ..< *i*] ∧
      (*a* = *sumlist* (*map* (λ*j*. − μ *i j* ·$_v$ *gso j*) [*0*..<*j*]))
  using *res us us-gs wits j*
proof (*induct us arbitrary*: *wits wits' a j*)
  case (*Cons u us wits wits' a j*)
  note *us-gs* = *Cons*(*4*)
  note *wits* = *Cons*(*5*)
  note *jn* = *Cons*(*6*−*7*)
  from *us-gs* **obtain** *jj* **where** *j*: *j* = *Suc jj* **by** (*cases j*, *auto*)
  from *jn j* **have** *jj*: *jj* ≤ *n jj* < *n jj* ≤ *i jj* < *i* **by** *auto*
  have *zj*: [*0* ..< *j*] = [*0* ..< *jj*] @ [*jj*] **unfolding** *j* **by** *simp*
 have *jjn*: [*jj* ..< *i*] = *jj* # [*j* ..< *i*] **using** *jj* **unfolding** *j* **by** (*metis upt-conv-Cons*)
  from *us-gs*[*unfolded zj*] **have** *ugs*: *u* = *gso jj* **and** *us*: *us* = *map gso* (*rev* [*0*..<*jj*])
by *auto*
  let *?w* = *w* • *u* / (*u* • *u*)
 have *muij*: *?w* = μ *i jj* **unfolding** μ*.simps*[*of i jj*] *ugs wi sq-norm-vec-as-cscalar-prod*
using *jj* **by** *auto*
  have *wwits*: *?w* # *wits* = *map* (μ *i*) [*jj*..<*i*] **unfolding** *jjn wits muij* **by** *simp*

obtain *wwits b* **where** *rec*: *adjuster-wit* (*?w # wits*) *w us* = (*wwits*,*b*) **by** *force*
from *Cons*(*1*)[*OF this Cons*(*3*) *us wwits jj*(*1*,*3*),*unfolded j*] **have** *IH*:
adjuster n w us = *b wwits* = *map* (*μ i*) [*0*..<*i*]
b = *sumlist* (*map* (λ*j*. − *μ i j* ·$_v$ *gso j*) [*0*..<*jj*])
and *b*: *b* ∈ *carrier-vec n* **by** *auto*
from *Cons*(*2*)[*simplified*, *unfolded Let-def rec split sq-norm-vec-as-cscalar-prod cscalar-prod-is-scalar-prod*]
have *id*: *wits′* = *wwits* **and** *a*: *a* = − *?w* ·$_v$ *u* + *b* **by** *auto*
have *1*: *adjuster n w* (*u # us*) = *a* **unfolding** *a IH*(*1*)[*symmetric*] **by** *auto*
from *id IH*(*2*) **have** *wits′*: *wits′* = *map* (*μ i*) [*0*..<*i*] **by** *simp*
have *carr*:*set* (*map* (λ*j*. − *μ i j* ·$_v$ *gso j*) [*0*..<*j*]) ⊆ *carrier-vec n*
set (*map* (λ*j*. − *μ i j* ·$_v$ *gso j*) [*0*..<*jj*]) ⊆ *carrier-vec n* **and** *u*:*u* ∈ *carrier-vec n*
using *Cons j* **by** (*auto intro*!:*gso-carrier′*)
from *u b a* **have** *ac*: *a* ∈ *carrier-vec n dim-vec* (−*?w* ·$_v$ *u*) = *n dim-vec b* = *n dim-vec u* = *n* **by** *auto*
show *?case*
apply (*intro conjI*[*OF 1*] *ac exI conjI wits′*)
unfolding *carr a IH zj muij ugs*[*symmetric*] *map-append*
apply (*subst sumlist-append*)
using *Cons.prems j* **apply** *force*
using *b u ugs IH*(*3*) **by** *auto*
qed *auto*

lemma *sub2-wit*:
assumes *set us* ⊆ *carrier-vec n set ws* ⊆ *carrier-vec n length us* + *length ws* = *m*
and *ws* = *map* (λ *i*. *fs* ! *i*) [*i* ..< *m*]
and *us* = *map gso* (*rev* [*0* ..< *i*])
and *us*: ⋀ *j*. *j* < *m* ⟹ *fs* ! *j* ∈ *carrier-vec n*
and *mn*: *m* ≤ *n*
shows *sub2-wit us ws* = (*wits*,*vvs*) ⟹ *gram-schmidt-sub2 n us ws* = *vvs*
∧ *vvs* = *map gso* [*i* ..< *m*] ∧ *wits* = *map* (λ *i*. *map* (*μ i*) [*0*..<*i*]) [*i* ..< *m*]
using *assms*(*1*−*6*)
proof (*induct ws arbitrary*: *us vvs i wits*)
case (*Cons w ws us vs*)
note *us* = *Cons*(*3*) **note** *wws* = *Cons*(*4*)
note *wsf′* = *Cons*(*6*)
note *us-gs* = *Cons*(*7*)
from *wsf′* **have** *i* < *m i* ≤ *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* ≤ *n i* ≤ *m* **by** *auto*
hence *i-n*: [*i* ..< *n*] = *i* # [*Suc i* ..< *n*] **by** (*metis upt-conv-Cons*)
from *wsf′ i-m* **have** *wsf*: *ws* = *map* (λ *i*. *fs* ! *i*) [*Suc i* ..< *m*]
and *fiw*: *fs* ! *i* = *w* **by** *auto*
from *wws* **have** *w*: *w* ∈ *carrier-vec n* **and** *ws*: *set ws* ⊆ *carrier-vec n* **by** *auto*
have *list*: *map* (*μ i*) [*i* ..< *i*] = [] **by** *auto*
let *?a* = *adjuster-wit* [] *w us*
obtain *wit a* **where** *a*: *?a* = (*wit*,*a*) **by** *force*

obtain *wits' vv* **where** *gs*: *sub2-wit* ((*a* + *w*) # *us*) *ws* = (*wits'*,*vv*) **by** *force*
from *adjuster-wit*[*OF a w Cons*(*8*) *us-gs list*[*symmetric*] ‹*i* ≤ *n*› - *fiw*[*symmetric*]]
us wws ‹*i* < *m*›
have *awus*: *set* ((*a* + *w*) # *us*) ⊆ *carrier-vec n*
and *aa*: *adjuster n w us* = *a a* ∈ *carrier-vec n*
and *aaa*: *a* = *sumlist* (*map* (λ*j*. − *μ i j* ·$_v$ *gso j*) [*0*..<*i*])
and *wit*: *wit* = *map* (*μ i*) [*0*..<*i*]
by *auto*
have *aw-gs*: *a* + *w* = *gso i* **unfolding** *gso.simps*[*of i*] *fiw aaa*[*symmetric*] **using**
aa(*2*) *w* **by** *auto*
with *us-gs* **have** *us-gs'*: (*a* + *w*) # *us* = *map gso* (*rev* [*0*..<*Suc i*]) **by** *auto*
from *Cons*(*1*)[*OF gs awus ws - wsf us-gs' Cons*(*8*)] *Cons*(*5*)
have *IH*: *gram-schmidt-sub2 n* ((*a* + *w*) # *us*) *ws* = *vv*
and *vv*: *vv* = *map gso* [*Suc i*..<*m*]
and *wits'*: *wits'* = *map* (λ*i*. *map* (*μ i*) [*0*..<*i*]) [*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 *Cons*(*2*)[*unfolded sub2-wit.simps a split Let-def gs*] **have** *wits*: *wits* = *wit*
wits' **by** *auto*
from *vs vv aw-gs* **have** *vs*: *vs* = *map gso* [*i* ..< *m*] **unfolding** *i-m* **by** *auto*
with *gs-vs* **show** *?case* **unfolding** *wits wit wits'* **by** (*auto simp*: *i-m*)
qed *auto*

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* ((!) *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*
note *assms*(*5*)[*unfolded main-def*]
have *gram-schmidt-sub2 n* [] (*take k fs*) = *vvs* ∧ *vvs* = *map gso* [*0*..<*k*] ∧ *wits*
= *map* (λ*i*. *map* (*μ i*) [*0*..<*i*]) [*0*..<*k*]
if *vvs* = *snd* (*sub2-wit* [] (*take k fs*)) *wits* = *fst* (*sub2-wit* [] (*take k fs*)) **for** *vvs*
wits
using *assms that* **by** (*intro sub2-wit*) (*auto*)
with *assms main-def*
show *gram-schmidt n us* = *vs vs* = *map gso* [*0*..<*k*] **unfolding** *gram-schmidt-code*
by (*auto simp add*: *main-def case-prod-beta'*)
qed

lemma *adjuster-wit-small*:
(*adjuster-wit v a xs*) = (*x1*,*x2*)
⟷ (*fst* (*adjuster-wit v a 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 *sub2*: *rev xs* @ *snd* (*sub2-wit xs us*) = *rev* (*gram-schmidt-sub n xs us*)
proof −
have *sub2-wit xs us* = (*x1*, *x2*) ⟹ *rev xs* @ *x2* = *rev* (*gram-schmidt-sub n xs us*)
for *x1 x2 xs us*
apply(*induct us arbitrary*: *xs x1 x2*)
by (*auto simp*:*Let-def rev-unsimp adjuster-wit-small split*:*prod.splits simp del*:*rev.simps*)
thus *?thesis*
apply (*cases us*)
by (*auto simp*:*Let-def rev-unsimp adjuster-wit-small split*:*prod.splits simp del*:*rev.simps*)
qed

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

definition *weakly-reduced* :: *'a* ⇒ *nat* ⇒ *bool*

where *weakly-reduced* α *k* = (∀ *i*. *Suc i* < *k* ⟶
sq-norm (*gso i*) ≤ α ∗ *sq-norm* (*gso* (*Suc i*)))

definition *reduced* :: *'a* ⇒ *nat* ⇒ *bool*

where *reduced* α *k* = (*weakly-reduced* α *k* ∧
(∀ *i j*. *i* < *k* ⟶ *j* < *i* ⟶ *abs* (μ *i j*) ≤ *1*/*2*))

end

locale *gram-schmidt-fs-Rn* = *gram-schmidt-fs* +
assumes *fs-carrier*: *set fs* ⊆ *carrier-vec n*
begin

abbreviation (*input*) *m* **where** *m* ≡ *length fs*

definition *M* **where** *M k* = *mat k k* (λ (*i*,*j*). μ *i j*)

lemma *f-carrier*[*simp*]: *i* < *m* ⟹ *fs* ! *i* ∈ *carrier-vec n*
using *fs-carrier* **unfolding** *set-conv-nth* **by** *force*

lemma *gso-carrier*[*simp*]: *i* < *m* ⟹ *gso i* ∈ *carrier-vec n*
using *gso-carrier'* *f-carrier* **by** *auto*

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

lemma *fs0-gso0*: $0 < m \Longrightarrow$ *fs* ! 0 = *gso* 0
unfolding *gso.simps*[*of* 0] **using** *f-dim*[*of* 0]
by (*cases fs*, *auto simp add*: *upt-rec*)

lemma *fs-by-gso-def* :
assumes *i*: $i < m$
shows *fs* ! *i* = *gso i* + *M.sumlist* (*map* ($\lambda ja.\ \mu\ i\ ja \cdot_v$ *gso ja*) [0..<*i*]) (**is** - = - + *?sum*)
proof −
{
fix *f*
have *a*: *M.sumlist* (*map* ($\lambda ja.$ *f ja* $\cdot_v$ *gso ja*) [0..<*i*]) ∈ *carrier-vec n*
using *gso-carrier i* **by** (*intro M.sumlist-carrier*, *auto*)
hence *dim-vec* (*M.sumlist* (*map* ($\lambda ja.$ *f ja* $\cdot_v$ *gso ja*) [0..<*i*])) = *n* **by** *auto*
note *a this*
} note *sum-carrier* = *this*
note [*simp*] = *sum-carrier*(2)
have *f*: *fs* ! *i* ∈ *carrier-vec n* **using** *i* **by** *simp*
have *gso i* + *?sum* = *fs* ! *i* + *M.sumlist* (*map* ($\lambda j.$ − μ *i j* $\cdot_v$ *gso j*) [0..<*i*]) + *?sum*
(**is** - = - + *?minus-sum* + -)
unfolding *gso.simps*[*of i*] **by** *simp*
also have *?minus-sum* = − *?sum*
using *gso-carrier i sum-carrier*
by (*intro eq-vecI*, *auto simp*: *sumlist-nth sum-negf*)
also have *fs* ! *i* + (−*?sum*) + *?sum* = *fs* ! *i*
using *sum-carrier fs-carrier f* **by** *simp*
finally show *?thesis* **by** *auto*
qed

lemma *main-connect*:
assumes $m \leq n$
shows *gram-schmidt n fs* = *map gso* [0..<*m*]
proof −
obtain *vs* **where** *snd-main*: *snd* (*main fs*) = *vs* **by** *auto*
have *gram-schmidt-sub2 n* [] *fs* = *snd* (*sub2-wit* [] *fs*) ∧ *snd* (*sub2-wit* [] *fs*) = *map gso* [0..<*length fs*]
∧ *wits* = *map* ($\lambda i.$ *map* (μ *i*) [0..<*i*]) [0..<*length fs*]
if *wits* = *fst* (*sub2-wit* [] *fs*) **for** *wits*
using *assms that fs-carrier* **by** (*intro sub2-wit*) (*auto simp add*: *map-nth*)
then have *gram-schmidt-sub2 n* [] *fs* = *vs* ∧ *vs* = *map gso* [0..<*m*]
using *snd-main main-def* **by** *auto*
thus *gram-schmidt n fs* = *map gso* [0..<*m*] **by** (*auto simp*: *gram-schmidt-code*)
qed

lemma *reduced-gso-E*: *weakly-reduced* α $k \Longrightarrow k \leq m \Longrightarrow Suc\ i < k \Longrightarrow$
sq-norm (*gso i*) $\leq \alpha *$ *sq-norm* (*gso* (*Suc i*))
unfolding *weakly-reduced-def* **by** *auto*

abbreviation (*input*) *FF* **where** *FF* $\equiv$ *mat-of-rows n fs*
abbreviation (*input*) *Fs* **where** *Fs* $\equiv$ *mat-of-rows n* (*map gso* [*0*..<*m*])

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*)

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*: *main-connect*)

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

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

lemma *M-index*[*simp*]:*i* < *m* $\Longrightarrow$ *j* < *m* $\Longrightarrow$ (*M m*) $$ (*i*,*j*) = μ *i j*
unfolding *M-def* **by** *auto*

lemma *matrix-equality*: *FF* = (*M m*) $*$ *Fs*
proof −
let *?P* = (*M m*) $*$ *Fs*
have *dim*: *dim-row FF* = *m dim-col FF* = *n dim-row ?P* = *m dim-col ?P* = *n dim-row* (*M m*) = *m dim-col* (*M m*) = *m*
dim-row Fs = *m dim-col Fs* = *n*
by (*auto simp*: *mat-of-rows-def mat-of-rows-list-def main-connect*)
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* = λ *k*. μ *i k* $*$ *gso k* $ *j*
have *dim2*: *dim-vec* (*col Fs j*) = *m* **using** *j dim* **by** *auto*
define *idx* **where** *idx* = [*0*..<*i*]
have *idx*: *set idx* $\subseteq$ {*0* ..< *i*} **unfolding** *idx-def* **using** *i* **by** *auto*
let *?vec* = *sumlist* (*map* (λ*j*. − μ *i j* $\cdot_v$ *gso j*) *idx*)
have *vec*: *?vec* $\in$ *carrier-vec n* **by** (*rule sumlist-carrier*, *insert idx gso-carrier i*, *auto*)
hence *dimv*: *dim-vec ?vec* = *n* **by** *auto*
have *?P* $$ (*i*,*j*) = *row* (*M m*) *i* $\cdot$ *col Fs j* **using** *dim i j* **by** *auto*

```
    also have ... = (∑ k = 0..<m. row (M m) i $ k * col Fs j $ k)
      unfolding scalar-prod-def dim2 by auto
    also have ... = (∑ k = 0..<m. ?prod k)
       by (rule sum.cong[OF refl], insert i j dim, auto simp: mat-of-rows-list-def
mat-of-rows-def)
    also have ... = sum-list (map ?prod [0 ..< m])
      by (subst sum-list-distinct-conv-sum-set, auto)
     also have ... = sum-list (map ?prod idx) + ?prod i + sum-list (map ?prod
[Suc i ..< m])
      unfolding split idx-def by auto
    also have ?prod i = gso i $ j unfolding μ.simps by simp
     also have ... = fs ! i $ j + sum-list (map (λk. − μ i k * gso k $ j) 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 idx gso-carrier i j,
      auto simp: o-def intro!: arg-cong[OF map-cong])
    also have sum-list (map (λk. − μ i k * gso k $ j) idx) = − sum-list (map (λk.
μ i k * gso k $ j) idx)
      by (induct idx, auto)
    also have sum-list (map ?prod [Suc i ..< m]) = 0
      by (rule sum-list-neutral, auto simp: μ.simps)
    finally have ?P $$ (i,j) = fs ! i $ j by simp
    with FF-index[OF i j]
    show FF $$ (i,j) = ?P $$ (i,j) by simp
  qed auto
qed

lemma fi-is-sum-of-mu-gso: assumes i: i < m
  shows fs ! i = sumlist (map (λ j. μ i j ·v gso j) [0 ..< Suc i])
proof −
  let ?l = sumlist (map (λ j. μ i j ·v gso j) [0 ..< Suc i])
  have ?l ∈ carrier-vec n by (rule sumlist-carrier, insert gso-carrier i, auto)
  hence dim: dim-vec ?l = n by (rule carrier-vecD)
  show ?thesis
  proof (rule eq-vecI, unfold dim f-dim[OF i])
    fix j
    assume j: j < n
    from i have split: [0 ..< m] = [0 ..< Suc i] @ [Suc i ..< m]
       by (metis Suc-lessI append.assoc append-same-eq less-imp-add-positive or-
der-refl upt-add-eq-append zero-le)
    let ?prod = λ k. μ i k * gso k $ j
    have fs ! i $ j = FF $$ (i,j) using i j by simp
    also have ... = ((M m) * Fs) $$ (i,j) using matrix-equality by simp
    also have ... = row (M m) i · col Fs j using i j by auto
    also have ... = (∑ k = 0..<m. row (M m) i $ k * col Fs j $ k)
      unfolding scalar-prod-def by auto
    also have ... = (∑ k = 0..<m. ?prod k)
       by (rule sum.cong[OF refl], insert i j dim, auto simp: mat-of-rows-list-def
mat-of-rows-def)
```

```
    also have ... = sum-list (map ?prod [0 ..< m])
      by (subst sum-list-distinct-conv-sum-set, auto)
    also have ... = sum-list (map ?prod [0 ..< Suc i]) + sum-list (map ?prod [Suc
i ..< m])
      unfolding split by auto
    also have sum-list (map ?prod [Suc i ..< m]) = 0
      by (rule sum-list-neutral, auto simp: μ.simps)
    also have sum-list (map ?prod [0 ..< Suc i]) = ?l $ j
     by (subst sumlist-vec-index[OF - j], (insert i, auto simp: intro!: gso-carrier)[1],

         rule arg-cong[of - - sum-list], insert i j, auto)
    finally show fs ! i $ j = ?l $ j by simp
  qed simp
qed

lemma gi-is-fi-minus-sum-mu-gso:
  assumes i: i < m
  shows gso i = fs ! i − sumlist (map (λ j. μ i j ·v gso j) [0 ..< i]) (is - = - −
?sum)
proof −
  have sum: ?sum ∈ carrier-vec n
    by (rule sumlist-carrier, insert gso-carrier i, auto)
  show ?thesis unfolding fs-by-gso-def[OF i]
    by (intro eq-vecI, insert gso-carrier[OF i] sum, auto)
qed


lemma det: assumes m: m = n shows det FF = det Fs
  unfolding matrix-equality
  apply (subst det-mult[OF M-dim(3)], (insert Fs-dim(3) m, auto)[1])
  apply (subst det-lower-triangular[OF - M-dim(3)])
  by (subst M-index, (auto simp: μ.simps)[3], unfold prod-list-diag-prod, auto simp:
μ.simps)
end

locale gram-schmidt-fs-lin-indpt = gram-schmidt-fs-Rn +
  assumes lin-indpt: lin-indpt (set fs) and dist: distinct fs
begin

lemmas loc-assms = lin-indpt dist

lemma mn:
  shows m ≤ n
proof −
  have n: n = dim by (simp add: dim-is-n)
  have m: m = card (set fs)
    using distinct-card loc-assms by metis
  from m n have mn: m ≤ n ⟷ card (set fs) ≤ dim by simp
  show ?thesis unfolding mn
```

by (*rule li-le-dim*, *use loc-assms fs-carrier* **in** *auto*)
qed

lemma
shows *span-gso*: *span* (*gso* ‘ {*0*..<*m*}) = *span* (*set fs*)
and *orthogonal-gso*: *orthogonal* (*map gso* [*0*..<*m*])
and *dist-gso*: *distinct* (*map gso* [*0*..<*m*])
using *gram-schmidt-result*[*OF fs-carrier - - main-connect*[*symmetric*]] *loc-assms mn* **by** *auto*

lemma *gso-inj*[*intro*]:
assumes *i* < *m*
shows *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* (*map gso* [*0*..<*m*]) *x* < *length* (*map gso* [*0*..<*m*]) *y* < *length* (*map gso* [*0*..<*m*])
using *dist-gso assms mn assms′* **by** (*auto intro*!: *dist-gso*)
from *nth-eq-iff-index-eq*[*OF this*] *assms′* **have** *x* = *y* **by** *auto* **}**
then show *?thesis*
using *assms* **by** (*intro inj-onI*) *auto*
qed

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** *i* **by** *auto*
from *fs-carrier 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 - i mn us main refl fs-carrier*] *assms*
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 loc-assms*, *auto*)

lemma *orthogonal*:
assumes $i < m\ j < m\ i \neq j$
shows $gso\ i \cdot gso\ j = 0$
using *assms mn orthogonal-gso*[*unfolded orthogonal-def*] **by** *auto*

lemma *same-base*:
shows $span\ (set\ fs) = span\ (gso\ `\ \{0..<m\})$
using *span-gso loc-assms* **by** *simp*

lemma *sq-norm-gso-le-f*:
assumes *i*: $i < m$
shows $sq\text{-}norm\ (gso\ i) \leq sq\text{-}norm\ (fs\ !\ i)$
proof −
have *id*: $[0\ ..<\ Suc\ i] = [0\ ..<\ i]\ @\ [i]$ **by** *simp*
let *?sum* = $sumlist\ (map\ (\lambda j.\ \mu\ i\ j \cdot_v gso\ j)\ [0..<i])$
have *sum*: $?sum \in carrier\text{-}vec\ n$ **and** *gsoi*: $gso\ i \in carrier\text{-}vec\ n$ **using** *i*
by (*auto intro*!: *sumlist-carrier gso-carrier*)
from *fi-is-sum-of-mu-gso*[*OF i*, *unfolded id*]
have $sq\text{-}norm\ (fs\ !\ i) = sq\text{-}norm\ (sumlist\ (map\ (\lambda j.\ \mu\ i\ j \cdot_v gso\ j)\ [0..<i]\ @\ [gso\ i]))$ **by** (*simp add*: *μ.simps*)
also have $\ldots = sq\text{-}norm\ (?sum + gso\ i)$
by (*subst sumlist-append*, *insert gso-carrier i*, *auto*)
also have $\ldots = (?sum + gso\ i) \cdot (?sum + gso\ i)$ **by** (*simp add*: *sq-norm-vec-as-cscalar-prod*)
also have $\ldots = ?sum \cdot (?sum + gso\ i) + gso\ i \cdot (?sum + gso\ i)$
by (*rule add-scalar-prod-distrib*[*OF sum gsoi*], *insert sum gsoi*, *auto*)
also have $\ldots = (?sum \cdot ?sum + ?sum \cdot gso\ i) + (gso\ i \cdot ?sum + gso\ i \cdot gso\ i)$
by (*subst* (*1 2*) *scalar-prod-add-distrib*[*of - n*], *insert sum gsoi*, *auto*)
also have $?sum \cdot ?sum = sq\text{-}norm\ ?sum$ **by** (*simp add*: *sq-norm-vec-as-cscalar-prod*)
also have $gso\ i \cdot gso\ i = sq\text{-}norm\ (gso\ i)$ **by** (*simp add*: *sq-norm-vec-as-cscalar-prod*)
also have $gso\ i \cdot ?sum = ?sum \cdot gso\ i$ **using** *gsoi sum* **by** (*simp add*: *comm-scalar-prod*)
finally have $sq\text{-}norm\ (fs\ !\ i) = sq\text{-}norm\ ?sum + 2 * (?sum \cdot gso\ i) + sq\text{-}norm\ (gso\ i)$ **by** *simp*
also have $\ldots \geq 2 * (?sum \cdot gso\ i) + sq\text{-}norm\ (gso\ i)$ **using** *sq-norm-vec-ge-0*[*of ?sum*] **by** *simp*
also have $?sum \cdot gso\ i = (\sum v \leftarrow map\ (\lambda j.\ \mu\ i\ j \cdot_v gso\ j)\ [0..<i].\ v \cdot gso\ i)$
by (*subst scalar-prod-left-sum-distrib*[*OF - gsoi*], *insert i gso-carrier*, *auto*)
also have $\ldots = 0$
proof (*rule sum-list-neutral*, *goal-cases*)
case (*1 x*)
then obtain *j* **where** *j*: $j < i$ **and** *x*: $x = (\mu\ i\ j \cdot_v gso\ j) \cdot gso\ i$ **by** *auto*
from *j i* **have** *gsoj*: $gso\ j \in carrier\text{-}vec\ n$ **by** *auto*
have $x = \mu\ i\ j * (gso\ j \cdot gso\ i)$ **using** *gsoi gsoj* **unfolding** *x* **by** *simp*
also have $gso\ j \cdot gso\ i = 0$
by (*rule orthogonal*, *insert j i assms*, *auto*)
finally show $x = 0$ **by** *simp*
qed

```
  finally show ?thesis by simp
qed


lemma oc-projection-exist:
  assumes i: i < m
  shows fs ! i − gso i ∈ span (gso ‘ {0..<i})
proof
  let ?A = gso ‘ {0..<i}
  show finA:finite ?A by auto
  have carA[intro!]:?A ⊆ carrier-vec n using gso-dim assms by auto
  let ?a v = ∑ n←[0..<i]. if v = gso n then μ i n else 0
  have d:(sumlist (map (λj. − μ i j ·v gso j) [0..<i])) ∈ carrier-vec n
    using gso.simps[of i] gso-dim[OF i] unfolding carrier-vec-def by auto
  note [intro] = f-carrier[OF i] gso-carrier[OF i] d
  have [intro!]:(λv. ?a v ·v v) ∈ gso ‘ {0..<i} → carrier-vec n
     using gso-carrier assms by auto
  {fix ia assume ia[intro]:ia < n
    have (∑ x∈gso ‘ {0..<i}. (?a x ·v x) $ ia) =
      − (∑ x←map (λj. − μ i j ·v gso j) [0..<i]. x $ ia)
   unfolding map-map comm-monoid-add-class.sum.reindex[OF gso-inj[OF assms]]
   unfolding atLeastLessThan-upt sum-set-upt-conv-sum-list-nat uminus-sum-list-map
o-def
    proof(rule arg-cong[OF map-cong, OF refl],goal-cases)
      case (1 x) hence x:x < m x < i using assms by auto
      hence d:insert x (set [0..<i]) = {0..<i}
            count (mset [0..<i]) x = 1 by auto
      hence inj-on gso (insert x (set [0..<i])) using gso-inj[OF assms] by auto
      from inj-on-filter-key-eq[OF this,folded replicate-count-mset-eq-filter-eq]
      have [n←[0..<i] . gso x = gso n] = [x] using x assms d replicate.simps(2)[of
0] by auto
      hence (∑ n←[0..<i]. if gso x = gso n then μ i n else 0) = μ i x
        unfolding sum-list-map-filter'[symmetric] by auto
      with ia gso-dim x show ?case apply(subst index-smult-vec) by force+
    qed
    hence (⨁V v∈gso ‘ {0..<i}. ?a v ·v v) $ ia =
        (− local.sumlist (map (λj. − μ i j ·v gso j) [0..<i])) $ ia
      using d assms
      apply (subst (0 0) finsum-index index-uminus-vec) apply force+
      apply (subst sumlist-vec-index) by force+
  }
  hence id: (⨁V v∈?A. ?a v ·v v) = − sumlist (map (λj. − μ i j ·v gso j) [0..<i])
    using d lincomb-dim[OF finA carA,unfolded lincomb-def] by(intro eq-vecI,auto)
  show fs ! i − gso i = lincomb ?a ?A unfolding lincomb-def gso.simps[of i] id
    by (rule eq-vecI, auto)
qed auto

lemma oc-projection-unique:
  assumes i < m
```

```
        v ∈ carrier-vec n
        ⋀ x. x ∈ gso ‘ {0..<i} ⟹ v • x = 0
        fs ! i − v ∈ span (gso ‘ {0..<i})
  shows v = gso i
proof −
  from assms have carr-span:span (gso ‘ {0..<i}) ⊆ carrier-vec n by(intro span-is-subset2)
auto
  from assms have carr: gso ‘ {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 - carr] assms
  have y ∈ span (gso ‘ {0..<i}) ⟹ v • y = 0 for y by auto
  hence oc1:fs ! i − (fs ! i − v) ∈ orthogonal-complement (span (gso ‘ {0..< i}))
    unfolding eq orthogonal-complement-def using assms by auto
  have x ∈ gso ‘ {0..<i} ⟹ gso i • x = 0 for x using assms orthogonal by auto
  hence y ∈ span (gso ‘ {0..<i}) ⟹ gso i • y = 0 for y
    by (rule orthocompl-span) (use carr gso-carrier assms in auto)
  hence oc2:fs ! i − (fs ! i − gso i) ∈ orthogonal-complement (span (gso ‘ {0..<
i}))
    unfolding eq orthogonal-complement-def using assms by auto
  note pe= oc-projection-exist[OF assms(1)]
 note prerec = carr-span f-carrier[OF assms(1)] assms(4) oc1 oc-projection-exist[OF
assms(1)] oc2
 note prerec = carr-span f-carrier[OF assms(1)] assms(4) oc1 oc-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 oc-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 = gso i
  proof (intro eq-vecI)
    fix j
    show j < dim-vec (gso i) ⟹ v $ j = gso i $ j
      using assms gsoi main[of j] by (auto)
  qed (insert assms gsoi, auto)
qed

lemma gso-oc-projection:
  assumes i < m
  shows gso i = oc-projection (gso ‘ {0..<i}) (fs ! i)
  unfolding oc-projection-def is-oc-projection-def
proof (rule some-equality[symmetric,OF - oc-projection-unique[OF assms]])
  have orthogonal:⋀ xa. xa < i ⟹ gso i • gso xa = 0 by (rule orthogonal,insert
assms, auto)
  show gso i ∈ carrier-vec n ∧
       fs ! i − gso i ∈ span (gso ‘ {0..<i}) ∧
       (∀ x. x ∈ gso ‘ {0..<i} ⟶ gso i • x = 0)
    using gso-carrier oc-projection-exist assms orthogonal by auto
qed auto
```

lemma *gso-oc-projection-span*:
assumes $i < m$
shows *gso i* = *oc-projection* (*span* (*gso* ‘ {*0*..<*i*})) (*fs* ! *i*)
and *is-oc-projection* (*gso i*) (*span* (*gso* ‘ {*0*..<*i*})) (*fs* ! *i*)
unfolding *oc-projection-def is-oc-projection-def*
proof (*rule some-equality*[*symmetric*,*OF - oc-projection-unique*[*OF assms*]])
let *?P* = λ*v*. *v* ∈ *carrier-vec n* ∧ *fs* ! *i* − *v* ∈ *span* (*span* (*gso* ‘ {*0*..<*i*}))
∧ (∀ *x*. *x* ∈ *span* (*gso* ‘ {*0*..<*i*}) ⟶ *v* · *x* = *0*)
have *carr*:*gso* ‘ {*0*..<*i*} ⊆ *carrier-vec n* **using** *assms* **by** *auto*
have ∗: ⋀ *xa*. *xa* < *i* ⟹ *gso i* · *gso xa* = *0* **by** (*rule orthogonal*,*insert assms*,
auto)
have *orthogonal*:⋀*x*. *x* ∈ *span* (*gso* ‘ {*0*..<*i*}) ⟹ *gso i* · *x* = *0*
apply(*rule orthocompl-span*) **using** *assms* ∗ **by** *auto*
show *?P* (*gso i*) *?P* (*gso i*) **unfolding** *span-span*[*OF carr*]
using *gso-carrier oc-projection-exist assms orthogonal* **by** *auto*
fix *v* **assume** *p*:*?P v*
then show *v* ∈ *carrier-vec n* **by** *auto*
from *p* **show** *fs* ! *i* − *v* ∈ *span* (*gso* ‘ {*0*..<*i*}) **unfolding** *span-span*[*OF carr*]
by *auto*
fix *xa* **assume** *xa* ∈ *gso* ‘ {*0*..<*i*}
hence *xa* ∈ *span* (*gso* ‘ {*0*..<*i*}) **using** *in-own-span*[*OF carr*] **by** *auto*
thus *v* · *xa* = *0* **using** *p* **by** *auto*
qed

lemma *gso-is-oc-projection*:
assumes $i < m$
shows *is-oc-projection* (*gso i*) (*set* (*take i fs*)) (*fs* ! *i*)
proof −
have [*simp*]: *v* ∈ *carrier-vec n* **if** *v* ∈ *set* (*take i fs*) **for** *v*
using *that* **by** (*meson contra-subsetD fs-carrier in-set-takeD*)
have *span* (*gso* ‘ {*0*..<*i*}) = *span* (*set* (*take i fs*))
by (*rule partial-span*) (*auto simp add*: *assms less-or-eq-imp-le*)
moreover have *is-oc-projection* (*gso i*) (*span* (*gso* ‘ {*0*..<*i*})) (*fs* ! *i*)
by (*rule gso-oc-projection-span*) (*auto simp add*: *assms less-or-eq-imp-le*)
ultimately have *is-oc-projection* (*gso i*) (*span* (*set* (*take i fs*))) (*fs* ! *i*)
by *auto*
moreover have *set* (*take i fs*) ⊆ *span* (*set* (*take i fs*))
by (*auto intro*!: *span-mem*)
ultimately show *?thesis*
unfolding *is-oc-projection-def* **by** (*subst* (*asm*) *span-span*) (*auto*)
qed

lemma *fi-scalar-prod-gso*:
assumes *i*: $i < m$ **and** *j*: $j < m$
shows *fs* ! *i* · *gso j* = *μ i j* ∗ ‖*gso j*‖^2
proof −
let *?mu* = λ*j*. *μ i j* ·$_v$ *gso j*
from *i* **have** *list1*: [*0*..< *m*] = [*0*..< *Suc i*] @ [*Suc i* ..< *m*]

by (*intro nth-equalityI*, *auto simp*: *nth-append*, *rename-tac j*, *case-tac j − i*, *auto*)
from *j* **have** *list2*: $[0..< m] = [0..< j] @ [j] @ [Suc\ j ..< m]$
by (*intro nth-equalityI*, *auto simp*: *nth-append*, *rename-tac k*, *case-tac k − j*, *auto*)
have $fs\ !\ i \cdot gso\ j = sumlist\ (map\ ?mu\ [0..<Suc\ i]) \cdot gso\ j$
unfolding *fi-is-sum-of-mu-gso*[*OF i*] **by** *simp*
also have $\ldots = (\sum v \leftarrow map\ ?mu\ [0..<Suc\ i].\ v \cdot gso\ j) + 0$
by (*subst scalar-prod-left-sum-distrib*, *insert gso-carrier assms*, *auto*)
also have $\ldots = (\sum v \leftarrow map\ ?mu\ [0..<Suc\ i].\ v \cdot gso\ j) + (\sum v \leftarrow map\ ?mu\ [Suc\ i..<m].\ v \cdot gso\ j)$
by (*subst* (*3*) *sum-list-neutral*, *insert assms gso-carrier*, *auto intro*!: *orthogonal simp*: *μ.simps*)
also have $\ldots = (\sum v \leftarrow map\ ?mu\ [0..< m].\ v \cdot gso\ j)$
unfolding *list1* **by** *simp*
also have $\ldots = (\sum v \leftarrow map\ ?mu\ [0..< j].\ v \cdot gso\ j) + ?mu\ j \cdot gso\ j + (\sum v \leftarrow map\ ?mu\ [Suc\ j..< m].\ v \cdot gso\ j)$
unfolding *list2* **by** *simp*
also have $(\sum v \leftarrow map\ ?mu\ [0..< j].\ v \cdot gso\ j) = 0$
by (*rule sum-list-neutral*, *insert assms gso-carrier*, *auto intro*!: *orthogonal*)
also have $(\sum v \leftarrow map\ ?mu\ [Suc\ j..< m].\ v \cdot gso\ j) = 0$
by (*rule sum-list-neutral*, *insert assms gso-carrier*, *auto intro*!: *orthogonal*)
also have $?mu\ j \cdot gso\ j = \mu\ i\ j * sq\text{-}norm\ (gso\ j)$
using *gso-carrier*[*OF j*] **by** (*simp add*: *sq-norm-vec-as-cscalar-prod*)
finally show *?thesis* **by** *simp*
qed

lemma *gso-scalar-zero*:
assumes $k < m\ i < k$
shows $(gso\ k) \cdot (fs\ !\ i) = 0$
by (*subst comm-scalar-prod*[*OF gso-carrier*]; (*subst fi-scalar-prod-gso*)*?*, *insert assms*, *auto simp*: *μ.simps*)

lemma *scalar-prod-lincomb-gso*:
assumes *k*: $k \leq m$
shows $sumlist\ (map\ (\lambda\ i.\ g\ i \cdot_v gso\ i)\ [0\ ..< k]) \cdot sumlist\ (map\ (\lambda\ i.\ h\ i \cdot_v gso\ i)\ [0\ ..< k])$
$= sum\text{-}list\ (map\ (\lambda\ i.\ g\ i * h\ i * (gso\ i \cdot gso\ i))\ [0\ ..< k])$
proof −
have *id1*: $map\ (\lambda i.\ g\ i \cdot_v map\ (gso)\ [0..<m]\ !\ i)\ [0..<k] = map\ (\lambda i.\ g\ i \cdot_v gso\ i)\ [0..<k]$ **for** *g* **using** *k*
by *auto*
have *id2*: $(\sum i \leftarrow [0..<k].\ g\ i * h\ i * (map\ (gso)\ [0..<m]\ !\ i \cdot map\ (gso)\ [0..<m]\ !\ i))$
$= (\sum i \leftarrow [0..<k].\ g\ i * h\ i * (gso\ i \cdot gso\ i))$ **using** *k*
by (*intro arg-cong*[*OF map-cong*], *auto*)
define *gs* **where** $gs = map\ (gso)\ [0..<m]$
have *gs-gso*: $gs\ !\ i = gso\ i$ **if** $i < k$ **for** *i*
using *that assms* **unfolding** *gs-def* **by** *auto*

have *M.sumlist* (*map* (λ*i*. *g i* ·$_v$ *gs* ! *i*) [*0*..<*k*]) · *M.sumlist* (*map* (λ*i*. *h i* ·$_v$ *gs* ! *i*) [*0*..<*k*]) =
(∑ *i*←[*0*..<*k*]. *g i* ∗ *h i* ∗ (*gs* ! *i* · *gs* ! *i*))
unfolding *gs-def* **using** *assms orthogonal-gso*
by (*intro scalar-prod-lincomb-orthogonal*) *auto*
also have *map* (λ*i*. *g i* ·$_v$ *gs* ! *i*) [*0*..<*k*] = *map* (λ*i*. *g i* ·$_v$ *gso i*) [*0*..<*k*]
using *gs-gso* **by** (*intro map-cong*) (*auto*)
also have *map* (λ*i*. *h i* ·$_v$ *gs* ! *i*) [*0*..<*k*] = *map* (λ*i*. *h i* ·$_v$ *gso i*) [*0*..<*k*]
using *gs-gso* **by** (*intro map-cong*) (*auto*)
also have *map* (λ*i*. *g i* ∗ *h i* ∗ (*gs* ! *i* · *gs* ! *i*)) [*0*..<*k*] = *map* (λ*i*. *g i* ∗ *h i* ∗ (*gso i* · *gso i*)) [*0*..<*k*]
using *gs-gso* **by** (*intro map-cong*) (*auto*)
finally show *?thesis* **by** *simp*
qed

lemma *gso-times-self-is-norm*:
assumes *j* < *m*
shows *fs* ! *j* · *gso j* = *sq-norm* (*gso j*)
by (*subst fi-scalar-prod-gso*, *insert assms*, *auto simp*: *μ.simps*)

lemma *gram-schmidt-short-vector*:
assumes *in-L*: *h* ∈ *lattice-of fs* − {*0*$_v$ *n*}
shows ∃ *i* < *m*. ‖*h*‖^2 ≥ ‖*gso i*‖^2
proof −
from *in-L* **have** *non-0*: *h* ≠ *0*$_v$ *n* **by** *auto*
from *in-L*[*unfolded lattice-of-def*] **obtain** *lam* **where**
h: *h* = *sumlist* (*map* (λ *i*. *of-int* (*lam i*) ·$_v$ *fs* ! *i*) [*0* ..< *length fs*])
by *auto*
have *in-L*: *h* = *sumlist* (*map* (λ *i*. *of-int* (*lam i*) ·$_v$ *fs* ! *i*) [*0* ..< *m*]) **unfolding** *length-map h*
by (*rule arg-cong*[*of - - sumlist*], *rule map-cong*, *auto*)
let *?n* = [*0* ..< *m*]
let *?f* = (λ *i*. *of-int* (*lam i*) ·$_v$ *fs* ! *i*)
let *?vs* = *map ?f ?n*
let *?P* = λ *k*. *k* < *m* ∧ *lam k* ≠ *0*
define *k* **where** *k* = (*GREATEST kk*. *?P kk*)
{
assume ∗: ∀ *i* < *m*. *lam i* = *0*
have *vs*: *?vs* = *map* (λ *i*. *0*$_v$ *n*) *?n*
by (*rule map-cong*, *insert f-dim* ∗, *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* = (λ*i j*. *μ i j* ·$_v$ *gso j*)
have *k*: *k* < *i* ⟹ *i* < *m* ⟹ *lam i* = *0* **for** *i*
using *Greatest-le-nat*[*of ?P i m*, *folded k-def*] **by** *auto*
define *l* **where** *l* = *lam k*
from *Pk* **have** *l*: *l* ≠ *0* **unfolding** *l-def* **by** *auto*
define *idx* **where** *idx* = [*0* ..< *k*]
have *idx*: ⋀ *i*. *i* ∈ *set idx* ⟹ *i* < *k* ⋀ *i*. *i* ∈ *set idx* ⟹ *i* < *m* **unfolding** *idx-def* **using** *kn* **by** *auto*
from *Pk* **have** *split*: [*0* ..< *m*] = *idx* @ [*k*] @ [*Suc k* ..< *m*] **unfolding** *idx-def*
by (*metis append-Cons append-self-conv2 less-Suc-eq-le less-imp-add-positive upt-add-eq-append*
upt-rec zero-less-Suc)
define *gg* **where** *gg* = *sumlist*
(*map* (λ*i*. *of-int* (*lam i*) ·$_v$ *fs* ! *i*) *idx*) + *of-int l* ·$_v$ *sumlist* (*map* (λ*j*. *μ k j* ·$_v$ *gso j*) *idx*)
have *h* = *sumlist ?vs* **unfolding** *in-L* **..**
also have ... = *sumlist* ((*map ?f idx* @ [*?f k*]) @ *map ?f* [*Suc k* ..< *m*]) **unfolding** *split* **by** *auto*
also have ... = *sumlist* (*map ?f idx* @ [*?f k*]) + *sumlist* (*map ?f* [*Suc k* ..< *m*])
by (*rule sumlist-append*, *auto intro*!: *f-carrier*, *insert Pk idx*, *auto*)
also have *sumlist* (*map ?f* [*Suc k* ..< *m*]) = *0*$_v$ *n* **by** (*rule sumlist-neutral*, *auto simp*: *k*)
also have *sumlist* (*map ?f idx* @ [*?f k*]) = *sumlist* (*map ?f idx*) + *?f k*
by (*subst sumlist-append*, *auto intro*!: *f-carrier*, *insert Pk idx*, *auto*)
also have *fs* ! *k* = *sumlist* (*map* (*?gso k*) [*0*..<*Suc k*]) **using** *fi-is-sum-of-mu-gso*[*OF kn*] **by** *simp*
also have ... = *sumlist* (*map* (*?gso k*) *idx* @ [*gso k*]) **by** (*simp add*: *μ.simps*[*of k k*] *idx-def*)
also have ... = *sumlist* (*map* (*?gso k*) *idx*) + *gso k*
by (*subst sumlist-append*, *auto intro*!: *f-carrier*, *insert Pk idx*, *auto*)
also have *of-int* (*lam k*) ·$_v$ ... = *of-int* (*lam k*) ·$_v$ (*sumlist* (*map* (*?gso k*) *idx*))
+ *of-int* (*lam k*) ·$_v$ *gso k*
unfolding *idx-def*
by (*rule smult-add-distrib-vec*[*OF sumlist-carrier*], *auto intro*!: *gso-carrier*, *insert kn*, *auto*)
finally have *h* = *sumlist* (*map ?f idx*) +
(*of-int* (*lam k*) ·$_v$ *sumlist* (*map* (*?gso k*) *idx*) + *of-int* (*lam k*) ·$_v$ *gso k*) + *0*$_v$ *n* **by** *simp*
also have ... = *gg* + *of-int l* ·$_v$ *gso k* **unfolding** *gg-def l-def*
by (*rule eq-vecI*, *insert idx kn*, *auto simp*: *sumlist-vec-index*,
subst index-add-vec, *auto simp*: *sumlist-dim kn*, *subst sumlist-dim*, *auto*)
finally have *hgg*: *h* = *gg* + *of-int l* ·$_v$ *gso k* **.**
let *?k* = [*0* ..< *k*]
define *R* **where** *R* = {*gg*. ∃ *nu*. *gg* = *sumlist* (*map* (λ *i*. *nu i* ·$_v$ *gso i*) *idx*)}
{
fix *nu*
have *dim-vec* (*sumlist* (*map* (λ *i*. *nu i* ·$_v$ *gso i*) *idx*)) = *n*
by (*rule sumlist-dim*, *insert kn*, *auto simp*: *idx-def*)

```
  } note dim-nu[simp] = this
  define kk where kk = ?k
  {
    fix v
    assume v ∈ R
   then obtain nu where v: v = sumlist (map (λ i. nu i ·_v gso i) idx) unfolding
R-def by auto
    have dim-vec v = n unfolding gg-def v by simp
  } note dim-R = this
  {
    fix v1 v2
    assume v1 ∈ R v2 ∈ R
    then obtain nu1 nu2 where v1: v1 = sumlist (map (λ i. nu1 i ·_v gso i) idx)
and
      v2: v2 = sumlist (map (λ i. nu2 i ·_v gso i) idx)
      unfolding R-def by auto
    have v1 + v2 ∈ R unfolding R-def
      by (standard, rule exI[of - λ i. nu1 i + nu2 i], unfold v1 v2, rule eq-vecI,
        (subst sumlist-vec-index, insert idx, auto intro!: gso-carrier simp: o-def)+,
        unfold sum-list-addf[symmetric], induct idx, auto simp: algebra-simps)
  } note add-R = this
  have gg ∈ R unfolding gg-def
  proof (rule add-R)
    show of-int l ·_v sumlist (map (λj. μ k j ·_v gso j) idx) ∈ R
      unfolding R-def
      by (standard, rule exI[of - λi. of-int l * μ 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) ∈ R using idx
    proof (induct idx)
      case Nil
      show ?case by (simp add: R-def, intro exI[of - λ 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 ... = ?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* = λ *x*. ((*if x* < *Suc i then of-int* (*lam i*) ∗ μ *i x else 0*) ·$_v$ *gso x*) $ *j*
have (∑ *x*←*idx*. *?f x*) = (∑ *x*←[*0* ..< *Suc i*]. *?f x*) + (∑ *x*← [*Suc i* ..< *k*]. *?f x*)
unfolding *idx-split* **by** *auto*
also have (∑ *x*← [*Suc i* ..< *k*]. *?f x*) = *0* **by** (*rule sum-list-neutral*, *insert j kn*, *auto*)
also have (∑ *x*←[*0* ..< *Suc i*]. *?f x*) = (∑ *x*←*idxs*. *of-int* (*lam i*) ∗ (μ *i x* ·$_v$ *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 ... = *of-int* (*lam i*) ∗ ((∑ *x*←[*0*..<*Suc i*]. (μ *i x* ·$_v$ *gso x*) $ *j*))
unfolding *idxs-def*[*symmetric*] **by** (*induct idxs*, *auto simp*: *algebra-simps*)
finally have (∑ *x*←*idx*. *?f x*) = *of-int* (*lam i*) ∗ ((∑ *x*←[*0*..<*Suc i*]. (μ *i x* ·$_v$ *gso x*) $ *j*))
by *simp*
} **note** *main* = *this*
show *?f i* ∈ *R* **unfolding** *fi-is-sum-of-mu-gso*[*OF i*(*1*)] *R-def*
apply (*standard*, *rule exI*[*of* - λ *j*. *if j* < *Suc i then of-int* (*lam i*) ∗ μ *i j else 0*], *rule eq-vecI*)
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 *?ff* = *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*: *?ff* = *?hh* **by** (*simp add*: *idx-def*)
from *hgg*[*unfolded gg*]
have *h*: *h* = *?ff* **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** $l * l \geq 1$ **by** (*meson eq-iff int-one-le-iff-zero-less mult-le-0-iff not-le*)
thus $1 \leq (\textit{of-int}\ (l * l) :: 'a)$ **by** *presburger*
show $0 \leq \textit{gso}\ k \cdot \textit{gso}\ k$ **by** (*rule scalar-prod-ge-0*)
qed
also have $\ldots = 0 + \textit{of-int}\ (l * l) * (\textit{gso}\ k \cdot \textit{gso}\ k)$ **by** *simp*
also have $\ldots \leq \textit{sum-list}\ (\textit{map}\ (\lambda\ i.\ (\textit{nu}\ i * \textit{nu}\ i) * (\textit{gso}\ i \cdot \textit{gso}\ i))\ \textit{idx}) + \textit{of-int}\ (l * l) * (\textit{gso}\ k \cdot \textit{gso}\ k)$
by (*rule add-right-mono*, *rule sum-list-nonneg*, *auto*, *rule mult-nonneg-nonneg*, *auto simp*: *scalar-prod-ge-0*)
also have $\textit{map}\ (\lambda\ i.\ (\textit{nu}\ i * \textit{nu}\ i) * (\textit{gso}\ i \cdot \textit{gso}\ i))\ \textit{idx} = \textit{map}\ (\lambda\ i.\ \textit{hh}\ i * \textit{hh}\ i * (\textit{gso}\ i \cdot \textit{gso}\ i))\ [0..<k]$
unfolding *idx-def* **by** (*rule map-cong*, *auto simp*: *hh-def*)
also have $\textit{of-int}\ (l * l) = \textit{hh}\ k * \textit{hh}\ k$ **unfolding** *hh-def* **by** *auto*
also have $(\sum i{\leftarrow}[0..<k].\ \textit{hh}\ i * \textit{hh}\ i * (\textit{gso}\ i \cdot \textit{gso}\ i)) + \textit{hh}\ k * \textit{hh}\ k * (\textit{gso}\ k \cdot \textit{gso}\ k)$
$= (\sum i{\leftarrow}[0..<\textit{Suc}\ k].\ \textit{hh}\ i * \textit{hh}\ i * (\textit{gso}\ i \cdot \textit{gso}\ i))$ **by** *simp*
also have $\ldots = ?hh \cdot ?hh$ **by** (*rule sym*, *rule scalar-prod-lincomb-gso*, *insert kn assms*, *auto*)
also have $\ldots = ?ff \cdot ?ff$ **by** (*simp add*: *ffhh*)
also have $\ldots = h \cdot h$ **unfolding** h **..**
finally show *?thesis* **using** *kn* **unfolding** *sq-norm-vec-as-cscalar-prod* **by** *auto*
qed

lemma *weakly-reduced-imp-short-vector*:
assumes *weakly-reduced* α m
and *in-L*: $h \in \textit{lattice-of}\ fs - \{0_v\ n\}$ **and** α*-pos*:$\alpha \geq 1$
shows $fs \neq [] \wedge \textit{sq-norm}\ (fs\ !\ 0) \leq \alpha\hat{}(m-1) * \textit{sq-norm}\ h$
proof −
from *gram-schmidt-short-vector assms* **obtain** i **where**
i: $i < m$ **and** *le*: $\textit{sq-norm}\ (\textit{gso}\ i) \leq \textit{sq-norm}\ h$ **by** *auto*
have *small*: $\textit{sq-norm}\ (fs\ !\ 0) \leq \alpha\hat{}i * \textit{sq-norm}\ (\textit{gso}\ i)$ **using** i
proof (*induct i*)
case *0*
show *?case* **unfolding** *fs0-gso0*[*OF 0*] **by** *auto*
next
case (*Suc i*)
hence $\textit{sq-norm}\ (fs\ !\ 0) \leq \alpha\hat{}i * \textit{sq-norm}\ (\textit{gso}\ i)$ **by** *auto*
also have $\ldots \leq \alpha\hat{}i * (\alpha * (\textit{sq-norm}\ (\textit{gso}\ (\textit{Suc}\ i))))$
using *reduced-gso-E*[*OF assms*(*1*) *le-refl Suc*(*2*)] α*-pos* **by** *auto*
finally show *?case* **unfolding** *class-semiring.nat-pow-Suc*[*of* α i] **by** *auto*
qed
also have $\ldots \leq \alpha\hat{}(m-1) * \textit{sq-norm}\ h$
by (*rule mult-mono*[*OF power-increasing le*], *insert i* α*-pos*, *auto*)
finally show *?thesis* **using** i **by** (*cases fs*, *auto*)
qed

lemma *sq-norm-pos*:
assumes *j*: $j < m$
shows *sq-norm* (*gso j*) > *0*
proof −
from *j* **have** *jj*: $j < m - 0$ **by** *simp*
from *orthogonalD*[*OF orthogonal-gso*, *unfolded length-map length-upt*, *OF jj jj*] *assms*
have *sq-norm* (*gso j*) ≠ *0* **using** *j* **by** (*simp add*: *sq-norm-vec-as-cscalar-prod*)

moreover have *sq-norm* (*gso j*) ≥ *0* **by** *auto*
ultimately show *0* < *sq-norm* (*gso j*) **by** *auto*
qed

lemma *Gramian-determinant*:
assumes *k*: $k \leq m$
shows *Gramian-determinant fs k* = ($\prod$ *j*<*k*. *sq-norm* (*gso j*))
Gramian-determinant fs k > *0*
proof −
define *Gk* **where** *Gk* = *mat k n* (λ (*i*,*j*). *fs* ! *i* $ *j*)
have *Gk*: *Gk* ∈ *carrier-mat k n* **unfolding** *Gk-def* **by** *auto*
define *Mk* **where** *Mk* = *mat k k* (λ (*i*,*j*). *μ i j*)
have *Mk-μ*: $i < k \Longrightarrow j < k \Longrightarrow$ *Mk* $$ (*i*,*j*) = *μ i j* **for** *i j*
unfolding *Mk-def* **using** *k* **by** *auto*
have *Mk*: *Mk* ∈ *carrier-mat k k* **and** [*simp*]: *dim-row Mk* = *k dim-col Mk* = *k* **unfolding** *Mk-def* **by** *auto*
have *det Mk* = *prod-list* (*diag-mat Mk*)
by (*rule det-lower-triangular*[*OF - Mk*], *auto simp*: *Mk-μ μ.simps*)
also have ... = *1*
by (*rule prod-list-neutral*, *auto simp*: *diag-mat-def Mk-μ μ.simps*)
finally have *detMk*: *det Mk* = *1* **.**
define *Gsk* **where** *Gsk* = *mat k n* (λ (*i*,*j*). *gso i* $ *j*)
have *Gsk*: *Gsk* ∈ *carrier-mat k n* **unfolding** *Gsk-def* **by** *auto*
have *Gsk'*: Gsk^T ∈ *carrier-mat n k* **using** *Gsk* **by** *auto*
let *?Rn* = *carrier-vec n*
have *id*: *Gk* = *Mk* ∗ *Gsk*
proof (*rule eq-matI*)
from *Gk Mk Gsk*
have *dim*: *dim-row Gk* = *k dim-row* (*Mk* ∗ *Gsk*) = *k dim-col Gk* = *n dim-col* (*Mk* ∗ *Gsk*) = *n* **by** *auto*
from *dim* **show** *dim-row Gk* = *dim-row* (*Mk* ∗ *Gsk*) *dim-col Gk* = *dim-col* (*Mk* ∗ *Gsk*) **by** *auto*
fix *i j*
assume *i* < *dim-row* (*Mk* ∗ *Gsk*) *j* < *dim-col* (*Mk* ∗ *Gsk*)
hence *ij*: $i < k$ $j < n$ **and** *i*: $i < m$ **using** *dim k* **by** *auto*
have *Gi*: *fs* ! *i* ∈ *?Rn* **using** *i* **by** *simp*
have *Gk* $$ (*i*, *j*) = *fs* ! *i* $ *j* **unfolding** *Gk-def* **using** *ij k Gi* **by** *auto*
also have ... = *FF* $$ (*i*,*j*) **using** *ij i* **by** *simp*
also have *FF* = (*M m*) ∗ *Fs* **by** (*rule matrix-equality*)

```
    also have ... $$ (i,j) = row (M m) i · col Fs j
      by (rule index-mult-mat(1), insert i ij, auto simp: mat-of-rows-list-def)
    also have row (M m) i = vec m (λ j. if j < k then Mk $$ (i,j) else 0)
      (is - = vec m ?Mk)
      unfolding Mk-def using ij i
      by (auto simp: mat-of-rows-list-def μ.simps)
   also have col Fs j = vec m (λ i'. if i' < k then Gsk $$ (i',j) else (Fs $$ (i',j)))

      (is - = vec m ?Gsk)
      unfolding Gsk-def using ij i by (auto simp: mat-of-rows-def)
    also have vec m ?Mk · vec m ?Gsk = (∑ i ∈ {0 ..< m}. ?Mk i * ?Gsk i)
      unfolding scalar-prod-def by auto
    also have ... = (∑ i ∈ {0 ..< k} ∪ {k ..< m}. ?Mk i * ?Gsk i)
      by (rule sum.cong, insert k, auto)
    also have ... = (∑ i ∈ {0 ..< k}. ?Mk i * ?Gsk i) + (∑ i ∈ {k ..< m}.
?Mk i * ?Gsk i)
      by (rule sum.union-disjoint, auto)
    also have (∑ i ∈ {k ..< m}. ?Mk i * ?Gsk i) = 0
      by (rule sum.neutral, auto)
    also have (∑ i ∈ {0 ..< k}. ?Mk i * ?Gsk i) = (∑ i' ∈ {0 ..< k}. Mk $$
(i,i') * Gsk $$ (i',j))
      by (rule sum.cong, auto)
    also have ... = row Mk i · 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: ⋀ a b c d. a = b ⟹ c = d ⟹ 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)
  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 (gso j) else 0)
(is - = ?M)
  proof (rule eq-matI)
    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: gso i ∈ ?Rn by auto
      have row Gsk i = gso i unfolding row-def Gsk-def
        by (rule eq-vecI, insert Gs ‹i < k›, auto)
    } note row = this
    have (Gsk * ?TGsk) $$ (i,j) = row Gsk i · row Gsk j using ij Gsk by auto
    also have ... = gso i · gso j using row ij by simp
    also have ... = (if i = j then sq-norm (gso j) else 0)
    proof (cases i = j)
      assume i = j
      thus ?thesis by (simp add: sq-norm-vec-as-cscalar-prod)
    next
      assume i ≠ j
      from ‹i ≠ j› orthogonalD[OF orthogonal-gso] ijn assms
      show ?thesis by auto
    qed
    also have ... = ?M $$ (i,j) using ij by simp
    finally show (Gsk * ?TGsk) $$ (i,j) = ?M $$ (i,j) .
  qed
  also have det ?M = prod-list (diag-mat ?M)
    by (rule det-upper-triangular, auto)
  also have diag-mat ?M = map (λ j. sq-norm (gso j)) [0 ..< k] unfolding
diag-mat-def by auto
  also have prod-list ... = (∏ j < k. sq-norm (gso j))
    by (subst prod.distinct-set-conv-list[symmetric], force, rule prod.cong, auto)
  finally show Gramian-determinant fs k = (∏j<k. ‖gso j‖²) .
  also have ... > 0
    by (rule prod-pos, intro ballI sq-norm-pos, insert k assms, auto)
  finally show 0 < Gramian-determinant fs k by auto
qed

lemma Gramian-determinant-div:
  assumes l < m
  shows Gramian-determinant fs (Suc l) / Gramian-determinant fs l = ‖gso l‖²
proof −
  note gram = Gramian-determinant(1)[symmetric]
  from assms have le: Suc l ≤ m l ≤ m by auto
  have (∏j<Suc l. ‖gso j‖²) = (∏j ∈ {0..<l} ∪ {l}. ‖gso j‖²)
    using assms by (intro prod.cong) (auto)
  also have ... = (∏j<l. ‖gso j‖²) * ‖gso l‖²
    using assms by (subst prod-Un) (auto simp add: atLeast0LessThan)
  finally show ?thesis unfolding gram[OF le(1)] gram[OF le(2)]
    using Gramian-determinant(2)[OF le(2)] assms by auto
```

qed

end

lemma (**in** *gram-schmidt-fs-Rn*) *Gramian-determinant-Ints*:
assumes $k \leq m$ $\bigwedge i\ j.\ i < n \Longrightarrow j < m \Longrightarrow fs\ !\ j\ \$\ i \in \mathbb{Z}$
shows *Gramian-determinant fs k* $\in \mathbb{Z}$
proof −
let *?oi* = *of-int* :: *int* ⇒ *'a*
from *assms* **have** $\bigwedge\ i.\ i < n \Longrightarrow \forall j.\ \exists\ c.\ j < m \longrightarrow fs\ !\ j\ \$\ i = ?oi\ c$ **unfolding** *Ints-def* **by** *auto*
from *choice*[*OF this*] **have** $\forall\ i.\ \exists\ c.\ \forall\ j.\ i < n \longrightarrow j < m \longrightarrow fs\ !\ j\ \$\ i = ?oi\ (c\ j)$ **by** *blast*
from *choice*[*OF this*] **obtain** *c* **where** *c*: $\bigwedge\ i\ j.\ i < n \Longrightarrow j < m \Longrightarrow fs\ !\ j\ \$\ i = ?oi\ (c\ i\ j)$ **by** *blast*
define *d* **where** $d = map\ (\lambda\ j.\ vec\ n\ (\lambda\ i.\ c\ i\ j))\ [0..<m]$
have *fs*: $fs = map\ (map\text{-}vec\ ?oi)\ d$
unfolding *d-def* **by** (*rule nth-equalityI*, *auto intro*!: *eq-vecI c*)
have *id*: $mat\ k\ n\ (\lambda(i,\ y).\ map\ (map\text{-}vec\ ?oi)\ d\ !\ i\ \$\ y) = map\text{-}mat\ of\text{-}int\ (mat\ k\ n\ (\lambda(i,\ y).\ d\ !\ i\ \$\ y))$
by (*rule eq-matI*, *insert* ‹$k \leq m$›, *auto simp*: *d-def o-def*)
show *?thesis* **unfolding** *fs Gramian-determinant-def Gramian-matrix-def Let-def id*
map-mat-transpose
by (*subst of-int-hom.mat-hom-mult*[*symmetric*], *auto*)
qed

locale *gram-schmidt-fs-int* = *gram-schmidt-fs-lin-indpt* +
assumes *fs-int*: $\bigwedge i\ j.\ i < n \Longrightarrow j < m \Longrightarrow fs\ !\ j\ \$\ i \in \mathbb{Z}$
begin

lemma *Gramian-determinant-ge1*:
assumes $k \leq m$
shows $1 \leq$ *Gramian-determinant fs k*
proof −
have $0 <$ *Gramian-determinant fs k*
by (*simp add*: *assms Gramian-determinant*(*2*) *less-or-eq-imp-le*)
moreover have *Gramian-determinant fs k* $\in \mathbb{Z}$
by (*simp add*: *Gramian-determinant-Ints assms fs-int*)
ultimately show *?thesis*
using *Ints-nonzero-abs-ge1* **by** *fastforce*
qed

lemma *mu-bound-Gramian-determinant*:
assumes $l < k$ $k < m$
shows $(\mu\ k\ l)^2 \leq$ *Gramian-determinant fs l* $* \|fs\ !\ k\|^2$
proof −
have $(\mu\ k\ l)^2\ =\ (fs\ !\ k \cdot gso\ l)^2\ /\ (\|gso\ l\|^2)^2$
using *assms* **by** (*simp add*: *power-divide* *μ.simps*)

also have ... $\leq (\|fs\ !\ k\|^2 * \|gso\ l\|^2)\ /\ (\|gso\ l\|^2)^2$
using *assms* **by** (*auto intro*!: *scalar-prod-Cauchy divide-right-mono*)
also have ... $= \|fs\ !\ k\|^2\ /\ \|gso\ l\|^2$
by (*auto simp add*: *field-simps power2-eq-square*)
also have ... $=$ *‖fs ! k‖² / (Gramian-determinant fs (Suc l) / Gramian-determinant fs l)*
using *assms* **by** (*subst Gramian-determinant-div*[*symmetric*]) *auto*
also have ... $=$ *Gramian-determinant fs l * ‖fs ! k‖² / Gramian-determinant fs (Suc l)*
by (*auto simp add*: *field-simps*)
also have ... $\leq$ *Gramian-determinant fs l * ‖fs ! k‖² / 1*
by (*rule divide-left-mono*, *insert Gramian-determinant-ge1*[*of l*] *Gramian-determinant-ge1*[*of Suc l*] *assms*,
auto intro!: *mult-nonneg-nonneg*)
finally show *?thesis*
by *simp*
qed

end

context *gram-schmidt*
begin

lemma *gso-cong*:
fixes *f1 f2* :: *'a vec list*
assumes $\bigwedge$ *i. i ≤ x ⟹ f1 ! i = f2 ! i*
shows *gram-schmidt-fs.gso n f1 x = gram-schmidt-fs.gso n f2 x*
using *assms*
proof(*induct x rule*:*nat-less-induct*[*rule-format*])
case (*1 x*)
interpret *f1*: *gram-schmidt-fs n f1* **.**
interpret *f2*: *gram-schmidt-fs n f2* **.**
have ∗: *map* (*λj. − f1.μ x j ·ᵥ f1.gso j*) [*0..<x*] = *map* (*λj. − f2.μ x j ·ᵥ f2.gso j*) [*0..<x*]
using *1* **by** (*intro map-cong*) (*auto simp add*: *f1.μ.simps f2.μ.simps*)
show *?case*
using *1* **by** (*subst f1.gso.simps*, *subst f2.gso.simps*, *subst* ∗) *auto*
qed

lemma *μ-cong*:
fixes *f1 f2* :: *'a vec list*
assumes $\bigwedge$ *k. j < i ⟹ k ≤ j ⟹ f1 ! k = f2 ! k*
and *j < i ⟹ f1 ! i = f2 ! i*
shows *gram-schmidt-fs.μ n f1 i j = gram-schmidt-fs.μ n f2 i j*
proof −
interpret *f1*: *gram-schmidt-fs n f1* **.**
interpret *f2*: *gram-schmidt-fs n f2* **.**
from *gso-cong*[*of j f1 f2*] *assms* **have** *id*: *j < i ⟹ f1.gso j = f2.gso j* **by** *auto*
show *?thesis* **unfolding** *f1.μ.simps f2.μ.simps* **using** *assms id* **by** *auto*

qed

end

lemma *prod-list-le-mono*: **fixes** *us* :: *'a* :: {*linordered-nonzero-semiring*,*ordered-ring*} *list*
  assumes *length us* = *length vs*
  and ⋀ *i*. *i* < *length vs* ⟹ *0* ≤ *us* ! *i* ∧ *us* ! *i* ≤ *vs* ! *i*
shows *0* ≤ *prod-list us* ∧ *prod-list us* ≤ *prod-list vs*
  using *assms*
proof (*induction us vs rule*: *list-induct2*)
  case (*Cons u us v vs*)
  have *0* ≤ *prod-list us* ∧ *prod-list us* ≤ *prod-list vs*
    by (*rule Cons.IH*, *insert Cons.prems*[*of Suc i* **for** *i*], *auto*)
  moreover have *0* ≤ *u* ∧ *u* ≤ *v* **using** *Cons.prems*[*of 0*] **by** *auto*
  ultimately show *?case* **by** (*auto intro*: *mult-mono*)
qed *simp*

lemma *lattice-of-of-int*: **assumes** *G*: *set F* ⊆ *carrier-vec n*
  and *f* ∈ *vec-module.lattice-of n F*
shows *map-vec rat-of-int f* ∈ *vec-module.lattice-of n* (*map* (*map-vec of-int*) *F*)
  (**is** *?f* ∈ *vec-module.lattice-of - ?F*)
proof −
  let *?sl* = *abelian-monoid.sumlist* (*module-vec TYPE*(*'a*::*semiring-1*) *n*)
  note *d* = *vec-module.lattice-of-def*
  note *dim* = *vec-module.sumlist-dim*
  note *sumlist-vec-index* = *vec-module.sumlist-vec-index*
  from *G* **have** *Gi*: ⋀ *i*. *i* < *length F* ⟹ *F* ! *i* ∈ *carrier-vec n* **by** *auto*
  from *Gi* **have** *Gid*: ⋀ *i*. *i* < *length F* ⟹ *dim-vec* (*F* ! *i*) = *n* **by** *auto*
  from *assms*(*2*)[*unfolded d*]
  obtain *c* **where**
    ffc: *f* = *?sl* (*map* (λ*i*. *of-int* (*c i*) ·$_v$ *F* ! *i*) [*0*..<*length F*]) (**is** - = *?g*) **by** *auto*

  have *?f* = *?sl* (*map* (λ*i*. *of-int* (*c i*) ·$_v$ *?F* ! *i*) [*0*..<*length ?F*]) (**is** - = *?gg*)
  proof −
    have *d1*[*simp*]: *dim-vec ?g* = *n* **by** (*subst dim*, *auto simp*: *Gi*)
   have *d2*[*simp*]: *dim-vec ?gg* = *n* **unfolding** *length-map* **by** (*subst vec-module.sumlist-dim*, *auto simp*: *Gi G*)
    show *?thesis*
      unfolding *ffc length-map*
      apply (*rule eq-vecI*)
       apply (*insert d1 d2*, *auto*)[*2*]
      apply (*subst* (*1 2*) *sumlist-vec-index*, *auto simp*: *o-def Gi G*)
      apply (*unfold of-int-hom.hom-sum-list*)
      apply (*intro arg-cong*[*of - - sum-list*] *map-cong*)
        by (*auto simp*: *G Gi*, (*subst index-smult-vec*, *simp add*: *Gid*)+,
         subst index-map-vec, *auto simp*: *Gid*)
  qed
  thus *?f* ∈ *vec-module.lattice-of n ?F* **unfolding** *d* **by** *auto*

qed

lemma *Hadamard's-inequality*:
fixes *A*::*real mat*
assumes *A*: *A* ∈ *carrier-mat n n*
shows *abs* (*det A*) ≤ *sqrt* (*prod-list* (*map sq-norm* (*rows A*)))
proof −
let *?us* = *map* (*row A*) [*0* ..< *n*]
interpret *gso*: *gram-schmidt-fs n ?us* **.**
have *len*: *length ?us* = *n* **by** *simp*
have *us*: *set ?us* ⊆ *carrier-vec n* **using** *A* **by** *auto*
let *?vs* = *map gso.gso* [*0*..<*n*]
show *?thesis*
proof (*cases carrier-vec n* ⊆ *gso.span* (*set ?us*))
case *True*
with *us len* **have** *basis*: *gso.basis-list ?us* **unfolding** *gso.basis-list-def* **by** *auto*
note *in-dep* = *gso.basis-list-imp-lin-indpt-list*[*OF basis*]
interpret *gso*: *gram-schmidt-fs-lin-indpt n ?us*
by (*standard*) (*use in-dep gso.lin-indpt-list-def* **in** *auto*)
have *last*: *0* ≤ *prod-list* (*map sq-norm ?vs*) ∧ *prod-list* (*map sq-norm ?vs*) ≤
prod-list (*map sq-norm ?us*)
proof (*rule prod-list-le-mono*, *force*, *unfold length-map length-upt*)
fix *i*
assume *i* < *n* − *0*
hence *i*: *i* < *n* **by** *simp*
have *vsi*: *map sq-norm ?vs* ! *i* = *sq-norm* (*?vs* ! *i*) **using** *i* **by** *simp*
have *usi*: *map sq-norm ?us* ! *i* = *sq-norm* (*row A i*) **using** *i* **by** *simp*
have *zero*: *0* ≤ *sq-norm* (*?vs* ! *i*) **by** *auto*
have *le*: *sq-norm* (*?vs* ! *i*) ≤ *sq-norm* (*row A i*)
using *gso.sq-norm-gso-le-f i* **by** *simp*
show *0* ≤ *map sq-norm ?vs* ! *i* ∧ *map sq-norm ?vs* ! *i* ≤ *map sq-norm ?us* !
i
unfolding *vsi usi* **using** *zero le* **by** *auto*
qed
have *Fs*: *gso.FF* ∈ *carrier-mat n n* **by** *auto*
have *A-Fs*: *A* = *gso.FF*
by (*rule eq-matI*, *subst gso.FF-index*, *insert A*, *auto*)
hence *abs* (*det A*) = *abs* (*det* (*gso.FF*)) **by** *simp*

also have ... = *abs* (*sqrt* (*det* (*gso.FF*) ∗ *det* (*gso.FF*))) **by** *simp*
also have *det* (*gso.FF*) ∗ *det* (*gso.FF*) = *det* (*gso.FF*) ∗ *det* (*gso.FF*)^T
unfolding *det-transpose*[*OF Fs*] **..**
also have ... = *det* (*gso.FF* ∗ (*gso.FF*)^T)
by (*subst det-mult*[*OF Fs*], *insert Fs*, *auto*)
also have ... = *gso.Gramian-determinant ?us n*
unfolding *gso.Gramian-matrix-def gso.Gramian-determinant-def Let-def A-Fs*[*symmetric*]
by (*rule arg-cong*[*of - - det*], *rule arg-cong2*[*of - - - - (∗)*], *insert A*, *auto*)

also have ... = $(\prod j \in set\ [0\ ..<\ n].\ \|?vs\ !\ j\|^2)$
by (*subst gso.Gramian-determinant*) (*auto intro*!: *prod.cong*)
also have ... = *prod-list* (*map* (λ *i. sq-norm* (*?vs* ! *i*)) [*0* ..< *n*])
by (*subst prod.distinct-set-conv-list*, *auto*)
also have *map* (λ *i. sq-norm* (*?vs* ! *i*)) [*0* ..< *n*] = *map sq-norm ?vs*
by (*intro nth-equalityI*, *auto*)
also have *abs* (*sqrt* (*prod-list* ...)) $\leq$ *sqrt* (*prod-list* (*map sq-norm ?us*))
using *last* **by** *simp*
also have *?us* = *rows A* **unfolding** *rows-def* **using** *A* **by** *simp*
finally show *?thesis* **.**
next
case *False*
from *mat-of-rows-rows*[*unfolded rows-def*,*of A*] *A gram-schmidt.non-span-det-zero*[*OF len False us*]
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 *?thesis* **unfolding** *zero* **using** *ge* **by** *simp*
qed
qed

definition *gram-schmidt-wit* = *gram-schmidt.main*

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* $\Rightarrow$ *int vec list* $\Rightarrow$ *rat list list* $\times$ *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-fs.gso-connect*
by *metis*

Faster implementation for rational vectors which also avoid recomputations of square-norms

fun *adjuster-triv* :: *nat* $\Rightarrow$ *rat vec* $\Rightarrow$ (*rat vec* $\times$ *rat*) *list* $\Rightarrow$ *rat vec*
where *adjuster-triv n w* [] = 0_v *n*
| *adjuster-triv n w* ((*u*,*nu*)#*us*) = $-(w \cdot u)/$ *nu* $\cdot_v$ *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-vec-rat u*) # *us*) *ws*)

definition *gram-schmidt-triv* :: *nat* $\Rightarrow$ *rat vec list* $\Rightarrow$ (*rat vec* $\times$ *rat*) *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*])

context *gram-schmidt*
begin

fun *mus-adjuster* :: *'a vec* ⇒ (*'a vec* × *'a*) *list* ⇒ *'a list* ⇒ *'a vec* ⇒ *'a list* × *'a vec*
where
mus-adjuster f [] *mus g'* = (*mus*, *g'*) |
mus-adjuster f ((*g*, *ng*)#*n-gs*) *mus g'* = (*let a* = (*f* ⋅ *g*) / *ng in*
mus-adjuster f n-gs (*a* # *mus*) (−*a* ⋅$_v$ *g* + *g'*))

fun *norms-mus'* **where**
norms-mus' [] *n-gs mus* = (*map snd n-gs*, *mus*) |
norms-mus' (*f* # *fs*) *n-gs mus* =
(*let* (*mus-row*, *g'*) = *mus-adjuster f n-gs* [] (*0*$_v$ *n*);
g = *g'* + *f in*
norms-mus' fs ((*g*, *sq-norm-vec g*) # *n-gs*) (*mus-row*#*mus*))

lemma *adjuster-wit-carrier-vec*:
assumes *f* ∈ *carrier-vec n set gs* ⊆ *carrier-vec n*
shows *snd* (*adjuster-wit mus f gs*) ∈ *carrier-vec n*
using *assms*
by (*induction mus f gs rule*: *adjuster-wit.induct*) (*auto simp add*: *Let-def case-prod-beta'*)

lemma *adjuster-wit''*:
assumes *adjuster-wit mus-acc f gs* = (*mus*, *g'*) *n-gs* = *map* (λ*x*. (*x*, *sq-norm-vec x*)) *gs*
f ∈ *carrier-vec n acc* ∈ *carrier-vec n set gs* ⊆ *carrier-vec n*
shows *mus-adjuster f n-gs mus-acc acc* = (*mus*, *acc* + *g'*)
using *assms* **proof**(*induction f n-gs mus-acc acc arbitrary*: *g' gs mus rule*: *mus-adjuster.induct*)
case (*1 mus' f acc g*)
then show *?case*
by *auto*

next
case (*2 f g n-g n-gs mus-acc acc g′ gs mus*)
let *?gg* = *snd* (*adjuster-wit* (*f* ⋅ *g* / *n-g* # *mus-acc*) *f* (*tl gs*))
from *2* **have** *l*: *gs* = *g* # *tl gs*
by *auto*
have *gg*: *?gg* ∈ *carrier-vec n*
using *2* **by** (*auto intro*!: *adjuster-wit-carrier-vec*)
then have [*simp*]: *g′* = (− (*f* ⋅ *g* / ‖*g*‖²) ·$_v$ *g* + *?gg*)
using *2* **by** (*auto simp add*: *Let-def case-prod-beta′*)
have *mus-adjuster f* ((*g*, *n-g*) # *n-gs*) *mus-acc acc* =
mus-adjuster f n-gs (*f* ⋅ *g* / *n-g* # *mus-acc*) (− (*f* ⋅ *g* / *n-g*) ·$_v$ *g* + *acc*)
by (*auto simp add*: *Let-def*)
also have ... = (*mus*, − (*f* ⋅ *g* / *n-g*) ·$_v$ *g* + *acc* + *?gg*)
proof −
have *adjuster-wit* (*f* ⋅ *g* / *n-g* # *mus-acc*) *f* (*tl gs*) = (*mus*, *?gg*)
using *2* **by** (*subst* (*asm*) *l*) (*auto simp add*: *Let-def case-prod-beta′*)
then show *?thesis*
using *2* **by** (*subst 2*(*1*)[*of* - *tl gs*]) (*auto simp add*: *Let-def case-prod-beta′*)
qed
finally show *?case*
using *2 gg* **by** *auto*
qed

lemma *adjuster-wit′*:
assumes *n-gs* = *map* (λ*x*. (*x*, *sq-norm-vec x*)) *gs f* ∈ *carrier-vec n set gs* ⊆ *carrier-vec n*
shows *mus-adjuster f n-gs mus-acc* (*0*$_v$ *n*) = *adjuster-wit mus-acc f gs*
proof −
let *?g* = *snd* (*adjuster-wit mus-acc f gs*)
let *?mus* = *fst* (*adjuster-wit mus-acc f gs*)
have *?g* ∈ *carrier-vec n*
using *assms* **by** (*auto intro*!: *adjuster-wit-carrier-vec*)
then show *?thesis*
using *assms* **by** (*subst adjuster-wit′′*[*of* - - *gs ?mus ?g*]) (*auto simp add*: *case-prod-beta′*)
qed

lemma *sub2-wit-norms-mus′*:
assumes *n-gs′* = *map* (λ*v*. (*v*, *sq-norm-vec v*)) *gs′*
sub2-wit gs′ fs = (*mus*, *gs*) *set fs* ⊆ *carrier-vec n set gs′* ⊆ *carrier-vec n*
shows *norms-mus′ fs n-gs′ mus-acc* = (*map sq-norm-vec* (*rev gs* @ *gs′*), *rev mus* @ *mus-acc*)
using *assms* **proof** (*induction fs n-gs′ mus-acc arbitrary*: *gs′ mus gs rule*: *norms-mus′.induct*)
case (*1 n-gs mus-acc*)
then show *?case* **by** (*auto simp add*: *rev-map*)
next
case (*2 f fs n-gs mus-acc*)
note *aw1* = *conjunct1*[*OF conjunct2*[*OF gram-schmidt-fs.adjuster-wit*]]
let *?aw* = *mus-adjuster f n-gs* [] (*0*$_v$ *n*)

have *aw*: *?aw = adjuster-wit [] f gs′*
apply(*subst adjuster-wit′*) **using** *2* **by** *auto*
have *sub2-wit ((snd ?aw + f) # gs′) fs = sub2-wit ((snd (adjuster-wit [] f gs′) + f) # gs′) fs*
apply(*subst adjuster-wit′*) **using** *2* **by** *auto*
also have *... = (tl mus, tl gs)*
using *2* **by** (*auto simp add*: *Let-def case-prod-beta′*)
finally have *sub-tl*: *sub2-wit ((snd ?aw + f) # gs′) fs = (tl mus, tl gs)*
by *simp*
have *aw-c*: *snd ?aw ∈ carrier-vec n*
apply(*subst adjuster-wit′*[*of - gs′*])
using *2 adjuster-wit-carrier-vec* **by** (*auto*)
have *gs*: *gs = (snd ?aw + f) # tl gs*
apply(*subst aw*) **using** *2* **by** (*auto simp add*: *Let-def case-prod-beta′*)
have *mus*: *mus = fst ?aw # tl mus*
apply(*subst aw*) **using** *2* **by** (*auto simp add*: *Let-def case-prod-beta′*)
show *?case* **apply**(*simp add*: *Let-def case-prod-beta′*)
apply(*subst 2(1)*[*of - - - - (snd ?aw + f)#gs′ tl mus tl gs*]) **apply**(*simp*) **defer**
apply(*simp*)
apply (*simp add*: *2.prems*(*1*))
using *sub-tl* **apply**(*simp*)
using *2* **apply**(*simp*)
subgoal using *2 aw-c* **by** (*auto*)
defer
apply(*simp*)
apply(*auto*)
using *gs*
apply(*subst gs*) **apply**(*subst* (*2*) *gs*)
apply (*metis list.simps*(*9*) *rev.simps*(*2*) *rev-map*)
using *mus*
by (*metis rev.simps*(*2*))
qed

lemma *sub2-wit-gram-schmidt-sub-triv′′*:
assumes *sub2-wit [] fs = (mus, gs) set fs ⊆ carrier-vec n*
shows *norms-mus′ fs [] [] = (map sq-norm-vec (rev gs), rev mus)*
using *assms* **by** (*subst sub2-wit-norms-mus′*) (*simp*)+

definition *norms-mus* **where**
norms-mus fs = (let (n-gs, mus) = norms-mus′ fs [] [] in (rev n-gs, rev mus))

lemma *sub2-wit-gram-schmidt-norm-mus*:
assumes *sub2-wit [] fs = (mus, gs) set fs ⊆ carrier-vec n*
shows *norms-mus fs = (map sq-norm-vec gs, mus)*
unfolding *norms-mus-def* **using** *assms sub2-wit-gram-schmidt-sub-triv′′*
by (*auto simp add*: *Let-def case-prod-beta′ rev-map*)

lemma (**in** *gram-schmidt-fs-Rn*) *norms-mus*: **assumes** *set fs ⊆ carrier-vec n length fs ≤ n*

shows *norms-mus fs* = (*map* (λ*j*. ‖*gso j*‖²) [*0*..<*length fs*], *map* (λ*i*. *map* (μ *i*) [*0*..<*i*]) [*0*..<*length fs*])
proof −
let *?s* = *sub2-wit* [] *fs*
have *gram-schmidt-sub2 n* [] *fs* = *snd ?s* ∧ *snd ?s* = *map* (*gso*) [*0*..<*length fs*] ∧ *fst ?s* = *map* (λ*i*. *map* (μ *i*) [*0*..<*i*]) [*0*..<*length fs*]
using *assms* **by** (*intro sub2-wit*) (*auto simp add*: *map-nth*)
then have *1*: *snd ?s* = *map* (*gso*) [*0*..<*length fs*] **and** *2*: *fst ?s* = *map* (λ*i*. *map* (μ *i*) [*0*..<*i*]) [*0*..<*length fs*]
by *auto*
have *s*: *?s* = (*fst ?s*, *snd ?s*) **by** *auto*
show *?thesis*
unfolding *sub2-wit-gram-schmidt-norm-mus*[*OF s assms*(*1*)]
unfolding *1 2 o-def map-map* **by** *auto*
qed

end

fun *mus-adjuster-rat* :: *rat vec* ⇒ (*rat vec* × *rat*) *list* ⇒ *rat list* ⇒ *rat vec* ⇒ *rat list* × *rat vec*
where
mus-adjuster-rat f [] *mus g'* = (*mus*, *g'*) |
mus-adjuster-rat f ((*g*, *ng*)#*n-gs*) *mus g'* = (*let a* = (*f* • *g*) / *ng in*
mus-adjuster-rat f n-gs (*a* # *mus*) (−*a* ·$_v$ *g* + *g'*))

fun *norms-mus-rat'* **where**
norms-mus-rat' n [] *n-gs mus* = (*map snd n-gs*, *mus*) |
norms-mus-rat' n (*f* # *fs*) *n-gs mus* =
(*let* (*mus-row*, *g'*) = *mus-adjuster-rat f n-gs* [] (*0*$_v$ *n*);
g = *g'* + *f in*
norms-mus-rat' n fs ((*g*, *sq-norm-vec g*) # *n-gs*) (*mus-row*#*mus*))

definition *norms-mus-rat* **where**
norms-mus-rat n fs = (*let* (*n-gs*, *mus*) = *norms-mus-rat' n fs* [] [] *in* (*rev n-gs*, *rev mus*))

lemma *norms-mus-rat-norms-mus*:
norms-mus-rat n fs = *gram-schmidt.norms-mus n fs*
proof −
have *mus-adjuster-rat f n-gs mus-acc g-acc* = *gram-schmidt.mus-adjuster f n-gs mus-acc g-acc*
for *f n-gs mus-acc g-acc*
by(*induction f n-gs mus-acc g-acc rule*: *mus-adjuster-rat.induct*)
(*auto simp add*: *gram-schmidt.mus-adjuster.simps*)
then have *norms-mus-rat' n fs n-gs mus* = *gram-schmidt.norms-mus' n fs n-gs mus* **for** *n fs n-gs mus*
by(*induction n fs n-gs mus rule*: *norms-mus-rat'.induct*)
(*auto simp add*: *gram-schmidt.norms-mus'.simps case-prod-beta'*)

then show *?thesis*
unfolding *norms-mus-rat-def gram-schmidt.norms-mus-def* **by** *auto*
qed

lemma *of-int-dvd*:
b dvd a **if** *of-int a / (of-int b :: 'a :: field-char-0) ∈ ℤ b ≠ 0*
using *that* **by** (*cases rule*: *Ints-cases*)
(*simp add*: *field-simps flip*: *of-int-mult*)

lemma *denom-dvd-ints*:
fixes *i*::*int*
assumes *quotient-of r = (z, n) of-int i * r ∈ ℤ*
shows *n dvd i*
proof −
have *rat-of-int i * (rat-of-int z / rat-of-int n) ∈ ℤ*
using *assms quotient-of-div* **by** *blast*
then have *n dvd i * z*
using *quotient-of-denom-pos assms* **by** (*auto intro*!: *of-int-dvd*)
then show *n dvd i*
using *assms algebraic-semidom-class.coprime-commute*
quotient-of-coprime coprime-dvd-mult-left-iff **by** *blast*
qed

lemma *quotient-of-bounds*:
assumes *quotient-of r = (n, d) rat-of-int i * r ∈ ℤ 0 < i |r| ≤ b*
shows *of-int |n| ≤ of-int i * b d ≤ i*
proof −
show *ni*: *d ≤ i*
using *assms denom-dvd-ints* **by** (*intro zdvd-imp-le*) *blast+*
have *|r| = |rat-of-int n / rat-of-int d|*
using *assms quotient-of-div* **by** *blast*
also have *... = rat-of-int |n| / rat-of-int d*
using *assms* **using** *quotient-of-denom-pos* **by** *force*
finally have *of-int |n| = rat-of-int d * |r|*
using *assms* **by** *auto*
also have *... ≤ rat-of-int d * b*
using *assms quotient-of-denom-pos* **by** *auto*
also have *... ≤ rat-of-int i * b*
using *ni assms of-int-le-iff* **by** (*auto intro*!: *mult-right-mono*)
finally show *rat-of-int |n| ≤ rat-of-int i * b*
by *simp*
qed

context *gram-schmidt-fs-Rn*
begin

lemma *ex-κ*:

assumes $i < \mathit{length}\ \mathit{fs}\ l \leq i$
shows $\exists \kappa.\ \mathit{sumlist}\ (\mathit{map}\ (\lambda j.\ -\ \mu\ i\ j \cdot_v \mathit{gso}\ j)\ [0\ ..<\ l]) =$
$\mathit{sumlist}\ (\mathit{map}\ (\lambda j.\ \kappa\ j \cdot_v \mathit{fs}\ !\ j)\ [0\ ..<\ l])$ (**is** $\exists \kappa.\ ?\mathit{Prop}\ l\ i\ \kappa$)
using *assms*
proof (*induction l arbitrary*: *i*)
case (*Suc l*)
then obtain κ_i **where** κ_i-*def*: *?Prop l i* κ_i
by *force*
from *Suc* **obtain** κ_l **where** κ_l-*def*: *?Prop l l* κ_l
by *force*
have [*simp*]: *dim-vec* (*M.sumlist* (*map* ($\lambda j.$ *f j* $\cdot_v$ *fs* ! *j*) [*0*..<*y*])) = *n* **if** $y \leq \mathit{Suc}\ l$ **for** *f y*
using *Suc that* **by** (*auto intro*!: *dim-sumlist*)
define κ **where** $\kappa = (\lambda x.\ (\mathit{if}\ x < l\ \mathit{then}\ \kappa_i\ x - \kappa_l\ x * \mu\ i\ l\ \mathit{else}\ \ -\ \mu\ i\ l))$
let *?sum* = $\lambda i.\ \mathit{sumlist}\ (\mathit{map}\ (\lambda j.\ -\ \mu\ i\ j \cdot_v \mathit{gso}\ j)\ [0..<l])$
have *M.sumlist* (*map* ($\lambda j.\ -\ \mu\ i\ j \cdot_v$ *gso j*) [*0*..<*Suc l*]) =
M.sumlist (*map* ($\lambda j.\ \kappa_i\ j \cdot_v$ *fs* ! *j*) [*0*..<*l*]) + $-\ \mu\ i\ l \cdot_v$ *gso l*
using *Suc* **by** (*subst* κ_i-*def*[*symmetric*], *subst sumlist-snoc*[*symmetric*]) (*auto*)
also have *gso l* = *fs* ! *l* + *M.sumlist* (*map* ($\lambda j.\ \kappa_l\ j \cdot_v$ *fs* ! *j*) [*0*..<*l*])
by (*subst gso.simps*) (*auto simp add*: κ_l-*def*)
also have *M.sumlist* (*map* ($\lambda j.\ \kappa_i\ j \cdot_v$ *fs* ! *j*) [*0*..<*l*]) +
$-\ \mu\ i\ l \cdot_v$ (*fs* ! *l* + *M.sumlist* (*map* ($\lambda j.\ \kappa_l\ j \cdot_v$ *fs* ! *j*) [*0*..<*l*]))
= *M.sumlist* (*map* ($\lambda j.\ \kappa\ j \cdot_v$ *fs* ! *j*) [*0*..<*Suc l*]) (**is** *?lhs* = *?rhs*)
proof −
have *?lhs* $ *k* = *?rhs* $ *k* **if** $k < n$ **for** *k*
proof −
have (*M.sumlist* (*map* ($\lambda j.\ \kappa_i\ j \cdot_v$ *fs* ! *j*) [*0*..<*l*]) +
$-\ \mu\ i\ l \cdot_v$ (*fs* ! *l* + *M.sumlist* (*map* ($\lambda j.\ \kappa_l\ j \cdot_v$ *fs* ! *j*) [*0*..<*l*]))) $ *k*
= (*M.sumlist* (*map* ($\lambda j.\ \kappa_i\ j \cdot_v$ *fs* ! *j*) [*0*..<*l*]) $ *k* +
$-\ \mu\ i\ l$ * (*fs* ! *l* $ *k* + *M.sumlist* (*map* ($\lambda j.\ \kappa_l\ j \cdot_v$ *fs* ! *j*) [*0*..<*l*]) $ *k*))
using *that* **by** *auto*
also have ... = ($\sum j = 0..<l.\ \kappa_i\ j$ * *fs* ! *j* $ *k*)
+ ($-\ \mu\ i\ l$ * ($\sum j = 0..<l.\ \kappa_l\ j$ * *fs* ! *j* $ *k*)) − $\mu\ i\ l$ * *fs* ! *l* $ *k*
using *that Suc* **by** (*auto simp add*: *algebra-simps sumlist-nth*)
also have $-\ \mu\ i\ l$ * ($\sum j = 0..<l.\ \kappa_l\ j$ * *fs* ! *j* $ *k*)
= ($\sum j = 0..<l.\ -\ \mu\ i\ l$ * ($\kappa_l\ j$ * *fs* ! *j* $ *k*))
using *sum-distrib-left* **by** *blast*
also have ($\sum j = 0..<l.\ \kappa_i\ j$ * *fs* ! *j* $ *k*) + ($\sum j = 0..<l.\ -\ \mu\ i\ l$ * ($\kappa_l\ j$ * *fs* ! *j* $ *k*)) =
($\sum x = 0..<l.\ (\kappa_i\ x - \kappa_l\ x * \mu\ i\ l)$ * *fs* ! *x* $ *k*)
by (*subst sum.distrib*[*symmetric*]) (*simp add*: *algebra-simps*)
also have ... = ($\sum x = 0..<l.\ \kappa\ x$ * *fs* ! *x* $ *k*)
unfolding κ-*def* **by** (*rule sum.cong*) (*auto*)
also have ($\sum x = 0..<l.\ \kappa\ x$ * *fs* ! *x* $ *k*) − $\mu\ i\ l$ * *fs* ! *l* $ *k* =
($\sum x = 0..<l.\ \kappa\ x$ * *fs* ! *x* $ *k*) + ($\sum x = l..<\mathit{Suc}\ l.\ \kappa\ x$ * *fs* ! *x* $ *k*)
unfolding κ-*def* **by** *auto*
also have ... = ($\sum x = 0..<\mathit{Suc}\ l.\ \kappa\ x$ * *fs* ! *x* $ *k*)
by (*subst sum.union-disjoint*[*symmetric*]) *auto*
also have ... = ($\sum x = 0..<\mathit{Suc}\ l.\ (\kappa\ x \cdot_v$ *fs* ! *x*) $ *k*)

```
    using that Suc by auto
  also have ... = M.sumlist (map (λj. κ j ·v fs ! j) [0..<Suc l]) $ k
    by (subst sumlist-nth, insert that Suc, auto simp: nth-append)
  finally show ?thesis by simp
 qed
 then show ?thesis
  using Suc by (auto simp add: dim-sumlist)
qed
finally show ?case by (intro exI[of - κ]) simp
qed auto


definition κ-SOME-def:
 κ = (SOME κ. ∀ i l. i < length fs ⟶ l ≤ i ⟶
      sumlist (map (λj. − μ i j ·v gso j) [0..<l]) =
      sumlist (map (λj. κ i l j ·v fs ! j) [0..<l]))

lemma κ-def:
 assumes i < length fs l ≤ i
 shows sumlist (map (λj. − μ i j ·v gso j) [0..<l]) =
       sumlist (map (λj. κ i l j ·v fs ! j) [0..<l])
proof −
 let ?P = λ i l κ. (i < length fs ⟶ l ≤ i ⟶
              sumlist (map (λj. − μ i j ·v gso j) [0..<l]) =
              sumlist (map (λj. κ j ·v fs ! j) [0..<l]))
 from ex-κ have ⋀ i. ∀ l. ∃ κ. ?P i l κ by blast
 from choice[OF this] have ∀ i. ∃ κ. ∀ l. ?P i l (κ l) by blast
 from choice[OF this] have ∃ κ. ∀ i l. ?P i l (κ i l) by blast
 from someI-ex[OF this] show ?thesis
  unfolding κ-SOME-def using assms by blast
qed

lemma (in gram-schmidt-fs-lin-indpt) fs-i-sumlist-κ:
 assumes i < m l ≤ i j < l
 shows (fs ! i + sumlist (map (λj. κ i l j ·v fs ! j) [0..<l])) • fs ! j = 0
proof −
 have fs ! i + sumlist (map (λj. κ i l j ·v fs ! j) [0..<l])
      = fs ! i − M.sumlist (map (λj. μ i j ·v gso j) [0..<l])
  using assms gso-carrier assms
  by (subst κ-def[symmetric]) (auto simp add: dim-sumlist sumlist-nth sum-negf)
 also have ... = M.sumlist (map (λj. μ i j ·v gso j) [l..<Suc i])
 proof −
  have fs ! i = M.sumlist (map (λj. μ i j ·v gso j) [0..<Suc i])
   using assms by (intro fi-is-sum-of-mu-gso) auto
  also have ... = M.sumlist (map (λj. μ i j ·v gso j) [0..<l]) +
                  M.sumlist (map (λj. μ i j ·v gso j) [l..<Suc i])
  proof −
   have ∗: [0..<Suc i] = [0..<l] @ [l..<Suc i]
        using assms by (metis diff-zero le-imp-less-Suc length-upt list-trisect
```

upt-conv-Cons)
show *?thesis*
by (*subst* ∗, *subst map-append*, *subst sumlist-append*) (*use gso-carrier assms*
in *auto*)
qed
finally show *?thesis*
using *assms gso-carrier assms* **by** (*auto simp add*: *algebra-simps dim-sumlist*)
qed
finally have *fs* ! *i* + *M.sumlist* (*map* (λ*j*. κ *i l j* ·$_v$ *fs* ! *j*) [*0*..<*l*]) =
M.sumlist (*map* (λ*j*. μ *i j* ·$_v$ *gso j*) [*l*..<*Suc i*])
by *simp*
moreover have . . . · (*fs* ! *j*) = *0*
using *assms gso-carrier assms* **unfolding** *lin-indpt-list-def*
by (*subst scalar-prod-left-sum-distrib*)
(*auto simp add*: *algebra-simps dim-sumlist gso-scalar-zero intro*!: *sum-list-zero*)
ultimately show *?thesis* **using** *assms* **by** *auto*
qed

end

lemma *Ints-sum*:
assumes ⋀*a*. *a* ∈ *A* ⟹ *f a* ∈ ℤ
shows *sum f A* ∈ ℤ
using *assms* **by** (*induction A rule*: *infinite-finite-induct*) *auto*

lemma *Ints-prod*:
assumes ⋀*a*. *a* ∈ *A* ⟹ *f a* ∈ ℤ
shows *prod f A* ∈ ℤ
using *assms* **by** (*induction A rule*: *infinite-finite-induct*) *auto*

lemma *Ints-scalar-prod*:
v ∈ *carrier-vec n* ⟹ *w* ∈ *carrier-vec n*
⟹ (⋀ *i*. *i* < *n* ⟹ *v* $ *i* ∈ ℤ) ⟹ (⋀ *i*. *i* < *n* ⟹ *w* $ *i* ∈ ℤ) ⟹ *v* · *w* ∈ ℤ
unfolding *scalar-prod-def* **by** (*intro Ints-sum Ints-mult*, *auto*)

lemma *Ints-det*: **assumes** ⋀ *i j*. *i* < *dim-row A* ⟹ *j* < *dim-col A*
⟹ *A* $$ (*i*,*j*) ∈ ℤ
shows *det A* ∈ ℤ
proof (*cases dim-row A* = *dim-col A*)
case *True*
show *?thesis* **unfolding** *Determinant.det-def* **using** *True assms*
by (*auto intro*!: *Ints-mult Ints-prod*)
next
case *False*
show *?thesis* **unfolding** *Determinant.det-def* **using** *False* **by** *simp*
qed

lemma (**in** *gram-schmidt-fs-Rn*) *Gramian-matrix-alt-alt-def*:
assumes $k \leq m$
shows *Gramian-matrix fs k* = *mat k k* (λ(*i*,*j*). *fs* ! *i* · *fs* ! *j*)
proof −
have ∗: *vec n* ((\$) (*fs* ! *i*)) = *fs* ! *i* **if** $i < m$ **for** *i*
using *that* **by** *auto*
then show *?thesis*
unfolding *Gramian-matrix-def* **using** *assms*
by (*intro eq-matI*) (*auto simp add*: *Let-def*)
qed

lemma (**in** *gram-schmidt-fs-int*) *fs-scalar-Ints*:
assumes $i < m$ $j < m$
shows *fs* ! *i* · *fs* ! *j* ∈ $\mathbb{Z}$
by (*rule Ints-scalar-prod*[*of - n*], *insert fs-int assms*, *auto*)

abbreviation (**in** *gram-schmidt-fs-lin-indpt*) *d* **where** *d* ≡ *Gramian-determinant fs*

lemma (**in** *gram-schmidt-fs-lin-indpt*) *fs-i-fs-j-sum-κ* :
assumes $i < m$ $l \leq i$ $j < l$
shows − (*fs* ! *i* · *fs* ! *j*) = ($\sum$ *t* = *0*..<*l*. *fs* ! *t* · *fs* ! *j* ∗ *κ i l t*)
proof −
have [*simp*]: *M.sumlist* (*map* (λ*j*. *κ i l j* ·$_v$ *fs* ! *j*) [*0*..<*l*]) ∈ *carrier-vec n*
using *assms* **by** (*auto intro*!: *sumlist-carrier simp add*: *dim-sumlist*)
have *0* = (*fs* ! *i* + *M.sumlist* (*map* (λ*j*. *κ i l j* ·$_v$ *fs* ! *j*) [*0*..<*l*])) · *fs* ! *j*
using *fs-i-sumlist-κ assms* **by** *simp*
also have ... = *fs* ! *i* · *fs* ! *j* + *M.sumlist* (*map* (λ*j*. *κ i l j* ·$_v$ *fs* ! *j*) [*0*..<*l*]) · *fs* ! *j*
using *assms* **by** (*subst add-scalar-prod-distrib*[*of - n*]) (*auto*)
also have *M.sumlist* (*map* (λ*j*. *κ i l j* ·$_v$ *fs* ! *j*) [*0*..<*l*]) · *fs* ! *j* =
($\sum$ *v*←*map* (λ*j*. *κ i l j* ·$_v$ *fs* ! *j*) [*0*..<*l*]. *v* · *fs* ! *j*)
using *assms* **by** (*intro scalar-prod-left-sum-distrib*) (*auto*)
also have ... = ($\sum$ *t*←[*0*..<*l*]. (*κ i l t* ·$_v$ *fs* ! *t*) · *fs* ! *j*)
by (*rule arg-cong*[**where** *f*=*sum-list*]) (*auto*)
also have ... = ($\sum$ *t* = *0*..<*l*. (*κ i l t* ·$_v$ *fs* ! *t*) · *fs* ! *j*)
by (*subst interv-sum-list-conv-sum-set-nat*) (*auto*)
also have ... = ($\sum$ *t* = *0*..<*l*. *fs* ! *t* · *fs* ! *j* ∗ *κ i l t*)
using *assms* **by** (*intro sum.cong*) *auto*
finally show *?thesis* **by** (*simp add*: *field-simps*)
qed

lemma (**in** *gram-schmidt-fs-lin-indpt*) *Gramian-matrix-times-κ* :
assumes $i < m$ $l \leq i$
shows *Gramian-matrix fs l* ∗$_v$ (*vec l* (λ*t*. *κ i l t*)) = (*vec l* (λ*j*. − (*fs* ! *i* · *fs* ! *j*)))
proof −
have − (*fs* ! *i* · *fs* ! *j*) = ($\sum$ *t* = *0*..<*l*. *fs* ! *t* · *fs* ! *j* ∗ *κ i l t*) **if** $j < l$ **for** *j*
using *fs-i-fs-j-sum-κ assms that* **by** *simp*

```
  then show ?thesis using assms
    by (subst Gramian-matrix-alt-alt-def) (auto simp add: scalar-prod-def alge-
bra-simps)
qed

lemma (in gram-schmidt-fs-int) d-κ-Ints :
  assumes i < m l ≤ i t < l
  shows d l * κ i l t ∈ ℤ
proof −
  let ?A = Gramian-matrix fs l
  let ?B = replace-col ?A (Gramian-matrix fs l *v vec l (κ i l)) t
  have deteq: d l = det ?A
    unfolding Gramian-determinant-def
    using Gramian-determinant-Ints
    by auto

  have **: Gramian-matrix fs l ∈ carrier-mat l l unfolding Gramian-matrix-def
Let-def  using fs-carrier by auto

  then have  κ i l t * det ?A = det ?B
    using assms fs-carrier cramer-lemma-mat[of ?A l  (vec l (λt. κ i l t)) t]
    by auto

  also have  ... ∈ ℤ
  proof −
    have *: t<l ⟹ (?A *v vec l (κ i l)) $ t ∈ ℤ for t
      using assms
      apply(subst Gramian-matrix-times-κ, force, force)
      using fs-int fs-carrier
      by (auto intro!: fs-scalar-Ints Ints-minus)

    define B where B = ?B

    have Bint: t1<l ⟶ s1 < l ⟶ B $$ (t1,s1) ∈ ℤ for t1 s1
    proof (cases s1 = t)
      case True
      from * ** this show ?thesis
        unfolding replace-col-def B-def
        by auto
    next
      case False
      from * ** Gramian-matrix-def this fs-carrier assms show ?thesis
        unfolding replace-col-def B-def
        by (auto simp: Gramian-matrix-def Let-def scalar-prod-def intro!: Ints-sum
Ints-mult fs-int)
    qed
    have B: B ∈ carrier-mat l l
      using ** replace-col-def unfolding B-def
      by (auto simp: replace-col-def)
```

```
    have det B ∈ ℤ
      using B Bint assms  det-col[of B l]
      by (auto intro!: Ints-sum Ints-mult Ints-prod)
    thus ?thesis unfolding B-def.
  qed
  finally show ?thesis using deteq by (auto simp add: algebra-simps)
qed

lemma (in gram-schmidt-fs-int) d-gso-Ints:
  assumes i < n k < m
  shows (d k ·v (gso k)) $ i ∈ ℤ
proof −
  note d-κ-Ints[intro!]
  then have (d k * κ k k j) * fs ! j $ i ∈ ℤ if j < k for j
    using that fs-int assms by (auto intro: Ints-mult )
  moreover have (d k * κ k k j) * fs ! j $ i = d k * κ k k j * fs ! j $ i for j
    by (auto simp add: field-simps)
  ultimately have d k * (∑j = 0..<k. κ k k j * fs ! j $ i) ∈ ℤ
     by (subst sum-distrib-left) (auto simp add: field-simps intro!: Ints-sum)
  moreover have (gso k) $ i = fs ! k $ i + sum (λj. (κ k k j ·v fs ! j) $ i) {0..<k}
  proof −
    have  i < dim-vec (M.sumlist (map (λj. κ k k j ·v fs ! j) [0..<k]))
      using assms by (subst sumlist-dim) auto
    then show ?thesis
       using assms by (subst gso.simps) (auto simp add: sumlist-nth sumlist-dim
κ-def)
  qed
  ultimately show ?thesis
    using assms
   by (auto simp add: distrib-left Gramian-determinant-Ints fs-int intro!: Ints-mult
Ints-add)
qed

lemma (in gram-schmidt-fs-int) d-mu-Ints:
  assumes l ≤ k k < m
  shows d (Suc l) * μ k l ∈ ℤ
proof (cases l < k)
  case True
  have ll: d l * gso l $ i = (d l ·v gso l) $ i if i < n for i
    using that assms by auto
  have d (Suc l) * μ k l =d (Suc l) * (fs ! k • gso l) / ‖gso l‖²
    using assms True unfolding μ.simps by simp
  also have ... = fs ! k • (d l ·v gso l)
    using assms Gramian-determinant(2)[of Suc l]
    by (subst Gramian-determinant-div[symmetric]) (auto)
  also have ... ∈ ℤ
  proof −
    have d l * gso l $ i ∈ ℤ if i < n for i
      using assms d-gso-Ints that ll by (simp)
```

then show *?thesis*
using *assms* **by** (*auto intro*!: *Ints-sum simp add*: *fs-int scalar-prod-def*)
qed
finally show *?thesis*
by *simp*
next
case *False*
with *assms* **have** *l*: $l = k$ **by** *auto*
show *?thesis* **unfolding** *l* μ*.simps* **using** *Gramian-determinant-Ints fs-int assms*
by *simp*
qed

lemma *max-list-Max*: $ls \neq [] \Longrightarrow$ *max-list ls* = *Max* (*set ls*)
by (*induction ls*) (*auto simp add*: *max-list-Cons*)

8.1 Explicit Bounds for Size of Numbers that Occur During GSO Algorithm

context *gram-schmidt-fs-lin-indpt*
begin

definition N = *Max* (*sq-norm* ‘ *set fs*)

lemma *N-ge-0*:
assumes $0 < m$
shows $0 \leq N$
proof −
have $x \in$ *sq-norm* ‘ *set fs* $\Longrightarrow 0 \leq x$ **for** x
by *auto*
then show *?thesis*
using *assms* **unfolding** *N-def* **by** *auto*
qed

lemma *N-fs*:
assumes $i < m$
shows $\|fs\ !\ i\|^2 \leq N$
using *assms* **unfolding** *N-def* **by** (*auto*)

lemma *N-gso*:
assumes $i < m$
shows $\|gso\ i\|^2 \leq N$
using *assms N-fs sq-norm-gso-le-f* **by** *fastforce*

lemma *N-d*:
assumes $i \leq m$
shows *Gramian-determinant fs i* $\leq N \mathbin{\hat{}} i$

proof −
have $(\prod j{<}i.\ \|gso\ j\|^2) \leq (\prod j{<}i.\ N)$
using *assms N-gso* **by** (*intro prod-mono*) *auto*
then show *?thesis*
using *assms Gramian-determinant* **by** *auto*
qed

end

lemma *ex-MAXIMUM*: **assumes** *finite A A* $\neq \{\}$
shows $\exists\, a \in A.\ Max\ (f\ `\ A) = f\ a$
proof −
have *Max* $(f\ `\ A) \in f\ `\ A$
using *assms* **by** (*auto intro*!: *Max-in*)
then show *?thesis*
using *assms imageE* **by** *blast*
qed

context *gram-schmidt-fs-int*
begin

lemma *fs-int'*: $k < n \Longrightarrow f \in set\ fs \Longrightarrow f\ \$\ k \in \mathbb{Z}$
by (*metis fs-int in-set-conv-nth*)

lemma
assumes $i < m$
shows *fs-sq-norm-Ints*: $\|fs\ !\ i\|^2 \in \mathbb{Z}$ **and** *fs-sq-norm-ge-1*: $1 \leq \|fs\ !\ i\|^2$
proof −
show *fs-Ints*: $\|fs\ !\ i\|^2 \in \mathbb{Z}$
using *assms fs-int' carrier-vecD fs-carrier*
by (*auto simp add*: *sq-norm-vec-as-cscalar-prod scalar-prod-def intro*!: *Ints-sum Ints-mult*)
have $fs\ !\ i \neq 0_v\ n$
using *assms fs-carrier loc-assms nth-mem vs-zero-lin-dep* **by** *force*
then have ∗: $0 \neq \|fs\ !\ i\|^2$
using *assms sq-norm-vec-eq-0 f-carrier* **by** *metis*
show $1 \leq \|fs\ !\ i\|^2$
by (*rule Ints-cases*[*OF fs-Ints*]) (*use* ∗ *sq-norm-vec-ge-0*[*of fs ! i*] *assms* **in** *auto*)
qed

lemma
assumes *set fs* $\neq \{\}$
shows *N-Ints*: $N \in \mathbb{Z}$ **and** *N-1*: $1 \leq N$
proof −
have $\exists\, v_m \in set\ fs.\ N = sq\text{-}norm\ v_m$
unfolding *N-def* **using** *assms* **by** (*auto intro*!: *ex-MAXIMUM*)
then obtain v_m::*'a vec* **where** v_m*-def*: $v_m \in set\ fs\ N = sq\text{-}norm\ v_m$
by *blast*

then show *N-Ints*: $N \in \mathbb{Z}$
using *fs-int' carrier-vecD fs-carrier*
by (*auto simp add*: *sq-norm-vec-as-cscalar-prod scalar-prod-def intro*!: *Ints-sum Ints-mult*)
have $*$: $0 \neq N$
using *N-gso sq-norm-pos assms* **by** *fastforce*
show $1 \leq N$
by (*rule Ints-cases*[*OF N-Ints*]) (*use* $*$ *N-ge-0 assms* **in** *force*)+
qed

lemma *N-mu*:
assumes $i < m\ j \leq i$
shows $(\mu\ i\ j)^2 \leq N \mathbin{\hat{}} (Suc\ j)$
proof −
{ assume *ji*: $j < i$
have $(\mu\ i\ j)^2 \leq$ *Gramian-determinant fs j* $*$ $\|fs\ !\ i\|^2$
using *assms ji* **by** (*intro mu-bound-Gramian-determinant*) *auto*
also have $\ldots \leq N \mathbin{\hat{}} j * \|fs\ !\ i\|^2$
using *assms N-d N-ge-0* **by** (*intro mult-mono*) *fastforce*+
also have $N \mathbin{\hat{}} j * \|fs\ !\ i\|^2 \leq N \mathbin{\hat{}} j * N$
using *assms N-fs N-ge-0* **by** (*intro mult-mono*) *fastforce*+
also have $\ldots = N \mathbin{\hat{}} (Suc\ j)$
by *auto*
finally have *?thesis*
by *simp* **}**
moreover
{ assume *ji*: $j = i$
have $(\mu\ i\ j)^2 = 1$
using *ji* **by** (*simp add*: μ.*simps*)
also have $\ldots \leq N$
using *assms N-1* **by** *fastforce*
also have $\ldots \leq N \mathbin{\hat{}} (Suc\ j)$
using *assms N-1* **by** *fastforce*
finally have *?thesis*
by *simp* **}**
ultimately show *?thesis*
using *assms* **by** *fastforce*
qed

end

lemma *vec-hom-Ints*:
assumes $i < n$ $xs \in$ *carrier-vec n*
shows *of-int-hom.vec-hom xs* \$ $i \in \mathbb{Z}$
using *assms* **by** *auto*

lemma *division-to-div*: (*of-int x* :: *'a* :: *floor-ceiling*) = *of-int y* / *of-int z* $\Longrightarrow$ $x = y$ *div* z

by (*metis floor-divide-of-int-eq floor-of-int*)

lemma *exact-division*: **assumes** *of-int x / (of-int y :: 'a :: floor-ceiling)* $\in \mathbb{Z}$
shows *of-int (x div y) = of-int x / (of-int y :: 'a)*
using *assms* **by** (*metis Ints-cases division-to-div*)

lemma *int-via-rat-eqI*: *rat-of-int x = rat-of-int y* $\Longrightarrow$ *x = y* **by** *auto*

locale *fs-int* =
fixes
n :: *nat* **and**
fs-init :: *int vec list*
begin

sublocale *vec-module TYPE(int) n* .

abbreviation *RAT* **where** *RAT* $\equiv$ *map (map-vec rat-of-int)*
abbreviation (*input*) *m* **where** *m* $\equiv$ *length fs-init*

sublocale *gs*: *gram-schmidt-fs n RAT fs-init* .

definition *d* :: *int vec list* $\Rightarrow$ *nat* $\Rightarrow$ *int* **where** *d fs k = gs.Gramian-determinant fs k*
definition *D* :: *int vec list* $\Rightarrow$ *nat* **where** *D fs = nat* ($\prod$ *i < length fs. d fs i*)

lemma *of-int-Gramian-determinant*:
assumes *k* $\leq$ *length F* $\bigwedge$*i. i < length F* $\Longrightarrow$ *dim-vec (F ! i) = n*
shows *gs.Gramian-determinant (map of-int-hom.vec-hom 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*])
let *?F = map of-int-hom.vec-hom F*
have *cong*: $\bigwedge$ *a b c d. a = b* $\Longrightarrow$ *c = d* $\Longrightarrow$ *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-hom-mult*[*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** *assms* **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
qed

end

locale *fs-int-indpt* = *fs-int n fs* **for** *n fs* +
  assumes *lin-indep*: *gs.lin-indpt-list* (*RAT fs*)
begin

sublocale *gs*: *gram-schmidt-fs-lin-indpt n RAT fs*
  by (*standard*) (*use lin-indep gs.lin-indpt-list-def* **in** *auto*)

sublocale *gs*: *gram-schmidt-fs-int n RAT fs*
  by (*standard*) (*use gs.f-carrier lin-indep gs.lin-indpt-list-def* **in** ‹*auto intro*!: *vec-hom-Ints*›)

lemma *f-carrier*[*dest*]: *i* < *m* ⟹ *fs* ! *i* ∈ *carrier-vec n*
  and *fs-carrier* [*simp*]: *set fs* ⊆ *carrier-vec n*
  using *lin-indep gs.f-carrier gs.gso-carrier* **unfolding** *gs.lin-indpt-list-def* **by** *auto*

lemma *Gramian-determinant*:
  assumes *k*: *k* ≤ *m*
  shows *of-int* (*gs.Gramian-determinant fs k*) = (∏ *j*<*k*. *sq-norm* (*gs.gso j*)) (**is** *?g1*)
    gs.Gramian-determinant fs k > *0* (**is** *?g2*)
proof −
  have *hom*: *gs.Gramian-determinant* (*RAT fs*) *k* = *of-int* (*gs.Gramian-determinant fs k*)
    using *k* **by** (*intro of-int-Gramian-determinant*) *auto*
  show *?g1*
    unfolding *hom*[*symmetric*] **using** *gs.Gramian-determinant assms* **by** *auto*
  show *?g2*
    using *hom gs.Gramian-determinant assms* **by** *fastforce*
qed

lemma *fs-int-d-pos* [*intro*]:
  assumes *k*: *k* ≤ *m*
shows *d fs k* > *0*
  unfolding *d-def* **using** *Gramian-determinant*[*OF k*] **by** *auto*

lemma *fs-int-d-Suc*:
  assumes *k*: *k* < *m*
shows *of-int* (*d fs* (*Suc k*)) = *sq-norm* (*gs.gso k*) ∗ *of-int* (*d fs k*)
proof −
  from *k* **have** *k*: *k* ≤ *m Suc k* ≤ *m* **by** *auto*
  show *?thesis* **unfolding** *Gramian-determinant*[*OF k*(*1*)] *Gramian-determinant*[*OF k*(*2*)] *d-def*
    by (*subst prod.remove*[*of - k*], *force*+, *rule arg-cong*[*of - - λ x. - ∗ x*], *rule prod.cong*, *auto*)
qed

lemma *fs-int-D-pos*:
shows *D fs* > *0*
proof −
  have (∏ *j* < *m*. *d fs j*) > *0*

by (*rule prod-pos*, *insert fs-int-d-pos*, *auto*)
thus *?thesis* **unfolding** *D-def* **by** *auto*
qed

definition $d\mu\ i\ j = int\text{-}of\text{-}rat\ (of\text{-}int\ (d\ fs\ (Suc\ j)) * gs.\mu\ i\ j)$

lemma *fs-int-mu-d-Z*:
assumes *j*: $j \leq ii$ **and** *ii*: $ii < m$
shows $of\text{-}int\ (d\ fs\ (Suc\ j)) * gs.\mu\ ii\ j \in \mathbb{Z}$
proof −
have *id*: $of\text{-}int\ (d\ fs\ (Suc\ j)) = gs.Gramian\text{-}determinant\ (RAT\ fs)\ (Suc\ j)$
unfolding *d-def*
by (*rule of-int-Gramian-determinant*[*symmetric*], *insert j ii* , *auto*)
have *of-int-hom.vec-hom* $(fs\ !\ j)\ \$\ i \in \mathbb{Z}$ **if** $i < n\ j < length\ fs$ **for** *i j*
using *that* **by** (*intro vec-hom-Ints*) *auto*
then show *?thesis*
unfolding *id* **using** *j ii* **unfolding** *gs.lin-indpt-list-def*
by (*intro gs.d-mu-Ints*) (*auto*)
qed

lemma *fs-int-mu-d-Z-m-m*:
assumes *j*: $j < m$ **and** *ii*: $ii < m$
shows $of\text{-}int\ (d\ fs\ (Suc\ j)) * gs.\mu\ ii\ j \in \mathbb{Z}$
proof (*cases* $j \leq ii$)
case *True*
thus *?thesis* **using** *fs-int-mu-d-Z*[*OF True ii*] **by** *auto*
next
case *False* **thus** *?thesis* **by** (*simp add*: *gs.μ.simps*)
qed

lemma *sq-norm-fs-via-sum-mu-gso*: **assumes** *i*: $i < m$
shows $of\text{-}int\ \|fs\ !\ i\|^2 = (\sum j \leftarrow [0..<Suc\ i].\ (gs.\mu\ i\ j)^2 * \|gs.gso\ j\|^2)$
proof −
let *?G* = *map* (*gs.gso*) [*0* ..< *m*]
let *?gso* = $\lambda\ fs\ j.\ ?G\ !\ j$
have $of\text{-}int\ \|fs\ !\ i\|^2 = \|RAT\ fs\ !\ i\|^2$ **unfolding** *sq-norm-of-int*[*symmetric*] **using** *insert i* **by** *auto*
also have $RAT\ fs\ !\ i = gs.sumlist\ (map\ (\lambda j.\ gs.\mu\ i\ j \cdot_v gs.gso\ j)\ [0..<Suc\ i])$
using *gs.fi-is-sum-of-mu-gso i* **by** *auto*
also have *id*: $map\ (\lambda j.\ gs.\mu\ i\ j \cdot_v gs.gso\ j)\ [0..<Suc\ i] = map\ (\lambda j.\ gs.\mu\ i\ j \cdot_v ?gso\ fs\ j)\ [0..<Suc\ i]$
by (*rule nth-equalityI*, *insert i*, *auto simp*: *nth-append*)
also have $sq\text{-}norm\ (gs.sumlist\ \ldots) = sum\text{-}list\ (map\ sq\text{-}norm\ (map\ (\lambda j.\ gs.\mu\ i\ j \cdot_v gs.gso\ j)\ [0..<Suc\ i]))$
unfolding *map-map o-def sq-norm-smult-vec*
unfolding *sq-norm-vec-as-cscalar-prod cscalar-prod-is-scalar-prod conjugate-id*
proof (*subst gs.scalar-prod-lincomb-orthogonal*)
show $Suc\ i \leq length\ ?G$ **using** *i* **by** *auto*

show *set ?G* ⊆ *carrier-vec n* **using** *gs.gso-carrier* **by** *auto*
show *orthogonal ?G* **using** *gs.orthogonal-gso* **by** *auto*
qed (*rule arg-cong*[*of - - sum-list*], *intro nth-equalityI*, *insert i*, *auto simp*: *nth-append*)
also have *map sq-norm* (*map* (λ*j*. *gs.μ i j* ·$_v$ *gs.gso j*) [*0*..<*Suc i*]) = *map* (λ*j*. (*gs.μ i j*)^2 ∗ *sq-norm* (*gs.gso j*)) [*0*..<*Suc i*]
unfolding *map-map o-def sq-norm-smult-vec* **by** (*rule map-cong*, *auto simp*: *power2-eq-square*)
finally show *?thesis* **.**
qed

lemma *dμ*: **assumes** *j* < *m ii* < *m*
shows *of-int* (*dμ ii j*) = *of-int* (*d fs* (*Suc j*)) ∗ *gs.μ ii j*
unfolding *dμ-def* **using** *fs-int-mu-d-Z-m-m assms* **by** *auto*

end

end

8.2 Gram-Schmidt Implementation for Integer Vectors

This theory implements the Gram-Schmidt algorithm on integer vectors using purely integer arithmetic. The formalization is based on [1].

theory *Gram-Schmidt-Int*
imports
Gram-Schmidt-2
More-IArray
begin

context fixes
fs :: *int vec iarray* **and** *m* :: *nat*
begin
fun *sigma-array* **where**
sigma-array dmus dmusi dmusj dll l = (*if l* = *0 then dmusi* !! *l* ∗ *dmusj* !! *l*
else let l1 = *l* − *1*; *dll1* = *dmus* !! *l1* !! *l1 in*
(*dll* ∗ *sigma-array dmus dmusi dmusj dll1 l1* + *dmusi* !! *l* ∗ *dmusj* !! *l*) *div*
dll1)

declare *sigma-array.simps*[*simp del*]

partial-function(*tailrec*) *dmu-array-row-main* **where**
[*code*]: *dmu-array-row-main fi i dmus j* = (*if j* = *i then dmus*
else let sj = *Suc j*;
dmus-i = *dmus* !! *i*;
djj = *dmus* !! *j* !! *j*;
dmu-ij = *djj* ∗ (*fi* · *fs* !! *sj*) − *sigma-array dmus dmus-i* (*dmus* !! *sj*) *djj j*;
dmus′ = *iarray-update dmus i* (*iarray-append dmus-i dmu-ij*)
in dmu-array-row-main fi i dmus′ sj)

definition *dmu-array-row* **where**

```
  dmu-array-row dmus i = (let fi = fs !! i in
      dmu-array-row-main fi i (iarray-append dmus (IArray [fi · fs !! 0])) 0)

partial-function (tailrec) dmu-array where
  [code]: dmu-array dmus i = (if i = m then dmus else
    let dmus' = dmu-array-row dmus i
      in dmu-array dmus' (Suc i))
end

definition dμ-impl :: int vec list ⇒ int iarray iarray where
  dμ-impl fs = dmu-array (IArray fs) (length fs) (IArray []) 0


definition (in gram-schmidt) β where β fs l = Gramian-determinant fs (Suc l)
/ Gramian-determinant fs l

context gram-schmidt-fs-lin-indpt
begin

lemma Gramian-beta:
  assumes i < m
  shows β fs i = ‖fs ! i‖² − (∑ j = 0..<i. (μ i j)² * β fs j)
proof −
  let ?S = M.sumlist (map (λj. − μ i j ·v gso j) [0..<i])
  have S: ?S ∈ carrier-vec n
    using assms by (auto intro!: M.sumlist-carrier gso-carrier)
  have fi: fs ! i ∈ carrier-vec n using assms by auto
  have β fs i = gso i · gso i
    unfolding β-def
   using assms dist by (auto simp add: Gramian-determinant-div sq-norm-vec-as-cscalar-prod)
  also have ... = (fs ! i + ?S) · (fs ! i + ?S)
    by (subst gso.simps, subst (2) gso.simps) auto
  also have ... = fs ! i · fs ! i + ?S · fs ! i + fs ! i · ?S + ?S · ?S
   using assms S by (auto simp add: add-scalar-prod-distrib[of - n] scalar-prod-add-distrib[of
- n])
  also have fs ! i · ?S = ?S · fs ! i
    by (rule comm-scalar-prod[OF fi S])
  also have ?S · fs ! i = ?S · gso i − ?S · ?S
  proof −
    have fs ! i = gso i − M.sumlist (map (λj. − μ i j ·v gso j) [0..<i])
       using assms S by (subst gso.simps) auto
    then show ?thesis
    using assms S by (auto simp add: minus-scalar-prod-distrib[of - n] scalar-prod-minus-distrib[of
- n])
  qed
  also have ?S · gso i = 0
    using assms orthogonal
    by(subst scalar-prod-left-sum-distrib)
      (auto intro!: sum-list-neutral M.sumlist-carrier gso-carrier)
```

```
  also have ?S • ?S = (∑ j = 0..<i. (μ i j)^2 * (gso j • gso j))
    using assms dist by (subst scalar-prod-lincomb-gso)
      (auto simp add: power2-eq-square interv-sum-list-conv-sum-set-nat)
  also have ... = (∑ j = 0..<i. (μ i j)^2 * β fs j)
    using assms dist
    by (auto simp add: β-def Gramian-determinant-div sq-norm-vec-as-cscalar-prod
        intro!: sum.cong)
  finally show ?thesis
    by (auto simp add: sq-norm-vec-as-cscalar-prod)
qed

lemma gso-norm-beta:
  assumes j < m
  shows β fs j = sq-norm (gso j)
  unfolding β-def
 using assms dist by (auto simp add: Gramian-determinant-div sq-norm-vec-as-cscalar-prod)

lemma mu-Gramian-beta-def:
  assumes j < i i < m
  shows μ i j = (fs ! i • fs ! j − (∑ k = 0..<j. μ j k * μ i k * β fs k)) / β fs j
proof −
  let ?list = map (λja. μ i ja ·_v gso ja) [0..<i]
  let ?neg-sum = M.sumlist (map (λja. − μ j ja ·_v gso ja) [0..<j])
  have list: set ?list ⊆ carrier-vec n using gso-carrier assms by auto
  define fi where fi = fs ! i
  have list-id: [0..<i] = [0..<j] @ [j..<i]
    using assms by (metis append.simps(1) neq0-conv upt.simps(1) upt-append)
  have μ i j = (fs ! i) • (gso j) / sq-norm (gso j)
    unfolding μ.simps using assms by auto
  also have  ... = fs ! i • (fs ! j + ?neg-sum) / sq-norm (gso j)
    by (subst gso.simps, simp)
  also have  ... = (fi • fs ! j + fs ! i • ?neg-sum) / sq-norm (gso j)
    using assms unfolding fi-def
   by (subst scalar-prod-add-distrib [of - n]) (auto intro!: M.sumlist-carrier gso-carrier)
  also have fs ! i = gso i + M.sumlist ?list
    by (rule fs-by-gso-def[OF assms(2)])
  also have ... • ?neg-sum = gso i • ?neg-sum + M.sumlist ?list • ?neg-sum
   using assms by (subst add-scalar-prod-distrib [of - n]) (auto intro!: M.sumlist-carrier
gso-carrier)
  also have  M.sumlist ?list = M.sumlist (map (λja. μ i ja ·_v gso ja) [0..<j])
     + M.sumlist (map (λja. μ i ja ·_v gso ja) [j..<i])  (is - = ?sumj + ?sumi)
    unfolding list-id
    by (subst M.sumlist-append[symmetric], insert gso-carrier assms, auto)
  also have gso i • ?neg-sum = 0
    by (rule orthogonal-sumlist, insert gso-carrier dist assms orthogonal, auto)
 also have  (?sumj + ?sumi) • ?neg-sum = ?sumj • ?neg-sum + ?sumi • ?neg-sum
    using assms
   by (subst add-scalar-prod-distrib [of - n], auto intro!: M.sumlist-carrier gso-carrier)
  also have  ?sumj • ?neg-sum = (∑ l = 0..<j. (μ i l) * (−μ j l) * (gso l • gso
```

l))
  using *assms*
 by (*subst scalar-prod-lincomb-gso*) (*auto simp add*: *interv-sum-list-conv-sum-set-nat*)
 also have ... = − (∑ *l* = *0*..<*j*. (*μ i l*) ∗ (*μ j l*) ∗ (*gso l* · *gso l*)) (**is** - = − *?sum*)
  by (*auto simp add*: *sum-negf*)
 also have *?sum* = (∑ *l* = *0*..<*j*. (*μ j l*) ∗ (*μ i l*) ∗ *β fs l*)
  using *assms*
  by (*intro sum.cong*, *auto simp*: *gso-norm-beta sq-norm-vec-as-cscalar-prod*)
 also have *?sumi* · *?neg-sum* = *0*
  apply (*rule orthogonal-sumlist*, *insert gso-carrier assms orthogonal*, *auto intro*!: *M.sumlist-carrier gso-carrier*)
  apply (*subst comm-scalar-prod*[*of - n*], *auto intro*!: *M.sumlist-carrier*)
  by (*rule orthogonal-sumlist*, *use dist* **in** *auto*)
 also have *sq-norm* (*gso j*) = *β fs j*
  using *assms*
  by (*subst gso-norm-beta*, *auto*)
 finally show *?thesis* **unfolding** *fi-def* **by** *simp*
qed

end

lemma (**in** *gram-schmidt*) *Gramian-matrix-alt-alt-alt-def*:
 assumes *k* ≤ *length fs set fs* ⊆ *carrier-vec n*
 shows *Gramian-matrix fs k* = *mat k k* (λ(*i*,*j*). *fs* ! *i* · *fs* ! *j*)
proof −
 have ∗: *vec n* (($) (*fs* ! *i*)) = *fs* ! *i* **if** *i* < *length fs* **for** *i*
  using *that assms*
  by (*metis carrier-vecD dim-vec eq-vecI index-vec nth-mem subsetCE*)
 then show *?thesis*
  unfolding *Gramian-matrix-def* **using** *assms*
  by (*intro eq-matI*) (*auto simp add*: *Let-def*)
qed

lemma (**in** *gram-schmidt-fs-Rn*) *Gramian-determinant-1* [*simp*]:
 assumes *0* < *length fs*
 shows *Gramian-determinant fs* (*Suc 0*) = ‖*fs* ! *0*‖²
proof −
 have *Gramian-determinant fs* (*Suc 0*) = *fs* ! *0* · *fs* ! *0*
  using *assms* **unfolding** *Gramian-determinant-def*
  by (*subst det-def'*) (*auto simp add*: *Gramian-matrix-def Let-def scalar-prod-def*)
 then show *?thesis*
  by (*subst sq-norm-vec-as-cscalar-prod*) *simp*
qed

context *gram-schmidt-fs-lin-indpt*
begin

definition μ' **where** $\mu'\ i\ j \equiv d\ (Suc\ j) * \mu\ i\ j$

fun σ **where**
  $\sigma\ 0\ i\ j = 0$
| $\sigma\ (Suc\ l)\ i\ j = (d\ (Suc\ l) * \sigma\ l\ i\ j + \mu'\ i\ l * \mu'\ j\ l)\ /\ d\ l$

lemma *d-Suc*: $d\ (Suc\ i) = \mu'\ i\ i$ **unfolding** μ'*-def* **by** (*simp add*: μ*.simps*)
lemma *d-0*: $d\ 0 = 1$ **by** (*rule Gramian-determinant-0*)

lemma σ: **assumes** *lj*: $l \leq m$
  shows $\sigma\ l\ i\ j = d\ l * (\sum k < l.\ \mu\ i\ k * \mu\ j\ k * \beta\ fs\ k)$
  using *lj*
proof (*induct l*)
  case (*Suc l*)
  from *Suc*(2−) **have** *lj*: $l \leq m$ **by** *auto*
  note *IH* = *Suc*(1)[*OF lj*]
  let *?f* = $\lambda\ k.\ \mu\ i\ k * \mu\ j\ k * \beta\ fs\ k$
  have *dl0*: $d\ l > 0$ **using** *lj Gramian-determinant dist* **unfolding** *lin-indpt-list-def*
by *auto*
  have $\sigma\ (Suc\ l)\ i\ j = (d\ (Suc\ l) * \sigma\ l\ i\ j + \mu'\ i\ l * \mu'\ j\ l)\ /\ d\ l$ **by** *simp*
  also have $\ldots = (d\ (Suc\ l) * \sigma\ l\ i\ j)\ /\ d\ l + (\mu'\ i\ l * \mu'\ j\ l)\ /\ d\ l$ **using** *dl0*
    by (*simp add*: *field-simps*)
  also have $(\mu'\ i\ l * \mu'\ j\ l)\ /\ d\ l = d\ (Suc\ l) *$ *?f l* (**is** - = *?one*)
    unfolding β*-def* μ'*-def* **by** *auto*
  also have $(d\ (Suc\ l) * \sigma\ l\ i\ j)\ /\ d\ l = d\ (Suc\ l) * (\sum k < l.$ *?f k*) (**is** - = *?sum*)
    using *dl0* **unfolding** *IH* **by** *simp*
  also have *?sum* + *?one* = $d\ (Suc\ l) * (?f\ l + (\sum k < l.$ *?f k*)) **by** (*simp add*:
field-simps)
  also have *?f l* + $(\sum k < l.$ *?f k*) = $(\sum k < Suc\ l.$ *?f k*) **by** *simp*
  finally show *?case* **.**
qed *auto*

lemma μ': **assumes** *j*: $j \leq i$ **and** *i*: $i < m$
  shows $\mu'\ i\ j = d\ j * (fs\ !\ i \cdot fs\ !\ j) - \sigma\ j\ i\ j$
proof (*cases* $j < i$)
  case *j*: *True*
  have *dsj*: $d\ (Suc\ j) > 0$
    using *j i Gramian-determinant dist* **unfolding** *lin-indpt-list-def*
    by (*meson less-trans-Suc nat-less-le*)
  let *?sum* = $(\sum k = 0..<j.\ \mu\ j\ k * \mu\ i\ k * \beta\ fs\ k)$
  have $\mu'\ i\ j = (fs\ !\ i \cdot fs\ !\ j -$ *?sum*$) * (d\ (Suc\ j)\ /\ \beta\ fs\ j)$
    unfolding *mu-Gramian-beta-def*[*OF j i*] μ'*-def* **by** *simp*
  also have $d\ (Suc\ j)\ /\ \beta\ fs\ j = d\ j$ **unfolding** β*-def* **using** *dsj* **by** *auto*
  also have $(fs\ !\ i \cdot fs\ !\ j -$ *?sum*$) * d\ j = (fs\ !\ i \cdot fs\ !\ j) * d\ j - d\ j *$ *?sum*
    by (*simp add*: *ring-distribs*)
  also have $d\ j *$ *?sum* = $\sigma\ j\ i\ j$
    by (*subst* σ, (*insert j i*, *force*), *intro arg-cong*[*of* - - $\lambda\ x.$ - $* x$] *sum.cong*, *auto*)

```
  finally show ?thesis by simp
next
  case False
  with j have j: j = i by auto
  have dsi: d (Suc i) > 0 d i > 0
    using i Suc-leI dist  unfolding lin-indpt-list-def
    by (simp-all add: Suc-leI Gramian-determinant(2))
  let ?sum =  (∑ k = 0..<i. μ i k * μ i k * β fs k)
  have bzero: β fs i ≠ 0 unfolding β-def using dsi by auto
  have μ' i i = d (Suc i) by (simp add: μ.simps μ'-def)
  also have ... = β fs i * (d (Suc i)  / β fs i) using bzero by simp
  also have d (Suc i) / β fs i = d i unfolding β-def using dsi by auto
  also have β fs i = (fs ! i • fs ! i − ?sum)
    unfolding Gramian-beta[OF i]
    by (rule arg-cong2[of - - - - (−), OF - sum.cong],
        auto simp: power2-eq-square sq-norm-vec-as-cscalar-prod)
  also have (fs ! i • fs ! i − ?sum) * d i = (fs ! i • fs ! i) * d i − d i * ?sum
    by (simp add: ring-distribs)
  also have d i * ?sum = σ i i i
    by (subst σ, (insert i i, force), intro arg-cong[of - - λ x. - * x] sum.cong, auto)
  finally show ?thesis using j by simp
qed

lemma σ-via-μ': σ (Suc l) i j =
  (if l = 0 then μ' i 0 * μ' j 0 else (μ' l l * σ l i j + μ' i l * μ' j l) / μ' (l − 1) (l
− 1))
  by (cases l, auto simp: d-Suc)

lemma μ'-via-σ: assumes j: j ≤ i and i: i < m
  shows μ' i j =
    (if j = 0 then fs ! i • fs ! j else μ' (j − 1) (j − 1) * (fs ! i • fs ! j) − σ j i j)
  unfolding μ'[OF assms] by (cases j, auto simp: d-Suc)

lemma fs-i-sumlist-κ:
  assumes i < m l ≤ i j < l
  shows (fs ! i + sumlist (map (λj. κ i l j ·v fs ! j) [0..<l])) • fs ! j = 0
proof −
  have fs ! i + sumlist (map (λj. κ i l j ·v fs ! j) [0..<l])
        = fs ! i − M.sumlist (map (λj. μ i j ·v gso j) [0..<l])
    using assms gso-carrier assms
    by (subst κ-def[symmetric]) (auto simp add: dim-sumlist sumlist-nth sum-negf)
  also have ... = M.sumlist (map (λj. μ i j ·v gso j) [l..<Suc i])
  proof −
    have fs ! i = M.sumlist (map (λj. μ i j ·v gso j) [0..<Suc i])
      using assms by (intro fi-is-sum-of-mu-gso) auto
    also have ... = M.sumlist (map (λj. μ i j ·v gso j) [0..<l]) +
                    M.sumlist (map (λj. μ i j ·v gso j) [l..<Suc i])
    proof −
      have *: [0..<Suc i] = [0..<l] @ [l..<Suc i]
```

using *assms* **by** (*metis diff-zero le-imp-less-Suc length-upt list-trisect upt-conv-Cons*)
show *?thesis*
by (*subst* ∗, *subst map-append*, *subst sumlist-append*) (*use gso-carrier assms* **in** *auto*)
qed
finally show *?thesis*
using *assms gso-carrier assms* **by** (*auto simp add*: *algebra-simps dim-sumlist*)
qed
finally have *fs* ! *i* + *M.sumlist* (*map* (λ*j*. κ *i l j* ·$_v$ *fs* ! *j*) [*0*..<*l*]) =
M.sumlist (*map* (λ*j*. μ *i j* ·$_v$ *gso j*) [*l*..<*Suc i*])
by *simp*
moreover have ... · (*fs* ! *j*) = *0*
using *assms gso-carrier assms* **unfolding** *lin-indpt-list-def*
by (*subst scalar-prod-left-sum-distrib*)
(*auto simp add*: *algebra-simps dim-sumlist gso-scalar-zero intro*!: *sum-list-zero*)
ultimately show *?thesis* **using** *assms* **by** *auto*
qed

end

context *gram-schmidt-fs-int*
begin

lemma *β-pos* : *i* < *m* ⟹ *β fs i* > *0*
using *Gramian-determinant*(*2*) **unfolding** *lin-indpt-list-def β-def* **by** *auto*

lemma *β-zero* : *i* < *m* ⟹ *β fs i* ≠ *0*
using *β-pos*[*of i*] **by** *simp*

lemma *σ-integer*:
assumes *l*: *l* ≤ *j* **and** *j*: *j* ≤ *i* **and** *i*: *i* < *m*
shows *σ l i j* ∈ ℤ
proof −
from *assms* **have** *ll*: *l* ≤ *m* **by** *auto*
have *fs-carr*: *j* < *m* ⟹ *fs* ! *j* ∈ *carrier-vec n* **for** *j* **using** *assms fs-carrier* **unfolding** *set-conv-nth* **by** *force*
with *assms* **have** *fs-carr-j*: *fs* ! *j* ∈ *carrier-vec n* **by** *auto*
have *dim-gso*: *i* < *m* ⟹ *dim-vec* (*gso i*) = *n* **for** *i* **using** *gso-carrier* **by** *auto*
have *dim-fs*: *k* < *m* ⟹ *dim-vec* (*fs* ! *k*) = *n* **for** *k* **using** *smult-carrier-vec fs-carr* **by** *auto*
have *i-l-m*: *i* < *l* ⟹ *i* < *m* **for** *i* **using** *assms* **by** *auto*
have *smult*: ⋀ *i j* . *j* < *n* ⟹ *i* < *l* ⟹ (*c* ·$_v$ *fs* ! *i*) $ *j* = *c* ∗ (*fs* ! *i* $ *j*) **for** *c*
using *i-l-m dim-fs* **by** *auto*
have *σ l i j* = *d l* ∗ (∑ *k* < *l*. μ *i k* ∗ μ *j k* ∗ *β fs k*)
unfolding *σ*[*OF ll*] **by** *simp*
also have ... = *d l* ∗ (∑ *k* < *l*. μ *i k* ∗ ((*fs* ! *j*) · (*gso k*) / *sq-norm* (*gso k*)) ∗

```
β fs k) (is - = - * ?sum)
    unfolding μ.simps using assms by auto
  also have ?sum =  (∑ k < l. μ i k * ((fs ! j) • (gso k) /  β fs k) * β fs k)
    using assms by (auto simp add: gso-norm-beta[symmetric] intro!: sum.cong)

  also have ... = (∑ k < l. μ i k * ((fs ! j) • (gso k) ))
    using β-zero assms by (auto intro!: sum.cong)

  also have  ... = (fs ! j) • M.sumlist (map (λk. (μ i k) ·v (gso k)) [0..<l] )
    using assms fs-carr[of j] gso-carrier
   by (subst scalar-prod-right-sum-distrib) (auto intro!: gso-carrier fs-carr sum.cong
simp: sum-list-sum-nth)

  also have d l * ... = (fs ! j) • (d l ·v M.sumlist (map (λk. (μ i k) ·v (gso k))
[0..<l])) (is - = - • (- ·v ?sum2))
    apply (rule scalar-prod-smult-distrib[symmetric])
     apply (rule fs-carr)
    using assms gso-carrier
    by (auto intro!: sumlist-carrier)

   also have ?sum2 = − sumlist (map (λk. (− μ i k) ·v (gso k)) [0..<l])
     apply(rule eq-vecI)
     using fs-carr gso-carrier assms i-l-m
      by(auto simp: sum-negf[symmetric] dim-sumlist sumlist-nth dim-gso intro!:
sum.cong)

   also have ... = − sumlist (map (λk. κ i l k ·v fs ! k) [0..<l])
     using assms gso-carrier assms
     apply (subst κ-def)
     by (auto)

   also have (d l ·v − sumlist (map (λk. κ i l k ·v fs ! k) [0..<l])) =
       (− sumlist (map (λk. (d l * κ i l k) ·v fs ! k) [0..<l]))
     apply(rule eq-vecI)
     using fs-carr smult-carrier-vec dim-fs
     using dim-fs i-l-m
    by (auto simp: smult dim-sumlist sumlist-nth sum-distrib-left intro!: sum.cong)

   finally have id:  σ l i j = fs ! j • − M.sumlist (map (λk. d l * κ i l k ·v fs ! k)
[0..<l])  .

   show σ l i j ∈ ℤ unfolding id
     using i-l-m fs-carr assms fs-int d-κ-Ints
     by (auto simp: dim-sumlist sumlist-nth smult
           intro!: sumlist-carrier Ints-minus Ints-sum Ints-mult[of - fs ! - $ -]
Ints-scalar-prod[OF fs-carr])
 qed

end
```

context *fs-int-indpt*
begin

fun σs **and** μ' **where**
  $\sigma s\ 0\ i\ j = \mu'\ i\ 0 * \mu'\ j\ 0$
| $\sigma s\ (Suc\ l)\ i\ j = (\mu'\ (Suc\ l)\ (Suc\ l) * \sigma s\ l\ i\ j + \mu'\ i\ (Suc\ l) * \mu'\ j\ (Suc\ l))\ div\ \mu'\ l\ l$
| $\mu'\ i\ j = (if\ j = 0\ then\ fs\ !\ i \cdot fs\ !\ j\ else\ \mu'\ (j - 1)\ (j - 1) * (fs\ !\ i \cdot fs\ !\ j) - \sigma s\ (j - 1)\ i\ j)$

declare $\mu'.simps[simp\ del]$

lemma σs-μ': $l < j \Longrightarrow j \leq i \Longrightarrow i < m \Longrightarrow of\text{-}int\ (\sigma s\ l\ i\ j) = gs.\sigma\ (Suc\ l)\ i\ j$
  $i < m \Longrightarrow j \leq i \Longrightarrow of\text{-}int\ (\mu'\ i\ j) = gs.\mu'\ i\ j$
proof (*induct l i j* **and** *i j rule*: σs-$\mu'.induct$)
  case (*1 i j*)
  thus *?case* **by** (*simp add*: *gs.σ.simps*)
next
  case (*2 l i j*)
  have $gs.\sigma(Suc\ (Suc\ l))\ i\ j \in \mathbb{Z}$
    by (*rule gs.σ-integer*, *insert 2 gs.fs-carrier*, *auto*)
  then have $rat\text{-}of\text{-}int\ (\mu'\ (Suc\ l)\ (Suc\ l) * \sigma s\ l\ i\ j + \mu'\ i\ (Suc\ l) * \mu'\ j\ (Suc\ l)) / rat\text{-}of\text{-}int\ (\mu'\ l\ l) \in \mathbb{Z}$
    using *2 gs.d-Suc* **by** (*auto*)
  then have $rat\text{-}of\text{-}int\ (\sigma s\ (Suc\ l)\ i\ j) =$
        $of\text{-}int\ (\mu'\ (Suc\ l)\ (Suc\ l) * \sigma s\ l\ i\ j + \mu'\ i\ (Suc\ l) * \mu'\ j\ (Suc\ l)) / of\text{-}int\ (\mu'\ l\ l)$
    by (*subst σs.simps*, *subst exact-division*) *auto*
  also have $\ldots = gs.\sigma\ (Suc\ (Suc\ l))\ i\ j$
    using *2 gs.d-Suc* **by** (*auto*)
  finally show *?case*
    by *simp*
next
  case (*3 i j*)
  have $dim\text{-}vec\ (fs\ !\ j) = dim\text{-}vec\ (fs\ !\ i)$
    using *3 f-carrier*[*of i*] *f-carrier*[*of j*] *carrier-vec-def* **by** *auto*
  then have *of-int-hom.vec-hom* $(fs\ !\ i)\ \$\ k = rat\text{-}of\text{-}int\ (fs\ !\ i\ \$\ k)$ **if** $k < dim\text{-}vec\ (fs\ !\ j)$ **for** k
    using *that* **by** *simp*
  then have $*$: *of-int-hom.vec-hom* $(fs\ !\ i) \cdot$ *of-int-hom.vec-hom* $(fs\ !\ j) = rat\text{-}of\text{-}int\ (fs\ !\ i \cdot fs\ !\ j)$
    using *3* **by** (*auto simp add*: *scalar-prod-def*)
  show *?case*
  proof (*cases* $j = 0$)
    case *True*
    have $dim\text{-}vec\ (fs\ !\ 0) = dim\text{-}vec\ (fs\ !\ i)$

```
    using 3 f-carrier[of i] f-carrier[of 0] carrier-vec-def by fastforce
   then have 1: of-int-hom.vec-hom (fs ! i) $ k = rat-of-int (fs ! i $ k) if k <
dim-vec (fs ! 0) for k
    using that by simp
   have (μ' i j) = fs ! i · fs ! j
    using True by (simp add: μ'.simps)
   also note *[symmetric]
   also have  of-int-hom.vec-hom (fs ! j) = map of-int-hom.vec-hom fs ! j
    using 3 by auto
   finally show ?thesis
    using 3 True by (subst gs.μ'-via-σ) (auto)
  next
   case False
   then have gs.μ' i j = gs.μ' (j − Suc 0) (j − Suc 0) * (rat-of-int (fs ! i · fs !
j)) − gs.σ j i j
    using * False 3 by (subst gs.μ'-via-σ) (auto)
   then show ?thesis
    using False 3 by (subst μ'.simps) (auto)
 qed
qed


lemma μ': assumes i < m j ≤ i
  shows μ' i j = dμ i j
   j = i ⟹ μ' i j = d fs (Suc i)
proof −
  let ?r = rat-of-int
  from assms have j < m by auto
  note dμ = dμ[OF this assms(1)]
  have ?r (μ' i j) = gs.μ' i j
   using σs-μ' assms by auto
  also have ... = ?r (dμ i j)
   unfolding gs.μ'-def dμ
   by (subst of-int-Gramian-determinant, insert assms fs-carrier, auto simp: d-def
subset-eq)
  finally show 1: μ' i j = dμ i j
   by simp
  assume j: j = i
  have ?r (μ' i j) = ?r (dμ i j)
   unfolding 1 ..
  also have ... = ?r (d fs (Suc i))
   unfolding dμ unfolding j by (simp add: gs.μ.simps)
  finally show μ' i j = d fs (Suc i)
   by simp
qed

lemma sigma-array: assumes mm: mm ≤ m and j: j < mm
  shows l ≤ j ⟹ sigma-array (IArray.of-fun (λi. IArray.of-fun (μ' i) (if i = mm
then Suc j else Suc i)) (Suc mm))
```

```
      (IArray.of-fun (μ' mm) (Suc j)) (IArray.of-fun (μ' (Suc j)) (if Suc j = mm
then Suc j else Suc (Suc j))) (μ' l l) l =
    σs l mm (Suc j)
proof (induct l)
  case 0
  show ?case unfolding σs.simps sigma-array.simps[of - - - - 0]
    using mm j by (auto simp: nth-append)
next
  case (Suc l)
  hence l: l < j l ≤ j by auto
  have id: (Suc l = 0) = False Suc l − 1 = l by auto
  have ineq: Suc l < Suc mm l < Suc mm
    Suc l < (if Suc l = mm then Suc j else Suc (Suc l))
    Suc l < (if Suc j = mm then Suc j else Suc (Suc j))
    l < (if l = mm then Suc j else Suc l)
    Suc l < Suc j
    using mm l j by auto
  note IH = Suc(1)[OF l(2)]
  show ?case unfolding sigma-array.simps[of - - - - Suc l] id if-False Let-def IH
     of-fun-nth[OF ineq(1)] of-fun-nth[OF ineq(2)] of-fun-nth[OF ineq(3)]
     of-fun-nth[OF ineq(4)] of-fun-nth[OF ineq(5)] of-fun-nth[OF ineq(6)]
   unfolding σs.simps by simp
qed

lemma dmu-array-row-main: assumes mm: mm ≤ m shows
  j ≤ mm ⟹ dmu-array-row-main (IArray fs) (IArray fs !! mm) mm
    (IArray.of-fun (λi. IArray.of-fun (μ' i) (if i = mm then Suc j else Suc i)) (Suc
mm))
      j = IArray.of-fun (λi. IArray.of-fun (μ' i) (Suc i)) (Suc mm)
proof (induct mm − j arbitrary: j)
  case 0
  thus ?case unfolding dmu-array-row-main.simps[of - - - - j] by simp
next
  case (Suc x j)
  hence prems: x = mm − Suc j Suc j ≤ mm and j: j < mm by auto
  note IH = Suc(1)[OF prems]
  have id: (j = mm) = False (mm = mm) = True using Suc(2−) by auto
  have id2: IArray.of-fun (μ' mm) (Suc j) = IArray (map (μ' mm) [0..<Suc j])
    by simp
  have id3: IArray fs !! mm = fs ! mm IArray fs !! Suc j = fs ! Suc j by auto
  have le: j < Suc j Suc j < Suc mm mm < Suc mm j < Suc mm
    j < (if j = mm then Suc j else Suc j) using j by auto
  show ?case unfolding dmu-array-row-main.simps[of - - - - j]
      IH[symmetric] Let-def id if-True if-False id3
      of-fun-nth[OF le(1)] of-fun-nth[OF le(2)]
      of-fun-nth[OF le(3)] of-fun-nth[OF le(4)]
      of-fun-nth[OF le(5)]
      sigma-array[OF mm j le-refl, folded id2]
      iarray-length-of-fun iarray-update-of-fun iarray-append-of-fun
```

proof (*rule arg-cong*[*of* - - λ *x*. *dmu-array-row-main* - - - *x* -], *rule iarray-cong*′, *goal-cases*)
case (*1 i*)
show *?case* **unfolding** *of-fun-nth*[*OF 1*] **using** *j 1*
by (*cases i* = *mm*, *auto simp*: μ′.*simps*[*of* - *Suc j*])
qed
qed

lemma *dmu-array-row*: **assumes** *mm*: *mm* ≤ *m* **shows**
dmu-array-row (*IArray fs*) (*IArray.of-fun* (λ*i*. *IArray.of-fun* (μ′ *i*) (*Suc i*)) *mm*) *mm* =
IArray.of-fun (λ*i*. *IArray.of-fun* (μ′ *i*) (*Suc i*)) (*Suc mm*)
proof −
have *0*: *0* ≤ *mm* **by** *auto*
show *?thesis* **unfolding** *dmu-array-row-def Let-def dmu-array-row-main*[*OF assms 0*, *symmetric*]
unfolding *iarray-append.simps IArray.of-fun-def id map-append list.simps*
by (*rule arg-cong*[*of* - - λ *x*. *dmu-array-row-main* - - - (*IArray x*) -], *rule nth-equalityI*,
auto simp: *nth-append* μ′.*simps*[*of* - *0*])
qed

lemma *dmu-array*: **assumes** *mm* ≤ *m*
shows *dmu-array* (*IArray fs*) *m* (*IArray.of-fun* (λ *i*. *IArray.of-fun* (λ *j*. μ′ *i j*) (*Suc i*)) *mm*) *mm*
= *IArray.of-fun* (λ *i*. *IArray.of-fun* (λ *j*. μ′ *i j*) (*Suc i*)) *m*
using *assms*
proof (*induct mm rule*: *wf-induct*[*OF wf-measure*[*of* λ *mm*. *m* − *mm*]])
case (*1 mm*)
show *?case*
proof (*cases mm* = *m*)
case *True*
thus *?thesis* **unfolding** *dmu-array.simps*[*of* - - - *mm*] **by** *simp*
next
case *False*
with *1*(*2*−)
have *mm*: *mm* ≤ *m* **and** *id*: (*Suc mm* = *0*) = *False Suc mm* − *1* = *mm* (*mm* = *m*) = *False*
and *prems*: (*Suc mm*, *mm*) ∈ *measure* ((−) *m*) *Suc mm* ≤ *m* **by** *auto*
have *list*: [*0*..<*Suc mm*] = [*0*..< *mm*] @ [*mm*] **by** *auto*
note *IH* = *1*(*1*)[*rule-format*, *OF prems*]
show *?thesis* **unfolding** *dmu-array.simps*[*of* - - - *mm*] *id if-False Let-def*
unfolding *dmu-array-row*[*OF mm*] *IH*[*symmetric*]
by (*rule arg-cong*[*of* - - λ *x*. *dmu-array* - - *x* -], *rule iarray-cong*, *auto*)
qed
qed

lemma *dμ-impl*: *dμ-impl fs* = *IArray.of-fun* (λ *i*. *IArray.of-fun* (λ *j*. *dμ i j*) (*Suc i*)) *m*

unfolding *dμ-impl-def* **using** *dmu-array*[*of 0*] **by** (*auto simp*: μ')

end

context *gram-schmidt-fs-int*
begin

lemma *N-μ'*:
assumes $i < m\ j \leq i$
shows $(\mu'\ i\ j)^2 \leq N \mathbin{\hat{}} (3 * Suc\ j)$
proof −
have *1*: $1 \leq N * N \mathbin{\hat{}} j$
using *assms N-1 one-le-power*[*of - Suc j*] **by** *fastforce*
have $0 < d\ (Suc\ j)$
using *assms* **by** (*intro Gramian-determinant*) *auto*
then have [*simp*]: $0 \leq d\ (Suc\ j)$
by *arith*
have *N-d*: $d\ (Suc\ j) \leq N \mathbin{\hat{}} (Suc\ j)$
using *assms* **by** (*intro N-d*) *auto*
have $(\mu'\ i\ j)^2 = (d\ (Suc\ j)) * (d\ (Suc\ j)) * (\mu\ i\ j)^2$
unfolding *μ'-def* **by** (*auto simp add*: *power2-eq-square*)
also have $\ldots \leq (d\ (Suc\ j)) * (d\ (Suc\ j)) * N \mathbin{\hat{}} (Suc\ j)$
proof −
have $(\mu\ i\ j)^2 \leq N \mathbin{\hat{}} (Suc\ j)$ **if** $i = j$
using *that 1* **by** (*auto simp add*: *μ.simps*)
moreover have $(\mu\ i\ j)^2 \leq N \mathbin{\hat{}} (Suc\ j)$ **if** $i \neq j$
using *N-mu assms that* **by** (*auto*)
ultimately have $(\mu\ i\ j)^2 \leq N \mathbin{\hat{}} (Suc\ j)$
by *fastforce*
then show *?thesis*
by (*intro mult-mono*[*of - - (μ i j)*^{2}]) (*auto*)
qed
also have $\ldots \leq N \mathbin{\hat{}} (Suc\ j) * N \mathbin{\hat{}} (Suc\ j) * N \mathbin{\hat{}} (Suc\ j)$
using *assms 1 N-d* **by** (*auto intro*!: *mult-mono*)
also have $N \mathbin{\hat{}} (Suc\ j) * N \mathbin{\hat{}} (Suc\ j) * N \mathbin{\hat{}} (Suc\ j) = N \mathbin{\hat{}} (3 * (Suc\ j))$
using *nat-pow-distrib nat-pow-pow power3-eq-cube* **by** *metis*
finally show *?thesis*
by *simp*
qed

lemma *N-σ*:
assumes $i < m\ j \leq i\ l \leq j$
shows $|\sigma\ l\ i\ j| \leq of\text{-}nat\ l * N \mathbin{\hat{}} (2 * l + 2)$
proof −
have *1*: $|d\ l| = d\ l$
using *Gramian-determinant*(*2*) *assms* **by** (*intro abs-of-pos*) *auto*
then have $|\sigma\ l\ i\ j| = d\ l * |\sum k{<}l.\ \mu\ i\ k * \mu\ j\ k * \beta\ fs\ k|$
using *assms* **by** (*subst σ*, *fastforce*, *subst abs-mult*) *auto*
also have $\ldots \leq N \mathbin{\hat{}} l * (of\text{-}nat\ l * N \mathbin{\hat{}} (l + 2))$

```
  proof −
    have |∑ k<l. μ i k * μ j k * β fs k| ≤ of-nat l * N ^ (l + 2)
    proof −
      have [simp]: 0 ≤ β fs k ‖gso k‖² ≤ N if k < l for k
        using that assms N-gso β-pos[of k] by auto
      have [simp]: 0 ≤ N * N ^ k for k
        using N-ge-0 assms by fastforce
      have |(∑ k < l. μ i k * μ j k * β fs k)| ≤ (∑ k < l. |μ i k * μ j k * β fs k|)
        using sum-abs by blast
      also have ... = (∑ k < l. |μ i k * μ j k| * β fs k)
       using assms by (auto intro!: sum.cong simp add: gso-norm-beta abs-mult-pos
sq-norm-vec-ge-0)
      also have ... = (∑ k < l. |μ i k| * |μ j k| * β fs k)
        using abs-mult by (fastforce intro!: sum.cong)
      also have ... ≤ (∑ k < l. (max |μ i k| |μ j k|) * (max |μ i k| |μ j k|) * β fs
k)
        by (auto intro!: sum-mono mult-mono)
      also have ... = (∑ k < l. (max |μ i k| |μ j k|)² * β fs k)
        by (auto simp add: power2-eq-square)
      also have ... ≤ (∑ k < l. N ^ (Suc k) * β fs k)
        using assms N-mu[of i] N-mu[of j] assms
        by (auto intro!: sum-mono mult-right-mono simp add: max-def)
      also have ... ≤ (∑ k < l. N ^ (Suc k) * N)
      using assms by (auto simp add: gso-norm-beta intro!: sum-mono mult-left-mono)
      also have ... ≤ (∑ k < l. N ^ (Suc l) * N)
      using assms N-1 N-ge-0 assms by (fastforce intro!: sum-mono mult-right-mono
power-increasing)
      also have ... = of-nat l * N ^ (l + 2)
        by auto
      finally show ?thesis
        by auto
    qed
    then show ?thesis
      using assms N-d N-ge-0 by (fastforce intro!: mult-mono zero-le-power)
  qed
  also have ... = of-nat l * N ^ (2 * l + 2)
    by (auto simp add: field-simps mult-2-right simp flip: power-add)
  finally show ?thesis
    by simp
qed

lemma leq-squared: (z::int) ≤ z²
proof (cases 0 < z)
  case True
  then show ?thesis
    by (auto intro!: self-le-power)
next
  case False
  then have z ≤ 0
```

```
    by (simp)
  also have 0 ≤ z^2
    by (auto)
  finally show ?thesis
    by simp
qed

lemma abs-leq-squared: |z::int| ≤ z^2
  using leq-squared[of |z|] by auto

end

context gram-schmidt-fs-int
begin

definition gso' where gso' i = d i ·_v (gso i)

fun a where
  a i 0 = fs ! i |
  a i (Suc l) = (1 / d l) ·_v ((d (Suc l) ·_v (a i l)) − ( μ' i l) ·_v gso' l)

lemma gso'-carrier-vec:
  assumes i < m
  shows gso' i ∈ carrier-vec n
  using assms by (auto simp add: gso'-def)

lemma a-carrier-vec:
  assumes l ≤ i i < m
  shows a i l ∈ carrier-vec n
  using assms by (induction l arbitrary: i) (auto simp add: gso'-def)

lemma a-l:
  assumes l ≤ i i < m
  shows a i l = d l ·_v (fs ! i + M.sumlist (map (λj. − μ i j ·_v gso j) [0..<l]))
using assms proof (induction l)
  case 0
  then show ?case by auto
next
  case (Suc l)
  have fsi: fs ! i ∈ carrier-vec n using f-carrier[of i] assms by auto
  have l-i-m: l ≤ i ⟹ l < m using assms by auto
  let ?a = fs ! i
  let ?sum = M.sumlist (map (λj. − μ i j ·_v gso j) [0..<l])
  let ?term = (− μ i l ·_v gso l)
  have carr: {?a,?sum,?term} ⊆ carrier-vec n
    using gso-dim l-i-m Suc(2) sumlist-dim assms
    by (auto intro!: sumlist-carrier)
  have a i (Suc l) =
        (1 / d l) ·_v ((d (Suc l) ·_v (d l ·_v (fs ! i + M.sumlist (map (λj. − μ i j ·_v
```

```
gso j) [0..<l])))))
        − ( μ' i l) ·v gso' l) using a.simps Suc by auto
  also have ... = (1 / d l) ·v ((d (Suc l) ·v (d l ·v (fs ! i + M.sumlist (map (λj.
− μ i j ·v gso j) [0..<l])))))
        + −d (Suc l) ∗ μ i l ∗ d l ·v gso l ) (is - = - ·v (?t1 + ?t2))
    unfolding μ'-def gso'-def by auto
  also have ?t2 = d l ·v (−d (Suc l) ∗ μ i l ·v gso l ) (is - = d l ·v ?tt2)
    using smult-smult-assoc by (auto)
  also have ?t1 = d l ·v ((d (Suc l) ·v (fs ! i + M.sumlist (map (λj. − μ i j ·v
gso j) [0..<l])))) (is - = d l ·v ?tt1)
    using smult-smult-assoc smult-smult-assoc[symmetric] by (auto)
  also have d l ·v ?tt1 + d l ·v ?tt2 = d l ·v (?tt1 + ?tt2)
    using gso-carrier l-i-m Suc fsi
    by (auto intro!: smult-add-distrib-vec[symmetric, of - n] add-carrier-vec sum-
list-carrier)
  also have (1 / d l) ·v ... = (d l / d l) ·v (?tt1 + ?tt2)
    by (intro eq-vecI, auto)
  also have d l / d l = 1
    using Gramian-determinant(2)[of l] l-i-m Suc by(auto simp: field-simps)
  also have 1 ·v (?tt1 + ?tt2) = ?tt1 + ?tt2 by simp
  also have ?tt2 = d (Suc l) ·v (− μ i l ·v gso l) by auto
  also have d (Suc l) ·v (fs ! i + ?sum) + ... =
          d (Suc l) ·v (fs ! i + ?sum + ?term)
    using carr by (subst smult-add-distrib-vec) (auto)
  also have (?a + ?sum) + ?term = ?a + (?sum + ?term)
    using carr by auto
  also have ?term = M.sumlist (map (λj. − μ i j ·v gso j) [l..<Suc l])
    using gso-carrier Suc l-i-m by auto
  also have ?sum + ... = M.sumlist (map (λj. − μ i j ·v gso j) [0..<Suc l])
    apply(subst sumlist-append[symmetric])
    using fsi l-i-m Suc sumlist-carrier gso-carrier by (auto intro!: sumlist-carrier)
  finally show ?case by auto
qed

lemma a-l':
  assumes i < m
  shows a i i = gso' i
proof −
  have a i i = d i ·v (fs ! i + M.sumlist (map (λj. − μ i j ·v gso j) [0..<i]))
    using a-l assms by auto
  also have ... = d i ·v gso i
    by (subst gso.simps, auto)
  finally have a i i = gso' i using gso'-def by auto
  from this show ?thesis by auto
qed

lemma
  assumes i < m l' ≤ i
  shows a i l' = (case l' of
```

```
      0 ⇒ fs ! i |
      Suc l ⇒ (1 / d l) ·v (d (Suc l) ·v (a i l) − (μ′ i l) ·v a l l))
proof (cases l′)
  case (Suc l)
  have a i (Suc l) = (1 / d l) ·v ((d (Suc l) ·v (a i l)) − ( μ′ i l) ·v a l l)
    using assms a-l Suc by(subst a-l′, auto)
  from this Suc show ?thesis by auto
qed auto

lemma a-Ints:
  assumes i < m l ≤ i k < n
  shows a i l $ k ∈ ℤ
proof −
  have fsi: fs ! i ∈ carrier-vec n using f-carrier[of i] assms by auto
  have a i l = d l ·v (fs ! i + M.sumlist (map (λj. − μ i j ·v gso j) [0..<l]))
    (is - = - ·v (- + ?sum))
    using assms by (subst a-l, auto)
  also have ?sum = sumlist (map (λk. κ i l k ·v fs ! k) [0..<l])
    using assms gso-carrier
    by (subst κ-def, auto)
  also have d l ·v (fs ! i + sumlist (map (λk. κ i l k ·v fs ! k) [0..<l]))
        = d l ·v fs ! i + d l ·v sumlist (map (λk. κ i l k ·v fs ! k) [0..<l])
    (is - = - + ?sum)
    using sumlist-carrier fsi apply
     (subst smult-add-distrib-vec[symmetric])
     apply force
    using assms fsi by (subst sumlist-carrier, auto)
  also have ?sum = sumlist  (map (λk. (d l ∗ κ i l k) ·v fs ! k) [0..<l])
    apply(subst eq-vecI sumlist-nth)
    using fsi assms
    by (auto simp: dim-sumlist sum-distrib-left sumlist-nth smult-smult-assoc alge-
bra-simps)
  finally have a i l = d l ·v fs ! i + sumlist  (map (λk. (d l ∗ κ i l k) ·v fs ! k)
[0..<l])
    by auto

  hence a i l $ k = (d l ·v fs ! i + sumlist (map (λk. (d l ∗ κ i l k) ·v fs ! k)
[0..<l])) $ k by simp
  also have ... = (d l ·v fs ! i) $ k + (sumlist (map (λk. (d l ∗ κ i l k) ·v fs ! k)
[0..<l])) $ k
    apply (subst index-add-vec)
    using assms fsi by (subst sumlist-dim, auto)
  finally have id: a i l $ k = (d l ·v fs ! i) $ k + (sumlist (map (λk. (d l ∗ κ i l
k) ·v fs ! k) [0..<l])) $ k.

  show ?thesis unfolding id
    using fsi assms d-κ-Ints fs-int
    by (auto simp: dim-sumlist sumlist-nth
     intro!: Gramian-determinant-Ints sumlist-carrier Ints-minus Ints-add Ints-sum
```

Ints-mult[*of - fs* ! - $ -] *Ints-scalar-prod*[*OF fsi*])
qed

lemma *a-alt-def*:
assumes $l < \textit{length fs}$
shows $a\ i\ (\textit{Suc}\ l) = (\textit{let}\ v = \mu'\ l\ l \cdot_v (a\ i\ l) - (\ \mu'\ i\ l) \cdot_v a\ l\ l\ \textit{in}$
$(\textit{if}\ l = 0\ \textit{then}\ v\ \textit{else}\ (1\ /\ \mu'\ (l - 1)\ (l - 1)) \cdot_v v))$
proof −
have [*simp*]: $\mu'\ (l - \textit{Suc}\ 0)\ (l - \textit{Suc}\ 0) = d\ l$ **if** $0 < l$
using *that* **unfolding** μ'-*def* **by** (*auto simp add*: μ.*simps*)
have [*simp*]: $\mu'\ l\ l = d\ (\textit{Suc}\ l)$
unfolding μ'-*def* **by** (*auto simp add*: μ.*simps*)
show *?thesis*
using *assms* **by** (*auto simp add*: *Let-def a-l'*)
qed

end

context *fs-int-indpt*
begin

fun *gso-int* :: *nat* ⇒ *nat* ⇒ *int vec* **where**
gso-int i 0 = *fs* ! *i* |
gso-int i (*Suc l*) = (*let v* = μ' *l l* ·$_v$ (*gso-int i l*) − μ' *i l* ·$_v$ *gso-int l l in*
(*if l* = *0 then v else map-vec* (λ*k*. *k div* μ' (*l* − *1*) (*l* − *1*)) *v*))

lemma *gso-int-carrier-vec*:
assumes $i < \textit{length fs}\ l \leq i$
shows *gso-int i l* ∈ *carrier-vec n*
using *assms* **by** (*induction l arbitrary*: *i*) (*fastforce simp add*: *Let-def*)+

lemma *gso-int*:
assumes $i < \textit{length fs}\ l \leq i$
shows *of-int-hom.vec-hom* (*gso-int i l*) = *gs.a i l*
proof −
have *dim-vec* (*gso-int i l*) = *n dim-vec* (*gs.a i l*) = *n*
using *gs.a-carrier-vec assms gso-int-carrier-vec carrier-dim-vec* **by** *auto*
moreover have *of-int-hom.vec-hom* (*gso-int i l*) $ *k* = *gs.a i l* $ *k* **if** *k*: *k* < *n*
for *k*
using *assms* **proof** (*induction l arbitrary*: *i*)
case (*Suc l*)
note *IH* = *Suc*(*1*)
have [*simp*]: *dim-vec* (*gso-int i l*) = *n dim-vec* (*gs.a i l*) = *n dim-vec* (*gso-int l l*) = *n*
dim-vec (*gs.a l l*) = *n*
using *Suc gs.a-carrier-vec gso-int-carrier-vec carrier-dim-vec gs.gso'-carrier-vec*
by *auto*

```
    have rat-of-int (gso-int i l $ k) = gs.a i l $ k rat-of-int (gso-int l l $ k) = gs.a
l l $ k
      using that Suc(1)[of l] Suc(1)[of i] Suc by auto
    then have ?case if l = 0
    proof −
      have [simp]: fs ≠ []
        using Suc by auto
       have [simp]: dim-vec (gso-int i 0) = n dim-vec (gso-int 0 0) = n dim-vec
(gs.a i 0) = n
        dim-vec (gs.a 0 0) = n
        using Suc fs-carrier carrier-dim-vec gs.a-carrier-vec f-carrier by auto
      have [simp]: rat-of-int (μ' i 0) = gs.μ' i 0 rat-of-int (μ' 0 0) = gs.μ' 0 0
        using Suc σs-μ' by (auto intro!: σs-μ')
      then show ?thesis
        using that k Suc IH[of i ] Suc(1)[of 0]
          by (subst gso-int.simps, subst gs.a-alt-def) (auto simp del: gso-int.simps
gs.a.simps)
    qed
    moreover have ?case if 0 < l
    proof −
      have ∗: rat-of-int (μ' l l ∗ gso-int i l $ k − μ' i l ∗ gso-int l l $ k) / rat-of-int
(μ' (l − Suc 0) (l − Suc 0))
     = gs.a i (Suc l) $ k
        using Suc IH[of l] IH[of i] σs-μ' k that by (subst gs.a-alt-def) (auto simp
add: Let-def )
      have of-int-hom.vec-hom (gso-int i (Suc l)) $ k =
            rat-of-int ((μ' l l ∗ gso-int i l $ k − μ' i l ∗ gso-int l l $ k)
                        div μ' (l − Suc 0) (l − Suc 0))
        using that gso-int-carrier-vec k by (auto)
      also have ... = rat-of-int (μ' l l ∗ gso-int i l $ k − μ' i l ∗ gso-int l l $ k) /
rat-of-int (μ' (l − Suc 0) (l − Suc 0))
        using gs.a-Ints k Suc by (intro exact-division, subst ∗, force)
      also note ∗
      finally show ?thesis
        by (auto)
    qed
    ultimately show ?case
      by blast
  qed (auto)
  ultimately show ?thesis
    by auto
qed

function gso-int-tail' :: nat ⇒ nat ⇒ int vec ⇒ int vec where
  gso-int-tail' i l acc = (if l ≥ i then acc
    else (let v = μ' l l ·v acc − μ' i l ·v gso-int l l;
              acc' = (map-vec (λk. k div μ' (l − 1) (l − 1)) v)
        in gso-int-tail' i (l + 1) acc'))
  by pat-completeness auto
```

```
termination
  by (relation (λ(i,l,acc). i − l) <*mlex*> {}, goal-cases) (auto intro!: mlex-less
wf-mlex)

fun gso-int-tail :: nat ⇒ int vec where
  gso-int-tail i = (if i = 0 then fs ! 0 else
     let acc = μ' 0 0 ·v fs ! i − μ' i 0 ·v fs ! 0 in
     gso-int-tail' i 1 acc)

lemma gso-int-tail':
  assumes acc = gso-int i l 0 < i 0 < l l ≤ i
  shows gso-int-tail' i l acc = gso-int i i
  using assms proof (induction i l acc rule: gso-int-tail'.induct)
  case (1 i l acc)
  { assume li: l < i
    then have gso-int-tail' i l acc =
        gso-int-tail' i (l + 1) (map-vec (λk. k div μ' (l − 1) (l − 1)) (μ' l l ·v acc
− μ' i l ·v gso-int l l))
      using 1 by (auto simp add: Let-def)
    also have ... = gso-int i i
      using 1 li by (intro 1) (auto)
  }
  then show ?case
    using 1 by fastforce
qed

lemma gso-int-tail: gso-int-tail i = gso-int i i
proof (cases 0 < i)
  assume i: 0 < i
  then have gso-int-tail i = gso-int-tail' i (Suc 0) (gso-int i 1)
    by (subst gso-int-tail.simps) (auto)
  also have ... = gso-int i i
    using i by (intro gso-int-tail') (auto intro!: gso-int-tail')
  finally show gso-int-tail i = gso-int i i
    by simp
qed (auto)

end

locale gso-array
begin

function while :: nat ⇒ nat ⇒ int vec iarray ⇒ int iarray iarray ⇒ int vec ⇒
int vec where
  while i l gsa dmusa acc =  (if l ≥ i then acc
    else (let v = dmusa !! l !! l ·v acc − dmusa !! i !! l ·v gsa !! l;
            acc' = (map-vec (λk. k div dmusa !! (l − 1) !! (l − 1)) v)
        in while i (l + 1) gsa dmusa acc'))
  by pat-completeness auto
```

termination
by (*relation* (λ(*i*,*l*,*acc*). *i* − *l*) <∗*mlex*∗> {}, *goal-cases*) (*auto intro*!: *mlex-less wf-mlex*)

declare *while.simps*[*simp del*]

definition *gso′* **where**
gso′ i fsa gsa dmusa = (*if i* = *0 then fsa* !! *0 else*
let acc = *dmusa* !! *0* !! *0* ·$_v$ *fsa* !! *i* − *dmusa* !! *i* !! *0* ·$_v$ *fsa* !! *0 in*
while i 1 gsa dmusa acc)

function *gsos′* **where**
gsos′ i n dmusa fsa gsa = (*if i* ≥ *n then gsa else*
gsos′ (*i* + *1*) *n dmusa fsa* (*iarray-append gsa* (*gso′ i fsa gsa dmusa*)))
by *pat-completeness auto*
termination
by (*relation* (λ(*i*,*n*,*dmusa*,*fsa*,*gsa*). *n* − *i*) <∗*mlex*∗> {}, *goal-cases*) (*auto intro*!: *mlex-less wf-mlex*)

declare *gsos′.simps*[*simp del*]

definition *gso′-array* **where**
gso′-array dmusa fs = *gsos′ 0* (*length fs*) *dmusa* (*IArray fs*) (*IArray* [])

definition *gso-array* **where**
gso-array fs = (*let dmusa* = *dμ-impl fs*; *gsa* = *gso′-array dmusa fs*
in IArray.of-fun (λ*i*. (*if i* = *0 then 1 else inverse* (*rat-of-int* (*dmusa* !! (*i* − *1*) !! (*i* − *1*))))
·$_v$ *of-int-hom.vec-hom* (*gsa* !! *i*)) (*length fs*))

end

declare *gso-array.gso-array-def*[*code*]
declare *gso-array.gso′-array-def*[*code*]
declare *gso-array.gsos′.simps*[*code*]
declare *gso-array.gso′-def*[*code*]
declare *gso-array.while.simps*[*code*]

lemma *map-vec-id*[*simp*]: *map-vec id* = *id*
by (*auto intro*!: *eq-vecI*)

context *fs-int-indpt*
begin

lemma *gso-array.gso′-array* (*dμ-impl fs*) *fs* = *IArray* (*map* (λ*k*. *gso-int k k*) [*0*..<*length fs*])
proof −
have *a*[*simp*]: *IArray* (*IArray.list-of a*) = *a* **for** *a*:: *'a iarray*
by (*metis iarray.exhaust list-of.simps*)

```
  have [simp]: length (IArray.list-of (iarray-append xs x)) = Suc (IArray.length
xs) for x xs
    unfolding iarray-append-code by (simp)
  have [simp]: map-iarray f as = IArray (map f (IArray.list-of as)) for f as
    by (metis a iarray.simps(4))
  have d[simp]: IArray.list-of (IArray.list-of (dμ-impl fs) ! i) ! j = μ' i j
    if i < length fs j ≤ i for j i
    using that by (auto simp add: μ' dμ-impl nth-append)
  let ?rat-vec = of-int-hom.vec-hom
  have ∗: gso-array.while i j gsa (dμ-impl fs) acc = gso-int-tail' i j acc'
      if i < length fs j ≤ i acc = acc'
        ⋀k. k < i ⟹ gsa !! k = gso-int k k for i j gsa acc acc'
    using that apply (induction i j acc arbitrary: acc' rule: gso-int-tail'.induct)
    by (subst gso-array.while.simps, subst gso-int-tail'.simps, auto)
  then have ∗: gso-array.gso' i (IArray fs) gsa (dμ-impl fs) = gso-int i i
    if assms: i < length fs ⋀k. k < i ⟹ gsa !! k = gso-int k k for i gsa
  proof −
    have IArray.list-of (IArray.list-of (dμ-impl fs) ! 0) ! 0 = μ' 0 0
      using that by (subst d) (auto)
    then have gso-array.gso' i (IArray fs) gsa (dμ-impl fs) = gso-int-tail i
      unfolding gso-array.gso'-def gso-int-tail.simps Let-def
      using that ∗ by (auto simp del: gso-int-tail'.simps)
    then show ?thesis
      using gso-int-tail by simp
  qed
  then have ∗: gso-array.gsos' i n (dμ-impl fs) (IArray fs) gsa =
        IArray (IArray.list-of gsa @ (map (λk. gso-int k k) [i..<n]))
    if n ≤ length fs
      gsa = IArray.of-fun (λk. gso-int k k) i for i n gsa
   using that proof (induction i n (dμ-impl fs) (IArray fs) gsa rule: gso-array.gsos'.induct)
    case (1 i n gsa)
    { assume i-n: i < n
      have [simp]: gso-array.gso' i (IArray fs) gsa (dμ-impl fs) = gso-int i i
        using 1 i-n by (intro ∗) auto
      have gso-array.gsos' i n (dμ-impl fs) (IArray fs) gsa = gso-array.gsos' (i +
1) n (dμ-impl fs) (IArray fs) (iarray-append gsa (gso-array.gso' i (IArray fs) gsa
(dμ-impl fs)))
        using i-n by (simp add: gso-array.gsos'.simps)
      also have ... = IArray (IArray.list-of gsa @ gso-int i i # map (λk. gso-int
k k) [Suc i..<n])
        using 1 i-n by (subst 1) (auto simp add: iarray-append-code)
      also have ... = IArray (IArray.list-of gsa @ map (λk. gso-int k k) [i..<n])
        using i-n by (auto simp add: upt-conv-Cons)
      finally have ?case
        by simp }
    then show ?case
      by (auto simp add: gso-array.gsos'.simps)
  qed
  then show ?thesis
```

unfolding *gso-array.gso'-array-def* **by** (*subst* ∗) *auto*
qed

end

8.3 Lemmas Summarizing All Bounds During GSO Computation

context *gram-schmidt-fs-int*
begin

lemma *combined-size-bound-integer*:
assumes x: $x \in \{fs\ !\ i\ \$\ j \mid i\ j.\ i < m \wedge j < n\}$
$\cup\ \{\mu'\ i\ j \mid i\ j.\ j \leq i \wedge i < m\}$
$\cup\ \{\sigma\ l\ i\ j \mid i\ j\ l.\ i < m \wedge j \leq i \wedge l \leq j\}$
(**is** $x \in ?fs \cup ?\mu' \cup ?\sigma$)
and m: $m \neq 0$
shows $|x| \leq$ *of-nat* $m * N \mathbin{\hat{}} (3 * Suc\ m)$
proof −
let $?m = (of\text{-}nat\ m)::'a::trivial\text{-}conjugatable\text{-}linordered\text{-}field$
have [*simp*]: $1 \leq ?m$
using m **by** (*metis Num.of-nat-simps One-nat-def Suc-leI neq0-conv of-nat-mono*)
have [*simp*]: $|(of\text{-}int\ z)::'a::trivial\text{-}conjugatable\text{-}linordered\text{-}field| \leq (of\text{-}int\ z)^2$ **for** z
using *abs-leq-squared* **by** (*metis of-int-abs of-int-le-iff of-int-power*)
have $|fs\ !\ i\ \$\ j| \leq$ *of-nat* $m * N \mathbin{\hat{}} (3 * Suc\ m)$ **if** $i < m\ j < n$ **for** $i\ j$
proof −
have $|fs\ !\ i\ \$\ j| \leq |fs\ !\ i\ \$\ j|^2$
by (*rule Ints-cases*[*of fs ! i $ j*]) (*use fs-int that* **in** *auto*)
also have $|fs\ !\ i\ \$\ j|^2 \leq \|fs\ !\ i\|^2$
using *that* **by** (*intro vec-le-sq-norm*) (*auto*)
also have ... $\leq 1 * N$
using *N-fs that* **by** *auto*
also have ... $\leq$ *of-nat* $m * N \mathbin{\hat{}} (3 * Suc\ m)$
using m *N-1 mult-mono self-le-power*
by (*intro mult-mono self-le-power*)
(*auto simp del*: *length-0-conv length-greater-0-conv*)
finally show *?thesis*
by (*auto*)
qed
then have $|x| \leq$ *of-nat* $m * N \mathbin{\hat{}} (3 * Suc\ m)$ **if** $x \in ?fs$
using *that* **by** *auto*
moreover have $|x| \leq$ *of-nat* $m * N \mathbin{\hat{}} (3 * Suc\ m)$ **if** $x \in ?\mu'$
proof −
have $|\mu'\ i\ j| \leq$ *of-nat* $m * N \mathbin{\hat{}} (3 + 3 * m)$ **if** $j \leq i\ i < m$ **for** $i\ j$
proof −
have $\mu'\ i\ j \in \mathbb{Z}$
unfolding μ'*-def* **using** *that d-mu-Ints* **by** *auto*
then have $|\mu'\ i\ j| \leq (\mu'\ i\ j)^2$

```
      by (rule Ints-cases[of μ' i j]) auto
    also have ... ≤ N ^ (3 * Suc j)
      using that N-μ' by auto
    also have ... ≤ 1 * N ^ (3 * Suc m)
      using that assms N-1 by (auto intro!: power-increasing)
    also have ... ≤ of-nat m * N ^ (3 * Suc m)
      using N-ge-0 assms zero-le-power by (intro mult-mono) auto
    finally show ?thesis
      by auto
  qed
  then show ?thesis
    using that by auto
 qed
 moreover have |x| ≤ of-nat m * N ^ (3 * Suc m) if x ∈ ?σ
 proof −
  have |σ l i j| ≤ of-nat m * N ^ (3 + 3 * m) if i < m j ≤ i l ≤ j for i j l
  proof −
    have |σ l i j| ≤ of-nat l * N ^ (2 * l + 2)
      using that N-σ by auto
    also have ... ≤ of-nat m * N ^ (2 * l + 2)
      using that N-ge-0 assms zero-le-power by (intro mult-mono) auto
    also have ... ≤ of-nat m * N ^ (3 * Suc m)
    proof −
      have N ^ (2 * l + 2) ≤ N ^ (3 * Suc m)
     using that assms N-1 by (intro power-increasing) (auto intro!: power-increasing)
      then show ?thesis
        using that assms N-1 by (intro mult-mono) (auto)
    qed
    finally show ?thesis
      by simp
  qed
  then show ?thesis
    using that by (auto)
 qed
 ultimately show ?thesis
  using assms by auto
qed

end



context fs-int-indpt
begin


lemma combined-size-bound-rat-log:
 assumes x: x ∈ {gs.μ' i j | i j. j ≤ i ∧ i < m}
  ∪ {gs.σ l i j | i j l. i < m ∧ j ≤ i ∧ l ≤ j}
```

```
    (is x ∈ ?μ' ∪ ?σ)
    and m: m ≠ 0 x ≠ 0
  shows log 2 |real-of-rat x| ≤ log 2 m + (3 + 3 * m) * log 2 (real-of-rat gs.N)
proof −
  let ?r-fs = map of-int-hom.vec-hom fs::rat vec list
  have 1: map of-int-hom.vec-hom fs ! i $ j = of-int (fs ! i $ j) if i < m j < n
for i j
    using that by auto
  then have {?r-fs ! i $ j |i j. i < length ?r-fs ∧ j < n} =
          {rat-of-int (fs ! i $ j) |i j. i < length fs ∧ j < n}
    by (metis (mono-tags, opaque-lifting) length-map)
  then have x ∈ {?r-fs ! i $ j |i j. i < length (map of-int-hom.vec-hom fs) ∧ j <
n}
                ∪ {gs.μ' i j |i j. j ≤ i ∧ i < length ?r-fs}
                ∪ {gs.σ l i j |i j l. i < length ?r-fs ∧ j ≤ i ∧ l ≤ j}
    using assms by auto
  then have 1: |x| ≤ rat-of-nat (length ?r-fs) * gs.N ^ (3 * Suc (length ?r-fs)) (is
?ax ≤ ?t)
    using assms by (intro gs.combined-size-bound-integer) auto
  then have 1: real-of-rat ?ax ≤ real-of-rat ?t
    using of-rat-less-eq 1 by auto
  have 2: |real-of-rat x| = real-of-rat |x|
    by auto
  have log 2 |real-of-rat x| ≤ log 2 (real-of-rat ?t)
  proof −
    have 0 < rat-of-nat (length fs) * gs.N ^ (3 + 3 * length fs)
      using assms gs.N-1 by (auto)
    then show ?thesis
      using 1 assms by (subst log-le-cancel-iff) (auto)
  qed
  also have real-of-rat ?t = real m * real-of-rat gs.N ^ (3 + 3 * m)
    by (auto simp add: of-rat-mult of-rat-power)
  also have log 2 (m * real-of-rat gs.N ^ (3 + 3 * m)) = log 2 m + log 2 (real-of-rat
gs.N ^ (3 + 3 * m))
    using gs.N-1 assms by (subst log-mult) auto
  also have log 2 (real-of-rat gs.N ^ (3 + 3 * m)) = real (3 + 3 * length fs) *
log 2 (real-of-rat gs.N)
    using gs.N-1 assms by (subst log-nat-power) auto
  finally show ?thesis
    by (auto)
qed

lemma combined-size-bound-integer-log:
  assumes x: x ∈ {μ' i j | i j. j ≤ i ∧ i < m}
    ∪ {σ s l i j | i j l. i < m ∧ j ≤ i ∧ l < j}
    (is x ∈ ?μ' ∪ ?σ)
    and m: m ≠ 0 x ≠ 0
  shows log 2 |real-of-int x| ≤ log 2 m + (3 + 3 * m) * log 2 (real-of-rat gs.N)
proof −
```

```
  let ?x = rat-of-int x
  from m have m: m ≠ 0 ?x ≠ 0 by auto
  show ?thesis
  proof (rule order-trans[OF - combined-size-bound-rat-log[OF - m]], force)
    from x consider (1) i j where x = μ' i j j ≤ i i < m
      | (2) l i j where x = σs l i j i < m j ≤ i l < j by blast
    thus ?x ∈ {gs.μ' i j |i j. j ≤ i ∧ i < m} ∪ {gs.σ l i j |i j l. i < m ∧ j ≤ i ∧ l
≤ j}
    proof (cases)
      case (1 i j)
      with σs-μ'(2) show ?thesis by blast
    next
      case (2 l i j)
      hence Suc l ≤ j by auto
      from σs-μ'(1) 2 this show ?thesis by blast
    qed
  qed
qed

end
end
```

9 The LLL Algorithm

Soundness of the LLL algorithm is proven in four steps. In the basic version, we do recompute the Gram-Schmidt ortogonal (GSO) basis in every step. This basic version will have a full functional soundness proof, i.e., termination and the property that the returned basis is reduced. Then in LLL-Number-Bounds we will strengthen the invariant and prove that all intermediate numbers stay polynomial in size. Moreover, in LLL-Impl we will refine the basic version, so that the GSO does not need to be recomputed in every step. Finally, in LLL-Complexity, we develop an cost-annotated version of the refined algorithm and prove a polynomial upper bound on the number of arithmetic operations.

This theory provides a basic implementation and a soundness proof of the LLL algorithm to compute a "short" vector in a lattice.

```
theory LLL
  imports
    Gram-Schmidt-2
    Missing-Lemmas
    Jordan-Normal-Form.Determinant
    Abstract−Rewriting.SN-Order-Carrier
begin
```

9.1 Core Definitions, Invariants, and Theorems for Basic Version

locale *LLL* =
  fixes *n* :: *nat*
    and *m* :: *nat*
    and *fs-init* :: *int vec list*
    and α :: *rat*

begin

sublocale *vec-module TYPE*(*int*) *n***.**

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*

sublocale *gs*: *gram-schmidt-fs n RAT fs-init* **.**

abbreviation *lin-indep* **where** *lin-indep fs* ≡ *gs.lin-indpt-list* (*RAT fs*)
abbreviation *gso* **where** *gso fs* ≡ *gram-schmidt-fs.gso n* (*RAT fs*)
abbreviation μ **where** μ *fs* ≡ *gram-schmidt-fs.*μ *n* (*RAT fs*)

abbreviation *reduced* **where** *reduced fs* ≡ *gram-schmidt-fs.reduced n* (*RAT fs*) α

abbreviation *weakly-reduced* **where** *weakly-reduced fs* ≡ *gram-schmidt-fs.weakly-reduced n* (*RAT fs*) α

lattice of initial basis

definition *L* = *lattice-of fs-init*

maximum squared norm of initial basis

definition *N* = *max-list* (*map* (*nat* ∘ *sq-norm*) *fs-init*)

maximum absolute value in initial basis

definition *M* = *Max* ({*abs* (*fs-init* ! *i* $ *j*) | *i j*. *i* < *m* ∧ *j* < *n*} ∪ {*0*})

This is the core invariant which enables to prove functional correctness.

definition μ*-small fs i* = (∀ *j* < *i*. *abs* (μ *fs i j*) ≤ *1/2*)

definition *LLL-invariant-weak* :: *int vec list* ⇒ *bool* **where**
  LLL-invariant-weak fs = (
    gs.lin-indpt-list (*RAT fs*) ∧
    lattice-of fs = *L* ∧
    length fs = *m*)

lemma *LLL-inv-wD*: **assumes** *LLL-invariant-weak fs*
shows
lin-indep fs
length (*RAT fs*) = *m*
set fs ⊆ *carrier-vec n*
⋀ *i*. *i* < *m* ⟹ *fs* ! *i* ∈ *carrier-vec n*
⋀ *i*. *i* < *m* ⟹ *gso fs i* ∈ *carrier-vec n*
length fs = *m*
lattice-of fs = *L*
proof (*atomize* (*full*), *goal-cases*)
case *1*
interpret *gs'*: *gram-schmidt-fs-lin-indpt n RAT fs*
by (*standard*) (*use assms LLL-invariant-weak-def gs.lin-indpt-list-def* **in** *auto*)
show *?case*
using *assms gs'.fs-carrier gs'.f-carrier gs'.gso-carrier*
by (*auto simp add*: *LLL-invariant-weak-def gram-schmidt-fs.reduced-def*)
qed

lemma *LLL-inv-wI*: **assumes**
set fs ⊆ *carrier-vec n*
length fs = *m*
lattice-of fs = *L*
lin-indep fs
shows *LLL-invariant-weak fs*
unfolding *LLL-invariant-weak-def Let-def* **using** *assms* **by** *auto*

definition *LLL-invariant* :: *bool* ⇒ *nat* ⇒ *int vec list* ⇒ *bool* **where**
LLL-invariant upw i fs = (
gs.lin-indpt-list (*RAT fs*) ∧
lattice-of fs = *L* ∧
reduced fs i ∧
i ≤ *m* ∧
length fs = *m* ∧
(*upw* ∨ *μ-small fs i*)
)

lemma *LLL-inv-imp-w*: *LLL-invariant upw i fs* ⟹ *LLL-invariant-weak fs*
unfolding *LLL-invariant-def LLL-invariant-weak-def* **by** *blast*

lemma *LLL-invD*: **assumes** *LLL-invariant upw i fs*
shows
lin-indep fs
length (*RAT fs*) = *m*
set fs ⊆ *carrier-vec n*
⋀ *i*. *i* < *m* ⟹ *fs* ! *i* ∈ *carrier-vec n*
⋀ *i*. *i* < *m* ⟹ *gso fs i* ∈ *carrier-vec n*
length fs = *m*
lattice-of fs = *L*
weakly-reduced fs i

```
  i ≤ m
  reduced fs i
  upw ∨ μ-small fs i
proof (atomize (full), goal-cases)
  case 1
  interpret gs': gram-schmidt-fs-lin-indpt n RAT fs
    by (standard) (use assms LLL-invariant-def gs.lin-indpt-list-def in auto)
  show ?case
    using assms gs'.fs-carrier gs'.f-carrier gs'.gso-carrier
    by (auto simp add: LLL-invariant-def gram-schmidt-fs.reduced-def)
qed

lemma LLL-invI: assumes
  set fs ⊆ carrier-vec n
  length fs = m
  lattice-of fs = L
  i ≤ m
  lin-indep fs
  reduced fs i
  upw ∨ μ-small fs i
shows LLL-invariant upw i fs
  unfolding LLL-invariant-def Let-def split using assms by auto



end

locale fs-int' =
  fixes n m fs-init fs
  assumes LLL-inv: LLL.LLL-invariant-weak n m fs-init fs

sublocale fs-int' ⊆ fs-int-indpt
   using LLL-inv unfolding LLL.LLL-invariant-weak-def by (unfold-locales) blast

context LLL
begin

lemma gso-cong: assumes ⋀ i. i ≤ x ⟹ f1 ! i = f2 ! i
   x < length f1 x < length f2
  shows gso f1 x = gso f2 x
  by (rule gs.gso-cong, insert assms, auto)

lemma μ-cong: assumes ⋀ k. j < i ⟹ k ≤ j ⟹ f1 ! k = f2 ! k
  and i: i < length f1 i < length f2
  and j < i ⟹ f1 ! i = f2 ! i
  shows μ f1 i j = μ f2 i j
  by (rule gs.μ-cong, insert assms, auto)

definition reduction where reduction = (4+α)/(4*α)
```

definition $d :: int\ vec\ list \Rightarrow nat \Rightarrow int$ **where** $d\ fs\ k = gs.Gramian\text{-}determinant\ fs\ k$
definition $D :: int\ vec\ list \Rightarrow nat$ **where** $D\ fs = nat\ (\prod\ i < m.\ d\ fs\ i)$

definition $d\mu\ gs\ i\ j = int\text{-}of\text{-}rat\ (of\text{-}int\ (d\ gs\ (Suc\ j)) * \mu\ gs\ i\ j)$

definition $logD :: int\ vec\ list \Rightarrow nat$
where $logD\ fs = (if\ \alpha = 4/3\ then\ (D\ fs)\ else\ nat\ (floor\ (log\ (1\ /\ of\text{-}rat\ reduction)\ (D\ fs))))$

definition $LLL\text{-}measure :: nat \Rightarrow int\ vec\ list \Rightarrow nat$ **where**
$LLL\text{-}measure\ i\ fs = (2 * logD\ fs + m - i)$

context
fixes fs
assumes $Linv$: $LLL\text{-}invariant\text{-}weak\ fs$
begin

interpretation fs: $fs\text{-}int'\ n\ m\ fs\text{-}init\ fs$
by $(standard)$ $(use\ Linv$ **in** $auto)$

lemma $Gramian\text{-}determinant$:
assumes k: $k \leq m$
shows $of\text{-}int\ (gs.Gramian\text{-}determinant\ fs\ k) = (\prod\ j{<}k.\ sq\text{-}norm\ (gso\ fs\ j))$ (**is** $?g1$)
$gs.Gramian\text{-}determinant\ fs\ k > 0$ (**is** $?g2$)
using $assms\ fs.Gramian\text{-}determinant\ LLL\text{-}inv\text{-}wD[OF\ Linv]$ **by** $auto$

lemma $LLL\text{-}d\text{-}pos$ $[intro]$: **assumes** k: $k \leq m$
shows $d\ fs\ k > 0$
unfolding $d\text{-}def$ **using** $fs.Gramian\text{-}determinant\ k\ LLL\text{-}inv\text{-}wD[OF\ Linv]$ **by** $auto$

lemma $LLL\text{-}d\text{-}Suc$: **assumes** k: $k < m$
shows $of\text{-}int\ (d\ fs\ (Suc\ k)) = sq\text{-}norm\ (gso\ fs\ k) * of\text{-}int\ (d\ fs\ k)$
using $assms\ fs.fs\text{-}int\text{-}d\text{-}Suc\ \ LLL\text{-}inv\text{-}wD[OF\ Linv]$ **unfolding** $fs.d\text{-}def\ d\text{-}def$ **by** $auto$

lemma $LLL\text{-}D\text{-}pos$:
shows $D\ fs > 0$
using $fs.fs\text{-}int\text{-}D\text{-}pos\ LLL\text{-}inv\text{-}wD[OF\ Linv]$ **unfolding** $D\text{-}def\ fs.D\text{-}def\ fs.d\text{-}def\ d\text{-}def$ **by** $auto$
end

Condition when we can increase the value of i

lemma $increase\text{-}i$:
assumes $Linv$: $LLL\text{-}invariant\ upw\ i\ fs$
assumes i: $i < m$

and *upw*: $upw \Longrightarrow i = 0$
and *red-i*: $i \neq 0 \Longrightarrow$ *sq-norm* $(gso\ fs\ (i - 1)) \leq \alpha *$ *sq-norm* $(gso\ fs\ i)$
shows *LLL-invariant True* $(Suc\ i)$ *fs LLL-measure i fs* $>$ *LLL-measure* $(Suc\ i)$ *fs*
proof −
note *inv* = *LLL-invD*[*OF Linv*]
from *inv*(*8*,*10*) **have** *red*: *weakly-reduced fs i*
and *sred*: *reduced fs i* **by** (*auto*)
from *red red-i i* **have** *red*: *weakly-reduced fs* (*Suc i*)
unfolding *gram-schmidt-fs.weakly-reduced-def*
by (*intro allI impI*, *rename-tac ii*, *case-tac Suc ii* = *i*, *auto*)
from *inv*(*11*) *upw* **have** *sred-i*: $\bigwedge j.\ j < i \Longrightarrow |\mu\ fs\ i\ j| \leq 1\ /\ 2$
unfolding *μ-small-def* **by** *auto*
from *sred sred-i* **have** *sred*: *reduced fs* (*Suc i*)
unfolding *gram-schmidt-fs.reduced-def*
by (*intro conjI*[*OF red*] *allI impI*, *rename-tac ii j*, *case-tac ii* = *i*, *auto*)
show *LLL-invariant True* (*Suc i*) *fs*
by (*intro LLL-invI*, *insert inv red sred i*, *auto*)
show *LLL-measure i fs* $>$ *LLL-measure* (*Suc i*) *fs* **unfolding** *LLL-measure-def*
using *i* **by** *auto*
qed

Standard addition step which makes $\mu_{i,j}$ small

definition *μ-small-row i fs j* = $(\forall\ j'.\ j \leq j' \longrightarrow j' < i \longrightarrow abs\ (\mu\ fs\ i\ j') \leq inverse\ 2)$

lemma *basis-reduction-add-row-main*: **assumes** *Linv*: *LLL-invariant-weak fs*
and *i*: $i < m$ **and** *j*: $j < i$
and *fs'*: $fs' = fs[\ i := fs\ !\ i - c \cdot_v fs\ !\ j]$
shows *LLL-invariant-weak fs'*
LLL-invariant True i fs $\Longrightarrow$ *LLL-invariant True i fs'*
$c = round\ (\mu\ fs\ i\ j) \Longrightarrow$ *μ-small-row i fs* (*Suc j*) $\Longrightarrow$ *μ-small-row i fs' j*
$c = round\ (\mu\ fs\ i\ j) \Longrightarrow abs\ (\mu\ fs'\ i\ j) \leq 1/2$
LLL-measure i fs' = *LLL-measure i fs*

$\bigwedge i.\ i < m \Longrightarrow gso\ fs'\ i = gso\ fs\ i$

$\bigwedge i'\ j'.\ i' < m \Longrightarrow j' < m \Longrightarrow$
$\mu\ fs'\ i'\ j' = (if\ i' = i \wedge j' \leq j\ then\ \mu\ fs\ i\ j' - of\text{-}int\ c * \mu\ fs\ j\ j'\ else\ \mu\ fs\ i'\ j')$

$\bigwedge ii.\ ii \leq m \Longrightarrow d\ fs'\ ii = d\ fs\ ii$
proof −
define *bnd* :: *rat* **where** *bnd*: $bnd = 4\ \hat{}\ (m - 1 - Suc\ j) *$ *of-nat* $(N\ \hat{}\ (m - 1) * m)$
define *M* **where** $M = map\ (\lambda i.\ map\ (\mu\ fs\ i)\ [0..<m])\ [0..<m]$
note *inv* = *LLL-inv-wD*[*OF Linv*]
note *Gr* = *inv*(*1*)
have *ji*: $j \leq i\ j < m$ **and** *jstrict*: $j < i$
and *add*: *set fs* $\subseteq$ *carrier-vec n i* $<$ *length fs j* $<$ *length fs i* $\neq j$
and *len*: *length fs* = *m*

and *indep*: *lin-indep fs*
using *inv j i* **by** *auto*
let *?R* = *rat-of-int*
let *?RV* = *map-vec ?R*
from *inv i j*
have *Fij*: *fs* ! *i* ∈ *carrier-vec n fs* ! *j* ∈ *carrier-vec n* **by** *auto*
let *?x* = *fs* ! *i* − *c* ·$_v$ *fs* ! *j*
let *?g* = *gso fs*
let *?g′* = *gso fs′*
let *?mu* = *μ fs*
let *?mu′* = *μ fs′*
from *inv j i*
have *Fi*:⋀ *i*. *i* < *length* (*RAT fs*) ⟹ (*RAT fs*) ! *i* ∈ *carrier-vec n*
and *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*
and *len′*: *length* (*RAT fs*) = *m*
and *add′*:*set* (*map ?RV fs*) ⊆ *carrier-vec n*
by *auto*
have *RAT-F1*: *RAT fs′* = (*RAT fs*)[*i* := (*RAT fs*) ! *i* − *?R c* ·$_v$ (*RAT fs*) ! *j*]
unfolding *fs′*
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* (*fs* ! *i* − *c* ·$_v$ *fs* ! *j*) =
?RV (*fs* ! *i*) − *?R c* ·$_v$ *?RV* (*fs* ! *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 fs′* ! *i* = (*RAT fs*) ! *i* − *?R c* ·$_v$ (*RAT fs*) ! *j* (**is** - = - − *?mui*)
using *i len* **by** *auto*
have *uminus*: *fs* ! *i* − *c* ·$_v$ *fs* ! *j* = *fs* ! *i* + −*c* ·$_v$ *fs* ! *j*
by (*subst minus-add-uminus-vec*, *insert Fij*, *auto*)
have *lattice-of fs′* = *lattice-of fs* **unfolding** *fs′ uminus*
by (*rule lattice-of-add*[*OF add*, *of* - − *c*], *auto*)
with *inv* **have** *lattice*: *lattice-of fs′* = *L* **by** *auto*
from *add len*
have *k* < *length fs* ⟹ ¬ *k* ≠ *i* ⟹ *fs′* ! *k* ∈ *carrier-vec n* **for** *k*
unfolding *fs′*
by (*metis* (*no-types*, *lifting*) *nth-list-update nth-mem subset-eq carrier-dim-vec*

index-minus-vec(2)
index-smult-vec(2))
hence $k < \textit{length fs} \Longrightarrow \textit{fs}' ! k \in \textit{carrier-vec n}$ **for** k
unfolding $\textit{fs}'$ **using** *add len* **by** (*cases* $k \neq i$,*auto*)
with *len* **have** *F1*: $\textit{set fs}' \subseteq \textit{carrier-vec n length fs}' = m$ **unfolding** $\textit{fs}'$ **by** (*auto simp*: *set-conv-nth*)
hence *F1'*: $\textit{length } (\textit{RAT fs}') = m\ \textit{SRAT fs}' \subseteq \textit{Rn}$ **by** *auto*
from *indep* **have** *dist*: *distinct* (*RAT fs*) **by** (*auto simp*: *gs.lin-indpt-list-def*)
have *Fij'*: $(\textit{RAT fs}) ! i \in \textit{Rn}\ (\textit{RAT fs}) ! j \in \textit{Rn}$ **using** *add'*[*unfolded set-conv-nth*] $i\ \langle j < m \rangle\ \textit{len}$ **by** *auto*
have *uminus'*: $(\textit{RAT fs}) ! i - \textit{?R } c \cdot_v (\textit{RAT fs}) ! j = (\textit{RAT fs}) ! i + - \textit{?R } c \cdot_v (\textit{RAT fs}) ! j$
by (*subst minus-add-uminus-vec*[**where** $n = n$], *insert Fij'*, *auto*)
have *span-F-F1*: $\textit{gs.span } (\textit{SRAT fs}) = \textit{gs.span } (\textit{SRAT fs}')$ **unfolding** *RAT-F1 uminus'*
by (*rule gs.add-vec-span*, *insert len add*, *auto*)
have $**$: $\textit{?RV } (\textit{fs} ! i) + - \textit{?R } c \cdot_v (\textit{RAT fs}) ! j = \textit{ ?RV } (\textit{fs} ! i - c \cdot_v \textit{fs} ! j)$
by (*rule eq-vecI*, *insert Fij len i j*, *auto*)
from *i j len* **have** $j < \textit{length } (\textit{RAT fs})\ i < \textit{length } (\textit{RAT fs})\ i \neq j$ **by** *auto*
from *gs.lin-indpt-list-add-vec*[*OF this indep*, *of* $-$ *of-int c*]
have $\textit{gs.lin-indpt-list } ((\textit{RAT fs}) [i := (\textit{RAT fs}) ! i + - \textit{?R } c \cdot_v (\textit{RAT fs}) ! j])$ (**is** *gs.lin-indpt-list ?F1*) **.**
also have $\textit{?F1} = \textit{RAT fs}'$ **unfolding** $\textit{fs}'$ **using** *i len Fij'* $**$
by (*auto simp*: *map-update*)
finally have *indep-F1*: *lin-indep* $\textit{fs}'$ **.**
have *conn1*: $\textit{set } (\textit{RAT fs}) \subseteq \textit{carrier-vec n}\ \ \textit{length } (\textit{RAT fs}) = m\ \textit{distinct } (\textit{RAT fs})$
gs.lin-indpt (*set* (*RAT fs*))
using *inv* **unfolding** *gs.lin-indpt-list-def* **by** *auto*
have *conn2*: $\textit{set } (\textit{RAT fs}') \subseteq \textit{carrier-vec n}\ \ \textit{length } (\textit{RAT fs}') = m\ \textit{distinct } (\textit{RAT fs}')$
gs.lin-indpt (*set* (*RAT fs'*))
using *indep-F1 F1'* **unfolding** *gs.lin-indpt-list-def* **by** *auto*
interpret *gs1*: *gram-schmidt-fs-lin-indpt n RAT fs*
by (*standard*) (*use inv gs.lin-indpt-list-def* **in** *auto*)
interpret *gs2*: *gram-schmidt-fs-lin-indpt n RAT fs'*
by (*standard*) (*use indep-F1 F1' gs.lin-indpt-list-def* **in** *auto*)
let $\textit{?G} = \textit{map ?g } [0\ ..<\ m]$
let $\textit{?G}' = \textit{map ?g}' [0\ ..<\ m]$
from *gs1.span-gso gs2.span-gso gs1.gso-carrier gs2.gso-carrier conn1 conn2 span-F-F1 len*
have *span-G-G1*: $\textit{gs.span } (\textit{set ?G}) = \textit{gs.span } (\textit{set ?G}')$
and *lenG*: $\textit{length ?G} = m$
and *Gi*: $i < \textit{length ?G} \Longrightarrow \textit{?G} ! i \in \textit{Rn}$
and *G1i*: $i < \textit{length ?G}' \Longrightarrow \textit{?G}' ! i \in \textit{Rn}$ **for** i
by *auto*
have *eq*: $x \neq i \Longrightarrow \textit{RAT fs}' ! x = (\textit{RAT fs}) ! x$ **for** x **unfolding** *RAT-F1* **by** *auto*
hence *eq-part*: $x < i \Longrightarrow \textit{?g}'\ x = \textit{?g}\ x$ **for** x

by (*intro gs.gso-cong*, *insert len*, *auto*)
have *G*: $i < m \Longrightarrow (RAT\ fs)\ !\ i \in Rn$
$i < m \Longrightarrow fs\ !\ i \in carrier\text{-}vec\ n$ **for** *i* **by**(*insert add len'*, *auto*)
note *carr1*[*intro*] = *this*[*OF i*] *this*[*OF ji*(*2*)]
have $x < m \Longrightarrow ?g\ x \in Rn$
$x < m \Longrightarrow ?g'\ x \in Rn$
$x < m \Longrightarrow dim\text{-}vec\ (gso\ fs\ x) = n$
$x < m \Longrightarrow dim\text{-}vec\ (gso\ fs'\ x) = n$
for *x* **using** *inv G1i* **by** (*auto simp*:*o-def Gi G1i*)
hence *carr2*[*intro*!]:*?g i* ∈ *Rn ?g' i* ∈ *Rn*
?g ‘ {*0*..<*i*} ⊆ *Rn*
?g ‘ {*0*..<*Suc i*} ⊆ *Rn* **using** *i* **by** *auto*
have *F1-RV*: *?RV* (*fs'* ! *i*) = *RAT fs'* ! *i* **using** *i F1* **by** *auto*
have *F-RV*: *?RV* (*fs* ! *i*) = (*RAT fs*) ! *i* **using** *i len* **by** *auto*
from *eq-part*
have *span-G1-G*: *gs.span* (*?g'* ‘ {*0*..<*i*}) = *gs.span* (*?g* ‘ {*0*..<*i*}) (**is** *?ls* = *?rs*)
apply(*intro cong*[*OF refl*[*of gs.span*]],*rule image-cong*[*OF refl*]) **using** *eq* **by** *auto*
have (*RAT fs'*) ! *i* − *?g' i* = ((*RAT fs*) ! *i* − *?g' i*) − *?mui*
unfolding *RAT-F1-i* **using** *carr1 carr2*
by (*intro eq-vecI*, *auto*)
hence *in1*:((*RAT fs*) ! *i* − *?g' i*) − *?mui* ∈ *?rs*
using *gs2.oc-projection-exist*[*of i*] *conn2 i* **unfolding** *span-G1-G* **by** *auto*
from ‹*j* < *i*› **have** *Gj-mem*: (*RAT fs*) ! *j* ∈ (λ *x*. ((*RAT fs*) ! *x*)) ‘ {*0* ..< *i*} **by** *auto*
have *id1*: *set* (*take i* (*RAT fs*)) = (λ*x*. *?RV* (*fs* ! *x*)) ‘ {*0*..<*i*}
using ‹*i* < *m*› *len*
by (*subst nth-image*[*symmetric*], *force*+)
have (*RAT fs*) ! *j* ∈ *?rs* ⟷ (*RAT fs*) ! *j* ∈ *gs.span* ((λ*x*. *?RV* (*fs* ! *x*)) ‘ {*0*..<*i*})
using *gs1.partial-span* ‹*i* < *m*› *id1 inv* **by** *auto*
also have (λ*x*. *?RV* (*fs* ! *x*)) ‘ {*0*..<*i*} = (λ*x*. ((*RAT fs*) ! *x*)) ‘ {*0*..<*i*} **using** ‹*i* < *m*› *len* **by** *force*
also have (*RAT fs*) ! *j* ∈ *gs.span* ...
by (*rule gs.span-mem*[*OF - Gj-mem*], *insert* ‹*i* < *m*› *G*, *auto*)
finally have (*RAT fs*) ! *j* ∈ *?rs* **.**
hence *in2*:*?mui* ∈ *?rs*
apply(*intro gs.prod-in-span*) **by** *force*+
have *ineq*:((*RAT fs*) ! *i* − *?g' i*) + *?mui* − *?mui* = ((*RAT fs*) ! *i* − *?g' i*)
using *carr1 carr2* **by** (*intro eq-vecI*, *auto*)
have *cong'*: *A* = *B* ⟹ *A* ∈ *C* ⟹ *B* ∈ *C* **for** *A B* :: *'a vec* **and** *C* **by** *auto*
have ∗: *?g* ‘ {*0*..<*i*} ⊆ *Rn* **by** *auto*
have *in-span*: (*RAT fs*) ! *i* − *?g' i* ∈ *?rs*
by (*rule cong'*[*OF eq-vecI gs.span-add1*[*OF* ∗ *in1 in2*,*unfolded ineq*]], *insert carr1 carr2*, *auto*)
{
fix *x* **assume** *x*:*x* < *i* **hence** *x* < *m i* ≠ *x* **using** *i* **by** *auto*
from *gs2.orthogonal this inv assms*
have *?g' i* ⋅ *?g' x* = *0* **by** *auto*
}

hence *G1-G*: *?g′ i = ?g i*
by (*intro gs1.oc-projection-unique*) (*use inv i eq-part in-span* **in** *auto*)
show *eq-fs*:$x < m \Longrightarrow$ *?g′ x = ?g x*
for *x* **proof**(*induct x rule*:*nat-less-induct*[*rule-format*])
case (*1 x*)
hence *ind*: $m < x \Longrightarrow$ *?g′ m = ?g m*
for *m* **by** *auto*
{ **assume** $x > i$
hence *?case* **unfolding** *gs2.gso.simps*[*of x*] *gs1.gso.simps*[*of x*] **unfolding** *gs1.μ.simps gs2.μ.simps*
using *ind eq* **by** (*auto intro*: *cong*[*OF - cong*[*OF refl*[*of gs.sumlist*]]])
} **note** *eq-rest* = *this*
show *?case* **by** (*rule linorder-class.linorder-cases*[*of x i*],*insert G1-G eq-part eq-rest*,*auto*)
qed
hence *Hs*:*?G′ = ?G* **by** (*auto simp*:*o-def*)
have *red*: *weakly-reduced fs i* $\Longrightarrow$ *weakly-reduced fs′ i* **using** *eq-fs* ‹$i < m$› **unfolding** *gram-schmidt-fs.weakly-reduced-def* **by** *simp*
let *?Mi* = *M ! i ! j*
have *Gjn*: *dim-vec* (*fs ! j*) = *n* **using** *Fij*(*2*) *carrier-vecD* **by** *blast*
define *E* **where** *E* = *addrow-mat m* (− *?R c*) *i j*
define *M′* **where** *M′* = *gs1.M m*
define *N′* **where** *N′* = *gs2.M m*
have *E*: *E* ∈ *carrier-mat m m* **unfolding** *E-def* **by** *simp*
have *M*: *M′* ∈ *carrier-mat m m* **unfolding** *gs1.M-def M′-def* **by** *auto*
have *N*: *N′* ∈ *carrier-mat m m* **unfolding** *gs2.M-def N′-def* **by** *auto*
let *?mat* = *mat-of-rows n*
let *?GsM* = *?mat ?G*
have *Gs*: *?GsM* ∈ *carrier-mat m n* **by** *auto*
hence *GsT*: $?GsM^T \in$ *carrier-mat n m* **by** *auto*
have *Gnn*: *?mat* (*RAT fs*) ∈ *carrier-mat m n* **unfolding** *mat-of-rows-def* **using** *len* **by** *auto*
have *?mat* (*RAT fs′*) = *addrow* (− *?R c*) *i j* (*?mat* (*RAT fs*))
unfolding *RAT-F1* **by** (*rule eq-matI*, *insert Gjn ji*(*2*), *auto simp*: *len mat-of-rows-def*)
also have ... = *E* ∗ *?mat* (*RAT fs*) **unfolding** *E-def*
by (*rule addrow-mat*, *insert j i*, *auto simp*: *mat-of-rows-def len*)
finally have *HEG*: *?mat* (*RAT fs′*) = *E* ∗ *?mat* (*RAT fs*) **.**
have (*E* ∗ *M′*) ∗ *?mat ?G* = *E* ∗ (*M′* ∗ *?mat ?G*)
by (*rule assoc-mult-mat*[*OF E M Gs*])
also have *M′* ∗ *?GsM* = *?mat* (*RAT fs*) **using** *gs1.matrix-equality conn1 M′-def* **by** *simp*
also have *E* ∗ ... = *?mat* (*RAT fs′*) **unfolding** *HEG* **..**
also have ... = *N′* ∗ *?mat ?G′* **using** *gs2.matrix-equality conn2* **unfolding** *N′-def* **by** *simp*
also have *?mat ?G′* = *?GsM* **unfolding** *Hs* **..**
finally have (*E* ∗ *M′*) ∗ *?GsM* = *N′* ∗ *?GsM* **.**
from *arg-cong*[*OF this*, *of* λ *x*. *x* ∗ $?GsM^T$] *E M N*
have *EMN*: (*E* ∗ *M′*) ∗ (*?GsM* ∗ $?GsM^T$) = *N′* ∗ (*?GsM* ∗ $?GsM^T$)
by (*subst* (*1 2*) *assoc-mult-mat*[*OF - Gs GsT*, *of - m*, *symmetric*], *auto*)

```
  have det (?GsM * ?GsM^T) = gs.Gramian-determinant ?G m
    unfolding gs.Gramian-determinant-def
    by (subst gs.Gramian-matrix-alt-def, auto simp: Let-def)
  also have ... > 0
  proof -
    have 1: gs.lin-indpt-list ?G
     using conn1 gs1.orthogonal-gso gs1.gso-carrier by (intro gs.orthogonal-imp-lin-indpt-list)
(auto)
    interpret G: gram-schmidt-fs-lin-indpt n ?G
      by  (standard) (use 1 gs.lin-indpt-list-def in auto)
    show ?thesis
      by (intro G.Gramian-determinant) auto
  qed
  finally have det (?GsM * ?GsM^T) ≠ 0 by simp
  from vec-space.det-nonzero-congruence[OF EMN this - - N] Gs E M
  have EMN: E * M' = N' by auto
  {
    fix i' j'
    assume ij: i' < m j' < m and choice: i' ≠ i ∨ j < j'
    have ?mu' i' j'
      = N' $$ (i',j') using ij F1 unfolding N'-def gs2.M-def by auto
   also have ... = addrow (- ?R c) i j M' $$ (i',j') unfolding EMN[symmetric]
E-def
      by (subst addrow-mat[OF M], insert ji, auto)
    also have ... = (if i = i' then - ?R c * M' $$ (j, j') + M' $$ (i', j') else M'
$$ (i', j'))
      by (rule index-mat-addrow, insert ij M, auto)
    also have ... = M' $$ (i', j')
    proof (cases i = i')
      case True
      with choice have jj: j < j' by auto
      have M' $$ (j, j') = ?mu j j'
        using ij ji len unfolding M'-def gs1.M-def by auto
      also have ... = 0 unfolding gs1.μ.simps using jj by auto
      finally show ?thesis using True by auto
    qed auto
    also have ... = ?mu i' j'
      using ij len unfolding M'-def gs1.M-def by auto
    also note calculation
  } note mu-no-change = this
  {
    fix j'
    assume jj': j' ≤ j with j i have j': j' < m by auto
    have ?mu' i j'
      = N' $$ (i,j') using jj' j i F1 unfolding N'-def gs2.M-def by auto
   also have ... = addrow (- ?R c) i j M' $$ (i,j') unfolding EMN[symmetric]
E-def
      by (subst addrow-mat[OF M], insert ji, auto)
    also have ... = - ?R c * M' $$ (j, j') + M' $$ (i, j')
```

```
    by (rule index-mat-addrow, insert j' i M, auto)
  also have ... = M' $$ (i, j') - ?R c * M' $$ (j, j') by simp
  also have M' $$ (i, j') = ?mu i j'
    using i j' len unfolding M'-def gs1.M-def by auto
  also have M' $$ (j, j') = ?mu j j'
    using i j j' len unfolding M'-def gs1.M-def by auto
  finally have ?mu' i j' = ?mu i j' - ?R c * ?mu j j' by auto
 } note mu-change = this
 show mu-update: i' < m ⟹ j' < m ⟹
  ?mu' i' j' = (if i' = i ∧ j' ≤ j then ?mu i j' - ?R c * ?mu j j' else ?mu i' j')
  for i' j' using mu-change[of j'] mu-no-change[of i' j']
  by auto
 {
  assume LLL-invariant True i fs
   from LLL-invD[OF this] have weakly-reduced fs i and sred: reduced fs i by
auto
  from red[OF this(1)] have red: weakly-reduced fs' i .
  have sred: reduced fs' i
    unfolding gram-schmidt-fs.reduced-def
  proof (intro conjI[OF red] impI allI, goal-cases)
    case (1 i' j)
    with mu-no-change[of i' j] sred[unfolded gram-schmidt-fs.reduced-def, THEN
conjunct2, rule-format, of i' j] i
    show ?case by auto
  qed
  show LLL-invariant True i fs'
    by (intro LLL-invI[OF F1 lattice ‹i ≤ m› indep-F1 sred], auto)
 }
 show Linv': LLL-invariant-weak fs'
  by (intro LLL-inv-wI[OF F1 lattice indep-F1])

 have mudiff:?mu i j - of-int c = ?mu' i j
  by (subst mu-change, auto simp: gs1.μ.simps)
 have lin-indpt-list-fs: gs.lin-indpt-list (RAT fs')
  unfolding gs.lin-indpt-list-def using conn2 by auto
 {
  assume c: c = round (μ fs i j)
  have small: abs (?mu i j - of-int c) ≤ inverse 2 unfolding j c
    using of-int-round-abs-le by (auto simp add: abs-minus-commute)
  from this[unfolded mudiff]
  show mu'-2: abs (?mu' i j) ≤ 1 / 2 by simp
  assume mu-small: μ-small-row i fs (Suc j)

  show μ-small-row i fs' j
    unfolding μ-small-row-def
  proof (intro allI, goal-cases)
    case (1 j')
    show ?case using mu'-2 mu-small[unfolded μ-small-row-def, rule-format, of
j']
```

```
      by (cases j' > j, insert mu-update[of i j'] i, auto)
    qed
  }

  {
    fix i
    assume i: i ≤ m
    have rat-of-int (d fs' i) = of-int (d fs i)
    unfolding d-def Gramian-determinant(1)[OF Linv i] Gramian-determinant(1)[OF
Linv' i]
      by (rule prod.cong[OF refl], subst eq-fs, insert i, auto)
    thus d fs' i = d fs i by simp
  } note d = this
  have D: D fs' = D fs
    unfolding D-def
    by (rule arg-cong[of - - nat], rule prod.cong[OF refl], auto simp: d)
  show LLL-measure i fs' = LLL-measure i fs
    unfolding LLL-measure-def logD-def D ..
qed
```

Addition step which can be skipped since μ-value is already small

```
lemma basis-reduction-add-row-main-0: assumes Linv: LLL-invariant-weak fs
  and i: i < m  and j: j < i
  and 0: round (μ fs i j) = 0
  and mu-small: μ-small-row i fs (Suc j)
shows μ-small-row i fs j (is ?g1)
proof −
  note inv = LLL-inv-wD[OF Linv]
  from inv(5)[OF i] inv(5)[of j] i j
  have id: fs[i := fs ! i − 0 ·_v fs ! j] = fs
    by (intro nth-equalityI, insert inv i, auto)
  show ?g1
    using basis-reduction-add-row-main[OF Linv i j -, of fs] 0 id mu-small by auto
qed

lemma μ-small-row-refl: μ-small-row i fs i
  unfolding μ-small-row-def by auto

lemma basis-reduction-add-row-done: assumes Linv: LLL-invariant True i fs
  and i: i < m
  and mu-small: μ-small-row i fs 0
shows LLL-invariant False i fs
proof −
  note inv = LLL-invD[OF Linv]
  from mu-small
  have mu-small: μ-small fs i unfolding μ-small-row-def μ-small-def by auto
  show ?thesis
    using i mu-small by (intro LLL-invI[OF inv(3,6,7,9,1,10)], auto)
qed
```

lemma *d-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 *d F1 k* = *d F2 k*
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*
hence *H*: *?F2-M* = *?F1-M* **unfolding** *swap*
by (*intro eq-matI*, *auto*)
let *?P* = *1*$_m$ *k*
have *?P* ∈ *carrier-mat k k det ?P* ∈ {−*1*, *1*} *?F2-M* = *?P* ∗ *?F1-M* **unfolding** *H* **by** *auto*
thus *?thesis* **by** *blast*
next
assume ¬ *k* < *i*
with *ki* **have** *ki*: *k* > *i* **by** *auto*
let *?P* = *swaprows-mat k i* (*i* − *1*)
from *i0 ki* **have** *neq*: *i* ≠ *i* − *1* **and** *kmi*: *i* − *1* < *k* **by** *auto*
have ∗: *?P* ∈ *carrier-mat k k det ?P* ∈ {−*1*, *1*} **using** *det-swaprows-mat*[*OF ki kmi neq*] *ki* **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 ... = *?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* ∈ *carrier-mat k k* **and** *detP*: *det P* ∈ {−*1*, *1*} **and** *H'*: *?F2-M* = *P* ∗ *?F1-M* **by** *auto*
have *d F2 k* = *det* (*gs.Gramian-matrix F2 k*)
unfolding *d-def gs.Gramian-determinant-def* **by** *simp*
also have ... = *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 ... = *P* ∗ (*?F1-M* ∗ (*?F1-M*^{T} ∗ *P*^{T})) **unfolding** *H'*
by (*subst assoc-mult-mat*[*OF P*], *auto*)
also have *det* ... = *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* ... = *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 ... = *d F1 k*
unfolding *d-def gs.Gramian-determinant-def* **by** *simp*
finally have *d F2 k* = (*det P* $*$ *det P*) $*$ *d F1 k* **by** *simp*
also have *det P* $*$ *det P* = *1* **using** *detP* **by** *auto*
finally show *d F1 k* = *d F2 k* **by** *simp*
qed

definition *base* **where** *base* = *real-of-rat* ((*4* $*$ α) / (*4* + α))

definition *g-bound* :: *int vec list* $\Rightarrow$ *bool* **where**
g-bound fs = ($\forall$ *i* < *m*. *sq-norm* (*gso fs i*) $\leq$ *of-nat N*)

end

locale *LLL-with-assms* = *LLL* +
assumes α: $\alpha \geq 4/3$
and *lin-dep*: *lin-indep fs-init*
and *len*: *length fs-init* = *m*
begin
lemma $\alpha 0$: $\alpha > 0$ $\alpha \neq 0$
using α **by** *auto*

lemma *fs-init*: *set fs-init* $\subseteq$ *carrier-vec n*
using *lin-dep*[*unfolded gs.lin-indpt-list-def*] **by** *auto*

lemma *reduction*: *0* < *reduction reduction* $\leq$ *1*
$\alpha > 4/3 \Longrightarrow$ *reduction* < *1*
$\alpha = 4/3 \Longrightarrow$ *reduction* = *1*
using α **unfolding** *reduction-def* **by** *auto*

lemma *base*: $\alpha > 4/3 \Longrightarrow$ *base* > *1* **using** *reduction*(*1*,*3*) **unfolding** *reduction-def base-def* **by** *auto*

lemma *basis-reduction-swap-main*: **assumes** *Linvw*: *LLL-invariant-weak fs*
and *small*: *LLL-invariant False i fs* $\vee$ *abs* (μ *fs i* (*i* − *1*)) $\leq$ *1/2*
and *i*: *i* < *m*
and *i0*: *i* $\neq$ *0*
and *norm-ineq*: *sq-norm* (*gso fs* (*i* − *1*)) > α $*$ *sq-norm* (*gso fs i*)
and *fs'-def*: *fs'* = *fs*[*i* := *fs* ! (*i* − *1*), *i* − *1* := *fs* ! *i*]
shows *LLL-invariant-weak fs'*
and *LLL-invariant False i fs* $\Longrightarrow$ *LLL-invariant False* (*i* − *1*) *fs'*
and *LLL-measure i fs* > *LLL-measure* (*i* − *1*) *fs'*

and $\bigwedge$ *k*. *k* < *m* $\Longrightarrow$ *gso fs' k* = (*if k* = *i* − *1 then*
gso fs i + μ *fs i* (*i* − *1*) $\cdot_v$ *gso fs* (*i* − *1*)

```
      else if k = i then
        gso fs (i − 1) − (RAT fs ! (i − 1) • gso fs′ (i − 1) / sq-norm (gso fs′ (i
− 1))) ·v gso fs′ (i − 1)
      else gso fs k) (is ⋀ k. - ⟹ - = ?newg k)

  and ⋀ k. k < m ⟹ sq-norm (gso fs′ k) = (if k = i − 1 then
        sq-norm (gso fs i) + (μ fs i (i − 1) ∗ μ fs i (i − 1)) ∗ sq-norm (gso fs (i
− 1))
      else if k = i then
        sq-norm (gso fs i) ∗ sq-norm (gso fs (i − 1)) / sq-norm (gso fs′ (i − 1))
      else sq-norm (gso fs k)) (is ⋀ k. - ⟹ - = ?new-norm k)

  and ⋀ ii j. ii < m ⟹ j < ii ⟹ μ fs′ ii j = (
       if ii = i − 1 then
         μ fs i j
       else if ii = i then
         if j = i − 1 then
           μ fs i (i − 1) ∗ sq-norm (gso fs (i − 1)) / sq-norm (gso fs′ (i − 1))
         else
           μ fs (i − 1) j
       else if ii > i ∧ j = i then
         μ fs ii (i − 1) − μ fs i (i − 1) ∗ μ fs ii i
       else if ii > i ∧ j = i − 1 then
         μ fs ii (i − 1) ∗ μ fs′ i (i − 1) + μ fs ii i ∗ sq-norm (gso fs i) / sq-norm
(gso fs′ (i − 1))
       else μ fs ii j) (is ⋀ ii j. - ⟹ - ⟹ - = ?new-mu ii j)

  and ⋀ ii. ii ≤ m ⟹ of-int (d fs′ ii) = (if ii = i then
      sq-norm (gso fs′ (i − 1)) / sq-norm (gso fs (i − 1)) ∗ of-int (d fs i)
      else of-int (d fs ii))
proof −
  note inv = LLL-inv-wD[OF Linvw]
  interpret fs: fs-int′ n m fs-init fs
    by (standard) (use Linvw in auto)
  let ?mu1 = μ fs
  let ?mu2 = μ fs′
  let ?g1 = gso fs
  let ?g2 = gso fs′
  have m12: |?mu1 i (i − 1)| ≤ inverse 2 using small
  proof
    assume LLL-invariant False i fs
    from LLL-invD(11)[OF this] i0 show ?thesis unfolding μ-small-def by auto
  qed auto
  note d = d-def
  note Gd = Gramian-determinant(1)
  note Gd12 = Gd[OF Linvw]
  let ?x = ?g1 (i − 1) let ?y = ?g1 i
  let ?cond = α ∗ sq-norm ?y < sq-norm ?x
  from inv have len: length fs = m and HC: set fs ⊆ carrier-vec n
```

```
    and L: lattice-of fs = L
    using i by auto
  from i0 inv i have swap: set fs ⊆ carrier-vec n i < length fs i − 1 < length fs i
≠ i − 1
    unfolding Let-def by auto
  have RAT-fs': RAT fs' = (RAT fs)[i := (RAT fs) ! (i − 1), i − 1 := (RAT fs)
! i]
   unfolding fs'-def using swap by (intro nth-equalityI, auto simp: nth-list-update)
  have span': gs.span (SRAT fs) = gs.span (SRAT fs') unfolding fs'-def
    by (rule arg-cong[of - - gs.span], insert swap, auto)
  have lfs': lattice-of fs' = lattice-of fs unfolding fs'-def
    by (rule lattice-of-swap[OF swap refl])
  with inv have lattice: lattice-of fs' = L by auto
  have len': length fs' = m using inv unfolding fs'-def by auto
  have fs': set fs' ⊆ carrier-vec n using swap unfolding fs'-def set-conv-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(1) have indepH: lin-indep fs .
  from i i0 len have i < length (RAT fs) i − 1 < length (RAT fs) by auto
   with distinct-swap[OF this] len have distinct (RAT fs') = distinct (RAT fs)
unfolding RAT-fs'
    by (auto simp: map-update)
  with len' fs' span' indepH have indepH': lin-indep fs' unfolding fs'-def using
i i0
    by (auto simp: gs.lin-indpt-list-def)
  have lenR': length (RAT fs') = m using len' by auto
  have conn1: set (RAT fs) ⊆ carrier-vec n  length (RAT fs) = m distinct (RAT
fs)
    gs.lin-indpt (set (RAT fs))
    using inv unfolding gs.lin-indpt-list-def by auto
  have conn2: set (RAT fs') ⊆ carrier-vec n  length (RAT fs') = m distinct (RAT
fs')
    gs.lin-indpt (set (RAT fs'))
    using indepH' lenR' unfolding gs.lin-indpt-list-def by auto
  interpret gs2: gram-schmidt-fs-lin-indpt n RAT fs'
    by (standard) (use indepH' lenR' gs.lin-indpt-list-def in auto)
  have fs'-fs: k < i − 1 ⟹ fs' ! k = fs ! k for k unfolding fs'-def by auto
  {
    fix k
    assume ki: k < i − 1
    with i have kn: k < m by simp
    have ?g2 k = ?g1 k
      by (rule gs.gso-cong, insert ki kn len, auto simp: fs'-def)
  } note G2-G = this
  have take-eq: take (Suc i − 1 − 1) fs' = take (Suc i − 1 − 1) fs
    by (intro nth-equalityI, insert len len' i swap(2−), auto intro!: fs'-fs)
  have i1n: i − 1 < m using i by auto
  let ?R = rat-of-int
  let ?RV = map-vec ?R
```

let *?f1* = λ *i*. *RAT fs* ! *i*
let *?f2* = λ *i*. *RAT fs′* ! *i*
let *?n1* = λ *i*. *sq-norm* (*?g1 i*)
let *?n2* = λ *i*. *sq-norm* (*?g2 i*)
have *heq*:*fs* ! (*i* − *1*) = *fs′* ! *i take* (*i*−*1*) *fs* = *take* (*i*−*1*) *fs′*
?f2 (*i* − *1*) = *?f1 i ?f2 i* = *?f1* (*i* − *1*)
unfolding *fs′-def* **using** *i len i0* **by** *auto*
have *norm-pos2*: *j* < *m* ⟹ *?n2 j* > *0* **for** *j*
using *gs2.sq-norm-pos len′* **by** *simp*
have *norm-pos1*: *j* < *m* ⟹ *?n1 j* > *0* **for** *j*
using *fs.gs.sq-norm-pos inv* **by** *simp*
have *norm-zero2*: *j* < *m* ⟹ *?n2 j* ≠ *0* **for** *j* **using** *norm-pos2*[*of j*] **by** *linarith*
have *norm-zero1*: *j* < *m* ⟹ *?n1 j* ≠ *0* **for** *j* **using** *norm-pos1*[*of j*] **by** *linarith*
have *gs*: ⋀ *j*. *j* < *m* ⟹ *?g1 j* ∈ *Rn* **using** *inv* **by** *blast*
have *gs2*: ⋀ *j*. *j* < *m* ⟹ *?g2 j* ∈ *Rn* **using** *fs.gs.gso-carrier conn2* **by** *auto*
have *g*: ⋀ *j*. *j* < *m* ⟹ *?f1 j* ∈ *Rn* **using** *inv* **by** *auto*
have *g2*: ⋀ *j*. *j* < *m* ⟹ *?f2 j* ∈ *Rn* **using** *gs2.f-carrier conn2* **by** *blast*
let *?fs1* = *?f1* ‘ {*0*..< (*i* − *1*)}
have *G*: *?fs1* ⊆ *Rn* **using** *g i* **by** *auto*
let *?gs1* = *?g1* ‘ {*0*..< (*i* − *1*)}
have *G′*: *?gs1* ⊆ *Rn* **using** *gs i* **by** *auto*
let *?S* = *gs.span ?fs1*
let *?S′* = *gs.span ?gs1*
have *S′S*: *?S′* = *?S*
by (*rule fs.gs.partial-span′*, *insert conn1 i*, *auto*)
have *gs.is-oc-projection* (*?g2* (*i* − *1*)) (*gs.span* (*?g2* ‘ {*0*..< (*i* − *1*)})) (*?f2* (*i* − *1*))
using *i len′* **by** (*intro gs2.gso-oc-projection-span*(*2*)) *auto*
also have *?f2* (*i* − *1*) = *?f1 i* **unfolding** *fs′-def* **using** *len i* **by** *auto*
also have *gs.span* (*?g2* ‘ {*0* ..< (*i* − *1*)}) = *gs.span* (*?f2* ‘ {*0* ..< (*i* − *1*)})
using *i len′* **by** (*intro gs2.partial-span′*) *auto*
also have *?f2* ‘ {*0* ..< (*i* − *1*)} = *?fs1*
by (*rule image-cong*[*OF refl*], *insert len i*, *auto simp*: *fs′-def*)
finally have *claim1*: *gs.is-oc-projection* (*?g2* (*i* − *1*)) *?S* (*?f1 i*) **.**
have *list-id*: [*0*..<*Suc* (*i* − *1*)] = [*0*..< *i* − *1*] @ [*i* − *1*]
[*0*..< *Suc i*] = [*0*..< *i*] @ [*i*] *map f* [*x*] = [*f x*] **for** *f x* **using** *i* **by** *auto*

have *f1i-sum*: *?f1 i* = *gs.sumlist* (*map* (λ*j*. *?mu1 i j* ·$_v$ *?g1 j*) [*0* ..< *i*]) + *?g1 i* (**is** - = *?sum* + -)
apply(*subst fs.gs.fi-is-sum-of-mu-gso*, *insert len i*, *force*)
unfolding *map-append list-id*
by (*subst gs.M.sumlist-snoc*, *insert i gs conn1*, *auto simp*: *fs.gs.μ.simps*)
have *f1im1-sum*: *?f1* (*i* − *1*) = *gs.sumlist* (*map* (λ*j*. *?mu1* (*i* − *1*) *j* ·$_v$ *?g1 j*) [*0*..<*i* − *1*]) + *?g1* (*i* − *1*) (**is** - = *?sum1* + -)
apply(*subst fs.gs.fi-is-sum-of-mu-gso*, *insert len i*, *force*)
unfolding *map-append list-id*
by (*subst gs.M.sumlist-snoc*, *insert i gs*, *auto simp*: *fs.gs.μ.simps*)

have *sum*: *?sum* ∈ *Rn* **by** (*rule gs.sumlist-carrier*, *insert gs i*, *auto*)

have *sum1*: *?sum1* ∈ *Rn* **by** (*rule gs.sumlist-carrier*, *insert gs i*, *auto*)
from *gs.span-closed*[*OF G*] **have** *S*: *?S* ⊆ *Rn* **by** *auto*
from *gs i* **have** *gs'*: ⋀ *j*. *j* < *i* − *1* ⟹ *?g1 j* ∈ *Rn* **and** *gsi*: *?g1* (*i* − *1*) ∈ *Rn* **by** *auto*
have [*0* ..< *i*] = [*0* ..< *Suc* (*i* − *1*)] **using** *i0* **by** *simp*
also have ... = [*0* ..< *i* − *1*] @ [*i* − *1*] **by** *simp*
finally have *list*: [*0* ..< *i*] = [*0* ..< *i* − *1*] @ [*i* − *1*] **.**

{
fix *k*
assume *kn*: *k* ≤ *m* **and** *ki*: *k* ≠ *i*
from *d-swap-unchanged*[*OF len i0 i ki kn fs'-def*]
have *d fs k* = *d fs' k* **by** *simp*
} **note** *d* = *this*

have *g2-im1*: *?g2* (*i* − *1*) = *?g1 i* + *?mu1 i* (*i* − *1*) ·$_v$ *?g1* (*i* − *1*) (**is** - = - + *?mu-f1*)
proof (*rule gs.is-oc-projection-eq*[*OF claim1 - S g*[*OF i*]])
show *gs.is-oc-projection* (*?g1 i* + *?mu-f1*) *?S* (*?f1 i*) **unfolding** *gs.is-oc-projection-def*
proof (*intro conjI allI impI*)
let *?sum'* = *gs.sumlist* (*map* (λ*j*. *?mu1 i j* ·$_v$ *?g1 j*) [*0* ..< *i* − *1*])
have *sum'*: *?sum'* ∈ *Rn* **by** (*rule gs.sumlist-carrier*, *insert gs i*, *auto*)
show *inRn*: (*?g1 i* + *?mu-f1*) ∈ *Rn* **using** *gs*[*OF i*] *gsi i* **by** *auto*
have *carr*: *?sum* ∈ *Rn ?g1 i* ∈ *Rn ?mu-f1* ∈ *Rn ?sum'* ∈ *Rn* **using** *sum' sum gs*[*OF i*] *gsi i* **by** *auto*
have *?f1 i* − (*?g1 i* + *?mu-f1*) = (*?sum* + *?g1 i*) − (*?g1 i* + *?mu-f1*)
unfolding *f1i-sum* **by** *simp*
also have ... = *?sum* − *?mu-f1* **using** *carr* **by** *auto*
also have *?sum* = *gs.sumlist* (*map* (λ*j*. *?mu1 i j* ·$_v$ *?g1 j*) [*0* ..< *i* − *1*] @ [*?mu-f1*])
unfolding *list* **by** *simp*
also have ... = *?sum'* + *?mu-f1*
by (*subst gs.sumlist-append*, *insert gs' gsi*, *auto*)
also have ... − *?mu-f1* = *?sum'* **using** *sum' gsi* **by** *auto*
finally have *id*: *?f1 i* − (*?g1 i* + *?mu-f1*) = *?sum'* **.**
show *?f1 i* − (*?g1 i* + *?mu-f1*) ∈ *gs.span ?S* **unfolding** *id gs.span-span*[*OF G*]
proof (*rule gs.sumlist-in-span*[*OF G*])
fix *v*
assume *v* ∈ *set* (*map* (λ*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* ∈ *?S* **unfolding** *v*
by (*rule gs.smult-in-span*[*OF G*], *unfold S'S*[*symmetric*], *rule gs.span-mem*, *insert gs i j*, *auto*)
qed
fix *x*
assume *x* ∈ *?S*
hence *x*: *x* ∈ *?S'* **using** *S'S* **by** *simp*

```
      show (?g1 i + ?mu-f1) · x = 0
      proof (rule gs.orthocompl-span[OF - G' inRn x])
        fix x
        assume x ∈ ?gs1
        then obtain j where j: j < i − 1 and x-id: x = ?g1 j by auto
        from j i x-id gs[of j] have x: x ∈ Rn by auto
        {
          fix k
          assume k: k > j k < m
          have ?g1 k · x = 0 unfolding x-id
            by (rule fs.gs.orthogonal, insert conn1 k, auto)
        }
        from this[of i] this[of i − 1] j i
        have main: ?g1 i · x = 0 ?g1 (i − 1) · x = 0 by auto
        have (?g1 i + ?mu-f1) · x = ?g1 i · x + ?mu-f1 · x
          by (rule add-scalar-prod-distrib[OF gs[OF i] - x], insert gsi, auto)
        also have ... = 0 using main
          by (subst smult-scalar-prod-distrib[OF gsi x], auto)
        finally show (?g1 i + ?mu-f1) · x = 0 .
      qed
    qed
  qed
  {
    fix k
    assume kn: k < m
      and ki: k ≠ i k ≠ i − 1
    have ?g2 k = gs.oc-projection (gs.span (?g2 ‘ {0..<k})) (?f2 k)
      by (rule gs2.gso-oc-projection-span, insert kn conn2, auto)
    also have gs.span (?g2 ‘ {0..<k}) = gs.span (?f2 ‘ {0..<k})
      by (rule gs2.partial-span', insert conn2 kn, auto)
    also have ?f2 ‘ {0..<k} = ?f1 ‘ {0..<k}
    proof(cases k≤i)
      case True hence k < i − 1 using ki by auto
       then show ?thesis apply(intro image-cong) unfolding fs'-def using len i
by auto
    next
      case False
      have ?f2 ‘ {0..<k} = (?f1 ∘ transpose i (i − 1)) ‘ {0..<k}
        unfolding transpose-def fs'-def o-def using len i
        by (intro image-cong, insert len kn, force+)
      also have ... = ?f1 ‘ {0..<k}
        apply(rule swap-image-eq) using False by auto
      finally show ?thesis.
    qed
    also have gs.span ... = gs.span (?g1 ‘ {0..<k})
      by (rule sym, rule fs.gs.partial-span', insert conn1 kn, auto)
    also have ?f2 k = ?f1 k using ki kn len unfolding fs'-def by auto
    also have gs.oc-projection (gs.span (?g1 ‘ {0..<k})) ... = ?g1 k
      by (subst fs.gs.gso-oc-projection-span, insert kn conn1, auto)
```

```
    finally have ?g2 k = ?g1 k .
  } note g2-g1-identical = this


  {
    fix jj ii
    assume ii: ii < i − 1
    have ?mu2 ii jj = ?mu1 ii jj using ii i len
      by (subst gs.μ-cong[of - - RAT fs RAT fs'], auto simp: fs'-def)
  } note mu'-mu-small-i = this
  {
    fix jj
    assume jj: jj < i − 1
    hence id1: jj < i − 1 ⟷ True jj < i ⟷ True by auto
    have id2: ?g2 jj = ?g1 jj by (subst g2-g1-identical, insert jj i, auto)
    have ?mu2 i jj = ?mu1 (i − 1) jj ?mu2 (i − 1) jj = ?mu1 i jj
      unfolding gs2.μ.simps fs.gs.μ.simps id1 id2 if-True using len i i0 by (auto
simp: fs'-def)
  } note mu'-mu-i-im1-j = this

  have im1: i − 1 < m using i by auto


  let ?g2-im1 = ?g2 (i − 1)
  have g2-im1-Rn: ?g2-im1 ∈ Rn using i conn2 by (auto intro!: fs.gs.gso-carrier)
  {
    let ?mu2-f2 = λ j. − ?mu2 i j ·v ?g2 j
    let ?sum = gs.sumlist (map (λj. − ?mu1 (i − 1) j ·v ?g1 j) [0 ..< i − 1])
   have mhs: ?mu2-f2 (i − 1) ∈ Rn using i conn2 by (auto intro!: fs.gs.gso-carrier)
    have sum': ?sum ∈ Rn by (rule gs.sumlist-carrier, insert gs i, auto)
    have gim1: ?f1 (i − 1) ∈ Rn using g i by auto
   have ?g2 i = ?f2 i + gs.sumlist (map ?mu2-f2 [0 ..< i−1] @ [?mu2-f2 (i−1)])

      unfolding gs2.gso.simps[of i] list by simp
    also have ?f2 i = ?f1 (i − 1) unfolding fs'-def using len i i0 by auto
    also have map ?mu2-f2 [0 ..< i−1] = map (λj. − ?mu1 (i − 1) j ·v ?g1 j)
[0 ..< i − 1]
    by (rule map-cong[OF refl], subst g2-g1-identical, insert i, auto simp: mu'-mu-i-im1-j)
    also have gs.sumlist (... @ [?mu2-f2 (i − 1)]) = ?sum + ?mu2-f2 (i − 1)
      by (subst gs.sumlist-append, insert gs i mhs, auto)
    also have ?f1 (i − 1) + ... = (?f1 (i − 1) + ?sum) + ?mu2-f2 (i − 1)
      using gim1 sum' mhs by auto
    also have ?f1 (i − 1) + ?sum = ?g1 (i − 1) unfolding fs.gs.gso.simps[of i
− 1] by simp
   also have ?mu2-f2 (i − 1) = − (?f2 i • ?g2-im1 / sq-norm ?g2-im1) ·v ?g2-im1
unfolding gs2.μ.simps using i0 by simp
    also have ... = − ((?f2 i • ?g2-im1 / sq-norm ?g2-im1) ·v ?g2-im1) by auto
    also have ?g1 (i − 1) + ... = ?g1 (i − 1) − ((?f2 i • ?g2-im1 / sq-norm
?g2-im1) ·v ?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)* **by** *fact*
finally have *?g2 i = ?g1 (i − 1) − (?f1 (i − 1) · ?g2 (i − 1) / sq-norm (?g2 (i − 1))) ·$_v$?g2 (i − 1)* **.**
} note *g2-i = this*

let *?n1 = λ i. sq-norm (?g1 i)*
let *?n2 = λ i. sq-norm (?g2 i)*

{
have *?n2 (i − 1) = sq-norm (?g1 i + ?mu-f1)* **unfolding** *g2-im1* **by** *simp*
also have *... = (?g1 i + ?mu-f1) · (?g1 i + ?mu-f1)*
by (*simp add*: *sq-norm-vec-as-cscalar-prod*)
also have *... = (?g1 i + ?mu-f1) · ?g1 i + (?g1 i + ?mu-f1) · ?mu-f1*
by (*rule scalar-prod-add-distrib*, *insert gs i*, *auto*)
also have *(?g1 i + ?mu-f1) · ?g1 i = ?g1 i · ?g1 i + ?mu-f1 · ?g1 i*
by (*rule add-scalar-prod-distrib*, *insert gs i*, *auto*)
also have *(?g1 i + ?mu-f1) · ?mu-f1 = ?g1 i · ?mu-f1 + ?mu-f1 · ?mu-f1*
by (*rule add-scalar-prod-distrib*, *insert gs i*, *auto*)
also have *?mu-f1 · ?g1 i = ?g1 i · ?mu-f1*
by (*rule comm-scalar-prod*, *insert gs i*, *auto*)
also have *?g1 i · ?g1 i = sq-norm (?g1 i)*
by (*simp add*: *sq-norm-vec-as-cscalar-prod*)
also have *?g1 i · ?mu-f1 = ?mu1 i (i − 1) ∗ (?g1 i · ?g1 (i − 1))*
by (*rule scalar-prod-smult-right*, *insert gs*[*OF i*] *gs*[*OF ‹i − 1 < m›*], *auto*)
also have *?g1 i · ?g1 (i − 1) = 0*
using *orthogonalD*[*OF fs.gs.orthogonal-gso*, *of i i − 1*] *i len i0*
by (*auto simp*: *o-def*)
also have *?mu-f1 · ?mu-f1 = ?mu1 i (i − 1) ∗ (?mu-f1 · ?g1 (i − 1))*
by (*rule scalar-prod-smult-right*, *insert gs*[*OF i*] *gs*[*OF ‹i − 1 < m›*], *auto*)
also have *?mu-f1 · ?g1 (i − 1) = ?mu1 i (i − 1) ∗ (?g1 (i − 1) · ?g1 (i − 1))*
by (*rule scalar-prod-smult-left*, *insert gs*[*OF i*] *gs*[*OF ‹i − 1 < m›*], *auto*)
also have *?g1 (i − 1) · ?g1 (i − 1) = sq-norm (?g1 (i − 1))*
by (*simp add*: *sq-norm-vec-as-cscalar-prod*)
finally have *?n2 (i − 1) = ?n1 i + (?mu1 i (i − 1) ∗ ?mu1 i (i − 1)) ∗ ?n1 (i − 1)*
by (*simp add*: *ac-simps o-def*)
} note *sq-norm-g2-im1 = this*

from *norm-pos1*[*OF i*] *norm-pos1*[*OF im1*] *norm-pos2*[*OF i*] *norm-pos2*[*OF im1*]
have *norm0*: *?n1 i ≠ 0 ?n1 (i − 1) ≠ 0 ?n2 i ≠ 0 ?n2 (i − 1) ≠ 0* **by** *auto*
hence *norm0′*: *?n2 (i − 1) ≠ 0* **using** *i* **by** *auto*

{
have *si*: *Suc i ≤ m* **and** *im1*: *i − 1 ≤ m* **using** *i* **by** *auto*
have *det1*: *gs.Gramian-determinant (RAT fs) (Suc i) = (∏ j<Suc i. ‖fs.gs.gso j‖^2)*

```
    using fs.gs.Gramian-determinant si len by auto
  have det2: gs.Gramian-determinant (RAT fs') (Suc i) = (∏j<Suc i. ‖gs2.gso j‖²)
    using gs2.Gramian-determinant si len' by auto
  from norm-zero1[OF less-le-trans[OF - im1]] have 0: (∏j < i−1. ?n1 j) ≠ 0

    by (subst prod-zero-iff, auto)
  have rat-of-int (d fs' (Suc i)) = rat-of-int (d fs (Suc i))
    using d-swap-unchanged[OF len i0 i - si fs'-def] by auto
  also have rat-of-int (d fs' (Suc i)) = gs.Gramian-determinant (RAT fs') (Suc i) unfolding d-def
    by (subst fs.of-int-Gramian-determinant[symmetric], insert conn2 i g fs', auto simp: set-conv-nth)
  also have ... = (∏j<Suc i. ?n2 j) unfolding det2 by (rule prod.cong, insert i, auto)
  also have rat-of-int (d fs (Suc i)) = gs.Gramian-determinant (RAT fs) (Suc i) unfolding d-def
    by (subst fs.of-int-Gramian-determinant[symmetric], insert conn1 i g, auto)
  also have ... = (∏j<Suc i. ?n1 j) unfolding det1 by (rule prod.cong, insert i, auto)
  also have {..<Suc i} = insert i (insert (i−1) {..<i−1}) (is - = ?set) by auto
  also have (∏j∈ ?set. ?n2 j) = ?n2 i * ?n2 (i − 1) * (∏j < i−1. ?n2 j) using i0
    by (subst prod.insert; (subst prod.insert)?; auto)
  also have (∏j∈ ?set. ?n1 j) = ?n1 i * ?n1 (i − 1) * (∏j < i−1. ?n1 j) using i0
    by (subst prod.insert; (subst prod.insert)?; auto)
  also have (∏j < i−1. ?n2 j) = (∏j < i−1. ?n1 j)
    by (rule prod.cong, insert G2-G, auto)
  finally have ?n2 i = ?n1 i * ?n1 (i − 1) / ?n2 (i − 1)
    using 0 norm0' by (auto simp: field-simps)
} note sq-norm-g2-i = this


{
  fix ii j
  assume ii: ii > i ii < m
   and ji: j ≠ i j ≠ i − 1
  {
    assume j: j < ii
    have ?mu2 ii j = (?f2 ii • ?g2 j) / sq-norm (?g2 j)
      unfolding gs2.μ.simps using j by auto
    also have ?f2 ii = ?f1 ii using ii len unfolding fs'-def by auto
    also have ?g2 j = ?g1 j using g2-g1-identical[of j] j ii ji by auto
    finally have ?mu2 ii j = ?mu1 ii j
      unfolding fs.gs.μ.simps using j by auto
  }
    hence ?mu2 ii j = ?mu1 ii j by (cases j < ii, auto simp: gs2.μ.simps fs.gs.μ.simps)
```

} **note** *mu-no-change-large-row = this*

{
have *?mu2 i (i − 1) = (?f2 i · ?g2 (i − 1)) / ?n2 (i − 1)*
unfolding *gs2.μ.simps* **using** *i0* **by** *auto*
also have *?f2 i · ?g2 (i − 1) = ?f1 (i − 1) · ?g2 (i − 1)*
using *len i i0* **unfolding** *fs′-def* **by** *auto*
also have *... = ?f1 (i − 1) · (?g1 i + ?mu1 i (i − 1) ·$_v$?g1 (i − 1))*
unfolding *g2-im1* **by** *simp*
also have *... = ?f1 (i − 1) · ?g1 i + ?f1 (i − 1) · (?mu1 i (i − 1) ·$_v$?g1 (i − 1))*
by (*rule scalar-prod-add-distrib*[*of - n*], *insert i gs g*, *auto*)
also have *?f1 (i − 1) · ?g1 i = 0*
by (*subst fs.gs.fi-scalar-prod-gso*, *insert conn1 im1 i i0*, *auto simp*: *fs.gs.μ.simps fs.gs.μ.simps*)
also have *?f1 (i − 1) · (?mu1 i (i − 1) ·$_v$?g1 (i − 1)) =*
?mu1 i (i − 1) ∗ (?f1 (i − 1) · ?g1 (i − 1))
by (*rule scalar-prod-smult-distrib*, *insert gs g i*, *auto*)
also have *?f1 (i − 1) · ?g1 (i − 1) = ?n1 (i − 1)*
by (*subst fs.gs.fi-scalar-prod-gso*, *insert conn1 im1*, *auto simp*: *fs.gs.μ.simps*)
finally
have *?mu2 i (i − 1) = ?mu1 i (i − 1) ∗ ?n1 (i − 1) / ?n2 (i − 1)*
by (*simp add*: *sq-norm-vec-as-cscalar-prod*)
} **note** *mu′-mu-i-im1 = this*

{
fix *ii* **assume** *iii*: *ii > i* **and** *ii*: *ii < m*
hence *iii1*: *i − 1 < ii* **by** *auto*
have *?mu2 ii (i − 1) = (?f2 ii · ?g2 (i − 1)) / ?n2 (i − 1)*
unfolding *gs2.μ.simps* **using** *i0 iii1* **by** *auto*
also have *?f2 ii · ?g2 (i−1) = ?f1 ii · ?g2 (i − 1)*
using *len i i0 iii ii* **unfolding** *fs′-def* **by** *auto*
also have *... = ?f1 ii · (?g1 i + ?mu1 i (i − 1) ·$_v$?g1 (i − 1))*
unfolding *g2-im1* **by** *simp*
also have *... = ?f1 ii · ?g1 i + ?f1 ii · (?mu1 i (i − 1) ·$_v$?g1 (i − 1))*
by (*rule scalar-prod-add-distrib*[*of - n*], *insert i ii gs g*, *auto*)
also have *?f1 ii · ?g1 i = ?mu1 ii i ∗ ?n1 i*
by (*rule fs.gs.fi-scalar-prod-gso*, *insert conn1 ii i*, *auto*)
also have *?f1 ii · (?mu1 i (i − 1) ·$_v$?g1 (i − 1)) =*
?mu1 i (i − 1) ∗ (?f1 ii · ?g1 (i − 1))
by (*rule scalar-prod-smult-distrib*, *insert gs g i ii*, *auto*)
also have *?f1 ii · ?g1 (i − 1) = ?mu1 ii (i − 1) ∗ ?n1 (i − 1)*
by (*rule fs.gs.fi-scalar-prod-gso*, *insert conn1 ii im1*, *auto*)
finally have *?mu2 ii (i − 1) = ?mu1 ii (i − 1) ∗ ?mu2 i (i − 1) + ?mu1 ii i ∗ ?n1 i / ?n2 (i − 1)*
unfolding *mu′-mu-i-im1* **using** *norm0* **by** (*auto simp*: *field-simps*)
} **note** *mu′-mu-large-row-im1 = this*

{

```
    fix ii assume iii: ii > i and ii: ii < m
    have ?mu2 ii i = (?f2 ii • ?g2 i) / ?n2 i
      unfolding gs2.μ.simps using i0 iii by auto
    also have ?f2 ii • ?g2 i = ?f1 ii • ?g2 i
      using len i i0 iii ii unfolding fs'-def by auto
    also have ... = ?f1 ii • (?g1 (i − 1) − (?f1 (i − 1) • ?g2 (i − 1) / ?n2 (i
− 1)) ·v ?g2 (i − 1))
      unfolding g2-i by simp
    also have ?f1 (i − 1) = ?f2 i using i i0 len unfolding fs'-def by auto
    also have ?f2 i • ?g2 (i − 1) / ?n2 (i − 1) = ?mu2 i (i − 1)
      unfolding gs2.μ.simps using i i0 by auto
    also have ?f1 ii • (?g1 (i − 1) − ?mu2 i (i − 1) ·v ?g2 (i − 1))
      = ?f1 ii • ?g1 (i − 1) − ?f1 ii • (?mu2 i (i − 1) ·v ?g2 (i − 1))
      by (rule scalar-prod-minus-distrib[OF g gs], insert gs2 ii i, auto)
    also have ?f1 ii • ?g1 (i − 1) = ?mu1 ii (i − 1) ∗ ?n1 (i − 1)
      by (rule fs.gs.fi-scalar-prod-gso, insert conn1 ii im1, auto)
    also have ?f1 ii • (?mu2 i (i − 1) ·v ?g2 (i − 1)) =
       ?mu2 i (i − 1) ∗ (?f1 ii • ?g2 (i − 1))
      by (rule scalar-prod-smult-distrib, insert gs gs2 g i ii, auto)
    also have ?f1 ii • ?g2 (i − 1) = (?f1 ii • ?g2 (i − 1) / ?n2 (i − 1)) ∗ ?n2
(i − 1)
      using norm0 by (auto simp: field-simps)
    also have ?f1 ii • ?g2 (i − 1) = ?f2 ii • ?g2 (i − 1)
      using len ii iii unfolding fs'-def by auto
    also have ... / ?n2 (i − 1) = ?mu2 ii (i − 1) unfolding gs2.μ.simps using
iii by auto
    finally
    have ?mu2 ii i =
       (?mu1 ii (i − 1) ∗ ?n1 (i − 1) − ?mu2 i (i − 1) ∗ ?mu2 ii (i − 1) ∗ ?n2
(i − 1)) / ?n2 i by simp
    also have ... = (?mu1 ii (i − 1) − ?mu1 i (i − 1) ∗ ?mu2 ii (i − 1)) ∗ ?n2
(i − 1) / ?n1 i
      unfolding sq-norm-g2-i mu'-mu-i-im1 using norm0 by (auto simp: field-simps)
    also have ... = (?mu1 ii (i − 1) ∗ ?n2 (i − 1) −
      ?mu1 i (i − 1) ∗ ((?mu1 ii i ∗ ?n1 i + ?mu1 i (i − 1) ∗ ?mu1 ii (i − 1) ∗
?n1 (i − 1)))) / ?n1 i
      unfolding mu'-mu-large-row-im1[OF iii ii] mu'-mu-i-im1 using norm0 by
(auto simp: field-simps)
    also have ... = ?mu1 ii (i − 1) − ?mu1 i (i − 1) ∗ ?mu1 ii i
      unfolding sq-norm-g2-im1 using norm0 by (auto simp: field-simps)
    finally have ?mu2 ii i = ?mu1 ii (i − 1) − ?mu1 i (i − 1) ∗ ?mu1 ii i .
  } note mu'-mu-large-row-i = this


  {
    fix k assume k: k < m
    show ?g2 k = ?newg k
      unfolding g2-i[symmetric]
      unfolding g2-im1[symmetric]
```

```
    using g2-g1-identical[OF k] by auto
  show ?n2 k = ?new-norm k
    unfolding sq-norm-g2-i[symmetric]
    unfolding sq-norm-g2-im1[symmetric]
    using g2-g1-identical[OF k] by auto
  fix j assume jk: j < k hence j: j < m using k by auto
  have k < i − 1 ∨ k = i − 1 ∨ k = i ∨ k > i by linarith
  thus ?mu2 k j = ?new-mu k j
    unfolding mu'-mu-i-im1[symmetric]
    using
      mu'-mu-large-row-i[OF - k]
      mu'-mu-large-row-im1 [OF - k]
      mu-no-change-large-row[OF - k, of j]
      mu'-mu-small-i
      mu'-mu-i-im1-j jk j k
    by auto
} note new-g = this

{
  note sq-norm-g2-im1
  also have ?n1 i + (?mu1 i (i − 1) ∗ ?mu1 i (i − 1)) ∗ ?n1 (i − 1)
    < 1/α ∗ (?n1 (i − 1)) + (1/2 ∗ 1/2) ∗ (?n1 (i − 1))
  proof (rule add-less-le-mono[OF - mult-mono])
    from norm-ineq[unfolded mult.commute[of α],
        THEN linordered-field-class.mult-imp-less-div-pos[OF α0(1)]]
    show ?n1 i < 1/α ∗ ?n1 (i − 1) using len i by auto
    from m12 have abs: abs (?mu1 i (i − 1)) ≤ 1/2 by auto
    have ?mu1 i (i − 1) ∗ ?mu1 i (i − 1) ≤ abs (?mu1 i (i − 1)) ∗ abs (?mu1
i (i − 1)) by auto
    also have ... ≤ 1/2 ∗ 1/2 using mult-mono[OF abs abs] by auto
    finally show ?mu1 i (i − 1) ∗ ?mu1 i (i − 1) ≤ 1/2 ∗ 1/2 by auto
  qed auto
  also have ... = reduction ∗ sq-norm (?g1 (i − 1)) unfolding reduction-def
    using α0 by (simp add: ring-distribs add-divide-distrib)
  finally have ?n2 (i − 1) < reduction ∗ ?n1 (i − 1) .
} note g-reduction = this

have lin-indpt-list-fs': gs.lin-indpt-list (RAT fs')
  unfolding gs.lin-indpt-list-def using conn2 by auto

{

  assume LLL-invariant False i fs
  note inv = LLL-invD[OF this]
  from inv have weakly-reduced fs i by auto
  hence weakly-reduced fs (i − 1) unfolding gram-schmidt-fs.weakly-reduced-def
by auto
  hence red: weakly-reduced fs' (i − 1)
    unfolding gram-schmidt-fs.weakly-reduced-def using i G2-G by simp
```

from *inv* **have** *sred*: *reduced fs i* **by** *auto*
have *sred*: *reduced fs′ (i − 1)*
unfolding *gram-schmidt-fs.reduced-def*
proof (*intro conjI*[*OF red*] *allI impI*, *goal-cases*)
case (*1 i′ j*)
with *sred* **have** |*?mu1 i′ j*| ≤ *1 / 2* **unfolding** *gram-schmidt-fs.reduced-def* **by** *auto*
thus *?case* **using** *mu′-mu-small-i*[*OF 1*(*1*)] **by** *simp*
qed
have *mu-small*: *μ-small fs′ (i − 1)*
unfolding *μ-small-def*
proof (*intro allI impI*, *goal-cases*)
case (*1 j*)
thus *?case* **using** *inv*(*11*) **unfolding** *mu′-mu-i-im1-j*[*OF 1*] *μ-small-def* **by** *auto*
qed
show *LLL-invariant False (i − 1) fs′*
by (*rule LLL-invI*, *insert lin-indpt-list-fs′ conn2 mu-small span′ lattice fs′ sred i*, *auto*)
}

show *newInvw*: *LLL-invariant-weak fs′*
by (*rule LLL-inv-wI*, *insert lin-indpt-list-fs′ conn2 span′ lattice fs′*, *auto*)

{
have *ile*: *i* ≤ *m* **using** *i* **by** *auto*
from *Gd*[*OF newInvw*, *folded d-def*, *OF ile*]
have *?R (d fs′ i)* = (∏*j*<*i*. *?n2 j*) **by** *auto*
also have ... = *prod ?n2* ({*0* ..< *i−1*} ∪ {*i − 1*})
by (*rule sym*, *rule prod.cong*, (*insert i0*, *auto*)[*1*], *insert i*, *auto*)
also have ... = *?n2 (i − 1)* ∗ *prod ?n2* ({*0* ..< *i−1*})
by *simp*
also have *prod ?n2* ({*0* ..< *i−1*}) = *prod ?n1* ({*0* ..< *i−1*})
by (*rule prod.cong*[*OF refl*], *subst g2-g1-identical*, *insert i*, *auto*)
also have ... = (*prod ?n1* ({*0* ..< *i−1*} ∪ {*i − 1*})) / *?n1 (i − 1)*
by (*subst prod.union-disjoint*, *insert norm-pos1*[*OF im1*], *auto*)
also have *prod ?n1* ({*0* ..< *i−1*} ∪ {*i − 1*}) = *prod ?n1* {*0*..<*i*}
by (*rule arg-cong*[*of - - prod ?n1*], *insert i0*, *auto*)
also have ... = (∏*j*<*i*. *?n1 j*)
by (*rule prod.cong*, *insert i0*, *auto*)
also have ... = *?R (d fs i)* **unfolding** *d-def Gd*[*OF Linvw ile*]
by (*rule prod.cong*[*OF refl*], *insert i*, *auto*)
finally have *new-di*: *?R (d fs′ i)* = *?n2 (i − 1) / ?n1 (i − 1)* ∗ *?R (d fs i)* **by** *simp*
also have ... < (*reduction* ∗ *?n1 (i − 1)*) / *?n1 (i − 1)* ∗ *?R (d fs i)*
by (*rule mult-strict-right-mono*[*OF divide-strict-right-mono*[*OF g-reduction*

norm-pos1[*OF im1*]]],
insert LLL-d-pos[*OF Linvw*] *i*, *auto*)
also have ... = *reduction* ∗ *?R* (*d fs i*) **using** *norm-pos1*[*OF im1*] **by** *auto*
finally have *d fs′ i* < *real-of-rat reduction* ∗ *d fs i*
using *of-rat-less of-rat-mult of-rat-of-int-eq* **by** *metis*
note *this new-di*
} note *d-i* = *this*
show *ii* ≤ *m* ⟹ *?R* (*d fs′ ii*) = (*if ii* = *i then ?n2* (*i* − *1*) / *?n1* (*i* − *1*) ∗ *?R* (*d fs i*) *else ?R* (*d fs ii*))
for *ii* **using** *d-i d* **by** *auto*
have *pos*: *k* < *m* ⟹ *0* < *d fs′ k k* < *m* ⟹ *0* ≤ *d fs′ k* **for** *k*
using *LLL-d-pos*[*OF newInvw*, *of k*] **by** *auto*
have *prodpos*:*0*< (∏ *i*<*m*. *d fs′ i*) **apply** (*rule prod-pos*)
using *LLL-d-pos*[*OF newInvw*] **by** *auto*
have *prod-pos′*:*0* < (∏ *x*∈{*0*..<*m*} − {*i*}. *real-of-int* (*d fs′ x*)) **apply** (*rule prod-pos*)
using *LLL-d-pos*[*OF newInvw*] *pos* **by** *auto*
have *prod-nonneg*:*0* ≤ (∏ *x*∈{*0*..<*m*} − {*i*}. *real-of-int* (*d fs′ x*)) **apply** (*rule prod-nonneg*)
using *LLL-d-pos*[*OF newInvw*] *pos* **by** *auto*
have *prodpos2*:*0*<(∏ *ia*<*m*. *d fs ia*) **apply** (*rule prod-pos*)
using *LLL-d-pos*[*OF Linvw*] **by** *auto*
have *D fs′* = *real-of-int* (∏ *i*<*m*. *d fs′ i*) **unfolding** *D-def* **using** *prodpos* **by** *simp*
also have (∏ *i*<*m*. *d fs′ i*) = (∏ *j* ∈ {*0* ..< *m*} − {*i*} ∪ {*i*}. *d fs′ j*)
by (*rule prod.cong*, *insert i*, *auto*)
also have *real-of-int* ... = *real-of-int* (∏ *j* ∈ {*0* ..< *m*} − {*i*}. *d fs′ j*) ∗ *real-of-int* (*d fs′ i*)
by (*subst prod.union-disjoint*, *auto*)
also have ... < (∏ *j* ∈ {*0* ..< *m*} − {*i*}. *d fs′ j*) ∗ (*of-rat reduction* ∗ *d fs i*)
by(*rule mult-strict-left-mono*[*OF d-i*(*1*)],*insert prod-pos′*,*auto*)
also have (∏ *j* ∈ {*0* ..< *m*} − {*i*}. *d fs′ j*) = (∏ *j* ∈ {*0* ..< *m*} − {*i*}. *d fs j*)
by (*rule prod.cong*, *insert d*, *auto*)
also have ... ∗ (*of-rat reduction* ∗ *d fs i*)
= *of-rat reduction* ∗ (∏ *j* ∈ {*0* ..< *m*} − {*i*} ∪ {*i*}. *d fs j*)
by (*subst prod.union-disjoint*, *auto*)
also have (∏ *j* ∈ {*0* ..< *m*} − {*i*} ∪ {*i*}. *d fs j*) = (∏ *j*<*m*. *d fs j*)
by (*subst prod.cong*, *insert i*, *auto*)
finally have *D*: *D fs′* < *real-of-rat reduction* ∗ *D fs*
unfolding *D-def* **using** *prodpos2* **by** *auto*
have *logD*: *logD fs′* < *logD fs*
proof (*cases* α = *4*/*3*)
case *True*
show *?thesis* **using** *D* **unfolding** *reduction*(*4*)[*OF True*] *logD-def* **unfolding** *True* **by** *simp*
next
case *False*
hence *False′*: α = *4*/*3* ⟷ *False* **by** *simp*
from *False* α **have** α > *4*/*3* **by** *simp*

with *reduction* **have** *reduction1*: *reduction* < *1* **by** *simp*
let *?new* = *real* (*D fs'*)
let *?old* = *real* (*D fs*)
let *?log* = *log* (*1*/*of-rat reduction*)
note *pos* = *LLL-D-pos*[*OF newInvw*] *LLL-D-pos*[*OF Linvw*]
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*) ∗ *?new* ≤ ((*1*/*of-rat reduction*) ∗ *of-rat reduction*) ∗ *?old*
unfolding *mult.assoc mult-le-cancel-left-pos*[*OF gediv*] **using** *D* **by** *simp*
also have (*1*/*of-rat reduction*) ∗ *of-rat reduction* = *1* **using** *reduction* **by** *auto*
finally have (*1*/*of-rat reduction*) ∗ *?new* ≤ *?old* **by** *auto*
hence *?log* ((*1*/*of-rat reduction*) ∗ *?new*) ≤ *?log ?old*
by (*subst log-le-cancel-iff*, *auto simp*: *pos reduction1 reduction*)
hence *floor* (*?log* ((*1*/*of-rat reduction*) ∗ *?new*)) ≤ *floor* (*?log ?old*)
by (*rule floor-mono*)
hence *nat* (*floor* (*?log* ((*1*/*of-rat reduction*) ∗ *?new*))) ≤ *nat* (*floor* (*?log ?old*)) **by** *simp*
also have ... = *logD fs* **unfolding** *logD-def False'* **by** *simp*
also have *?log* ((*1*/*of-rat reduction*) ∗ *?new*) = *1* + *?log ?new*
by (*subst log-mult*, *insert reduction reduction1*, *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 (*subst nat-add-distrib*, *insert pos reduction reduction1*, *auto*)
also have *nat* (*floor* (*?log ?new*)) = *logD fs'* **unfolding** *logD-def False'* **by** *simp*
finally show *logD fs'* < *logD fs* **by** *simp*
qed
show *LLL-measure i fs* > *LLL-measure* (*i* − *1*) *fs'* **unfolding** *LLL-measure-def*
using *i logD* **by** *simp*
qed

lemma *LLL-inv-initial-state*: *LLL-invariant True 0 fs-init*
proof −
from *lin-dep*[*unfolded gs.lin-indpt-list-def*]
have *set* (*RAT fs-init*) ⊆ *Rn* **by** *auto*
hence *fs-init*: *set fs-init* ⊆ *carrier-vec n* **by** *auto*
show *?thesis*
by (*rule LLL-invI*[*OF fs-init len - - lin-dep*], *auto simp*: *L-def gs.reduced-def gs.weakly-reduced-def*)
qed

lemma *LLL-inv-m-imp-reduced*: **assumes** *LLL-invariant True m fs*
shows *reduced fs m*
using *LLL-invD*[*OF assms*] **by** *blast*

lemma *basis-reduction-short-vector*: **assumes** *LLL-inv*: *LLL-invariant True m fs*
and *v*: *v* = *hd fs*

```
  and m0: m ≠ 0
shows v ∈ carrier-vec n
  v ∈ L − {0_v n}
  h ∈ L − {0_v n} ⟹ rat-of-int (sq-norm v) ≤ α ^(m − 1) * rat-of-int (sq-norm
h)
  v ≠ 0_v j
proof −
  let ?L = lattice-of fs-init
  have a1: α ≥ 1 using α by auto
  from LLL-invD[OF LLL-inv] have
    L: lattice-of fs = L
    and red: gram-schmidt-fs.weakly-reduced n (RAT fs) α (length (RAT fs))
    and basis: lin-indep fs
    and lenH: length fs = m
    and H: set fs ⊆ carrier-vec n
    by (auto simp: gs.lin-indpt-list-def gs.reduced-def)
  from lin-dep have G: set fs-init ⊆ carrier-vec n unfolding gs.lin-indpt-list-def
by auto
  with m0 len have dim-vec (hd fs-init) = n by (cases fs-init, auto)
  from v m0 lenH v have v: v = fs ! 0 by (cases fs, auto)
  interpret gs1: gram-schmidt-fs-lin-indpt n RAT fs
    by (standard) (use assms LLL-invariant-def gs.lin-indpt-list-def in auto)
  let ?r = rat-of-int
  let ?rv = map-vec ?r
  let ?F = RAT fs
  let ?h = ?rv h
  { assume h:h ∈ L − {0_v n} (is ?h-req)
    from h[folded L] have h: h ∈ lattice-of fs h ≠ 0_v n by auto
    {
      assume f: ?h = 0_v n
      have ?h = ?rv (0_v n) unfolding f by (intro eq-vecI, auto)
      hence h = 0_v n
        using of-int-hom.vec-hom-zero-iff[of h] of-int-hom.vec-hom-inj by auto
      with h have False by simp
    } hence h0: ?h ≠ 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 gs1.weakly-reduced-imp-short-vector[OF red this a1] lenH
  show h ∈ L − {0_v n} ⟹ ?r (sq-norm v) ≤ α ^(m − 1) * ?r (sq-norm h)
    using basis unfolding L v gs.lin-indpt-list-def by (auto simp: sq-norm-of-int)
  from m0 H lenH show vn: v ∈ carrier-vec n unfolding v by (cases fs, auto)
  have vL: v ∈ L unfolding L[symmetric] v using m0 H lenH
    by (intro basis-in-latticeI, cases fs, auto)
  {
    assume v = 0_v n
    hence hd ?F = 0_v n unfolding v using m0 lenH by (cases fs, auto)
    with gs.lin-indpt-list-nonzero[OF basis] have False using m0 lenH by (cases
fs, auto)
```

}
with *vL* **show** *v*: $v \in L - \{0_v\ n\}$ **by** *auto*
have *jn*:$0_v\ j \in$ *carrier-vec n* $\Longrightarrow j = n$ **unfolding** *zero-vec-def carrier-vec-def* **by** *auto*
with *v vn* **show** $v \neq 0_v\ j$ **by** *auto*
qed

lemma *LLL-mu-d-Z*: **assumes** *inv*: *LLL-invariant-weak fs*
and *j*: $j \leq ii$ **and** *ii*: $ii < m$
shows *of-int* (*d fs* (*Suc j*)) $*$ μ *fs ii j* $\in \mathbb{Z}$
proof −
interpret *fs*: *fs-int'* *n m fs-init fs*
by *standard* (*use inv* **in** *auto*)
show *?thesis*
using *assms fs.fs-int-mu-d-Z LLL-inv-wD*[*OF inv*] **unfolding** *d-def fs.d-def* **by** *auto*
qed

context fixes *fs*
assumes *Linv*: *LLL-invariant-weak fs* **and** *gbnd*: *g-bound fs*
begin

interpretation *gs1*: *gram-schmidt-fs-lin-indpt n RAT fs*
by (*standard*) (*use Linv LLL-invariant-weak-def gs.lin-indpt-list-def* **in** *auto*)

lemma *LLL-inv-N-pos*: **assumes** *m*: $m \neq 0$
shows $N > 0$
proof −
let *?r* = *rat-of-int*
note *inv* = *LLL-inv-wD*[*OF Linv*]
from *inv* **have** *F*: *RAT fs* ! 0 $\in$ *Rn fs* ! 0 $\in$ *carrier-vec n* **using** *m* **by** *auto*
from *m* **have** *upt*: [0..< *m*] = 0 # [1 ..< *m*] **using** *upt-add-eq-append*[*of 0 1 m* − 1] **by** *auto*
from *inv*(6) *m* **have** *map-vec ?r* (*fs* ! 0) $\neq 0_v$ *n* **using** *gs.lin-indpt-list-nonzero*[*OF inv*(1)]
unfolding *set-conv-nth* **by** *force*
hence *F0*: *fs* ! 0 $\neq 0_v$ *n* **by** *auto*
hence *sq-norm* (*fs* ! 0) $\neq$ 0 **using** *F* **by** *simp*
hence *1*: *sq-norm* (*fs* ! 0) $\geq$ 1 **using** *sq-norm-vec-ge-0*[*of fs* ! 0] **by** *auto*
from *gbnd m* **have** *sq-norm* (*gso fs 0*) $\leq$ *of-nat N* **unfolding** *g-bound-def* **by** *auto*
also have *gso fs 0* = *RAT fs* ! 0 **unfolding** *upt* **using** *F* **by** (*simp add*: *gs1.gso.simps*[*of 0*])
also have *RAT fs* ! 0 = *map-vec ?r* (*fs* ! 0) **using** *inv*(6) *m* **by** *auto*
also have *sq-norm* ... = *?r* (*sq-norm* (*fs* ! 0)) **by** (*simp add*: *sq-norm-of-int*)
finally show *?thesis* **using** *1* **by** (*cases N*, *auto*)
qed

lemma *d-approx-main*: **assumes** *i*: $ii \leq m$ $m \neq 0$
shows *rat-of-int* $(d\ fs\ ii) \leq$ *rat-of-nat* $(N \hat{} ii)$
proof −
note $inv = LLL\text{-}inv\text{-}wD[OF\ Linv]$
from *LLL-inv-N-pos i* **have** *A*: $0 < N$ **by** *auto*
note $main = inv(2)[unfolded\ gram\text{-}schmidt\text{-}int\text{-}def\ gram\text{-}schmidt\text{-}wit\text{-}def]$
have *rat-of-int* $(d\ fs\ ii) = (\prod j{<}ii.\ \|gso\ fs\ j\|^2)$ **unfolding** *d-def* **using** *i*
by (*auto simp*: *Gramian-determinant* $[OF\ Linv]$)
also have $\ldots \leq (\prod j{<}ii.\ of\text{-}nat\ N)$ **using** *i*
by (*intro prod-mono ballI conjI prod-nonneg*, *insert gbnd*[*unfolded g-bound-def*], *auto*)
also have $\ldots = (of\text{-}nat\ N) \hat{} ii$ **unfolding** *prod-constant* **by** *simp*
also have $\ldots =$ *of-nat* $(N \hat{} ii)$ **by** *simp*
finally show *?thesis* **by** *simp*
qed

lemma *d-approx*: **assumes** *i*: $ii < m$
shows *rat-of-int* $(d\ fs\ ii) \leq$ *rat-of-nat* $(N \hat{} ii)$
using *d-approx-main*[*of ii*] *assms* **by** *auto*

lemma *d-bound*: **assumes** *i*: $ii < m$
shows $d\ fs\ ii \leq N \hat{} ii$
using *d-approx*[*OF assms*] **unfolding** *d-def* **by** *linarith*

lemma *D-approx*: $D\ fs \leq N \hat{} (m * m)$
proof −
note $inv = LLL\text{-}inv\text{-}wD[OF\ Linv]$
from *LLL-inv-N-pos* **have** *N*: $m \neq 0 \Longrightarrow 0 < N$ **by** *auto*
note $main = inv(2)[unfolded\ gram\text{-}schmidt\text{-}int\text{-}def\ gram\text{-}schmidt\text{-}wit\text{-}def]$
have *rat-of-int* $(\prod i{<}m.\ d\ fs\ i) = (\prod i{<}m.$ *rat-of-int* $(d\ fs\ i))$ **by** *simp*
also have $\ldots \leq (\prod i{<}m.\ (of\text{-}nat\ N) \hat{}\ i)$
by (*rule prod-mono*, *insert d-approx LLL-d-pos*[*OF Linv*], *auto simp*: *less-le*)
also have $\ldots \leq (\prod i{<}m.\ (of\text{-}nat\ N \hat{}\ m))$
by (*rule prod-mono*, *insert N*, *auto intro*: *pow-mono-exp*)
also have $\ldots = (of\text{-}nat\ N) \hat{} (m * m)$ **unfolding** *prod-constant power-mult* **by** *simp*
also have $\ldots =$ *of-nat* $(N \hat{} (m * m))$ **by** *simp*
finally have $(\prod i{<}m.\ d\ fs\ i) \leq N \hat{} (m * m)$ **by** *linarith*
also have $(\prod i{<}m.\ d\ fs\ i) = D\ fs$ **unfolding** *D-def*
by (*subst nat-0-le*, *rule prod-nonneg*, *insert LLL-d-pos*[*OF Linv*], *auto simp*: *le-less*)
finally show $D\ fs \leq N \hat{} (m * m)$ **by** *linarith*
qed

lemma *LLL-measure-approx*: **assumes** $\alpha > 4/3$ $m \neq 0$
shows *LLL-measure i fs* $\leq m + 2 * m * m *$ *log base N*
proof −
have *b1*: *base* > 1 **using** *base assms* **by** *auto*
have *id*: *base* $= 1$ / *real-of-rat reduction* **unfolding** *base-def reduction-def* **using** $\alpha 0$ **by**
(*auto simp*: *field-simps of-rat-divide*)
from *LLL-D-pos*[*OF Linv*] **have** *D1*: *real* (*D fs*) ≥ 1 **by** *auto*
note *invD* = *LLL-inv-wD*[*OF Linv*]
from *invD*
have *F*: *set fs* $\subseteq$ *carrier-vec n* **and** *len*: *length fs* $= m$ **by** *auto*
have *N0*: $N > 0$ **using** *LLL-inv-N-pos*[*OF assms*(*2*)] **.**
from *D-approx*
have *D*: *D fs* $\leq N \,\hat{}\, (m * m)$ **.**
hence *real* (*D fs*) $\leq$ *real* $(N \,\hat{}\, (m * m))$ **by** *linarith*
also have $\ldots$ = *real* $N \,\hat{}\, (m * m)$ **by** *simp*
finally have *log*: *log base* (*real* (*D fs*)) $\leq$ *log base* (*real* $N \,\hat{}\, (m * m)$)
by (*subst log-le-cancel-iff*[*OF b1*], *insert D1 N0*, *auto*)

have *real* (*logD fs*) = *real* (*nat* $\lfloor$*log base* (*real* (*D fs*))$\rfloor$)
unfolding *logD-def id* **using** *assms* **by** *auto*
also have $\ldots \leq$ *log base* (*real* (*D fs*)) **using** *b1 D1* **by** *auto*
also have $\ldots \leq$ *log base* (*real* $N \,\hat{}\, (m * m)$) **by** *fact*
also have $\ldots$ = $(m * m) *$ *log base* (*real N*)
by (*rule log-nat-power*, *insert N0*, *auto*)
finally have *main*: *logD fs* $\leq m * m *$ *log base N* **by** *simp*

have *real* (*LLL-measure i fs*) = *real* $(2 *$ *logD fs* $+ m - i)$
unfolding *LLL-measure-def split invD*(*1*) **by** *simp*
also have $\ldots \leq 2 *$ *real* (*logD fs*) $+ m$ **using** *invD* **by** *simp*
also have $\ldots \leq 2 * (m * m *$ *log base N*) $+ m$ **using** *main* **by** *auto*
finally show *?thesis* **by** *simp*
qed
end

lemma *g-bound-fs-init*: *g-bound fs-init*
proof −
{
fix *i*
assume *i*: $i < m$
let *?N* = *map* (*nat o sq-norm*) *fs-init*
let *?r* = *rat-of-int*
from *i* **have** *mem*: *nat* (*sq-norm* (*fs-init* ! *i*)) $\in$ *set ?N* **using** *fs-init len* **unfolding** *set-conv-nth* **by** *force*
interpret *gs*: *gram-schmidt-fs-lin-indpt n RAT fs-init*
by (*standard*) (*use len lin-dep LLL-invariant-def gs.lin-indpt-list-def* **in** *auto*)
from *mem-set-imp-le-max-list*[*OF - mem*]
have *FN*: *nat* (*sq-norm* (*fs-init* ! *i*)) $\leq N$ **unfolding** *N-def* **by** *force*
hence $\|$*fs-init* ! *i*$\|^2 \leq$ *int N* **using** *i* **by** *auto*

```
    also have ... ≤ int (N * m) using i by fastforce
    finally have f-bnd: ‖fs-init ! i‖² ≤ int (N * m) .
    from FN have rat-of-nat (nat (sq-norm (fs-init ! i))) ≤ rat-of-nat N by simp
    also have rat-of-nat (nat (sq-norm (fs-init ! i))) = ?r (sq-norm (fs-init ! i))
      using sq-norm-vec-ge-0[of fs-init ! i] by auto
   also have ... = sq-norm (RAT fs-init ! i) unfolding sq-norm-of-int[symmetric]
using fs-init len i by auto
    finally have sq-norm (RAT fs-init ! i) ≤ rat-of-nat N .
    with gs.sq-norm-gso-le-f i len lin-dep
    have g-bnd: ‖gs.gso i‖² ≤ rat-of-nat N
      unfolding gs.lin-indpt-list-def by fastforce
    note f-bnd g-bnd
  }
  thus g-bound fs-init unfolding g-bound-def by auto
qed

lemma LLL-measure-approx-fs-init:
  LLL-invariant upw i fs-init ⟹ 4 / 3 < α ⟹ m ≠ 0 ⟹
  real (LLL-measure i fs-init) ≤ real m + real (2 * m * m) * log base (real N)
  using LLL-measure-approx[OF LLL-inv-imp-w g-bound-fs-init] .

lemma N-le-MMn: assumes m0: m ≠ 0
  shows N ≤ nat M * nat M * n
  unfolding N-def
proof (rule max-list-le, unfold set-map o-def)
  fix ni
  assume ni ∈ (λx. nat ‖x‖²) ` set fs-init
  then obtain fi where ni: ni = nat (‖fi‖²) and fi: fi ∈ set fs-init by auto
   from fi len obtain i where fii: fi = fs-init ! i and i: i < m unfolding
set-conv-nth by auto
  from fi fs-init have fi: fi ∈ carrier-vec n by auto
  let ?set = {|fs-init ! i $ j| |i j. i < m ∧ j < n} ∪ {0}
  have id: ?set = (λ (i,j). abs (fs-init ! i $ j)) ` ({0..<m} × {0..<n}) ∪ {0}
    by force
  have fin: finite ?set unfolding id by auto
  {
    fix j assume j < n
    hence M ≥ |fs-init ! i $ j| unfolding M-def using i
      by (intro Max-ge[of - abs (fs-init ! i $ j)], intro fin, auto)
  } note M = this
  from Max-ge[OF fin, of 0] have M0: M ≥ 0 unfolding M-def by auto
  have ni = nat (‖fi‖²) unfolding ni by auto
  also have ... ≤ nat (int n * ‖fi‖_∞²) using sq-norm-vec-le-linf-norm[OF fi]
    by (intro nat-mono, auto)
  also have ... = n * nat (‖fi‖_∞²)
    by (simp add: nat-mult-distrib)
  also have ... ≤ n * nat (M^2)
  proof (rule mult-left-mono[OF nat-mono])
    have fi: ‖fi‖_∞ ≤ M unfolding linf-norm-vec-def
```

```
    proof (rule max-list-le, unfold set-append set-map, rule ccontr)
      fix x
      assume x ∈ abs ‘ set (list-of-vec fi) ∪ set [0] and xM: ¬ x ≤ M
      with M0 obtain fij where fij: fij ∈ set (list-of-vec fi) and x: x = abs fij by
auto
      from fij fi obtain j where j: j < n and fij: fij = fi $ j
        unfolding set-list-of-vec vec-set-def by auto
      from M[OF j] xM[unfolded x fij fii] show False by auto
    qed auto
    show ‖fi‖∞^2 ≤ M^2 unfolding abs-le-square-iff[symmetric] using fi
      using linf-norm-vec-ge-0[of fi] by auto
  qed auto
  finally show ni ≤ nat M * nat M * n using M0
    by (subst nat-mult-distrib[symmetric], auto simp: power2-eq-square ac-simps)
qed (insert m0 len, auto)
```

9.2 Basic LLL implementation based on previous results

We now assemble a basic implementation of the LLL algorithm, where only the lattice basis is updated, and where the GSO and the μ-values are always computed from scratch. This enables a simple soundness proof and permits to separate an efficient implementation from the soundness reasoning.

```
fun basis-reduction-add-rows-loop where
  basis-reduction-add-rows-loop i fs 0 = fs
| basis-reduction-add-rows-loop i fs (Suc j) = (
    let c = round (μ fs i j);
        fs' = (if c = 0 then fs else fs[ i := fs ! i - c ·v fs ! j])
     in basis-reduction-add-rows-loop i fs' j)

definition basis-reduction-add-rows where
  basis-reduction-add-rows upw i fs =
    (if upw then basis-reduction-add-rows-loop i fs i else fs)

definition basis-reduction-swap where
  basis-reduction-swap i fs = (False, i - 1, fs[i := fs ! (i - 1), i - 1 := fs ! i])

definition basis-reduction-step where
  basis-reduction-step upw i fs = (if i = 0 then (True, Suc i, fs)
    else let
      fs' = basis-reduction-add-rows upw i fs
    in if sq-norm (gso fs' (i - 1)) ≤ α * sq-norm (gso fs' i) then
        (True, Suc i, fs')
      else basis-reduction-swap i fs')

function basis-reduction-main where
  basis-reduction-main (upw,i,fs) = (if i < m ∧ LLL-invariant upw i fs
    then basis-reduction-main (basis-reduction-step upw i fs) else
    fs)
```

by *pat-completeness auto*

definition *reduce-basis* = *basis-reduction-main* (*True*, *0*, *fs-init*)

definition *short-vector* = *hd reduce-basis*

Soundness of this implementation is easily proven

lemma *basis-reduction-add-rows-loop*: **assumes**
inv: *LLL-invariant True i fs*
and *mu-small*: *μ-small-row i fs j*
and *res*: *basis-reduction-add-rows-loop i fs j* = *fs′*
and *i*: *i* < *m*
and *j*: *j* ≤ *i*
shows *LLL-invariant False i fs′ LLL-measure i fs′* = *LLL-measure i fs*
proof (*atomize*(*full*), *insert assms*, *induct j arbitrary*: *fs*)
case (*0 fs*)
thus *?case* **using** *basis-reduction-add-row-done*[*of i fs*] **by** *auto*
next
case (*Suc j fs*)
hence *j*: *j* < *i* **by** *auto*
let *?c* = *round* (*μ fs i j*)
show *?case*
proof (*cases ?c* = *0*)
case *True*
thus *?thesis* **using** *Suc*(*1*)[*OF Suc*(*2*) *basis-reduction-add-row-main-0*[*OF LLL-inv-imp-w*[*OF Suc*(*2*)] *i j True Suc*(*3*)]]
Suc(*2*−) **by** *auto*
next
case *False*
note *step* = *basis-reduction-add-row-main*(*2*−)[*OF LLL-inv-imp-w*[*OF Suc*(*2*)] *i j refl*]
note *step* = *step*(*1*)[*OF Suc*(*2*)] *step*(*2*−)
show *?thesis* **using** *Suc*(*1*)[*OF step*(*1*−*2*)] *False Suc*(*2*−) *step*(*4*) **by** *simp*
qed
qed

lemma *basis-reduction-add-rows*: **assumes**
inv: *LLL-invariant upw i fs*
and *res*: *basis-reduction-add-rows upw i fs* = *fs′*
and *i*: *i* < *m*
shows *LLL-invariant False i fs′ LLL-measure i fs′* = *LLL-measure i fs*
proof (*atomize*(*full*), *goal-cases*)
case *1*
note *def* = *basis-reduction-add-rows-def*
show *?case*
proof (*cases upw*)
case *False*
with *res inv* **show** *?thesis* **by** (*simp add*: *def*)
next

```
    case True
    with inv have LLL-invariant True i fs by auto
    note start = this μ-small-row-reft[of i fs]
    from res[unfolded def] True have basis-reduction-add-rows-loop i fs i = fs' by
auto
    from basis-reduction-add-rows-loop[OF start this i]
    show ?thesis by auto
  qed
qed

lemma basis-reduction-swap: assumes
  inv: LLL-invariant False i fs
  and res: basis-reduction-swap i fs = (upw',i',fs')
  and cond: sq-norm (gso fs (i − 1)) > α ∗ sq-norm (gso fs i)
  and i: i < m i ≠ 0
shows LLL-invariant upw' i' fs' (is ?g1)
  LLL-measure i' fs' < LLL-measure i fs (is ?g2)
proof −
  note invw = LLL-inv-imp-w[OF inv]
  note def = basis-reduction-swap-def
  from res[unfolded basis-reduction-swap-def]
  have id: upw' = False i' = i − 1 fs' = fs[i := fs ! (i − 1), i − 1 := fs ! i] by
auto
  from basis-reduction-swap-main(2−3)[OF invw - i cond id(3)] inv show ?g1 ?g2
unfolding id by auto
qed

lemma basis-reduction-step: assumes
  inv: LLL-invariant upw i fs
  and res: basis-reduction-step upw i fs = (upw',i',fs')
  and i: i < m
shows LLL-invariant upw' i' fs' LLL-measure i' fs' < LLL-measure i fs
proof (atomize(full), goal-cases)
  case 1
  note def = basis-reduction-step-def
  note invw = LLL-inv-imp-w[OF inv]
  obtain fs'' where fs'': basis-reduction-add-rows upw i fs = fs'' by auto
  show ?case
  proof (cases i = 0)
    case True
    from increase-i[OF inv i] True
      res show ?thesis by (auto simp: def)
  next
    case False
    hence id: (i = 0) = False by auto
    note res = res[unfolded def id if-False fs'' Let-def]
    let ?x = sq-norm (gso fs'' (i − 1))
    let ?y = α ∗ sq-norm (gso fs'' i)
    from basis-reduction-add-rows[OF inv fs'' i]
```

```
    have inv: LLL-invariant False i fs''
      and meas: LLL-measure i fs'' = LLL-measure i fs by auto
    note invw = LLL-inv-imp-w[OF inv]
    show ?thesis
    proof (cases ?x ≤ ?y)
      case True
      from increase-i[OF inv i] id True res meas
      show ?thesis by simp
    next
      case gt: False
      hence ?x > ?y by auto
      from basis-reduction-swap[OF inv - this i False] gt res meas
      show ?thesis by auto
    qed
  qed
qed

termination by (relation measure (λ (upw,i,fs). LLL-measure i fs), insert ba-
sis-reduction-step, auto split: prod.splits)

declare basis-reduction-main.simps[simp del]

lemma basis-reduction-main: assumes LLL-invariant upw i fs
  and res: basis-reduction-main (upw,i,fs) = fs'
shows LLL-invariant True m fs'
  using assms
proof (induct LLL-measure i fs arbitrary: i fs upw rule: less-induct)
  case (less i fs upw)
  have id: LLL-invariant upw i fs = True using less by auto
  note res = less(3)[unfolded basis-reduction-main.simps[of upw i fs] id]
  note inv = less(2)
  note IH = less(1)
  show ?case
  proof (cases i < m)
    case i: True
    obtain i' fs' upw' where step: basis-reduction-step upw i fs = (upw',i',fs')
      (is ?step = -) by (cases ?step, auto)
    from IH[OF basis-reduction-step(2,1)[OF inv step i]] res[unfolded step] i
    show ?thesis by auto
  next
    case False
    with LLL-invD[OF inv] have i: i = m by auto
    with False res inv have LLL-invariant upw m fs' by auto
    thus LLL-invariant True m fs' unfolding LLL-invariant-def by auto
  qed
qed

lemma reduce-basis-inv: assumes res: reduce-basis = fs
  shows LLL-invariant True m fs
```

using *basis-reduction-main*[*OF LLL-inv-initial-state res*[*unfolded reduce-basis-def*]]
.

lemma *reduce-basis*: **assumes** *res*: *reduce-basis* = *fs*
shows *lattice-of fs* = *L*
reduced fs m
lin-indep fs
length fs = *m*
using *LLL-invD*[*OF reduce-basis-inv*[*OF res*]] **by** *blast+*

lemma *short-vector*: **assumes** *res*: *short-vector* = *v*
and *m0*: *m* ≠ *0*
shows *v* ∈ *carrier-vec n*
v ∈ *L* − {*0*$_v$ *n*}
h ∈ *L* − {*0*$_v$ *n*} ⟹ *rat-of-int* (*sq-norm v*) ≤ α ^ (*m* − *1*) ∗ *rat-of-int* (*sq-norm h*)
v ≠ *0*$_v$ *j*
using *basis-reduction-short-vector*[*OF reduce-basis-inv*[*OF refl*] *res*[*symmetric, unfolded short-vector-def*] *m0*]
by *blast+*
end

end

9.3 Integer LLL Implementation which Stores Multiples of the μ-Values

In this part we aim to update the integer values $d\,(j+1) * \mu_{i,j}$ as well as the Gramian determinants $d\,i$.

theory *LLL-Impl*
imports
LLL
List-Representation
Gram-Schmidt-Int
begin

9.3.1 Updates of the integer values for Swap, Add, etc.

We provide equations how to implement the LLL-algorithm by storing the integer values $d\,(j+1) * \mu_{i,j}$ and all $d\ i$ in addition to the vectors in f. Moreover, we show how to check condition like the one on norms via the integer values.

definition *round-num-denom* :: *int* ⇒ *int* ⇒ *int* **where**
round-num-denom n d = ((*2* ∗ *n* + *d*) *div* (*2* ∗ *d*))

lemma *round-num-denom*: *round-num-denom num denom* =
round (*of-int num* / *rat-of-int denom*)

proof (*cases denom = 0*)
case *False*
have *denom* $\neq$ *0* $\Longrightarrow$ *?thesis*
unfolding *round-def round-num-denom-def*
unfolding *floor-divide-of-int-eq*[**where** *?'a = rat*, *symmetric*]
by (*rule arg-cong*[*of - - floor*], *simp add*: *add-divide-distrib*)
with *False* **show** *?thesis* **by** *auto*
next
case *True*
show *?thesis* **unfolding** *True round-num-denom-def* **by** *auto*
qed

context *fs-int-indpt*
begin
lemma *round-num-denom-dμ-d*:
assumes *j*: *j* $\leq$ *i* **and** *i*: *i* $<$ *m*
shows *round-num-denom* (*dμ i j*) (*d fs* (*Suc j*)) = *round* (*gs.μ i j*)
proof −
from *j i* **have** *sj*: *Suc j* $\leq$ *m* **by** *auto*
show *?thesis* **unfolding** *round-num-denom*
by (*rule arg-cong*[*of - - round*], *subst dμ*[*OF - i*], *insert j i fs-int-d-pos*[*OF sj*], *auto*)
qed

lemma *d-sq-norm-comparison*:
assumes *quot*: *quotient-of* α = (*num*,*denom*)
and *i*: *i* $<$ *m*
and *i0*: *i* $\neq$ *0*
shows (*d fs i* $*$ *d fs i* $*$ *denom* $\leq$ *num* $*$ *d fs* (*i* − *1*) $*$ *d fs* (*Suc i*))
= (*sq-norm* (*gs.gso* (*i* − *1*)) $\leq$ α $*$ *sq-norm* (*gs.gso i*))
proof −
let *?r* = *rat-of-int*
let *?x* = *sq-norm* (*gs.gso* (*i* − *1*))
let *?y* = α $*$ *sq-norm* (*gs.gso i*)
from *i* **have** *le*: *i* − *1* $\leq$ *m* *i* $\leq$ *m Suc i* $\leq$ *m* **by** *auto*
note *pos* = *fs-int-d-pos*[*OF le*(*1*)] *fs-int-d-pos*[*OF le*(*2*)] *quotient-of-denom-pos*[*OF quot*]
have (*d fs i* $*$ *d fs i* $*$ *denom* $\leq$ *num* $*$ *d fs* (*i* − *1*) $*$ *d fs* (*Suc i*))
= (*?r* (*d fs i* $*$ *d fs i* $*$ *denom*) $\leq$ *?r* (*num* $*$ *d fs* (*i* − *1*) $*$ *d fs* (*Suc i*))) (**is** *?cond* = -) **by** *presburger*
also have ... = (*?r* (*d fs i*) $*$ *?r* (*d fs i*) $*$ *?r denom* $\leq$ *?r num* $*$ *?r* (*d fs* (*i* − *1*)) $*$ *?r* (*d fs* (*Suc i*))) **by** *simp*
also have ... = (*?r* (*d fs i*) $*$ *?r* (*d fs i*) $\leq$ α $*$ *?r* (*d fs* (*i* − *1*)) $*$ *?r* (*d fs* (*Suc i*)))
using *pos* **unfolding** *quotient-of-div*[*OF quot*] **by** (*auto simp*: *field-simps*)
also have ... = (*?r* (*d fs i*) / *?r* (*d fs* (*i* − *1*)) $\leq$ α $*$ (*?r* (*d fs* (*Suc i*)) / *?r* (*d fs i*)))
using *pos* **by** (*auto simp*: *field-simps*)
also have *?r* (*d fs i*) / *?r* (*d fs* (*i* − *1*)) = *?x* **using** *fs-int-d-Suc*[*of i* − *1*] *pos*

```
i i0
    by (auto simp: field-simps)
  also have α * (?r (d fs (Suc i)) / ?r (d fs i)) = ?y using fs-int-d-Suc[OF i] pos
i i0
    by (auto simp: field-simps)
  finally show ?cond = (?x ≤ ?y) .
qed

end


context LLL
begin

lemma d-dμ-add-row: assumes Linv: LLL-invariant-weak fs
  and i: i < m  and j: j < i
  and fs': fs' = fs[ i := fs ! i − c ·_v fs ! j]
shows

  ⋀ ii. ii ≤ m ⟹ d fs' ii = d fs ii

  ⋀ i' j'. i' < m ⟹ j' < i' ⟹
     dμ fs' i' j' = (
       if i' = i ∧ j' < j
         then dμ fs i' j' − c * dμ fs j j'
       else if i' = i ∧ j' = j
         then dμ fs i' j' − c * d fs (Suc j)
       else dμ fs i' j')
    (is ⋀ i' j'. - ⟹ - ⟹ - = ?new-mu i' j')
proof −
  interpret fs: fs-int' n m fs-init fs
    by standard (use Linv in auto)
  note add = basis-reduction-add-row-main[OF Linv i j fs']
  interpret fs': fs-int' n m fs-init fs'
    by standard (use add in auto)
  show d: ⋀ ii. ii ≤ m ⟹ d fs' ii = d fs ii by fact
  fix i' j'
  assume i': i' < m and j': j' < i'
  hence j'm: j' < m and j'': j' ≤ i' by auto
  note updates = add(7)[OF i' j'm]
  show dμ fs' i' j' = ?new-mu i' j'
  proof (cases i' = i)
    case False
    thus ?thesis using d i' j' unfolding dμ-def updates by auto
  next
    case True
    have id': d fs' (Suc j') = d fs (Suc j') by (rule d, insert i' j', auto)
    note fs'.dμ[]
    have *: rat-of-int (dμ fs' i' j') = rat-of-int (d fs' (Suc j')) * fs'.gs.μ i' j'
```

```
    unfolding dμ-def d-def
    apply(rule fs'.dμ[unfolded fs'.dμ-def fs'.d-def])
    using j' i'  LLL-inv-wD[OF add(1)]  by (auto)
  have **: rat-of-int (dμ fs i' j') = rat-of-int (d fs (Suc j')) * fs.gs.μ i' j'
    unfolding dμ-def d-def
    apply(rule fs.dμ[unfolded fs.dμ-def fs.d-def])
    using j' i' LLL-inv-wD[OF Linv]  by (auto)
  have ***: rat-of-int (dμ fs j j') = rat-of-int (d fs (Suc j')) * fs.gs.μ j j' if j' < j
    unfolding dμ-def d-def
    apply(rule fs.dμ[unfolded fs.dμ-def fs.d-def])
    using that j i LLL-inv-wD[OF Linv]  by (auto)

  show ?thesis
    apply(intro int-via-rat-eqI)
     apply(unfold if-distrib[of rat-of-int] of-int-diff of-int-mult ** * updates id'
ring-distribs)
    apply(insert True i' j' i j)
    by(auto simp: fs.gs.μ.simps algebra-simps ***)
 qed
qed

end

context LLL-with-assms
begin

lemma d-dμ-swap: assumes invw: LLL-invariant-weak fs
 and small: LLL-invariant False k fs ∨ abs (μ fs k (k − 1)) ≤ 1/2
 and k: k < m
 and k0: k ≠ 0
 and norm-ineq: sq-norm (gso fs (k − 1)) > α * sq-norm (gso fs k)
 and fs'-def: fs' = fs[k := fs ! (k − 1), k − 1 := fs ! k]
shows
 ⋀ i. i ≤ m ⟹
    d fs' i = (
      if i = k then
        (d fs (Suc k) * d fs (k − 1) + dμ fs k (k − 1) * dμ fs k (k − 1)) div d fs
k
      else d fs i)
and
 ⋀ i j. i < m ⟹ j < i ⟹
    dμ fs' i j = (
      if i = k − 1 then
         dμ fs k j
      else if i = k ∧ j ≠ k − 1 then
           dμ fs (k − 1) j
      else if i > k ∧ j = k then
        (d fs (Suc k) * dμ fs i (k − 1) − dμ fs k (k − 1) * dμ fs i j) div d fs k
      else if i > k ∧ j = k − 1 then
```

```
        (dμ fs k (k − 1) ∗ dμ fs i j + dμ fs i k ∗ d fs (k − 1)) div d fs k
      else dμ fs i j)
   (is ⋀ i j. - ⟹ - ⟹ - = ?new-mu i j)
proof −
  note swap = basis-reduction-swap-main[OF invw small k k0 norm-ineq fs'-def]
  note invw2 = swap(1)
  note swap = swap(1,3−)
  from k k0 have kk: k − 1 < k and le-m: k − 1 ≤ m k ≤ m Suc k ≤ m by auto
  from LLL-inv-wD[OF invw] have len: length fs = m by auto
  interpret fs: fs-int' n m fs-init fs
    by standard (use invw in auto)
  interpret fs': fs-int' n m fs-init fs'
    by standard (use invw2 in auto)
  let ?r = rat-of-int
  let ?n = λ i. sq-norm (gso fs i)
  let ?n' = λ i. sq-norm (gso fs' i)
  let ?dn = λ i. ?r (d fs i ∗ d fs i) ∗ ?n i
  let ?dn' = λ i. ?r (d fs' i ∗ d fs' i) ∗ ?n' i
  let ?dmu = λ i j. ?r (d fs (Suc j)) ∗ μ fs i j
  let ?dmu' = λ i j. ?r (d fs' (Suc j)) ∗ μ fs' i j
  note dmu = fs.dμ
  note dmu' = fs'.dμ
  note inv' = LLL-inv-wD[OF invw]
  have nim1: ?n k + square-rat (μ fs k (k − 1)) ∗ ?n (k − 1) =
    ?n' (k − 1) by (subst swap(4), insert k, auto)
  have ni: ?n k ∗ (?n (k − 1) / ?n' (k − 1)) = ?n' k
    by (subst swap(4)[of k], insert k k0, auto)
  have mu': μ fs k (k − 1) ∗ (?n (k − 1) / ?n' (k − 1)) = μ fs' k (k − 1)
    by (subst swap(5), insert k k0, auto)
  have fi: fs ! (k − 1) = fs' ! k fs ! k = fs' ! (k − 1)
    unfolding fs'-def using inv'(6) k k0 by auto
  let ?d'i = (d fs (Suc k) ∗ d fs (k − 1) + dμ fs k (k − 1) ∗ dμ fs k (k − 1)) div
(d fs k)
  have rat': i < m ⟹ j < i ⟹ ?r (dμ fs' i j) = ?dmu' i j for i j
     using dmu'[of j i] LLL-inv-wD[OF invw2] unfolding dμ-def fs'.dμ-def d-def
fs'.d-def by auto
   have rat: i < m ⟹ j < i ⟹ ?r (dμ fs i j) = ?dmu i j for i j
      using dmu[of j i] LLL-inv-wD[OF invw] unfolding dμ-def fs.dμ-def d-def
fs.d-def by auto
  from k k0 have sim1: Suc (k − 1) = k and km1: k − 1 < m by auto
  from LLL-d-Suc[OF invw km1, unfolded sim1]
  have dn-km1: ?dn (k − 1) = ?r (d fs k) ∗ ?r (d fs (k − 1)) by simp
  note pos = Gramian-determinant[OF invw le-refl]
  from pos(2) have ?r (gs.Gramian-determinant fs m) ≠ 0 by auto
  from this[unfolded pos(1)] have nzero: i < m ⟹ ?n i ≠ 0 for i by auto
  note pos = Gramian-determinant[OF invw2 le-refl]
  from pos(2) have ?r (gs.Gramian-determinant fs' m) ≠ 0 by auto
  from this[unfolded pos(1)] have nzero': i < m ⟹ ?n' i ≠ 0 for i by auto
  have dzero: i ≤ m ⟹ d fs i ≠ 0 for i using LLL-d-pos[OF invw, of i] by auto
```

have *dzero'*: $i \leq m \Longrightarrow d\ fs'\ i \neq 0$ **for** i **using** *LLL-d-pos*[*OF invw2*, *of i*] **by** *auto*

{
define *start* **where** $start = ?dmu'\ k\ (k - 1)$
have $start = (?n'\ (k - 1)\ /\ ?n\ (k - 1) * ?r\ (d\ fs\ k)) * \mu\ fs'\ k\ (k - 1)$
using *start-def swap*(6)[*of k*] *k k0* **by** *simp*
also have $\mu\ fs'\ k\ (k - 1) = \mu\ fs\ k\ (k - 1) * (?n\ (k - 1)\ /\ ?n'\ (k - 1))$
using *mu'* **by** *simp*
also have $(?n'\ (k - 1)\ /\ ?n\ (k - 1) * ?r\ (d\ fs\ k)) * \ldots = ?r\ (d\ fs\ k) * \mu\ fs\ k\ (k - 1)$
using *nzero*[*OF km1*] *nzero'*[*OF km1*] **by** *simp*
also have $\ldots = ?dmu\ k\ (k - 1)$ **using** *k0* **by** *simp*
finally have $?dmu'\ k\ (k - 1) = ?dmu\ k\ (k - 1)$ **unfolding** *start-def* **.**
} note *dmu-i-im1* = *this*
{
fix j
assume *j*: $j \leq m$
define *start* **where** $start = d\ fs'\ j$
{
assume *jj*: $j \neq k$
have $?r\ start = ?r\ (d\ fs'\ j)$ **unfolding** *start-def* **..**
also have $?r\ (d\ fs'\ j) = ?r\ (d\ fs\ j)$
by (*subst swap*(6), *insert j jj*, *auto*)
finally have $start = d\ fs\ j$ **by** *simp*
} note *d-j* = *this*
{
assume *jj*: $j = k$
have $?r\ start = ?r\ (d\ fs'\ k)$ **unfolding** *start-def* **unfolding** *jj* **by** *simp*
also have $\ldots = ?n'\ (k - 1)\ /\ ?n\ (k - 1) * ?r\ (d\ fs\ k)$
by (*subst swap*(6), *insert k*, *auto*)
also have $?n'\ (k - 1) = (?r\ (d\ fs\ k)\ /\ ?r\ (d\ fs\ k)) * (?r\ (d\ fs\ k)\ /\ ?r\ (d\ fs\ k))$
$* (?n\ k\ + \mu\ fs\ k\ (k - 1) * \mu\ fs\ k\ (k - 1) * ?n\ (k - 1))$
by (*subst swap*(4)[*OF km1*], *insert dzero*[*of k*], *insert k*, *simp*)
also have $?n\ (k - 1) = ?r\ (d\ fs\ k)\ /\ ?r\ (d\ fs\ (k - 1))$
unfolding *LLL-d-Suc*[*OF invw km1*, *unfolded sim1*] **using** *dzero*[*of k − 1*] *k k0* **by** *simp*
finally have $?r\ start =$
$((?r\ (d\ fs\ k) * ?n\ k) * ?r\ (d\ fs\ (k - 1)) + ?dmu\ k\ (k - 1) * ?dmu\ k\ (k - 1))$
$/ (?r\ (d\ fs\ k))$
using *k k0 dzero*[*of k*] *dzero*[*of k − 1*]
by (*simp add*: *ring-distribs*)
also have $?r\ (d\ fs\ k) * ?n\ k = ?r\ (d\ fs\ (Suc\ k))$
unfolding *LLL-d-Suc*[*OF invw k*] **by** *simp*
also have $?dmu\ k\ (k - 1) = ?r\ (d\mu\ fs\ k\ (k - 1))$ **by** (*subst rat*, *insert k k0*, *auto*)
finally have $?r\ start = (?r\ (d\ fs\ (Suc\ k) * d\ fs\ (k - 1) + d\mu\ fs\ k\ (k - 1) *$

```
dμ fs k (k − 1)))
        / (?r (d fs k)) by simp
    from division-to-div[OF this]
    have start = ?d'i .
  } note d-i = this
  from d-j d-i show d fs' j = (if j = k then ?d'i else d fs j) unfolding start-def
by auto
 }
 have length fs' = m
  using fs'-def inv'(6) by auto
 {
  fix i j
  assume i: i < m and j: j < i
  from j i have sj: Suc j ≤ m by auto
  note swaps = swap(5)[OF i j] swap(6)[OF sj]
  show dμ fs' i j = ?new-mu i j
  proof (cases i < k − 1)
   case small: True
   hence id: ?new-mu i j = dμ fs i j by auto
   show ?thesis using swaps small i j k k0 by (auto simp: dμ-def)
  next
   case False
   from j i have sj: Suc j ≤ m by auto
   let ?start = dμ fs' i j
   define start where start = ?start
   note rat'[OF i j]
   note rat-i-j = rat[OF i j]
   from False consider (i-k) i = k j = k − 1 | (i-small) i = k j ≠ k − 1 |
      (i-km1) i = k − 1 | (i-large) i > k by linarith
   thus ?thesis
   proof cases
    case ∗: i-small
    show ?thesis unfolding swaps dμ-def using ∗ i k k0 by auto
   next
    case ∗: i-k
    show ?thesis using dmu-i-im1 rat-i-j ∗ k0 by (auto simp: dμ-def)
   next
    case ∗: i-km1
    show ?thesis unfolding swaps dμ-def using ∗ i j k k0 by auto
   next
    case ∗: i-large
    consider (jj) j ≠ k − 1 j ≠ k | (ji) j = k | (jim1) j = k − 1 by linarith
    thus ?thesis
    proof cases
     case jj
     show ?thesis unfolding swaps dμ-def using ∗ i j jj k k0 by auto
    next
     case ji
     have ?r start = ?dmu' i j unfolding start-def by fact
```

also have *?r (d fs' (Suc j)) = ?r (d fs (Suc k))* **unfolding** *swaps* **unfolding** *ji* **by** *simp*
also have μ *fs' i j* = μ *fs i (k − 1) − μ fs k (k − 1) * μ fs i k*
unfolding *swaps* **unfolding** *ji* **using** *k0* * **by** *auto*
also have *?r (d fs (Suc k)) * ... = ?r (d fs (Suc k)) * ?r (d fs k) / ?r (d fs k) * ...*
using *dzero[of k] k* **by** *auto*
also have *... =*
*(?r (d fs (Suc k)) * ?dmu i (k − 1) − ?dmu k (k − 1) * ?dmu i k) / ?r (d fs k)*
using *k0* **by** (*simp add*: *field-simps*)
also have *... =*
*(?r (d fs (Suc k)) * ?r (dμ fs i (k − 1)) − ?r (dμ fs k (k − 1)) * ?r (dμ fs i k)) / ?r (d fs k)*
by (*subst* (*1 2 3*) *rat*, *insert k k0 i* *, *auto*)
also have *... = ?r (d fs (Suc k) * dμ fs i (k − 1) − dμ fs k (k − 1) * dμ fs i k) / ?r (d fs k)*
(**is** - = *of-int ?x* / -)
by *simp*
finally have *?r start = ?r ?x / ?r (d fs k)* **.**
from *division-to-div[OF this]*
have *id*: *?start = (d fs (Suc k) * dμ fs i (k − 1) − dμ fs k (k − 1) * dμ fs i j) div d fs k*
unfolding *start-def ji* **.**
show *?thesis* **unfolding** *id* **using** * *ji* **by** *simp*
next
case *jim1*
hence *id''*: *(j = k − 1) = True (j = k) = False* **using** *k0* **by** *auto*
have *?r (start) = ?dmu' i j* **unfolding** *start-def* **by** *fact*
also have μ *fs' i j* = μ *fs i (k − 1) * μ fs' k (k − 1) +*
*μ fs i k * ?n k / ?n' (k − 1)* (**is** - = *?x1 + ?x2*)
unfolding *swaps* **unfolding** *jim1* **using** *k0* * **by** *auto*
also have *?r (d fs' (Suc j)) * (?x1 + ?x2)*
*= ?r (d fs' (Suc j)) * ?x1 + ?r (d fs' (Suc j)) * ?x2* **by** (*simp add*: *ring-distribs*)
also have *?r (d fs' (Suc j)) * ?x1 = ?dmu' k (k − 1) * (?r (d fs k) * μ fs i (k − 1))*
/ ?r (d fs k)
unfolding *jim1* **using** *k0 dzero[of k] k* **by** *simp*
also have *?dmu' k (k − 1) = ?dmu k (k − 1)* **by** *fact*
also have *?r (d fs k) * μ fs i (k − 1) = ?dmu i (k − 1)* **using** *k0* **by** *simp*
also have *?r (d fs' (Suc j)) = ?n' (k − 1) * ?r (d fs k) / ?n (k − 1)*
unfolding *swaps* **unfolding** *jim1* **using** *k k0* **by** *simp*
also have *... * ?x2 = (?n k * ?r (d fs k)) / ?n (k − 1) * μ fs i k*
using *k k0 nzero'[of k − 1]* **by** *simp*
also have *?n k * ?r (d fs k) = ?r (d fs (Suc k))* **unfolding** *LLL-d-Suc[OF invw k]* **..**
also have *?r (d fs (Suc k)) / ?n (k − 1) * μ fs i k = ?dmu i k / ?n (k*

```
− 1) by simp
        also have ... = ?dmu i k * ?r (d fs (k − 1) * d fs (k − 1)) / ?dn (k −
1)
          using dzero[of k − 1] k by simp
        finally have ?r start = (?dmu k (k − 1) * ?dmu i j * ?dn (k − 1) +
           ?dmu i k * (?r (d fs (k − 1) * d fs (k − 1) * d fs k))) / (?r (d fs k) *
?dn (k − 1))
          unfolding add-divide-distrib of-int-mult jim1
          using dzero[of k − 1] nzero[of k − 1] k dzero[of k] by auto
        also have ... = (?r (dμ fs k (k − 1)) * ?r (dμ fs i j) * (?r (d fs k) * ?r
(d fs (k − 1))) +
           ?r (dμ fs i k) * (?r (d fs (k − 1) * d fs (k − 1) * d fs k))) / (?r (d fs
k) * (?r (d fs k) * ?r (d fs (k − 1))))
          unfolding dn-km1
          by (subst (1 2 3) rat, insert k k0 i * j, auto)
        also have ... = (?r (dμ fs k (k − 1)) * ?r (dμ fs i j) + ?r (dμ fs i k) *
?r (d fs (k − 1)))
            / ?r (d fs k)
          unfolding of-int-mult using dzero[of k] dzero[of k − 1] k k0 by (auto
simp: field-simps)
        also have ... = ?r (dμ fs k (k − 1) * dμ fs i j + dμ fs i k * d fs (k −
1)) / ?r (d fs k)
          (is - = of-int ?x / - )
          by simp
        finally have ?r start = ?r ?x / ?r (d fs k) .
        from division-to-div[OF this]
        have id: ?start = (dμ fs k (k − 1) * dμ fs i j + dμ fs i k * d fs (k − 1))
div (d fs k)
          unfolding start-def .
        show ?thesis unfolding id using * jim1 k0 by auto
      qed
    qed
  qed
 }
qed
end
```

9.3.2 Implementation of LLL via Integer Operations and Arrays

```
hide-fact (open) Word.inc-i

type-synonym LLL-dmu-d-state = int vec list-repr × int iarray iarray × int
iarray

fun fi-state :: LLL-dmu-d-state ⇒ int vec where
  fi-state (f,mu,d) = get-nth-i f

fun fim1-state :: LLL-dmu-d-state ⇒ int vec where
  fim1-state (f,mu,d) = get-nth-im1 f
```

```
fun d-state :: LLL-dmu-d-state ⇒ nat ⇒ int where
  d-state (f,mu,d) i = d !! i

fun fs-state :: LLL-dmu-d-state ⇒ int vec list where
  fs-state (f,mu,d) = of-list-repr f

fun upd-fi-mu-state :: LLL-dmu-d-state ⇒ nat ⇒ int vec ⇒ int iarray ⇒ LLL-dmu-d-state
where
  upd-fi-mu-state (f,mu,d) i fi mu-i = (update-i f fi, iarray-update mu i mu-i,d)

fun small-fs-state :: LLL-dmu-d-state ⇒ int vec list where
  small-fs-state (f,-) = fst f

fun dmu-ij-state :: LLL-dmu-d-state ⇒ nat ⇒ nat ⇒ int where
  dmu-ij-state (f,mu,-) i j = mu !! i !! j

fun inc-state :: LLL-dmu-d-state ⇒ LLL-dmu-d-state where
  inc-state (f,mu,d) = (inc-i f, mu, d)

fun basis-reduction-add-rows-loop where
  basis-reduction-add-rows-loop n state i j [] = state
| basis-reduction-add-rows-loop n state i sj (fj # fjs) = (
    let fi = fi-state state;
        dsj = d-state state sj;
        j = sj − 1;
        c = round-num-denom (dmu-ij-state state i j) dsj;
        state' = (if c = 0 then state else upd-fi-mu-state state i (vec n (λ i. fi $ i
− c ∗ fj $ i))
            (IArray.of-fun (λ jj. let mu = dmu-ij-state state i jj in
                if jj < j then mu − c ∗ dmu-ij-state state j jj else
                if jj = j then mu − dsj ∗ c else mu) i))
     in basis-reduction-add-rows-loop n state' i j fjs)
```

More efficient code which breaks abstraction of state.

```
lemma basis-reduction-add-rows-loop-code:
  basis-reduction-add-rows-loop n state i sj (fj # fjs) = (
    case state of ((f1,f2),mus,ds) ⇒
    let fi = hd f2;
        j = sj − 1;
        dsj = ds !! sj;
        mui = mus !! i;
        c = round-num-denom (mui !! j) dsj
     in (if c = 0 then
          basis-reduction-add-rows-loop n state i j fjs
        else
            let muj = mus !! j in
          basis-reduction-add-rows-loop n
              ((f1,  vec n (λ i. fi $ i − c ∗ fj $ i) # tl f2), iarray-update mus i
```

```
            (IArray.of-fun (λ jj. let mu = mui !! jj in
                if jj < j then mu − c ∗ muj !! jj else
                if jj = j then mu − dsj ∗ c else mu) i),
                ds) i j fjs))
proof −
  obtain f1 f2 mus ds where state: state = ((f1,f2),mus, ds) by (cases state, auto)
  show ?thesis unfolding basis-reduction-add-rows-loop.simps Let-def
    state split dmu-ij-state.simps fi-state.simps get-nth-i-def update-i-def upd-fi-mu-state.simps
     d-state.simps
    by simp
qed

lemmas basis-reduction-add-rows-loop-code-equations =
  basis-reduction-add-rows-loop.simps(1) basis-reduction-add-rows-loop-code

declare basis-reduction-add-rows-loop-code-equations[code]


definition basis-reduction-add-rows where
  basis-reduction-add-rows n upw i state =
    (if upw
       then basis-reduction-add-rows-loop n state i i (small-fs-state state)
       else state)

context
  fixes α :: rat and n m :: nat and fs-init :: int vec list
begin

definition swap-mu :: int iarray iarray ⇒ nat ⇒ int ⇒ int ⇒ int ⇒ int ⇒ int
iarray iarray where
  swap-mu dmu i dmu-i-im1 dim1 di dsi = (let im1 = i − 1 in
      IArray.of-fun (λ ii. if ii < im1 then dmu !! ii else
      if ii > i then let dmu-ii = dmu !! ii in
          IArray.of-fun (λ j. let dmu-ii-j = dmu-ii !! j in
              if j = i then (dsi ∗ dmu-ii !! im1 − dmu-i-im1 ∗ dmu-ii-j) div di
             else if j = im1 then (dmu-i-im1 ∗ dmu-ii-j + dmu-ii !! i ∗ dim1) div di
              else dmu-ii-j) ii else
      if ii = i then let mu-im1 = dmu !! im1 in
          IArray.of-fun (λ j. if j = im1 then dmu-i-im1 else mu-im1 !! j) ii
        else IArray.of-fun (λ j. dmu !! i !! j) ii) — ii = i - 1
      m)

definition basis-reduction-swap where
  basis-reduction-swap i state = (let
      di = d-state state i;
      dsi = d-state state (Suc i);
      dim1 = d-state state (i − 1);
      fi = fi-state state;
      fim1 = fim1-state state;
```

```
    dmu-i-im1 = dmu-ij-state state i (i − 1);
    fi′ = fim1;
    fim1′ = fi
  in (case state of (f,dmus,djs) ⇒
    (False, i − 1,
     (dec-i (update-im1 (update-i f fi′) fim1′),
      swap-mu dmus i dmu-i-im1 dim1 di dsi,
      iarray-update djs i ((dsi ∗ dim1 + dmu-i-im1 ∗ dmu-i-im1) div di)))))
```

More efficient code which breaks abstraction of state.

```
lemma basis-reduction-swap-code[code]:
 basis-reduction-swap i ((f1,f2), dmus, ds) = (let
    di = ds !! i;
    dsi = ds !! (Suc i);
    im1 = i − 1;
    dim1 = ds !! im1;
    fi = hd f2;
    fim1 = hd f1;
    dmu-i-im1 = dmus !! i !! im1;
    fi′ = fim1;
    fim1′ = fi
  in (False, im1,
      ((tl f1,fim1′ # fi′ # tl f2),
       swap-mu dmus i dmu-i-im1 dim1 di dsi,
       iarray-update ds i ((dsi ∗ dim1 + dmu-i-im1 ∗ dmu-i-im1) div di))))
proof −
 show ?thesis unfolding basis-reduction-swap-def split Let-def fi-state.simps fim1-state.simps
   d-state.simps get-nth-im1-def get-nth-i-def update-i-def update-im1-def dec-i-def
   by simp
qed

definition basis-reduction-step where
 basis-reduction-step upw i state = (if i = 0 then (True, Suc i, inc-state state)
   else let
     state′ = basis-reduction-add-rows n upw i state;
     di = d-state state′ i;
     dsi = d-state state′ (Suc i);
     dim1 = d-state state′ (i − 1);
     (num,denom) = quotient-of α
   in if di ∗ di ∗ denom ≤ num ∗ dim1 ∗ dsi then
       (True, Suc i, inc-state state′)
      else basis-reduction-swap i state′)

partial-function (tailrec) basis-reduction-main where
 [code]: basis-reduction-main upw i state = (if i < m
   then case basis-reduction-step upw i state of (upw′,i′,state′) ⇒
     basis-reduction-main upw′ i′ state′ else
     state)
```

definition *initial-state* = (*let*
  dmus = *dμ-impl fs-init*;
  ds = *IArray.of-fun* (λ *i*. *if i* = *0 then 1 else let i1* = *i* − *1 in dmus* !! *i1* !! *i1*)
(*Suc m*);
  dmus′ = *IArray.of-fun* (λ *i*. *let row-i* = *dmus* !! *i in*
      IArray.of-fun (λ *j*. *row-i* !! *j*) *i*) *m*
  in ((), *fs-init*), *dmus′*, *ds*) :: *LLL-dmu-d-state*)

end

definition *basis-reduction* α *n fs* = (*let m* = *length fs in*
  basis-reduction-main α *n m True 0* (*initial-state m fs*))

definition *reduce-basis* α *fs* = (*case fs of Nil* ⇒ *fs* | *Cons f* - ⇒ *fs-state* (*basis-reduction*
α (*dim-vec f*) *fs*))

definition *short-vector* α *fs* = *hd* (*reduce-basis* α *fs*)

lemma *map-rev-Suc*: *map f* (*rev* [*0*..<*Suc j*]) = *f j* # *map f* (*rev* [*0*..< *j*]) **by** *simp*

context *LLL*
begin

definition *mu-repr* :: *int iarray iarray* ⇒ *int vec list* ⇒ *bool* **where**
  mu-repr mu fs = (*mu* = *IArray.of-fun* (λ *i*. *IArray.of-fun* (*dμ fs i*) *i*) *m*)

definition *d-repr* :: *int iarray* ⇒ *int vec list* ⇒ *bool* **where**
  d-repr ds fs = (*ds* = *IArray.of-fun* (*d fs*) (*Suc m*))

fun *LLL-impl-inv* :: *LLL-dmu-d-state* ⇒ *nat* ⇒ *int vec list* ⇒ *bool* **where**
  LLL-impl-inv (*f*,*mu*,*ds*) *i fs* = (*list-repr i f* (*map* (λ *j*. *fs* ! *j*) [*0*..<*m*])
    ∧ *d-repr ds fs*
    ∧ *mu-repr mu fs*)

context fixes *state i fs upw f mu ds*
  assumes *impl*: *LLL-impl-inv state i fs*
  and *inv*: *LLL-invariant upw i fs*
  and *state*: *state* = (*f*,*mu*,*ds*)
begin
lemma *to-list-repr*: *list-repr i f* (*map* ((!) *fs*) [*0*..<*m*])
  using *impl*[*unfolded state*] **by** *auto*

lemma *to-mu-repr*: *mu-repr mu fs* **using** *impl*[*unfolded state*] **by** *auto*
lemma *to-d-repr*: *d-repr ds fs* **using** *impl*[*unfolded state*] **by** *auto*

lemma *dmu-ij-state*: **assumes** *j*: *j* < *ii*
  and *ii*: *ii* < *m*
shows *dmu-ij-state state ii j* = *dμ fs ii j*
  unfolding *to-mu-repr*[*unfolded mu-repr-def*] *state* **using** *ii j* **by** *auto*

lemma *fi-state*: $i < m \Longrightarrow$ *fi-state state* = *fs* ! *i*
using *get-nth-i*[*OF to-list-repr*(*1*)] **unfolding** *state* **by** *auto*

lemma *fim1-state*: $i < m \Longrightarrow i \neq 0 \Longrightarrow$ *fim1-state state* = *fs* ! $(i - 1)$
using *get-nth-im1*[*OF to-list-repr*(*1*)] **unfolding** *state* **by** *auto*

lemma *d-state*: $ii \leq m \Longrightarrow$ *d-state state ii* = *d fs ii*
using *to-d-repr*[*unfolded d-repr-def*] *state*
unfolding *state* **by** (*auto simp*: *nth-append*)

lemma *fs-state*: *length fs* = *m* $\Longrightarrow$ *fs-state state* = *fs*
using *of-list-repr*[*OF to-list-repr*(*1*)] **unfolding** *state* **by** (*auto simp*: *o-def intro*!: *nth-equalityI*)

lemma *LLL-state-inc-state*: **assumes** *i*: $i < m$
shows *LLL-impl-inv* (*inc-state state*) (*Suc i*) *fs*
fs-state (*inc-state state*) = *fs-state state*
proof −
from *LLL-invD*[*OF inv*] **have** *len*: *length fs* = *m* **by** *auto*
note *inc* = *inc-i*[*OF to-list-repr*(*1*)]
from *inc i impl* **show** *LLL-impl-inv* (*inc-state state*) (*Suc i*) *fs*
unfolding *state* **by** *auto*
from *of-list-repr*[*OF inc*(*1*)] *of-list-repr*[*OF to-list-repr*(*1*)] *i*
show *fs-state* (*inc-state state*) = *fs-state state* **unfolding** *state* **by** *auto*
qed
end
end

context *LLL-with-assms*
begin

lemma *basis-reduction-add-rows-loop-impl*: **assumes**
impl: *LLL-impl-inv state i fs*
and *inv*: *LLL-invariant True i fs*
and *mu-small*: μ-*small-row i fs j*
and *res*: *LLL-Impl.basis-reduction-add-rows-loop n state i j*
(*map* ((!) *fs*) (*rev* [*0* ..< *j*])) = *state*′
(**is** *LLL-Impl.basis-reduction-add-rows-loop n state i j* (*?mapf fs j*) = -)
and *j*: $j \leq i$
and *i*: $i < m$
and *fs*′: *fs*′ = *fs-state state*′
shows
LLL-impl-inv state′ *i fs*′
basis-reduction-add-rows-loop i fs j = *fs*′
proof (*atomize*(*full*), *insert assms*(*1*−*6*), *induct j arbitrary*: *fs state*)
case (*0 fs state*)
from *LLL-invD*[*OF 0*(*2*)] **have** *len*: *length fs* = *m* **by** *auto*
from *fs-state*[*OF 0*(*1*−*2*) - *len*] **have** *fs-state state* = *fs* **by** (*cases state*, *auto*)

```
  thus ?case using 0 i fs' by auto
next
  case (Suc j fs state)
  hence j: j < i and jj: j ≤ i and id: (j < i) = True by auto
  obtain f mu ds where state: state = (f,mu,ds) by (cases state, auto)
  note Linv = Suc(3)
  note inv = LLL-invD[OF Linv]
  note impl = Suc(2)
  from fi-state[OF impl Linv state i] have fi: fi-state state = fs ! i by auto
  have id: Suc j - 1 = j by simp
  note mu = dmu-ij-state[OF impl Linv state j i]
  let ?c = round (μ fs i j)
  note Linvw = LLL-inv-imp-w[OF Linv]
  interpret fs: fs-int' n m fs-init fs
    by standard (use Linvw in auto)
  have floor: round-num-denom (dμ fs i j) (d fs (Suc j)) = round (fs.gs.μ i j)
    using jj i inv unfolding dμ-def d-def
    by (intro fs.round-num-denom-dμ-d[unfolded fs.dμ-def fs.d-def]) auto
  from LLL-d-pos[OF Linvw] j i have dj: d fs (Suc j) > 0 by auto
  note updates = d-dμ-add-row[OF Linvw i j refl]
  note d-state = d-state[OF impl Linv state]
  from d-state[of Suc j] j i have djs: d-state state (Suc j) = d fs (Suc j) by auto
 note res = Suc(5)[unfolded floor map-rev-Suc djs append.simps LLL-Impl.basis-reduction-add-rows-loop.simps
     fi Let-def mu id int-times-rat-def]
  show ?case
  proof (cases ?c = 0)
    case True
    from res[unfolded True]
     have res: LLL-Impl.basis-reduction-add-rows-loop n state i j (?mapf fs j) =
state'
      by simp
    note step = Linv basis-reduction-add-row-main-0[OF Linvw i j True Suc(4)]
    show ?thesis using Suc(1)[OF impl step(1-2) res - i] j True by auto
  next
    case False
    hence id: (?c = 0) = False by auto
    from i j have jm: j < m by auto
    have idd: vec n (λia. fs ! i $ ia - ?c * fs ! j $ ia) =
      fs ! i - ?c ·v fs ! j
      by (intro eq-vecI, insert inv(4)[OF i] inv(4)[OF jm], auto)
    define fi' where fi' = fs ! i - ?c ·v fs ! j
    obtain fs'' where fs'': fs[i := fs ! i - ?c ·v fs ! j] = fs'' by auto
    note step = basis-reduction-add-row-main[OF Linvw i j fs''[symmetric]]
    note Linvw2 = step(1)
    note step = step(2)[OF Linv] step(3,5-)
    note updates = updates[where c = ?c, unfolded fs'']
    have map-id-f: ?mapf fs j = ?mapf fs'' j
      by (rule nth-equalityI, insert j i, auto simp: rev-nth fs''[symmetric])
    have nth-id: [0..<m] ! i = i using i by auto
```

note *res* = *res*[*unfolded False map-id-f id if-False idd*]
have *fi*: *fi'* = *fs''* ! *i* **unfolding** *fs''*[*symmetric*] *fi'-def* **using** *inv*(*6*) *i* **by** *auto*
let *?fn* = λ *fs i*. (*fs* ! *i*, *sq-norm* (*gso fs i*))
let *?d* = λ *fs i*. *d fs* (*Suc i*)
let *?mu'* = *IArray.of-fun*
(λ*jj*. *if jj* < *j then dmu-ij-state state i jj* − *?c* ∗ *dmu-ij-state state j jj*
else if jj = *j then dmu-ij-state state i jj* − *?d fs j* ∗ *?c else dmu-ij-state state i jj*) *i*
have *mu'*: *?mu'* = *IArray.of-fun* (*dμ fs'' i*) *i* (**is** - = *?mu'i*)
proof (*rule iarray-cong'*, *goal-cases*)
case (*1 jj*)
from *1 j i* **have** *jm*: *j* < *m* **by** *auto*
show *?case* **unfolding** *dmu-ij-state*[*OF impl Linv state 1 i*] **using** *dmu-ij-state*[*OF impl Linv state - jm*]
by (*subst updates*(*2*)[*OF i 1*], *auto*)
qed
{
fix *ii*
assume *ii*: *ii* < *m ii* ≠ *i*
hence (*IArray.of-fun* (λ*i*. *IArray.of-fun* (*dμ fs i*) *i*) *m*) !! *ii*
= *IArray.of-fun* (*dμ fs ii*) *ii* **by** *auto*
also have ... = *IArray.of-fun* (*dμ fs'' ii*) *ii*
proof (*rule iarray-cong'*, *goal-cases*)
case (*1 j*)
with *ii* **have** *j*: *Suc j* ≤ *m* **by** *auto*
show *?case* **unfolding** *updates*(*2*)[*OF ii*(*1*) *1*] **using** *ii* **by** *auto*
qed
finally have (*IArray.of-fun* (λ*i*. *IArray.of-fun* (*dμ fs i*) *i*) *m*) !! *ii*
= *IArray.of-fun* (*dμ fs'' ii*) *ii* **by** *auto*
} **note** *ii* = *this*
let *?mu''* = *iarray-update mu i* (*IArray.of-fun* (*dμ fs'' i*) *i*)
have *new-array*: *?mu''* = *IArray.of-fun* (λ *i*. *IArray.of-fun* (*dμ fs'' i*) *i*) *m*
unfolding *iarray-update-of-fun to-mu-repr*[*OF impl Linv state*, *unfolded mu-repr-def*]
by (*rule iarray-cong'*, *insert ii*, *auto*)
have *d'*: (*map* (*?d fs*) (*rev* [*0*..<*j*])) = (*map* (*?d fs''*) (*rev* [*0*..<*j*]))
by (*rule nth-equalityI*, *force*, *simp*, *subst updates*(*1*), *insert j i*, *auto*
simp: *rev-nth*)
have *repr-id*:
(*map* ((!) *fs*) [*0*..<*m*])[*i* := (*fs''* ! *i*)] = *map* ((!) *fs''*) [*0*..<*m*] (**is** *?xs* = *?ys*)
proof (*rule nth-equalityI*, *force*)
fix *j*
assume *j* < *length ?xs*
thus *?xs* ! *j* = *?ys* ! *j* **unfolding** *fs''*[*symmetric*] *i* **by** (*cases j* = *i*, *auto*)
qed
have *repr-id-d*:
map (*d fs*) [*0*..<*Suc m*] = *map* (*d fs''*) [*0*..<*Suc m*]
by (*rule nth-equalityI*, *force*, *insert step*(*4*,*6*), *auto simp*: *nth-append*)
have *mu*: *fs* ! *i* − *?c* ·$_v$ *fs* ! *j* = *fs''* ! *i* **unfolding** *fs''*[*symmetric*] **using** *inv*(*6*)

i **by** *auto*
note *res = res*[*unfolded mu' mu d'*]
show *?thesis* **unfolding** *basis-reduction-add-rows-loop.simps Let-def id if-False fs''*
proof (*rule Suc*(*1*)[*OF - step*(*1*,*2*) *res - i*])
note *list-repr = to-list-repr*[*OF impl Linv state*]
from *i* **have** *ii*: *i < length* [*0..<m*] **by** *auto*
show *LLL-impl-inv* (*upd-fi-mu-state state i* (*fs''* ! *i*) *?mu'i*) *i fs''*
unfolding *upd-fi-mu-state.simps state LLL-impl-inv.simps new-array*
proof (*intro conjI*)
show *list-repr i* (*update-i f* (*fs''* ! *i*)) (*map* ((!) *fs''*) [*0..<m*])
using *update-i*[*OF list-repr*(*1*), *unfolded length-map*, *OF ii*] **unfolding** *repr-id*[*symmetric*] **.**
show *d-repr ds fs''* **unfolding** *to-d-repr*[*OF impl Linv state*, *unfolded d-repr-def*] *d-repr-def*
by (*rule iarray-cong'*, *subst step*(*6*), *auto*)
qed (*auto simp*: *mu-repr-def*)
qed (*insert i j*, *auto simp*: *Suc*(*4*))
qed
qed

lemma *basis-reduction-add-rows-loop*: **assumes**
impl: *LLL-impl-inv state i fs*
and *inv*: *LLL-invariant True i fs*
and *mu-small*: *μ-small-row i fs j*
and *res*: *LLL-Impl.basis-reduction-add-rows-loop n state i j*
(*map* ((!) *fs*) (*rev* [*0* ..< *j*])) = *state'*
(**is** *LLL-Impl.basis-reduction-add-rows-loop n state i j* (*?mapf fs j*) = -)
and *j*: *j* ≤ *i*
and *i*: *i < m*
and *fs'*: *fs'* = *fs-state state'*
shows
LLL-impl-inv state' i fs'
LLL-invariant False i fs'
LLL-measure i fs' = *LLL-measure i fs*
basis-reduction-add-rows-loop i fs j = *fs'*
using *basis-reduction-add-rows-loop-impl*[*OF assms*]
basis-reduction-add-rows-loop[*OF inv mu-small - i j*] **by** *blast+*

lemma *basis-reduction-add-rows-impl*: **assumes**
impl: *LLL-impl-inv state i fs*
and *inv*: *LLL-invariant upw i fs*
and *res*: *LLL-Impl.basis-reduction-add-rows n upw i state = state'*
and *i*: *i < m*
and *fs'*: *fs'* = *fs-state state'*
shows
LLL-impl-inv state' i fs'
basis-reduction-add-rows upw i fs = fs'
proof (*atomize*(*full*), *goal-cases*)

```
  case 1
  obtain f mu ds where state: state = (f,mu,ds) by (cases state, auto)
  note def = LLL-Impl.basis-reduction-add-rows-def basis-reduction-add-rows-def
  show ?case
  proof (cases upw)
    case False
    from LLL-invD[OF inv] have len: length fs = m by auto
    from fs-state[OF impl inv state len] have fs-state state = fs by auto
    with assms False show ?thesis by (auto simp: def)
  next
    case True
    with inv have LLL-invariant True i fs by auto
    note start = this μ-small-row-refl[of i fs]
    have id: small-fs-state state = map (λ i. fs ! i) (rev [0..<i])
      unfolding state using to-list-repr[OF impl inv state] i
      unfolding list-repr-def by (auto intro!: nth-equalityI simp: rev-nth min-def)
    from i have mm: [0..<m] = [0 ..< i] @ [i] @ [Suc i ..< m]
      by (intro nth-equalityI, auto simp: nth-append nth-Cons split: nat.splits)
    from res[unfolded def] True
     have LLL-Impl.basis-reduction-add-rows-loop n state i i (small-fs-state state)
= state' by auto
    from basis-reduction-add-rows-loop-impl[OF impl start(1−2) this[unfolded id]
le-refl i fs']
    show ?thesis unfolding def using True by auto
  qed
qed

lemma basis-reduction-add-rows: assumes
     impl: LLL-impl-inv state i fs
  and inv: LLL-invariant upw i fs
  and res: LLL-Impl.basis-reduction-add-rows n upw i state = state'
  and i: i < m
  and fs': fs' = fs-state state'
shows
  LLL-impl-inv state' i fs'
  LLL-invariant False i fs'
  LLL-measure i fs' = LLL-measure i fs
  basis-reduction-add-rows upw i fs = fs'
  using basis-reduction-add-rows-impl[OF impl inv res i fs']
    basis-reduction-add-rows[OF inv - i] by blast+

lemma basis-reduction-swap-impl: assumes
  impl: LLL-impl-inv state i fs
  and inv: LLL-invariant False i fs
  and res: LLL-Impl.basis-reduction-swap m i state = (upw',i',state')
  and cond: sq-norm (gso fs (i − 1)) > α ∗ sq-norm (gso fs i)
  and i: i < m and i0: i ≠ 0
  and fs': fs' = fs-state state'
shows
```

LLL-impl-inv state' i' fs' (**is** *?g1*)
basis-reduction-swap i fs = (*upw'*,*i'*,*fs'*) (**is** *?g2*)
proof −
note *invw* = *LLL-inv-imp-w*[*OF inv*]
from *i i0* **have** *ii*: *i* − *1* < *i* **and** *le-m*: *i* − *1* ≤ *m i* ≤ *m Suc i* ≤ *m* **by** *auto*
obtain *f mu ds* **where** *state*: *state* = (*f*,*mu*,*ds*) **by** (*cases state*, *auto*)
note *dmu-ij-state* = *dmu-ij-state*[*OF impl inv state*]
note *d-state* = *d-state*[*OF impl inv state*]
note *res* = *res*[*unfolded LLL-Impl.basis-reduction-swap-def Let-def split state*, *folded state*,
unfolded fi-state[*OF impl inv state i*] *fim1-state*[*OF impl inv state i i0*]]
note *state-id* = *dmu-ij-state*[*OF ii i*]
note *d-state-i* = *d-state*[*OF le-m*(*1*)] *d-state*[*OF le-m*(*2*)] *d-state*[*OF le-m*(*3*)]
from *LLL-invD*[*OF inv*] **have** *len*: *length fs* = *m* **by** *auto*
from *fs-state*[*OF impl inv state len*] **have** *fs*: *fs-state state* = *fs* **by** *auto*
obtain *fs''* **where** *fs''*: *fs*[*i* := *fs* ! (*i* − *1*), *i* − *1* := *fs* ! *i*] = *fs''* **by** *auto*
let *?r* = *rat-of-int*
let *?d* = *d fs*
let *?d'* = *d fs''*
let *?dmus* = *dmu-ij-state state*
let *?ds* = *d-state state*
note *swap* = *basis-reduction-swap-main*[*OF invw disjI1*[*OF inv*] *i i0 cond refl*, *unfolded fs''*]
note *invw2* = *swap*(*1*)
note *swap* = *swap*(*2*)[*OF inv*] *swap*(*3*−)
interpret *fs*: *fs-int' n m fs-init fs*
by *standard* (*use invw* **in** *auto*)
interpret *fs''*: *fs-int' n m fs-init fs''*
by *standard* (*use invw2* **in** *auto*)
note *dmu* = *fs*.*dμ*
note *dmu'* = *fs''*.*dμ*
note *inv'* = *LLL-invD*[*OF inv*]
have *fi*: *fs* ! (*i* − *1*) = *fs''* ! *i fs* ! *i* = *fs''* ! (*i* − *1*)
unfolding *fs''*[*symmetric*] **using** *inv'*(*6*) *i i0* **by** *auto*
from *res* **have** *upw'*: *upw'* = *False i'* = *i* − *1* **by** *auto*
let *?dmu-repr'* = *swap-mu m mu i* (*?dmus i* (*i* − *1*)) (*?d* (*i* − *1*)) (*?d i*) (*?d* (*Suc i*))
let *?d'i* = (*?d* (*Suc i*) ∗ *?d* (*i* − *1*) + *?dmus i* (*i* − *1*) ∗ *?dmus i* (*i* − *1*)) *div* (*?d i*)
from *res*[*unfolded fi d-state-i*]
have *res*: *upw'* = *False i'* = *i* − *1*
state' = (*dec-i* (*update-im1* (*update-i f* (*fs''* ! *i*)) (*fs''* ! (*i* − *1*))),
?dmu-repr', *iarray-update ds i ?d'i*) **by** *auto*
from *i* **have** *ii*: *i* < *length* [*0*..<*m*] **and** *im1*: *i* − *1* < *m* **by** *auto*
note *list-repr* = *to-list-repr*[*OF impl inv state*]
from *dec-i*[*OF update-im1*[*OF update-i*[*OF list-repr*(*1*)]], *unfolded length-map*, *OF ii i0 i0*]
have
list-repr (*i* − *1*) (*dec-i* (*update-im1* (*update-i f* (*fs''* ! *i*)) (*fs''* ! (*i* − *1*)))) ((*map*

```
((!) fs) [0..<m])[i := (fs'' ! i),
      i − 1 := (fs'' ! (i − 1))]) (is list-repr - ?fr ?xs) .
  also have ?xs = map ((!) fs'') [0..<m] unfolding fs''[symmetric]
    by (intro nth-equalityI, insert i i0 len, auto simp: nth-append, rename-tac ii,
case-tac ii ∈ {i−1,i}, auto)
  finally have f-repr: list-repr (i − 1) ?fr (map ((!) fs'') [0..<m]) .
  from i0 have sim1: Suc (i − 1) = i by simp
  from LLL-d-Suc[OF invw im1, unfolded sim1]
  have length fs'' = m
    using fs'' inv'(6) by auto
   hence fs-id: fs' = fs'' unfolding fs' res fs-state.simps using of-list-repr[OF
f-repr]
    by (intro nth-equalityI, auto simp: o-def)
  from to-mu-repr[OF impl inv state] have mu: mu-repr mu fs by auto
  from to-d-repr[OF impl inv state] have d-repr: d-repr ds fs by auto
  note mu-def = mu[unfolded mu-repr-def]
  note updates = d-dμ-swap[OF invw disjI1[OF inv] i i0 cond fs''[symmetric]]
  note dmu-ii = dmu-ij-state[OF ‹i − 1 < i› i]
  show ?g1 unfolding fs-id LLL-impl-inv.simps res
  proof (intro conjI f-repr)
    show d-repr (iarray-update ds i ?d'i) fs''
      unfolding d-repr[unfolded d-repr-def] d-repr-def iarray-update-of-fun dmu-ii
     by (rule iarray-cong', subst updates(1), auto simp: nth-append intro: arg-cong)
     show mu-repr ?dmu-repr' fs'' unfolding mu-repr-def swap-mu-def Let-def
dmu-ii
    proof (rule iarray-cong', goal-cases)
      case ii: (1 ii)
      show ?case
      proof (cases ii < i − 1)
        case small: True
        hence id: (ii = i) = False (ii = i − 1) = False (i < ii) = False (ii < i −
1) = True by auto
        have mu: mu !! ii = IArray.of-fun (dμ fs ii) ii
          using ii unfolding mu-def by auto
        show ?thesis unfolding id if-True if-False mu
            by (rule iarray-cong', insert small ii i i0, subst updates(2), simp-all,
linarith)
      next
        case False
        hence iFalse: (ii < i − 1) = False by auto
        show ?thesis unfolding iFalse if-False if-distrib[of λ f. IArray.of-fun f ii,
symmetric]
          dmu-ij-state.simps[of f mu ds, folded state, symmetric]
        proof (rule iarray-cong', goal-cases)
          case j: (1 j)
          note upd = updates(2)[OF ii j] dmu-ii dmu-ij-state[OF j ii] if-distrib[of λ
x. x j]
          note simps = dmu-ij-state[OF - ii] dmu-ij-state[OF - im1] dmu-ij-state[OF
- i]
```

from *False* **consider** (*I*) *ii* = *i j* = *i* − *1* | (*Is*) *ii* = *i j* ≠ *i* − *1* | (*Im1*) *ii* = *i* − *1* | (*large*) *ii* > *i* **by** *linarith*
thus *?case*
proof (*cases*)
case (*I*)
show *?thesis* **unfolding** *upd* **using** *I* **by** *auto*
next
case (*Is*)
show *?thesis* **unfolding** *upd* **using** *Is j simps* **by** *auto*
next
case (*Im1*)
hence *id*: (*i* < *ii*) = *False* (*ii* = *i*) = *False* (*ii* = *i* − *1*) = *True* **using** *i0* **by** *auto*
show *?thesis* **unfolding** *upd* **unfolding** *id if-False if-True* **by** (*rule simps*, *insert j Im1*, *auto*)
next
case (*large*)
hence *i* − *1* < *ii i* < *ii* **by** *auto*
note *simps* = *simps*(*1*)[*OF this*(*1*)] *simps*(*1*)[*OF this*(*2*)]
from *large* **have** *id*: (*i* < *ii*) = *True* (*ii* = *i* − *1*) = *False* ⋀ *x*. (*ii* = *i* ∧ *x*) = *False* **by** *auto*
show *?thesis* **unfolding** *id if-True if-False upd simps* **by** *auto*
qed
qed
qed
qed
qed
show *?g2* **unfolding** *fs-id fs''*[*symmetric*] *basis-reduction-swap-def* **unfolding** *res* **..**
qed

lemma *basis-reduction-swap*: **assumes**
impl: *LLL-impl-inv state i fs*
and *inv*: *LLL-invariant False i fs*
and *res*: *LLL-Impl.basis-reduction-swap m i state* = (*upw'*,*i'*,*state'*)
and *cond*: *sq-norm* (*gso fs* (*i* − *1*)) > *α* ∗ *sq-norm* (*gso fs i*)
and *i*: *i* < *m* **and** *i0*: *i* ≠ *0*
and *fs'*: *fs'* = *fs-state state'*
shows
LLL-impl-inv state' i' fs'
LLL-invariant upw' i' fs'
LLL-measure i' fs' < *LLL-measure i fs*
basis-reduction-swap i fs = (*upw'*,*i'*,*fs'*)
using *basis-reduction-swap-impl*[*OF assms*] *basis-reduction-swap*[*OF inv - cond i i0*] **by** *blast+*

lemma *basis-reduction-step-impl*: **assumes**
impl: *LLL-impl-inv state i fs*
and *inv*: *LLL-invariant upw i fs*

```
  and res: LLL-Impl.basis-reduction-step α n m upw i state = (upw',i',state')
  and i: i < m
  and fs': fs' = fs-state state'
shows
  LLL-impl-inv state' i' fs'
  basis-reduction-step upw i fs = (upw',i',fs')
proof (atomize(full), goal-cases)
  case 1
  obtain f mu ds where state: state = (f,mu,ds) by (cases state, auto)
  note def = LLL-Impl.basis-reduction-step-def basis-reduction-step-def
  from LLL-invD[OF inv] have len: length fs = m by auto
  from fs-state[OF impl inv state len] have fs: fs-state state = fs by auto
  show ?case
  proof (cases i = 0)
    case True
    from LLL-state-inc-state[OF impl inv state i] i
      assms increase-i[OF inv i True] True
      res fs' fs
    show ?thesis by (auto simp: def)
  next
    case False
    hence id: (i = 0) = False by auto
    obtain state'' where state'': LLL-Impl.basis-reduction-add-rows n upw i state
= state'' by auto
    define fs'' where fs'': fs'' = fs-state state''
    obtain f mu ds where state: state'' = (f,mu,ds) by (cases state'', auto)
    from basis-reduction-add-rows[OF impl inv state'' i fs'']
    have inv: LLL-invariant False i fs''
      and meas: LLL-measure i fs = LLL-measure i fs''
      and impl: LLL-impl-inv state'' i fs''
      and impl': basis-reduction-add-rows upw i fs = fs''
      by auto
    note invw = LLL-inv-imp-w[OF inv]
    obtain num denom where quot: quotient-of α = (num,denom) by force
    note d-state = d-state[OF impl inv state]
    from i have le: i − 1 ≤ m  i ≤ m Suc i ≤ m by auto
    note d-state = d-state[OF le(1)] d-state[OF le(2)] d-state[OF le(3)]
    interpret fs'': fs-int' n m fs-init fs''
      by standard (use invw in auto)
    have i < length fs''
      using LLL-invD[OF inv] i by auto
    note d-sq-norm-comparison = fs''.d-sq-norm-comparison[OF quot this False]
    note res = res[unfolded def id if-False Let-def state'' quot split d-state this]
  note pos = LLL-d-pos[OF invw le(1)] LLL-d-pos[OF invw le(2)] quotient-of-denom-pos[OF
quot]
    from False have sim1: Suc (i − 1) = i by simp
    let ?r = rat-of-int
    let ?x = sq-norm (gso fs'' (i − 1))
    let ?y = α ∗ sq-norm (gso fs'' i)
```

```
    show ?thesis
    proof (cases ?x ≤ ?y)
      case True
      from increase-i[OF inv i - True] True res meas LLL-state-inc-state[OF impl
inv state i] fs' fs''
        d-def d-sq-norm-comparison fs''.d-def impl' False
      show ?thesis by (auto simp: def)
    next
      case F: False
      hence gt: ?x > ?y and id: (?x ≤ ?y) = False by auto
      from res[unfolded id if-False] d-def d-sq-norm-comparison fs''.d-def id
      have LLL-Impl.basis-reduction-swap m i state'' = (upw', i', state')
        by auto
      from basis-reduction-swap[OF impl inv this gt i False fs'] show ?thesis using
meas F False
        by (auto simp: def Let-def impl')
    qed
  qed
qed

lemma basis-reduction-step: assumes
  impl: LLL-impl-inv state i fs
  and inv: LLL-invariant upw i fs
  and res: LLL-Impl.basis-reduction-step α n m upw i state = (upw',i',state')
  and i: i < m
  and fs': fs' = fs-state state'
shows
  LLL-impl-inv state' i' fs'
  LLL-invariant upw' i' fs'
  LLL-measure i' fs' < LLL-measure i fs
  basis-reduction-step upw i fs = (upw',i',fs')
  using basis-reduction-step-impl[OF assms] basis-reduction-step[OF inv - i] by
blast+

lemma basis-reduction-main-impl: assumes
  impl: LLL-impl-inv state i fs
  and inv: LLL-invariant upw i fs
  and res: LLL-Impl.basis-reduction-main α n m upw i state = state'
  and fs': fs' = fs-state state'
shows LLL-impl-inv state' m fs'
  basis-reduction-main (upw,i,fs) = fs'
proof (atomize(full), insert assms(1−3), induct LLL-measure i fs arbitrary: i fs
upw state rule: less-induct)
  case (less i fs upw)
  have id: LLL-invariant upw i fs = True using less by auto
  note res = less(4)[unfolded LLL-Impl.basis-reduction-main.simps[of - - - upw]]
  note inv = less(3)
  note impl = less(2)
  note IH = less(1)
```

```
  show ?case
  proof (cases i < m)
    case i: True
    obtain i'' state'' upw'' where step: LLL-Impl.basis-reduction-step α n m upw
i state = (upw'',i'',state'')
      (is ?step = -) by (cases ?step, auto)
    with res i have res: LLL-Impl.basis-reduction-main α n m upw'' i'' state'' =
state' by auto
    note main = basis-reduction-step[OF impl inv step i refl]
    from IH[OF main(3,1,2) res] main(4) step res
    show ?thesis by (simp add: i inv basis-reduction-main.simps)
  next
    case False
    from LLL-invD[OF inv] have len: length fs = m by auto
    obtain f mu ds where state: state = (f,mu,ds) by (cases state, auto)
    from fs-state[OF impl inv state len] have fs: fs-state state = fs by auto
    from False fs res fs' have fs-id: fs = fs' by simp
    from False LLL-invD[OF inv] have i: i = m by auto
    with False res inv impl fs have LLL-invariant upw m fs' ∧ LLL-impl-inv state'
m fs'
      by (auto simp: fs')
    thus ?thesis unfolding basis-reduction-main.simps[of upw i fs] using False
      by (auto simp: LLL-invariant-def fs-id)
  qed
qed

lemma basis-reduction-main: assumes
  impl: LLL-impl-inv state i fs
  and inv: LLL-invariant upw i fs
  and res: LLL-Impl.basis-reduction-main α n m upw i state = state'
  and fs': fs' = fs-state state'
shows
  LLL-invariant True m fs'
  LLL-impl-inv state' m fs'
  basis-reduction-main (upw,i,fs) = fs'
 using basis-reduction-main-impl[OF assms] basis-reduction-main[OF inv] by blast+

lemma initial-state: LLL-impl-inv (initial-state m fs-init) 0 fs-init (is ?g1)
  fs-state (initial-state m fs-init) = fs-init (is ?g2)
proof −
  have f-repr: list-repr 0 ([], fs-init) (map ((!) fs-init) [0..<m])
    unfolding list-repr-def by (simp, intro nth-equalityI, auto simp: len)
  from fs-init have Rn: set (RAT fs-init) ⊆ Rn by auto
  have 1: 1 = d fs-init 0 unfolding d-def by simp
  define j where j = m
  have jm: j ≤ m unfolding j-def by auto
  have 0: 0 = m − j unfolding j-def by auto
  interpret fs-init: fs-int-indpt n fs-init
    by (standard) (use lin-dep in auto)
```

```
  have mu-repr: mu-repr (IArray.of-fun (λi. IArray.of-fun ((!!) (dμ-impl fs-init !!
i)) i) m) fs-init
    unfolding fs-init.dμ-impl mu-repr-def fs-init.dμ-def dμ-def fs-init.d-def d-def
    apply(rule iarray-cong′)
    unfolding len[symmetric] by (auto simp add: nth-append)
  have d-repr: d-repr (IArray.of-fun (λi. if i = 0 then 1 else dμ-impl fs-init !! (i
− 1) !! (i − 1)) (Suc m)) fs-init
    unfolding fs-init.dμ-impl d-repr-def
  proof (intro iarray-cong′, goal-cases)
    case (1 i)
    show ?case
    proof (cases i = 0)
      case False
      hence le: i − 1 < length fs-init i − 1 < i and id: (i = 0) = False Suc (i −
1) = i
        using 1 len by auto
      show ?thesis unfolding of-fun-nth[OF le(1)] of-fun-nth[OF le(2)] id if-False
        dμ-def fs-init.dμ-def fs-init.d-def d-def
        by (auto simp add: gs.μ.simps )
    next
      case True
      have d fs-init 0 = 1 unfolding d-def gs.Gramian-determinant-0 by simp
      thus ?thesis unfolding True by simp
    qed
  qed
  show ?g1 unfolding initial-state-def Let-def LLL-impl-inv.simps id
    by (intro conjI f-repr mu-repr d-repr)
  from fs-state[OF this LLL-inv-initial-state]
  show ?g2 unfolding initial-state-def Let-def by (simp add: of-list-repr-def)
qed

lemma basis-reduction: assumes res: basis-reduction α n fs-init = state
  and fs: fs = fs-state state
shows LLL-invariant True m fs
  LLL-impl-inv state m fs
  basis-reduction-main (True, 0, fs-init) = fs
  using basis-reduction-main[OF initial-state(1) LLL-inv-initial-state res[unfolded
basis-reduction-def len Let-def] fs]
  by auto

lemma reduce-basis-impl: LLL-Impl.reduce-basis α fs-init = reduce-basis
proof −
  obtain fs where res: LLL-Impl.reduce-basis α fs-init = fs by blast
  have reduce-basis = fs
  proof (cases fs-init)
    case (Cons f)
    from fs-init[unfolded Cons] have dim-vec f = n by auto
    from res[unfolded LLL-Impl.reduce-basis-def Cons list.simps this, folded Cons]
    have fs-state (LLL-Impl.basis-reduction α n fs-init) = fs by auto
```

```
    from basis-reduction(3)[OF refl refl, unfolded this]
    show reduce-basis = fs unfolding reduce-basis-def .
  next
    case Nil
    with len have m0: m = 0 by auto
    show ?thesis using res
    unfolding reduce-basis-def LLL-Impl.reduce-basis-def basis-reduction-main.simps
using Nil m0
      by simp
  qed
  with res show ?thesis by simp
qed

lemma reduce-basis: assumes LLL-Impl.reduce-basis α fs-init = fs
  shows lattice-of fs = L
  reduced fs m
  lin-indep fs
  length fs = m
  LLL-invariant True m fs
  using reduce-basis-impl assms reduce-basis reduce-basis-inv by metis+

lemma short-vector-impl: LLL-Impl.short-vector α fs-init = short-vector
   using reduce-basis-impl unfolding LLL-Impl.short-vector-def short-vector-def
by simp

lemma short-vector: assumes res: LLL-Impl.short-vector α fs-init = v
  and m0: m ≠ 0
shows
  v ∈ carrier-vec n
  v ∈ L − {0_v n}
  h ∈ L − {0_v n} ⟹ rat-of-int (sq-norm v) ≤ α ^(m − 1) ∗ rat-of-int (sq-norm
h)
  v ≠ 0_v j
  using short-vector[OF assms[unfolded short-vector-impl]] by metis+

end
end
```

9.4 Bound on Number of Arithmetic Operations for Integer Implementation

In this section we define a version of the LLL algorithm which explicitly returns the costs of running the algorithm. Its soundness is mainly proven by stating that projecting away yields the original result.

The cost model counts the number of arithmetic operations that occur in vector-addition, scalar-products, and scalar multiplication and we prove a polynomial bound on this number.

```
theory LLL-Complexity
```

```
  imports
    LLL-Impl
    Cost
    HOL−Library.Discrete-Functions
begin

definition round-num-denom-cost :: int ⇒ int ⇒ int cost where
  round-num-denom-cost n d = ((2 * n + d) div (2 * d), 4) — 4 arith. operations

lemma round-num-denom-cost:
  shows result (round-num-denom-cost n d) = round-num-denom n d
    cost (round-num-denom-cost n d) ≤ 4
  unfolding round-num-denom-cost-def round-num-denom-def by (auto simp: cost-simps)


context LLL-with-assms
begin

context
  assumes α-gt: α > 4/3 and m0: m ≠ 0
begin


fun basis-reduction-add-rows-loop-cost where
  basis-reduction-add-rows-loop-cost state i j [] = (state, 0)
| basis-reduction-add-rows-loop-cost state i sj (fj # fjs) = (
    let fi = fi-state state;
        dsj = d-state state sj;
        j = sj − 1;
        (c,cost1) = round-num-denom-cost (dmu-ij-state state i j) dsj;
        state' = (if c = 0 then state else upd-fi-mu-state state i (vec n (λ i. fi $ i
− c * fj $ i)) — 2n arith. operations
              (IArray.of-fun (λ jj. let mu = dmu-ij-state state i jj in — 3 sj arith.
operations
                  if jj < j then mu − c * dmu-ij-state state j jj else
                  if jj = j then mu − dsj * c else mu) i));
        local-cost = 2 * n + 3 * sj;
        (res,cost2) = basis-reduction-add-rows-loop-cost state' i j fjs
    in (res, cost1 + local-cost + cost2))


lemma basis-reduction-add-rows-loop-cost: assumes length fs = j
  shows result (basis-reduction-add-rows-loop-cost state i j fs) = LLL-Impl.basis-reduction-add-rows-loop
n state i j fs
    cost (basis-reduction-add-rows-loop-cost state i j fs) ≤ sum (λ j. (2 * n + 4 +
3 * (Suc j))) {0..<j}
  using assms
proof (atomize(full), induct fs arbitrary: state j)
  case (Cons fj fs state j)
```

```
  let ?dm-ij = dmu-ij-state state i (j − 1)
  let ?dj = d-state state j
  obtain c1 fc where flc: round-num-denom-cost ?dm-ij ?dj = (fc, c1) by force
  from result-costD[OF round-num-denom-cost flc]
  have fl: round-num-denom ?dm-ij ?dj = fc and c1: c1 ≤ 4 by auto
  obtain st where st: (if fc = 0 then state
            else upd-fi-mu-state state i (vec n (λ i. fi-state state $ i − fc ∗ fj $ i))
                 (IArray.of-fun
                   (λjj. if jj < j − 1 then dmu-ij-state state i jj − fc ∗ dmu-ij-state
state (j − 1) jj
                         else if jj = j − 1 then dmu-ij-state state i jj − d-state state j
∗ fc else dmu-ij-state state i jj)
                   i)) = st  by auto
  obtain res c2 where rec: basis-reduction-add-rows-loop-cost st i (j − 1) fs =
(res,c2) (is ?x = -) by (cases ?x, auto)
  from Cons(2) have length fs = j − 1 by auto
  from result-costD'[OF Cons(1)[OF this] rec]
  have res: LLL-Impl.basis-reduction-add-rows-loop n st i (j − 1) fs = res
    and c2: c2 ≤ (∑ j = 0..<j − 1. 2 ∗ n + 4 + 3 ∗ Suc j) by auto
  show ?case unfolding basis-reduction-add-rows-loop-cost.simps Let-def flc split
      LLL-Impl.basis-reduction-add-rows-loop.simps fl st rec res cost-simps
  proof (intro conjI refl, goal-cases)
    case 1
    have c1 + (2 ∗ n + 3 ∗ j) + c2 ≤ (∑ j = 0..<j − 1. 2 ∗ n + 4 + 3 ∗ Suc
j) + (2 ∗ n + 4 + 3 ∗ Suc (j − 1))
      using c1 c2 by auto
    also have ... = (∑ j = 0..<j. 2 ∗ n + 4 + 3 ∗ (Suc j))
      by (subst (2) sum.remove[of - j − 1], insert Cons(2), auto intro: sum.cong)
    finally show ?case .
  qed
qed (auto simp: cost-simps)

definition basis-reduction-add-rows-cost where
  basis-reduction-add-rows-cost upw i state =
     (if upw then basis-reduction-add-rows-loop-cost state i i (small-fs-state state)
       else (state,0))

lemma basis-reduction-add-rows-cost: assumes impl: LLL-impl-inv state i fs and
inv: LLL-invariant upw i fs
 shows result (basis-reduction-add-rows-cost upw i state) = LLL-Impl.basis-reduction-add-rows
n upw i state
   cost (basis-reduction-add-rows-cost upw i state) ≤ (2 ∗ n + 2 ∗ i + 7) ∗ i
proof (atomize (full), goal-cases)
  case 1
 note d = basis-reduction-add-rows-cost-def LLL-Impl.basis-reduction-add-rows-def
  show ?case
  proof (cases upw)
    case False
    thus ?thesis by (auto simp: d cost-simps)
```

```
  next
    case True
    hence upw: upw = True by simp
    obtain f mu ds where state: state = (f,mu,ds) by (cases state, auto)
    from to-list-repr[OF impl inv state]
    have len: length (small-fs-state state) = i
      unfolding small-fs-state.simps state list-repr-def by auto
    let ?call = basis-reduction-add-rows-cost upw i state
    have res: result ?call = LLL-Impl.basis-reduction-add-rows n upw i state
      and cost: cost ?call ≤ sum (λ j. (2 * n + 4 + 3 * (Suc j))) {0..<i}
      unfolding d upw if-True using basis-reduction-add-rows-loop-cost[OF len, of
state i] by auto
    note cost
    also have sum (λ j. (2 * n + 4 + 3 * (Suc j))) {0..<i} = (2 * n + 7) * i +
3 * (∑ j = 0..<i. j)
      by (auto simp: algebra-simps  sum.distrib sum-distrib-right sum-distrib-left)
    also have (∑ j = 0..<i. j) = (i * (i − 1) div 2)
    proof (induct i)
      case (Suc i)
      thus ?case by (cases i, auto)
    qed auto
    finally have cost ?call ≤ (2 * n + 7) * i + 3 * (i * (i − 1) div 2) .
    also have ... ≤ (2 * n + 7) * i + 2 * i * i
    proof (rule add-left-mono)
      have 3 * (i * (i − 1) div 2) ≤ 2 * i * (i − 1) by simp
      also have ... ≤ 2 * i * i by (intro mult-mono, auto)
      finally show 3 * (i * (i − 1) div 2) ≤ 2 * i * i .
    qed
    also have ... = (2 * n + 2 * i + 7) * i by (simp add: algebra-simps)
    finally have cost: cost ?call ≤ (2 * n + 2 * i + 7) * i .
    show ?thesis using res cost by simp
  qed
qed

definition swap-mu-cost :: int iarray iarray ⇒ nat ⇒ int ⇒ int ⇒ int ⇒ int ⇒
int iarray iarray cost where
  swap-mu-cost dmu i dmu-i-im1 dim1 di dsi = (let im1 = i − 1;
      res = IArray.of-fun (λ ii. if ii < im1 then dmu !! ii else
        if ii > i then let dmu-ii = dmu !! ii in
           IArray.of-fun (λ j. let dmu-ii-j = dmu-ii !! j in   — 8 arith. operations
for whole line
              if j = i then (dsi * dmu-ii !! im1 − dmu-i-im1 * dmu-ii-j) div di —
4 arith. operations for this entry
              else if j = im1 then (dmu-i-im1 * dmu-ii-j + dmu-ii !! i * dim1) div
di — 4 arith. operations for this entry
              else dmu-ii-j) ii else
        if ii = i then let mu-im1 = dmu !! im1 in
           IArray.of-fun (λ j. if j = im1 then dmu-i-im1 else mu-im1 !! j) ii
        else IArray.of-fun (λ j. dmu !! i !! j) ii) — ii = i - 1
```

```
        m; — in total, there are m - (i+1) many lines that require arithmetic
operations: i + 1, ..., m - 1
      cost = 8 * (m - Suc i)
    in (res,cost))

lemma swap-mu-cost:
  result (swap-mu-cost dmu i dmu-i-im1 dim1 di dsi) = swap-mu m dmu i dmu-i-im1
dim1 di dsi
   cost (swap-mu-cost dmu i dmu-i-im1 dim1 di dsi) ≤ 8 * (m - Suc i)
  by (auto simp: swap-mu-cost-def swap-mu-def Let-def cost-simps)

definition basis-reduction-swap-cost where
  basis-reduction-swap-cost i state = (let
      di = d-state state i;
      dsi = d-state state (Suc i);
      dim1 = d-state state (i - 1);
      fi = fi-state state;
      fim1 = fim1-state state;
      dmu-i-im1 = dmu-ij-state state i (i - 1);
      fi' = fim1;
      fim1' = fi;
      di' = (dsi * dim1 + dmu-i-im1 * dmu-i-im1) div di; — 4 arith. operations
      local-cost = 4
    in (case state of (f,dmus,djs) ⇒
       case swap-mu-cost dmus i dmu-i-im1 dim1 di dsi of
         (swap-res, swap-cost) ⇒
         let res = (False, i - 1,
                   (dec-i (update-im1 (update-i f fi') fim1'),
                    swap-res,
                    iarray-update djs i di'));
             cost = local-cost + swap-cost
         in (res, cost)))

lemma basis-reduction-swap-cost:
    result (basis-reduction-swap-cost i state) = LLL-Impl.basis-reduction-swap m i
state
   cost (basis-reduction-swap-cost i state) ≤ 8 * (m - Suc i) + 4
proof (atomize(full), goal-cases)
  case 1
  obtain f dmus djs where state: state = (f,dmus,djs) by (cases state, auto)
  let ?mu = dmu-ij-state (f, dmus, djs) i (i - 1)
  let ?di1 = d-state (f, dmus, djs) (i - 1)
  let ?di = d-state (f, dmus, djs) i
  let ?dsi = d-state (f, dmus, djs) (Suc i)
 show ?case unfolding basis-reduction-swap-cost-def LLL-Impl.basis-reduction-swap-def
Let-def state split
    using swap-mu-cost[of dmus i ?mu ?di1 ?di ?dsi]
    by (cases swap-mu-cost dmus i ?mu ?di1 ?di ?dsi, auto simp: cost-simps)
qed
```

definition *basis-reduction-step-cost* **where**
basis-reduction-step-cost upw i state = (*if i* = *0 then* ((*True*, *Suc i*, *inc-state state*), *0*)
else let
(*state'*,*cost-add*) = *basis-reduction-add-rows-cost upw i state*;
di = *d-state state' i*;
dsi = *d-state state'* (*Suc i*);
dim1 = *d-state state'* (*i* − *1*);
(*num*,*denom*) = *quotient-of* α;
cond = (*di* ∗ *di* ∗ *denom* ≤ *num* ∗ *dim1* ∗ *dsi*); — 5 arith. operations
local-cost = *5*
in if cond then
((*True*, *Suc i*, *inc-state state'*), *local-cost* + *cost-add*)
else case basis-reduction-swap-cost i state' of (*res*, *cost-swap*) ⇒ (*res*, *local-cost* + *cost-swap* + *cost-add*))

definition *body-cost* = *2* + (*8* + *2* ∗ *n* + *2* ∗ *m*) ∗ *m*

lemma *basis-reduction-step-cost*: **assumes**
impl: *LLL-impl-inv state i fs*
and *inv*: *LLL-invariant upw i fs*
and *i*: *i* < *m*
shows *result* (*basis-reduction-step-cost upw i state*) = *LLL-Impl.basis-reduction-step* α *n m upw i state* (**is** *?g1*)
cost (*basis-reduction-step-cost upw i state*) ≤ *body-cost* (**is** *?g2*)
proof −
obtain *state' c-add* **where** *add*: *basis-reduction-add-rows-cost upw i state* = (*state'*,*c-add*)
(**is** *?add* = -) **by** (*cases ?add*, *auto*)
obtain *state''* *c-swap* **where** *swapc*: *basis-reduction-swap-cost i state'* = (*state''*,*c-swap*)
(**is** *?swap* = -) **by** (*cases ?swap*, *auto*)
note *res* = *basis-reduction-step-cost-def*[*of upw i state*, *unfolded add split swap*]
from *result-costD*[*OF basis-reduction-add-rows-cost*[*OF impl inv*] *add*]
have *add*: *LLL-Impl.basis-reduction-add-rows n upw i state* = *state'*
and *c-add*: *c-add* ≤ (*2* ∗ *n* + *2* ∗ *i* + *7*) ∗ *i*
by *auto*
from *result-costD*[*OF basis-reduction-swap-cost swapc*]
have *swap*: *LLL-Impl.basis-reduction-swap m i state'* = *state''*
and *c-swap*: *c-swap* ≤ *8* ∗ (*m* − *Suc i*) + *4* **by** *auto*
have *c-add* + *c-swap* + *5* ≤ *8* ∗ *m* + *2* + (*2* ∗ *n* + *2* ∗ *i*) ∗ *i*
using *c-add c-swap i* **by** (*auto simp*: *field-simps*)
also have ... ≤ *8* ∗ *m* + *2* + (*2* ∗ *n* + *2* ∗ *m*) ∗ *m*
by (*intro add-left-mono mult-mono*, *insert i*, *auto*)
also have ... = *2* + (*8* + *2* ∗ *n* + *2* ∗ *m*) ∗ *m* **by** (*simp add*: *field-simps*)
finally have *body*: *c-add* + *c-swap* + *5* ≤ *body-cost* **unfolding** *body-cost-def* **.**
obtain *num denom* **where** *alpha*: *quotient-of* α = (*num*,*denom*) **by** *force*
note *res'* = *LLL-Impl.basis-reduction-step-def*[*of* α *n m upw i state*, *unfolded*

add swap Let-def alpha split]
note *d* = *res res'*
show *?g1* **unfolding** *d* **by** (*auto split*: *if-splits simp*: *cost-simps Let-def alpha swapc*)
show *?g2* **unfolding** *d nat-distrib* **using** *body* **by** (*auto split*: *if-splits simp*: *cost-simps alpha Let-def swapc*)
qed

partial-function (*tailrec*) *basis-reduction-main-cost* **where**
basis-reduction-main-cost upw i state c = (*if i* < *m*
then let ((*upw'*,*i'*,*state'*), *c-step*) = *basis-reduction-step-cost upw i state*
in basis-reduction-main-cost upw' i' state' (*c* + *c-step*)
else (*state*, *c*))

definition *num-loops* = *m* + *2* ∗ *m* ∗ *m* ∗ *nat* (*ceiling* (*log base* (*real N*)))

lemma *basis-reduction-main-cost*: **assumes** *impl*: *LLL-impl-inv state i* (*fs-state state*)
and *inv*: *LLL-invariant upw i* (*fs-state state*)
and *state*: *state* = *initial-state m fs-init*
and *i*: *i* = *0*
shows *result* (*basis-reduction-main-cost upw i state c*) = *LLL-Impl.basis-reduction-main α n m upw i state* (**is** *?g1*)
cost (*basis-reduction-main-cost upw i state c*) ≤ *c* + *body-cost* ∗ *num-loops* (**is** *?g2*)
proof −
have *?g1* **and** *cost*: *cost* (*basis-reduction-main-cost upw i state c*) ≤ *c* + *body-cost* ∗ *LLL-measure i* (*fs-state state*)
using *assms*(*1*−*2*)
proof (*atomize* (*full*), *induct LLL-measure i* (*fs-state state*) *arbitrary*: *upw i state c rule*: *less-induct*)
case (*less i state upw c*)
note *inv* = *less*(*3*)
note *impl* = *less*(*2*)
obtain *i' upw' state' c-step* **where** *step*: *basis-reduction-step-cost upw i state* = ((*upw'*,*i'*,*state'*),*c-step*)
(**is** *?step* = -) **by** (*cases ?step*, *auto*)
obtain *state'' c-rec* **where** *rec*: *basis-reduction-main-cost upw' i' state'* (*c* + *c-step*) = (*state''*, *c-rec*)
(**is** *?rec* = -) **by** (*cases ?rec*, *auto*)
note *step'* = *result-costD*[*OF basis-reduction-step-cost*[*OF impl inv*] *step*]
note *d* = *basis-reduction-main-cost.simps*[*of upw*] *step split rec*
LLL-Impl.basis-reduction-main.simps[*of* - - - *upw*]
show *?case*
proof (*cases i* < *m*)
case *i*: *True*
from *step' i* **have** *step'*: *LLL-Impl.basis-reduction-step α n m upw i state* = (*upw'*,*i'*,*state'*)
and *c-step*: *c-step* ≤ *body-cost*

```
      by auto
    note d = d step'
    from basis-reduction-step[OF impl inv step' i refl]
    have impl': LLL-impl-inv state' i' (fs-state state')
     and inv': LLL-invariant upw' i' (fs-state state')
     and meas: LLL-measure i' (fs-state state') < LLL-measure i (fs-state state)
      by auto
    from result-costD'[OF less(1)[OF meas impl' inv'] rec]
    have rec': LLL-Impl.basis-reduction-main α n m upw' i' state' = state''
     and c-rec: c-rec ≤ c + c-step + body-cost * LLL-measure i' (fs-state state')
by auto
    from c-step c-rec have c-rec ≤ c + body-cost * Suc (LLL-measure i' (fs-state
state'))
      by auto
    also have ... ≤ c + body-cost * LLL-measure i (fs-state state)
      using meas by (intro plus-right-mono mult-left-mono) auto
    finally show ?thesis using i inv impl by (auto simp: cost-simps d rec')
   next
    case False
    thus ?thesis unfolding d by (auto simp: cost-simps)
   qed
  qed
  show ?g1 by fact
  note cost also have body-cost * LLL-measure i (fs-state state) ≤ body-cost *
num-loops
  proof (rule mult-left-mono; linarith?)
   define l where l = log base (real N)
   define k where k = 2 * m * m
    obtain f mu ds where init: initial-state m fs-init = (f,mu,ds) by (cases
initial-state m fs-init, auto)
   from initial-state
   have fs: fs-state (initial-state m fs-init) = fs-init by auto
    have LLL-measure i (fs-state state) ≤ nat (ceiling (m + k * l)) unfolding
l-def k-def
    using LLL-measure-approx-fs-init[OF LLL-inv-initial-state α-gt m0] unfold-
ing state fs i
     by linarith
  also have ... ≤ num-loops unfolding num-loops-def l-def[symmetric] k-def[symmetric]
    by (simp add: of-nat-ceiling times-right-mono)
   finally show LLL-measure i (fs-state state) ≤ num-loops .
  qed
  finally show ?g2
   by auto
qed


context fixes
  fs :: int vec iarray
begin
```

```
fun sigma-array-cost where
  sigma-array-cost dmus dmusi dmusj dll l = (if l = 0 then (dmusi !! l * dmusj !!
l, 1)
      else let l1 = l - 1; dll1 = dmus !! l1 !! l1;
        (sig, cost-rec) = sigma-array-cost dmus dmusi dmusj dll1 l1;
        res = (dll * sig + dmusi !! l * dmusj !! l) div dll1; — 4 arith. operations
        local-cost = (4 :: nat)
        in
      (res, local-cost + cost-rec))

declare sigma-array-cost.simps[simp del]

lemma sigma-array-cost:
   result (sigma-array-cost dmus dmusi dmusj dll l) = sigma-array dmus dmusi
dmusj dll l
  cost (sigma-array-cost dmus dmusi dmusj dll l) ≤ 4 * l + 1
proof (atomize(full), induct l arbitrary: dll)
  case 0
  show ?case unfolding sigma-array-cost.simps[of - - - - 0] sigma-array.simps[of
- - - - 0]
    by (simp add: cost-simps)
next
  case (Suc l)
  let ?sl = Suc l
  let ?dll = dmus !! (Suc l - 1) !! (Suc l - 1)
  show ?case unfolding sigma-array-cost.simps[of - - - - ?sl] sigma-array.simps[of
- - - - ?sl] Let-def
    using Suc[of ?dll]
    by (auto split: prod.splits simp: cost-simps)
qed

function dmu-array-row-main-cost where
  dmu-array-row-main-cost fi i dmus j = (if j ≥ i then (dmus, 0)
     else let sj = Suc j;
       dmus-i = dmus !! i;
       djj = dmus !! j !! j;
       (sigma, cost-sigma) = sigma-array-cost dmus dmus-i (dmus !! sj) djj j;
       dmu-ij = djj * (fi · fs !! sj) - sigma; — 2n + 2 arith. operations
       dmus' = iarray-update dmus i (iarray-append dmus-i dmu-ij);
       (res, cost-rec) = dmu-array-row-main-cost fi i dmus' sj;
       local-cost = 2 * n + 2
     in (res, cost-rec + cost-sigma + local-cost))
  by pat-completeness auto

termination by (relation measure (λ (fi,i,dmus,j). i - j), auto)

declare dmu-array-row-main-cost.simps[simp del]

lemma dmu-array-row-main-cost: assumes j ≤ i
```

shows *result* (*dmu-array-row-main-cost fi i dmus j*) = *dmu-array-row-main fs fi i dmus j*
cost (*dmu-array-row-main-cost fi i dmus j*) ≤ (∑ *jj* ∈ {*j* ..< *i*}. *2* * *n* + *2* + *4* * *jj* + *1*)
using *assms*
proof (*atomize*(*full*), *induct i* − *j arbitrary*: *j dmus*)
case (*0 j dmus*)
hence *j*: *j* = *i* **by** *auto*
thus *?case* **unfolding** *dmu-array-row-main-cost.simps*[*of* - - - *j*]
dmu-array-row-main.simps[*of* - - - - *j*]
by (*simp add*: *cost-simps*)
next
case (*Suc l j dmus*)
from *Suc*(*2*) **have** *id*: (*i* ≤ *j*) = *False* (*j* = *i*) = *False* **by** *auto*
let *?sl* = *Suc l*
let *?dll* = *dmus* !! (*Suc l* − *1*) !! (*Suc l* − *1*)
obtain *sig c-sig* **where**
sig-c: *sigma-array-cost dmus* (*dmus* !! *i*) (*dmus* !! *Suc j*) (*dmus* !! *j* !! *j*) *j* = (*sig*,*c-sig*) **by** *force*
from *result-costD*[*OF sigma-array-cost sig-c*]
have *sig*: *sigma-array dmus* (*dmus* !! *i*) (*dmus* !! *Suc j*) (*dmus* !! *j* !! *j*) *j* = *sig*
and *c-sig*: *c-sig* ≤ *4* * *j* + *1* **by** *auto*
obtain *dmus′* **where**
dmus′: *iarray-update dmus i* (*iarray-append* (*dmus* !! *i*) (*dmus* !! *j* !! *j* * (*fi* · *fs* !! *Suc j*) − *sig*)) = *dmus′*
by *auto*
obtain *res c-rec* **where** *rec-c*: *dmu-array-row-main-cost fi i dmus′* (*Suc j*) = (*res*, *c-rec*) **by** *force*
let *?c* = λ *j*. *2* * *n* + *2* + *4* * *j* + *1*
from *Suc*(*2*−*3*) **have** *l* = *i* − *Suc j Suc j* ≤ *i* **by** *auto*
from *Suc*(*1*)[*OF this*, *of dmus′*, *unfolded rec-c cost-simps*]
have *rec*: *dmu-array-row-main fs fi i dmus′* (*Suc j*) = *res*
and *c-rec*: *c-rec* ≤ (∑ *jj* = *Suc j*..<*i*. *?c jj*) **by** *auto*
have *c-rec* + *c-sig* + *2* * *n* + *2* ≤ *?c j* + (∑ *jj* = *Suc j*..<*i*. *?c jj*)
using *c-rec c-sig* **by** *auto*
also have ... = (∑ *jj* = *j*..<*i*. *?c jj*)
by (*subst* (*2*) *sum.remove*[*of* - *j*], *insert Suc*(*2*−), *auto intro*: *sum.cong*)
finally have *cost*: *c-rec* + *c-sig* + *2* * *n* + *2* ≤ (∑ *jj* = *j*..<*i*. *?c jj*) **by** *auto*
thus *?case* **unfolding** *dmu-array-row-main-cost.simps*[*of* - - - *j*] *dmu-array-row-main.simps*[*of* - - - - *j*] *Let-def*
id if-False sig-c split sig dmus′ rec rec-c cost-simps **by** *auto*
qed

definition *dmu-array-row-cost* **where**
dmu-array-row-cost dmus i = (*let fi* = *fs* !! *i*;
sp = *fi* · *fs* !! *0* — 2n arith. operations;
local-cost = *2* * *n*;
(*res*, *main-cost*) = *dmu-array-row-main-cost fi i* (*iarray-append dmus* (*IArray* [*sp*])) *0 in*

```
    (res, local-cost + main-cost))

lemma dmu-array-row-cost:
   result (dmu-array-row-cost dmus i) = dmu-array-row fs dmus i
   cost (dmu-array-row-cost dmus i) ≤ 2 * n + (2 * n + 1 + 2 * i) * i
proof (atomize(full), goal-cases)
  case 1
  let ?fi = fs !! i
  let ?arr = iarray-append dmus (IArray [?fi · fs !! 0])
  obtain res c-main where res-c: dmu-array-row-main-cost ?fi i ?arr 0 = (res,
c-main) by force
  from result-costD[OF dmu-array-row-main-cost res-c]
  have res: dmu-array-row-main fs ?fi i ?arr 0 = res
    and c-main: c-main ≤ (∑ jj = 0..<i. 2 * n + 2 + 4 * jj + 1) by auto
  have 2 * n + c-main ≤ 2 * n + (∑ jj = 0..<i. 2 * n + 2 + 4 * jj + 1) using
c-main by auto
  also have ... = 2 * n + (2 * n + 3) * i + 2 * (∑ jj < i. 2 * jj)
   unfolding sum.distrib by (auto simp: sum-distrib-left field-simps intro: sum.cong)
  also have (∑ jj < i. 2 * jj) = i * (i − 1)
    by (induct i, force, rename-tac i, case-tac i, auto)
  finally have 2 * n + c-main ≤ 2 * n + (2 * n + 3 + 2 * (i − 1)) * i by
(simp add: field-simps)
   also have ... = 2 * n + (2 * n + 1 + 2 * i) * i by (cases i, auto simp:
field-simps)
  finally have 2 * n + c-main ≤ 2 * n + (2 * n + 1 + 2 * i) * i .
   thus ?case unfolding dmu-array-row-cost-def Let-def dmu-array-row-def res-c
res split cost-simps
    by auto
qed

function dmu-array-cost where
  dmu-array-cost dmus i = (if i ≥ m then (dmus,0) else
    let (dmus′, cost-row) = dmu-array-row-cost dmus i;
        (res, cost-rec) = dmu-array-cost dmus′ (Suc i)
     in (res, cost-row + cost-rec))
  by pat-completeness auto

termination by (relation measure (λ (dmus, i). m − i), auto)

declare dmu-array-cost.simps[simp del]

lemma dmu-array-cost: assumes i ≤ m
  shows result (dmu-array-cost dmus i) = dmu-array fs m dmus i
   cost (dmu-array-cost dmus i) ≤ (∑ ii ∈ {i ..< m}. 2 * n + (2 * n + 1 + 2 *
ii) * ii)
  using assms
proof (atomize(full), induct m − i arbitrary: i dmus)
  case (0 i dmus)
  hence i: i = m by auto
```

thus *?case* **unfolding** *dmu-array-cost.simps*[*of - i*]
dmu-array.simps[*of - - - i*]
by (*simp add*: *cost-simps*)
next
case (*Suc k i dmus*)
obtain *dmus′ c-row* **where** *row-c*: *dmu-array-row-cost dmus i* = (*dmus′*,*c-row*)
by *force*
from *result-costD*[*OF dmu-array-row-cost row-c*]
have *row*: *dmu-array-row fs dmus i* = *dmus′*
and *c-row*: *c-row* ≤ *2* ∗ *n* + (*2* ∗ *n* + *1* + *2* ∗ *i*) ∗ *i* (**is** - ≤ *?c i*) **by** *auto*
from *Suc* **have** *k* = *m* − *Suc i Suc i* ≤ *m*
and *id*: (*m* ≤ *i*) = *False* (*i* = *m*) = *False* **by** *auto*
note *IH* = *Suc*(*1*)[*OF this*(*1*−*2*)]
obtain *res c-rec* **where** *rec-c*: *dmu-array-cost dmus′* (*Suc i*) = (*res*, *c-rec*) **by**
force
from *result-costD′*[*OF IH rec-c*]
have *rec*: *dmu-array fs m dmus′* (*Suc i*) = *res*
and *c-rec*: *c-rec* ≤ (∑ *ii* = *Suc i..*<*m*. *?c ii*) **by** *auto*
have *c-row* + *c-rec* ≤ *?c i* + (∑ *ii* = *Suc i..*<*m*. *?c ii*)
using *c-rec c-row* **by** *auto*
also have . . . = (∑ *ii* = *i..*<*m*. *?c ii*)
by (*subst sum.atLeast-Suc-lessThan* [*of i*]) (*use Suc* **in** *auto*)
finally show *?case* **unfolding** *dmu-array-cost.simps*[*of - i*]
dmu-array.simps[*of - - - i*] *id if-False Let-def rec-c row-c row rec split cost-simps*
by *auto*
qed
end

definition *dμ-impl-cost* :: *int vec list* ⇒ *int iarray iarray cost* **where**
dμ-impl-cost fs = *dmu-array-cost* (*IArray fs*) (*IArray* []) *0*

lemma *dμ-impl-cost*: *result* (*dμ-impl-cost fs-init*) = *dμ-impl fs-init*
cost (*dμ-impl-cost fs-init*) ≤ *m* ∗ (*m* ∗ (*m* + *n* + *2*) + *2* ∗ *n* + *1*)
proof (*atomize*(*full*), *goal-cases*)
case *1*
let *?fs* = *IArray fs-init*
let *?dmus* = *IArray* []
obtain *res cost* **where** *res-c*: *dmu-array-cost ?fs ?dmus 0* = (*res*, *cost*) **by** *force*
from *result-costD*[*OF dmu-array-cost res-c*]
have *res*: *dmu-array ?fs m ?dmus 0* = *res*
and *cost*: *cost* ≤ (∑ *ii* = *0..*<*m*. *2* ∗ *n* + (*2* ∗ *n* + *1* + *2* ∗ *ii*) ∗ *ii*) **by** *auto*
note *cost*
also have (∑ *ii* = *0..*<*m*. *2* ∗ *n* + (*2* ∗ *n* + *1* + *2* ∗ *ii*) ∗ *ii*)
= *2* ∗ *n* ∗ *m* + (*2* ∗ *n* + *1*) ∗ (∑ *ii* = *0..*<*m*. *ii*) + *2* ∗ (∑ *ii* = *0..*<*m*. *ii* ∗
ii)
by (*auto simp*: *field-simps sum.distrib sum-distrib-left intro*: *sum.cong*)
also have . . . ≤ *2* ∗ *n* ∗ *m* + (*2* ∗ *n* + *2*) ∗ (∑ *ii* = *0..*<*m*. *ii*) + *2* ∗ (∑ *ii*
= *0..*<*m*. *ii* ∗ *ii*)
by *auto*

also have $(2 * n + 2) * (\sum ii = 0..<m.\ \ ii) = (n + 1) * (2 * (\sum ii = 0..<m.\ ii))$ **by** *auto*
also have $2 * (\sum ii = 0..<m.\ \ ii) = m * (m - 1)$
by (*induct m*, *force*, *rename-tac i*, *case-tac i*, *auto*)
also have $2 * (\sum ii = 0..<m.\ \ ii * ii) = (6 * (\sum ii = 0..<m.\ \ ii * ii))\ div\ 3$ **by** *simp*
also have $6 * (\sum ii = 0..<m.\ \ ii * ii) = 2 * (m - 1)*(m-1)*(m-1) + 3 * (m - 1) * (m - 1) + (m - 1)$
by (*induct m*, *simp*, *rename-tac i*, *case-tac i*, *auto simp*: *field-simps*)
finally have $cost \leq 2 * n * m + (n + 1) * (m * (m - 1))$
$+ (2 * (m - 1) * (m - 1) * (m - 1) + 3 * (m - 1) * (m - 1) + (m - 1))\ div\ 3$ **.**
also have $\ldots \leq 2 * n * m + (n + 1) * (m * m) + (3 * m * m * m + 3 * m * m + 3 * m)\ div\ 3$
by (*intro add-mono div-le-mono mult-mono*, *auto*)
also have $\ldots = 2 * n * m + (n + 1) * (m * m) + (m * m * m + m * m + m)$
by *simp*
also have $\ldots = m * (m * (m + n + 2) + 2 * n + 1)$
by (*simp add*: *algebra-simps*)
finally
show *?case* **unfolding** *dμ-impl-cost-def dμ-impl-def len res res-c cost-simps* **by** *simp*
qed

definition *initial-gso-cost* $= m * (m * (m + n + 2) + 2 * n + 1)$

definition *initial-state-cost fs* = (*let*
(*dmus*, *cost*) = *dμ-impl-cost fs*;
ds = *IArray.of-fun* (λ *i*. *if i* = *0 then 1 else let i1* = *i* − *1 in dmus* !! *i1* !! *i1*) (*Suc m*);
dmus′ = *IArray.of-fun* (λ *i*. *let row-i* = *dmus* !! *i in*
IArray.of-fun (λ *j*. *row-i* !! *j*) *i*) *m*
in ((([], *fs*), *dmus′*, *ds*), *cost*) :: *LLL-dmu-d-state cost*)

definition *basis-reduction-cost* :: - ⇒ *LLL-dmu-d-state cost* **where**
basis-reduction-cost fs = (
case initial-state-cost fs of (*state1*, *c1*) ⇒
case basis-reduction-main-cost True 0 state1 0 of (*state2*, *c2*) ⇒
(*state2*, *c1* + *c2*))

definition *reduce-basis-cost* :: - ⇒ *int vec list cost* **where**
reduce-basis-cost fs = (*case fs of Nil* ⇒ (*fs*, *0*) | *Cons f* - ⇒
case basis-reduction-cost fs of (*state*,*c*) ⇒
(*fs-state state*, *c*))

lemma *initial-state-cost*: *result* (*initial-state-cost fs-init*) = *initial-state m fs-init* (**is** *?g1*)

```
  cost (initial-state-cost fs-init) ≤ initial-gso-cost (is ?g2)
proof −
  obtain st c where dmu: dμ-impl-cost fs-init = (st,c) by force
  from dμ-impl-cost[unfolded dmu cost-simps]
  have dmu': dμ-impl fs-init = st and c: c ≤ initial-gso-cost
    unfolding initial-gso-cost-def by auto
  show ?g1 ?g2 using c unfolding initial-state-cost-def dmu dmu' split cost-simps

    initial-state-def Let-def by auto
qed

lemma basis-reduction-cost:
   result (basis-reduction-cost fs-init) = basis-reduction α n fs-init  (is ?g1)
   cost (basis-reduction-cost fs-init) ≤ initial-gso-cost + body-cost * num-loops (is
?g2)
proof −
  obtain state1 c1 where init: initial-state-cost fs-init = (state1, c1) (is ?init =
-) by (cases ?init, auto)
   obtain state2 c2 where main: basis-reduction-main-cost True 0 state1 0 =
(state2, c2) (is ?main = -) by (cases ?main, auto)
  have res: basis-reduction-cost fs-init = (state2, c1 + c2)
    unfolding basis-reduction-cost-def init main split by simp
  from result-costD[OF initial-state-cost init]
  have c1: c1 ≤ initial-gso-cost and init: initial-state m fs-init = state1 by auto
  note inv = LLL-inv-initial-state(1)
  note impl = initial-state
  have fs: fs-state (initial-state m fs-init) = fs-init by fact
   from basis-reduction-main-cost[of initial-state m fs-init - - 0, unfolded fs, OF
impl(1) inv,
    unfolded init main cost-simps]
  have main: LLL-Impl.basis-reduction-main α n m True 0 state1 = state2 and
c2: c2 ≤ body-cost * num-loops
    by auto
  have res': basis-reduction α n fs-init = state2 unfolding basis-reduction-def len
init main Let-def ..
  show ?g1 unfolding res res' cost-simps ..
  show ?g2 unfolding res cost-simps using c1 c2 by auto
qed
```

The lemma for the LLL algorithm with explicit cost annotations *reduce-basis-cost* shows that the termination measure indeed gives rise to an explicit cost bound. Moreover, the computed result is the same as in the non-cost counting *local.reduce-basis*.

```
lemma reduce-basis-cost:
   result (reduce-basis-cost fs-init) = LLL-Impl.reduce-basis α fs-init  (is ?g1)
  cost (reduce-basis-cost fs-init) ≤ initial-gso-cost + body-cost * num-loops (is ?g2)
proof (atomize(full), goal-cases)
  case 1
  note d = reduce-basis-cost-def LLL-Impl.reduce-basis-def
```

```
  show ?case
  proof (cases fs-init)
    case Nil
    show ?thesis unfolding d unfolding Nil by (auto simp: cost-simps)
  next
    case (Cons f)
    obtain state c where b: basis-reduction-cost fs-init = (state,c) (is ?b = -) by
(cases ?b, auto)
    from result-costD[OF basis-reduction-cost b]
     have bb: basis-reduction α n fs-init = state and c: c ≤ initial-gso-cost +
body-cost * num-loops
      by auto
    from fs-init[unfolded Cons] have dim: dim-vec f = n by auto
   show ?thesis unfolding d b split unfolding Cons list.simps unfolding Cons[symmetric]
dim bb
      using c by (auto simp: cost-simps)
  qed
qed

lemma mn: m ≤ n
 unfolding len[symmetric] using lin-dep length-map unfolding gs.lin-indpt-list-def
  by (metis distinct-card gs.dim-is-n gs.fin-dim gs.li-le-dim(2))
```

Theorem with expanded costs: $O(n \cdot m^3 \cdot \log(maxnorm\ F))$ arithmetic operations

```
lemma reduce-basis-cost-expanded:
  assumes Lg ≥ nat ⌈log (of-rat (4 * α / (4 + α))) N⌉
  shows cost (reduce-basis-cost fs-init)
  ≤ 4 * Lg * m * m * m * n
    + 4 * Lg * m * m * m * m
    + 16 * Lg * m * m * m
    + 4 * Lg * m * m
    + 3 * m * m * m
    + 3 * m * m * n
    + 10 * m * m
    + 2 * n * m
    + 3 * m
  (is ?cost ≤ ?exp Lg)
proof −
  define Log where Log = nat ⌈log (of-rat (4 * α / (4 + α))) N⌉
  have Lg: Log ≤ Lg using assms unfolding Log-def .
  have ?cost ≤ ?exp Log
    unfolding Log-def
   using reduce-basis-cost(2)[unfolded num-loops-def body-cost-def initial-gso-cost-def
base-def]
    by (auto simp: algebra-simps)
  also have ... ≤ ?exp Lg
    by (intro add-mono mult-mono Lg, auto)
  finally show ?thesis .
```

qed

lemma *reduce-basis-cost-0*: **assumes** $m = 0$
shows *cost* (*reduce-basis-cost fs-init*) = 0
proof −
from *len assms* **have** *fs-init*: *fs-init* = [] **by** *auto*
thus *?thesis* **unfolding** *reduce-basis-cost-def* **by** (*simp add*: *cost-simps*)
qed

lemma *reduce-basis-cost-N*:
assumes $Lg \geq nat \lceil log\ (of\text{-}rat\ (4 * \alpha\ /\ (4 + \alpha)))\ N \rceil$
and *0*: $Lg > 0$
shows *cost* (*reduce-basis-cost fs-init*) $\leq 49 * m\ \hat{}\ 3 * n * Lg$
proof (*cases* $m > 0$)
case *True*
with *mn 0* **have** *0*: $0 < Lg\ 0 < m\ 0 < n$ **by** *auto*
note *reduce-basis-cost-expanded*[*OF assms*(*1*)]
also have $4 * Lg * m * m * m * n = 4 * m\ \hat{}\ 3 * n * Lg$
using *0* **by** (*auto simp add*: *power3-eq-cube*)
also have $4 * Lg * m * m * m * m \leq 4 * m\ \hat{}\ 3 * n * Lg$
using *0 mn* **by** (*auto simp add*: *power3-eq-cube*)
also have $16 * Lg * m * m * m \leq 16 * m\ \hat{}\ 3 * n * Lg$
using *0* **by** (*auto simp add*: *power3-eq-cube*)
also have $4 * Lg * m * m \leq 4 * \ m\ \hat{}\ 3 * n * Lg$
using *0* **by** (*auto simp add*: *power3-eq-cube*)
also have $3 * m * m * m \leq 3 * \ m\ \hat{}\ 3 * n * Lg$
using *0* **by** (*auto simp add*: *power3-eq-cube*)
also have $3 * m * m * n \leq 3 * m\ \hat{}\ 3 * n * Lg$
using *0* **by** (*auto simp add*: *power3-eq-cube*)
also have $10 * m * m \leq 10 * m\ \hat{}\ 3 * n * Lg$
using *0* **by** (*auto simp add*: *power3-eq-cube*)
also have $2 * n * m\ \leq 2 * m\ \hat{}\ 3 * n * Lg$
using *0* **by** (*auto simp add*: *power3-eq-cube*)
also have $3 * m \leq 3 * m\ \hat{}\ 3 * n * Lg$
using *0* **by** (*auto simp add*: *power3-eq-cube*)
finally show *?thesis*
by (*auto simp add*: *algebra-simps*)
next
case *False*
with *reduce-basis-cost-0* **show** *?thesis* **by** *simp*
qed

lemma *reduce-basis-cost-M*:
assumes $Lg \geq nat \lceil log\ (of\text{-}rat\ (4 * \alpha\ /\ (4 + \alpha)))\ (M * n) \rceil$
and *0*: $Lg > 0$
shows *cost* (*reduce-basis-cost fs-init*) $\leq 98 * m\ \hat{}\ 3 * n * Lg$
proof (*cases* $m > 0$)
case *True*
let *?prod* = *nat M* ∗ *nat M* ∗ *n*

```
  let ?p = nat M * nat M * n * n
  let ?lg = real-of-int (M * n)
  from 0 True have m0: m ≠ 0 by simp
 from LLL-inv-N-pos[OF LLL-inv-imp-w[OF LLL-inv-initial-state] g-bound-fs-init
m0] have N0: N > 0 .
  from N-le-MMn[OF m0] have N-prod: N ≤ ?prod by auto
  from N0 N-prod have M0: M > 0 by (cases M ≤ 0, auto)
  from N0 N-prod have prod0: 0 < ?prod by linarith
  from prod0 have n0: n > 0 by auto
  from n0 prod0 M0 have prod-p: ?prod ≤ ?p by auto
  with N-prod prod0 have N-p: N ≤ ?p and p0: 0 < ?p by linarith+
  let ?base = real-of-rat (4 * α / (4 + α))
  have base: 1 < ?base using α-gt by auto
  have Lg: nat ⌈log ?base N⌉ ≤ nat ⌈log ?base ?p⌉
   by (intro nat-mono ceiling-mono floor-log-mono, subst log-le-cancel-iff[OF base],
      insert M0 N-p N0 p0 n0, auto simp flip: of-int-mult of-nat-mult)
  also have log ?base ?p = log ?base (?lg^2)
    using M0 by (simp add: power2-eq-square ac-simps)
  also have ... = 2 * log ?base ?lg
    by (subst log-nat-power, insert M0 n0, auto)
  finally have nat ⌈log ?base N⌉ ≤ nat ⌈2 * log ?base ?lg⌉ .
  also have ... ≤ 2 * Lg using assms
    by linarith
  finally have Log: nat ⌈log ?base N⌉ ≤ 2 * Lg .
  from 0 have 0 < 2 * Lg by simp
  from reduce-basis-cost-N[OF Log this]
  show ?thesis by simp
next
  case False
  with reduce-basis-cost-0 show ?thesis by simp
qed


end
end
end
```

9.5 Explicit Bounds for Size of Numbers that Occur During LLL Algorithm

The LLL invariant does not contain bounds on the number that occur during the execution. We here strengthen the invariant so that it enforces bounds on the norms of the f_i and g_i and we prove that the stronger invariant is maintained throughout the execution of the LLL algorithm.

Based on the stronger invariant we prove bounds on the absolute values of the $\mu_{i,j}$, and on the absolute values of the numbers in the vectors f_i and g_i. Moreover, we further show that also the denominators in all of these numbers doesn't grow to much. Finally, we prove that each number (i.e.,

numerator or denominator) during the execution can be represented with at most $\mathcal{O}(m \cdot \log(M \cdot n))$ bits, where m is the number of input vectors, n is the dimension of the input vectors, and M is the maximum absolute value of all numbers in the input vectors. Hence, each arithmetic operation in the LLL algorithm can be performed in polynomial time.

theory *LLL-Number-Bounds*
  imports *LLL*
    Gram-Schmidt-Int
begin

context *LLL*
begin

The bounds for the f_i distinguishes whether we are inside or outside the inner for-loop.

definition *f-bound* :: *bool* ⇒ *nat* ⇒ *int vec list* ⇒ *bool* **where**
  f-bound outside ii fs = (∀ *i* < *m*. *sq-norm* (*fs* ! *i*) ≤ (*if i* ≠ *ii* ∨ *outside then int* (*N* ∗ *m*) *else*
    int (*4* ^ (*m* − *1*) ∗ *N* ^ *m* ∗ *m* ∗ *m*)))

definition *g-bnd* :: *rat* ⇒ *int vec list* ⇒ *bool* **where**
  g-bnd B fs = (∀ *i* < *m*. *sq-norm* (*gso fs i*) ≤ *B*)

definition *μ-bound-row fs bnd i* = (∀ *j* ≤ *i*. (*μ fs i j*)^2 ≤ *bnd*)
abbreviation *μ-bound-row-inner fs i j* ≡ *μ-bound-row fs* (*4* ^ (*m* − *1* − *j*) ∗ *of-nat* (*N* ^ (*m* − *1*) ∗ *m*)) *i*

definition *LLL-bound-invariant outside upw i fs* =
  (*LLL-invariant upw i fs* ∧ *f-bound outside i fs* ∧ *g-bound fs*)

lemma *bound-invD*: **assumes** *LLL-bound-invariant outside upw i fs*
  shows *LLL-invariant upw i fs f-bound outside i fs g-bound fs*
  using *assms* **unfolding** *LLL-bound-invariant-def* **by** *auto*

lemma *bound-invI*: **assumes** *LLL-invariant upw i fs f-bound outside i fs g-bound fs*
  shows *LLL-bound-invariant outside upw i fs*
  using *assms* **unfolding** *LLL-bound-invariant-def* **by** *auto*

lemma *μ-bound-rowI*: **assumes** ⋀ *j*. *j* ≤ *i* ⟹ (*μ fs i j*)^2 ≤ *bnd*
  shows *μ-bound-row fs bnd i*
  using *assms* **unfolding** *μ-bound-row-def* **by** *auto*

lemma *μ-bound-rowD*: **assumes** *μ-bound-row fs bnd i j* ≤ *i*
  shows (*μ fs i j*)^2 ≤ *bnd*
  using *assms* **unfolding** *μ-bound-row-def* **by** *auto*

lemma *μ-bound-row-1*: **assumes** *μ-bound-row fs bnd i*
shows *bnd ≥ 1*
proof −
interpret *gs1*: *gram-schmidt-fs n RAT fs* **.**
show *?thesis*
using *μ-bound-rowD*[*OF assms*, *of i*]
by (*auto simp*: *gs1.μ.simps*)
qed

lemma *reduced-μ-bound-row*: **assumes** *red*: *reduced fs i*
and *ii*: *ii* < *i*
shows *μ-bound-row fs 1 ii*
proof (*intro μ-bound-rowI*)
fix *j*
assume *j* ≤ *ii*
interpret *gs1*: *gram-schmidt-fs n RAT fs* **.**
show (*μ fs ii j*)^2 ≤ *1*
proof (*cases j* < *ii*)
case *True*
from *red*[*unfolded gram-schmidt-fs.reduced-def*, *THEN conjunct2*, *rule-format*, *OF ii True*]
have *abs* (*μ fs ii j*) ≤ *1/2* **by** *auto*
from *mult-mono*[*OF this this*]
show *?thesis* **by** (*auto simp*: *power2-eq-square*)
qed (*auto simp*: *gs1.μ.simps*)
qed

lemma *f-bound-True-arbitrary*: **assumes** *f-bound True ii fs*
shows *f-bound outside j fs*
unfolding *f-bound-def*
proof (*intro allI impI*, *rule ccontr*, *goal-cases*)
case (*1 i*)
from *1* **have** *nz*: ‖*fs* ! *i*‖^2 ≠ *0* **by** (*auto split*: *if-splits*)
hence *gt*: ‖*fs* ! *i*‖^2 > *0* **using** *sq-norm-vec-ge-0*[*of fs* ! *i*] **by** *auto*
from *assms*(*1*)[*unfolded f-bound-def*, *rule-format*, *OF 1*(*1*)]
have *one*: ‖*fs* ! *i*‖^2 ≤ *int* (*N* ∗ *m*) ∗ *1* **by** *auto*
from *less-le-trans*[*OF gt one*] **have** *N0*: *N* ≠ *0* **by** (*cases N* = *0*, *auto*)
note *one*
also have *int* (*N* ∗ *m*) ∗ *1* ≤ *int* (*N* ∗ *m*) ∗ *int* (*4* ^ (*m* − *1*) ∗ *N* ^ (*m* − *1*) ∗ *m*)
by (*rule mult-left-mono*, *unfold of-nat-mult*, *intro mult-ge-one*, *insert 1 N0*, *auto*)
also have ... = *int* (*4* ^ (*m* − *1*) ∗ *N* ^ (*Suc* (*m* − *1*)) ∗ *m* ∗ *m*) **unfolding** *of-nat-mult* **by** *simp*
also have *Suc* (*m* − *1*) = *m* **using** *1* **by** *simp*
finally show *?case* **using** *one 1* **by** (*auto split*: *if-splits*)
qed

context fixes *fs* :: *int vec list*
assumes *lin-indep*: *lin-indep fs*
and *len*: *length fs* = *m*
begin

interpretation *fs*: *fs-int-indpt n fs*
by (*standard*) (*use lin-indep* **in** *simp*)

lemma *sq-norm-fs-mu-g-bound*: **assumes** *i*: *i* < *m*
and *mu-bound*: *μ-bound-row fs bnd i*
and *g-bound*: *g-bound fs*
shows *of-int* ‖*fs* ! *i*‖² ≤ *of-nat* (*Suc i* ∗ *N*) ∗ *bnd*
proof −
have *of-int* ‖*fs* ! *i*‖² = (∑*j*←[*0*..<*Suc i*]. (*μ fs i j*)² ∗ ‖*gso fs j*‖²)
by (*rule fs.sq-norm-fs-via-sum-mu-gso*) (*use assms lin-indep len* **in** *auto*)
also have ... ≤ (∑*j*←[*0*..<*Suc i*]. *bnd* ∗ *of-nat N*)
proof (*rule sum-list-ge-mono*, *force*, *unfold length-map length-upt*,
subst (*1 2*) *nth-map-upt*, *force*, *goal-cases*)
case (*1 j*)
hence *ji*: *j* ≤ *i* **by** *auto*
from *g-bound*[*unfolded g-bound-def*] *i ji*
have *GB*: *sq-norm* (*gso fs j*) ≤ *of-nat N* **by** *auto*
show *?case*
by (*rule mult-mono*, *insert μ-bound-rowD*[*OF mu-bound ji*]
GB order.trans[*OF zero-le-power2*], *auto*)
qed
also have ... = *of-nat* (*Suc i*) ∗ (*bnd* ∗ *of-nat N*) **unfolding** *sum-list-triv length-upt* **by** *simp*
also have ... = *of-nat* (*Suc i* ∗ *N*) ∗ *bnd* **unfolding** *of-nat-mult* **by** *simp*
finally show *?thesis* **.**
qed
end

lemma *increase-i-bound*: **assumes** *LLL*: *LLL-bound-invariant True upw i fs*
and *i*: *i* < *m*
and *upw*: *upw* ⟹ *i* = *0*
and *red-i*: *i* ≠ *0* ⟹ *sq-norm* (*gso fs* (*i* − *1*)) ≤ *α* ∗ *sq-norm* (*gso fs i*)
shows *LLL-bound-invariant True True* (*Suc i*) *fs*
proof −
from *bound-invD*[*OF LLL*] **have** *LLL*: *LLL-invariant upw i fs*
and *f-bound True i fs* **and** *gbnd*: *g-bound fs* **by** *auto*
hence *fbnd*: *f-bound True* (*Suc i*) *fs* **by** (*auto simp*: *f-bound-def*)
from *increase-i*[*OF LLL i upw red-i*]
have *inv*: *LLL-invariant True* (*Suc i*) *fs* **and** *LLL-measure* (*Suc i*) *fs* < *LLL-measure i fs* (**is** *?g2*)
by *auto*
show *LLL-bound-invariant True True* (*Suc i*) *fs*

by (*rule bound-invI*[*OF inv fbnd gbnd*])
qed

Addition step preserves *LLL-bound-invariant False*

lemma *basis-reduction-add-row-main-bound*: **assumes** *Linv*: *LLL-bound-invariant False True i fs*
and *i*: $i < m$ **and** *j*: $j < i$
and *c*: $c = round\ (\mu\ fs\ i\ j)$
and *fs'*: $fs' = fs[\ i := fs\ !\ i - c \cdot_v fs\ !\ j]$
and *mu-small*: μ-*small-row i fs* (*Suc j*)
and *mu-bnd*: μ-*bound-row-inner fs i* (*Suc j*)
shows *LLL-bound-invariant False True i fs'*
μ-*bound-row-inner fs' i j*
proof (*rule bound-invI*)
from *bound-invD*[*OF Linv*]
have *Linv*: *LLL-invariant True i fs* **and** *fbnd*: *f-bound False i fs* **and** *gbnd*: *g-bound fs*
by *auto*
note *Linvw* = *LLL-inv-imp-w*[*OF Linv*]
note *main* = *basis-reduction-add-row-main*[*OF Linvw i j fs'*]
note *main* = *main*(*2*)[*OF Linv*] *main*(*3*,*5*−)
note *main* = *main*(*1*) *main*(*2*)[*OF c mu-small*] *main*(*3*−)
show *Linv'*: *LLL-invariant True i fs'* **by** *fact*
define *bnd* :: *rat* **where** *bnd*: $bnd = 4\ \hat{}\ (m - 1 - Suc\ j) * of\text{-}nat\ (N\ \hat{}\ (m - 1) * m)$
note *mu-bnd* = *mu-bnd*[*folded bnd*]
note *inv* = *LLL-invD*[*OF Linv*]
let *?mu* = $\mu\ fs$
let *?mu'* = $\mu\ fs'$
from *j* **have** $j \le i$ **by** *simp*
let *?R* = *rat-of-int*

have *mu-bound-factor*: μ-*bound-row fs'* $(4 * bnd)$ *i*
proof (*intro* μ-*bound-rowI*)
fix *k*
assume *ki*: $k \le i$
from μ-*bound-rowD*[*OF mu-bnd*] **have** *bnd-i*: $\bigwedge j.\ j \le i \Longrightarrow (?mu\ i\ j)\hat{}2 \le bnd$ **by** *auto*
have *bnd-ik*: $(?mu\ i\ k)^2 \le bnd$ **using** *bnd-i*[*OF ki*] **by** *auto*
have *bnd-ij*: $(?mu\ i\ j)^2 \le bnd$ **using** *bnd-i*[*OF* ‹$j \le i$›] **by** *auto*
from μ-*bound-row-1*[*OF mu-bnd*] **have** *bnd1*: $bnd \ge 1$ $bnd \ge 0$ **by** *auto*
show $(?mu'\ i\ k)^2 \le 4 * bnd$
proof (*cases* $k > j$)
case *True*
show *?thesis*
by (*subst main*(*5*), (*insert True ki i bnd1*, *auto*)[*3*], *intro order.trans*[*OF bnd-ik*], *auto*)
next

```
      case False
      hence kj: k ≤ j by auto
      show ?thesis
      proof (cases k = j)
        case True
          have small: abs (?mu' i k) ≤ 1/2 using main(2) j unfolding True
μ-small-row-def by auto
        show ?thesis using mult-mono[OF small small] using bnd1
          by (auto simp: power2-eq-square)
      next
        case False
        with kj have k-j: k < j by auto
        define M where M = max (abs (?mu i k)) (max (abs (?mu i j)) (1/2))
        have M0: M ≥ 0 unfolding M-def by auto
        let ?new-mu = ?mu i k − ?R c ∗ ?mu j k
        have abs ?new-mu ≤ abs (?mu i k) + abs (?R c ∗ ?mu j k) by simp
         also have ... = abs (?mu i k) + abs (?R c) ∗ abs (?mu j k) unfolding
abs-mult ..
        also have ... ≤ abs (?mu i k) + (abs (?mu i j) + 1/2) ∗ (1/2)
        proof (rule add-left-mono[OF mult-mono], unfold c)
          show |?R (round (?mu i j))| ≤ |?mu i j| + 1 / 2 unfolding round-def
by linarith
            from inv(10)[unfolded gram-schmidt-fs.reduced-def, THEN conjunct2,
rule-format, OF ‹j < i› k-j]
          show |?mu j k| ≤ 1/2 .
        qed auto
        also have ... ≤ M + (M + M) ∗ (1/2)
        by (rule add-mono[OF - mult-right-mono[OF add-mono]], auto simp: M-def)
        also have ... = 2 ∗ M by auto
        finally have le: abs ?new-mu ≤ 2 ∗ M .
        have (?mu' i k)^2 = ?new-mu^2
          by (subst main(5), insert kj False i j, auto)
        also have ... ≤ (2 ∗ M)^2 unfolding abs-le-square-iff[symmetric] using
le M0 by auto
        also have ... = 4 ∗ M^2 by simp
        also have ... ≤ 4 ∗ bnd
        proof (rule mult-left-mono)
          show M^2 ≤ bnd using bnd-ij bnd-ik bnd1 unfolding M-def
            by (auto simp: max-def power2-eq-square)
        qed auto
        finally show ?thesis .
      qed
    qed
  qed
  also have 4 ∗ bnd = (4 ^(1 + (m − 1 − Suc j)) ∗ of-nat (N ^(m − 1) ∗ m))
unfolding bnd
    by simp
  also have 1 + (m − 1 − Suc j) = m − 1 − j using i j by auto
  finally show bnd: μ-bound-row-inner fs' i j by auto
```

```
  show gbnd: g-bound fs' using gbnd unfolding g-bound-def
    using main(4) by auto

  note inv' = LLL-invD[OF Linv']
  show f-bound False i fs'
    unfolding f-bound-def
  proof (intro allI impI, goal-cases)
    case (1 jj)
    show ?case
    proof (cases jj = i)
      case False
      with 1 fbnd[unfolded f-bound-def] have ‖fs ! jj‖² ≤ int (N * m) by auto
      thus ?thesis unfolding fs' using False 1 inv(2−) by auto
    next
      case True
    have of-int ‖fs' ! i‖² = ‖RAT fs' ! i‖² using i inv' by (auto simp: sq-norm-of-int)
      also have ... ≤ rat-of-nat (Suc i * N) * (4 ^ (m − 1 − j) * rat-of-nat (N ^
(m − 1) * m))
        using sq-norm-fs-mu-g-bound[OF inv'(1,6) i bnd gbnd] i inv'
        unfolding sq-norm-of-int[symmetric]
        by (auto simp: ac-simps)
      also have ... = rat-of-int ( int (Suc i * N) * (4 ^ (m − 1 − j) * (N ^ (m
− 1) * m)))
        by simp
      finally have ‖fs' ! i‖² ≤ int (Suc i * N) * (4 ^ (m − 1 − j) * (N ^ (m −
1) * m)) by linarith
      also have ... = int (Suc i) * 4 ^ (m − 1 − j) * (int N ^ (Suc (m − 1))) *
int m
        unfolding of-nat-mult by (simp add: ac-simps)
      also have ... = int (Suc i) * 4 ^ (m − 1 − j) * int N ^ m * int m using i
j by simp
      also have ... ≤ int m * 4 ^ (m − 1) * int N ^ m * int m
      by (rule mult-right-mono[OF mult-right-mono[OF mult-mono[OF - pow-mono-exp]]],
insert i, auto)
       finally have ‖fs' ! i‖² ≤ int (4 ^ (m − 1) * N ^ m * m * m) unfolding
of-nat-mult by (simp add: ac-simps)
      thus ?thesis unfolding True by auto
    qed
  qed
qed
end

context LLL-with-assms
begin
```

9.5.1 *LLL-bound-invariant* is maintained during execution of *reduce-basis*

lemma *basis-reduction-add-rows-enter-bound*: **assumes** *binv*: *LLL-bound-invariant True True i fs*
and *i*: $i < m$
shows *LLL-bound-invariant False True i fs*
μ*-bound-row-inner fs i i*
proof (*rule bound-invI*)
from *bound-invD*[*OF binv*]
have *Linv*: *LLL-invariant True i fs* (**is** *?g1*) **and** *fbnd*: *f-bound True i fs*
and *gbnd*: *g-bound fs* **by** *auto*
note *Linvw* = *LLL-inv-imp-w*[*OF Linv*]
interpret *fs*: *fs-int'* *n m fs-init fs*
by *standard* (*use Linvw* **in** *auto*)
note *inv* = *LLL-invD*[*OF Linv*]
show *LLL-invariant True i fs* **by** *fact*
show *fbndF*: *f-bound False i fs* **using** *f-bound-True-arbitrary*[*OF fbnd*] **.**
have *N0*: $N > 0$ **using** *LLL-inv-N-pos*[*OF Linvw gbnd*] *i* **by** *auto*
{
fix *j*
assume *ji*: $j < i$
have $(\mu\ fs\ i\ j)^2 \leq$ *gs.Gramian-determinant* (*RAT fs*) $j * \|RAT\ fs\ !\ i\|^2$
using *ji i inv* **by** (*intro fs.gs.mu-bound-Gramian-determinant*) (*auto*)
also have *gs.Gramian-determinant* (*RAT fs*) *j* = *of-int* (*d fs j*) **unfolding** *d-def*
by (*subst fs.of-int-Gramian-determinant*, *insert ji i inv*(2−), *auto simp*: *set-conv-nth*)
also have $\|RAT\ fs\ !\ i\|^2$ = *of-int* $\|fs\ !\ i\|^2$ **using** *i inv*(2−) **by** (*auto simp*: *sq-norm-of-int*)
also have *of-int* (*d fs j*) * ... ≤ *rat-of-nat* ($N\hat{}j$) * *of-int* $\|fs\ !\ i\|^2$
by (*rule mult-right-mono*, *insert ji i d-approx*[*OF Linvw gbnd*, *of j*], *auto*)
also have ... ≤ *rat-of-nat* ($N\hat{}(m-2)$) * *of-int* (*int* ($N * m$))
by (*intro mult-mono*, *unfold of-nat-le-iff of-int-le-iff*, *rule pow-mono-exp*,
insert fbnd[*unfolded f-bound-def*, *rule-format*, *of i*] *N0 ji i*, *auto*)
also have ... = *rat-of-nat* ($N\hat{}(m-2) * N * m$) **by** *simp*
also have $N\hat{}(m-2) * N = N\hat{}(Suc\ (m-2))$ **by** *simp*
also have *Suc* $(m-2) = m-1$ **using** *ji i* **by** *auto*
finally have $(\mu\ fs\ i\ j)^2 \leq$ *of-nat* ($N\ \hat{}\ (m-1) * m$) **.**
} **note** *mu-bound* = *this*
show *mu-bnd*: μ*-bound-row-inner fs i i*
proof (*rule* μ*-bound-rowI*)
fix *j*
assume *j*: $j \leq i$
have $(\mu\ fs\ i\ j)^2 \leq 1 *$ *of-nat* ($N\ \hat{}\ (m-1) * m$)
proof (*cases* $j = i$)
case *False*
with *mu-bound*[*of j*] *j* **show** *?thesis* **by** *auto*
next
case *True*

```
    show ?thesis unfolding True fs.gs.μ.simps using i N0 by auto
   qed
   also have ... ≤ 4 ^ (m − 1 − i) ∗ of-nat (N ^ (m − 1) ∗ m)
    by (rule mult-right-mono, auto)
   finally show (μ fs i j)² ≤ 4 ^ (m − 1 − i) ∗ rat-of-nat (N ^ (m − 1) ∗ m) .
  qed
  show g-bound fs by fact
qed

lemma basis-basis-reduction-add-rows-loop-leave:
  assumes binv: LLL-bound-invariant False True i fs
  and mu-small: μ-small-row i fs 0
  and mu-bnd: μ-bound-row-inner fs i 0
  and i: i < m
shows LLL-bound-invariant True False i fs
proof −
  note Linv = bound-invD(1)[OF binv]
 from mu-small have mu-small: μ-small fs i unfolding μ-small-row-def μ-small-def
by auto
  note inv = LLL-invD[OF Linv]
  interpret gs1: gram-schmidt-fs-int n RAT fs
    by (standard) (use inv gs.lin-indpt-list-def in ‹auto simp add: vec-hom-Ints›)
  note fbnd = bound-invD(2)[OF binv]
  note gbnd = bound-invD(3)[OF binv]
  {
    fix ii
    assume ii: ii < m
    have ‖fs ! ii‖² ≤ int (N ∗ m)
    proof (cases ii = i)
      case False
      thus ?thesis using ii fbnd[unfolded f-bound-def] by auto
    next
      case True
      have row: μ-bound-row fs 1 i
      proof (intro μ-bound-rowI)
        fix j
        assume j: j ≤ i
        from mu-small[unfolded μ-small-def, rule-format, of j]
         have abs (μ fs i j) ≤ 1 using j unfolding μ-small-def by (cases j = i,
force simp: gs1.μ.simps, auto)
          from mult-mono[OF this this] show (μ fs i j)² ≤ 1 by (auto simp:
power2-eq-square)
      qed
      have rat-of-int ‖fs ! i‖² ≤ rat-of-int (int (Suc i ∗ N))
        using sq-norm-fs-mu-g-bound[OF inv(1,6) i row gbnd] by auto
      hence ‖fs ! i‖² ≤ int (Suc i ∗ N) by linarith
      also have ... = int N ∗ int (Suc i) unfolding of-nat-mult by simp
      also have ... ≤ int N ∗ int m
        by (rule mult-left-mono, insert i, auto)
```

```
    also have ... = int (N * m) by simp
    finally show ?thesis unfolding True .
  qed
 }
 hence f-bound: f-bound True i fs unfolding f-bound-def by auto
 with binv show ?thesis using basis-reduction-add-row-done[OF Linv i assms(2)]

  by (auto simp: LLL-bound-invariant-def)
qed


lemma basis-reduction-add-rows-loop-bound: assumes
 binv: LLL-bound-invariant False True i fs
 and mu-small: μ-small-row i fs j
 and mu-bnd: μ-bound-row-inner fs i j
 and res: basis-reduction-add-rows-loop i fs j = fs'
 and i: i < m
 and j: j ≤ i
shows LLL-bound-invariant True False i fs'
 using assms
proof (induct j arbitrary: fs)
 case (0 fs)
 note binv = 0(1)
 from basis-basis-reduction-add-rows-loop-leave[OF 0(1-3) i] 0(4)
 show ?case by auto
next
 case (Suc j fs)
 note binv = Suc(2)
 note Linv = bound-invD(1)[OF binv]
 note Linvw = LLL-inv-imp-w[OF Linv]
 from Suc have j: j < i by auto
 let ?c = round (μ fs i j)
 note step = basis-reduction-add-row-main-bound[OF Suc(2) i j refl refl Suc(3-4)]
 note step' = basis-reduction-add-row-main(2,3,5)[OF Linvw i j refl]
 note step' = step'(1)[OF Linv] step'(2-)
 show ?case
 proof (cases ?c = 0)
   case True
   note inv = LLL-invD[OF Linv]
   from inv(5)[OF i] inv(5)[of j] i j
   have id: fs[i := fs ! i - 0 ·v fs ! j] = fs
     by (intro nth-equalityI, insert inv i, auto)
   show ?thesis
     by (rule Suc(1), insert step step' id True Suc(2-), auto)
 next
   case False
  show ?thesis using Suc(1)[OF step(1) step'(2) step(2)] Suc(2-) False step'(3)
by auto
 qed
```

qed

lemma *basis-reduction-add-rows-bound*: **assumes**
binv: *LLL-bound-invariant True upw i fs*
and *res*: *basis-reduction-add-rows upw i fs* = *fs′*
and *i*: *i* < *m*
shows *LLL-bound-invariant True False i fs′*
proof −
note *def* = *basis-reduction-add-rows-def*
show *?thesis*
proof (*cases upw*)
case *False*
with *res binv* **show** *?thesis* **by** (*simp add*: *def*)
next
case *True*
with *binv* **have** *binv*: *LLL-bound-invariant True True i fs* **by** *auto*
note *start* = *basis-reduction-add-rows-enter-bound*[*OF this i*]
from *res*[*unfolded def*] *True*
have *basis-reduction-add-rows-loop i fs i* = *fs′* **by** *auto*
from *basis-reduction-add-rows-loop-bound*[*OF start*(*1*) *μ-small-row-refl start*(*2*) *this i le-refl*]
show *?thesis* **by** *auto*
qed
qed

lemma *g-bnd-swap*:
assumes *i*: *i* < *m i* ≠ *0*
and *Linv*: *LLL-invariant-weak fs*
and *mu-F1-i*: |*μ fs i* (*i*−*1*)| ≤ *1* / *2*
and *cond*: *sq-norm* (*gso fs* (*i* − *1*)) > *α* ∗ *sq-norm* (*gso fs i*)
and *fs′-def*: *fs′* = *fs*[*i* := *fs* ! (*i* − *1*), *i* − *1* := *fs* ! *i*]
and *g-bnd*: *g-bnd B fs*
shows *g-bnd B fs′*
proof −
note *inv* = *LLL-inv-wD*[*OF Linv*]
have *choice*: *fs′* ! *k* = *fs* ! *k* ∨ *fs′* ! *k* = *fs* ! *i* ∨ *fs′* ! *k* = *fs* ! (*i* − *1*) **for** *k*
unfolding *fs′-def* **using** *i inv*(*6*) **by** (*cases k* = *i*; *cases k* = *i* − *1*, *auto*)

let *?g1* = λ *i*. *gso fs i*
let *?g2* = λ *i*. *gso fs′ i*
let *?n1* = λ *i*. *sq-norm* (*?g1 i*)
let *?n2* = λ *i*. *sq-norm* (*?g2 i*)
from *g-bnd*[*unfolded g-bnd-def*] **have** *short*: ⋀ *k*. *k* < *m* ⟹ *?n1 k* ≤ *B* **by** *auto*
from *short*[*of i* − *1*] *i*
have *short-im1*: *?n1* (*i* − *1*) ≤ *B* **by** *auto*
note *swap* = *basis-reduction-swap-main*[*OF Linv disjI2*[*OF mu-F1-i*] *i cond fs′-def*]
note *updates* = *swap*(*4*,*5*)
note *Linv′* = *swap*(*1*)

note *inv′ = LLL-inv-wD[OF Linv′]*
note *inv = LLL-inv-wD[OF Linv]*
interpret *gs1*: *gram-schmidt-fs-int n RAT fs*
by (*standard*) (*use inv gs.lin-indpt-list-def* **in** ‹*auto simp add*: *vec-hom-Ints*›)
interpret *gs2*: *gram-schmidt-fs-int n RAT fs′*
by (*standard*) (*use inv′ gs.lin-indpt-list-def* **in** ‹*auto simp add*: *vec-hom-Ints*›)
let *?mu1 = μ fs*
let *?mu2 = μ fs′*
let *?mu = ?mu1 i (i − 1)*
have *mu*: *abs ?mu ≤ 1/2* **using** *mu-F1-i* **.**
have *?n2 (i − 1) = ?n1 i + ?mu ∗ ?mu ∗ ?n1 (i − 1)*
by (*subst updates*(*2*), *insert i*, *auto*)
also have *. . . = inverse α ∗ (α ∗ ?n1 i) + (?mu ∗ ?mu) ∗ ?n1 (i − 1)*
using *α* **by** *auto*
also have *. . . ≤ inverse α ∗ ?n1 (i − 1) + (abs ?mu ∗ abs ?mu) ∗ ?n1 (i − 1)*
by (*rule add-mono*[*OF mult-left-mono*], *insert cond α*, *auto*)
also have *. . . = (inverse α + abs ?mu ∗ abs ?mu) ∗ ?n1 (i − 1)* **by** (*auto simp*: *field-simps*)
also have *. . . ≤ (inverse α + (1/2) ∗ (1/2)) ∗ ?n1 (i − 1)*
by (*rule mult-right-mono*[*OF add-left-mono*[*OF mult-mono*]], *insert mu*, *auto*)

also have *inverse α + (1/2) ∗ (1/2) = reduction* **unfolding** *reduction-def* **using** *α0*
by (*auto simp*: *field-simps*)
also have *. . . ∗ ?n1 (i − 1) ≤ 1 ∗ ?n1 (i − 1)*
by (*rule mult-right-mono*, *auto simp*: *reduction*)
finally have *n2im1*: *?n2 (i − 1) ≤ ?n1 (i − 1)* **by** *simp*
show *g-bnd B fs′* **unfolding** *g-bnd-def*
proof (*intro allI impI*)
fix *k*
assume *km*: *k < m*
consider (*ki*) *k = i* | (*im1*) *k = i − 1* | (*other*) *k ≠ i k ≠ i−1* **by** *blast*
thus *?n2 k ≤ B*
proof *cases*
case *other*
from *short*[*OF km*] **have** *?n1 k ≤ B* **by** *auto*
also have *?n1 k = ?n2 k* **using** *km other*
by (*subst updates*(*2*), *auto*)
finally show *?thesis* **by** *simp*
next
case *im1*
have *?n2 k = ?n2 (i − 1)* **unfolding** *im1* **..**
also have *. . . ≤ ?n1 (i − 1)* **by** *fact*
also have *. . . ≤ B* **using** *short-im1* **by** *auto*
finally show *?thesis* **by** *simp*
next
case *ki*
have *?n2 k = ?n2 i* **unfolding** *ki* **using** *i* **by** *auto*
also have *. . . ≤ ?n1 (i − 1)*

```
    proof −
      let ?f1 = λ i. RAT fs ! i
      let ?f2 = λ i. RAT fs′ ! i
      define u where u = gs.sumlist (map (λj. ?mu1 (i − 1) j ·v ?g1 j) [0..<i
− 1])
      define U where U = ?f1 ‘ {0 ..< i − 1} ∪ {?f1 i}
      have g2i: ?g2 i ∈ Rn using i inv′ by simp
      have U: U ⊆ Rn unfolding U-def using inv i by auto
      have uU: u ∈ gs.span U
      proof −
        have im1: i − 1 ≤ m using i by auto
        have G1: ?g1 ‘ {0..< i − 1} ⊆ Rn using inv(5) i by auto
        have u ∈ gs.span (?g1 ‘ {0 ..< i − 1}) unfolding u-def
          by (rule gs.sumlist-in-span[OF G1], unfold set-map, insert G1,
            auto intro!: gs.smult-in-span intro: gs.span-mem)
        also have gs.span (?g1 ‘ {0 ..< i − 1}) = gs.span (?f1 ‘ {0 ..< i − 1})
          apply(subst gs1.partial-span, insert im1 inv, unfold gs.lin-indpt-list-def)
           apply(blast)
          apply(rule arg-cong[of - - gs.span])
          apply(subst nth-image[symmetric])
          by (insert i inv, auto)
        also have ... ⊆ gs.span U unfolding U-def
          by (rule gs.span-is-monotone, auto)
        finally show ?thesis .
      qed
      from i have im1: i − 1 < m by auto
      have u: u ∈ Rn using uU U by simp
      have id-u: u + (?g1 (i − 1) − ?g2 i) = u + ?g1 (i − 1) − ?g2 i
        using u g2i inv(5)[OF im1] by auto
      have list-id: [0..<Suc (i − 1)] = [0..< i − 1] @ [i − 1]
        map f [x] = [f x] for f x by auto
      have gs.is-oc-projection (gs2.gso i) (gs.span (gs2.gso ‘ {0..<i})) ((RAT fs′)
! i)
        using i inv′ unfolding gs.lin-indpt-list-def
        by (intro gs2.gso-oc-projection-span(2)) auto
       then have gs.is-oc-projection (?g2 i) (gs.span (gs2.gso ‘ {0 ..< i})) (?f1
(i − 1))
        unfolding fs′-def using inv(6) i by auto
      also have ?f1 (i − 1) = u + ?g1 (i − 1)
      apply(subst gs1.fi-is-sum-of-mu-gso, insert im1 inv, unfold gs.lin-indpt-list-def)
        apply(blast)
        unfolding list-id map-append u-def
       by (subst gs.M.sumlist-snoc, insert i, auto simp: gs1.μ.simps intro!: inv(5))
      also have gs.span (gs2.gso ‘ {0 ..< i}) = gs.span (set (take i (RAT fs′)))
        using inv′ ‹i < m› unfolding gs.lin-indpt-list-def
        by (subst gs2.partial-span) auto
      also have set (take i (RAT fs′)) = ?f2 ‘ {0 ..< i} using inv′(6) i
        by (subst nth-image[symmetric], auto)
      also have {0 ..< i} = {0 ..< i − 1} ∪ {(i − 1)} using i by auto
```

```
      also have ?f2 ` ... = ?f2 ` {0 ..< i - 1} ∪ {?f2 (i - 1)} by auto
      also have ... = U unfolding U-def fs'-def
        by (rule arg-cong2[of - - - - (∪)], insert i inv(6), force+)
       finally have gs.is-oc-projection (?g2 i) (gs.span U) (u + ?g1 (i - 1)) .

      hence proj: gs.is-oc-projection (?g2 i) (gs.span U) (?g1 (i - 1))
        unfolding gs.is-oc-projection-def using gs.span-add[OF U uU, of ?g1 (i
- 1) - ?g2 i]
        inv(5)[OF im1] g2i u id-u by (auto simp: U)
    from gs.is-oc-projection-sq-norm[OF this gs.span-is-subset2[OF U] inv(5)[OF
im1]]
      show ?n2 i ≤ ?n1 (i - 1) .
    qed
    also have ... ≤ B by fact
    finally show ?thesis .
  qed
 qed
qed


lemma basis-reduction-swap-bound: assumes
  binv: LLL-bound-invariant True False i fs
  and res: basis-reduction-swap i fs = (upw',i',fs')
  and cond: sq-norm (gso fs (i - 1)) > α * sq-norm (gso fs i)
  and i: i < m i ≠ 0
shows LLL-bound-invariant True upw' i' fs'
proof (rule bound-invI)
  note Linv = bound-invD(1)[OF binv]
  from basis-reduction-swap[OF Linv res cond i]
  show Linv': LLL-invariant upw' i' fs' by auto
  from res[unfolded basis-reduction-swap-def]
  have id: i' = i - 1 fs' = fs[i := fs ! (i - 1), i - 1 := fs ! i] by auto
  from LLL-invD(6)[OF Linv] i
  have choice: fs' ! k = fs ! k ∨ fs' ! k = fs ! i ∨ fs' ! k = fs ! (i - 1) for k
    unfolding id by (cases k = i; cases k = i - 1, auto)
  from bound-invD(2)[OF binv] i
  show f-bound True i' fs' unfolding id(1) f-bound-def
  proof (intro allI impI, goal-cases)
    case (1 k)
    thus ?case using choice[of k] by auto
  qed

  from bound-invD(3)[OF binv, unfolded g-bound-def]
  have gbnd: g-bnd (of-nat N) fs unfolding g-bnd-def .
  from LLL-invD(11)[OF Linv, unfolded μ-small-def] i
  have abs (μ fs i (i - 1)) ≤ 1/2 by auto
  from g-bnd-swap[OF i LLL-inv-imp-w[OF Linv] this cond id(2) gbnd]
  have g-bnd (rat-of-nat N) fs' .
  thus g-bound fs' unfolding g-bnd-def g-bound-def .
```

qed

lemma *basis-reduction-step-bound*: **assumes**
binv: *LLL-bound-invariant True upw i fs*
and *res*: *basis-reduction-step upw i fs* = (*upw'*,*i'*,*fs'*)
and *i*: $i < m$
shows *LLL-bound-invariant True upw' i' fs'*
proof −
note *def* = *basis-reduction-step-def*
obtain *fs''* **where** *fs''*: *basis-reduction-add-rows upw i fs* = *fs''* **by** *auto*
show *?thesis*
proof (*cases i* = *0*)
case *True*
from *increase-i-bound*[*OF binv i True*] *res True*
show *?thesis* **by** (*auto simp*: *def*)
next
case *False*
hence *id*: (*i* = *0*) = *False* **by** *auto*
note *res* = *res*[*unfolded def id if-False fs'' Let-def*]
let *?x* = *sq-norm* (*gso fs''* (*i* − *1*))
let *?y* = α ∗ *sq-norm* (*gso fs''* *i*)
from *basis-reduction-add-rows-bound*[*OF binv fs''* *i*]
have *binv*: *LLL-bound-invariant True False i fs''* **by** *auto*
show *?thesis*
proof (*cases ?x* ≤ *?y*)
case *True*
from *increase-i-bound*[*OF binv i - True*] *True res*
show *?thesis* **by** *auto*
next
case *gt*: *False*
hence *?x* > *?y* **by** *auto*
from *basis-reduction-swap-bound*[*OF binv - this i False*] *gt res*
show *?thesis* **by** *auto*
qed
qed
qed

lemma *basis-reduction-main-bound*: **assumes** *LLL-bound-invariant True upw i fs*

and *res*: *basis-reduction-main* (*upw*,*i*,*fs*) = *fs'*
shows *LLL-bound-invariant True True m fs'*
using *assms*
proof (*induct LLL-measure i fs arbitrary*: *i fs upw rule*: *less-induct*)
case (*less i fs upw*)
have *id*: *LLL-bound-invariant True upw i fs* = *True* **using** *less* **by** *auto*
note *res* = *less*(*3*)[*unfolded basis-reduction-main.simps*[*of upw i fs*] *id*]
note *inv* = *less*(*2*)
note *IH* = *less*(*1*)
note *Linv* = *bound-invD*(*1*)[*OF inv*]

show *?case*
proof (*cases i < m*)
case *i*: *True*
obtain *i' fs' upw'* **where** *step*: *basis-reduction-step upw i fs* = (*upw'*,*i'*,*fs'*)
(**is** *?step* = -) **by** (*cases ?step*, *auto*)
note *decrease* = *basis-reduction-step*(*2*)[*OF Linv step i*]
from *IH*[*OF decrease basis-reduction-step-bound*(*1*)[*OF inv step i*]] *res*[*unfolded step*] *i Linv*
show *?thesis* **by** *auto*
next
case *False*
with *LLL-invD*[*OF Linv*] **have** *i*: *i* = *m* **by** *auto*
with *False res inv* **have** *LLL-bound-invariant True upw m fs'* **by** *auto*
thus *?thesis* **by** (*auto simp*: *LLL-invariant-def LLL-bound-invariant-def*)
qed
qed

lemma *LLL-inv-initial-state-bound*: *LLL-bound-invariant True True 0 fs-init*
proof (*intro bound-invI*[*OF LLL-inv-initial-state - g-bound-fs-init*])
{
fix *i*
assume *i*: *i* < *m*
let *?N* = *map* (*nat o sq-norm*) *fs-init*
let *?r* = *rat-of-int*
from *i* **have** *mem*: *nat* (*sq-norm* (*fs-init* ! *i*)) ∈ *set ?N* **using** *fs-init len* **unfolding** *set-conv-nth* **by** *force*
from *mem-set-imp-le-max-list*[*OF - mem*]
have *FN*: *nat* (*sq-norm* (*fs-init* ! *i*)) ≤ *N* **unfolding** *N-def* **by** *force*
hence ‖*fs-init* ! *i*‖² ≤ *int N* **using** *i* **by** *auto*
also have ... ≤ *int* (*N* ∗ *m*) **using** *i* **by** *fastforce*
finally have *f-bnd*: ‖*fs-init* ! *i*‖² ≤ *int* (*N* ∗ *m*) **.**
}
thus *f-bound True 0 fs-init* **unfolding** *f-bound-def* **by** *auto*
qed

lemma *reduce-basis-bound*: **assumes** *res*: *reduce-basis* = *fs*
shows *LLL-bound-invariant True True m fs*
using *basis-reduction-main-bound*[*OF LLL-inv-initial-state-bound res*[*unfolded reduce-basis-def*]] **.**

9.5.2 Bound extracted from *LLL-bound-invariant*.

fun *f-bnd* :: *bool* ⇒ *nat* **where**
f-bnd False = *2* ^ (*m* − *1*) ∗ *N* ^ *m* ∗ *m*
| *f-bnd True* = *N* ∗ *m*

lemma *f-bnd-mono*: *f-bnd outside* ≤ *f-bnd False*
proof (*cases outside*)
case *out*: *True*

```
  show ?thesis
  proof (cases N = 0 ∨ m = 0)
    case True
    thus ?thesis using out by auto
  next
    case False
    hence 0: N > 0 m > 0 by auto
    let ?num = (2 ^ (m - 1) * N ^ m)
    have (N * m) * 1 ≤ (N * m) * (2 ^ (m - 1) * N ^ (m - 1))
      by (rule mult-left-mono, insert 0, auto)
    also have ... = 2 ^ (m - 1) * N ^ (Suc (m - 1)) * m  by simp
    also have Suc (m - 1) = m using 0 by simp
    finally show ?thesis using out by auto
  qed
qed auto

lemma aux-bnd-mono: N * m ≤ (4 ^ (m - 1) * N ^ m * m * m)
proof (cases N = 0 ∨ m = 0)
  case False
  hence 0: N > 0 m > 0 by auto
  let ?num = (4 ^ (m - 1) * N ^ m * m * m)
  have (N * m) * 1 ≤ (N * m) * (4 ^ (m - 1) * N ^ (m - 1) * m)
    by (rule mult-left-mono, insert 0, auto)
  also have ... = 4 ^ (m - 1) * N ^ (Suc (m - 1)) * m * m by simp
  also have Suc (m - 1) = m using 0 by simp
  finally show N * m ≤ ?num by simp
qed auto

context fixes outside upw k fs
  assumes binv: LLL-bound-invariant outside upw k fs
begin

lemma LLL-f-bnd:
  assumes i: i < m and j: j < n
shows |fs ! i $ j| ≤ f-bnd outside
proof −
  from bound-invD[OF binv]
  have inv: LLL-invariant upw k fs
    and fbnd: f-bound outside k fs
    and gbnd: g-bound fs by auto
  note invw = LLL-inv-imp-w[OF inv]
  from LLL-inv-N-pos[OF invw gbnd] i have N0: N > 0 by auto
  note inv = LLL-invD[OF inv]
  from inv i have fsi: fs ! i ∈ carrier-vec n by auto
  have one: |fs ! i $ j|^1 ≤ |fs ! i $ j|^2
    by (cases fs ! i $ j ≠ 0, intro pow-mono-exp, auto)
  let ?num = (4 ^ (m - 1) * N ^ m * m * m)
  let ?sq-bnd = if i ≠ k ∨ outside then int (N * m) else int ?num
  have |fs ! i $ j|^2 ≤ ‖fs ! i‖² using fsi j by (metis vec-le-sq-norm)
```

```
  also have ... ≤ ?sq-bnd
    using fbnd[unfolded f-bound-def, rule-format, OF i] by auto
  finally have two: (fs ! i $ j)^2 ≤ ?sq-bnd by simp
  show ?thesis
  proof (cases outside)
    case True
    with one two show ?thesis by auto
  next
    case False
    let ?num2 = (2 ^ (m - 1) * N ^ m * m)
    have four: (4 :: nat) = 2^2 by auto
    have (fs ! i $ j)^2 ≤ int (max (N * m) ?num)
      by (rule order.trans[OF two], auto simp: of-nat-mult[symmetric] simp del:
of-nat-mult)
    also have max (N * m) ?num = ?num using aux-bnd-mono by presburger
    also have int ?num = int ?num * 1 by simp
    also have ... ≤ int ?num * N ^ m
      by (rule mult-left-mono, insert N0, auto)
    also have ... = int (?num * N ^ m) by simp
     also have ?num * N ^ m = ?num2^2 unfolding power2-eq-square four
power-mult-distrib
      by simp
    also have int ... = (int ?num2)^2 by simp
    finally have (fs ! i $ j)^2 ≤ (int (f-bnd outside))^2 using False by simp
    thus ?thesis unfolding abs-le-square-iff[symmetric] by simp
  qed
qed

lemma LLL-gso-bound:
  assumes i: i < m and j: j < n
  and quot: quotient-of (gso fs i $ j) = (num, denom)
shows |num|   ≤ N ^ m
  and |denom| ≤ N ^ m
proof −
  from bound-invD[OF binv]
  have inv: LLL-invariant upw k fs
    and gbnd: g-bound fs by auto
  note invw = LLL-inv-imp-w[OF inv]
  note * = LLL-invD[OF inv]
  interpret fs: fs-int' n m fs-init fs
    by standard (use invw in auto)
  note d-approx[OF invw gbnd i, unfolded d-def]
  let ?r = rat-of-int
  have int: (gs.Gramian-determinant (RAT fs) i ·_v (gso fs i)) $ j ∈ ℤ
  proof −
    have of-int-hom.vec-hom (fs ! j) $ i ∈ ℤ if i < n j < m for i j
      using that assms * by (intro vec-hom-Ints) (auto)
    then show ?thesis
    using * gs.gso-connect snd-gram-schmidt-int assms unfolding gs.lin-indpt-list-def
```

```
    by (intro fs.gs.d-gso-Ints) (auto)
  qed
  have gsi: gso fs i ∈ Rn using ∗(5)[OF i] .
  have gs-sq: |(gso fs i $ j)|^2 ≤ rat-of-nat N
    by(rule order-trans, rule vec-le-sq-norm[of - n])
      (use gsi assms gbnd ∗ LLL.g-bound-def in auto)
  from i have m ∗ m ≠ 0
    by auto
  then have N0: N ≠ 0
    using less-le-trans[OF LLL-D-pos[OF invw] D-approx[OF invw gbnd]] by auto
  have |(gso fs i $ j)| ≤ max 1 |(gso fs i $ j)|
    by simp
  also have ... ≤ (max 1 |gso fs i $ j|)^2
    by (rule self-le-power, auto)
  also have ... ≤ of-nat N
    using gs-sq N0 unfolding max-def by auto
  finally have gs-bound: |(gso fs i $ j)| ≤ of-nat N .
  have gs.Gramian-determinant (RAT fs) i = rat-of-int (gs.Gramian-determinant
fs i)
   using assms ∗(4−6) carrier-vecD nth-mem by (intro fs.of-int-Gramian-determinant)
(simp, blast)
  with int have (of-int (d fs i) ·_v gso fs i) $ j ∈ ℤ
    unfolding d-def by simp
  also have (of-int (d fs i) ·_v gso fs i) $ j = of-int (d fs i) ∗ (gso fs i $ j)
    using gsi i j by auto
  finally have l: of-int (d fs i) ∗ gso fs i $ j ∈ ℤ
    by auto
  have num: rat-of-int |num| ≤ of-int (d fs i ∗ int N) and denom: denom ≤ d fs i
    using quotient-of-bounds[OF quot l LLL-d-pos[OF invw] gs-bound] i by auto
  from num have num: |num| ≤ d fs i ∗ int N
    by linarith
  from d-approx[OF invw gbnd i] have d: d fs i ≤ int (N ^ i)
    by linarith
  from denom d have denom: denom ≤ int (N ^ i)
    by auto
  note num also have d fs i ∗ int N ≤ int (N ^ i) ∗ int N
    by (rule mult-right-mono[OF d], auto)
  also have ... = int (N ^ (Suc i))
    by simp
  finally have num: |num| ≤ int (N ^ (i + 1))
    by auto
  {
    fix jj
    assume jj ≤ i + 1
    with i have jj ≤ m by auto
    from pow-mono-exp[OF - this, of N] N0
    have N^jj ≤ N^m by auto
    hence int (N^jj) ≤ int (N^m) by linarith
  } note j-m = this
```

```
  have |denom| = denom
    using quotient-of-denom-pos[OF quot] by auto
  also have ... ≤ int (N ^ i)
    by fact
  also have ... ≤ int (N ^ m)
    by (rule j-m, auto)
  finally show |denom| ≤ int (N ^ m)
    by auto
  show |num| ≤ int (N ^ m)
    using j-m[of i+1] num by auto
qed

lemma LLL-f-bound:
  assumes i: i < m and j: j < n
shows |fs ! i $ j| ≤ N ^ m * 2 ^ (m - 1) * m
proof -
  have |fs ! i $ j| ≤ int (f-bnd outside) using LLL-f-bnd[OF i j] by auto
  also have ... ≤ int (f-bnd False) using f-bnd-mono[of outside] by presburger
  also have ... = int (N ^ m * 2 ^ (m - 1) * m) by simp
  finally show ?thesis .
qed

lemma LLL-d-bound:
  assumes i: i ≤ m
shows abs (d fs i) ≤ N ^ i ∧ abs (d fs i) ≤ N ^ m
proof (cases m = 0)
  case True
  with i have id: m = 0 i = 0 by auto
  show ?thesis unfolding id(2) using id unfolding gs.Gramian-determinant-0
d-def by auto
next
  case m: False
  from bound-invD[OF binv]
  have inv: LLL-invariant upw k fs
    and gbnd: g-bound fs by auto
  note invw = LLL-inv-imp-w[OF inv]
  from LLL-inv-N-pos[OF invw gbnd] m have N: N > 0 by auto
  let ?r = rat-of-int
  from d-approx-main[OF invw gbnd i m]
  have rat-of-int (d fs i) ≤ of-nat (N ^ i)
    by auto
  hence one: d fs i ≤ N ^ i by linarith
  also have ... ≤ N ^ m unfolding of-nat-le-iff
    by (rule pow-mono-exp, insert N i, auto)
  finally have d fs i ≤ N ^ m by simp
  with LLL-d-pos[OF invw i] one
  show ?thesis by auto
qed
```

lemma *LLL-mu-abs-bound*:
assumes *i*: $i < m$
and *j*: $j < i$
shows $|\mu\ fs\ i\ j| \leq rat\text{-}of\text{-}nat\ (N\ \hat{}\ (m - 1) * 2\ \hat{}\ (m - 1) * m)$
proof −
from *bound-invD*[*OF binv*]
have *inv*: *LLL-invariant upw k fs*
and *fbnd*: *f-bound outside k fs*
and *gbnd*: *g-bound fs* **by** *auto*
note *invw* = *LLL-inv-imp-w*[*OF inv*]
from *LLL-inv-N-pos*[*OF invw gbnd*] *i* **have** *N*: $N > 0$ **by** *auto*
note ∗ = *LLL-invD*[*OF inv*]
interpret *fs*: *fs-int′ n m fs-init fs*
by *standard* (*use invw* **in** *auto*)
let *?mu* = $\mu\ fs\ i\ j$
from *j i* **have** *jm*: $j < m$ **by** *auto*
from *d-approx*[*OF invw gbnd jm*]
have *dj*: $d\ fs\ j \leq int\ (N\ \hat{}\ j)$ **by** *linarith*
let *?num* = $4\ \hat{}\ (m - 1) * N\ \hat{}\ m * m * m$
let *?bnd* = $N\hat{}(m - 1) * 2\ \hat{}\ (m - 1) * m$
from *fbnd*[*unfolded f-bound-def*, *rule-format*, *OF i*]
aux-bnd-mono[*folded of-nat-le-iff*[**where** *?′a* = *int*]]
have *sq-f-bnd*: *sq-norm* $(fs\ !\ i) \leq int$ *?num* **by** (*auto split*: *if-splits*)
have *four*: $(4 :: nat) = 2\hat{}2$ **by** *auto*
have *?mu*$\hat{}2 \leq$ (*gs.Gramian-determinant* (*RAT fs*) *j*) ∗ *sq-norm* (*RAT fs* ! *i*)
proof −
have *1*: *of-int-hom.vec-hom* (*fs* ! *j*) $ *i* ∈ $\mathbb{Z}$ **if** $i < n\ j < length\ fs$ **for** *j i*
using ∗ *that* **by** (*metis vec-hom-Ints*)
then show *?thesis*
by (*intro fs.gs.mu-bound-Gramian-determinant*[*OF j*], *insert* ∗ *j i*,
auto simp: *set-conv-nth gs.lin-indpt-list-def*)
qed
also have *sq-norm* (*RAT fs* ! *i*) = *of-int* (*sq-norm* (*fs* ! *i*))
unfolding *sq-norm-of-int*[*symmetric*] **using** ∗(*6*) *i* **by** *auto*
also have (*gs.Gramian-determinant* (*RAT fs*) *j*) = *of-int* (*d fs j*)
unfolding *d-def* **by** (*rule fs.of-int-Gramian-determinant*, *insert i j* ∗(*3*,*6*), *auto simp*: *set-conv-nth*)
also have ... ∗ *of-int* (*sq-norm* (*fs* ! *i*)) = *of-int* (*d fs j* ∗ *sq-norm* (*fs* ! *i*)) **by** *simp*
also have ... ≤ *of-int* (*int* (*N*$\hat{}$*j*) ∗ *int ?num*) **unfolding** *of-int-le-iff*
by (*rule mult-mono*[*OF dj sq-f-bnd*], *auto*)
also have ... = *of-nat* (*N*$\hat{}$(*j* + *m*) ∗ (*4* $\hat{}$ (*m* − *1*) ∗ *m* ∗ *m*)) **by** (*simp add*: *power-add*)
also have ... ≤ *of-nat* (*N*$\hat{}$((*m* − *1*) + (*m*−*1*)) ∗ (*4* $\hat{}$ (*m* − *1*) ∗ *m* ∗ *m*)) **unfolding** *of-nat-le-iff*
by (*rule mult-right-mono*[*OF pow-mono-exp*], *insert N j i jm*, *auto*)
also have ... = *of-nat* (*?bnd*$\hat{}$*2*)
unfolding *four power-mult-distrib power2-eq-square of-nat-mult* **by** (*simp add*: *power-add*)

finally have *?mu^2 ≤ (of-nat ?bnd)^2* **by** *auto*
from *this*[*folded abs-le-square-iff*]
show *abs ?mu ≤ of-nat ?bnd* **by** *auto*
qed

lemma *LLL-dμ-bound*:
assumes *i*: *i < m* **and** *j*: *j < i*
shows *abs (dμ fs i j) ≤ N ^ (2 * (m − 1)) * 2 ^ (m − 1) * m*
proof −
from *bound-invD*[*OF binv*]
have *inv*: *LLL-invariant upw k fs*
and *fbnd*: *f-bound outside k fs*
and *gbnd*: *g-bound fs* **by** *auto*
note *invw* = *LLL-inv-imp-w*[*OF inv*]
interpret *fs*: *fs-int′ n m fs-init fs*
by *standard* (*use invw* **in** *auto*)
from *LLL-inv-N-pos*[*OF invw gbnd*] *i* **have** *N*: *N > 0* **by** *auto*
from *j i* **have** *jm*: *j < m − 1 j < m* **by** *auto*
let *?r* = *rat-of-int*
from *LLL-d-bound*[*of Suc j*] *jm*
have *abs (d fs (Suc j)) ≤ N ^ Suc j* **by** *linarith*
also have *. . . ≤ N ^ (m − 1)* **unfolding** *of-nat-le-iff*
by (*rule pow-mono-exp*, *insert N jm*, *auto*)
finally have *dsj*: *abs (d fs (Suc j)) ≤ int N ^ (m − 1)* **by** *auto*
from *fs.dμ*[*of j i*] *j i LLL-invD*[*OF inv*]
have *?r (abs (dμ fs i j)) = abs (?r (d fs (Suc j)) * μ fs i j)*
unfolding *d-def fs.d-def dμ-def fs.dμ-def* **by** *auto*
also have *. . . = ?r (abs (d fs (Suc j))) * abs (μ fs i j)* **by** (*simp add*: *abs-mult*)
also have *. . . ≤ ?r (int N ^ (m − 1)) * rat-of-nat (N ^ (m − 1) * 2 ^ (m − 1) * m)*
by (*rule mult-mono*[*OF - LLL-mu-abs-bound*[*OF i j*]], *insert dsj*, *linarith*, *auto*)
also have *. . . = ?r (int (N ^ ((m − 1) + (m − 1)) * 2 ^ (m − 1) * m))*
by (*simp add*: *power-add*)
also have *(m − 1) + (m − 1) = 2 * (m − 1)* **by** *simp*
finally show *abs (dμ fs i j) ≤ N ^ (2 * (m − 1)) * 2 ^ (m − 1) * m* **by** *linarith*
qed

lemma *LLL-mu-num-denom-bound*:
assumes *i*: *i < m*
and *quot*: *quotient-of (μ fs i j) = (num, denom)*
shows *|num| ≤ N ^ (2 * m) * 2 ^ m * m*
and *|denom| ≤ N ^ m*
proof (*atomize*(*full*))
from *bound-invD*[*OF binv*]
have *inv*: *LLL-invariant upw k fs*
and *fbnd*: *f-bound outside k fs*
and *gbnd*: *g-bound fs* **by** *auto*

```
  note invw = LLL-inv-imp-w[OF inv]
  from LLL-inv-N-pos[OF invw gbnd] i have N: N > 0 by auto
  note * = LLL-invD[OF inv]
  interpret fs: fs-int' n m fs-init fs
    by standard (use invw in auto)
  let ?mu = μ fs i j
  let ?bnd = N^(m - 1) * 2 ^ (m - 1) * m
  show |num| ≤ N ^ (2 * m) * 2 ^ m * m ∧ |denom| ≤ N ^ m
  proof (cases j < i)
    case j: True
    with i have jm: j < m by auto
    from LLL-d-pos[OF invw, of Suc j] i j have dsj: 0 < d fs (Suc j) by auto
    from quotient-of-square[OF quot]
    have quot-sq: quotient-of (?mu^2) = (num * num, denom * denom)
      unfolding power2-eq-square by auto
    from LLL-mu-abs-bound[OF assms(1) j]
    have mu-bound: abs ?mu ≤ of-nat ?bnd by auto
    have gs.Gramian-determinant (RAT fs) (Suc j) * ?mu ∈ ℤ
      by (rule fs.gs.d-mu-Ints,
     insert j *(1,3-6) i, auto simp: set-conv-nth gs.lin-indpt-list-def vec-hom-Ints)
    also have (gs.Gramian-determinant (RAT fs) (Suc j)) = of-int (d fs (Suc j))
      unfolding d-def by (rule fs.of-int-Gramian-determinant, insert i j *(3,6),
auto simp: set-conv-nth)
    finally have ints: of-int (d fs (Suc j)) * ?mu ∈ ℤ .
    from LLL-d-bound[of Suc j] jm
    have d-j: d fs (Suc j) ≤ N ^ m by auto
    note quot-bounds = quotient-of-bounds[OF quot ints dsj mu-bound]
    have abs denom ≤ denom using quotient-of-denom-pos[OF quot] by auto
    also have ... ≤ d fs (Suc j) by fact
    also have ... ≤ N ^ m by fact
    finally have denom: abs denom ≤ N ^ m by auto
    from quot-bounds(1) have |num| ≤ d fs (Suc j) * int ?bnd
      unfolding of-int-le-iff[symmetric, where ?'a = rat] by simp
    also have ... ≤ N ^ m * int ?bnd by (rule mult-right-mono[OF d-j], auto)
    also have ... = (int N ^ (m + (m - 1))) * (2 ^ (m - 1)) * int m unfolding
power-add of-nat-mult by simp
    also have ... ≤ (int N ^ (2 * m)) * (2 ^ m) * int m unfolding of-nat-mult
      by (intro mult-mono pow-mono-exp, insert N, auto)
    also have ... = int (N ^ (2 * m) * 2 ^ m * m) by simp
    finally have num: |num| ≤ N ^ (2 * m) * 2 ^ m * m .
    from denom num show ?thesis by blast
  next
    case False
    hence ?mu = 0 ∨ ?mu = 1 unfolding fs.gs.μ.simps by auto
    hence quotient-of ?mu = (1,1) ∨ quotient-of ?mu = (0,1) by auto
    from this[unfolded quot] show ?thesis using N i by (auto intro!: mult-ge-one)
  qed
qed
```

Now we have bounds on each number $(f_i)_j$, $(g_i)_j$, and $\mu_{i,j}$, i.e., for rational

numbers bounds on the numerators and denominators.

lemma *logN-le-2log-Mn*: **assumes** *m*: $m \neq 0$ $n \neq 0$ **and** *N*: $N > 0$
shows $log\ 2\ N \leq 2 * log\ 2\ (M * n)$
proof −
have $N \leq nat\ M * nat\ M * n * 1$ **using** *N-le-MMn m* **by** *auto*
also have $\ldots \leq nat\ M * nat\ M * n * n$ **by** (*intro mult-mono*, *insert m*, *auto*)
finally have *NM*: $N \leq nat\ M * nat\ M * n * n$ **by** *simp*
with *N* **have** $nat\ M \neq 0$ **by** *auto*
hence *M*: $M > 0$ **by** *simp*

have $log\ 2\ N \leq log\ 2\ (M * M * n * n)$
proof (*subst log-le-cancel-iff*)
show $real\ N \leq (M * M * int\ n * int\ n)$ **using** *NM*[*folded of-nat-le-iff*[**where** *?'a* = *real*]] *M*
by *simp*
qed (*insert N M m*, *auto*)
also have $\ldots = log\ 2\ (of\text{-}int\ (M * n) * of\text{-}int\ (M * n))$
unfolding *of-int-mult* **by** (*simp add*: *ac-simps*)
also have $\ldots = 2 * log\ 2\ (M * n)$
by (*subst log-mult*, *insert m M*, *auto*)
finally show $log\ 2\ N \leq 2 * log\ 2\ (M * n)$ **by** *auto*
qed

We now prove a combined size-bound for all of these numbers. The bounds clearly indicate that the size of the numbers grows at most polynomial, namely the sizes are roughly bounded by $\mathcal{O}(m \cdot \log(M \cdot n))$ where m is the number of vectors, n is the dimension of the vectors, and M is the maximum absolute value that occurs in the input to the LLL algorithm.

lemma *combined-size-bound*: **fixes** *number* :: *int*
assumes *i*: $i < m$ **and** *j*: $j < n$
and *x*: $x \in \{of\text{-}int\ (fs\ !\ i\ \$\ j), gso\ fs\ i\ \$\ j, \mu\ fs\ i\ j\}$
and *quot*: *quotient-of x* = (*num*, *denom*)
and *number*: *number* $\in \{num, denom\}$
and *number0*: *number* $\neq 0$
shows $log\ 2\ |number| \leq 2 * m * log\ 2\ N + m + log\ 2\ m$
$log\ 2\ |number| \leq 4 * m * log\ 2\ (M * n) + m + log\ 2\ m$
proof −
from *bound-invD*[*OF binv*]
have *inv*: *LLL-invariant upw k fs*
and *fbnd*: *f-bound outside k fs*
and *gbnd*: *g-bound fs*
by *auto*
note *invw* = *LLL-inv-imp-w*[*OF inv*]
from *LLL-inv-N-pos*[*OF invw gbnd*] *i* **have** *N*: $N > 0$ **by** *auto*
let *?bnd* = $N\ \hat{}\ (2 * m) * 2\ \hat{}\ m * m$
have $N\ \hat{}\ m * int\ 1 \leq N\ \hat{}\ (2 * m) * (2\hat{}m * int\ m)$
by (*rule mult-mono*, *unfold of-nat-le-iff*, *rule pow-mono-exp*, *insert N i*, *auto*)
hence *le*: $int\ (N\ \hat{}\ m) \leq N\ \hat{}\ (2 * m) * 2\hat{}m * m$ **by** *auto*

from x **consider** $(xfs)\ x = of\text{-}int\ (fs\ !\ i\ \$\ j) \mid (xgs)\ x = gso\ fs\ i\ \$\ j \mid (xmu)\ x = \mu\ fs\ i\ j$
by *auto*
hence *num-denom-bound*: $|num| \leq ?bnd \wedge |denom| \leq N \mathbin{\hat{}} m$
proof (*cases*)
case *xgs*
from *LLL-gso-bound*[*OF i j quot*[*unfolded xgs*]] *le*
show *?thesis* **by** *auto*
next
case *xmu*
from *LLL-mu-num-denom-bound*[*OF i*, *of j*, *OF quot*[*unfolded xmu*]]
show *?thesis* **by** *auto*
next
case *xfs*
have $|denom| = 1$ **using** *quot*[*unfolded xfs*] **by** *auto*
also have $\ldots \leq N \mathbin{\hat{}} m$ **using** N **by** *auto*
finally have *denom*: $|denom| \leq N \mathbin{\hat{}} m$ **.**
have $|num| = |fs\ !\ i\ \$\ j|$ **using** *quot*[*unfolded xfs*] **by** *auto*
also have $\ldots \leq int\ (N \mathbin{\hat{}} m * 2 \mathbin{\hat{}} (m - 1) * m)$ **using** *LLL-f-bound*[*OF i j*] **by** *auto*
also have $\ldots \leq ?bnd$ **unfolding** *of-nat-mult of-nat-power*
using N **by** (*auto intro*!: *mult-mono pow-mono-exp*)
finally show *?thesis* **using** *denom* **by** *auto*
qed
from *number* **consider** (*num*) *number* = *num* | (*denom*) *number* = *denom* **by** *auto*
hence *number-bound*: $|number| \leq ?bnd$
proof (*cases*)
case *num*
with *num-denom-bound* **show** *?thesis* **by** *auto*
next
case *denom*
with *num-denom-bound* **have** $|number| \leq N \mathbin{\hat{}} m$ **by** *auto*
with *le* **show** *?thesis* **by** *auto*
qed
from *number-bound* **have** *bnd*: *of-int* $|number| \leq$ *real ?bnd* **by** *linarith*
have *log 2* $|number| \leq$ *log 2 ?bnd*
by (*subst log-le-cancel-iff*, *insert number0 bnd*, *auto*)
also have $\ldots$ = *log 2* $(N \hat{} (2 * m))$ + *log 2* $(2 \hat{} m)$ + *log 2 m*
using *i N* **by** (*simp add*: *log-mult*)
also have *log 2* $(N \hat{} (2 * m))$ = *log 2* $(N\ powr\ (2 * m))$
by (*rule arg-cong*[*of - - log 2*], *subst powr-realpow*, *insert N*, *auto*)
also have $\ldots$ = $(2 * m) *$ *log 2 N*
by (*subst log-powr*, *insert N*, *auto*)
finally show *boundN*: *log 2* $|number| \leq 2 * m *$ *log 2 N* + *m* + *log 2 m* **by** *simp*
also have $\ldots \leq 2 * m * (2 *$ *log 2* $(M * n))$ + *m* + *log 2 m*
by (*intro add-right-mono mult-mono logN-le-2log-Mn N*, *insert i j N*, *auto*)
finally show *log 2* $|number| \leq 4 * m *$ *log 2* $(M * n)$ + *m* + *log 2 m* **by** *simp*

qed

And a combined size bound for an integer implementation which stores values f_i, $d_{j+1}\mu_{ij}$ and d_i.

interpretation *fs*: *fs-int-indpt n fs-init*
by (*standard*) (*use lin-dep* **in** *auto*)

lemma *fs-gs-N-N'*: **assumes** $m \neq 0$
shows *fs.gs.N* = *of-nat N*
proof −
have *0*: *Max* (*sq-norm* ‘ *set fs-init*) ∈ *sq-norm* ‘ *set fs-init*
using *len assms* **by** *auto*
then have *1*: *nat* (*Max* (*sq-norm* ‘ *set fs-init*)) ∈ (*nat* ∘ *sq-norm*) ‘ *set fs-init*
by (*auto*)
have [*simp*]: *0* ≤ *Max* (*sq-norm* ‘ *set fs-init*)
using *0* **by** *force*
have [*simp*]: *sq-norm* ‘ *of-int-hom.vec-hom* ‘ *set fs-init* = *rat-of-int* ‘ *sq-norm* ‘ *set fs-init*
by (*auto simp add*: *sq-norm-of-int image-iff*)
then have [*simp*]: *rat-of-int* (*Max* (*sq-norm* ‘ *set fs-init*)) ∈ *rat-of-int* ‘ *sq-norm* ‘ *set fs-init*
using *0* **by** *auto*
have (*Missing-Lemmas.max-list* (*map* (*nat* ∘ *sq-norm*) *fs-init*)) = *Max* ((*nat* ∘ *sq-norm*) ‘ *set fs-init*)
using *assms len* **by** (*subst max-list-Max*) (*auto*)
also have ... = *nat* (*Max* (*sq-norm-vec* ‘ *set fs-init*))
using *assms 1* **by** (*auto intro*!: *nat-mono Max-eqI*)
also have *int* ... = *Max* (*sq-norm-vec* ‘ *set fs-init*)
by (*subst int-nat-eq*) (*auto*)
also have *rat-of-int* ... = *Max* (*sq-norm* ‘ *set* (*map of-int-hom.vec-hom fs-init*))
by (*rule Max-eqI*[*symmetric*]) (*auto simp add*: *sq-norm-of-int*)
finally show *?thesis*
unfolding *N-def fs.gs.N-def* **by** (*auto*)
qed

lemma *fs-gs-N-N*: $m \neq 0 \Longrightarrow$ *real-of-rat fs.gs.N* = *real N*
using *fs-gs-N-N'* **by** *simp*

lemma *combined-size-bound-gso-integer*:
assumes $x \in$
$\{fs.\mu'\ i\ j\ |i\ j.\ j \leq i \wedge i < m\} \cup$
$\{fs.\sigma s\ l\ i\ j\ |i\ j\ l.\ i < m \wedge j \leq i \wedge l < j\}$
and *m*: $m \neq 0$ **and** $x \neq 0$ $n \neq 0$
shows *log 2* |*real-of-int x*| ≤ (*6* + *6* ∗ *m*) ∗ *log 2* (*M* ∗ *n*) + *log 2 m* + *m*
proof −
from *bound-invD*[*OF binv*]
have *inv*: *LLL-invariant upw k fs*
and *gbnd*: *g-bound fs*
by *auto*

```
  note invw = LLL-inv-imp-w[OF inv]
  from LLL-inv-N-pos[OF invw gbnd m] have N: N > 0 by auto
  have log 2 |real-of-int x| ≤ log 2 m + real (3 + 3 * m) * log 2 N
   using assms len fs.combined-size-bound-integer-log by (auto simp add: fs-gs-N-N)
  also have ... ≤ log 2 m + (3 + 3 * m) * (2 * log 2 (M * n))
    using logN-le-2log-Mn assms N by (intro add-left-mono, intro mult-left-mono)
(auto)
  also have ... = log 2 m + (6 + 6 * m) * log 2 (M * n)
    by (auto simp add: algebra-simps)
  finally show ?thesis
    by auto
qed

lemma combined-size-bound-integer':
  assumes x: x ∈ {fs ! i $ j | i j. i < m ∧ j < n}
    ∪ {dμ fs i j | i j. j < i ∧ i < m}
    ∪ {d fs i | i. i ≤ m}
    (is x ∈ ?fs ∪ ?dμ ∪ ?d)
  and m: m ≠ 0 and n: n ≠ 0
shows abs x ≤ N ^ (2 * m) * 2 ^ m * m
  x ≠ 0 ⟹ log 2 |x| ≤ 2 * m * log 2 N       + m + log 2 m (is - ⟹ ?l1 ≤ ?b1)
  x ≠ 0 ⟹ log 2 |x| ≤ 4 * m * log 2 (M * n) + m + log 2 m (is - ⟹ - ≤ ?b2)
proof −
  let ?bnd = int N ^ (2 * m) * 2 ^ m * int m
  from bound-invD[OF binv]
  have inv: LLL-invariant upw k fs
    and fbnd: f-bound outside k fs
    and gbnd: g-bound fs
    by auto
  note invw = LLL-inv-imp-w[OF inv]
  from LLL-inv-N-pos[OF invw gbnd m] have N: N > 0 by auto
  let ?r = real-of-int
  from x consider (fs) x ∈ ?fs | (dμ) x ∈ ?dμ | (d) x ∈ ?d by auto
  hence abs x ≤ ?bnd
  proof cases
    case fs
    then obtain i j where i: i < m and j: j < n and x: x = fs ! i $ j by auto
    from LLL-f-bound[OF i j, folded x]
    have |x| ≤ int N ^ m * 2 ^ (m − 1) * int m by simp
    also have ... ≤ ?bnd
      by (intro mult-mono pow-mono-exp, insert N, auto)
    finally show ?thesis .
  next
    case dμ
    then obtain i j where i: i < m and j: j < i and x: x = dμ fs i j by auto
    from LLL-dμ-bound[OF i j, folded x]
    have |x| ≤ int N ^ (2 * (m − 1)) * 2 ^ (m − 1) * int m by simp
    also have ... ≤ ?bnd
      by (intro mult-mono pow-mono-exp, insert N, auto)
```

```
    finally show ?thesis .
  next
    case d
    then obtain i where i: i ≤ m and x: x = d fs i by auto
    from LLL-d-bound[OF i, folded x]
    have |x| ≤ int N ^ m * 2 ^ 0 * 1 by simp
    also have ... ≤ ?bnd
      by (intro mult-mono pow-mono-exp, insert N m, auto)
    finally show ?thesis .
  qed
  thus abs x ≤ N ^ (2 * m) * 2 ^ m * m by simp
  hence abs: ?r (abs x) ≤ ?r (N ^ (2 * m) * 2 ^ m * m) by linarith
  assume x ≠ 0 hence x: abs x > 0 by auto
  from abs have log 2 (abs x) ≤ log 2 (?r (N ^ (2 * m)) * 2 ^ m * ?r m)
    by (subst log-le-cancel-iff, insert x N m, auto)
  also have ... = log 2 (?r N ^ (2 * m)) + m + log 2 (?r m)
    using N m by (auto simp: log-mult)
  also have log 2 (?r N ^ (2 * m)) = real (2 * m) * log 2 (?r N)
    by (subst log-nat-power, insert N, auto)
  finally show ?l1 ≤ ?b1 by simp
  also have ... ≤ 2 * m * (2 * log 2 (M * n)) + m + log 2 m
    by (intro add-right-mono mult-left-mono logN-le-2log-Mn, insert m n N, auto)
  finally show ?l1 ≤ ?b2 by simp
qed

lemma combined-size-bound-integer:
  assumes x: x ∈
      {fs ! i $ j | i j. i < m ∧ j < n}
    ∪ {dμ fs i j | i j. j < i ∧ i < m}
    ∪ {d fs i | i. i ≤ m}
    ∪ {fs.μ' i j |i j. j ≤ i ∧ i < m}
    ∪ {fs.σs l i j |i j l. i < m ∧ j ≤ i ∧ l < j}
    (is ?x ∈ ?s1 ∪ ?s2 ∪ ?s3 ∪ ?g1 ∪ ?g2)
  and m: m ≠ 0 and n: n ≠ 0 and x ≠ 0 and 0 < M
shows log 2 |x| ≤ (6 + 6 * m) * log 2 (M * n) + log 2 m + m
proof −
  show ?thesis
  proof (cases ?x ∈ ?g1 ∪ ?g2)
    case True
    then show ?thesis
      using combined-size-bound-gso-integer assms by simp
  next
    case False
    then have x: x ∈ ?s1 ∪ ?s2 ∪ ?s3 using x by auto
    from combined-size-bound-integer'(3)[OF this m n ‹x ≠ 0›]
    have log 2 |x| ≤ 4 * m * log 2 (M * n) + m + log 2 m by simp
    also have ... ≤ (6 + 6 * m) * log 2 (M * n) + m + log 2 m
      using assms by (intro add-right-mono, intro mult-right-mono) auto
    finally show ?thesis
```

by *simp*
qed
qed

end
end
end

10 Certification of External LLL Invocations

Instead of using a fully verified algorithm, we also provide a technique to invoke an external LLL solver. In order to check its result, we not only need the reduced basic, but also the matrices which translate between the input basis and the reduced basis. Then we can easily check whether the resulting lattices are indeed identical and just have to start the verified algorithm on the already reduced basis. This invocation will then usually just require one computation of Gram–Schmidt in order to check that the basis is already reduced. Alternatively, one could also throw an error message in case the basis is not reduced.

10.1 Checking Results of External LLL Solvers

theory *LLL-Certification*
imports
LLL-Impl
Jordan-Normal-Form.Show-Matrix
begin

definition *gauss-jordan-integer-inverse n A B I* = (*case gauss-jordan A B of*
(*C*,*D*) ⇒ *C* = *I* ∧ *list-all is-int-rat* (*concat* (*mat-to-list D*)))

definition *integer-equivalent n fs gs* = (*let*
fs′ = *map-mat rat-of-int* (*mat-of-cols n fs*);
gs′ = *map-mat rat-of-int* (*mat-of-cols n gs*);
I = 1_m *n*
in gauss-jordan-integer-inverse n fs′ *gs*′ *I* ∧ *gauss-jordan-integer-inverse n gs*′ *fs*′
I)

context *vec-module*
begin

lemma *mat-mult-sub-lattice*: **assumes** *fs*: *set fs* ⊆ *carrier-vec n*
and *gs*: *set gs* ⊆ *carrier-vec n*
and *A*: *A* ∈ *carrier-mat* (*length fs*) (*length gs*)
and *prod*: *mat-of-rows n fs* = *map-mat of-int A* ∗ *mat-of-rows n gs*
shows *lattice-of fs* ⊆ *lattice-of gs*

proof
let *?m* = *length fs*
let *?m′* = *length gs*
let *?i* = *of-int* :: *int* ⇒ *'a*
let *?I* = *map-mat ?i*
let *?A* = *?I A*
have *gsC*: *mat-of-rows n gs* ∈ *carrier-mat ?m′ n* **by** *auto*
from *A* **have** *A*: *?A* ∈ *carrier-mat ?m ?m′* **by** *auto*
from *fs* **have** *fsi*[*simp*]: ⋀ *i*. *i* < *?m* ⟹ *fs* ! *i* ∈ *carrier-vec n* **by** *auto*
hence *fsi′*[*simp*]: ⋀ *i*. *i* < *?m* ⟹ *dim-vec* (*fs* ! *i*) = *n* **by** *simp*
from *gs* **have** *fsi*[*simp*]: ⋀ *i*. *i* < *?m′* ⟹ *gs* ! *i* ∈ *carrier-vec n* **by** *auto*
hence *gsi′*[*simp*]: ⋀ *i*. *i* < *?m′* ⟹ *dim-vec* (*gs* ! *i*) = *n* **by** *simp*
fix *v*
assume *v* ∈ *lattice-of fs*
from *in-latticeE*[*OF this*]
obtain *c* **where** *v*: *v* = *M.sumlist* (*map* (λ*i*. *?i* (*c i*) ·$_v$ *fs* ! *i*) [*0..<?m*]) **by** *auto*
let *?c* = *vec ?m* (λ *i*. *?i* (*c i*))
let *?d* = A^T ∗$_v$ *vec ?m c*
note *v*
also have … = *mat-of-cols n fs* ∗$_v$ *?c*
by (*rule eq-vecI*, *auto intro*!: *dim-sumlist sum.cong*
simp: *sumlist-nth scalar-prod-def mat-of-cols-def*)
also have *mat-of-cols n fs* = (*mat-of-rows n fs*)^T
by (*simp add*: *transpose-mat-of-rows*)
also have … = (*?A* ∗ *mat-of-rows n gs*)^T **unfolding** *prod* **..**
also have … = (*mat-of-rows n gs*)^T ∗ *?A*^{T}
by (*rule transpose-mult*[*OF A gsC*])
also have (*mat-of-rows n gs*)^T = *mat-of-cols n gs*
by (*simp add*: *transpose-mat-of-rows*)
finally have *v* = (*mat-of-cols n gs* ∗ *?A*^{T}) ∗$_v$ *?c* **.**
also have … = *mat-of-cols n gs* ∗$_v$ (*?A*^{T} ∗$_v$ *?c*)
by (*rule assoc-mult-mat-vec*, *insert A*, *auto*)
also have *?A*^{T} = *?I* (A^T) **by** *fastforce*
also have *?c* = *map-vec ?i* (*vec ?m c*) **by** *auto*
also have *?I* (A^T) ∗$_v$ … = *map-vec ?i ?d*
using *A* **by** (*simp add*: *of-int-hom.mult-mat-vec-hom*)
finally have *v* = *mat-of-cols n gs* ∗$_v$ *map-vec ?i ?d* **.**
define *d* **where** *d* = *?d*
have *d*: *d* ∈ *carrier-vec ?m′* **unfolding** *d-def* **using** *A* **by** *auto*
have *v* = *mat-of-cols n gs* ∗$_v$ *map-vec ?i d* **unfolding** *d-def* **by** *fact*
also have … = *M.sumlist* (*map* (λ*i*. *?i* (*d* $ *i*) ·$_v$ *gs* ! *i*) [*0..<?m′*])
by (*rule sym*, *rule eq-vecI*, *insert d*, *auto intro*!: *dim-sumlist sum.cong*
simp: *sumlist-nth scalar-prod-def mat-of-cols-def*)
finally show *v* ∈ *lattice-of gs*
by (*intro in-latticeI*, *auto*)
qed
end

context *LLL-with-assms*

begin

lemma *mult-left-identity*:
defines $B \equiv$ (*map-mat rat-of-int* (*mat-of-rows n fs-init*))
assumes *P-carrier*[*simp*]: $P \in$ *carrier-mat m m*
and *PB*: $P * B = B$
shows $P = 1_m\ m$
proof −
let *?set-rows* = *set* (*rows B*)
let *?hom* = *of-int-hom.vec-hom* :: *int vec* ⇒ *rat vec*
have *set-rows-carrier*: *?set-rows* ⊆ (*carrier-vec n*) **by** (*auto simp add*: *rows-def B-def*)
have *set-rows-eq*: *?set-rows* = *set* (*map of-int-hom.vec-hom fs-init*)
proof −
have $x \in$ *of-int-hom.vec-hom* ‘ *set fs-init* **if** *x*: $x \in$ *set* (*rows B*) **for** *x*
using *x* **unfolding** *B-def*
by (*metis cof-vec-space.lin-indpt-list-def fs-init image-set lin-dep mat-of-rows-map rows-mat-of-rows*)
moreover have *of-int-hom.vec-hom xa* ∈ *set* (*rows B*) **if** *xa*: *xa* ∈ *set fs-init* **for** *xa*
proof −
obtain *i* **where** *xa*: *xa* = *fs-init* ! *i* **and** *i*: *i*<*m*
by (*metis in-set-conv-nth len xa*)
have *?hom* (*fs-init* ! *i*) = *row B i* **unfolding** *B-def*
by (*metis i cof-vec-space.lin-indpt-list-def fs-init index-map-mat*(*2*) *len lin-dep mat-of-rows-carrier*(*2*) *mat-of-rows-map nth-map nth-rows rows-mat-of-rows*)
thus *?thesis*
by (*metis B-def xa i cof-vec-space.lin-indpt-list-def fs-init index-map-mat*(*2*) *len length-rows lin-dep mat-of-rows-map nth-map nth-mem rows-mat-of-rows*)
qed
ultimately show *?thesis* **by** *auto*
qed
have *ind-set-rows*: *gs.lin-indpt ?set-rows*
using *lin-dep set-rows-eq* **unfolding** *gs.lin-indpt-list-def* **by** *auto*
have *inj-on-rowB*: *inj-on* (*row B*) {*0*..<*m*}
proof −
have *x* = *y* **if** *x*: *x* < *m* **and** *y*: *y* < *m* **and** *row-xy*: *row B x* = *row B y* **for** *x y*
proof (*rule ccontr*)
assume *xy*: *x* ≠ *y*
have *1*: *?hom* (*fs-init* ! *x*) = *row B x* **unfolding** *B-def*
by (*metis fs-init index-map-mat*(*2*) *len local.set-rows-carrier mat-of-rows-carrier*(*2*) *mat-of-rows-map nth-map nth-rows rows-mat-of-rows set-rows-eq that*(*1*))
moreover have *2*: *?hom* (*fs-init* ! *y*) = *row B y* **unfolding** *B-def*
by (*metis fs-init index-map-mat*(*2*) *len local.set-rows-carrier mat-of-rows-carrier*(*2*) *mat-of-rows-map nth-map nth-rows rows-mat-of-rows set-rows-eq that*(*2*))

ultimately have *?hom* (*fs-init* ! *x*) = *?hom* (*fs-init* ! *y*) **using** *row-xy* **by** *auto*
thus *False* **using** *lin-dep x y row-xy* **unfolding** *gs.lin-indpt-list-def*
using *xy x y len* **unfolding** *distinct-conv-nth* **by** *auto*
qed
thus *?thesis* **unfolding** *inj-on-def* **by** *auto*
qed
have *the-x*: (*THE k*. *k* < *m* ∧ *row B x* = *row B k*) = *x* **if** *x*: *x* < *m* **for** *x*
proof (*rule theI2*)
show *x* < *m* ∧ *row B x* = *row B x* **using** *x* **by** *auto*
fix *xa* **assume** *xa*: *xa* < *m* ∧ *row B x* = *row B xa*
show *xa* = *x* **using** *xa inj-on-rowB x* **unfolding** *inj-on-def* **by** *auto*
thus *xa* = *x* **.**
qed
let *?h*= *row B*
show *?thesis*
proof (*rule eq-matI*, *unfold one-mat-def*, *auto*)
fix *j* **assume** *j*: *j* < *m*
let *?f* = (λ*v*. *P* $$ (*j*, *THE k*. *k* < *m* ∧ *v* = *row B k*))
let *?g* = λ*v*. *if v* = *row B j then* (*?f v*) − *1 else ?f v*
have *finsum-closed*[*simp*]:
finsum-vec TYPE(*rat*) *n* (λ*k*. *P* $$ (*j*, *k*) ·$_v$ *row B k*) {*0*..<*m*} ∈ *carrier-vec n*
by (*rule finsum-vec-closed*, *insert len B-def*, *auto*)
have *B-carrier*[*simp*]: *B* ∈ *carrier-mat m n* **using** *len fs-init B-def* **by** *auto*
define *v* **where** *v* ≡ *row B j*
have *v-set-rows*: *v* ∈ *set* (*rows B*) **using** *nth-rows j* **unfolding** *v-def*
by (*metis B-carrier carrier-matD*(*1*) *length-rows nth-mem*)
have [*simp*]: *mat-of-rows n fs-init* ∈ *carrier-mat m n* **using** *len fs-init* **by** *auto*
have *B* = *P*∗*B* **using** *PB* **by** *auto*
also have ... = *mat*$_r$ *m n* (λ*i*. *finsum-vec TYPE*(*rat*) *n* (λ*k*. *P* $$ (*i*, *k*) ·$_v$ *row B k*) {*0*..<*m*})
by (*rule mat-mul-finsum-alt*, *auto*)
also have *row* (...) *j* = *finsum-vec TYPE*(*rat*) *n* (λ*k*. *P* $$ (*j*, *k*) ·$_v$ *row B k*) {*0*..<*m*}
by (*rule row-mat-of-row-fun*[*OF j*], *simp*)
also have ... = *finsum-vec TYPE*(*rat*) *n* (λ*v*. *?f v* ·$_v$ *v*) *?set-rows* (**is** *?lhs* = *?rhs*)
proof (*rule eq-vecI*)
have *rhs-carrier*: *?rhs* ∈ *carrier-vec n*
by (*rule finsum-vec-closed*, *insert set-rows-carrier*, *auto*)
have *dim-vec ?lhs* = *n* **using** *vec-space.finsum-dim* **by** *simp*
also have *dim-rhs*: ... = *dim-vec ?rhs* **using** *rhs-carrier* **by** *auto*
finally show *dim-vec ?lhs* = *dim-vec ?rhs* **.**
fix *i* **assume** *i*: *i* < *dim-vec ?rhs*
have *i-n*: *i* < *n* **using** *i dim-rhs* **by** *auto*
let *?g* = λ*v*. (*?f v* ·$_v$ *v*) $ *i*
have *image-h*: *?h* ‘{*0*..<*m*} = *?set-rows* **by** (*auto simp add*: *B-def len rows-def*)

```
    have ?lhs $ i = (∑ k∈{0..<m}. (P $$ (j, k) ·v row B k) $ i)
      by (rule index-finsum-vec[OF - i-n], auto)
    also have ... = sum (?g ∘ ?h) {0..<m} unfolding o-def
      by (rule sum.cong, insert the-x, auto)
    also have ... = sum (λv. (?f v ·v v) $ i) (?h ‘{0..<m})
      by (rule sum.reindex[symmetric, OF inj-on-rowB])
    also have ... =  (∑ v∈?set-rows. (?f v ·v v) $ i) using image-h by auto
    also have ... = ?rhs $ i
     by (rule index-finsum-vec[symmetric, OF - i-n], insert set-rows-carrier, auto)
    finally show ?lhs $ i = ?rhs $ i by auto
  qed
  also have ... = (⨁ gs.V v∈?set-rows. ?f v ·v v) unfolding vec-space.finsum-vec
..
  also have ... = gs.lincomb ?f ?set-rows unfolding gs.lincomb-def by auto
  finally have lincomb-rowBj: gs.lincomb ?f ?set-rows = row B j ..
  have lincomb-0: gs.lincomb ?g (?set-rows) = 0v n
  proof −
    have v-closed[simp]: v ∈ Rn unfolding v-def using j by auto
    have lincomb-f-closed[simp]: gs.lincomb ?f (?set-rows−{v}) ∈ Rn
      by (rule gs.lincomb-closed, insert set-rows-carrier, auto)
    have fv-v-closed[simp]: ?f v ·v v ∈ Rn by auto
   have lincomb-f: gs.lincomb ?f ?set-rows = ?f v ·v v + gs.lincomb ?f (?set-rows−{v})
      by (rule gs.lincomb-del2, insert set-rows-carrier v-set-rows, auto)
    have fvv-gvv: ?f v ·v v − v = ?g v ·v v unfolding v-def
      by (rule eq-vecI, auto, simp add: left-diff-distrib)
   have lincomb-fg: gs.lincomb ?f (?set-rows−{v}) = gs.lincomb ?g (?set-rows−{v})

      (is ?lhs = ?rhs)
    proof (rule eq-vecI)
      show dim-vec-eq: dim-vec ?lhs = dim-vec ?rhs
      by (smt DiffE carrier-vecD gs.lincomb-closed local.set-rows-carrier subsetCE
subsetI)
      fix i assume i: i<dim-vec ?rhs
      hence i-n: i<n using dim-vec-eq lincomb-f-closed by auto
      have ?lhs $ i =  (∑ x∈(?set-rows−{v}). ?f x ∗ x $ i)
        by (rule gs.lincomb-index[OF i-n], insert set-rows-carrier, auto)
      also have ... = (∑ x∈(?set-rows−{v}). ?g x ∗ x $ i)
        by (rule sum.cong, auto simp add: v-def)
      also have ... = ?rhs $ i
       by (rule gs.lincomb-index[symmetric, OF i-n], insert set-rows-carrier, auto)
      finally show ?lhs $ i = ?rhs $ i .
    qed
     have 0v n = gs.lincomb ?f ?set-rows − v using lincomb-rowBj unfolding
v-def B-def by auto
   also have ... = ?f v ·v v + gs.lincomb ?f (?set-rows−{v}) − v using lincomb-f
by auto
    also have ... = (gs.lincomb ?f (?set-rows−{v}) + ?f v ·v v) + − v
      unfolding gs.M.a-comm[OF lincomb-f-closed fv-v-closed] by auto
    also have ... = gs.lincomb ?f (?set-rows−{v}) + (?f v ·v v + − v)
```

```
    by (rule gs.M.a-assoc, auto)
  also have ... = gs.lincomb ?f (?set-rows−{v}) + (?f v ·v v − v) by auto
  also have ... = gs.lincomb ?g (?set-rows−{v}) + (?g v ·v v)
    unfolding lincomb-fg fvv-gvv by auto
  also have ... = (?g v ·v v) + gs.lincomb ?g (?set-rows−{v})
    by (rule gs.M.a-comm, auto, rule gs.lincomb-closed, insert set-rows-carrier,
auto)
  also have ... = gs.lincomb ?g (?set-rows)
   by (rule gs.lincomb-del2[symmetric], insert v-set-rows set-rows-carrier, auto)
  finally show ?thesis ..
 qed
 have g0: ?g ∈ ?set-rows → {0}
  by (rule gs.not-lindepD[of ?set-rows, OF ind-set-rows - - - lincomb-0], auto)
 hence ?g (row B j) = 0 using v-set-rows unfolding v-def Pi-def by blast
 hence ?f (row B j) − 1 = 0 by auto
 hence P $$ (j,j) − 1 = 0 using the-x j by auto
 thus P$$(j,j) = 1 by auto
 fix i assume i: i < m and ji: j ≠ i
 have row-ij: row B i ≠ row B j using inj-on-rowB ji i j unfolding inj-on-def
by fastforce
 have row B i ∈ ?set-rows using nth-rows i
  by (metis B-carrier carrier-matD(1) length-rows nth-mem)
 hence ?g (row B i) = 0 using g0 unfolding Pi-def by blast
 hence ?f (row B i) = 0 using row-ij by auto
 thus P $$ (j, i) = 0 using the-x i by auto
next
  show dim-row P = m and dim-col P = m using P-carrier unfolding car-
rier-mat-def by auto
qed
qed
```

This is the key lemma. It permits to change from the initial basis *fs-init* to an arbitrary *gs* that has been computed by some external tool. Here, two change-of-basis matrices U1 and U2 are required to certify the change via the conditions prod1 and prod2.

```
lemma LLL-change-basis: assumes gs: set gs ⊆ carrier-vec n
 and len′: length gs = m
 and U1: U1 ∈ carrier-mat m m
 and U2: U2 ∈ carrier-mat m m
 and prod1: mat-of-rows n fs-init = U1 ∗ mat-of-rows n gs
 and prod2: mat-of-rows n gs = U2 ∗ mat-of-rows n fs-init
shows lattice-of gs = lattice-of fs-init LLL-with-assms n m gs α
proof −
 let ?i = of-int :: int ⇒ int
 have U1 = map-mat ?i U1 by (intro eq-matI, auto)
 with prod1 have prod1: mat-of-rows n fs-init = map-mat ?i U1 ∗ mat-of-rows
n gs by simp
 have U2 = map-mat ?i U2 by (intro eq-matI, auto)
 with prod2 have prod2: mat-of-rows n gs = map-mat ?i U2 ∗ mat-of-rows n
```

fs-init **by** *simp*
have *lattice-of gs* ⊆ *lattice-of fs-init*
by (*rule mat-mult-sub-lattice*[*OF gs fs-init - prod2*], *auto simp*: *U2 len len'*)
moreover have *lattice-of gs* ⊇ *lattice-of fs-init*
by (*rule mat-mult-sub-lattice*[*OF fs-init gs - prod1*], *auto simp*: *U1 len len'*)
ultimately show *lattice-of gs* = *lattice-of fs-init* **by** *blast*
show *LLL-with-assms n m gs* α
proof
show *4/3* ≤ α **by** (*rule* α)
show *length gs* = *m* **by** *fact*
show *lin-indep gs*
proof −
let *?fs* = *map-mat rat-of-int* (*mat-of-rows n fs-init*)
let *?gs* = *map-mat rat-of-int* (*mat-of-rows n gs*)
let *?U1* = *map-mat rat-of-int U1*
let *?U2* = *map-mat rat-of-int U2*
let *?P* = *?U1* ∗ *?U2*
have *rows-gs-eq*: *rows ?gs* = *map of-int-hom.vec-hom gs*
proof (*rule nth-equalityI*)
fix *i* **assume** *i*: *i* < *length* (*rows ?gs*)
have *rows ?gs* ! *i* = *row ?gs i* **by** (*rule nth-rows*, *insert i*, *auto*)
also have ... = *of-int-hom.vec-hom* (*gs* ! *i*)
by (*metis* (*mono-tags*, *lifting*) *gs i index-map-mat*(*2*) *length-map length-rows map-carrier-vec*
mat-of-rows-map mat-of-rows-row nth-map nth-mem rows-mat-of-rows subset-code(*1*))
also have ... = *map of-int-hom.vec-hom gs* ! *i*
by (*rule nth-map*[*symmetric*], *insert i*, *auto*)
finally show *rows ?gs* ! *i* = *map of-int-hom.vec-hom gs* ! *i* **.**
qed (*simp*)
have *fs-hom*: *?fs* ∈ *carrier-mat m n* **unfolding** *carrier-mat-def* **using** *len* **by** *auto*
have *gs-hom*: *?gs* ∈ *carrier-mat m n* **unfolding** *carrier-mat-def* **using** *len'* **by** *auto*
have *U1U2*: *U1* ∗ *U2* ∈ *carrier-mat m m* **by** (*meson assms*(*3*) *assms*(*4*) *mult-carrier-mat*)
have *U1-hom*: *?U1* ∈ *carrier-mat m m* **by** (*simp add*: *U1*)
have *U2-hom*: *?U2* ∈ *carrier-mat m m* **by** (*simp add*: *U2*)
have *U1U2-hom*: *?U1* ∗ *?U2* ∈ *carrier-mat m m* **using** *U1 U2* **by** *auto*
have *Gs-U2Fs*: *?gs* = *?U2* ∗ *?fs* **using** *prod2*
by (*metis U2 assms*(*6*) *len mat-of-rows-carrier*(*1*) *of-int-hom.mat-hom-mult*)
have *fs-hom-eq*: *?fs* = *?P* ∗ *?fs*
by (*smt U1 U1U2 U2 assms*(*5*) *assms*(*6*) *assoc-mult-mat fs-hom*
map-carrier-mat of-int-hom.mat-hom-mult)
have *P-id*: *?P* = 1_m *m* **by** (*rule mult-left-identity*[*OF U1U2-hom fs-hom-eq*[*symmetric*]])
hence *det* (*?U1*) ∗ *det* (*?U2*) = *1* **by** (*smt U1-hom U2-hom det-mult det-one of-int-hom.hom-det*)
hence *det-U2*: *det ?U2* ≠ *0* **and** *det-U1*: *det ?U1* ≠ *0* **by** *auto*
from *det-non-zero-imp-unit*[*OF U2-hom det-U2*, *unfolded Units-def*, *of* ()]

```
    have inv-U2: invertible-mat ?U2
      using U1-hom U2-hom
      unfolding invertible-mat-def inverts-mat-def by (auto simp: ring-mat-def)
    interpret Rs: vectorspace class-ring (gs.vs (gs.row-space ?gs))
      by (rule gs.vector-space-row-space[OF gs-hom])
    interpret RS-fs: vectorspace class-ring (gs.vs (gs.row-space (?fs)))
      by (rule gs.vector-space-row-space[OF fs-hom])
    have submoduleGS: submodule class-ring (gs.row-space ?gs) gs.V
      and submoduleFS: submodule class-ring (gs.row-space ?fs) gs.V
      by (metis gs.row-space-def gs.span-is-submodule index-map-mat(3)
          mat-of-rows-carrier(3) rows-carrier)+
    have set-rows-fs-in: set (rows ?fs) ⊆ gs.row-space ?fs
      and rows-gs-row-space: set (rows ?gs) ⊆ gs.row-space ?gs
      unfolding gs.row-space-def
    by (metis gs.in-own-span index-map-mat(3) mat-of-rows-carrier(3) rows-carrier)+
    have Rs-fs-dim: RS-fs.dim = m
    proof −
      have RS-fs.dim = card (set (rows ?fs))
      proof (rule RS-fs.dim-basis)
        have RS-fs.span (set (rows ?fs)) = gs.span (set (rows ?fs))
       by (rule gs.span-li-not-depend[OF - submoduleFS], simp add: set-rows-fs-in)
        also have ... = carrier (gs.vs (gs.row-space ?fs))
         unfolding gs.row-space-def unfolding gs.carrier-vs-is-self by auto
        finally have RS-fs.gen-set (set (rows ?fs)) by auto
        moreover have RS-fs.lin-indpt (set (rows ?fs))
        proof −
         have module.lin-dep class-ring (gs.vs (gs.row-space ?fs)) (set (rows ?fs))
            = gs.lin-dep (set (rows ?fs))
        by (rule gs.span-li-not-depend[OF - submoduleFS], simp add: set-rows-fs-in)

          thus ?thesis using lin-dep unfolding gs.lin-indpt-list-def
            by (metis fs-init mat-of-rows-map rows-mat-of-rows)
        qed
        moreover have set (rows ?fs) ⊆ carrier (gs.vs (gs.row-space ?fs))
          by (simp add: set-rows-fs-in)
         ultimately show RS-fs.basis (set (rows ?fs)) unfolding RS-fs.basis-def
by simp
      qed (simp)
      also have ... = m
        by (metis cof-vec-space.lin-indpt-list-def distinct-card fs-init len
            length-map lin-dep mat-of-rows-map rows-mat-of-rows)
      finally show ?thesis .
    qed
    have gs.row-space ?fs = gs.row-space (?U2∗?fs)
      by (rule gs.row-space-is-preserved[symmetric, OF inv-U2 U2-hom fs-hom])
    also have ... = gs.row-space ?gs using Gs-U2Fs by auto
    finally have gs.row-space ?fs = gs.row-space ?gs by auto
    hence vectorspace.dim class-ring (gs.vs (gs.row-space ?gs)) = m
      using Rs-fs-dim fs-hom-eq by auto
```

hence *Rs-dim-is-m*: *Rs.dim* = *m* **by** *blast*
have *card-set-rows*: *card* (*set* (*rows ?gs*)) ≤ *m*
by (*metis assms*(*2*) *card-length length-map rows-gs-eq*)
have *Rs-basis*: *Rs.basis* (*set* (*rows ?gs*))
proof (*rule Rs.dim-gen-is-basis*)
show *card* (*set* (*rows ?gs*)) ≤ *Rs.dim* **using** *card-set-rows Rs-dim-is-m* **by** *auto*
have *Rs.span* (*set* (*rows ?gs*)) = *gs.span* (*set* (*rows ?gs*))
by (*rule gs.span-li-not-depend*[*OF rows-gs-row-space submoduleGS*])
also have ... = *carrier* (*gs.vs* (*gs.row-space ?gs*))
unfolding *gs.row-space-def* **unfolding** *gs.carrier-vs-is-self* **by** *auto*
finally show *Rs.gen-set* (*set* (*rows ?gs*)) **by** *auto*
show *set* (*rows ?gs*) ⊆ *carrier* (*gs.vs* (*gs.row-space ?gs*)) **using** *rows-gs-row-space* **by** *auto*
qed (*simp*)
hence *indpt-Rs*: *Rs.lin-indpt* (*set* (*rows ?gs*)) **unfolding** *Rs.basis-def* **by** *auto*
have *gs-lin-indpt-rows*: *gs.lin-indpt* (*set* (*rows ?gs*))

proof
define *N* **where** *N* ≡ (*gs.row-space ?gs*)
assume *gs.lin-dep* (*set* (*rows ?gs*))
from *this* **obtain** *A f v* **where** *A1*: *finite A* **and** *A2*: *A* ⊆ *set* (*rows ?gs*)
and *lc-gs*: *gs.lincomb f A* = 0_v *n* **and** *v*: *v* ∈ *A* **and** *fv*: *f v* ≠ *0*
unfolding *gs.lin-dep-def* **by** *blast*
have *gs.lincomb f A* = *module.lincomb* (*gs.vs N*) *f A*
by (*rule gs.lincomb-not-depend*, *insert submoduleGS A1 A2 gs.row-space-def*

rows-gs-row-space, *auto simp add*: *N-def gs.row-space-def*)
also have ... = *Rs.lincomb f A* **using** *N-def* **by** *blast*
finally have *Rs.lin-dep* (*set* (*rows ?gs*))
unfolding *Rs.lin-dep-def* **using** *A1 A2 v fv lc-gs* **by** *auto*
thus *False* **using** *indpt-Rs* **by** *auto*
qed
have *card* (*set* (*rows ?gs*)) ≥ *Rs.dim*
by (*rule Rs.gen-ge-dim*, *insert rows-gs-row-space Rs-basis*, *auto simp add*: *Rs.basis-def*)
hence *card-m*: *card* (*set* (*rows ?gs*)) = *m* **using** *card-set-rows Rs-dim-is-m* **by** *auto*
have *distinct* (*map* (*of-int-hom.vec-hom*::*int vec* ⇒ *rat vec*) *gs*)
using *rows-gs-eq assms*(*2*) *card-m card-distinct* **by** *force*
moreover have *set* (*map of-int-hom.vec-hom gs*) ⊆ *Rn* **using** *gs* **by** *auto*
ultimately show *gs.lin-indpt-list* (*map of-int-hom.vec-hom gs*)
using *gs-lin-indpt-rows*
unfolding *rows-gs-eq gs.lin-indpt-list-def*
by *auto*
qed
qed
qed

lemma *gauss-jordan-integer-inverse*: **fixes** *fs gs* :: *int vec list*
assumes *gs*: *set gs* ⊆ *carrier-vec n*
and *len-gs*: *length gs* = *n*
and *fs*: *set fs* ⊆ *carrier-vec n*
and *len-fs*: *length fs* = *n*
and *gauss*: *gauss-jordan-integer-inverse n* (*map-mat rat-of-int* (*mat-of-cols n fs*))

(*map-mat rat-of-int* (*mat-of-cols n gs*)) (*1*$_m$ *n*) (**is** *gauss-jordan-integer-inverse - ?fs ?gs -*)
shows ∃ *U*. *U* ∈ *carrier-mat n n* ∧ *mat-of-rows n gs* = *U* ∗ *mat-of-rows n fs*
proof −
have *fs'*: *?fs* ∈ *carrier-mat n n* **using** *fs len-fs* **by** *auto*
have *gs'*: *?gs* ∈ *carrier-mat n n* **using** *gs len-gs* **by** *auto*
note *gauss* = *gauss*[*unfolded gauss-jordan-integer-inverse-def*]
from *gauss* **obtain** *A* **where** *gauss*: *gauss-jordan ?fs ?gs* = (*1*$_m$ *n*, *A*)
and *int*: *list-all is-int-rat* (*concat* (*mat-to-list A*)) **by** *auto*
note *gauss* = *gauss-jordan*[*OF fs' gs' gauss*]
note *A* = *gauss*(*4*)
let *?A* = *map-mat int-of-rat A*
from *gauss*(*2*)[*OF A*] *A*
have *id*: *?fs* ∗ *A* = *?gs* **by** *auto*
let *?U* = (*map-mat int-of-rat A*)^T
from *A* **have** *U*: *?U* ∈ *carrier-mat n n* **by** *auto*
have *A* = *map-mat of-int ?A* **using** *int*[*unfolded list-all-iff*] *A*
by (*intro eq-matI*, *auto simp*: *mat-to-list-def*)
with *id* **have** *?gs* = *?fs* ∗ *map-mat of-int ?A* **by** *auto*
also have ... = *map-mat of-int* (*mat-of-cols n fs* ∗ *?A*)
by (*rule of-int-hom.mat-hom-mult*[*symmetric*], *insert fs' A*, *auto*)
finally have *mat-of-cols n fs* ∗ *?A* = *mat-of-cols n gs*
using *of-int-hom.mat-hom-inj* **by** *fastforce*
hence (*mat-of-cols n gs*)^T = (*mat-of-cols n fs* ∗ *?A*)^T **by** *simp*
also have ... = *?U* ∗ (*mat-of-cols n fs*)^T
by (*rule transpose-mult*, *insert fs' A*, *auto*)
also have (*mat-of-cols n fs*)^T = *mat-of-rows n fs*
using *fs len-fs* **unfolding** *mat-of-rows-def mat-of-cols-def*
by (*intro eq-matI*, *auto*)
also have (*mat-of-cols n gs*)^T = *mat-of-rows n gs*
using *gs len-gs* **unfolding** *mat-of-rows-def mat-of-cols-def*
by (*intro eq-matI*, *auto*)
finally show *?thesis* **using** *U* **by** *blast*
qed

lemma *LLL-change-basis-mat-inverse*: **assumes** *gs*: *set gs* ⊆ *carrier-vec n*
and *len'*: *length gs* = *n*
and *m* = *n*
and *eq*: *integer-equivalent n fs-init gs*
shows *lattice-of gs* = *lattice-of fs-init LLL-with-assms n m gs α*

```
proof −
  from eq[unfolded integer-equivalent-def Let-def]
 have 1: gauss-jordan-integer-inverse n (of-int-hom.mat-hom (mat-of-cols n fs-init))
   (of-int-hom.mat-hom (mat-of-cols n gs)) (1_m n)
    and 2: gauss-jordan-integer-inverse n (of-int-hom.mat-hom (mat-of-cols n gs))
     (of-int-hom.mat-hom (mat-of-cols n fs-init)) (1_m n)
    by auto
  note len = len[unfolded ‹m = n›]
  from gauss-jordan-integer-inverse[OF gs len' fs-init len 1] ‹m = n›
  obtain U where U: U ∈ carrier-mat m m mat-of-rows n gs = U * mat-of-rows
n fs-init by auto
  from gauss-jordan-integer-inverse[OF fs-init len gs len' 2] ‹m = n›
 obtain V where V: V ∈ carrier-mat m m mat-of-rows n fs-init = V * mat-of-rows
n gs by auto
  from LLL-change-basis[OF gs len'[folded ‹m = n›] V(1) U(1) V(2) U(2)]
  show lattice-of gs = lattice-of fs-init LLL-with-assms n m gs α by blast+
qed

end
```

External solvers must deliver a reduced basis and optionally two matrices to convert between the input and the reduced basis. These two matrices are mandatory if the input matrix is not a square matrix.

```
consts external-lll-solver :: integer × integer ⇒ integer list list ⇒
  integer list list × (integer list list × integer list list)option

definition reduce-basis-external :: rat ⇒ int vec list ⇒ int vec list where
  reduce-basis-external α fs = (case fs of Nil ⇒ [] | Cons f - ⇒ (let
    rb = reduce-basis α;
    fsi = map (map integer-of-int o list-of-vec) fs;
    n = dim-vec f;
    m = length fs in
  case external-lll-solver (map-prod integer-of-int integer-of-int (quotient-of α)) fsi
of
    (gsi, co) ⇒
    let gs = (map (vec-of-list o map int-of-integer) gsi) in
    if ¬ (length gs = m ∧ (∀ gi ∈ set gs. dim-vec gi = n)) then
      Code.abort (STR ''error in external LLL invocation: dimensions of reduced
basis do not fit↩input to external solver: ''
       + String.implode (show fs) + STR ''↩|↩'') (λ -. rb fs)
     else
      case co of Some (u1i, u2i) ⇒ (let
        u1 = mat-of-rows-list m (map (map int-of-integer) u1i);
        u2 = mat-of-rows-list m (map (map int-of-integer) u2i);
        gs = (map (vec-of-list o map int-of-integer) gsi);
        Fs = mat-of-rows n fs;
        Gs = mat-of-rows n gs in
        if (dim-row u1 = m ∧ dim-col u1 = m ∧ dim-row u2 = m ∧ dim-col u2
= m
```

```
        ∧ Fs = u1 * Gs ∧ Gs = u2 * Fs)
      then rb gs
      else Code.abort (STR ''error in external lll invocation↩f,g,u1,u2 are as
follows↩''
        + String.implode (show Fs) + STR ''↩|↩''
        + String.implode (show Gs) + STR ''↩|↩''
        + String.implode (show u1) + STR ''↩|↩''
        + String.implode (show u2) + STR ''↩|↩''
        ) (λ -. rb fs))
  | None ⇒ (if (n = m ∧ integer-equivalent n fs gs) then
     rb gs
    else Code.abort (STR ''error in external LLL invocation:↩'' +
     (if n = m then STR ''reduced matrix does not span same lattice'' else
      STR ''no certificate only allowed for square matrices'')) (λ -. rb fs))
  ))

definition short-vector-external :: rat ⇒ int vec list ⇒ int vec where
  short-vector-external α fs = (hd (reduce-basis-external α fs))

context LLL-with-assms
begin

lemma reduce-basis-external: assumes res: reduce-basis-external α fs-init = fs
  shows reduced fs m LLL-invariant True m fs

proof (atomize(full), goal-cases)
  case 1
  show ?case
  proof (cases LLL-Impl.reduce-basis α fs-init = fs)
    case True
    from reduce-basis[OF this] show ?thesis by simp
  next
    case False
    show ?thesis
    proof (cases fs-init)
      case Nil
      with res have fs = [] unfolding reduce-basis-external-def by auto
      with False Nil have False by (simp add: LLL-Impl.reduce-basis-def)
      thus ?thesis ..
    next
      case (Cons f rest)
      from Cons fs-init len have dim-fs-n: dim-vec f = n by auto
     let ?ext = external-lll-solver (map-prod integer-of-int integer-of-int (quotient-of
α))
        (map (map integer-of-int ∘ list-of-vec) fs-init)
     note res = res[unfolded reduce-basis-external-def Cons Let-def list.case Code.abort-def
dim-fs-n,
         folded Cons]
       from res False obtain gsi co where ext: ?ext = (gsi, co) by (cases ?ext,
```

auto)
define *gs* **where** *gs* = *map* (*vec-of-list o map int-of-integer*) *gsi*
note *res* = *res*[*unfolded ext option.simps split len dim-fs-n*, *folded gs-def*]
from *res False* **have** *not*: (¬ (*length gs* = *m* ∧ (∀ *gi*∈*set gs*. *dim-vec gi* = *n*)))
= *False*
by (*auto split*: *if-splits*)
note *res* = *res*[*unfolded this if-False*]
from *not* **have** *gs*: *set gs* ⊆ *carrier-vec n*
and *len-gs*: *length gs* = *m* **by** *auto*
have *lattice-of gs* = *lattice-of fs-init* ∧ *LLL-with-assms n m gs α* ∧ *LLL-Impl.reduce-basis*
α gs = *fs*
proof (*cases co*)
case (*Some pair*)
from *res Some* **obtain** *u1i u2i* **where** *co*: *co* = *Some* (*u1i*, *u2i*) **by** (*cases*
co, *auto*)
define *u1* **where** *u1* = *mat-of-rows-list m* (*map* (*map int-of-integer*) *u1i*)
define *u2* **where** *u2* = *mat-of-rows-list m* (*map* (*map int-of-integer*) *u2i*)
note *res* = *res*[*unfolded co option.simps split len dim-fs-n*, *folded u1-def*
u2-def gs-def]
from *res False*
have *u1*: *u1* ∈ *carrier-mat m m*
and *u2*: *u2* ∈ *carrier-mat m m*
and *prod1*: *mat-of-rows n fs-init* = *u1* ∗ *mat-of-rows n gs*
and *prod2*: *mat-of-rows n gs* = *u2* ∗ *mat-of-rows n fs-init*
and *gs-v*: *LLL-Impl.reduce-basis α gs* = *fs*
by (*auto split*: *if-splits*)
from *LLL-change-basis*[*OF gs len-gs u1 u2 prod1 prod2*] *gs-v*
show *?thesis* **by** *auto*
next
case *None*
from *res*[*unfolded None option.simps*] *False*
have *id*: *fs* = *LLL-Impl.reduce-basis α gs* **and** *nm*: *n* = *m*
and *equiv*: *integer-equivalent n fs-init gs*
by (*auto split*: *if-splits*)
from *LLL-change-basis-mat-inverse*[*OF gs len-gs*[*folded nm*] *nm*[*symmetric*]
equiv] *id*
show *?thesis* **by** *auto*
qed
hence *id*: *lattice-of gs* = *lattice-of fs-init*
and *assms*: *LLL-with-assms n m gs α*
and *gs-fs*: *LLL-Impl.reduce-basis α gs* = *fs* **by** *auto*
from *LLL-with-assms.reduce-basis*[*OF assms gs-fs*]
have *red*: *reduced fs m* **and** *inv*: *LLL.LLL-invariant n m gs α True m fs* **by**
auto
from *inv*[*unfolded LLL.LLL-invariant-def LLL.L-def id*]
have *lattice*: *lattice-of fs* = *lattice-of fs-init* **by** *auto*
show *?thesis*
proof (*intro conjI red lattice*)
show *LLL-invariant True m fs* **using** *inv* **unfolding** *LLL.LLL-invariant-def*

LLL.L-def id .
    qed
  qed
 qed
qed

lemma *short-vector-external*: **assumes** *res*: *short-vector-external* α *fs-init* = *v*
 and *m0*: $m \neq 0$
shows $v \in$ *carrier-vec n*
 $v \in L - \{0_v\ n\}$
 $h \in L - \{0_v\ n\} \Longrightarrow$ *rat-of-int* (*sq-norm v*) $\leq \alpha\ \hat{}\ (m - 1) *$ *rat-of-int* (*sq-norm h*)
 $v \neq 0_v\ j$
proof (*atomize*(*full*), *goal-cases*)
 case *1*
 obtain *fs* **where** *red*: *reduce-basis-external* α *fs-init* = *fs* **by** *blast*
 from *res*[*unfolded short-vector-external-def red*] **have** *v*: *v* = *hd fs* **by** *auto*
 from *reduce-basis-external*[*OF red*]
 have *red*: *reduced fs m* **and** *inv*: *LLL-invariant True m fs* **by** *blast+*
 from *basis-reduction-short-vector*[*OF inv v m0*]
 show *?case* **by** *blast*
qed
end

Unspecified constant to easily enable/disable external lll solver in generated code

consts *enable-external-lll-solver* :: *bool*

definition *short-vector-hybrid* :: *rat* ⇒ *int vec list* ⇒ *int vec* **where**
 short-vector-hybrid = (*if enable-external-lll-solver then short-vector-external else short-vector*)

definition *reduce-basis-hybrid* :: *rat* ⇒ *int vec list* ⇒ *int vec list* **where**
 reduce-basis-hybrid = (*if enable-external-lll-solver then reduce-basis-external else reduce-basis*)

context *LLL-with-assms*
begin
lemma *short-vector-hybrid*: **assumes** *res*: *short-vector-hybrid* α *fs-init* = *v*
 and *m0*: $m \neq 0$
shows $v \in$ *carrier-vec n*
 $v \in L - \{0_v\ n\}$
 $h \in L - \{0_v\ n\} \Longrightarrow$ *rat-of-int* (*sq-norm v*) $\leq \alpha\ \hat{}\ (m - 1) *$ *rat-of-int* (*sq-norm h*)
 $v \neq 0_v\ j$
 using *short-vector*[*of v, OF - m0*] *short-vector-external*[*of v, OF - m0*]
  res[*unfolded short-vector-hybrid-def*]
 by (*auto split*: *if-splits*)

lemma *reduce-basis-hybrid*: **assumes** *res*: *reduce-basis-hybrid* α *fs-init* = *fs*
shows *reduced fs m LLL-invariant True m fs*
using *reduce-basis-external*[*of fs*] *reduce-basis*[*of fs*] *res*[*unfolded reduce-basis-hybrid-def*]
by (*auto split*: *if-splits*)
end

lemma *lll-oracle-default-code*[*code*]:
external-lll-solver x = *Code.abort* (*STR ''no implementation of external-lll-solver specified''*) (λ -. *external-lll-solver x*)
by *simp*

By default, external solvers are disabled. For enabling an external solver, load it via a separate theory like `FPLLL_Solver.thy`

overloading *enable-external-lll-solver* ≡ *enable-external-lll-solver*
begin
definition *enable-external-lll-solver* **where** *enable-external-lll-solver* = *False*
end

definition *short-vector-test-hybrid xs* =
(*let ys* = *map* (*vec-of-list o map int-of-integer*) *xs*
in integer-of-int (*sq-norm* (*short-vector-hybrid* (*3*/*2*) *ys*)))

end

10.2 A Haskell Interface to the FPLLL-Solver

theory *FPLLL-Solver*
imports *LLL-Certification*
begin

We define *external-lll-solver* via an invocation of the fplll solver. For eta we use the default value of fplll, and delta is chosen so that the required precision of alpha will be guaranteed. We use the command-line option -bvu in order to get the witnesses that are required for certification.

Warning: Since we only define a Haskell binding for FPLLL, the target languages do no longer evaluate to the same results on *short-vector-hybrid*!

code-printing
code-module *FPLLL-Solver* ⇀ (*Haskell*)
‹*module FPLLL-Solver where* {

import System.Process (*proc*,*createProcess*,*waitForProcess*,*CreateProcess*(..),*StdStream*(..));
import System.IO.Unsafe (*unsafePerformIO*);
import System.IO (*stderr*,*hPutStrLn*,*hPutStr*,*hClose*);
import Data.ByteString.Lazy (*hPut*,*hGetContents*,*intercalate*,*ByteString*);

```
import Data.ByteString.Lazy.Char8 (pack,unpack,uncons,cons);
import GHC.IO.Exception (ExitCode(ExitSuccess));
import Data.Char (isNumber, isSpace);
import GHC.IO.Handle (hSetBinaryMode,hSetBuffering,BufferMode(BlockBuffering));
import Control.Exception;
import Data.IORef;

fplll-command :: String;
fplll-command = fplll;

default-eta :: Double;
default-eta = 0.51;

alpha-to-delta :: (Integer,Integer) -> Double;
alpha-to-delta (num,denom) = (fromIntegral denom / fromIntegral num) +
  (default-eta * default-eta);

showrow :: [Integer] -> ByteString;
showrow rowA = (pack []) `mappend` intercalate (pack " ") (map (pack . show) rowA)
`mappend` (pack "\n");
showmat :: [[Integer]] -> ByteString;
showmat matA = (pack []) `mappend` intercalate (pack "\n") (map showrow matA)
`mappend` (pack "\n");

data Mode = Simple | Certificate;

flags :: Mode -> String;
flags Simple = "b";
flags Certificate = "bvu";

getMode xs = (let m = length xs in if m == 0 then Certificate
  else if m == length (head xs) then Simple else Certificate);

fplll-solver :: (Integer,Integer) -> [[Integer]] -> ([[Integer]], Maybe ([[Integer]],[[Integer]]));
fplll-solver alpha in-mat = unsafePerformIO $ catchE $ do {
  (Just f-in,Just f-out,Just f-err,f-pid) <- createProcess (proc fplll-command ["-e",
show default-eta, "-d", show (alpha-to-delta alpha), "-of", flags mode]){std-in = Cre-
atePipe, std-err = CreatePipe, std-out = CreatePipe};
  hSetBinaryMode f-in True;
  hSetBinaryMode f-out True;
  hSetBinaryMode f-err True;
  hSetBuffering f-out (BlockBuffering Nothing);
  hPut f-in (showmat in-mat);
  res <- hGetContents f-out;
  hClose f-in;
  parseRes res}
 where {
   mode = getMode in-mat;
   catchE m = catch m def;
```

```
def :: SomeException -> IO ([[Integer]], Maybe ([[Integer]], [[Integer]]));
def - = seq sendError $ default-answer;
unconsIO a = case uncons a of{
   Just b -> return b;
   - -> abort Unexpected end of file / input};
parseMat ('[',as)
 = do {
   (h0,rem0) <- parseSpaces =<< unconsIO as;
   (rows,(h1,rem1)) <- parseRows (h0,rem0);
   case seq rows h1 of{
     ']' -> return (rows,rem1);
     - -> abort$ Expecting closing ']' while parsing a matrix.\n}
   } :: IO ([[Integer]], ByteString);
parseMat - = abort Expecting opening '[' while parsing a matrix;
parseRows ('[',rem0)
 = do {
  (nums,(h2,rem2))<-parseNums =<< parseSpaces =<< unconsIO rem0;
  case seq nums h2 of
    ']' -> do { (h4,rem4) <- parseSpaces =<< unconsIO rem2;
                (rows,rem5) <- parseRows (h4,rem4);
                return (nums:rows,rem5) }
    - -> abort$ Expecting closing ']' while parsing a row\n
    } :: IO ([[Integer]],(Char, ByteString));
parseRows r = return ([],r);
parseNums (a,rem0) =
  (if isNumber a || a == '-' then do {
    (n,(h1,rem1)) <- parseNum =<< unconsIO rem0;
    rem2 <- parseSpaces (h1,rem1);
    num <- return (read (a:n));
    (nums,rem3) <- seq (num==num)$ parseNums rem2;
    return (seq nums $ num:nums,rem3) }
  else if isSpace a then do {
    rem1 <- parseSpaces (a,rem0);
    parseNums rem1}
  else return ([],(a, rem0))) :: IO ([Integer], (Char, ByteString));
parseNum (a,rem0) =
  if isNumber a then do {
    (num,rem1) <- parseNum =<< unconsIO rem0;
    return (a:num,rem1)
    }
  else return (mempty,(a,rem0));
parseSpaces (a,as) = if isSpace a then case uncons as of { Nothing -> return
(a,mempty); Just v -> parseSpaces v } else return (a,as);
parseRes :: ByteString -> IO ([[Integer]], Maybe ([[Integer]], [[Integer]]));
parseRes res = if res == mempty
    then default-answer
    else do {
      rem0' <- parseSpaces =<< unconsIO res;
      (m1,rem1) <- parseMat rem0';
```

```
        -- putStrLn Parsed a matrix;
        case mode of
          Simple -> return (m1, Nothing);
          - -> do {
            rem1' <- parseSpaces =<< unconsIO rem1;
            (m2,rem2) <- seq m1$ parseMat rem1';
            -- putStrLn Parsed a matrix;
            rem2' <- parseSpaces =<< unconsIO rem2;
            (m3,rem3) <- seq m2$ parseMat rem2';
            seq m3$ return ();
            -- putStrLn Parsed a matrix;
            if rem3 /= mempty
               then do { (-,rem2') <- parseSpaces =<< unconsIO rem3;
                         if rem2' /= mempty
                            then abort Unexpected output after parsing three matrices.
                            else return (m1, Just (m2,m3)) }
               else return (m1,Just (m2,m3))
               }
      };
  fail-to-execute = seq sendError default-answer;

  default-answer = -- not small enough, but it'll be accepted
    return (in-mat, case mode of Simple -> Nothing; - -> Just (id-ofsize (length
in-mat),id-ofsize (length in-mat)));
  abort str = error$ Runtime exception in parsing fplll output:\n++str;
  };


sendError :: (); -- bad trick using unsafeIO to make this error only appear once.
I believe this is OK since the error is non-critical and the 'only appear once' is
non-critical too.
sendError = unsafePerformIO $ do {
  hPutStrLn stderr --- WARNING ---;
  hPutStrLn stderr Failed to run fplll.;
  hPutStrLn stderr To remove this warning, either:;
  hPutStrLn stderr   - install fplll and ensure it is in your path.;
  hPutStrLn stderr   - create an executable fplll that always returns successfully
without generating output.;
  hPutStrLn stderr Installing fplll correctly helps to reduce time spent verifying your
certificate.;
  hPutStrLn stderr --- END OF WARNING ---
  };

id-ofsize :: Int -> [[Integer]];
id-ofsize n = [[if i == j then 1 else 0 | j <- [0..n-1]] | i <- [0..n-1]];
}›
```

code-reserved (*Haskell*) *FPLLL-Solver fplll-solver*

code-printing
  constant *external-lll-solver* ⇀ (*Haskell*) *FPLLL'-Solver.fplll'-solver*
| **constant** *enable-external-lll-solver* ⇀ (*Haskell*) *True*

Note that since we only enabled the external LLL solver for Haskell, the result of *short-vector-hybrid* will usually differ when executed in Haskell in comparison to any of the other target languages. For instance, consider the invocation of:

value (*code*) *short-vector-test-hybrid* [[*1*,*4903*,*4902*], [*0*,*39023*,*0*], [*0*,*0*,*39023*]]

The above value-command evaluates the expression in Eval/SML to 77714 (by computing a short vector solely by the verified *short-vector* algorithm, whereas the generated Haskell-code via the external LLL solver yields 60414!

end

References

[1] Ú. Erlingsson, E. Kaltofen, and D. R. Musser. Generic Gram-Schmidt orthogonalization by exact division. In *Proceedings of the 1996 International Symposium on Symbolic and Algebraic Computation, ISSAC '96, Zurich, Switzerland, July 24-26, 1996*, pages 275–282. ACM, 1996.

[2] J. v. z. Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 2nd edition, 2003.

[3] A. K. Lenstra, H. W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:515–534, 1982.

[4] R. Thiemann and A. Yamada. Formalizing Jordan normal forms in Isabelle/HOL. In *CPP 2016*, pages 88–99. ACM, 2016.