

A verified LLL algorithm*

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

December 14, 2021

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	13
4	Norms	14
4.1	L- ∞ Norms	14
4.2	Square Norms	15
4.2.1	Square norms for vectors	15
4.2.2	Square norm for polynomials	16
4.3	Relating Norms	17
5	Optimized Code for Integer-Rational Operations	24

*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	25
7	List representation	25
8	Gram-Schmidt	27
8.1	Explicit Bounds for Size of Numbers that Occur During GSO Algorithm	44
8.2	Gram-Schmidt Implementation for Integer Vectors	47
8.3	Lemmas Summarizing All Bounds During GSO Computation	55
9	The LLL Algorithm	55
9.1	Core Definitions, Invariants, and Theorems for Basic Version	56
9.2	Basic LLL implementation based on previous results	63
9.3	Integer LLL Implementation which Stores Multiples of the μ -Values	66
9.3.1	Updates of the integer values for Swap, Add, etc.	66
9.3.2	Implementation of LLL via Integer Operations and Arrays	68
9.4	Bound on Number of Arithmetic Operations for Integer Implementation	75
9.5	Explicit Bounds for Size of Numbers that Occur During LLL Algorithm	82
9.5.1	<i>LLL-bound-invariant</i> is maintained during execution of <i>reduce-basis</i>	85
9.5.2	Bound extracted from <i>LLL-bound-invariant</i>	86
10	Certification of External LLL Invocations	89
10.1	Checking Results of External LLL Solvers	89
10.2	A Haskell Interface to the FPLLL-Solver	93

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 $fs_0, \dots, fs_{m-1} \in \mathbb{Z}^n$, 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  $v \in \text{lattice\_of } fs - \{0_v\}$ 
and  $h \in \text{lattice\_of } fs - \{0_v\} \longrightarrow \|v\|^2 \leq \alpha^{m-1} \cdot \|h\|^2$ 

```

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 execution 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 standard library and some other AFP entries.

```

theory Missing-Lemmas
imports
  Berlekamp-Zassenhaus.Sublist-Iteration
  Berlekamp-Zassenhaus.Square-Free-Int-To-Square-Free-GFp
  Algebraic-Numbers.Resultant
  Jordan-Normal-Form.Conjugate
  Jordan-Normal-Form.Missing-VectorSpace
  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 <proof>

lemma of-nat-ge-zero[intro!]: of-nat n ≥ 0
  <proof>

lemma zero-le-power [simp]:  $0 \leq a \implies 0 \leq a^n$ 
  <proof>

lemma power-mono:  $a \leq b \implies 0 \leq a \implies a^n \leq b^n$ 
  <proof>

lemma one-le-power [simp]:  $1 \leq a \implies 1 \leq a^n$ 
  <proof>

lemma power-le-one:  $0 \leq a \implies a \leq 1 \implies a^n \leq 1$ 
  <proof>

lemma power-gt1-lemma:
  assumes gt1:  $1 < a$ 
  shows  $1 < a * a^n$ 
  <proof>

lemma power-gt1:  $1 < a \implies 1 < a^{Suc\ n}$ 
  <proof>

lemma one-less-power [simp]:  $1 < a \implies 0 < n \implies 1 < a^n$ 
  <proof>

lemma power-decreasing:  $n \leq N \implies 0 \leq a \implies a \leq 1 \implies a^N \leq a^n$ 

```

<proof>

lemma *power-increasing*: $n \leq N \implies 1 \leq a \implies a^n \leq a^N$
<proof>

lemma *power-Suc-le-self*: $0 \leq a \implies a \leq 1 \implies a^{Suc\ n} \leq a$
<proof>

end

lemma *prod-list-nonneg*: $(\bigwedge x. (x :: 'a :: ordered-semiring-1) \in set\ xs \implies x \geq 0) \implies prod-list\ xs \geq 0$
<proof>

subclass (in *ordered-idom*) *ordered-semiring-1* *<proof>*

lemma *log-prod*: **assumes** $0 < a\ a \neq 1 \bigwedge x. x \in X \implies 0 < f\ x$
shows $log\ a\ (prod\ f\ X) = sum\ (log\ a\ o\ f)\ X$
<proof>

subclass (in *ordered-idom*) *zero-less-one* *<proof>*
hide-fact *Missing-Ring.zero-less-one*

instance *real* :: *ordered-semiring-strict* *<proof>*
instance *real* :: *linordered-idom* *<proof>*

lemma *less-1-mult'*:
fixes $a :: 'a :: linordered-semidom$
shows $1 < a \implies 1 \leq b \implies 1 < a * b$
<proof>

lemma *upt-minus-eq-append*: $i \leq j \implies i \leq j - k \implies [i..<j] = [i..<j-k] @ [j-k..<j]$
<proof>

lemma *list-trisect*: $x < length\ lst \implies [0..<length\ lst] = [0..<x] @ x \#[Suc\ x..<length\ lst]$
<proof>

lemma *id-imp-bij-betw*:
assumes $f: f : A \rightarrow A$
and $ff: \bigwedge a. a \in A \implies f\ (f\ a) = a$
shows *bij-betw* $f\ A\ A$

<proof>

lemma *range-subsetI*:

assumes $\bigwedge x. f x = g (h x)$ **shows** $\text{range } f \subseteq \text{range } g$

<proof>

lemma *Gcd-uminus*:

fixes $A::\text{int set}$

assumes *finite A*

shows $\text{Gcd } A = \text{Gcd } (\text{uminus } ` A)$

<proof>

lemma *aux-abs-int*: **fixes** $c :: \text{int}$

assumes $c \neq 0$

shows $|x| \leq |x * c|$

<proof>

lemma *mod-0-abs-less-imp-0*:

fixes $a::\text{int}$

assumes $a1: [a = 0] \pmod m$

and $a2: \text{abs}(a) < m$

shows $a = 0$

<proof>

lemma *sum-list-zero*:

assumes $\text{set } xs \subseteq \{0\}$ **shows** $\text{sum-list } xs = 0$

<proof>

lemma *max-idem [simp]*: **shows** $\text{max } a a = a$ *<proof>*

lemma *hom-max*:

assumes $a \leq b \longleftrightarrow f a \leq f b$

shows $f (\text{max } a b) = \text{max } (f a) (f b)$ *<proof>*

lemma *le-max-self*:

fixes $a b :: 'a :: \text{preorder}$

assumes $a \leq b \vee b \leq a$ **shows** $a \leq \text{max } a b$ **and** $b \leq \text{max } a b$

<proof>

lemma *le-max*:

fixes $a b :: 'a :: \text{preorder}$

assumes $c \leq a \vee c \leq b$ **and** $a \leq b \vee b \leq a$ **shows** $c \leq \text{max } a b$

<proof>

fun *max-list* **where**

$\text{max-list } [] = (\text{THE } x. \text{False})$

$|\ \text{max-list } [x] = x$

| $max-list (x \# y \# xs) = max x (max-list (y \# xs))$

declare $max-list.simps(1)$ [simp del]

declare $max-list.simps(2-3)$ [code]

lemma $max-list-Cons$: $max-list (x\#xs) = (if xs = [] then x else max x (max-list xs))$

$\langle proof \rangle$

lemma $max-list-mem$: $xs \neq [] \implies max-list xs \in set xs$

$\langle proof \rangle$

lemma $mem-set-imp-le-max-list$:

fixes $xs :: 'a :: preorder list$

assumes $\bigwedge a b. a \in set xs \implies b \in set xs \implies a \leq b \vee b \leq a$

and $a \in set xs$

shows $a \leq max-list xs$

$\langle proof \rangle$

lemma $le-max-list$:

fixes $xs :: 'a :: preorder list$

assumes $ord: \bigwedge a b. a \in set xs \implies b \in set xs \implies a \leq b \vee b \leq a$

and $ab: a \leq b$

and $b: b \in set xs$

shows $a \leq max-list xs$

$\langle proof \rangle$

lemma $max-list-le$:

fixes $xs :: 'a :: preorder list$

assumes $a: \bigwedge x. x \in set xs \implies x \leq a$

and $xs: xs \neq []$

shows $max-list xs \leq a$

$\langle proof \rangle$

lemma $max-list-as-Greatest$:

assumes $\bigwedge x y. x \in set xs \implies y \in set xs \implies x \leq y \vee y \leq x$

shows $max-list xs = (GREATEST a. a \in set xs)$

$\langle proof \rangle$

lemma $hom-max-list-commute$:

assumes $xs \neq []$

and $\bigwedge x y. x \in set xs \implies y \in set xs \implies h (max x y) = max (h x) (h y)$

shows $h (max-list xs) = max-list (map h xs)$

$\langle proof \rangle$

primrec $rev-upt :: nat \Rightarrow nat \Rightarrow nat list ((1[->..-]))$ **where**

$rev-upt-0: [0>..j] = []$ |

rev-upt-Suc: $[(\text{Suc } i) > .. j] = (\text{if } i \geq j \text{ then } i \# [i > .. j] \text{ else } [])$

lemma *rev-upt-rec*: $[i > .. j] = (\text{if } i > j \text{ then } [i > .. \text{Suc } j] @ [j] \text{ else } [])$
<proof>

definition *rev-upt-aux* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat list} \Rightarrow \text{nat list}$ **where**
rev-upt-aux $i j js = [i > .. j] @ js$

lemma *upt-aux-rec* [code]:
rev-upt-aux $i j js = (\text{if } j \geq i \text{ then } js \text{ else } \text{rev-upt-aux } i (\text{Suc } j) (j \# js))$
<proof>

lemma *rev-upt-code*[code]: $[i > .. j] = \text{rev-upt-aux } i j []$
<proof>

lemma *upt-rev-upt*:
rev $[j > .. i] = [i .. < j]$
<proof>

lemma *rev-upt-upt*:
rev $[i .. < j] = [j > .. i]$
<proof>

lemma *length-rev-upt* [simp]: $\text{length } [i > .. j] = i - j$
<proof>

lemma *nth-rev-upt* [simp]: $j + k < i \implies [i > .. j] ! k = i - 1 - k$
<proof>

lemma *nth-map-rev-upt*:
assumes $i < m - n$
shows $(\text{map } f [m > .. n]) ! i = f (m - 1 - i)$
<proof>

lemma *coeff-mult-monom*:
coeff $(p * \text{monom } a d) i = (\text{if } d \leq i \text{ then } a * \text{coeff } p (i - d) \text{ else } 0)$
<proof>

lemma *vec-of-poly-0* [simp]: $\text{vec-of-poly } 0 = 0_v \ 1$ *<proof>*

lemma *vec-index-vec-of-poly* [simp]: $i \leq \text{degree } p \implies \text{vec-of-poly } p \$ i = \text{coeff } p$
 $(\text{degree } p - i)$
<proof>

lemma *poly-of-vec-vec*: $\text{poly-of-vec } (\text{vec } n f) = \text{Poly } (\text{rev } (\text{map } f [0 .. < n]))$
<proof>

lemma *sum-list-map-dropWhile0*:

assumes $f0: f\ 0 = 0$

shows $sum\text{-}list\ (map\ f\ (dropWhile\ ((=)\ 0)\ xs)) = sum\text{-}list\ (map\ f\ xs)$

<proof>

lemma *coeffs-poly-of-vec*:

$coeffs\ (poly\text{-}of\text{-}vec\ v) = rev\ (dropWhile\ ((=)\ 0)\ (list\text{-}of\text{-}vec\ v))$

<proof>

lemma *poly-of-vec-vCons*:

$poly\text{-}of\text{-}vec\ (vCons\ a\ v) = monom\ a\ (dim\text{-}vec\ v) + poly\text{-}of\text{-}vec\ v$ (**is** $?l = ?r$)

<proof>

lemma *poly-of-vec-as-Poly*: $poly\text{-}of\text{-}vec\ v = Poly\ (rev\ (list\text{-}of\text{-}vec\ v))$

<proof>

lemma *poly-of-vec-add*:

assumes $dim\text{-}vec\ a = dim\text{-}vec\ b$

shows $poly\text{-}of\text{-}vec\ (a + b) = poly\text{-}of\text{-}vec\ a + poly\text{-}of\text{-}vec\ b$

<proof>

lemma *degree-poly-of-vec-less*:

assumes $0 < dim\text{-}vec\ v$ **and** $dim\text{-}vec\ v \leq n$ **shows** $degree\ (poly\text{-}of\text{-}vec\ v) < n$

<proof>

lemma (**in** *vec-module*) *poly-of-vec-finsum*:

assumes $f \in X \rightarrow carrier\text{-}vec\ n$

shows $poly\text{-}of\text{-}vec\ (finsum\ V\ f\ X) = (\sum\ i \in X.\ poly\text{-}of\text{-}vec\ (f\ i))$

<proof>

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*: $vec\text{-}of\text{-}poly\text{-}n\ p\ (Suc\ (degree\ p)) = vec\text{-}of\text{-}poly\ p$

<proof>

lemma *vec-of-poly-n-0* [*simp*]: $vec\text{-}of\text{-}poly\text{-}n\ p\ 0 = vNil$

<proof>

lemma *vec-dim-vec-of-poly-n* [*simp*]:

$dim\text{-}vec\ (vec\text{-}of\text{-}poly\text{-}n\ p\ n) = n$

$vec\text{-}of\text{-}poly\text{-}n\ p\ n \in carrier\text{-}vec\ n$

<proof>

lemma *dim-vec-of-poly* [*simp*]: $dim\text{-}vec\ (vec\text{-}of\text{-}poly\ f) = degree\ f + 1$

$\langle proof \rangle$

lemma *vec-index-of-poly-n*:

assumes $i < n$

shows $vec\text{-of-poly-n } p \ n \ \$ \ i =$

$(if \ i < \ n - \ Suc \ (degree \ p) \ then \ 0 \ else \ coeff \ p \ (n - i - 1))$

$\langle proof \rangle$

lemma *vec-of-poly-n-pCons[simp]*:

shows $vec\text{-of-poly-n } (pCons \ a \ p) \ (Suc \ n) = vec\text{-of-poly-n } p \ n \ @_v \ vec\text{-of-list } [a]$

(is $?l = ?r$)

$\langle proof \rangle$

lemma *vec-of-poly-pCons*:

shows $vec\text{-of-poly } (pCons \ a \ p) =$

$(if \ p = 0 \ then \ vec\text{-of-list } [a] \ else \ vec\text{-of-poly } p \ @_v \ vec\text{-of-list } [a])$

$\langle proof \rangle$

lemma *list-of-vec-of-poly [simp]*:

$list\text{-of-vec } (vec\text{-of-poly } p) = (if \ p = 0 \ then \ [0] \ else \ rev \ (coeffs \ p))$

$\langle proof \rangle$

lemma *poly-of-vec-of-poly-n*:

assumes $p: degree \ p < n$

shows $poly\text{-of-vec } (vec\text{-of-poly-n } p \ n) = p$

$\langle proof \rangle$

lemma *vec-of-poly-n0[simp]*: $vec\text{-of-poly-n } 0 \ n = 0_v \ n$

$\langle proof \rangle$

lemma *vec-of-poly-n-add*: $vec\text{-of-poly-n } (a + b) \ n = vec\text{-of-poly-n } a \ n + vec\text{-of-poly-n } b \ n$

$\langle proof \rangle$

lemma *vec-of-poly-n-poly-of-vec*:

assumes $n: dim\text{-vec } g = n$

shows $vec\text{-of-poly-n } (poly\text{-of-vec } g) \ n = g$

$\langle proof \rangle$

lemma *poly-of-vec-scalar-mult*:

assumes $degree \ b < n$

shows $poly\text{-of-vec } (a \cdot_v \ (vec\text{-of-poly-n } b \ n)) = smult \ a \ b$

$\langle proof \rangle$

definition *vec-of-poly-rev-shifted* **where**

$vec\text{-of-poly-rev-shifted } p \ n \ s \ j \equiv$

$vec \ n \ (\lambda i. \ if \ i \leq j \wedge j \leq s + i \ then \ coeff \ p \ (s + i - j) \ else \ 0)$

lemma *vec-of-poly-rev-shifted-dim*[simp]: $\dim\text{-vec } (\text{vec-of-poly-rev-shifted } p \ n \ s \ j) = n$
 ⟨proof⟩

lemma *col-sylvester-sub*:
assumes $j: j < m + n$
shows $\text{col } (\text{sylvester-mat-sub } m \ n \ p \ q) \ j = \text{vec-of-poly-rev-shifted } p \ n \ m \ j \ @_v \ \text{vec-of-poly-rev-shifted } q \ m \ n \ j$ (**is** ?l = ?r)
 ⟨proof⟩

lemma *vec-of-poly-rev-shifted-scalar-prod*:
fixes $p \ v$
defines $q \equiv \text{poly-of-vec } v$
assumes $m: \text{degree } p \leq m$ **and** $n: \dim\text{-vec } v = n$
assumes $j: j < m+n$
shows $\text{vec-of-poly-rev-shifted } p \ n \ m \ (n+m-\text{Suc } j) \cdot v = \text{coeff } (p * q) \ j$ (**is** ?l = ?r)
 ⟨proof⟩

lemma *sylvester-sub-poly*:
fixes $p \ q :: 'a :: \text{comm-semiring-0 } \text{poly}$
assumes $m: \text{degree } p \leq m$
assumes $n: \text{degree } q \leq n$
assumes $v: v \in \text{carrier-vec } (m+n)$
shows $\text{poly-of-vec } ((\text{sylvester-mat-sub } m \ n \ p \ q)^T *_v v) = \text{poly-of-vec } (\text{vec-first } v \ n) * p + \text{poly-of-vec } (\text{vec-last } v \ m) * q$ (**is** ?l = ?r)
 ⟨proof⟩

lemma *normalize-field* [simp]: $\text{normalize } (a :: 'a :: \{\text{field}, \text{semiring-gcd}\}) = (\text{if } a = 0 \text{ then } 0 \text{ else } 1)$
 ⟨proof⟩

lemma *content-field* [simp]: $\text{content } (p :: 'a :: \{\text{field}, \text{semiring-gcd}\} \text{poly}) = (\text{if } p = 0 \text{ then } 0 \text{ else } 1)$
 ⟨proof⟩

lemma *primitive-part-field* [simp]: $\text{primitive-part } (p :: 'a :: \{\text{field}, \text{semiring-gcd}\} \text{poly}) = p$
 ⟨proof⟩

lemma *primitive-part-dvd*: $\text{primitive-part } a \ \text{dvd } a$
 ⟨proof⟩

lemma *degree-abs* [simp]:
 $\text{degree } |p| = \text{degree } p$ ⟨proof⟩

lemma *degree-gcd1*:
assumes *a-not0*: $a \neq 0$
shows $\text{degree } (\text{gcd } a \ b) \leq \text{degree } a$
<proof>

lemma *primitive-part-neg [simp]*:
fixes $a :: 'a :: \{\text{factorial-ring-gcd}, \text{factorial-semiring-multiplicative}\}$ *poly*
shows $\text{primitive-part } (-a) = - \text{primitive-part } a$
<proof>

lemma *content-uminus[simp]*:
fixes $f :: \text{int } \text{poly}$
shows $\text{content } (-f) = \text{content } f$
<proof>

lemma *pseudo-mod-monic*:
fixes $f \ g :: 'a :: \{\text{comm-ring-1}, \text{semiring-1-no-zero-divisors}\}$ *poly*
defines $r \equiv \text{pseudo-mod } f \ g$
assumes *monic-g*: *monic g*
shows $\exists q. f = g * q + r \wedge r = 0 \vee \text{degree } r < \text{degree } g$
<proof>

lemma *monic-imp-div-mod-int-poly-degree*:
fixes $p :: 'a :: \{\text{comm-ring-1}, \text{semiring-1-no-zero-divisors}\}$ *poly*
assumes *m*: *monic u*
shows $\exists q \ r. p = q * u + r \wedge (r = 0 \vee \text{degree } r < \text{degree } u)$
<proof>

corollary *monic-imp-div-mod-int-poly-degree2*:
fixes $p :: 'a :: \{\text{comm-ring-1}, \text{semiring-1-no-zero-divisors}\}$ *poly*
assumes *m*: *monic u* **and** *deg-u*: $\text{degree } u > 0$
shows $\exists q \ r. p = q * u + r \wedge (\text{degree } r < \text{degree } u)$
<proof>

lemma (**in** *zero-hom*) *hom-upper-triangular*:
 $A \in \text{carrier-mat } n \ n \implies \text{upper-triangular } A \implies \text{upper-triangular } (\text{map-mat } \text{hom } A)$
<proof>

end

3 Auxiliary Lemmas and Definitions for Immutable Arrays

We define some definitions on immutable arrays, and modify the simplification 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 ($\lambda j. \text{if } j = i \text{ then } x \text{ else } a \text{ !! } j$) (IArray.length a)

lemma *iarray-cong*: $n = m \Longrightarrow (\bigwedge i. i < m \Longrightarrow f i = g i) \Longrightarrow \text{IArray.of-fun } f \ n = \text{IArray.of-fun } g \ m$
 $\langle \text{proof} \rangle$

lemma *iarray-cong'*: $(\bigwedge i. i < n \Longrightarrow f i = g i) \Longrightarrow \text{IArray.of-fun } f \ n = \text{IArray.of-fun } g \ n$
 $\langle \text{proof} \rangle$

lemma *iarray-update-length[simp]*: $\text{IArray.length } (\text{iarray-update } a \ i \ x) = \text{IArray.length } a$
 $\langle \text{proof} \rangle$

lemma *iarray-length-of-fun[simp]*: $\text{IArray.length } (\text{IArray.of-fun } f \ n) = n$ $\langle \text{proof} \rangle$

lemma *iarray-update-of-fun[simp]*: $\text{iarray-update } (\text{IArray.of-fun } f \ n) \ i \ x = \text{IArray.of-fun } (f \ (i := x)) \ n$
 $\langle \text{proof} \rangle$

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

lemma *iarray-append-code[code]*: $\text{iarray-append } xs \ x = \text{IArray } (\text{IArray.list-of } xs \ @ \ [x])$
 $\langle \text{proof} \rangle$

lemma *iarray-append-of-fun[simp]*: $\text{iarray-append } (\text{IArray.of-fun } f \ n) \ x = \text{IArray.of-fun } (f \ (n := x)) \ (\text{Suc } n)$
 $\langle \text{proof} \rangle$

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

lemma *iarray-of-fun-sub[simp]*: $i < n \Longrightarrow \text{IArray.of-fun } f \ n \ \! \! i = f \ i$
 $\langle \text{proof} \rangle$

lemma *IArray-of-fun-conv*: $\text{IArray } xs = \text{IArray.of-fun } (\lambda i. xs \ ! \ i) \ (\text{length } xs)$

```

    <proof>

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
        HOL-Library.Adhoc-Overloading
        Jordan-Normal-Form.Conjugate
        Algebraic-Numbers.Resultant
        Missing-Lemmas
begin

```

4.1 L- ∞ Norms

```

consts linf-norm :: 'a  $\Rightarrow$  'b ( $\|(-)\|_\infty$ )

```

```

definition linf-norm-vec where linf-norm-vec v  $\equiv$  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  $\equiv$  max-list (map abs (coeffs f)
@ [0])

```

```

adhoc-overloading linf-norm linf-norm-poly

```

```

lemma linf-norm-vec:  $\|vec\ n\ f\|_\infty = \text{max-list (map (abs \circ f) [0..<n]) @ [0]}$ 
    <proof>

```

```

lemma linf-norm-vec-vCons[simp]:  $\|vCons\ a\ v\|_\infty = \text{max } |a| \|v\|_\infty$ 
    <proof>

```

```

lemma linf-norm-vec-0 [simp]:  $\|vec\ 0\ f\|_\infty = 0$  <proof>

```

```

lemma linf-norm-zero-vec [simp]:  $\|0_v\ n :: 'a :: \text{ordered-ab-group-add-abs}\ \|vec\|_\infty = 0$ 
    <proof>

```

```

lemma linf-norm-vec-ge-0 [intro!]:
    fixes v :: 'a :: ordered-ab-group-add-abs vec

```

shows $\|v\|_\infty \geq 0$
<proof>

lemma *linf-norm-vec-eq-0* [simp]:
fixes $v :: 'a :: \text{ordered-ab-group-add-abs } \text{vec}$
assumes $v \in \text{carrier-vec } n$
shows $\|v\|_\infty = 0 \longleftrightarrow v = 0_v \ n$
<proof>

lemma *linf-norm-vec-greater-0* [simp]:
fixes $v :: 'a :: \text{ordered-ab-group-add-abs } \text{vec}$
assumes $v \in \text{carrier-vec } n$
shows $\|v\|_\infty > 0 \longleftrightarrow v \neq 0_v \ n$
<proof>

lemma *linf-norm-poly-0* [simp]: $\|0::\text{- poly}\|_\infty = 0$
<proof>

lemma *linf-norm-pCons* [simp]:
fixes $p :: 'a :: \text{ordered-ab-group-add-abs } \text{poly}$
shows $\|p\text{Cons } a \ p\|_\infty = \max |a| \ \|p\|_\infty$
<proof>

lemma *linf-norm-poly-ge-0* [intro!]:
fixes $f :: 'a :: \text{ordered-ab-group-add-abs } \text{poly}$
shows $\|f\|_\infty \geq 0$
<proof>

lemma *linf-norm-poly-eq-0* [simp]:
fixes $f :: 'a :: \text{ordered-ab-group-add-abs } \text{poly}$
shows $\|f\|_\infty = 0 \longleftrightarrow f = 0$
<proof>

lemma *linf-norm-poly-greater-0* [simp]:
fixes $f :: 'a :: \text{ordered-ab-group-add-abs } \text{poly}$
shows $\|f\|_\infty > 0 \longleftrightarrow f \neq 0$
<proof>

4.2 Square Norms

consts *sq-norm* :: $'a \Rightarrow 'b \ (\|(-)\|^2)$

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

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

lemma *sq-norm-vec-vCons[simp]*: $\|v\text{Cons } a \ v\|^2 = \|a\|^2 + \|v\|^2$
 $\langle \text{proof} \rangle$

lemma *sq-norm-vec-0[simp]*: $\|\text{vec } 0 \ f\|^2 = 0$
 $\langle \text{proof} \rangle$

lemma *sq-norm-vec-as-cscalar-prod*:
fixes $v :: 'a :: \text{conjugatable-ring vec}$
shows $\|v\|^2 = v \cdot c \ v$
 $\langle \text{proof} \rangle$

lemma *sq-norm-zero-vec[simp]*: $\|0_v \ n :: 'a :: \text{conjugatable-ring vec}\|^2 = 0$
 $\langle \text{proof} \rangle$

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

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

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

4.2.2 Square norm for polynomials

definition *sq-norm-poly* **where** *sq-norm-poly* $p \equiv \sum a \leftarrow \text{coeffs } p. \|a\|^2$

adhoc-overloading *sq-norm sq-norm-poly*

lemma *sq-norm-poly-0 [simp]*: $\|0::\text{-poly}\|^2 = 0$
 $\langle \text{proof} \rangle$

lemma *sq-norm-poly-pCons [simp]*:
fixes $a :: 'a :: \text{conjugatable-ring}$
shows $\|p\text{Cons } a \ p\|^2 = \|a\|^2 + \|p\|^2$
 $\langle \text{proof} \rangle$

lemma *sq-norm-poly-ge-0 [intro!]*:
fixes $p :: 'a :: \text{conjugatable-ordered-ring poly}$
shows $\|p\|^2 \geq 0$
 $\langle \text{proof} \rangle$

lemma *sq-norm-poly-eq-0 [simp]*:
fixes $p :: 'a :: \{\text{conjugatable-ordered-ring, ring-no-zero-divisors}\}$ *poly*
shows $\|p\|^2 = 0 \iff p = 0$
 $\langle \text{proof} \rangle$

lemma *sq-norm-poly-pos [simp]*:
fixes $p :: 'a :: \{\text{conjugatable-ordered-ring, ring-no-zero-divisors}\}$ *poly*

shows $\|p\|^2 > 0 \longleftrightarrow p \neq 0$
 ⟨proof⟩

lemma *sq-norm-vec-of-poly* [simp]:
fixes $p :: 'a :: \text{conjugatable-ring poly}$
shows $\|\text{vec-of-poly } p\|^2 = \|p\|^2$
 ⟨proof⟩

lemma *sq-norm-poly-of-vec* [simp]:
fixes $v :: 'a :: \text{conjugatable-ring vec}$
shows $\|\text{poly-of-vec } v\|^2 = \|v\|^2$
 ⟨proof⟩

4.3 Relating Norms

A class where ordering around 0 is linear.

abbreviation (in *ordered-semiring*) *is-real* **where** $\text{is-real } a \equiv a < 0 \vee a = 0 \vee 0 < a$

class *semiring-real-line* = *ordered-semiring-strict* + *ordered-semiring-0* +
assumes *add-pos-neg-is-real*: $a > 0 \implies b < 0 \implies \text{is-real } (a + b)$
and *mult-neg-neg*: $a < 0 \implies b < 0 \implies 0 < a * b$
and *pos-pos-linear*: $0 < a \implies 0 < b \implies a < b \vee a = b \vee b < a$
and *neg-neg-linear*: $a < 0 \implies b < 0 \implies a < b \vee a = b \vee b < a$
begin

lemma *add-neg-pos-is-real*: $a < 0 \implies b > 0 \implies \text{is-real } (a + b)$
 ⟨proof⟩

lemma *nonneg-linorder-cases* [consumes 2, case-names less eq greater]:
assumes $0 \leq a$ **and** $0 \leq b$
and $a < b \implies \text{thesis } a = b \implies \text{thesis } b < a \implies \text{thesis}$
shows *thesis*
 ⟨proof⟩

lemma *nonpos-linorder-cases* [consumes 2, case-names less eq greater]:
assumes $a \leq 0$ **and** $b \leq 0$
and $a < b \implies \text{thesis } a = b \implies \text{thesis } b < a \implies \text{thesis}$
shows *thesis*
 ⟨proof⟩

lemma *real-linear*:
assumes *is-real* a **and** *is-real* b **shows** $a < b \vee a = b \vee b < a$
 ⟨proof⟩

lemma *real-linorder-cases* [consumes 2, case-names less eq greater]:
assumes *real*: *is-real* a *is-real* b
and *cases*: $a < b \implies \text{thesis } a = b \implies \text{thesis } b < a \implies \text{thesis}$
shows *thesis*

<proof>

lemma

assumes *a: is-real a and b: is-real b*

shows *real-add-le-cancel-left-pos: $c + a \leq c + b \iff a \leq b$*

and *real-add-less-cancel-left-pos: $c + a < c + b \iff a < b$*

and *real-add-le-cancel-right-pos: $a + c \leq b + c \iff a \leq b$*

and *real-add-less-cancel-right-pos: $a + c < b + c \iff a < b$*

<proof>

lemma

assumes *a: is-real a and b: is-real b and c: $0 < c$*

shows *real-mult-le-cancel-left-pos: $c * a \leq c * b \iff a \leq b$*

and *real-mult-less-cancel-left-pos: $c * a < c * b \iff a < b$*

and *real-mult-le-cancel-right-pos: $a * c \leq b * c \iff a \leq b$*

and *real-mult-less-cancel-right-pos: $a * c < b * c \iff a < b$*

<proof>

lemma

assumes *a: is-real a and b: is-real b*

shows *not-le-real: $\neg a \geq b \iff a < b$*

and *not-less-real: $\neg a > b \iff a \leq b$*

<proof>

lemma *real-mult-eq-0-iff:*

assumes *a: is-real a and b: is-real b*

shows *$a * b = 0 \iff a = 0 \vee b = 0$*

<proof>

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 *$c * \max a b = \max (c * a) (c * b)$*

<proof>

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 <proof>*

lemma *power-both-mono: $1 \leq a \implies m \leq n \implies a \leq b \implies a^m \leq b^n$*

<proof>

lemma *power-pos:*

assumes *a0: $0 < a$ shows $0 < a^n$*

```

    <proof>

lemma power-neg:
  assumes  $a0$ :  $a < 0$  shows  $odd\ n \implies a^n < 0$  and  $even\ n \implies a^n > 0$ 
  <proof>

lemma power-ge-0-iff:
  assumes  $a$ : is-real  $a$ 
  shows  $0 \leq a^n \iff 0 \leq a \vee even\ n$ 
  <proof>

lemma nonneg-power-less:
  assumes  $0 \leq a$  and  $0 \leq b$  shows  $a^n < b^n \iff n > 0 \wedge a < b$ 
  <proof>

lemma power-strict-mono:
  shows  $a < b \implies 0 \leq a \implies 0 < n \implies a^n < b^n$ 
  <proof>

lemma nonneg-power-le:
  assumes  $0 \leq a$  and  $0 \leq b$  shows  $a^n \leq b^n \iff n = 0 \vee a \leq b$ 
  <proof>

end

subclass (in linordered-idom) semiring-1-real-line
  <proof>

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

subclass ring-1 <proof>

lemma abs-cases:
  assumes  $a = 0 \implies thesis$  and  $|a| > 0 \implies thesis$  shows  $thesis$ 
  <proof>

lemma abs-linorder-cases[case-names less eq greater]:
  assumes  $|a| < |b| \implies thesis$  and  $|a| = |b| \implies thesis$  and  $|b| < |a| \implies thesis$ 
  shows  $thesis$ 
  <proof>

lemma [simp]:
  shows not-le-abs-abs:  $\neg |a| \geq |b| \iff |a| < |b|$ 
  and not-less-abs-abs:  $\neg |a| > |b| \iff |a| \leq |b|$ 
  <proof>

lemma abs-power-less [simp]:  $|a|^n < |b|^n \iff n > 0 \wedge |a| < |b|$ 
  <proof>

```

lemma *abs-power-le* [*simp*]: $|a|^{\widehat{n}} \leq |b|^{\widehat{n}} \longleftrightarrow n = 0 \vee |a| \leq |b|$
<proof>

lemma *abs-power-pos* [*simp*]: $|a|^{\widehat{n}} > 0 \longleftrightarrow a \neq 0 \vee n = 0$
<proof>

lemma *abs-power-nonneg* [*intro!*]: $|a|^{\widehat{n}} \geq 0$ *<proof>*

lemma *abs-power-eq-0* [*simp*]: $|a|^{\widehat{n}} = 0 \longleftrightarrow a = 0 \wedge n \neq 0$
<proof>

end

instance *nat* :: *semiring-1-real-line* *<proof>*

instance *int* :: *ring-1-abs-real-line* *<proof>*

lemma *vec-index-vec-of-list* [*simp*]: $\text{vec-of-list } xs \ \$ \ i = xs \ ! \ i$
<proof>

lemma *vec-of-list-append*: $\text{vec-of-list } (xs \ @ \ ys) = \text{vec-of-list } xs \ @_v \ \text{vec-of-list } ys$
<proof>

lemma *linf-norm-vec-of-list*:
 $\|\text{vec-of-list } xs\|_{\infty} = \text{max-list } (\text{map } \text{abs } xs \ @ \ [0])$
<proof>

lemma *linf-norm-vec-as-Greatest*:
fixes $v :: 'a :: \text{ring-1-abs-real-line } \text{vec}$
shows $\|v\|_{\infty} = (\text{GREATEST } a. a \in \text{abs } ' \text{set } (\text{list-of-vec } v) \cup \{0\})$
<proof>

lemma *vec-of-poly-pCons*:
assumes $f \neq 0$
shows $\text{vec-of-poly } (\text{pCons } a \ f) = \text{vec-of-poly } f \ @_v \ \text{vec-of-list } [a]$
<proof>

lemma *vec-of-poly-as-vec-of-list*:
assumes $f \neq 0$
shows $\text{vec-of-poly } f = \text{vec-of-list } (\text{rev } (\text{coeffs } f))$
<proof>

lemma *linf-norm-vec-of-poly* [*simp*]:
fixes $f :: 'a :: \text{ring-1-abs-real-line } \text{poly}$
shows $\|\text{vec-of-poly } f\|_{\infty} = \|f\|_{\infty}$
<proof>

lemma *linf-norm-poly-as-Greatest*:
fixes $f :: 'a :: \text{ring-1-abs-real-line } \text{poly}$

shows $\|f\|_\infty = (\text{GREATEST } a. a \in \text{abs } ' \text{set } (\text{coeffs } f) \cup \{0\})$
 <proof>

lemma *vec-index-le-linf-norm*:
fixes $v :: 'a :: \text{ring-1-abs-real-line } \text{vec}$
assumes $i < \text{dim-vec } v$
shows $|v\$i| \leq \|v\|_\infty$
 <proof>

lemma *coeff-le-linf-norm*:
fixes $f :: 'a :: \text{ring-1-abs-real-line } \text{poly}$
shows $|\text{coeff } f \ i| \leq \|f\|_\infty$
 <proof>

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* <proof>
end

instance *int* :: *conjugatable-ring-1-abs-real-line*
 <proof>

instance *rat* :: *conjugatable-ring-1-abs-real-line*
 <proof>

instance *real* :: *conjugatable-ring-1-abs-real-line*
 <proof>

instance *complex* :: *semiring-1-real-line*
 <proof>

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* <proof>

lemma *sq-norm-as-sq-abs* [*simp*]: $(\text{sq-norm} :: 'a :: \text{conjugatable-ring-1-abs-real-line}$
 $\Rightarrow 'a) = \text{power2} \circ \text{abs}$
 <proof>

lemma *sq-norm-vec-le-linf-norm*:
fixes $v :: 'a :: \{\text{conjugatable-ring-1-abs-real-line}\} \text{vec}$
assumes $v \in \text{carrier-vec } n$
shows $\|v\|^2 \leq \text{of-nat } n * \|v\|_\infty^2$
 <proof>

lemma *sq-norm-poly-le-linf-norm*:
fixes $p :: 'a :: \{\text{conjugatable-ring-1-abs-real-line}\} \text{poly}$

shows $\|p\|^2 \leq \text{of-nat } (\text{degree } p + 1) * \|p\|_\infty^2$
 ⟨proof⟩

lemma *coeff-le-sq-norm*:
fixes $f :: 'a :: \{\text{conjugatable-ring-1-abs-real-line}\}$ *poly*
shows $|\text{coeff } f \ i|^2 \leq \|f\|^2$
 ⟨proof⟩

lemma *max-norm-witness*:
fixes $f :: 'a :: \text{ordered-ring-abs poly}$
shows $\exists i. \|f\|_\infty = |\text{coeff } f \ i|$
 ⟨proof⟩

lemma *max-norm-le-sq-norm*:
fixes $f :: 'a :: \text{conjugatable-ring-1-abs-real-line poly}$
shows $\|f\|_\infty^2 \leq \|f\|^2$
 ⟨proof⟩

lemma (in *conjugatable-ring*) *conjugate-minus*: $\text{conjugate } (x - y) = \text{conjugate } x - \text{conjugate } y$
 ⟨proof⟩

lemma *conjugate-1[simp]*: $(\text{conjugate } 1 :: 'a :: \{\text{conjugatable-ring, ring-1}\}) = 1$
 ⟨proof⟩

lemma *conjugate-of-int[simp]*:
 $(\text{conjugate } (\text{of-int } x) :: 'a :: \{\text{conjugatable-ring, ring-1}\}) = \text{of-int } x$
 ⟨proof⟩

lemma *sq-norm-of-int*: $\|\text{map-vec of-int } v :: 'a :: \{\text{conjugatable-ring, ring-1}\} \text{vec}\|^2 = \text{of-int } \|v\|^2$
 ⟨proof⟩

definition *norm1* $p = \text{sum-list } (\text{map abs } (\text{coeffs } p))$

lemma *norm1-ge-0*: $\text{norm1 } (f :: 'a :: \{\text{abs, ordered-semiring-0, ordered-ab-group-add-abs}\} \text{poly}) \geq 0$
 ⟨proof⟩

lemma *norm2-norm1-main-equality*: **fixes** $f :: \text{nat} \Rightarrow 'a :: \text{linordered-idom}$
shows $(\sum i = 0..<n. |f \ i|^2) = (\sum i = 0..<n. f \ i * f \ i) + (\sum i = 0..<n. \sum j = 0..<n. \text{if } i = j \text{ then } 0 \text{ else } |f \ i| * |f \ j|)$
 ⟨proof⟩

lemma *norm2-norm1-main-inequality*: **fixes** $f :: \text{nat} \Rightarrow 'a :: \text{linordered-idom}$
shows $(\sum i = 0..<n. f \ i * f \ i) \leq (\sum i = 0..<n. |f \ i|^2)$
 ⟨proof⟩

```

lemma norm2-le-norm1-int:  $\|f :: \text{int poly}\|^2 \leq (\text{norm1 } f)^2$ 
<proof>

lemma sq-norm-smult-vec:  $\text{sq-norm } ((c :: 'a :: \{\text{conjugatable-ring, comm-semiring-0}\}) \cdot_v v) = (c * \text{conjugate } c) * \text{sq-norm } v$ 
<proof>

lemma vec-le-sq-norm:
  fixes  $v :: 'a :: \text{conjugatable-ring-1-abs-real-line } \text{vec}$ 
  assumes  $v \in \text{carrier-vec } n \ i < n$ 
  shows  $|v \ \$ \ i|^2 \leq \|v\|^2$ 
<proof>

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
  <proof>
end

instance rat :: trivial-conjugatable-linordered-field
  <proof>

instance real :: trivial-conjugatable-linordered-field
  <proof>

lemma scalar-prod-ge-0:  $(x :: 'a :: \text{linordered-idom } \text{vec}) \cdot x \geq 0$ 
  <proof>

lemma cscalar-prod-is-scalar-prod[simp]:  $(x :: 'a :: \text{trivial-conjugatable-ordered-field } \text{vec}) \cdot c \ y = x \cdot y$ 
  <proof>

lemma scalar-prod-Cauchy:
  fixes  $u \ v :: 'a :: \{\text{trivial-conjugatable-linordered-field}\} \text{Matrix.vec}$ 
  assumes  $u \in \text{carrier-vec } n \ v \in \text{carrier-vec } n$ 
  shows  $(u \cdot v)^2 \leq \|u\|^2 * \|v\|^2$ 
  <proof>

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  $\Rightarrow$  rat  $\Rightarrow$  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)  $\Rightarrow$  Rat.normalize (i * n, d))
   $\langle$ proof $\rangle$ 

definition square-rat :: rat  $\Rightarrow$  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)
   $\langle$ proof $\rangle$ 

lemma square-rat-code[code abstract]: quotient-of (square-rat x) = (case quotient-of
x of (n,d)
   $\Rightarrow$  (n * n, d * d))  $\langle$ proof $\rangle$ 

definition scalar-prod-int-rat :: int vec  $\Rightarrow$  rat vec  $\Rightarrow$  rat (infix  $\cdot$  i 70) where
  x  $\cdot$  i y = (y  $\cdot$  map-vec rat-of-int x)

lemma scalar-prod-int-rat-code[code]: v  $\cdot$  i w = ( $\sum$  i = 0..dim-vec v. int-times-rat
(v $ i) (w $ i))
   $\langle$ proof $\rangle$ 

lemma scalar-prod-int-rat[simp]: dim-vec x = dim-vec y  $\implies$  x  $\cdot$  i y = map-vec
of-int x  $\cdot$  y
   $\langle$ proof $\rangle$ 

definition sq-norm-vec-rat :: rat vec  $\Rightarrow$  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  $\leftarrow$  list-of-vec x. square-rat
x)
   $\langle$ proof $\rangle$ 

end

```


6 Representing Computation Costs as Pairs of Results and Costs

```

theory Cost
  imports Main
begin

type-synonym 'a cost = 'a × 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
  ⟨proof⟩

lemma result-costD: assumes result f-c = f
  cost f-c ≤ b
  f-c = (a,c)
shows a = f c ≤ b ⟨proof⟩

lemma result-costD': assumes result f-c = f ∧ cost f-c ≤ b
  f-c = (a,c)
  shows a = f c ≤ b ⟨proof⟩

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⟩

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

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

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

lemma of-list-repr: list-repr i ba xs ⇒ of-list-repr ba = xs
  ⟨proof⟩

```

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

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

lemma *get-nth-i*: list-repr i ba xs \Longrightarrow $i < \text{length } xs \Longrightarrow$ *get-nth-i* ba = xs ! i
(proof)

lemma *get-nth-im1*: list-repr i ba xs \Longrightarrow $i \neq 0 \Longrightarrow$ *get-nth-im1* ba = xs ! (i - 1)
(proof)

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

lemma *Cons-tl-drop-update*: $i < \text{length } xs \Longrightarrow$ $x \# \text{tl } (\text{drop } i \text{ } xs) = \text{drop } i \text{ } (xs[i := x])$
(proof)

lemma *update-i*: list-repr i ba xs \Longrightarrow $i < \text{length } xs \Longrightarrow$ list-repr i (*update-i* ba x)
(xs [i := x])
(proof)

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])
(proof)

lemma *tl-drop-Suc*: $\text{tl } (\text{drop } i \text{ } xs) = \text{drop } (\text{Suc } i) \text{ } xs$
(proof)

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 $i < \text{length } xs \Longrightarrow$ list-repr (Suc i) (*inc-i* ba) xs
(proof)

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
(proof)

lemma *dec-i-Suc*: list-repr (Suc i) ba xs \Longrightarrow list-repr i (*dec-i* ba) xs
(proof)

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

no-notation *Group.m-inv* (*inv1* - [81] 80)

fun *find-index* :: 'b list \Rightarrow 'b \Rightarrow nat **where**

find-index [] - = 0 |

find-index (x#xs) y = (if x = y then 0 else *find-index* xs y + 1)

lemma *find-index-not-in-set*: $x \notin \text{set } xs \longleftrightarrow \text{find-index } xs \ x = \text{length } xs$
<proof>

lemma *find-index-in-set*: $x \in \text{set } xs \implies xs \ ! (\text{find-index } xs \ x) = x$
<proof>

lemma *find-index-inj*: *inj-on* (*find-index* xs) (set xs)
<proof>

lemma *find-index-leq-length*: $\text{find-index } xs \ x < \text{length } xs \longleftrightarrow x \in \text{set } xs$
<proof>

lemma *rev-unsimp*: $\text{rev } xs \ @ (r \ # \ rs) = \text{rev } (r \ # \ xs) \ @ \ rs$ <proof>

lemma *corthogonal-is-orthogonal[simp]*:

corthogonal (xs :: 'a :: trivial-conjugatable-ordered-field vec list) = *orthogonal* xs

<proof>

context *vec-module* **begin**

definition *lattice-of* :: 'a vec list \Rightarrow 'a vec set **where**
lattice-of fs = range (λ c. *sumlist* (*map* (λ i. *of-int* (c i) \cdot_v fs ! i) [0 ..< length fs]))

lemma *lattice-of-finsum*:
assumes set fs \subseteq carrier-vec n
shows *lattice-of fs* = range (λ c. *finsum* V (λ i. *of-int* (c i) \cdot_v fs ! i) {0 ..< length fs})
 <proof>

lemma *in-latticeE*: **assumes** $f \in$ *lattice-of fs* **obtains** c **where**
 $f =$ *sumlist* (*map* (λ i. *of-int* (c i) \cdot_v fs ! i) [0 ..< length fs])
 <proof>

lemma *in-latticeI*: **assumes** $f =$ *sumlist* (*map* (λ i. *of-int* (c i) \cdot_v fs ! i) [0 ..< length fs])
shows $f \in$ *lattice-of fs*
 <proof>

lemma *finsum-over-indexes-to-vectors*:
assumes set vs \subseteq carrier-vec n $l =$ length vs
shows \exists c. ($\bigoplus_{v \in \{0..<l\}}$ *of-int* (g x) \cdot_v vs ! x) = ($\bigoplus_{v \in \text{set vs}}$ *of-int* (c v) \cdot_v v)
 <proof>

lemma *lattice-of-altdef*:
assumes set vs \subseteq carrier-vec n
shows *lattice-of vs* = range (λ c. $\bigoplus_{v \in \text{set vs}}$ *of-int* (c v) \cdot_v v)
 <proof>

lemma *basis-in-latticeI*:
assumes fs: set fs \subseteq carrier-vec n **and** $f \in$ set fs
shows $f \in$ *lattice-of fs*
 <proof>

lemma *lattice-of-eq-set*:
assumes set fs = set gs set fs \subseteq carrier-vec n
shows *lattice-of fs* = *lattice-of gs*
 <proof>

lemma *lattice-of-swap*: **assumes** fs: set fs \subseteq carrier-vec n
and ij : $i <$ length fs $j <$ length fs $i \neq j$
and gs: gs = fs[i := fs ! j, j := fs ! i]
shows *lattice-of gs* = *lattice-of fs*
 <proof>

lemma *lattice-of-add*: **assumes** fs: set fs \subseteq carrier-vec n
and ij : $i <$ length fs $j <$ length fs $i \neq j$
and gs: gs = fs[i := fs ! i + *of-int* l \cdot_v fs ! j]

shows $\text{lattice-of } gs = \text{lattice-of } fs$
 $\langle \text{proof} \rangle$

definition $\text{orthogonal-complement } W = \{x. x \in \text{carrier-vec } n \wedge (\forall y \in W. x \cdot y = 0)\}$

lemma $\text{orthogonal-complement-subset}$:
 assumes $A \subseteq B$
 shows $\text{orthogonal-complement } B \subseteq \text{orthogonal-complement } A$
 $\langle \text{proof} \rangle$

end

context vec-space
begin

lemma $\text{in-orthogonal-complement-span[simp]}$:
 assumes $[\text{intro}]: S \subseteq \text{carrier-vec } n$
 shows $\text{orthogonal-complement } (\text{span } S) = \text{orthogonal-complement } S$
 $\langle \text{proof} \rangle$

end

context cof-vec-space
begin

definition $\text{lin-indpt-list} :: 'a \text{ vec list} \Rightarrow \text{bool}$ **where**
 $\text{lin-indpt-list } fs = (\text{set } fs \subseteq \text{carrier-vec } n \wedge \text{distinct } fs \wedge \text{lin-indpt } (\text{set } fs))$

definition $\text{basis-list} :: 'a \text{ vec list} \Rightarrow \text{bool}$ **where**
 $\text{basis-list } fs = (\text{set } fs \subseteq \text{carrier-vec } n \wedge \text{length } fs = n \wedge \text{carrier-vec } n \subseteq \text{span } (\text{set } fs))$

lemma $\text{upper-triangular-imp-lin-indpt-list}$:
 assumes $A: A \in \text{carrier-mat } n \ n$
 and $\text{tri}: \text{upper-triangular } A$
 and $\text{diag}: 0 \notin \text{set } (\text{diag-mat } A)$
 shows $\text{lin-indpt-list } (\text{rows } A)$
 $\langle \text{proof} \rangle$

lemma basis-list-basis : **assumes** $\text{basis-list } fs$
 shows $\text{distinct } fs \ \text{lin-indpt } (\text{set } fs) \ \text{basis } (\text{set } fs)$
 $\langle \text{proof} \rangle$

lemma $\text{basis-list-imp-lin-indpt-list}$: **assumes** $\text{basis-list } fs$ **shows** $\text{lin-indpt-list } fs$
 $\langle \text{proof} \rangle$

lemma basis-det-nonzero :

assumes *db:basis* (set G) **and** *len:length* $G = n$
shows $\det (\text{mat-of-rows } n \ G) \neq 0$
 ⟨*proof*⟩

lemma *lin-indpt-list-add-vec*: **assumes**
 i: $j < \text{length } us \ i < \text{length } us \ i \neq j$
and *indep*: *lin-indpt-list* us
shows *lin-indpt-list* ($us \ [i := us \ ! \ i + c \cdot_v \ us \ ! \ j]$) (**is** *lin-indpt-list* ? V)
 ⟨*proof*⟩

lemma *scalar-prod-lincomb-orthogonal*: **assumes** *ortho*: *orthogonal* gs **and** gs : set
 $gs \subseteq \text{carrier-vec } n$
shows $k \leq \text{length } gs \implies \text{sumlist } (\text{map } (\lambda \ i. \ g \ i \cdot_v \ gs \ ! \ i) \ [0 \ ..< \ k]) \cdot \text{sumlist}$
 $(\text{map } (\lambda \ i. \ h \ i \cdot_v \ gs \ ! \ i) \ [0 \ ..< \ k])$
 $= \text{sum-list } (\text{map } (\lambda \ i. \ g \ i * h \ i * (gs \ ! \ i \cdot gs \ ! \ i)) \ [0 \ ..< \ k])$
 ⟨*proof*⟩
end

locale *gram-schmidt* = *cof-vec-space* $n \ f\text{-ty}$
for $n :: \text{nat}$ **and** $f\text{-ty} :: 'a :: \{\text{trivial-conjugatable-linordered-field}\}$ *itself*
begin

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

lemma *Gramian-matrix-alt-def*: $k \leq \text{length } G \implies$
Gramian-matrix $G \ k = (\text{let } M = \text{mat-of-rows } n \ (\text{take } k \ G) \ \text{in } M * M^T)$
 ⟨*proof*⟩

definition *Gramian-determinant* **where**
Gramian-determinant $G \ k = \det (\text{Gramian-matrix } G \ k)$

lemma *Gramian-determinant-0* [*simp*]: *Gramian-determinant* $G \ 0 = 1$
 ⟨*proof*⟩

lemma *orthogonal-imp-lin-indpt-list*:
assumes *ortho*: *orthogonal* gs **and** gs : set $gs \subseteq \text{carrier-vec } n$
shows *lin-indpt-list* gs
 ⟨*proof*⟩

lemma *orthocompl-span*:
assumes $\bigwedge x. x \in S \implies v \cdot x = 0$ $S \subseteq \text{carrier-vec } n$ **and** [*intro*]: $v \in \text{carrier-vec } n$
and $y \in \text{span } S$
shows $v \cdot y = 0$
 ⟨*proof*⟩

lemma *orthogonal-sumlist*:

assumes *ortho*: $\bigwedge x. x \in \text{set } S \implies v \cdot x = 0$ **and** *S*: $\text{set } S \subseteq \text{carrier-vec } n$ **and**
v: $v \in \text{carrier-vec } n$
shows $v \cdot \text{sumlist } S = 0$
 $\langle \text{proof} \rangle$

lemma *oc-projection-alt-def*:
assumes *carr*: $(W :: 'a \text{ vec set}) \subseteq \text{carrier-vec } n$ $x \in \text{carrier-vec } n$
and *alt1*: $y1 \in W$ $x - y1 \in \text{orthogonal-complement } W$
and *alt2*: $y2 \in W$ $x - y2 \in \text{orthogonal-complement } W$
shows $y1 = y2$
 $\langle \text{proof} \rangle$

definition
is-oc-projection $w S v = (w \in \text{carrier-vec } n \wedge v - w \in \text{span } S \wedge (\forall u. u \in S \longrightarrow w \cdot u = 0))$

lemma *is-oc-projection-sq-norm*: **assumes** *is-oc-projection* $w S v$
and *S*: $S \subseteq \text{carrier-vec } n$
and *v*: $v \in \text{carrier-vec } n$
shows $\text{sq-norm } w \leq \text{sq-norm } v$
 $\langle \text{proof} \rangle$

definition *oc-projection where*
oc-projection $S fi \equiv (\text{SOME } v. \text{is-oc-projection } v S fi)$

lemma *inv-in-span*:
assumes *incarr*[*intro*]: $U \subseteq \text{carrier-vec } n$ **and** *insp*: $a \in \text{span } U$
shows $-a \in \text{span } U$
 $\langle \text{proof} \rangle$

lemma *non-span-det-zero*:
assumes *len*: $\text{length } G = n$
and *nonb*: $\neg (\text{carrier-vec } n \subseteq \text{span } (\text{set } G))$
and *carr*: $\text{set } G \subseteq \text{carrier-vec } n$
shows $\text{det } (\text{mat-of-rows } n G) = 0$ $\langle \text{proof} \rangle$

lemma *span-basis-det-zero-iff*:
assumes $\text{length } G = n$ $\text{set } G \subseteq \text{carrier-vec } n$
shows $\text{carrier-vec } n \subseteq \text{span } (\text{set } G) \iff \text{det } (\text{mat-of-rows } n G) \neq 0$ (**is** ?*q1*)
 $\text{carrier-vec } n \subseteq \text{span } (\text{set } G) \iff \text{basis } (\text{set } G)$ (**is** ?*q2*)
 $\text{det } (\text{mat-of-rows } n G) \neq 0 \iff \text{basis } (\text{set } G)$ (**is** ?*q3*)
 $\langle \text{proof} \rangle$

lemma *lin-indpt-list-nonzero*:
assumes *lin-indpt-list* G
shows $0_v \notin \text{set } G$
 $\langle \text{proof} \rangle$

lemma *is-oc-projection-eq*:

```

assumes ispr:is-oc-projection a S v is-oc-projection b S v
and carr: S  $\subseteq$  carrier-vec n v  $\in$  carrier-vec n
shows a = b
<proof>

```

```

fun adjuster-wit :: 'a list  $\Rightarrow$  'a vec  $\Rightarrow$  'a vec list  $\Rightarrow$  'a list  $\times$  'a vec
where adjuster-wit wits w [] = (wits, 0_v n)
| adjuster-wit wits w (u#us) = (let a = (w  $\cdot$  u) / sq-norm u in
  case adjuster-wit (a # wits) w us of (wit, v)
   $\Rightarrow$  (wit, -a  $\cdot$ _v u + v))

```

```

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

```

```

definition main :: 'a vec list  $\Rightarrow$  'a list list  $\times$  '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) <proof>

```

```

fun gso and  $\mu$  where
  gso i = fs ! i + sumlist (map ( $\lambda$  j. -  $\mu$  i j  $\cdot$ _v gso j) [0 ..< i])
|  $\mu$  i j = (if j < i then (fs ! i  $\cdot$  gso j) / sq-norm (gso j) else if i = j then 1 else 0)

```

```

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

```

```

lemma gso-carrier'[intro]:
  assumes  $\bigwedge$  i. i  $\leq$  j  $\Longrightarrow$  fs ! i  $\in$  carrier-vec n
  shows gso j  $\in$  carrier-vec n
<proof>

```

```

lemma adjuster-wit: assumes res: adjuster-wit wits w us = (wits',a)
and w: w  $\in$  carrier-vec n
  and us:  $\bigwedge$  i. i  $\leq$  j  $\Longrightarrow$  fs ! i  $\in$  carrier-vec n
  and us-gs: us = map gso (rev [0 ..< j])
  and wits: wits = map ( $\mu$  i) [j ..< i]
  and j: j  $\leq$  n j  $\leq$  i

```


and $wi: w = fs ! i$
shows $adjuster\ n\ w\ us = a \wedge a \in carrier\text{-}vec\ n \wedge wits' = map\ (\mu\ i)\ [0\ ..< i] \wedge$
 $(a = sumlist\ (map\ (\lambda j. -\ \mu\ i\ j\ \cdot_v\ gso\ j)\ [0..<j]))$
 $\langle proof \rangle$

lemma *sub2-wit*:
assumes $set\ us \subseteq carrier\text{-}vec\ n\ set\ ws \subseteq carrier\text{-}vec\ n\ length\ us + length\ ws =$
 m
and $ws = map\ (\lambda\ i.\ fs\ !\ i)\ [i\ ..< m]$
and $us = map\ gso\ (rev\ [0\ ..< i])$
and $us: \bigwedge j. j < m \implies fs\ !\ j \in carrier\text{-}vec\ n$
and $mn: m \leq n$
shows $sub2\text{-}wit\ us\ ws = (wits, vvs) \implies gram\text{-}schmidt\text{-}sub2\ n\ us\ ws = vvs$
 $\wedge vvs = map\ gso\ [i\ ..< m] \wedge wits = map\ (\lambda\ i.\ map\ (\mu\ i)\ [0..<i])\ [i\ ..< m]$
 $\langle proof \rangle$

lemma *partial-connect*: **fixes** vs
assumes $length\ fs = m\ k \leq m\ m \leq n\ set\ us \subseteq carrier\text{-}vec\ n\ snd\ (main\ us) = vs$
 $us = take\ k\ fs\ set\ fs \subseteq carrier\text{-}vec\ n$
shows $gram\text{-}schmidt\ n\ us = vs$
 $vs = map\ gso\ [0..<k]$
 $\langle proof \rangle$

lemma *adjuster-wit-small*:
 $(adjuster\text{-}wit\ v\ a\ xs) = (x1, x2)$
 $\longleftrightarrow (fst\ (adjuster\text{-}wit\ v\ a\ xs) = x1 \wedge x2 = adjuster\ n\ a\ xs)$
 $\langle proof \rangle$

lemma *sub2*: $rev\ xs\ @\ snd\ (sub2\text{-}wit\ xs\ us) = rev\ (gram\text{-}schmidt\text{-}sub\ n\ xs\ us)$
 $\langle proof \rangle$

lemma *gso-connect*: $snd\ (main\ us) = gram\text{-}schmidt\ n\ us\ \langle proof \rangle$

definition *weakly-reduced* $:: 'a \Rightarrow nat \Rightarrow bool$
where $weakly\text{-}reduced\ \alpha\ k = (\forall\ i.\ Suc\ i < k \longrightarrow$
 $sq\text{-}norm\ (gso\ i) \leq \alpha * sq\text{-}norm\ (gso\ (Suc\ i)))$

definition *reduced* $:: 'a \Rightarrow nat \Rightarrow bool$
where $reduced\ \alpha\ k = (weakly\text{-}reduced\ \alpha\ k \wedge$
 $(\forall\ i\ j.\ i < k \longrightarrow j < i \longrightarrow abs\ (\mu\ i\ j) \leq 1/2))$

end

locale *gram-schmidt-fs-Rn* = $gram\text{-}schmidt\text{-}fs +$

assumes *fs-carrier*: $set\ fs \subseteq carrier\text{-}vec\ n$
begin

abbreviation (*input*) *m* **where** $m \equiv length\ fs$

definition *M* **where** $M\ k = mat\ k\ k\ (\lambda\ (i,j). \mu\ i\ j)$

lemma *f-carrier[simp]*: $i < m \implies fs\ !\ i \in carrier\text{-}vec\ n$
<proof>

lemma *gso-carrier[simp]*: $i < m \implies gso\ i \in carrier\text{-}vec\ n$
<proof>

lemma *gso-dim[simp]*: $i < m \implies dim\text{-}vec\ (gso\ i) = n$ *<proof>*

lemma *f-dim[simp]*: $i < m \implies dim\text{-}vec\ (fs\ !\ i) = n$ *<proof>*

lemma *fs0-gso0*: $0 < m \implies fs\ !\ 0 = gso\ 0$
<proof>

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>

lemma *main-connect*:

assumes $m \leq n$

shows $gram\text{-}schmidt\ n\ fs = map\ gso\ [0..<m]$
<proof>

lemma *reduced-gso-E*: $weakly\text{-}reduced\ \alpha\ k \implies k \leq m \implies Suc\ i < k \implies$
 $sq\text{-}norm\ (gso\ i) \leq \alpha * sq\text{-}norm\ (gso\ (Suc\ i))$
<proof>

abbreviation (*input*) *FF* **where** $FF \equiv mat\text{-}of\text{-}rows\ n\ fs$

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

lemma *FF-dim[simp]*: $dim\text{-}row\ FF = m\ dim\text{-}col\ FF = n\ FF \in carrier\text{-}mat\ m\ n$
<proof>

lemma *Fs-dim[simp]*: $dim\text{-}row\ Fs = m\ dim\text{-}col\ Fs = n\ Fs \in carrier\text{-}mat\ m\ n$
<proof>

lemma *M-dim[simp]*: $dim\text{-}row\ (M\ m) = m\ dim\text{-}col\ (M\ m) = m\ (M\ m) \in carrier\text{-}mat\ m\ m$
<proof>

lemma *FF-index[simp]*: $i < m \implies j < n \implies FF \ \$\$ (i,j) = fs ! i \$ j$
 ⟨proof⟩

lemma *M-index[simp]*: $i < m \implies j < m \implies (M \ m) \ \$\$ (i,j) = \mu \ i \ j$
 ⟨proof⟩

lemma *matrix-equality*: $FF = (M \ m) * Fs$
 ⟨proof⟩

lemma *fi-is-sum-of-mu-gso*: **assumes** $i: i < m$
shows $fs ! i = \text{sumlist } (\text{map } (\lambda j. \mu \ i \ j \cdot_v \text{gso } j) [0 ..< \text{Suc } i])$
 ⟨proof⟩

lemma *gi-is-fi-minus-sum-mu-gso*:
assumes $i: i < m$
shows $\text{gso } i = fs ! i - \text{sumlist } (\text{map } (\lambda j. \mu \ i \ j \cdot_v \text{gso } j) [0 ..< i])$ (**is - = -**
 ?sum)
 ⟨proof⟩

lemma *det*: **assumes** $m: m = n$ **shows** $\text{det } FF = \text{det } Fs$
 ⟨proof⟩
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 $m \leq n$
 ⟨proof⟩

lemma
shows *span-gso*: $\text{span } (\text{gso } \{0..< m\}) = \text{span } (\text{set } fs)$
and *orthogonal-gso*: $\text{orthogonal } (\text{map } \text{gso } [0..< m])$
and *dist-gso*: $\text{distinct } (\text{map } \text{gso } [0..< m])$
 ⟨proof⟩

lemma *gso-inj[intro]*:
assumes $i < m$
shows *inj-on* *gso* $\{0..< i\}$
 ⟨proof⟩

lemma *partial-span*:
assumes $i: i \leq m$
shows $\text{span } (\text{gso } \{0 ..< i\}) = \text{span } (\text{set } (\text{take } i \text{ fs}))$

$\langle proof \rangle$

lemma *partial-span'*:

assumes $i \leq m$

shows $span (gso \text{ ' } \{0 ..< i\}) = span ((\lambda j. fs ! j) \text{ ' } \{0 ..< i\})$

$\langle proof \rangle$

lemma *orthogonal*:

assumes $i < m \ j < m \ i \neq j$

shows $gso \ i \cdot gso \ j = 0$

$\langle proof \rangle$

lemma *same-base*:

shows $span (set \ fs) = span (gso \text{ ' } \{0..<m\})$

$\langle proof \rangle$

lemma *sq-norm-gso-le-f*:

assumes $i < m$

shows $sq\text{-norm} (gso \ i) \leq sq\text{-norm} (fs \ ! \ i)$

$\langle proof \rangle$

lemma *oc-projection-exist*:

assumes $i < m$

shows $fs \ ! \ i - gso \ i \in span (gso \text{ ' } \{0..<i\})$

$\langle proof \rangle$

lemma *oc-projection-unique*:

assumes $i < m$

$v \in carrier\text{-vec} \ n$

$\bigwedge x. x \in gso \text{ ' } \{0..<i\} \implies v \cdot x = 0$

$fs \ ! \ i - v \in span (gso \text{ ' } \{0..<i\})$

shows $v = gso \ i$

$\langle proof \rangle$

lemma *gso-oc-projection*:

assumes $i < m$

shows $gso \ i = oc\text{-projection} (gso \text{ ' } \{0..<i\}) (fs \ ! \ i)$

$\langle proof \rangle$

lemma *gso-oc-projection-span*:

assumes $i < m$

shows $gso \ i = oc\text{-projection} (span (gso \text{ ' } \{0..<i\})) (fs \ ! \ i)$

and $is\text{-oc-projection} (gso \ i) (span (gso \text{ ' } \{0..<i\})) (fs \ ! \ i)$

$\langle proof \rangle$

lemma *gso-is-oc-projection*:

assumes $i < m$

shows *is-oc-projection* (*gso* i) (*set* (*take* i *fs*)) (*fs* ! i)

<proof>

lemma *fi-scalar-prod-gso*:

assumes $i: i < m$ **and** $j: j < m$

shows $fs ! i \cdot gso j = \mu i j * \|gso j\|^2$

<proof>

lemma *gso-scalar-zero*:

assumes $k < m$ $i < k$

shows $(gso k) \cdot (fs ! i) = 0$

<proof>

lemma *scalar-prod-lincomb-gso*:

assumes $k: k \leq m$

shows $sumlist (map (\lambda i. g i \cdot_v gso i) [0 ..< k]) \cdot sumlist (map (\lambda i. h i \cdot_v gso i) [0 ..< k])$

$= sum-list (map (\lambda i. g i * h i * (gso i \cdot gso i)) [0 ..< k])$

<proof>

lemma *gso-times-self-is-norm*:

assumes $j < m$

shows $fs ! j \cdot gso j = sq-norm (gso j)$

<proof>

lemma *gram-schmidt-short-vector*:

assumes *in-L*: $h \in lattice-of fs - \{0_v n\}$

shows $\exists i < m. \|h\|^2 \geq \|gso i\|^2$

<proof>

lemma *weakly-reduced-imp-short-vector*:

assumes *weakly-reduced* αm

and *in-L*: $h \in lattice-of fs - \{0_v n\}$ **and** $\alpha-pos:\alpha \geq 1$

shows $fs \neq [] \wedge sq-norm (fs ! 0) \leq \alpha^{m-1} * sq-norm h$

<proof>

lemma *sq-norm-pos*:

assumes $j: j < m$

shows $sq-norm (gso j) > 0$

<proof>

lemma *Gramian-determinant*:

assumes $k: k \leq m$

shows *Gramian-determinant fs k* = $(\prod_{j < k} \text{sq-norm } (\text{gso } j))$
Gramian-determinant fs k > 0
 <proof>

lemma *Gramian-determinant-div:*

assumes $l < m$

shows *Gramian-determinant fs (Suc l)* / *Gramian-determinant fs l* = $\|\text{gso } l\|^2$
 <proof>

end

lemma (in *gram-schmidt-fs-Rn*) *Gramian-determinant-Ints:*

assumes $k \leq m \wedge i j. i < n \implies j < m \implies \text{fs } ! j \ \$ i \in \mathbf{Z}$

shows *Gramian-determinant fs k* $\in \mathbf{Z}$
 <proof>

locale *gram-schmidt-fs-int* = *gram-schmidt-fs-lin-indpt* +

assumes *fs-int*: $\wedge i j. i < n \implies j < m \implies \text{fs } ! j \ \$ i \in \mathbf{Z}$

begin

lemma *Gramian-determinant-ge1:*

assumes $k \leq m$

shows $1 \leq \text{Gramian-determinant fs } k$
 <proof>

lemma *mu-bound-Gramian-determinant:*

assumes $l < k \ k < m$

shows $(\mu \ k \ l)^2 \leq \text{Gramian-determinant fs } l * \|\text{fs } ! k\|^2$
 <proof>

end

context *gram-schmidt*

begin

lemma *gso-cong:*

fixes $f1 \ f2 :: 'a \ \text{vec list}$

assumes $\wedge i. i \leq x \implies f1 \ ! i = f2 \ ! i$

shows *gram-schmidt-fs.gso n f1 x* = *gram-schmidt-fs.gso n f2 x*
 <proof>

lemma *mu-cong:*

fixes $f1 \ f2 :: 'a \ \text{vec list}$

assumes $\wedge k. j < i \implies k \leq j \implies f1 \ ! k = f2 \ ! k$

and $j < i \implies f1 \ ! i = f2 \ ! i$

shows *gram-schmidt-fs.mu n f1 i j* = *gram-schmidt-fs.mu n f2 i j*
 <proof>

end

lemma *prod-list-le-mono*: **fixes** $us :: 'a :: \{\text{linordered-nonzero-semiring}, \text{ordered-ring}\}$
list
assumes $\text{length } us = \text{length } vs$
and $\bigwedge i. i < \text{length } vs \implies 0 \leq us ! i \wedge us ! i \leq vs ! i$
shows $0 \leq \text{prod-list } us \wedge \text{prod-list } us \leq \text{prod-list } vs$
 $\langle \text{proof} \rangle$

lemma *lattice-of-of-int*: **assumes** $G: \text{set } F \subseteq \text{carrier-vec } n$
and $f \in \text{vec-module.lattice-of } n F$
shows $\text{map-vec rat-of-int } f \in \text{vec-module.lattice-of } n (\text{map } (\text{map-vec of-int}) F)$
(is $?f \in \text{vec-module.lattice-of } - ?F)$
 $\langle \text{proof} \rangle$

lemma *Hadamard's-inequality*:
fixes $A::\text{real mat}$
assumes $A: A \in \text{carrier-mat } n n$
shows $\text{abs } (\text{det } A) \leq \text{sqrt } (\text{prod-list } (\text{map sq-norm } (\text{rows } A)))$
 $\langle \text{proof} \rangle$

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* :: $\text{nat} \Rightarrow \text{int vec list} \Rightarrow \text{rat list list} \times \text{rat vec list}$
where
 $\text{gram-schmidt-int } n \ us = \text{gram-schmidt-wit } n (\text{map } (\text{map-vec of-int}) \ us)$

lemma *snd-gram-schmidt-int* : $\text{snd } (\text{gram-schmidt-int } n \ us) = \text{gram-schmidt } n$
 $(\text{map } (\text{map-vec of-int}) \ us)$
 $\langle \text{proof} \rangle$

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

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

fun *gram-schmidt-sub-triv*
where $\text{gram-schmidt-sub-triv } n \ us \ [] = us$
 $| \text{gram-schmidt-sub-triv } n \ us \ (w \# ws) = (\text{let } u = \text{adjuster-triv } n \ w \ us + w \ \text{in}$
 $\text{gram-schmidt-sub-triv } n \ ((u, \text{sq-norm-vec-rat } u) \# us) \ ws)$

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

where *gram-schmidt-triv* n $ws = \text{rev } (\text{gram-schmidt-sub-triv } n \ [] \ ws)$

lemma *adjuster-triv*: *adjuster-triv* n w $(\text{map } (\lambda x. (x, \text{sq-norm } x)) \ us) = \text{adjuster } n \ w \ us$
 <proof>

lemma *gram-schmidt-sub-triv*: *gram-schmidt-sub-triv* n $((\text{map } (\lambda x. (x, \text{sq-norm } x)) \ us)) \ ws =$
 $\text{map } (\lambda x. (x, \text{sq-norm } x)) \ (\text{gram-schmidt-sub } n \ us \ ws)$
 <proof>

lemma *gram-schmidt-triv[simp]*: *gram-schmidt-triv* n $ws = \text{map } (\lambda x. (x, \text{sq-norm } x)) \ (\text{gram-schmidt } n \ ws)$
 <proof>

context *gram-schmidt*
begin

fun *mus-adjuster* :: $'a \ \text{vec} \Rightarrow ('a \ \text{vec} \times 'a) \ \text{list} \Rightarrow 'a \ \text{list} \Rightarrow 'a \ \text{vec} \Rightarrow 'a \ \text{list} \times 'a \ \text{vec}$

where

mus-adjuster $f \ [] \ \text{mus } g' = (\text{mus}, g') \ |$
mus-adjuster $f \ ((g, ng) \# n\text{-gs}) \ \text{mus } g' = (\text{let } a = (f \cdot g) / ng \ \text{in}$
 $\text{mus-adjuster } f \ n\text{-gs} \ (a \ \# \ \text{mus}) \ (-a \cdot_v \ g + g')$

fun *norms-mus'* **where**

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

lemma *adjuster-wit-carrier-vec*:

assumes $f \in \text{carrier-vec } n \ \text{set } gs \subseteq \text{carrier-vec } n$
shows $\text{snd } (\text{adjuster-wit } \text{mus } f \ gs) \in \text{carrier-vec } n$
 <proof>

lemma *adjuster-wit''*:

assumes *adjuster-wit* $\text{mus-acc } f \ gs = (\text{mus}, g') \ n\text{-gs} = \text{map } (\lambda x. (x, \text{sq-norm-vec } x)) \ gs$
 $f \in \text{carrier-vec } n \ \text{acc} \in \text{carrier-vec } n \ \text{set } gs \subseteq \text{carrier-vec } n$
shows *mus-adjuster* $f \ n\text{-gs} \ \text{mus-acc } \text{acc} = (\text{mus}, \text{acc} + g')$
 <proof>

lemma *adjuster-wit'*:

assumes $n\text{-gs} = \text{map } (\lambda x. (x, \text{sq-norm-vec } x)) \ gs \ f \in \text{carrier-vec } n \ \text{set } gs \subseteq \text{carrier-vec } n$
shows *mus-adjuster* $f \ n\text{-gs} \ \text{mus-acc} \ (0_v \ n) = \text{adjuster-wit } \text{mus-acc } f \ gs$
 <proof>

lemma *sub2-wit-norms-mus'*:

assumes $n\text{-}gs' = \text{map } (\lambda v. (v, \text{sq-norm-vec } v)) \text{ } gs'$
 $\text{sub2-wit } gs' \text{ } fs = (mus, gs) \text{ set } fs \subseteq \text{carrier-vec } n \text{ set } gs' \subseteq \text{carrier-vec } n$
shows $\text{norms-mus}' \text{ } fs \text{ } n\text{-}gs' \text{ } mus\text{-acc} = (\text{map } \text{sq-norm-vec } (\text{rev } gs \text{ } @ \text{ } gs'), \text{rev } mus$
 $@ \text{ } mus\text{-acc})$
 $\langle \text{proof} \rangle$

lemma *sub2-wit-gram-schmidt-sub-triv''*:

assumes $\text{sub2-wit } [] \text{ } fs = (mus, gs) \text{ set } fs \subseteq \text{carrier-vec } n$
shows $\text{norms-mus}' \text{ } fs \text{ } [] \text{ } [] = (\text{map } \text{sq-norm-vec } (\text{rev } gs), \text{rev } mus)$
 $\langle \text{proof} \rangle$

definition *norms-mus where*

$\text{norms-mus } fs = (\text{let } (n\text{-}gs, mus) = \text{norms-mus}' \text{ } fs \text{ } [] \text{ } [] \text{ in } (\text{rev } n\text{-}gs, \text{rev } mus))$

lemma *sub2-wit-gram-schmidt-norm-mus*:

assumes $\text{sub2-wit } [] \text{ } fs = (mus, gs) \text{ set } fs \subseteq \text{carrier-vec } n$
shows $\text{norms-mus } fs = (\text{map } \text{sq-norm-vec } gs, mus)$
 $\langle \text{proof} \rangle$

lemma (**in** *gram-schmidt-fs-Rn*) **norms-mus**: **assumes** $\text{set } fs \subseteq \text{carrier-vec } n \text{ length } fs \leq n$

shows $\text{norms-mus } fs = (\text{map } (\lambda j. \|\text{gs } j\|^2) [0..<\text{length } fs], \text{map } (\lambda i. \text{map } (\mu \text{ } i) [0..<i]) [0..<\text{length } fs])$
 $\langle \text{proof} \rangle$

end

fun *mus-adjuster-rat* :: $\text{rat } \text{vec} \Rightarrow (\text{rat } \text{vec} \times \text{rat}) \text{ list} \Rightarrow \text{rat } \text{list} \Rightarrow \text{rat } \text{vec} \Rightarrow \text{rat } \text{list} \times \text{rat } \text{vec}$

where

$\text{mus-adjuster-rat } f \text{ } [] \quad \text{mus } g' = (mus, g') \mid$
 $\text{mus-adjuster-rat } f \text{ } ((g, ng)\#n\text{-}gs) \text{ } \text{mus } g' = (\text{let } a = (f \cdot g) / ng \text{ in}$
 $\text{mus-adjuster-rat } f \text{ } n\text{-}gs \text{ } (a \# \text{mus}) \text{ } (-a \cdot_v g +$
 $g'))$

fun *norms-mus-rat'* **where**

$\text{norms-mus-rat}' \text{ } n \text{ } [] \quad n\text{-}gs \text{ } mus = (\text{map } \text{snd } n\text{-}gs, mus) \mid$
 $\text{norms-mus-rat}' \text{ } n \text{ } (f \# fs) \text{ } n\text{-}gs \text{ } mus =$
 $(\text{let } (\text{mus-row}, g') = \text{mus-adjuster-rat } f \text{ } n\text{-}gs \text{ } [] \text{ } (0_v \text{ } n);$
 $g = g' + f \text{ in}$
 $\text{norms-mus-rat}' \text{ } n \text{ } fs \text{ } ((g, \text{sq-norm-vec } g) \# n\text{-}gs) \text{ } (\text{mus-row}\#\text{mus}))$

definition *norms-mus-rat where*

$\text{norms-mus-rat } n \text{ } fs = (\text{let } (n\text{-}gs, mus) = \text{norms-mus-rat}' \text{ } n \text{ } fs \text{ } [] \text{ } [] \text{ in } (\text{rev } n\text{-}gs, \text{rev } mus))$

lemma *norms-mus-rat-norms-mus*:

norms-mus-rat n fs = *gram-schmidt.norms-mus* n fs
 ⟨*proof*⟩

lemma *of-int-dvd*:
b *dvd* *a* **if** *of-int* *a* / (*of-int* *b* :: '*a* :: field-char-0) ∈ \mathbb{Z} *b* ≠ 0
 ⟨*proof*⟩

lemma *denom-dvd-ints*:
fixes *i*::*int*
assumes *quotient-of* $r = (z, n)$ *of-int* $i * r \in \mathbb{Z}$
shows n *dvd* i
 ⟨*proof*⟩

lemma *quotient-of-bounds*:
assumes *quotient-of* $r = (n, d)$ *rat-of-int* $i * r \in \mathbb{Z}$ $0 < i$ $|r| \leq b$
shows *of-int* $|n| \leq$ *of-int* $i * b$ $d \leq i$
 ⟨*proof*⟩

context *gram-schmidt-fs-Rn*
begin

lemma *ex-κ*:
assumes $i < \text{length } fs$ $l \leq i$
shows $\exists \kappa. \text{sumlist } (\text{map } (\lambda j. - \mu \ i \ j \cdot_v \ gso \ j) [0..<l]) =$
 $\text{sumlist } (\text{map } (\lambda j. \kappa \ j \cdot_v \ fs \ ! \ j) [0..<l])$ (**is** $\exists \kappa. ?Prop \ l \ i \ \kappa$)
 ⟨*proof*⟩

definition *κ-SOME-def*:
 $\kappa = (SOME \ \kappa. \forall i \ l. \ i < \text{length } fs \longrightarrow l \leq i \longrightarrow$
 $\text{sumlist } (\text{map } (\lambda j. - \mu \ i \ j \cdot_v \ gso \ j) [0..<l]) =$
 $\text{sumlist } (\text{map } (\lambda j. \kappa \ i \ l \ j \cdot_v \ fs \ ! \ j) [0..<l]))$

lemma *κ-def*:
assumes $i < \text{length } fs$ $l \leq i$
shows $\text{sumlist } (\text{map } (\lambda j. - \mu \ i \ j \cdot_v \ gso \ j) [0..<l]) =$
 $\text{sumlist } (\text{map } (\lambda j. \kappa \ i \ l \ j \cdot_v \ fs \ ! \ j) [0..<l])$
 ⟨*proof*⟩

lemma (**in** *gram-schmidt-fs-lin-indpt*) *fs-i-sumlist-κ*:
assumes $i < m$ $l \leq i$ $j < l$
shows $(fs \ ! \ i + \text{sumlist } (\text{map } (\lambda j. \kappa \ i \ l \ j \cdot_v \ fs \ ! \ j) [0..<l])) \cdot fs \ ! \ j = 0$
 ⟨*proof*⟩

end

lemma *Ints-sum*:

assumes $\bigwedge a. a \in A \implies f a \in \mathbb{Z}$

shows $\text{sum } f A \in \mathbb{Z}$

<proof>

lemma *Ints-prod*:

assumes $\bigwedge a. a \in A \implies f a \in \mathbb{Z}$

shows $\text{prod } f A \in \mathbb{Z}$

<proof>

lemma *Ints-scalar-prod*:

$v \in \text{carrier-vec } n \implies w \in \text{carrier-vec } n$

$\implies (\bigwedge i. i < n \implies v \$ i \in \mathbb{Z}) \implies (\bigwedge i. i < n \implies w \$ i \in \mathbb{Z}) \implies v \cdot w \in \mathbb{Z}$

<proof>

lemma *Ints-det*: **assumes** $\bigwedge i j. i < \text{dim-row } A \implies j < \text{dim-col } A$

$\implies A \$\$ (i,j) \in \mathbb{Z}$

shows $\text{det } A \in \mathbb{Z}$

<proof>

lemma (in *gram-schmidt-fs-Rn*) *Gramian-matrix-alt-alt-def*:

assumes $k \leq m$

shows $\text{Gramian-matrix } fs \ k = \text{mat } k \ k (\lambda(i,j). fs ! i \cdot fs ! j)$

<proof>

lemma (in *gram-schmidt-fs-int*) *fs-scalar-Ints*:

assumes $i < m \ j < m$

shows $fs ! i \cdot fs ! j \in \mathbb{Z}$

<proof>

abbreviation (in *gram-schmidt-fs-lin-indpt*) *d* **where** $d \equiv \text{Gramian-determinant } fs$

lemma (in *gram-schmidt-fs-lin-indpt*) *fs-i-fs-j-sum- κ* :

assumes $i < m \ l \leq i \ j < l$

shows $-(fs ! i \cdot fs ! j) = (\sum t = 0..<l. fs ! t \cdot fs ! j * \kappa \ i \ l \ t)$

<proof>

lemma (in *gram-schmidt-fs-lin-indpt*) *Gramian-matrix-times- κ* :

assumes $i < m \ l \leq i$

shows $\text{Gramian-matrix } fs \ l *_v (\text{vec } l (\lambda t. \kappa \ i \ l \ t)) = (\text{vec } l (\lambda j. -(fs ! i \cdot fs ! j)))$

<proof>

lemma (in *gram-schmidt-fs-int*) *d- κ -Ints* :

assumes $i < m \ l \leq i \ t < l$

shows $d \ l * \kappa \ i \ l \ t \in \mathbb{Z}$

<proof>

lemma (in *gram-schmidt-fs-int*) *d-gso-Ints*:
assumes $i < n$ $k < m$
shows $(d\ k\ \cdot_v\ (gso\ k))\ \$\ i \in \mathbf{Z}$
 $\langle proof \rangle$

lemma (in *gram-schmidt-fs-int*) *d-mu-Ints*:
assumes $l \leq k$ $k < m$
shows $d\ (Suc\ l)\ * \ \mu\ k\ l \in \mathbf{Z}$
 $\langle proof \rangle$

lemma *max-list-Max*: $ls \neq [] \implies max-list\ ls = Max\ (set\ ls)$
 $\langle proof \rangle$

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

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

definition $N = Max\ (sq-norm\ \cdot\ set\ fs)$

lemma *N-ge-0*:
assumes $0 < m$
shows $0 \leq N$
 $\langle proof \rangle$

lemma *N-fs*:
assumes $i < m$
shows $\|fs\ !\ i\|^2 \leq N$
 $\langle proof \rangle$

lemma *N-gso*:
assumes $i < m$
shows $\|gso\ i\|^2 \leq N$
 $\langle proof \rangle$

lemma *N-d*:
assumes $i \leq m$
shows *Gramian-determinant fs i* $\leq N \wedge i$
 $\langle proof \rangle$

end

lemma *ex-MAXIMUM*: **assumes** *finite A* $A \neq \{\}$

shows $\exists a \in A. \text{Max } (f \text{ ' } A) = f a$
<proof>

context *gram-schmidt-fs-int*
begin

lemma *fs-int'*: $k < n \implies f \in \text{set } fs \implies f \$ k \in \mathbb{Z}$
<proof>

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>

lemma
assumes $\text{set } fs \neq \{\}$
shows *N-Ints*: $N \in \mathbb{Z}$ **and** *N-1*: $1 \leq N$
<proof>

lemma *N-mu*:
assumes $i < m \ j \leq i$
shows $(\mu \ i \ j)^2 \leq N \wedge (\text{Suc } j)$
<proof>

end

lemma *vec-hom-Ints*:
assumes $i < n \ xs \in \text{carrier-vec } n$
shows *of-int-hom.vec-hom xs* $\$ i \in \mathbb{Z}$
<proof>

lemma *division-to-div*: $(\text{of-int } x \ :: 'a \ :: \text{floor-ceiling}) = \text{of-int } y / \text{of-int } z \implies x = y \text{ div } z$
<proof>

lemma *exact-division*: **assumes** $\text{of-int } x / (\text{of-int } y \ :: 'a \ :: \text{floor-ceiling}) \in \mathbb{Z}$
shows $\text{of-int } (x \text{ div } y) = \text{of-int } x / (\text{of-int } y \ :: 'a)$
<proof>

lemma *int-via-rat-eqI*: $\text{rat-of-int } x = \text{rat-of-int } y \implies x = y$ *<proof>*

locale *fs-int* =
fixes
 $n \ :: \text{nat}$ **and**
 $fs\text{-init} \ :: \text{int vec list}$
begin

sublocale *vec-module TYPE(int) n* *<proof>*

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* \langle *proof* \rangle

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* \wedge *i* < *length F* \implies *dim-vec (F ! i)* = *n*

shows *gs.Gramian-determinant (map of-int-hom.vec-hom F) k* = *of-int (gs.Gramian-determinant F k)*

\langle *proof* \rangle

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*

\langle *proof* \rangle

sublocale *gs*: *gram-schmidt-fs-int n RAT fs*

\langle *proof* \rangle

lemma *f-carrier[dest]*: *i* < *m* \implies *fs ! i* \in *carrier-vec n*

and *fs-carrier [simp]*: *set fs* \subseteq *carrier-vec n*

\langle *proof* \rangle

lemma *Gramian-determinant*:

assumes *k*: *k* \leq *m*

shows *of-int (gs.Gramian-determinant fs k)* = (\prod *j* < *k*. *sq-norm (gs.gso j)*) (**is** *?g1*)

gs.Gramian-determinant fs k > 0 (**is** *?g2*)

\langle *proof* \rangle

lemma *fs-int-d-pos [intro]*:

assumes *k*: *k* \leq *m*

shows *d fs k* > 0

\langle *proof* \rangle

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)*

\langle *proof* \rangle

lemma *fs-int-D-pos*:
shows $D\ fs > 0$
 $\langle proof \rangle$

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

lemma *fs-int-mu-d-Z*:
assumes $j: j \leq ii$ **and** $ii: ii < m$
shows $of-int\ (d\ fs\ (Suc\ j)) * gs.\mu\ ii\ j \in \mathbb{Z}$
 $\langle proof \rangle$

lemma *fs-int-mu-d-Z-m-m*:
assumes $j: j < m$ **and** $ii: ii < m$
shows $of-int\ (d\ fs\ (Suc\ j)) * gs.\mu\ ii\ j \in \mathbb{Z}$
 $\langle proof \rangle$

lemma *sq-norm-fs-via-sum-mu-gso*: **assumes** $i: i < m$
shows $of-int\ \|fs\ !\ i\|^2 = (\sum j \leftarrow [0..<Suc\ i]. (gs.\mu\ i\ j)^2 * \|gs.gso\ j\|^2)$
 $\langle proof \rangle$

lemma *d μ* : **assumes** $j < m$ $ii < m$
shows $of-int\ (d\mu\ ii\ j) = of-int\ (d\ fs\ (Suc\ j)) * gs.\mu\ ii\ j$
 $\langle proof \rangle$

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* = $\|fs ! i\|^2 - (\sum j = 0..<i. (\mu i j)^2 * \beta fs j)$
<proof>

lemma *gso-norm-beta*:

assumes *j* < *m*

shows β *fs* *j* = *sq-norm* (*gso* *j*)

<proof>

lemma *μ-Gramian-beta-def*:

assumes *j* < *i* *i* < *m*

shows $\mu i j = (fs ! i \cdot fs ! j - (\sum k = 0..<j. \mu j k * \mu i k * \beta fs k)) / \beta fs j$
<proof>

end

lemma (in *gram-schmidt*) *Gramian-matrix-alt-alt-alt-def*:

assumes *k* ≤ *length* *fs* *set* *fs* ⊆ *carrier-vec* *n*

shows Gramian-matrix $fs\ k = mat\ k\ k\ (\lambda(i,j). fs\ !\ i \cdot fs\ !\ j)$
 <proof>

lemma (in *gram-schmidt-fs-Rn*) Gramian-determinant-1 [*simp*]:
assumes $0 < length\ fs$
shows Gramian-determinant $fs\ (Suc\ 0) = ||fs\ !\ 0||^2$
 <proof>

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*: $d\ (Suc\ i) = \mu'\ i\ i$ <proof>

lemma *d-0*: $d\ 0 = 1$ <proof>

lemma σ : **assumes** $lj: l \leq m$
shows $\sigma\ l\ i\ j = d\ l * (\sum k < l. \mu\ i\ k * \mu\ j\ k * \beta\ fs\ k)$
 <proof>

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>

lemma σ -via- μ' : $\sigma\ (Suc\ l)\ i\ j =$
 (if $l = 0$ then $\mu'\ i\ 0 * \mu'\ j\ 0$ else $(\mu'\ l\ l * \sigma\ l\ i\ j + \mu'\ i\ l * \mu'\ j\ l) / \mu'\ (l - 1) (l - 1)$)
 <proof>

lemma μ' -via- σ : **assumes** $j: j \leq i$ **and** $i: i < m$
shows $\mu'\ i\ j =$
 (if $j = 0$ then $fs\ !\ i \cdot fs\ !\ j$ else $\mu'\ (j - 1) (j - 1) * (fs\ !\ i \cdot fs\ !\ j) - \sigma\ j\ i\ j$)
 <proof>

lemma *fs-i-sumlist- κ* :
assumes $i < m\ l \leq i\ j < l$
shows $(fs\ !\ i + sumlist\ (map\ (\lambda j. \kappa\ i\ l\ j \cdot_v\ fs\ !\ j)\ [0..<l])) \cdot fs\ !\ j = 0$
 <proof>

end

context *gram-schmidt-fs-int*
begin

lemma $\beta\text{-pos} : i < m \implies \beta \text{ fs } i > 0$
 ⟨*proof*⟩

lemma $\beta\text{-zero} : i < m \implies \beta \text{ fs } i \neq 0$
 ⟨*proof*⟩

lemma $\sigma\text{-integer}$:
assumes $l : l \leq j$ **and** $j : j \leq i$ **and** $i : i < m$
shows $\sigma \text{ l } i \text{ j} \in \mathbf{Z}$
 ⟨*proof*⟩

end

context *fs-int-indpt*
begin

fun σs **and** μ' **where**

$\sigma s \ 0 \ i \ j = \mu' \ i \ 0 * \mu' \ j \ 0$
 $|\ \sigma s \ (Suc \ l) \ i \ j = (\mu' \ (Suc \ l) \ (Suc \ l) * \sigma s \ l \ i \ j + \mu' \ i \ (Suc \ l) * \mu' \ j \ (Suc \ l)) \ \text{div} \ \mu' \ l \ l$
 $|\ \mu' \ i \ j = (\text{if } j = 0 \text{ then } fs \ ! \ i \cdot fs \ ! \ j \ \text{else } \mu' \ (j - 1) \ (j - 1) * (fs \ ! \ i \cdot fs \ ! \ j) - \sigma s \ (j - 1) \ i \ j)$

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

lemma $\sigma s\text{-}\mu'$: $l < j \implies j \leq i \implies i < m \implies \text{of-int } (\sigma s \ l \ i \ j) = \text{gs.}\sigma \ (Suc \ l) \ i \ j$
 $i < m \implies j \leq i \implies \text{of-int } (\mu' \ i \ j) = \text{gs.}\mu' \ i \ j$
 ⟨*proof*⟩

lemma μ' : **assumes** $i < m \ j \leq i$
shows $\mu' \ i \ j = d\mu \ i \ j$
 $j = i \implies \mu' \ i \ j = d \text{ fs } (Suc \ i)$
 ⟨*proof*⟩

lemma $\sigma\text{-array}$: **assumes** $mm : mm \leq m$ **and** $j : j < mm$
shows $l \leq j \implies \sigma\text{-array} \ (IArray.\text{of-fun } (\lambda i. \ IArray.\text{of-fun } (\mu' \ i) \ (\text{if } i = mm \ \text{then } Suc \ j \ \text{else } Suc \ i)) \ (Suc \ mm))$
 $(IArray.\text{of-fun } (\mu' \ mm) \ (Suc \ j)) \ (IArray.\text{of-fun } (\mu' \ (Suc \ j)) \ (\text{if } Suc \ j = mm \ \text{then } Suc \ j \ \text{else } Suc \ (Suc \ j))) \ (\mu' \ l \ l) \ l =$
 $\sigma s \ l \ mm \ (Suc \ j)$
 ⟨*proof*⟩

lemma *dmu-array-row-main*: **assumes** *mm*: $mm \leq m$ **shows**

$j \leq mm \implies \text{dmu-array-row-main } (IArray\ fs) (IArray\ fs\ !!\ mm)\ mm$
 $(IArray.of\text{-fun } (\lambda i. IArray.of\text{-fun } (\mu' i) (if\ i = mm\ then\ Suc\ j\ else\ Suc\ i)) (Suc\ mm))$

$j = IArray.of\text{-fun } (\lambda i. IArray.of\text{-fun } (\mu' i) (Suc\ i)) (Suc\ mm)$
<proof>

lemma *dmu-array-row*: **assumes** *mm*: $mm \leq m$ **shows**

$\text{dmu-array-row } (IArray\ fs) (IArray.of\text{-fun } (\lambda i. IArray.of\text{-fun } (\mu' i) (Suc\ i))\ mm)$
 $mm =$

$IArray.of\text{-fun } (\lambda i. IArray.of\text{-fun } (\mu' i) (Suc\ i)) (Suc\ mm)$
<proof>

lemma *dmu-array*: **assumes** $mm \leq m$

shows $\text{dmu-array } (IArray\ fs)\ m (IArray.of\text{-fun } (\lambda i. IArray.of\text{-fun } (\lambda j. \mu' i\ j) (Suc\ i))\ mm)\ mm$

$= IArray.of\text{-fun } (\lambda i. IArray.of\text{-fun } (\lambda j. \mu' i\ j) (Suc\ i))\ m$
<proof>

lemma *d μ -impl*: $d\mu\text{-impl } fs = IArray.of\text{-fun } (\lambda i. IArray.of\text{-fun } (\lambda j. d\mu\ i\ j) (Suc\ i))\ m$

<proof>

end

context *gram-schmidt-fs-int*

begin

lemma *N- μ'* :

assumes $i < m\ j \leq i$

shows $(\mu' i\ j)^2 \leq N \wedge (3 * Suc\ j)$

<proof>

lemma *N- σ* :

assumes $i < m\ j \leq i\ l \leq j$

shows $|\sigma\ l\ i\ j| \leq of\text{-nat } l * N \wedge (2 * l + 2)$

<proof>

lemma *leq-squared*: $(z::int) \leq z^2$

<proof>

lemma *abs-leq-squared*: $|z::int| \leq z^2$

<proof>

end

context *gram-schmidt-fs-int*

begin

definition *gso'* where $gso' i = d i \cdot_v (gso i)$

fun *a* where

$a i 0 = fs ! i |$

$a i (Suc l) = (1 / d l) \cdot_v ((d (Suc l) \cdot_v (a i l)) - (\mu' i l) \cdot_v gso' l)$

lemma *gso'-carrier-vec*:

assumes $i < m$

shows $gso' i \in carrier-vec n$

$\langle proof \rangle$

lemma *a-carrier-vec*:

assumes $l \leq i i < m$

shows $a i l \in carrier-vec n$

$\langle proof \rangle$

lemma *a-l*:

assumes $l \leq i i < m$

shows $a i l = d l \cdot_v (fs ! i + M.sumlist (map (\lambda j. - \mu i j \cdot_v gso j) [0..<l]))$

$\langle proof \rangle$

lemma *a-l'*:

assumes $i < m$

shows $a i i = gso' i$

$\langle proof \rangle$

lemma

assumes $i < m l' \leq i$

shows $a i l' = (case l' of$

$0 \Rightarrow fs ! i |$

$Suc l \Rightarrow (1 / d l) \cdot_v (d (Suc l) \cdot_v (a i l) - (\mu' i l) \cdot_v a l l))$

$\langle proof \rangle$

lemma *a-Ints*:

assumes $i < m l \leq i k < n$

shows $a i l \$ k \in \mathbf{Z}$

$\langle proof \rangle$

lemma *a-alt-def*:

assumes $l < length fs$

shows $a i (Suc l) = (let v = \mu' l l \cdot_v (a i l) - (\mu' i l) \cdot_v a l l in$

$(if l = 0 then v else (1 / \mu' (l - 1) (l - 1)) \cdot_v v))$

$\langle proof \rangle$

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* ($\lambda k. k \text{ div } \mu' (l - 1) (l - 1)$) *v*))

lemma *gso-int-carrier-vec*:
assumes *i* < *length fs* *l* ≤ *i*
shows *gso-int* *i* *l* ∈ *carrier-vec* *n*
⟨*proof*⟩

lemma *gso-int*:
assumes *i* < *length fs* *l* ≤ *i*
shows *of-int-hom.vec-hom* (*gso-int* *i* *l*) = *gs.a* *i* *l*
⟨*proof*⟩

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* ($\lambda k. k \text{ div } \mu' (l - 1) (l - 1)$) *v*)
in *gso-int-tail'* *i* (*l* + 1) *acc'*)
⟨*proof*⟩

termination
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *gso-int-tail*: *gso-int-tail* *i* = *gso-int* *i* *i*
⟨*proof*⟩

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* ($\lambda k. k \text{ div } dmusa !! (l - 1) !! (l - 1)$) *v*))

```

      in while i (l + 1) gsa dmusa acc')
    ⟨proof⟩
termination
    ⟨proof⟩

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)))
  ⟨proof⟩
termination
  ⟨proof⟩

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 (λi. (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
  ⟨proof⟩

context fs-int-indpt
begin

lemma gso-array.gso'-array (dμ-impl fs) fs = IArray (map (λk. gso-int k k) [0..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 $|x| \leq of\text{-}nat\ m * N \wedge (3 * Suc\ m)$
 $\langle proof \rangle$

end

context *fs-int-indpt*
begin

lemma *combined-size-bound-rat-log*:
assumes $x: x \in \{gs.\mu'\ i\ j \mid i\ j. j \leq i \wedge i < m\}$
 $\cup \{gs.\sigma\ l\ i\ j \mid i\ j\ l. i < m \wedge j \leq i \wedge l \leq j\}$
(is $x \in ?\mu' \cup ?\sigma$)
and $m: m \neq 0\ x \neq 0$
shows $\log\ 2\ |real\text{-}of\text{-}rat\ x| \leq \log\ 2\ m + (3 + 3 * m) * \log\ 2\ (real\text{-}of\text{-}rat\ gs.N)$
 $\langle proof \rangle$

lemma *combined-size-bound-integer-log*:
assumes $x: x \in \{\mu'\ i\ j \mid i\ j. j \leq i \wedge i < m\}$
 $\cup \{\sigma\ l\ i\ j \mid i\ j\ l. i < m \wedge j \leq i \wedge l < j\}$
(is $x \in ?\mu' \cup ?\sigma$)
and $m: m \neq 0\ x \neq 0$
shows $\log\ 2\ |real\text{-}of\text{-}int\ x| \leq \log\ 2\ m + (3 + 3 * m) * \log\ 2\ (real\text{-}of\text{-}rat\ gs.N)$
 $\langle proof \rangle$

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 orthogonal (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 a 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  $\alpha :: rat$ 

```

```

begin

```

```

sublocale  $vec\text{-}module\ TYPE(int)\ n\langle proof \rangle$ 

```

```

abbreviation  $RAT$  where  $RAT \equiv map\ (map\text{-}vec\ rat\text{-}of\text{-}int)$ 

```

```

abbreviation  $SRAT$  where  $SRAT\ xs \equiv set\ (RAT\ xs)$ 

```

```

abbreviation  $Rn$  where  $Rn \equiv carrier\text{-}vec\ n :: rat\ vec\ set$ 

```

```

sublocale  $gs: gram\text{-}schmidt\text{-}fs\ n\ RAT\ fs\text{-}init\ \langle proof \rangle$ 

```

```

abbreviation  $lin\text{-}indep$  where  $lin\text{-}indep\ fs \equiv gs.lin\text{-}indpt\text{-}list\ (RAT\ fs)$ 

```

```

abbreviation  $gso$  where  $gso\ fs \equiv gram\text{-}schmidt\text{-}fs.gso\ n\ (RAT\ fs)$ 

```

```

abbreviation  $\mu$  where  $\mu\ fs \equiv gram\text{-}schmidt\text{-}fs.\mu\ n\ (RAT\ fs)$ 

```

```

abbreviation  $reduced$  where  $reduced\ fs \equiv gram\text{-}schmidt\text{-}fs.reduced\ n\ (RAT\ fs)\ \alpha$ 

```

```

abbreviation  $weakly\text{-}reduced$  where  $weakly\text{-}reduced\ fs \equiv gram\text{-}schmidt\text{-}fs.weakly\text{-}reduced\ n\ (RAT\ fs)\ \alpha$ 

```

```

lattice of initial basis

```


definition $L = \text{lattice-of } fs\text{-init}$

maximum squared norm of initial basis

definition $N = \text{max-list } (\text{map } (\text{nat } \circ \text{sq-norm}) \text{ } fs\text{-init})$

maximum absolute value in initial basis

definition $M = \text{Max } (\{\text{abs } (fs\text{-init } ! \ i \ \$ \ j) \mid i \ j. \ i < m \wedge j < n\} \cup \{0\})$

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

definition $\mu\text{-small } fs \ i = (\forall \ j < i. \ \text{abs } (\mu \ fs \ i \ j) \leq 1/2)$

definition $LLL\text{-invariant-weak} :: \text{int } \text{vec } \text{list} \Rightarrow \text{bool}$ **where**

$LLL\text{-invariant-weak } fs = ($
 $gs.\text{lin-indpt-list } (RAT \ fs) \wedge$
 $\text{lattice-of } fs = L \wedge$
 $\text{length } fs = m)$

lemma $LLL\text{-inv-wD}$: **assumes** $LLL\text{-invariant-weak } fs$
shows

$\text{lin-indep } fs$
 $\text{length } (RAT \ fs) = m$
 $\text{set } fs \subseteq \text{carrier-vec } n$
 $\bigwedge \ i. \ i < m \implies fs \ ! \ i \in \text{carrier-vec } n$
 $\bigwedge \ i. \ i < m \implies \text{gso } fs \ i \in \text{carrier-vec } n$
 $\text{length } fs = m$
 $\text{lattice-of } fs = L$

$\langle \text{proof} \rangle$

lemma $LLL\text{-inv-wI}$: **assumes**

$\text{set } fs \subseteq \text{carrier-vec } n$
 $\text{length } fs = m$
 $\text{lattice-of } fs = L$
 $\text{lin-indep } fs$

shows $LLL\text{-invariant-weak } fs$

$\langle \text{proof} \rangle$

definition $LLL\text{-invariant} :: \text{bool} \Rightarrow \text{nat} \Rightarrow \text{int } \text{vec } \text{list} \Rightarrow \text{bool}$ **where**

$LLL\text{-invariant } \text{upw } i \ fs = ($
 $gs.\text{lin-indpt-list } (RAT \ fs) \wedge$
 $\text{lattice-of } fs = L \wedge$
 $\text{reduced } fs \ i \wedge$
 $i \leq m \wedge$
 $\text{length } fs = m \wedge$
 $(\text{upw} \vee \mu\text{-small } fs \ i)$
 $)$

lemma $LLL\text{-inv-imp-w}$: $LLL\text{-invariant } \text{upw } i \ fs \implies LLL\text{-invariant-weak } fs$

$\langle \text{proof} \rangle$

lemma *LLL-invD*: **assumes** *LLL-invariant upw i fs*

shows

lin-indep fs

length (RAT fs) = m

set fs \subseteq carrier-vec n

$\bigwedge i. i < m \implies fs ! i \in \text{carrier-vec } n$

$\bigwedge i. i < m \implies \text{gso } fs \ i \in \text{carrier-vec } n$

length fs = m

lattice-of fs = L

weakly-reduced fs i

i \leq m

reduced fs i

upw \vee μ -small fs i

<proof>

lemma *LLL-invI*: **assumes**

set fs \subseteq carrier-vec n

length fs = m

lattice-of fs = L

i \leq m

lin-indep fs

reduced fs i

upw \vee μ -small fs i

shows *LLL-invariant upw i fs*

<proof>

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' \subseteq fs-int-indpt*

<proof>

context *LLL*

begin

lemma *gso-cong*: **assumes** $\bigwedge i. i \leq x \implies f1 ! i = f2 ! i$

x < length f1 x < length f2

shows *gso f1 x = gso f2 x*

<proof>

lemma *μ -cong*: **assumes** $\bigwedge k. j < i \implies k \leq j \implies f1 ! k = f2 ! k$

and *i: i < length f1 i < length f2*

and *j < i \implies f1 ! i = f2 ! i*

shows *μ f1 i j = μ f2 i j*

<proof>

definition *reduction* **where** $reduction = (4 + \alpha) / (4 * \alpha)$

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** $D\ fs = nat\ (\prod\ i < m. d\ fs\ i)$

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

definition $logD :: int\ vec\ list \Rightarrow nat$

where $logD\ fs = (if\ \alpha = 4/3\ then\ (D\ fs)\ else\ nat\ (floor\ (log\ (1 / of-rat\ reduction)\ (D\ fs))))$

definition *LLL-measure* $:: nat \Rightarrow int\ vec\ list \Rightarrow nat$ **where**

$LLL-measure\ i\ fs = (2 * logD\ fs + m - i)$

context

fixes fs

assumes $Liniv: LLL-invariant-weak\ fs$

begin

interpretation $fs: fs-int'\ n\ m\ fs-init\ fs$

<proof>

lemma *Gramian-determinant:*

assumes $k: k \leq m$

shows $of-int\ (gs.Gramian-determinant\ fs\ k) = (\prod\ j < k. sq-norm\ (gso\ fs\ j))$ **(is ?g1)**

$gs.Gramian-determinant\ fs\ k > 0$ **(is ?g2)**

<proof>

lemma *LLL-d-pos [intro]:* **assumes** $k: k \leq m$

shows $d\ fs\ k > 0$

<proof>

lemma *LLL-d-Suc:* **assumes** $k: k < m$

shows $of-int\ (d\ fs\ (Suc\ k)) = sq-norm\ (gso\ fs\ k) * of-int\ (d\ fs\ k)$

<proof>

lemma *LLL-D-pos:*

shows $D\ fs > 0$

<proof>

end

Condition when we can increase the value of i

lemma *increase-i:*

assumes $Liniv: LLL-invariant\ upw\ i\ fs$

assumes $i: i < m$
and $upw: upw \implies i = 0$
and $red-i: i \neq 0 \implies sq\text{-norm } (gso\ fs\ (i - 1)) \leq \alpha * sq\text{-norm } (gso\ fs\ i)$
shows $LLL\text{-invariant True } (Suc\ i)\ fs\ LLL\text{-measure } i\ fs > LLL\text{-measure } (Suc\ i)\ fs$
 $\langle proof \rangle$

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

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

lemma *basis-reduction-add-row-main*: **assumes** $Liniv: LLL\text{-invariant-weak } fs$

and $i: i < m$ **and** $j: j < i$

and $fs': fs' = fs[i := fs ! i - c \cdot_v fs ! j]$

shows $LLL\text{-invariant-weak } fs'$

$LLL\text{-invariant True } i\ fs \implies LLL\text{-invariant True } i\ fs'$

$c = round\ (\mu\ fs\ i\ j) \implies \mu\text{-small-row } i\ fs\ (Suc\ j) \implies \mu\text{-small-row } i\ fs'\ j$

$c = round\ (\mu\ fs\ i\ j) \implies abs\ (\mu\ fs'\ i\ j) \leq 1/2$

$LLL\text{-measure } i\ fs' = LLL\text{-measure } i\ fs$

$\wedge i. i < m \implies gso\ fs'\ i = gso\ fs\ i$

$\wedge i' j'. i' < m \implies j' < m \implies$

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

$\wedge ii. ii \leq m \implies d\ fs'\ ii = d\ fs\ ii$

$\langle proof \rangle$

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

lemma *basis-reduction-add-row-main-0*: **assumes** $Liniv: LLL\text{-invariant-weak } fs$

and $i: i < m$ **and** $j: j < i$

and $0: round\ (\mu\ fs\ i\ j) = 0$

and $mu\text{-small}: \mu\text{-small-row } i\ fs\ (Suc\ j)$

shows $\mu\text{-small-row } i\ fs\ j$ (**is** ?g1)

$\langle proof \rangle$

lemma $\mu\text{-small-row-reft}: \mu\text{-small-row } i\ fs\ i$

$\langle proof \rangle$

lemma *basis-reduction-add-row-done*: **assumes** $Liniv: LLL\text{-invariant True } i\ fs$

and $i: i < m$

and $mu\text{-small}: \mu\text{-small-row } i\ fs\ 0$

shows $LLL\text{-invariant False } i\ fs$

$\langle proof \rangle$

lemma *d-swap-unchanged*: **assumes** $len: length\ F1 = m$

and $i0: i \neq 0$ **and** $i: i < m$ **and** $ki: k \neq i$ **and** $km: k \leq m$

and $swap: F2 = F1[i := F1 ! (i - 1), i - 1 := F1 ! i]$

shows $d\ F1\ k = d\ F2\ k$

<proof>

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 $\alpha 0$: $\alpha > 0$ $\alpha \neq 0$
<proof>

lemma *fs-init*: *set fs-init* \subseteq *carrier-vec n*
<proof>

lemma *reduction*: $0 < \text{reduction}$ $\text{reduction} \leq 1$
 $\alpha > 4/3 \implies \text{reduction} < 1$
 $\alpha = 4/3 \implies \text{reduction} = 1$
<proof>

lemma *base*: $\alpha > 4/3 \implies \text{base} > 1$ *<proof>*

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)) $>$ $\alpha * \text{sq-norm}$ (*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* \implies *LLL-invariant False* (*i* - 1) *fs'*
and *LLL-measure i fs* $>$ *LLL-measure* (*i* - 1) *fs'*

and $\bigwedge k. k < m \implies \text{gso } fs' k = (\text{if } k = i - 1 \text{ then}$
 $\text{gso } fs i + \mu fs i (i - 1) \cdot_v \text{gso } fs (i - 1)$
 $\text{else if } k = i \text{ then}$
 $\text{gso } fs (i - 1) - (\text{RAT } fs ! (i - 1) \cdot \text{gso } fs' (i - 1) / \text{sq-norm } (\text{gso } fs' (i$
 $- 1))) \cdot_v \text{gso } fs' (i - 1)$
 $\text{else } \text{gso } fs k)$ (**is** $\bigwedge k. - \implies - = ?\text{newg } k$)

and $\bigwedge k. k < m \implies \text{sq-norm } (\text{gso } fs' k) = (\text{if } k = i - 1 \text{ then}$
 $\text{sq-norm } (\text{gso } fs i) + (\mu fs i (i - 1) * \mu fs i (i - 1)) * \text{sq-norm } (\text{gso } fs (i$
 $- 1))$
 $\text{else if } k = i \text{ then}$

$sq\text{-norm } (gso\ fs\ i) * sq\text{-norm } (gso\ fs\ (i - 1)) / sq\text{-norm } (gso\ fs'\ (i - 1))$
 $else\ sq\text{-norm } (gso\ fs\ k) \text{ (is } \wedge k. - \implies - = ?new\text{-norm } k)$

and $\wedge ii\ j. ii < m \implies j < ii \implies \mu\ fs'\ ii\ j =$
 $if\ ii = i - 1\ then$
 $\mu\ fs\ i\ j$
 $else\ if\ ii = i\ then$
 $if\ j = i - 1\ then$
 $\mu\ fs\ i\ (i - 1) * sq\text{-norm } (gso\ fs\ (i - 1)) / sq\text{-norm } (gso\ fs'\ (i - 1))$
 $else$
 $\mu\ fs\ (i - 1)\ j$
 $else\ if\ ii > i \wedge j = i\ then$
 $\mu\ fs\ ii\ (i - 1) - \mu\ fs\ i\ (i - 1) * \mu\ fs\ ii\ i$
 $else\ if\ ii > i \wedge j = i - 1\ then$
 $\mu\ fs\ ii\ (i - 1) * \mu\ fs'\ i\ (i - 1) + \mu\ fs\ ii\ i * sq\text{-norm } (gso\ fs\ i) / sq\text{-norm}$
 $(gso\ fs'\ (i - 1))$
 $else\ \mu\ fs\ ii\ j \text{ (is } \wedge ii\ j. - \implies - \implies - = ?new\text{-mu } ii\ j)$

and $\wedge ii. ii \leq m \implies of\text{-int } (d\ fs'\ ii) = (if\ ii = i\ then$
 $sq\text{-norm } (gso\ fs'\ (i - 1)) / sq\text{-norm } (gso\ fs\ (i - 1)) * of\text{-int } (d\ fs\ i)$
 $else\ of\text{-int } (d\ fs\ ii))$

$\langle proof \rangle$

lemma *LLL-inv-initial-state: LLL-invariant True 0 fs-init*

$\langle proof \rangle$

lemma *LLL-inv-m-imp-reduced: assumes LLL-invariant True m fs*

shows *reduced fs m*

$\langle proof \rangle$

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*

$v \in L - \{0_v\ n\}$

$h \in L - \{0_v\ n\} \implies rat\text{-of-int } (sq\text{-norm } v) \leq \alpha \wedge (m - 1) * rat\text{-of-int } (sq\text{-norm } h)$

$v \neq 0_v\ j$

$\langle proof \rangle$

lemma *LLL-mu-d-Z: assumes inv: LLL-invariant-weak fs*

and *j: j ≤ ii and ii: ii < m*

shows *of-int (d fs (Suc j)) * μ fs ii j ∈ ℤ*

$\langle proof \rangle$

context *fixes fs*

assumes *Liniv: LLL-invariant-weak fs and gbnd: g-bound fs*

begin

interpretation *gs1*: *gram-schmidt-fs-lin-indpt n RAT fs*
<proof>

lemma *LLL-inv-N-pos*: **assumes** *m*: $m \neq 0$
shows $N > 0$
<proof>

lemma *d-approx-main*: **assumes** *i*: $ii \leq m$ $m \neq 0$
shows $\text{rat-of-int } (d \text{ fs } ii) \leq \text{rat-of-nat } (N^{\wedge}ii)$
<proof>

lemma *d-approx*: **assumes** *i*: $ii < m$
shows $\text{rat-of-int } (d \text{ fs } ii) \leq \text{rat-of-nat } (N^{\wedge}ii)$
<proof>

lemma *d-bound*: **assumes** *i*: $ii < m$
shows $d \text{ fs } ii \leq N^{\wedge}ii$
<proof>

lemma *D-approx*: $D \text{ fs} \leq N^{\wedge}(m * m)$
<proof>

lemma *LLL-measure-approx*: **assumes** $\alpha > 4/3$ $m \neq 0$
shows $\text{LLL-measure } i \text{ fs} \leq m + 2 * m * m * \log \text{ base } N$
<proof>
end

lemma *g-bound-fs-init*: *g-bound fs-init*
<proof>

lemma *LLL-measure-approx-fs-init*:
LLL-invariant upw i fs-init $\implies 4/3 < \alpha \implies m \neq 0 \implies$
 $\text{real } (\text{LLL-measure } i \text{ fs-init}) \leq \text{real } m + \text{real } (2 * m * m) * \log \text{ base } (\text{real } N)$
<proof>

lemma *N-le-MMn*: **assumes** *m0*: $m \neq 0$
shows $N \leq \text{nat } M * \text{nat } M * n$
<proof>

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)
⟨*proof*⟩

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*⟩

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>

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*)) > $\alpha * \text{sq-norm}$ (*gso fs* *i*)

and *i*: *i* < *m* *i* $\neq 0$

shows *LLL-invariant* *upw'* *i'* *fs'* (**is** ?*g1*)

LLL-measure *i'* *fs'* < *LLL-measure* *i fs* (**is** ?*g2*)

<proof>

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>

termination *<proof>*

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'*

<proof>

lemma *reduce-basis-inv*: **assumes** *res*: *reduce-basis* = *fs*

shows *LLL-invariant* *True* *m fs*

<proof>

lemma *reduce-basis*: **assumes** *res*: *reduce-basis* = *fs*

shows *lattice-of* *fs* = *L*

reduced *fs* *m*

lin-indep *fs*

length *fs* = *m*

<proof>

lemma *short-vector*: **assumes** *res*: *short-vector* = *v*

and *m0*: *m* $\neq 0$

shows *v* \in *carrier-vec* *n*

v \in *L* - {*0_v* *n*}

h \in *L* - {*0_v* *n*} \implies *rat-of-int* (*sq-norm* *v*) $\leq \alpha \wedge (m - 1) * \text{rat-of-int}$ (*sq-norm* *h*)

v $\neq 0_v$ *j*

<proof>

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* \Rightarrow *int* \Rightarrow *int* **where**
round-num-denom $n\ d = ((2 * n + d) \text{ div } (2 * d))$

lemma *round-num-denom*: *round-num-denom* $num\ denom =$
round (*of-int* $num / \text{rat-of-int } denom$)
 <proof>

```
context fs-int-indpt
begin
```

lemma *round-num-denom-d μ -d*:
assumes $j: j \leq i$ **and** $i: i < m$
shows *round-num-denom* ($d\mu\ i\ j$) ($d\ fs\ (Suc\ j)$) = *round* ($gs.\mu\ i\ j$)
 <proof>

lemma *d-sq-norm-comparison*:
assumes *quot*: *quotient-of* $\alpha = (num, denom)$
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* ($gs.gso\ (i - 1)$)) $\leq \alpha * \text{sq-norm } (gs.gso\ i)$
 <proof>

```
end
```

```
context LLL
begin
```

lemma *d-d μ -add-row*: **assumes** *Lin**v*: *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

$\bigwedge ii. ii \leq m \implies d fs' ii = d fs ii$

$\bigwedge i' j'. i' < m \implies j' < i' \implies$

$d\mu fs' i' j' = ($
 if $i' = i \wedge j' < j$
 then $d\mu fs i' j' - c * d\mu fs j j'$
 else if $i' = i \wedge j' = j$
 then $d\mu fs i' j' - c * d fs (Suc j)$
 else $d\mu fs i' j'$
 (**is** $\bigwedge i' j'. - \implies - \implies - = ?new-mu i' j'$)

$\langle proof \rangle$

end

context *LLL-with-assms*

begin

lemma *d-dμ-swap*: **assumes** *invw*: *LLL-invariant-weak fs*

and *small*: *LLL-invariant False k fs \vee abs ($\mu fs k (k - 1)) \leq 1/2$*

and *k*: $k < m$

and *k0*: $k \neq 0$

and *norm-ineq*: $sq-norm (gso fs (k - 1)) > \alpha * sq-norm (gso fs k)$

and *fs'-def*: $fs' = fs[k := fs ! (k - 1), k - 1 := fs ! k]$

shows

$\bigwedge i. i \leq m \implies$

$d fs' i = ($

if $i = k$ then

$(d fs (Suc k) * d fs (k - 1) + d\mu fs k (k - 1) * d\mu fs k (k - 1)) \text{ div } d fs$

k

else $d fs i$)

and

$\bigwedge i j. i < m \implies j < i \implies$

$d\mu fs' i j = ($

if $i = k - 1$ then

$d\mu fs k j$

else if $i = k \wedge j \neq k - 1$ then

$d\mu fs (k - 1) j$

else if $i > k \wedge j = k$ then

$(d fs (Suc k) * d\mu fs i (k - 1) - d\mu fs k (k - 1) * d\mu fs i j) \text{ div } d fs k$

else if $i > k \wedge j = k - 1$ then

$(d\mu fs k (k - 1) * d\mu fs i j + d\mu fs i k * d fs (k - 1)) \text{ div } d fs k$

else $d\mu fs i j$)

(**is** $\bigwedge i j. - \implies - \implies - = ?new-mu i j$)

$\langle proof \rangle$

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 *fm1-state* :: *LLL-dmu-d-state* ⇒ *int vec* **where**
fm1-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>

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 $\alpha :: \text{rat}$ **and** $n\ m :: \text{nat}$ **and** *fs-init* :: *int vec list*
begin

definition *swap-mu* :: *int iarray iarray* \Rightarrow *nat* \Rightarrow *int* \Rightarrow *int* \Rightarrow *int* \Rightarrow *int* \Rightarrow *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 (dsj * dm-ii !! im1 - dm-ii-j) div di
        else if j = im1 then (dmu-i-im1 * dm-ii-j + dm-ii !! i * dim1) div di
        else dm-ii-j) ii else
    if ii = i then let mu-im1 = dm-ii !! im1 in
      IArray.of-fun (λ j. if j = im1 then dm-ii-j else mu-im1 !! j) ii
    else IArray.of-fun (λ j. dm-ii !! 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>

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  $\alpha$ 
  in if di * di * denom  $\leq$  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*]) *<proof>*

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*])
<proof>

lemma *to-mu-repr*: *mu-repr mu fs* *<proof>*

lemma *to-d-repr*: *d-repr ds fs* *<proof>*

lemma *dmu-ij-state*: **assumes** *j*: *j* < *ii*
and *ii*: *ii* < *m*
shows *dmu-ij-state state ii j* = *dμ fs ii j*

<proof>

lemma *fi-state*: $i < m \implies \text{fi-state state} = \text{fs } ! \ i$
<proof>

lemma *fim1-state*: $i < m \implies i \neq 0 \implies \text{fim1-state state} = \text{fs } ! \ (i - 1)$
<proof>

lemma *d-state*: $ii \leq m \implies \text{d-state state } ii = \text{d fs } ii$
<proof>

lemma *fs-state*: $\text{length fs} = m \implies \text{fs-state state} = \text{fs}$
<proof>

lemma *LLL-state-inc-state*: **assumes** $i: i < m$
shows *LLL-impl-inv* (*inc-state state*) (*Suc i*) *fs*
 $\text{fs-state } (\text{inc-state state}) = \text{fs-state state}$
<proof>
end
end

context *LLL-with-assms*
begin

lemma *basis-reduction-add-rows-loop-impl*: **assumes**
 $\text{impl}: \text{LLL-impl-inv state } i \ \text{fs}$
and $\text{inv}: \text{LLL-invariant True } i \ \text{fs}$
and $\text{mu-small}: \mu\text{-small-row } i \ \text{fs } j$
and $\text{res}: \text{LLL-Impl.basis-reduction-add-rows-loop } n \ \text{state } i \ j$
 $(\text{map } (!) \ \text{fs}) (\text{rev } [0 \ ..< j]) = \text{state}'$
 $(\text{is } \text{LLL-Impl.basis-reduction-add-rows-loop } n \ \text{state } i \ j \ (\text{?mapf } \text{fs } j) = -)$
and $j: j \leq i$
and $i: i < m$
and $\text{fs}': \text{fs}' = \text{fs-state state}'$
shows
 $\text{LLL-impl-inv state}' \ i \ \text{fs}'$
 $\text{basis-reduction-add-rows-loop } i \ \text{fs } j = \text{fs}'$
<proof>

lemma *basis-reduction-add-rows-loop*: **assumes**
 $\text{impl}: \text{LLL-impl-inv state } i \ \text{fs}$
and $\text{inv}: \text{LLL-invariant True } i \ \text{fs}$
and $\text{mu-small}: \mu\text{-small-row } i \ \text{fs } j$
and $\text{res}: \text{LLL-Impl.basis-reduction-add-rows-loop } n \ \text{state } i \ j$
 $(\text{map } (!) \ \text{fs}) (\text{rev } [0 \ ..< j]) = \text{state}'$
 $(\text{is } \text{LLL-Impl.basis-reduction-add-rows-loop } n \ \text{state } i \ j \ (\text{?mapf } \text{fs } j) = -)$
and $j: j \leq i$
and $i: i < m$
and $\text{fs}': \text{fs}' = \text{fs-state state}'$

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'
(proof)

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)

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'
(proof)

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)

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 $fs': fs' = fs\text{-state } state'$

shows

LLL-impl-inv $state' i' fs'$

LLL-invariant $upw' i' fs'$

LLL-measure $i' fs' < LLL\text{-measure } i fs$

basis-reduction-swap $i fs = (upw', i', fs')$

<proof>

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* $\alpha n m upw i state = (upw', i', state')$

and $i: i < m$

and $fs': fs' = fs\text{-state } state'$

shows

LLL-impl-inv $state' i' fs'$

basis-reduction-step $upw i fs = (upw', i', fs')$

<proof>

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* $\alpha n m upw i state = (upw', i', state')$

and $i: i < m$

and $fs': fs' = fs\text{-state } state'$

shows

LLL-impl-inv $state' i' fs'$

LLL-invariant $upw' i' fs'$

LLL-measure $i' fs' < LLL\text{-measure } i fs$

basis-reduction-step $upw i fs = (upw', i', fs')$

<proof>

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* $\alpha n m upw i state = state'$

and $fs': fs' = fs\text{-state } state'$

shows *LLL-impl-inv* $state' m fs'$

basis-reduction-main $(upw, i, fs) = fs'$

<proof>

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* $\alpha n m upw i state = state'$

and $fs': fs' = fs\text{-state } state'$

shows

LLL-invariant $True m fs'$

LLL-impl-inv $state' m fs'$

basis-reduction-main (*upw,i,fs*) = *fs'*
 ⟨*proof*⟩

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*⟩

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*
 ⟨*proof*⟩

lemma *reduce-basis-impl: LLL-Impl.reduce-basis* α *fs-init = reduce-basis*
 ⟨*proof*⟩

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
 ⟨*proof*⟩

lemma *short-vector-impl: LLL-Impl.short-vector* α *fs-init = short-vector*
 ⟨*proof*⟩

lemma *short-vector: assumes res: LLL-Impl.short-vector* 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} \implies rat-of-int (sq-norm v) \leq $\alpha^{\wedge}(m - 1) * \text{rat-of-int (sq-norm h)}$*
v \neq 0_v j
 ⟨*proof*⟩

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

begin

definition *round-num-denom-cost* :: *int* \Rightarrow *int* \Rightarrow *int* *cost* **where**

round-num-denom-cost *n d* = $((2 * n + d) \text{ div } (2 * d), 4) - 4$ 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

<proof>

context *LLL-with-assms*

begin

context

assumes *α-gt*: $\alpha > 4/3$ **and** *m0*: $m \neq 0$

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*) \leq *sum* (λ *j*. (2 * *n* + 4 + 3 * (*Suc j*))) {0..<*j*}

<proof>

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*⟩

definition *swap-mu-cost* :: *int iarray iarray ⇒ nat ⇒ int ⇒ int ⇒ int ⇒ int ⇒ int ⇒ int iarray iarray cost* **where**

swap-mu-cost dm u i dm u-i-im1 dim1 di dsi = (let im1 = i - 1;

res = IArray.of-fun (λ ii. if ii < im1 then dm u !! ii else

if ii > i then let dm u-ii = dm u !! ii in

IArray.of-fun (λ j. let dm u-ii-j = dm u-ii !! j in — 8 arith. operations

for whole line

*if j = i then (ds i * dm u-ii !! im1 - dm u-i-im1 * dm u-ii-j) div di —*

4 arith. operations for this entry

*else if j = im1 then (dm u-i-im1 * dm u-ii-j + dm u-ii !! i * dim1) div*

di — 4 arith. operations for this entry

else dm u-ii-j) ii else

if ii = i then let mu-im1 = dm u !! im1 in

IArray.of-fun (λ j. if j = im1 then dm u-i-im1 else mu-im1 !! j) ii

else IArray.of-fun (λ j. dm u !! 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 dm u i dm u-i-im1 dim1 di dsi) = swap-mu m dm u i dm u-i-im1
dim1 di dsi

*cost (swap-mu-cost dm u i dm u-i-im1 dim1 di dsi) ≤ 8 * (m - Suc i)*

⟨*proof*⟩

definition *basis-reduction-swap-cost* **where**

basis-reduction-swap-cost i state = (let

di = d-state state i;

ds i = d-state state (Suc i);

dim1 = d-state state (i - 1);

fi = fi-state state;

fi m1 = fi m1-state state;

dm u-i-im1 = dm u-ij-state state i (i - 1);

fi' = fi m1;

fi m1' = fi;

$di' = (dsi * dim1 + dmu-i-im1 * dmu-i-im1) \text{ div } di$; — 4 arith. operations
 $local-cost = 4$
in (case state of (f,dmus,djs) \Rightarrow
 case swap-mu-cost dmus i dmu-i-im1 dim1 di dsi of
 (swap-res, swap-cost) \Rightarrow
 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*:

$result \ (basis-reduction-swap-cost \ i \ state) = LLL-Impl.basis-reduction-swap \ m \ i$
 $state$
 $cost \ (basis-reduction-swap-cost \ i \ state) \leq 8 * (m - Suc \ i) + 4$
 <proof>

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 \leq 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) \Rightarrow (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 $result \ (basis-reduction-step-cost \ upw \ i \ state) = LLL-Impl.basis-reduction-step$
 $\alpha \ n \ m \ upw \ i \ state \ (\mathbf{is} \ ?g1)$
 $cost \ (basis-reduction-step-cost \ upw \ i \ state) \leq body-cost \ (\mathbf{is} \ ?g2)$
 <proof>

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) \leq c + body-cost * num-loops (**is**
 ?g2)

<proof>

context fixes

fs :: int vec iarray

begin

fun sigma-array-cost **where**

sigma-array-cost dmus dmusi dmsuj dll l = (if l = 0 then (dmusi !! l * dmsuj !!
 l, 1)

else let l1 = l - 1; dll1 = dmus !! l1 !! l1;

(sig, cost-rec) = sigma-array-cost dmus dmusi dmsuj dll1 l1;

res = (dll * sig + dmusi !! l * dmsuj !! 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 dmsuj dll l) = sigma-array dmus dmusi
 dmsuj dll l

cost (sigma-array-cost dmus dmusi dmsuj dll l) \leq 4 * l + 1

<proof>

function dmU-array-row-main-cost **where**

dmU-array-row-main-cost fi i dmus j = (if j \geq 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))

<proof>

termination $\langle proof \rangle$

declare $dmu\text{-array}\text{-row}\text{-main}\text{-cost}\text{-simps}[simp\ del]$

lemma $dmu\text{-array}\text{-row}\text{-main}\text{-cost}$: **assumes** $j \leq i$

shows $result\ (dmu\text{-array}\text{-row}\text{-main}\text{-cost}\ fi\ i\ dmus\ j) = dmu\text{-array}\text{-row}\text{-main}\ fs\ fi\ i\ dmus\ j$

$cost\ (dmu\text{-array}\text{-row}\text{-main}\text{-cost}\ fi\ i\ dmus\ j) \leq (\sum\ jj \in \{j \ ..< i\}. 2 * n + 2 + 4 * jj + 1)$
 $\langle proof \rangle$

definition $dmu\text{-array}\text{-row}\text{-cost}$ **where**

$dmu\text{-array}\text{-row}\text{-cost}\ dmus\ i = (let\ fi = fs\ !!\ i;$
 $sp = fi \cdot fs\ !!\ 0 - 2n\ arith.\ operations;$
 $local\text{-cost} = 2 * n;$
 $(res, main\text{-cost}) = dmu\text{-array}\text{-row}\text{-main}\text{-cost}\ fi\ i\ (iarray\text{-append}\ dmus\ (IArray\ [sp]))\ 0\ in$
 $(res, local\text{-cost} + main\text{-cost}))$

lemma $dmu\text{-array}\text{-row}\text{-cost}$:

$result\ (dmu\text{-array}\text{-row}\text{-cost}\ dmus\ i) = dmu\text{-array}\text{-row}\ fs\ dmus\ i$
 $cost\ (dmu\text{-array}\text{-row}\text{-cost}\ dmus\ i) \leq 2 * n + (2 * n + 1 + 2 * i) * i$
 $\langle proof \rangle$

function $dmu\text{-array}\text{-cost}$ **where**

$dmu\text{-array}\text{-cost}\ dmus\ i = (if\ i \geq m\ then\ (dmus, 0)\ else$
 $let\ (dmus', cost\text{-row}) = dmu\text{-array}\text{-row}\text{-cost}\ dmus\ i;$
 $(res, cost\text{-rec}) = dmu\text{-array}\text{-cost}\ dmus'\ (Suc\ i)$
 $in\ (res, cost\text{-row} + cost\text{-rec}))$
 $\langle proof \rangle$

termination $\langle proof \rangle$

declare $dmu\text{-array}\text{-cost}\text{-simps}[simp\ del]$

lemma $dmu\text{-array}\text{-cost}$: **assumes** $i \leq m$

shows $result\ (dmu\text{-array}\text{-cost}\ dmus\ i) = dmu\text{-array}\ fs\ m\ dmus\ i$
 $cost\ (dmu\text{-array}\text{-cost}\ dmus\ i) \leq (\sum\ ii \in \{i \ ..< m\}. 2 * n + (2 * n + 1 + 2 * ii) * ii)$
 $\langle proof \rangle$

end

definition $d\mu\text{-impl}\text{-cost} :: int\ vec\ list \Rightarrow int\ iarray\ iarray\ cost$ **where**

$d\mu\text{-impl}\text{-cost}\ fs = dmu\text{-array}\text{-cost}\ (IArray\ fs)\ (IArray\ [])\ 0$

lemma $d\mu\text{-impl}\text{-cost}$: $result\ (d\mu\text{-impl}\text{-cost}\ fs\ init) = d\mu\text{-impl}\ fs\ init$

$cost\ (d\mu\text{-impl}\text{-cost}\ fs\ init) \leq m * (m * (m + n + 2) + 2 * n + 1)$
 $\langle proof \rangle$

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

definition $initial-state-cost fs = (let$
 $(dmus, cost) = d\mu-impl-cost fs;$
 $ds = IArray.of-fun (\lambda i. if i = 0 then 1 else let i1 = i - 1 in dmus !! i1 !! i1)$
 $(Suc m);$
 $dmus' = IArray.of-fun (\lambda i. let row-i = dmus !! i in$
 $IArray.of-fun (\lambda j. row-i !! j) i) m$
 $in ((([], fs), dmus', ds), cost) :: LLL-dmu-d-state cost)$

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

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

lemma $initial-state-cost$: $result (initial-state-cost fs-init) = initial-state m fs-init$
 $(is ?g1)$
 $cost (initial-state-cost fs-init) \leq initial-gso-cost (is ?g2)$
 $\langle proof \rangle$

lemma $basis-reduction-cost$:
 $result (basis-reduction-cost fs-init) = basis-reduction \alpha n fs-init (is ?g1)$
 $cost (basis-reduction-cost fs-init) \leq initial-gso-cost + body-cost * num-loops (is$
 $?g2)$
 $\langle proof \rangle$

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 $reduce-basis-cost$:
 $result (reduce-basis-cost fs-init) = LLL-Impl.reduce-basis \alpha fs-init (is ?g1)$
 $cost (reduce-basis-cost fs-init) \leq initial-gso-cost + body-cost * num-loops (is ?g2)$
 $\langle proof \rangle$

lemma mn : $m \leq n$
 $\langle proof \rangle$

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

lemma $reduce-basis-cost-expanded$:

assumes $Lg \geq \text{nat } \lceil \log (\text{of-rat } (4 * \alpha / (4 + \alpha))) N \rceil$
shows $\text{cost } (\text{reduce-basis-cost } \text{fs-init})$
 $\leq 4 * Lg * m * m * m * n$
 $+ 4 * Lg * m * m * m * m$
 $+ 16 * Lg * m * m * m$
 $+ 4 * Lg * m * m$
 $+ 3 * m * m * m$
 $+ 3 * m * m * n$
 $+ 10 * m * m$
 $+ 2 * n * m$
 $+ 3 * m$
(is $\text{?cost} \leq \text{?exp } Lg$
 $\langle \text{proof} \rangle$

lemma *reduce-basis-cost-0*: **assumes** $m = 0$
shows $\text{cost } (\text{reduce-basis-cost } \text{fs-init}) = 0$
 $\langle \text{proof} \rangle$

lemma *reduce-basis-cost-N*:
assumes $Lg \geq \text{nat } \lceil \log (\text{of-rat } (4 * \alpha / (4 + \alpha))) N \rceil$
and $0: Lg > 0$
shows $\text{cost } (\text{reduce-basis-cost } \text{fs-init}) \leq 49 * m \wedge 3 * n * Lg$
 $\langle \text{proof} \rangle$

lemma *reduce-basis-cost-M*:
assumes $Lg \geq \text{nat } \lceil \log (\text{of-rat } (4 * \alpha / (4 + \alpha))) (M * n) \rceil$
and $0: Lg > 0$
shows $\text{cost } (\text{reduce-basis-cost } \text{fs-init}) \leq 98 * m \wedge 3 * n * Lg$
 $\langle \text{proof} \rangle$

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  $\Rightarrow$  nat  $\Rightarrow$  int vec list  $\Rightarrow$  bool where
  f-bound outside ii fs = ( $\forall i < m. \text{sq-norm } (fs ! i) \leq (\text{if } i \neq ii \vee \text{outside then int } (N * m) \text{ else int } (4 \wedge (m - 1) * N \wedge m * m * m))$ )
```

```
definition g-bnd :: rat  $\Rightarrow$  int vec list  $\Rightarrow$  bool where
  g-bnd B fs = ( $\forall i < m. \text{sq-norm } (gso fs i) \leq B$ )
```

```
definition  $\mu$ -bound-row fs bnd i = ( $\forall j \leq i. (\mu fs i j) \wedge 2 \leq bnd$ )
```

```
abbreviation  $\mu$ -bound-row-inner fs i j  $\equiv$   $\mu$ -bound-row fs ( $4 \wedge (m - 1 - j) * \text{of-nat } (N \wedge (m - 1) * m)$ ) i
```

```
definition LLL-bound-invariant outside upw i fs =
  (LLL-invariant upw i fs  $\wedge$  f-bound outside i fs  $\wedge$  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
   $\langle$ proof $\rangle$ 
```

```
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
   $\langle$ proof $\rangle$ 
```

```
lemma  $\mu$ -bound-rowI: assumes  $\bigwedge j. j \leq i \implies (\mu fs i j) \wedge 2 \leq bnd$ 
shows  $\mu$ -bound-row fs bnd i
   $\langle$ proof $\rangle$ 
```

```
lemma  $\mu$ -bound-rowD: assumes  $\mu$ -bound-row fs bnd i j  $\leq$  i
shows  $(\mu fs i j) \wedge 2 \leq bnd$ 
   $\langle$ proof $\rangle$ 
```

```
lemma  $\mu$ -bound-row-1: assumes  $\mu$ -bound-row fs bnd i
shows bnd  $\geq$  1
```

<proof>

lemma *reduced- μ -bound-row*: **assumes** *red: reduced fs i*
 and *ii: ii < i*
shows *μ -bound-row fs 1 ii*
<proof>

lemma *f-bound-True-arbitrary*: **assumes** *f-bound True ii fs*
 shows *f-bound outside j fs*
<proof>

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*
<proof>

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 $\|fs ! i\|^2 \leq$ of-nat (Suc i * N) * bnd*
<proof>
end

lemma *increase-i-bound*: **assumes** *LLL: LLL-bound-invariant True upw i fs*
 and *i: i < m*
 and *upw: upw \implies i = 0*
 and *red-i: i \neq 0 \implies sq-norm (gso fs (i - 1)) \leq α * sq-norm (gso fs i)*
shows *LLL-bound-invariant True True (Suc i) fs*
<proof>

Addition step preserves *LLL-bound-invariant False*

lemma *basis-reduction-add-row-main-bound*: **assumes** *Liniv: LLL-bound-invariant False True i fs*
 and *i: i < m* **and** *j: j < i*
 and *c: c = round (μ 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>
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*)

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*)

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 \leq i*
shows *LLL-bound-invariant True False i fs'*
(*proof*)

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*)

lemma *g-bnd-swap*:
assumes *i: i < m i \neq 0*
and *Linv: LLL-invariant-weak fs*
and *mu-F1-i: $|\mu fs i (i-1)| \leq 1 / 2$*
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*)

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* \neq 0
shows *LLL-bound-invariant* *True* *upw'* *i'* *fs'*
<*proof*>

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*>

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'*
<*proof*>

lemma *LLL-inv-initial-state-bound*: *LLL-bound-invariant* *True* *True* 0 *fs-init*
<*proof*>

lemma *reduce-basis-bound*: **assumes** *res*: *reduce-basis* = *fs*
shows *LLL-bound-invariant* *True* *True* *m* *fs*
<*proof*>

9.5.2 Bound extracted from *LLL-bound-invariant*.

fun *f-bnd* :: *bool* \Rightarrow *nat* **where**
f-bnd *False* = $2 \wedge (m - 1) * N \wedge m * m$
| *f-bnd* *True* = $N * m$

lemma *f-bnd-mono*: *f-bnd* *outside* \leq *f-bnd* *False*
<*proof*>

lemma *aux-bnd-mono*: $N * m \leq (4 \wedge (m - 1) * N \wedge m * m * m)$
<*proof*>

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 $|fs ! i \$ j| \leq f\text{-bnd } outside$
<*proof*>

lemma *LLL-gso-bound*:

assumes $i: i < m$ **and** $j: j < n$
and *quot*: *quotient-of* ($gso\ fs\ i\ \$\ j$) = ($num, denom$)
shows $|num| \leq N^m$
and $|denom| \leq N^m$
<proof>

lemma *LLL-f-bound*:

assumes $i: i < m$ **and** $j: j < n$
shows $|fs\ !\ i\ \$\ j| \leq N^m * 2^{(m-1)*m}$
<proof>

lemma *LLL-d-bound*:

assumes $i: i \leq m$
shows $abs\ (d\ fs\ i) \leq N^i \wedge abs\ (d\ fs\ i) \leq N^m$
<proof>

lemma *LLL-mu-abs-bound*:

assumes $i: i < m$
and $j: j < i$
shows $|\mu\ fs\ i\ j| \leq rat\ of\ nat\ (N^{(m-1)*2^{(m-1)*m}})$
<proof>

lemma *LLL-dmu-bound*:

assumes $i: i < m$ **and** $j: j < i$
shows $abs\ (d\mu\ fs\ i\ j) \leq N^{(2*(m-1))*2^{(m-1)*m}}$
<proof>

lemma *LLL-mu-num-denom-bound*:

assumes $i: i < m$
and *quot*: *quotient-of* ($\mu\ fs\ i\ j$) = ($num, denom$)
shows $|num| \leq N^{(2*m)*2^m}$
and $|denom| \leq N^m$
<proof>

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** $m: m \neq 0$ $n: n \neq 0$ **and** $N: N > 0$

shows $\log\ 2\ N \leq 2 * \log\ 2\ (M * n)$
<proof>

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 $x: x \in \{of-int (fs ! i \$ j), gso fs i \$ j, \mu fs i j\}$
and $quot: quotient-of x = (num, denom)$
and $number: number \in \{num, denom\}$
and $number0: number \neq 0$
shows $\log 2 |number| \leq 2 * m * \log 2 N + m + \log 2 m$
 $\log 2 |number| \leq 4 * m * \log 2 (M * n) + m + \log 2 m$
 $\langle proof \rangle$

And a combined size bound for an integer implementation which stores values $f_i, d_{j+1}\mu_{ij}$ and d_i .

interpretation $fs: fs-int-indpt n fs-init$
 $\langle proof \rangle$

lemma $fs-gs-N-N'$: **assumes** $m \neq 0$
shows $fs.gs.N = of-nat N$
 $\langle proof \rangle$

lemma $fs-gs-N-N$: $m \neq 0 \implies real-of-rat fs.gs.N = real N$
 $\langle proof \rangle$

lemma $combined-size-bound-gso-integer$:
assumes $x \in$
 $\{fs.\mu' i j \mid i j. j \leq i \wedge i < m\} \cup$
 $\{fs.\sigma s l i j \mid i j l. i < m \wedge j \leq i \wedge l < j\}$
and $m: m \neq 0$ **and** $x \neq 0$ $n \neq 0$
shows $\log 2 |real-of-int x| \leq (6 + 6 * m) * \log 2 (M * n) + \log 2 m + m$
 $\langle proof \rangle$

lemma $combined-size-bound-integer'$:
assumes $x: x \in \{fs ! i \$ j \mid i j. i < m \wedge j < n\}$
 $\cup \{d\mu fs i j \mid i j. j < i \wedge i < m\}$
 $\cup \{d fs i \mid i. i \leq m\}$
(is $x \in ?fs \cup ?d\mu \cup ?d$ **)**
and $m: m \neq 0$ **and** $n: n \neq 0$
shows $abs x \leq N \wedge (2 * m) * 2 \wedge m * m$
 $x \neq 0 \implies \log 2 |x| \leq 2 * m * \log 2 N + m + \log 2 m$ **(is** $- \implies ?l1 \leq ?b1$ **)**
 $x \neq 0 \implies \log 2 |x| \leq 4 * m * \log 2 (M * n) + m + \log 2 m$ **(is** $- \implies - \leq ?b2$ **)**
 $\langle proof \rangle$

lemma $combined-size-bound-integer$:
assumes $x: x \in$
 $\{fs ! i \$ j \mid i j. i < m \wedge j < n\}$
 $\cup \{d\mu fs i j \mid i j. j < i \wedge i < m\}$
 $\cup \{d fs i \mid i. i \leq m\}$
 $\cup \{fs.\mu' i j \mid i j. j \leq i \wedge i < m\}$
 $\cup \{fs.\sigma s l i j \mid i j l. i < m \wedge j \leq i \wedge l < j\}$
(is $?x \in ?s1 \cup ?s2 \cup ?s3 \cup ?g1 \cup ?g2$ **)**
and $m: m \neq 0$ **and** $n: n \neq 0$ **and** $x \neq 0$ **and** $0 < M$


```

shows  $\log 2 |x| \leq (6 + 6 * m) * \log 2 (M * n) + \log 2 m + m$ 
⟨proof⟩

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 basis, 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  $n A B I = (\text{case } \text{gauss-jordan } A B \text{ of}$ 
   $(C,D) \Rightarrow C = I \wedge \text{list-all is-int-rat } (\text{concat } (\text{mat-to-list } D)))$ 

```

```

definition integer-equivalent  $n fs gs = (\text{let}$ 

```

```

   $fs' = \text{map-mat rat-of-int } (\text{mat-of-cols } n fs);$ 

```

```

   $gs' = \text{map-mat rat-of-int } (\text{mat-of-cols } n gs);$ 

```

```

   $I = 1_m n$ 

```

```

   $\text{in } \text{gauss-jordan-integer-inverse } n fs' gs' I \wedge \text{gauss-jordan-integer-inverse } n gs' fs'$ 
 $I)$ 

```

```

context vec-module

```

```

begin

```

```

lemma mat-mult-sub-lattice: assumes  $fs: \text{set } fs \subseteq \text{carrier-vec } n$ 

```

```

and  $gs: \text{set } gs \subseteq \text{carrier-vec } n$ 

```

```

and  $A: A \in \text{carrier-mat } (\text{length } fs) (\text{length } gs)$ 

```

```

and  $\text{prod}: \text{mat-of-rows } n fs = \text{map-mat of-int } A * \text{mat-of-rows } n gs$ 

```

```

shows  $\text{lattice-of } fs \subseteq \text{lattice-of } gs$ 

```

```

⟨proof⟩

```

end

context *LLL-with-assms*

begin

lemma *mult-left-identity*:

defines $B \equiv (\text{map-mat rat-of-int } (\text{mat-of-rows } n \text{ fs-init}))$

assumes $P\text{-carrier}[simp]: P \in \text{carrier-mat } m \ m$

and $PB: P * B = B$

shows $P = 1_m \ m$

<proof>

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** $gs: \text{set } gs \subseteq \text{carrier-vec } n$

and $len': \text{length } gs = m$

and $U1: U1 \in \text{carrier-mat } m \ m$

and $U2: U2 \in \text{carrier-mat } m \ m$

and $prod1: \text{mat-of-rows } n \text{ fs-init} = U1 * \text{mat-of-rows } n \ gs$

and $prod2: \text{mat-of-rows } n \ gs = U2 * \text{mat-of-rows } n \ \text{fs-init}$

shows $\text{lattice-of } gs = \text{lattice-of } \text{fs-init} \ \text{LLL-with-assms } n \ m \ gs \ \alpha$

<proof>

lemma *gauss-jordan-integer-inverse*: **fixes** $fs \ gs :: \text{int vec list}$

assumes $gs: \text{set } gs \subseteq \text{carrier-vec } n$

and $len\text{-}gs: \text{length } gs = n$

and $fs: \text{set } fs \subseteq \text{carrier-vec } n$

and $len\text{-}fs: \text{length } fs = n$

and $gauss: \text{gauss-jordan-integer-inverse } n \ (\text{map-mat rat-of-int } (\text{mat-of-cols } n \ fs))$

$(\text{map-mat rat-of-int } (\text{mat-of-cols } n \ gs)) \ (1_m \ n) \ (\text{is gauss-jordan-integer-inverse } - \ ?fs \ ?gs \ -)$

shows $\exists U. U \in \text{carrier-mat } n \ n \wedge \text{mat-of-rows } n \ gs = U * \text{mat-of-rows } n \ fs$

<proof>

lemma *LLL-change-basis-mat-inverse*: **assumes** $gs: \text{set } gs \subseteq \text{carrier-vec } n$

and $len': \text{length } gs = n$

and $m = n$

and $eq: \text{integer-equivalent } n \ \text{fs-init } gs$

shows $\text{lattice-of } gs = \text{lattice-of } \text{fs-init} \ \text{LLL-with-assms } n \ m \ gs \ \alpha$

<proof>

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))

instance bool :: prime-card
 ⟨proof⟩

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>

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} \implies rat-of-int (sq-norm v) \leq α \wedge (m - 1) * rat-of-int (sq-norm h)*
v \neq 0_v j
<proof>
end

Unspecified constant to easily enable/disable external lll solver in generated code

consts *enable-external-lll-solver :: bool*

definition *short-vector-hybrid :: rat \Rightarrow int vec list \Rightarrow int vec* **where**
short-vector-hybrid = (if enable-external-lll-solver then short-vector-external else short-vector)

definition *reduce-basis-hybrid :: rat \Rightarrow int vec list \Rightarrow 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} \implies rat-of-int (sq-norm v) \leq α \wedge (m - 1) * rat-of-int (sq-norm h)*
v \neq 0_v j
<proof>

lemma *reduce-basis-hybrid*: **assumes** *res: reduce-basis-hybrid* α *fs-init = fs*
shows *reduced fs m LLL-invariant True m fs*
<proof>
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)
  ⟨proof⟩

```

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 `fpdll` solver. For `eta` we use the default value of `fpdll`, and `delta` is chosen so that the required precision of `alpha` will be guaranteed. We use the command-line option `-bv` 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;

fpdll-command :: String;
fpdll-command = fpdll;

```

```

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 []);
showmat :: [[Integer]] -> ByteString;
showmat matA = (pack []) 'mappend' intercalate (pack '\n ') (map showrow matA)
  'mappend' (pack []);

data Mode = Simple | Certificate;

flags :: Mode -> String;
flags Simple = b;
flags Certificate = bv;

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 = CreatePipe,
  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 =<< unsafeIO 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.