Two algorithms based on modular arithmetic: lattice basis reduction and Hermite normal form computation*

Ralph Bottesch Jose Divasón René Thiemann

March 17, 2025

Abstract

We verify two algorithms for which modular arithmetic plays an essential role: Storjohann's variant of the LLL lattice basis reduction algorithm and Kopparty's algorithm for computing the Hermite normal form of a matrix. To do this, we also formalize some facts about the modulo operation with symmetric range. Our implementations are based on the original papers, but are otherwise efficient. For basis reduction we formalize two versions: one that includes all of the optimizations/heuristics from Storjohann's paper, and one excluding a heuristic that we observed to often decrease efficiency. We also provide a fast, self-contained certifier for basis reduction, based on the efficient Hermite normal form algorithm.

Contents

*Supported by FWF (Austrian Science Fund) project Y757 and by project MTM2017-88804-P (Spanish Ministry of Science and Innovation).

1 Missing Matrix Operations

In this theory we provide an operation that can change a single row in a matrix efficiently, and all other rows in the matrix implementation will be reused.

theory *Matrix-Change-Row*
imports
Jordan-Normal-Form.Matrix-IArray-Impl
Polynomial-Interpolation.Missing-Unsorted
begin

definition *change-row* :: *nat* ⇒ (*nat* ⇒ *'a* ⇒ *'a*) ⇒ *'a mat* ⇒ *'a mat* **where**
change-row k f A = *mat* (*dim-row A*) (*dim-col A*) (λ (*i*,*j*).
if i = *k then f j* (*A* $$ (*k*,*j*)) *else A* $$ (*i*,*j*))

lemma *change-row-carrier*[*simp*]:
(*change-row k f A* ∈ *carrier-mat nr nc*) = (*A* ∈ *carrier-mat nr nc*)
dim-row (*change-row k f A*) = *dim-row A*
dim-col (*change-row k f A*) = *dim-col A*
unfolding *change-row-def carrier-mat-def* **by** *auto*

lemma *change-row-index*[*simp*]: $A \in$ *carrier-mat nr nc* $\Longrightarrow$ $i <$ *nr* $\Longrightarrow$ $j <$ *nc* $\Longrightarrow$
change-row k f A $$ (i,j) = (if i = k then f j (A $$ (k,j)) else A $$ (i,j))
$i <$ *dim-row A* $\Longrightarrow$ $j <$ *dim-col A* $\Longrightarrow$ *change-row k f A $$ (i,j) = (if i = k then f j (A $$ (k,j)) else A $$ (i,j))*
unfolding *change-row-def* **by** *auto*

lift-definition *change-row-impl* :: *nat* $\Rightarrow$ (*nat* $\Rightarrow$ $'a$ $\Rightarrow$ $'a$) $\Rightarrow$ $'a$ *mat-impl* $\Rightarrow$ $'a$ *mat-impl* **is**
λ *k f* (*nr*,*nc*,*A*). *let Ak = IArray.sub A k*; *Arows = IArray.list-of A*;
Ak' = IArray.IArray (*map* (λ (*i*,*c*). *f i c*) (*zip* [*0* ..< *nc*] (*IArray.list-of Ak*)));
A' = IArray.IArray (*Arows* [*k* := *Ak'*])
in (*nr*,*nc*,*A'*)
proof (*auto*, *goal-cases*)
case (*1 k f nc b row*)
show *?case*
proof (*cases b*)
case (*IArray rows*)
with *1* **have** *row* $\in$ *set rows* $\vee$ *k* $<$ *length rows*
$\wedge$ *row = IArray* (*map* (λ (*i*,*c*). *f i c*) (*zip* [*0* ..< *nc*] (*IArray.list-of* (*rows* ! *k*))))
by (*cases k* $<$ *length rows*, *auto simp*: *set-list-update dest*: *in-set-takeD in-set-dropD*)
with *1 IArray* **show** *?thesis* **by** (*cases*, *auto*)
qed
qed

lemma *change-row-code*[*code*]: *change-row k f* (*mat-impl A*) = (*if k* $<$ *dim-row-impl A*
then mat-impl (*change-row-impl k f A*)
else Code.abort (*STR ''index out of bounds in change-row''*) (λ -. *change-row k f* (*mat-impl A*)))
(**is** *?l = ?r*)
proof (*cases k* $<$ *dim-row-impl A*)
case *True*
hence *id*: *?r = mat-impl* (*change-row-impl k f A*) **by** *simp*
show *?thesis* **unfolding** *id* **unfolding** *change-row-def*
proof (*rule eq-matI*, *goal-cases*)
case (*1 i j*)
thus *?case* **using** *True*
by (*transfer*, *auto simp*: *mk-mat-def*)
qed (*transfer*, *auto*)+
qed *simp*

end

2 Signed Modulo Operation

theory *Signed-Modulo*
imports
Berlekamp-Zassenhaus.Poly-Mod
Sqrt-Babylonian.Sqrt-Babylonian-Auxiliary
begin

The upcoming definition of symmetric modulo is different to the HOL-Library-Signed_Division.smod, since here the modulus will be in range $\{-m/2, ..., m/2\}$, whereas there -1 symmod m = m - 1.

The advantage of have range $\{-m/2, ..., m/2\}$ is that small negative numbers are represented by small numbers.

One limitation is that the symmetric modulo is only working properly, if the modulus is a positive number.

definition *sym-mod* :: *int* ⇒ *int* ⇒ *int* (**infixl** ‹*symmod*› *70*) **where**
sym-mod x y = *poly-mod.inv-M y* (*x mod y*)

lemma *sym-mod-code*[*code*]: *sym-mod x y* = (*let m* = *x mod y*
in if m + *m* ≤ *y then m else m* − *y*)
unfolding *sym-mod-def poly-mod.inv-M-def Let-def* **..**

lemma *sym-mod-zero*[*simp*]: *n symmod 0* = *n n* > *0* ⟹ *0 symmod n* = *0*
unfolding *sym-mod-def poly-mod.inv-M-def* **by** *auto*

lemma *sym-mod-range*:
‹*x symmod y* ∈ {− ((*y* − *1*) *div 2*) .. *y div 2*}› **if** ‹*y* > *0*›
proof −
from *that* **have** ‹*x mod y* < *y*›
by (*rule pos-mod-bound*)
then have ‹*x mod y* − *y* < *0*›
by *simp*
moreover from *that* **have** ‹− ((*y* − *1*) *div 2*) ≤ *0*› ‹*0* ≤ *x mod y*›
by *simp-all*
then have ‹− ((*y* − *1*) *div 2*) ≤ *x mod y*›
by (*rule order-trans*)
ultimately show *?thesis*
by (*auto simp add*: *sym-mod-def poly-mod.inv-M-def*)
qed

The range is optimal in the sense that exactly y elements can be represented.

lemma *card-sym-mod-range*: *y* > *0* ⟹ *card* {− ((*y* − *1*) *div 2*) .. *y div 2*} = *y*
by *simp*

lemma *sym-mod-abs*: *y* > *0* ⟹ |*x symmod y*| < *y*
y ≥ *1* ⟹ |*x symmod y*| ≤ *y div 2*
using *sym-mod-range*[*of y x*] **by** *auto*

lemma *sym-mod-sym-mod*[*simp*]: *x symmod y symmod y = x symmod* (*y* :: *int*)
unfolding *sym-mod-def* **using** *poly-mod.M-def poly-mod.M-inv-M-id* **by** *auto*

lemma *sym-mod-diff-eq*: (*a symmod c − b symmod c*) *symmod c* = (*a − b*) *symmod c*
unfolding *sym-mod-def*
by (*metis mod-diff-cong mod-mod-trivial poly-mod.M-def poly-mod.M-inv-M-id*)

lemma *sym-mod-sym-mod-cancel*: *c dvd b* ⟹ *a symmod b symmod c* = *a symmod c*
using *mod-mod-cancel*[*of c b*] **unfolding** *sym-mod-def*
by (*metis poly-mod.M-def poly-mod.M-inv-M-id*)

lemma *sym-mod-diff-right-eq*: (*a − b symmod c*) *symmod c* = (*a − b*) *symmod c*
using *sym-mod-diff-eq* **by** (*metis sym-mod-sym-mod*)

lemma *sym-mod-mult-right-eq*: *a* ∗ (*b symmod c*) *symmod c* = *a* ∗ *b symmod c*
unfolding *sym-mod-def* **by** (*metis poly-mod.M-def poly-mod.M-inv-M-id mod-mult-right-eq*)

lemma *dvd-imp-sym-mod-0* [*simp*]:
b symmod a = *0* **if** *a* > *0 a dvd b*
unfolding *sym-mod-def poly-mod.inv-M-def* **using** *that* **by** *simp*

lemma *sym-mod-0-imp-dvd* [*dest*!]:
b dvd a **if** *a symmod b* = *0*
using *that* **apply** (*simp add*: *sym-mod-def poly-mod.inv-M-def not-le split*: *if-splits*)
using *pos-mod-bound* [*of b a*] **apply** *auto*
done

definition *sym-div* :: *int* ⇒ *int* ⇒ *int* (**infixl** ‹*symdiv*› *70*) **where**
sym-div x y = (*let d* = *x div y*; *m* = *x mod y in*
if m + *m* ≤ *y then d else d* + *1*)

lemma *of-int-mod-integer*: (*of-int* (*x mod y*) :: *integer*) = (*of-int x* :: *integer*) *mod* (*of-int y*)
using *integer-of-int-eq-of-int modulo-integer.abs-eq* **by** *presburger*

lemma *sym-div-code*[*code*]:
sym-div x y = (*let yy* = *integer-of-int y in*
(*case divmod-integer* (*integer-of-int x*) *yy*
of (*d*, *m*) ⇒ *if m* + *m* ≤ *yy then int-of-integer d else* (*int-of-integer* (*d* + *1*))))
unfolding *sym-div-def Let-def divmod-integer-def split*
apply (*rule if-cong*, *subst of-int-le-iff*[*symmetric*], *unfold of-int-add*)
by (*subst* (*1 2*) *of-int-mod-integer*, *auto*)

lemma *sym-mod-sym-div*: **assumes** *y*: *y* > *0* **shows** *x symmod y* = *x − sym-div x y* ∗ *y*
proof −
let *?z* = *x − y* ∗ (*x div y*)

let *?u* = *y* ∗ (*x div y*)
have *x* = *y* ∗ (*x div y*) + *x mod y* **using** *y* **by** *simp*
hence *id*: *x mod y* = *?z* **by** *linarith*
have *x symmod y* = *poly-mod.inv-M y ?z* **unfolding** *sym-mod-def id* **by** *auto*
also have ... = (*if ?z* + *?z* ≤ *y then ?z else ?z* − *y*) **unfolding** *poly-mod.inv-M-def*
..
also have ... = *x* − (*if* (*x mod y*) + (*x mod y*) ≤ *y then x div y else x div y* + *1*) ∗ *y*
by (*simp add*: *algebra-simps id*)
also have (*if* (*x mod y*) + (*x mod y*) ≤ *y then x div y else x div y* + *1*) = *sym-div x y*
unfolding *sym-div-def Let-def* **..**
finally show *?thesis* **.**
qed

lemma *dvd-sym-div-mult-right* [*simp*]:
(*a symdiv b*) ∗ *b* = *a* **if** *b* > *0 b dvd a*
using *sym-mod-sym-div*[*of b a*] *that* **by** *simp*

lemma *dvd-sym-div-mult-left* [*simp*]:
b ∗ (*a symdiv b*) = *a* **if** *b* > *0 b dvd a*
using *dvd-sym-div-mult-right*[*OF that*] **by** (*simp add*: *ac-simps*)

end

3 Storjohann's Lemma 13

This theory contains the result that one can always perform a mod-operation on the entries of the $d\mu$-matrix.

theory *Storjohann-Mod-Operation*
imports
LLL-Basis-Reduction.LLL-Certification
Signed-Modulo
begin

lemma *map-vec-map-vec*: *map-vec f* (*map-vec g v*) = *map-vec* (*f o g*) *v*
by (*intro eq-vecI*, *auto*)

context *semiring-hom*
begin

lemma *mat-hom-add*: **assumes** *A*: *A* ∈ *carrier-mat nr nc* **and** *B*: *B* ∈ *carrier-mat nr nc*
shows mat_h (*A* + *B*) = mat_h *A* + mat_h *B*
by (*intro eq-matI*, *insert A B*, *auto simp*: *hom-add*)
end

We now start to prove lemma 13 of Storjohann's paper.

```
context
  fixes A I :: 'a :: field mat and n :: nat
  assumes A: A ∈ carrier-mat n n
  and det: det A ≠ 0
  and I: I = the (mat-inverse A)
begin
lemma inverse-via-det: I * A = 1_m n A * I = 1_m n I ∈ carrier-mat n n
  I = mat n n (λ (i,j). det (replace-col A (unit-vec n j) i) / det A)
proof −
  from det-non-zero-imp-unit[OF A det]
  have Unit: A ∈ Units (ring-mat TYPE('a) n n) .
  from mat-inverse(1)[OF A, of n] Unit I have mat-inverse A = Some I
    by (cases mat-inverse A, auto)
  from mat-inverse(2)[OF A this]
  show left: I * A = 1_m n and right: A * I = 1_m n and I: I ∈ carrier-mat n n
    by blast+
  {
    fix i j
    assume i: i < n and j: j < n
    from I i j have cI: col I j $ i = I $$ (i,j) by simp
    from j have uv: unit-vec n j ∈ carrier-vec n by auto
    from j I have col: col I j ∈ carrier-vec n by auto
    from col-mult2[OF A I j, unfolded right] j
    have A *_v col I j = unit-vec n j by simp
    from cramer-lemma-mat[OF A col i, unfolded this cI]
    have I $$ (i,j) = det (replace-col A (unit-vec n j) i) / det A using det by simp
  }
  thus I = mat n n (λ (i,j). det (replace-col A (unit-vec n j) i) / det A)
    by (intro eq-matI, use I in auto)
qed

lemma matrix-for-singleton-entry: assumes i: i < n and
  j: j < n
  and Rdef: R = mat n n ( λ ij. if ij = (i,j) then c :: 'a else 0)
shows mat n n
   (λ(i', j'). if i' = i then c * det (replace-col A (unit-vec n j') j) / det A
       else 0) * A = R
proof −
  note I = inverse-via-det(3)
  have R: R ∈ carrier-mat n n unfolding Rdef by auto
  have (R * I) * A = R * (I * A) using I A R by auto
  also have I * A = 1_m n unfolding inverse-via-det(1) ..
  also have R * ... = R using R by simp
  also have R * I = mat n n (λ (i',j'). row R i' · col I j')
    using I R unfolding times-mat-def by simp
  also have ... = mat n n ( λ (i',j'). if i' = i then c * I $$ (j, j') else 0)
    (is mat n n ?f = mat n n ?g)
  proof −
```

```
    {
      fix i' j'
      assume i': i' < n and j': j' < n
      have ?f (i',j') = ?g (i',j')
      proof (cases i' = i)
        case False
        hence row R i' = 0_v n unfolding Rdef using i'
          by (intro eq-vecI, auto simp: Matrix.row-def)
        thus ?thesis using False i' j' I by simp
      next
        case True
        hence row R i' = c ·_v unit-vec n j unfolding Rdef using i' j' i j
          by (intro eq-vecI, auto simp: Matrix.row-def)
        with True show ?thesis using i' j' I j by simp
      qed
    }
    thus ?thesis by auto
  qed
  finally show ?thesis unfolding inverse-via-det(4) using j
    by (auto intro!: arg-cong[of - - λ x. x * A])
qed
end

lemma (in gram-schmidt-fs-Rn) det-M-1: det (M m) = 1
proof −
  have det (M m) = prod-list (diag-mat (M m))
    by (rule det-lower-triangular[of m], auto simp: μ.simps)
  also have ... = 1
    by (rule prod-list-neutral, auto simp: diag-mat-def μ.simps)
  finally show ?thesis .
qed

context gram-schmidt-fs-int
begin
lemma assumes IM: IM = the (mat-inverse (M m))
  shows inv-mu-lower-triangular: ⋀ k i. k < i ⟹ i < m ⟹ IM $$ (k, i) = 0
  and inv-mu-diag: ⋀ k. k < m ⟹ IM $$ (k, k) = 1
  and d-inv-mu-integer: ⋀ i j. i < m ⟹ j < m ⟹ d i * IM $$ (i,j) ∈ ℤ
  and inv-mu-inverse: IM * M m = 1_m m M m * IM = 1_m m IM ∈ carrier-mat
m m
proof −
  note * = inverse-via-det[OF M-dim(3) - IM, unfolded det-M-1]
  from * show inv: IM * M m = 1_m m M m * IM = 1_m m
    and IM: IM ∈ carrier-mat m m  by auto
  from * have IM-det: IM = mat m m (λ(i, j). det (replace-col (M m) ((unit-vec
m) j) i))
    by auto
  from matrix-equality have IM * FF = IM * ((M m) * Fs) by simp
  also have ... = (IM * M m) * Fs using M-dim(3) IM Fs-dim(3)
```

```
  by (metis assoc-mult-mat)
 also have ... = Fs unfolding inv using Fs-dim(3) by simp
 finally have equality: IM * FF = Fs .
 {
  fix i k
  assume i: k < i i < m
  show IM $$ (k, i) = 0 using i M-dim unfolding IM-det
   by (simp, subst det-lower-triangular[of m], auto simp: replace-col-def μ.simps
diag-mat-def)
 } note IM-lower-triag = this
 {
  fix k
  assume k: k < m
  show IM $$ (k,k) = 1 using k M-dim unfolding IM-det
   by (simp, subst det-lower-triangular[of m], auto simp: replace-col-def μ.simps
diag-mat-def
      intro!: prod-list-neutral)
 } note IM-diag-1 = this
 {
  fix k
  assume k: k < m
  let ?f = λ i. IM $$ (k, i) ·v fs ! i
  let ?sum = M.sumlist (map ?f [0..<m])
  let ?sumk = M.sumlist (map ?f [0..<k])
  have set: set (map ?f [0..<m]) ⊆ carrier-vec n using fs-carrier by auto
  hence sum: ?sum ∈ carrier-vec n by simp
  from set k have setk: set (map ?f [0..<k]) ⊆ carrier-vec n by auto
  hence sumk: ?sumk ∈ carrier-vec n by simp
  from sum have dim-sum: dim-vec ?sum = n by simp
  have gso k = row Fs k using k by auto
  also have ... = row (IM * FF) k unfolding equality ..
  also have IM * FF = mat m n (λ (i,j). row IM i  · col FF j)
   unfolding times-mat-def using IM FF-dim by auto
  also have row ... k = vec n (λ j. row IM k · col FF j)
   unfolding Matrix.row-def using IM FF-dim k by auto
  also have ... = vec n (λ j. ∑ i < m. IM $$ (k, i) * fs ! i $ j)
   by (intro eq-vecI, insert IM k, auto simp: scalar-prod-def Matrix.row-def intro!:
sum.cong)
  also have ... = ?sum
   by (intro eq-vecI, insert IM, unfold dim-sum, subst sumlist-vec-index,
     auto simp: o-def sum-list-sum-nth intro!: sum.cong)
  also have [0..<m] = [0..<k] @ [k] @ [Suc k ..<m] using k
   by (simp add: list-trisect)
  also have M.sumlist (map ?f ...) = ?sumk +
   (?f k + M.sumlist (map ?f [Suc k ..< m]))
   unfolding map-append
   by (subst M.sumlist-append; (subst M.sumlist-append)?, insert k fs-carrier,
auto)
  also have M.sumlist (map ?f [Suc k ..< m]) = 0v n
```

```
    by (rule sumlist-neutral, insert IM-lower-triag, auto)
  also have IM $$ (k,k) = 1 using IM-diag-1[OF k] .
  finally have gso: gso k = ?sumk + fs ! k  using k by simp
  define b where b = vec k (λ j. fs ! j · fs ! k)
  {
    fix j
    assume jk: j < k
    with k have j: j < m by auto
    have fs ! j · gso k = fs ! j · (?sumk + fs ! k)
      unfolding gso by simp
    also have fs ! j · gso k = 0 using jk k
      by (simp add: fi-scalar-prod-gso gram-schmidt-fs.μ.simps)
    also have fs ! j · (?sumk + fs ! k)
      = fs ! j · ?sumk + fs ! j · fs ! k
      by (rule scalar-prod-add-distrib[OF - sumk], insert j k, auto)
    also have fs ! j · fs ! k = b $ j unfolding b-def using jk by simp
    finally have b $ j = − (fs ! j · ?sumk) by linarith
  } note b-index = this
  let ?x = vec k (λ i. − IM $$ (k, i))
  have x: ?x ∈ carrier-vec k by auto
  from k have km: k ≤ m by simp
  have bGx: b = Gramian-matrix fs k *_v (vec k (λ i. − IM $$ (k, i)))
    unfolding Gramian-matrix-alt-alt-def[OF km]
  proof (rule eq-vecI; simp)
    fix i
    assume i: i < k
    have b $ i = − (∑ x←[0..<k]. fs ! i · (IM $$ (k, x) ·_v fs ! x))
      unfolding b-index[OF i]
      by (subst scalar-prod-right-sum-distrib, insert setk i k, auto simp: o-def)
    also have ... = vec k (λj. fs ! i · fs ! j) · vec k (λi. − IM $$ (k, i))
      by (subst (3) scalar-prod-def, insert i k, auto simp: o-def sum-list-sum-nth
simp flip: sum-negf
        intro!: sum.cong)
    finally show b $ i = vec k (λj. fs ! i · fs ! j) · vec k (λi. − IM $$ (k, i)) .
  qed (simp add: b-def)
  have G: Gramian-matrix fs k ∈ carrier-mat k k
    unfolding Gramian-matrix-alt-alt-def[OF km] by simp
  from cramer-lemma-mat[OF G x, folded bGx Gramian-determinant-def]
  have i < k ⟹
    d k * IM $$ (k, i) = − det (replace-col (Gramian-matrix fs k) (vec k (λ j. fs
! j · fs ! k)) i)
    for i unfolding b-def by simp
 } note IM-lower-values = this
 {
  fix i j
  assume i: i < m and j: j < m
  from i have im: i ≤ m by auto
  consider (1) j < i | (2) j = i | (3) i < j by linarith
  thus d i * IM $$ (i,j) ∈ ℤ
```

```
    proof cases
      case 1
    show ?thesis unfolding IM-lower-values[OF i 1] replace-col-def Gramian-matrix-alt-alt-def[OF
im]
        by (intro Ints-minus Ints-det, insert i j, auto intro!: Ints-scalar-prod[of - n]
fs-int)
    next
      case 3
      show ?thesis unfolding IM-lower-triag[OF 3 j] by simp
    next
      case 2
    show ?thesis unfolding IM-diag-1[OF i] 2 using i unfolding Gramian-determinant-def
         Gramian-matrix-alt-alt-def[OF im]
        by (intro Ints-mult Ints-det, insert i j, auto intro!: Ints-scalar-prod[of - n]
fs-int)
    qed
  }
qed

definition inv-mu-ij-mat :: nat ⇒ nat ⇒ int ⇒ int mat where
 inv-mu-ij-mat i j c = (let
    B = mat m m (λ ij. if ij = (i,j) then c else 0);
     C = mat m m (λ (i,j). the-inv (of-int :: - ⇒ 'a) (d i * the (mat-inverse (M
m)) $$ (i,j)))
   in B * C + 1_m m)

lemma inv-mu-ij-mat: assumes i: i < m and ji: j < i
  shows

   map-mat of-int (inv-mu-ij-mat i j c) * M m =
    mat m m (λij. if ij = (i, j) then of-int c * d j else 0) + M m

  A ∈ carrier-mat m n ⟹ c mod p = 0 ⟹ map-mat (λ x. x mod p) (inv-mu-ij-mat
i j c * A) =
    (map-mat (λ x. x mod p) A)

  inv-mu-ij-mat i j c ∈ carrier-mat m m
  i' < j' ⟹ j' < m ⟹ inv-mu-ij-mat i j c $$ (i',j') = 0
  k < m ⟹ inv-mu-ij-mat i j c $$ (k,k) = 1
proof −
  obtain IM where IM: IM = the (mat-inverse (M m)) by auto
  let ?oi = of-int :: - ⇒ 'a
  let ?C = mat m m (λ ij. if ij = (i,j) then ?oi c else 0)
  let ?D = mat m m (λ (i,j). d i * IM $$ (i,j))
  have oi: inj ?oi unfolding inj-on-def by auto
  have C: ?C ∈ carrier-mat m m by auto
  from i ji have j: j < m by auto
  from j have jm: {0..<m} = {0..<j} ∪ {j} ∪ {Suc j..<m} by auto
  note IM-props = d-inv-mu-integer[OF IM] inv-mu-inverse[OF IM]
```

```
  have mat-oi: map-mat ?oi (inv-mu-ij-mat i j c) = ?C * ?D + 1_m m (is ?MM
= -)
    unfolding inv-mu-ij-mat-def Let-def IM[symmetric]
    apply (subst of-int-hom.mat-hom-add, force, force)
    apply (rule arg-cong2[of - - - - (+)])
     apply (subst of-int-hom.mat-hom-mult, force, force)
     apply (rule arg-cong2[of - - - - (*)])
      apply force
     apply (rule eq-matI, (auto)[3], goal-cases)
  proof -
    case (1 i j)
    from IM-props(1)[OF 1]
    show ?case unfolding Ints-def using the-inv-f-f[OF oi] by auto
  qed auto
  have map-mat ?oi (inv-mu-ij-mat i j c) * M m = (?C * ?D) * M m + M m
unfolding mat-oi
    by (subst add-mult-distrib-mat[of - m m], auto)
  also have (?C * ?D) * M m = ?C * (?D * M m)
    by (rule assoc-mult-mat, auto)
  also have ?D = mat m m (λ (i,j). if i = j then d j else 0) * IM (is - = ?E * -)
  proof (rule eq-matI, insert IM-props(4), auto simp: scalar-prod-def, goal-cases)
    case (1 i j)
    hence id: {0..<m} = {0..<i} ∪ {i} ∪ {Suc i ..<m}
      by (auto simp add: list-trisect)
    show ?case unfolding id
      by (auto simp: sum.union-disjoint)
  qed
  also have ... * M m = ?E * (IM * M m)
    by (rule assoc-mult-mat[of - m m], insert IM-props, auto)
  also have IM * M m = 1_m m by fact
  also have ?E * 1_m m = ?E by simp
  also have ?C * ?E = mat m m (λ ij. if ij = (i,j) then ?oi c * d j else 0)
   by (rule eq-matI, auto simp: scalar-prod-def, auto simp: jm sum.union-disjoint)
  finally show map-mat ?oi (inv-mu-ij-mat i j c) * M m =
    mat m m (λ ij. if ij = (i,j) then ?oi c * d j else 0) + M m .
  show carr: inv-mu-ij-mat i j c ∈ carrier-mat m m
    unfolding inv-mu-ij-mat-def by auto
  {
    assume k: k < m
    have of-int (inv-mu-ij-mat i j c $$ (k,k)) = ?MM $$ (k,k)
      using carr k by auto
    also have ... = (?C * ?D) $$ (k,k) + 1 unfolding mat-oi using k by simp
    also have (?C * ?D) $$ (k,k) = 0 using k
      by (auto simp: scalar-prod-def, auto simp: jm sum.union-disjoint
        inv-mu-lower-triangular[OF IM ji i])
    finally show inv-mu-ij-mat i j c $$ (k,k) = 1 by simp
  }
  {
    assume ij': i' < j' j' < m
```

```
    have of-int (inv-mu-ij-mat i j c $$ (i',j')) = ?MM $$ (i',j')
      using carr ij' by auto
    also have ... = (?C * ?D) $$ (i',j') unfolding mat-oi using ij' by simp
    also have (?C * ?D) $$ (i',j') = (if i' = i then ?oi c * (d j * IM $$ (j, j')) else 0)
     using ij' i j by (auto simp: scalar-prod-def, auto simp: jm sum.union-disjoint)
    also have ... = 0 using inv-mu-lower-triangular[OF IM - ij'(2), of j] ij' i ji by auto
    finally show inv-mu-ij-mat i j c $$ (i',j') = 0 by simp
  }
  {
    assume A: A ∈ carrier-mat m n and c: c mod p = 0
    let ?mod = map-mat (λ x. x mod p)
    let ?C = mat m m (λ ij. if ij = (i,j) then c else 0)
    let ?D = mat m m (λ ij. if ij = (i,j) then 1 else (0 :: int))
   define B where B = mat m m (λ (i,j). the-inv ?oi (d i * the (mat-inverse (M m)) $$ (i,j)))
    have B: B ∈ carrier-mat m m unfolding B-def by auto
    define BA where BA = B * A
    have BA: BA ∈ carrier-mat m n unfolding BA-def using A B by auto
    define DBA where DBA = ?D * BA
    have DBA: DBA ∈ carrier-mat m n unfolding DBA-def using BA by auto
    have ?mod (inv-mu-ij-mat i j c * A) =
     ?mod ((?C * B + 1_m m) * A)
      unfolding inv-mu-ij-mat-def B-def by simp
    also have (?C * B + 1_m m) * A = ?C * B * A + A
      by (subst add-mult-distrib-mat, insert A B, auto)
    also have ?C * B * A = ?C * BA
      unfolding BA-def
      by (rule assoc-mult-mat, insert A B, auto)
    also have ?C = c ·_m ?D
      by (rule eq-matI, auto)
    also have ... * BA = c ·_m DBA using BA unfolding DBA-def by auto
    also have ?mod (... + A) = ?mod A
      by (rule eq-matI, insert DBA A c, auto simp: mult.assoc)
    finally show ?mod (inv-mu-ij-mat i j c * A) = ?mod A .
  }
qed
end

lemma Gramian-determinant-of-int: assumes fs: set fs ⊆ carrier-vec n
  and j: j ≤ length fs
shows of-int (gram-schmidt.Gramian-determinant n fs j)
  = gram-schmidt.Gramian-determinant n (map (map-vec rat-of-int) fs) j
proof −
  from j have j: k < j ⟹ k < length fs for k by auto
  show ?thesis
  unfolding gram-schmidt.Gramian-determinant-def
  by (subst of-int-hom.hom-det[symmetric], rule arg-cong[of - - det],
```

unfold gram-schmidt.Gramian-matrix-def Let-def, *subst of-int-hom.mat-hom-mult*, *force*, *force*,
unfold map-mat-transpose[*symmetric*],
rule arg-cong2[*of - - - - λ x y. x ∗ y*^{T}], *insert fs*[*unfolded set-conv-nth*]
j, (*fastforce intro*!: *eq-matI*)+)
qed

context *LLL*
begin

lemma *multiply-invertible-mat*: **assumes** *lin*: *lin-indep fs*
and *len*: *length fs* = *m*
and *A*: *A* ∈ *carrier-mat m m*
and *A-invertible*: ∃ *B*. *B* ∈ *carrier-mat m m* ∧ *B* ∗ *A* = *1*$_m$ *m*
and *fs′-prod*: *fs′* = *Matrix.rows* (*A* ∗ *mat-of-rows n fs*)
shows *lattice-of fs′* = *lattice-of fs*
lin-indep fs′
length fs′ = *m*
proof −
let *?Mfs* = *mat-of-rows n fs*
let *?Mfs′* = *mat-of-rows n fs′*
from *A-invertible* **obtain** *B* **where** *B*: *B* ∈ *carrier-mat m m* **and** *inv*: *B* ∗ *A* = *1*$_m$ *m* **by** *auto*
from *lin* **have** *fs*: *set fs* ⊆ *carrier-vec n* **unfolding** *gs.lin-indpt-list-def* **by** *auto*
with *len* **have** *Mfs*: *?Mfs* ∈ *carrier-mat m n* **by** *auto*
from *A Mfs* **have** *prod*: *A* ∗ *?Mfs* ∈ *carrier-mat m n* **by** *auto*
hence *fs′*: *length fs′* = *m set fs′* ⊆ *carrier-vec n* **unfolding** *fs′-prod*
by (*auto simp*: *Matrix.rows-def Matrix.row-def*)
have *Mfs-prod′*: *?Mfs′* = *A* ∗ *?Mfs*
unfolding *arg-cong*[*OF fs′-prod*, *of mat-of-rows n*]
by (*intro eq-matI*, *auto simp*: *mat-of-rows-def*)
have *B* ∗ *?Mfs′* = *B* ∗ (*A* ∗ *?Mfs*)
unfolding *Mfs-prod′* **by** *simp*
also have ... = (*B* ∗ *A*) ∗ *?Mfs*
by (*subst assoc-mult-mat*[*OF - A Mfs*], *insert B*, *auto*)
also have *B* ∗ *A* = *1*$_m$ *m* **by** *fact*
also have ... ∗ *?Mfs* = *?Mfs* **using** *Mfs* **by** *auto*
finally have *Mfs-prod*: *?Mfs* = *B* ∗ *?Mfs′* **..**
interpret *LLL*: *LLL-with-assms n m fs 2*
by (*unfold-locales*, *auto simp*: *len lin*)
from *LLL.LLL-change-basis*[*OF fs′*(*2*,*1*) *B A Mfs-prod Mfs-prod′*]
show *latt′*: *lattice-of fs′* = *lattice-of fs* **and** *lin′*: *gs.lin-indpt-list* (*RAT fs′*)
and *len′*: *length fs′* = *m*
by (*auto simp add*: *LLL-with-assms-def*)
qed

This is the key lemma.

lemma *change-single-element*: **assumes** *lin*: *lin-indep fs*

and *len*: *length fs* = *m*
and *i*: *i* < *m* **and** *ji*: *j* < *i*
and *A*: *A* = *gram-schmidt-fs-int.inv-mu-ij-mat n* (*RAT fs*) — the transformation matrix A
and *fs'-prod*: *fs'* = *Matrix.rows* (*A i j c* ∗ *mat-of-rows n fs*) — fs' is the new basis
and *latt*: *lattice-of fs* = *L*
shows *lattice-of fs'* = *L*
c mod p = *0* ⟹ *map* (*map-vec* (λ *x*. *x mod p*)) *fs'* = *map* (*map-vec* (λ *x*. *x mod p*)) *fs*
lin-indep fs'
length fs' = *m*
⋀ *k*. *k* < *m* ⟹ *gso fs' k* = *gso fs k*
⋀ *k*. *k* ≤ *m* ⟹ *d fs' k* = *d fs k*
i' < *m* ⟹ *j'* < *m* ⟹
μ fs' i' j' = (*if* (*i'*,*j'*) = (*i*,*j*) *then rat-of-int* (*c* ∗ *d fs j*) + *μ fs i' j' else μ fs i' j'*)
i' < *m* ⟹ *j'* < *m* ⟹
dμ fs' i' j' = (*if* (*i'*,*j'*) = (*i*,*j*) *then c* ∗ *d fs j* ∗ *d fs* (*Suc j*) + *dμ fs i' j' else dμ fs i' j'*)
proof −
let *?A* = *A i j c*
let *?Mfs* = *mat-of-rows n fs*
let *?Mfs'* = *mat-of-rows n fs'*
from *lin* **have** *fs*: *set fs* ⊆ *carrier-vec n* **unfolding** *gs.lin-indpt-list-def* **by** *auto*
with *len* **have** *Mfs*: *?Mfs* ∈ *carrier-mat m n* **by** *auto*
interpret *gsi*: *gram-schmidt-fs-int n RAT fs*
rewrites *gsi.inv-mu-ij-mat* = *A* **using** *lin* **unfolding** *A*
by (*unfold-locales*, *insert lin*[*unfolded gs.lin-indpt-list-def*], *auto simp*: *set-conv-nth*)
note *A* = *gsi.inv-mu-ij-mat*[*unfolded length-map len*, *OF i ji*, **where** *c* = *c*]
from *A*(*3*) *Mfs* **have** *prod*: *?A* ∗ *?Mfs* ∈ *carrier-mat m n* **by** *auto*
hence *fs'*: *length fs'* = *m set fs'* ⊆ *carrier-vec n* **unfolding** *fs'-prod*
by (*auto simp*: *Matrix.rows-def Matrix.row-def*)
have *Mfs-prod'*: *?Mfs'* = *?A* ∗ *?Mfs*
unfolding *arg-cong*[*OF fs'-prod*, *of mat-of-rows n*]
by (*intro eq-matI*, *auto simp*: *mat-of-rows-def*)
have *detA*: *det ?A* = *1*
by (*subst det-lower-triangular*[*OF A*(*4*) *A*(*3*)], *insert A*, *auto intro*!: *prod-list-neutral simp*: *diag-mat-def*)
have ∃ *B*. *B* ∈ *carrier-mat m m* ∧ *B* ∗ *?A* = 1_m *m*
by (*intro exI*[*of - adj-mat ?A*], *insert adj-mat*[*OF A*(*3*)], *auto simp*: *detA*)
from *multiply-invertible-mat*[*OF lin len A*(*3*) *this fs'-prod*] *latt*
show *latt'*: *lattice-of fs'* = *L* **and** *lin'*: *gs.lin-indpt-list* (*RAT fs'*)
and *len'*: *length fs'* = *m* **by** *auto*
interpret *LLL*: *LLL-with-assms n m fs 2*
by (*unfold-locales*, *auto simp*: *len lin*)
interpret *fs*: *fs-int-indpt n fs*
by (*standard*, *auto simp*: *lin*)
interpret *fs'*: *fs-int-indpt n fs'*

```
  by (standard, auto simp: lin')
 {
  assume c: c mod p = 0
  have id: rows (map-mat f A) = map (map-vec f) (rows A) for f A
   unfolding rows-def by auto
  have rows-id: set fs ⊆ carrier-vec n ⟹ rows (mat-of-rows n fs) = fs for fs
   unfolding mat-of-rows-def rows-def
   by (force simp: Matrix.row-def set-conv-nth intro!: nth-equalityI)
  from A(2)[OF Mfs c]
   have rows (map-mat (λx. x mod p) ?Mfs') = rows (map-mat (λx. x mod p)
?Mfs) unfolding Mfs-prod'
   by simp
  from this[unfolded id rows-id[OF fs] rows-id[OF fs'(2)]]
  show map (map-vec (λ x. x mod p)) fs' = map (map-vec (λ x. x mod p)) fs .
 }
 {
  define B where B = ?A
  have gs-eq: k < m ⟹ gso fs' k = gso fs k for k
  proof(induct rule: nat-less-induct)
   case (1 k)
   then show ?case
   proof(cases k = 0)
    case True
    then show ?thesis
    proof −
     have row ?Mfs' 0 = row ?Mfs 0
     proof −
      have 2: 0∈ {0..<m} and 3: {1..<m} = {0..<m} − {0}
       and 4: finite {0..<m} using 1 by auto
      have row ?Mfs' 0 = vec n (λj. row B 0 · col ?Mfs j)
       using row-mult A(3) Mfs 1 Mfs-prod' unfolding B-def by simp
      also have ... = vec n (λj. (∑ l∈{0..<m}. B $$ (0, l) * ?Mfs $$ (l, j)))
       using Mfs A(3) len 1 B-def unfolding scalar-prod-def by auto
      also have ... = vec n (λj. B $$ (0, 0) * ?Mfs $$ (0, j) +
       (∑ l∈{1..<m}. B $$ (0, l) * ?Mfs $$ (l, j)))
       using Groups-Big.comm-monoid-add-class.sum.remove[OF 4 2] 3
       by (simp add: ‹⋀g. sum g {0..<m} = g 0 + sum g ({0..<m} − {0})›)
      also have ... = row ?Mfs 0
       using A(4−) 1 unfolding B-def[symmetric] by (simp add: row-def)
      finally show ?thesis by (simp add: B-def Mfs-prod')
     qed
     then show ?thesis using True 1 fs'.f-carrier fs.f-carrier
      fs'.gs.fs0-gso0 len' len gsi.fs0-gso0 by auto
    qed
   next
    case False
    then show ?thesis
    proof −
     have gso0kcarr: gsi.gso ` {0 ..<k} ⊆ carrier-vec n
```

```
  using 1(2) gsi.gso-carrier len by auto
 hence gsospancarr: gs.span(gsi.gso ` {0 ..<k}) ⊆ carrier-vec n
  using span-is-subset2 by auto

 have fs'-gs-diff-span:
  (RAT fs') ! k - fs'.gs.gso k ∈ gs.span (gsi.gso ` {0 ..<k})
 proof -
  define gs'sum where gs'sum =
   gs.M.sumlist (map (λja. fs'.gs.μ k ja ·v fs'.gs.gso ja) [0..<k])
  define gssum where gssum =
   gs.M.sumlist (map (λja. fs'.gs.μ k ja ·v gsi.gso ja) [0..<k])
  have set (map (λja. fs'.gs.μ k ja ·v gsi.gso ja) [0..<k])
   ⊆ gs.span(gsi.gso ` {0 ..<k}) using 1(2) gs.span-mem gso0kcarr
   by auto
  hence gssumspan: gssum ∈ gs.span(gsi.gso ` {0 ..<k})
   using atLeastLessThan-iff gso0kcarr imageE set-map set-upt
    vec-space.sumlist-in-span
   unfolding gssum-def by (smt subsetD)
  hence gssumcarr: gssum ∈ carrier-vec n
   using gsospancarr gssum-def by blast
  have sumid: gs'sum = gssum
  proof -
   have map (λja. fs'.gs.μ k ja ·v fs'.gs.gso ja) [0..<k] =
    map (λja. fs'.gs.μ k ja ·v gsi.gso ja) [0..<k]
    using 1 by simp
   thus ?thesis unfolding gs'sum-def gssum-def by argo
  qed
  have (RAT fs') ! k = fs'.gs.gso k + gssum
   using fs'.gs.fs-by-gso-def len' False 1 sumid
   unfolding gs'sum-def by auto
  hence (RAT fs') ! k - fs'.gs.gso k = gssum
   using gssumcarr 1(2) len' by auto
  thus ?thesis using gssumspan by simp
 qed

define v2 where v2 = sumlist (map (λja. B $$ (k, ja) ·v fs ! ja) [0..< k])
 have v2carr: v2 ∈ carrier-vec n
 proof -
  have set (map (λja. B $$ (k, ja) ·v fs ! ja) [0..< k]) ⊆ carrier-vec n
   using len 1(2) fs.f-carrier by auto
  thus ?thesis unfolding v2-def by simp
 qed
 define ratv2 where ratv2 = (map-vec rat-of-int v2)
 have ratv2carr: ratv2 ∈ carrier-vec n
  unfolding ratv2-def using v2carr by simp
 have fs'id: (RAT fs') ! k = (RAT fs) ! k + ratv2
 proof -
  have zkm: [0..<m] = [0..<(Suc k)] @ [(Suc k)..<m] using 1(2)
   by (metis Suc-lessI append-Nil2 upt-append upt-rec zero-less-Suc)
```

```
    have prep: set (map (λja. B $$ (k, ja) ·v fs ! ja) [0..<m]) ⊆ carrier-vec
n
      using len fs.f-carrier by auto

    have fs' ! k = vec n (λj. row B k • col ?Mfs j)
      using 1(2) Mfs B-def A(3) fs'-prod by simp
    also have ... = sumlist (map (λja. B $$ (k, ja) ·v fs ! ja) [0..<m])
    proof −
      {
        fix i
        assume i: i < n
        have (vec n (λj. row B k • col ?Mfs j)) $ i = row B k • col ?Mfs i
          using i by auto
        also have ... = (∑j = 0..<m. B $$ (k, j) ∗ ?Mfs $$ (j,i))
          using A(3) unfolding B-def[symmetric]
        by (smt 1(2) Mfs R.finsum-cong' i atLeastLessThan-iff carrier-matD
            dim-col index-col index-row(1) scalar-prod-def)
        also have ... = (∑j = 0..<m. B $$ (k, j) ∗ (fs ! j $ i))
          by (metis (no-types, lifting) R.finsum-cong' atLeastLessThan-iff i
            len mat-of-rows-index)
        also have ... =
          (∑j = 0..<m. (map (λja.  B $$ (k, ja) ·v fs ! ja) [0..<m]) ! j $ i)
        proof −
          have ∀j<m. ∀i<n. B $$ (k, j) ∗ (fs ! j $ i) =
            (map (λja.  B $$ (k, ja) ·v fs ! ja) [0..<m]) ! j $ i
            using 1(2) i A(3) len fs.f-carrier
            unfolding B-def[symmetric] by auto
          then show ?thesis using i by auto
        qed
        also have ... = sumlist (map (λja. B $$ (k, ja) ·v fs ! ja) [0..<m])
$ i
          using sumlist-nth i fs.f-carrier carrier-vecD len by simp
        finally have (vec n (λj. row B k • col ?Mfs j)) $ i =
          sumlist (map (λja. B $$ (k, ja) ·v fs ! ja) [0..<m]) $ i by auto
      }
      then show ?thesis using fs.f-carrier len dim-sumlist by auto
    qed
    also have ... = sumlist (map (λja. B $$ (k, ja) ·v fs ! ja)
      ([0..<(Suc k)] @ [(Suc k)..<m]))
      using zkm by simp
    also have ... = sumlist (map (λja. B $$ (k, ja) ·v fs ! ja) [0..<(Suc k)])
+
      sumlist (map (λja. B $$ (k, ja) ·v fs ! ja) [(Suc k)..<m])
      (is ... = ?L2 + ?L3)
      using fs.f-carrier len dim-sumlist sumlist-append prep zkm by auto
    also have ?L3 = 0v n
      using A(4) fs.f-carrier len sumlist-nth carrier-vecD sumlist-carrier
        prep zkm unfolding B-def[symmetric] by auto
    also have ?L2 = sumlist (map (λja. B $$ (k, ja) ·v fs ! ja) [0..<k]) +
```

```
      B $$ (k, k) ·v fs ! k using prep zkm sumlist-snoc by simp
    also have ... = sumlist (map (λja. B $$ (k, ja) ·v fs ! ja) [0..<k]) + fs
! k
      using A(5) 1(2) unfolding B-def[symmetric] by simp
    finally have fs' ! k = fs ! k +
      sumlist (map (λja. B $$ (k, ja) ·v fs ! ja) [0..<k])
      using prep zkm by (simp add: M.add.m-comm)
    then have fs' ! k = fs ! k + v2 unfolding v2-def by simp
    then show ?thesis using v2carr 1(2) len len' ratv2-def by force
  qed
  have ratv2span: ratv2 ∈ gs.span (gsi.gso ` {0 ..<k})
  proof −
    have rat: ratv2 = gs.M.sumlist
      (map (λj. of-int (B $$ (k, j)) ·v (RAT fs) ! j) [0..<k])
    proof −
      have set (map (λj. of-int (B $$ (k, j)) ·v (RAT fs) ! j) [0..<k])
        ⊆ carrier-vec n
        using fs.f-carrier 1(2) len by auto
      hence carr: gs.M.sumlist
       (map (λj. of-int (B $$ (k, j)) ·v (RAT fs) ! j) [0..<k]) ∈ carrier-vec n
        by auto
      have set (map (λj. B $$ (k, j) ·v fs ! j) [0..<k]) ⊆ carrier-vec n
        using fs.f-carrier 1(2) len by auto
      hence ⋀i j. i < n ⟹ j < k ⟹ of-int ((B $$ (k, j) ·v fs ! j) $ i)
        = (of-int (B $$ (k, j)) ·v (RAT fs) ! j) $ i
        using 1(2) len by fastforce
      hence ⋀i. i < n ⟹ ratv2 $ i = gs.M.sumlist
        (map (λj. (of-int (B $$ (k, j)) ·v (RAT fs) ! j)) [0..<k]) $ i
        using fs.f-carrier 1(2) len v2carr gs.sumlist-nth sumlist-nth
          ratv2-def v2-def by simp
      then show ?thesis using ratv2carr carr by auto
    qed
    have ⋀i. i < k ⟹ (RAT fs) ! i =
      gs.M.sumlist (map (λ j. gsi.μ i j ·v gsi.gso j) [0 ..< Suc i])
      using gsi.fi-is-sum-of-mu-gso len 1(2) by auto
   moreover have ⋀i. i < k ⟹ (λ j. gsi.μ i j ·v gsi.gso j) ` {0 ..< Suc i}
      ⊆ gs.span (gsi.gso ` {0 ..<k})
      using gs.span-mem gso0kcarr by auto
    ultimately have ⋀i. i < k ⟹ (RAT fs) ! i ∈ gs.span (gsi.gso ` {0
..<k})
      using gso0kcarr set-map set-upt vec-space.sumlist-in-span subsetD by
smt
    then show ?thesis using rat atLeastLessThan-iff set-upt gso0kcarr
imageE
      set-map gs.smult-in-span vec-space.sumlist-in-span by smt
  qed
  have fs-gs-diff-span:
    (RAT fs) ! k − fs'.gs.gso k ∈ gs.span (gsi.gso ` {0 ..<k})
  proof −
```

```
          from fs'-gs-diff-span obtain v3 where sp: v3 ∈ gs.span (gsi.gso ` {0
..<k})
           and eq: (RAT fs) ! k − fs'.gs.gso k = v3 − ratv2
           using fs'.gs.gso-carrier len' 1(2) ratv2carr fs'id by fastforce
         have v3+(−ratv2) ∈ gs.span(gsi.gso ` {0 ..<k})
          by (metis sp gs.span-add1 gso0kcarr gram-schmidt.inv-in-span
              gso0kcarr ratv2span)
         moreover have v3+(−ratv2) = v3−ratv2 using ratv2carr by auto
         ultimately have v3 − ratv2 ∈ gs.span (gsi.gso ` {0 ..<k}) by simp
         then show ?thesis using eq by auto
        qed
        {
         fix i
         assume i: i < k
        hence fs'.gs.gso k • fs'.gs.gso i = 0 using 1(2) fs'.gs.orthogonal len' by
auto
         hence fs'.gs.gso k • gsi.gso i = 0 using 1 i by simp
        }
        hence ⋀x. x ∈ gsi.gso ` {0..<k} ⟹ fs'.gs.gso k • x = 0 by auto

        then show ?thesis
         using gsi.oc-projection-unique len len' fs-gs-diff-span 1(2) by auto
      qed
    qed
  qed

  have ⋀ i' j'. i' < m ⟹ j' < m ⟹ μ fs' i' j' =
    (map-mat of-int (A i j c) ∗ gsi.M m) $$ (i',j') and
    ⋀ k. k < m ⟹ gso fs' k = gso fs k
  proof −
    define rB where rB = map-mat rat-of-int B
   have rBcarr: rB ∈ carrier-mat m m using A(3) unfolding rB-def B-def by
simp
    define rfs where rfs = mat-of-rows n (RAT fs)
    have rfscarr: rfs ∈ carrier-mat m n using Mfs unfolding rfs-def by simp

    {
     fix i'
     fix j'
     assume i': i' < m
     assume j': j' < m
     have prep:
      of-int-hom.vec-hom (row (B ∗ mat-of-rows n fs) i') = row (rB ∗ rfs) i'
      using len i' B-def A(3) rB-def rfs-def by (auto simp: scalar-prod-def)
     have prep2: row (rB ∗ rfs) i' = vec n (λl. row rB i' • col rfs l)
      using len fs.f-carrier i' B-def A(3) scalar-prod-def rB-def
      unfolding rfs-def by auto
     have prep3: (vec m (λ j1. row rfs j1 • gsi.gso j' / ‖gsi.gso j'‖²)) =
      (vec m (λ j1. (gsi.M m) $$ (j1, j')))
```

```
proof −
  {
    fix x y
    assume x: x < m and y: y < m
   have (gsi.M m) $$ (x,y) = (if y < x then map of-int-hom.vec-hom fs ! x
     · fs'.gs.gso y / ‖fs'.gs.gso y‖² else if x = y then 1 else 0)
      using gsi.μ.simps x y j' len gs-eq gsi.M-index by auto
    hence row rfs x · gsi.gso y / ‖gsi.gso y‖² = (gsi.M m) $$ (x,y)
      unfolding rfs-def
     by (metis carrier-matD(1) divide-eq-eq fs'.gs.β-zero fs'.gs.gso-norm-beta

         gs-eq gsi.μ.simps gsi.fi-scalar-prod-gso gsi.fs-carrier len len'
         length-map nth-rows rfs-def rfscarr rows-mat-of-rows x y)
  }
  then show ?thesis using j' by auto
qed
have prep4: (1 / ‖gsi.gso j'‖²) ·ᵥ (vec m (λj1. row rfs j1 · gsi.gso j')) =
  (vec m (λj1. row rfs j1 · gsi.gso j' / ‖gsi.gso j'‖²)) by auto

have map of-int-hom.vec-hom fs' ! i' · fs'.gs.gso j' / ‖fs'.gs.gso j'‖²
   = map of-int-hom.vec-hom fs' ! i' · gsi.gso j' / ‖gsi.gso j'‖²
  using gs-eq j' by simp
also have ... = row (rB * rfs) i' · gsi.gso j' / ‖gsi.gso j'‖²
  using prep i' len' unfolding rB-def B-def by (simp add: fs'-prod)
also have ... =
  (vec n (λl. row rB i' · col rfs l)) · gsi.gso j' / ‖gsi.gso j'‖²
  using prep2 by auto
also have vec n (λl. row rB i' · col rfs l) =
    (vec n (λl. (∑j1=0..<m. (row rB i') $ j1 * (col rfs l) $ j1)))
  using gsi.gso-carrier
by (metis (no-types) carrier-matD(1) col-def dim-vec rfscarr scalar-prod-def)
also have ... =
    (vec n (λl. (∑j1=0..<m. rB $$ (i', j1) * rfs $$ (j1, l))))
  using rBcarr rfscarr i' by auto
also have ... · gsi.gso j' =
    (∑j2=0..<n. (vec n
    (λl. (∑j1=0..<m. rB $$ (i', j1) * rfs $$ (j1, l)))) $ j2 * (gsi.gso j') $
j2)
  using gsi.gso-carrier len j' scalar-prod-def
  by (smt gs.R.finsum-cong' gsi.gso-dim length-map)
also have ... = (∑j2=0..<n.
    (∑j1=0..<m. rB $$ (i', j1) * rfs $$ (j1, j2)) * (gsi.gso j') $ j2)
  using gsi.gso-carrier len j' by simp
also have ... = (∑j2=0..<n. (∑j1=0..<m.
    rB $$ (i', j1) * rfs $$ (j1, j2) * (gsi.gso j') $ j2))
  by (smt gs.R.finsum-cong' sum-distrib-right)
also have ... = (∑j1=0..<m. (∑j2=0..<n.
    rB $$ (i', j1) * rfs $$ (j1, j2) * (gsi.gso j') $ j2))
  using sum.swap by auto
```

```
      also have ... = (∑j1=0..<m. rB $$ (i', j1) * (∑j2=0..<n.
          rfs $$ (j1, j2) * (gsi.gso j') $ j2))
        using gs.R.finsum-cong' sum-distrib-left by (smt gs.m-assoc)
      also have ... = row rB i' • (vec m (λ j1. (∑j2=0..<n.
          rfs $$ (j1, j2) * (gsi.gso j') $ j2)))
        using rBcarr rfscarr i' scalar-prod-def
        by (smt atLeastLessThan-iff carrier-matD(1) carrier-matD(2) dim-vec
          gs.R.finsum-cong' index-row(1) index-vec)
      also have (vec m (λ j1. (∑j2=0..<n. rfs $$ (j1, j2) * (gsi.gso j') $ j2)))
        = (vec m (λ j1. row rfs j1 • gsi.gso j'))
      using rfscarr gsi.gso-carrier len j' rfscarr by (auto simp add: scalar-prod-def)
      also have row rB i' • ... / ‖gsi.gso j'‖^2 =
        row rB i' • vec m (λ j1. row rfs j1 • gsi.gso j' / ‖gsi.gso j'‖^2)
      using prep4 scalar-prod-smult-right rBcarr carrier-matD(2) dim-vec row-def

        by (smt gs.l-one times-divide-eq-left)
      also have ... = (rB * (gsi.M m)) $$ (i', j')
        using rBcarr i' j' prep3 gsi.M-def by (simp add: col-def)
      finally have
        map of-int-hom.vec-hom fs' ! i' • fs'.gs.gso j' / ‖fs'.gs.gso j'‖^2 =
        (rB * (gsi.M m)) $$ (i', j') by auto
    }
    then show ⋀ i' j'. i' < m ⟹ j' < m ⟹ μ fs' i' j' =
      (map-mat of-int (A i j c) * gsi.M m) $$ (i',j')
      using B-def fs'.gs.β-zero fs'.gs.fi-scalar-prod-gso fs'.gs.gso-norm-beta
        len' rB-def by auto
    show ⋀ k. k < m ⟹ gso fs' k = gso fs k using gs-eq by auto
  qed
 } note mu-gso = this

 show ⋀ k. k < m ⟹ gso fs' k = gso fs k by fact
 {
  fix k
  have k ≤ m ⟹ rat-of-int (d fs' k) = rat-of-int (d fs k) for k
  proof (induct k)
    case 0
    show ?case by (simp add: d-def)
  next
    case (Suc k)
    hence k: k ≤ m k < m by auto
    show ?case
        by (subst (1 2) LLL-d-Suc[OF - k(2)], auto simp: Suc(1)[OF k(1)]
mu-gso(2)[OF k(2)]
      LLL-invariant-weak-def lin lin' len len' latt latt')
  qed
  thus k ≤ m ⟹ d fs' k = d fs k by simp
 } note d = this
 {
  assume i': i' < m and j': j' < m
```

have μ *fs' i' j'* = (*of-int-hom.mat-hom* (*A i j c*) $*$ *gsi.M m*) $$ (*i'*,*j'*) **by** (*rule mu-gso*(*1*)[*OF i' j'*])
also have ... = (*if* (*i'*,*j'*) = (*i*,*j*) *then of-int c* $*$ *gsi.d j else 0*) + *gsi.M m* $$ (*i'*,*j'*)
unfolding *A*(*1*) **using** *i' j'* **by** (*auto simp*: *gsi.M-def*)
also have *gsi.M m* $$ (*i'*,*j'*) = μ *fs i' j'*
unfolding *gsi.M-def* **using** *i' j'* **by** *simp*
also have *gsi.d j* = *of-int* (*d fs j*)
unfolding *d-def* **by** (*subst Gramian-determinant-of-int*[*OF fs*], *insert ji i len*, *auto*)
finally show *mu*: μ *fs' i' j'* = (*if* (*i'*,*j'*) = (*i*,*j*) *then rat-of-int* (*c* $*$ *d fs j*) + μ *fs i' j' else* μ *fs i' j'*)
by *simp*
let *?d* = *d fs* (*Suc j'*)
have *d-fs*: *of-int* (*d*μ *fs i' j'*) = *rat-of-int ?d* $*$ μ *fs i' j'*
unfolding *d*μ-*def*
using *fs.fs-int-mu-d-Z-m-m*[*unfolded len*, *OF i' j'*]
by (*metis LLL.LLL.d-def assms*(*2*) *fs.fs-int-mu-d-Z-m-m fs-int.d-def i' int-of-rat*(*2*) *j'*)
have *rat-of-int* (*d*μ *fs' i' j'*) = *rat-of-int* (*d fs'* (*Suc j'*)) $*$ μ *fs' i' j'*
unfolding *d*μ-*def*
using *fs'.fs-int-mu-d-Z-m-m*[*unfolded len'*, *OF i' j'*]
using *LLL.LLL.d-def fs'*(*1*) *fs'.d*μ *fs'.d*μ-*def fs-int.d-def i' j'* **by** *auto*
also have *d fs'* (*Suc j'*) = *?d* **by** (*rule d*, *insert j'*, *auto*)
also have *rat-of-int* ... $*$ μ *fs' i' j'* =
(*if* (*i'*,*j'*) = (*i*,*j*) *then rat-of-int* (*c* $*$ *d fs j* $*$ *?d*) *else 0*) + *of-int* (*d*μ *fs i' j'*)
unfolding *mu d-fs* **by** (*simp add*: *field-simps*)
also have ... = *rat-of-int* ((*if* (*i'*,*j'*) = (*i*,*j*) *then c* $*$ *d fs j* $*$ *?d else 0*) + *d*μ *fs i' j'*)
by *simp*
also have ... = *rat-of-int* ((*if* (*i'*,*j'*) = (*i*,*j*) *then c* $*$ *d fs j* $*$ *d fs* (*Suc j*) + *d*μ *fs i' j' else d*μ *fs i' j'*))
by *simp*
finally show *d*μ *fs' i' j'* = (*if* (*i'*,*j'*) = (*i*,*j*) *then c* $*$ *d fs j* $*$ *d fs* (*Suc j*) + *d*μ *fs i' j' else d*μ *fs i' j'*)
by *simp*
}
qed

Eventually: Lemma 13 of Storjohann's paper.

lemma *mod-single-element*: **assumes** *lin*: *lin-indep fs*
and *len*: *length fs* = *m*
and *i*: *i* < *m* **and** *ji*: *j* < *i*
and *latt*: *lattice-of fs* = *L*
and *pgtz*: *p* > *0*
shows $\exists$ *fs'*. *lattice-of fs'* = *L* $\wedge$
map (*map-vec* (λ *x*. *x mod p*)) *fs'* = *map* (*map-vec* (λ *x*. *x mod p*)) *fs* $\wedge$
map (*map-vec* (λ *x*. *x symmod p*)) *fs'* = *map* (*map-vec* (λ *x*. *x symmod p*)) *fs* $\wedge$
lin-indep fs' $\wedge$

length fs′ = m ∧
(∀ *k < m. gso fs′ k = gso fs k*) ∧
(∀ *k ≤ m. d fs′ k = d fs k*) ∧
(∀ *i′ < m.* ∀ *j′ < m. dμ fs′ i′ j′ = (if (i′,j′) = (i,j) then dμ fs i j′ symmod (p * d fs j′ * d fs (Suc j′)) else dμ fs i′ j′)*)
proof −
have *inv*: *LLL-invariant-weak fs* **using** *LLL-invariant-weak-def assms* **by** *simp*
let *?mult = d fs j * d fs (Suc j)*
define *M* **where** *M = ?mult*
define *pM* **where** *pM = p * M*
then have *pMgtz*: *pM > 0* **using** *pgtz* **unfolding** *pM-def M-def* **using** *LLL-d-pos*[*OF inv*] *i ji* **by** *simp*
let *?d = dμ fs i j*
define *c* **where** *c = − (?d symdiv pM)*
have *d-mod*: *?d symmod pM = c * pM + ?d* **unfolding** *c-def* **using** *pMgtz sym-mod-sym-div* **by** *simp*
define *A* **where** *A = gram-schmidt-fs-int.inv-mu-ij-mat n (RAT fs)*
define *fs′* **where** *fs′*: *fs′ = Matrix.rows (A i j (c * p) * mat-of-rows n fs)*
note *main = change-single-element*[*OF lin len i ji A-def fs′ latt*]
have *map (map-vec (λx. x mod p)) fs′ = map (map-vec (λx. x mod p)) fs*
by (*intro main*, *auto*)
from *arg-cong*[*OF this*, *of map (map-vec (poly-mod.inv-M p))*]
have *id*: *map (map-vec (λx. x symmod p)) fs′ = map (map-vec (λx. x symmod p)) fs*
unfolding *map-map o-def sym-mod-def map-vec-map-vec* **.**
show *?thesis*
proof (*intro exI*[*of - fs′*] *conjI main allI impI id*)
fix *i′ j′*
assume *ij*: *i′ < m j′ < m*
have *dμ fs′ i′ j′ = (if (i′, j′) = (i, j) then (c * p) * M + ?d else dμ fs i′ j′)*
unfolding *main*(*8*)[*OF ij*] *M-def* **by** *simp*
also have *(c * p) * M + ?d = ?d symmod pM*
unfolding *d-mod* **by** (*simp add*: *pM-def*)
finally show *dμ fs′ i′ j′ = (if (i′,j′) = (i,j) then dμ fs i j′ symmod (p * d fs j′ * d fs (Suc j′)) else dμ fs i′ j′)*
by (*auto simp*: *pM-def M-def ac-simps*)
qed *auto*
qed

A slight generalization to perform modulo on arbitrary set of indices I.

lemma *mod-finite-set*: **assumes** *lin*: *lin-indep fs*
and *len*: *length fs = m*
and *I*: *I ⊆ {(i,j). i < m ∧ j < i}*
and *latt*: *lattice-of fs = L*
and *pgtz*: *p > 0*
shows ∃ *fs′. lattice-of fs′ = L* ∧
map (map-vec (λ x. x mod p)) fs′ = map (map-vec (λ x. x mod p)) fs ∧
map (map-vec (λ x. x symmod p)) fs′ = map (map-vec (λ x. x symmod p)) fs ∧
lin-indep fs′ ∧

```
  length fs' = m ∧
  (∀ k < m. gso fs' k = gso fs k) ∧
  (∀ k ≤ m. d fs' k = d fs k) ∧
  (∀ i' < m. ∀ j' < m. dμ fs' i' j' =
     (if (i',j') ∈ I then dμ fs i' j' symmod (p * d fs j' * d fs (Suc j')) else dμ fs i'
j'))
proof −
  let ?exp = λ fs' I i' j'.
    dμ fs' i' j' = (if (i',j') ∈ I then dμ fs i' j' symmod (p * d fs j' * d fs (Suc j'))
else dμ fs i' j')
  let ?prop = λ fs fs'. lattice-of fs' = L ∧
    map (map-vec (λ x. x mod p)) fs' = map (map-vec (λ x. x mod p)) fs ∧
    map (map-vec (λ x. x symmod p)) fs' = map (map-vec (λ x. x symmod p)) fs ∧
    lin-indep fs' ∧
    length fs' = m ∧
    (∀ k < m. gso fs' k = gso fs k) ∧
    (∀ k ≤ m. d fs' k = d fs k)
  have finite I
  proof (rule finite-subset[OF I], rule finite-subset)
    show {(i, j). i < m ∧ j < i} ⊆ {0..m} × {0..m} by auto
  qed auto
  from this I have ∃ fs'. ?prop fs fs' ∧ (∀ i' < m. ∀ j' < m. ?exp fs' I i' j')
  proof (induct I)
    case empty
    show ?case
      by (intro exI[of - fs], insert assms, auto)
  next
    case (insert ij I)
    obtain i j where ij: ij = (i,j) by force
    from ij insert(4) have i: i < m j < i by auto
    from insert(3,4) obtain gs where gs: ?prop fs gs
      and exp: ⋀ i' j'. i' < m ⟹ j' < m ⟹ ?exp gs I i' j' by auto
    from gs have lin-indep gs lattice-of gs = L length gs = m by auto
    from mod-single-element[OF this(1,3) i this(2), of p]
    obtain hs where hs: ?prop gs hs
      and exp': ⋀ i' j'. i' < m ⟹ j' < m ⟹
      dμ hs i' j' = (if (i', j') = (i, j)
        then dμ gs i j' symmod (p * d gs j' * d gs (Suc j')) else dμ gs i' j')
      using pgtz by auto
    from gs i have id: d gs j = d fs j d gs (Suc j) = d fs (Suc j) by auto
    show ?case
    proof (intro exI[of - hs], rule conjI; (intro allI impI)?)
      show ?prop fs hs using gs hs by auto
      fix i' j'
      assume *: i' < m j' < m
      show ?exp hs (insert ij I) i' j' unfolding exp'[OF *] ij using exp * i
        by (auto simp: id)
    qed
  qed
```

thus *?thesis* **by** *auto*
qed

end

end

4 Storjohann's basis reduction algorithm (abstract version)

This theory contains the soundness proofs of Storjohann's basis reduction algorithms, both for the normal and the improved-swap-order variant.

The implementation of Storjohann's version of LLL uses modular operations throughout. It is an abstract implementation that is already quite close to what the actual implementation will be. In particular, the swap operation here is derived from the computation lemma for the swap operation in the old, integer-only formalization of LLL.

theory *Storjohann*
imports
Storjohann-Mod-Operation
LLL-Basis-Reduction.LLL-Number-Bounds
Sqrt-Babylonian.NthRoot-Impl
begin

4.1 Definition of algorithm

In the definition of the algorithm, the first-flag determines, whether only the first vector of the reduced basis should be computed, i.e., a short vector. Then the modulus can be slightly decreased in comparison to the required modulus for computing the whole reduced matrix.

fun *max-list-rats-with-index* :: (*int* ∗ *int* ∗ *nat*) *list* ⇒ (*int* ∗ *int* ∗ *nat*) **where**
max-list-rats-with-index [*x*] = *x* |
max-list-rats-with-index ((*n1*,*d1*,*i1*) # (*n2*,*d2*,*i2*) # *xs*)
= *max-list-rats-with-index* ((*if n1* ∗ *d2* ≤ *n2* ∗ *d1 then* (*n2*,*d2*,*i2*) *else* (*n1*,*d1*,*i1*)) # *xs*)

context *LLL*
begin

definition *log-base* = (*10* :: *int*)

definition *bound-number* :: *bool* ⇒ *nat* **where**
bound-number first = (*if first* ∧ *m* ≠ *0 then 1 else m*)

definition *compute-mod-of-max-gso-norm* :: *bool* ⇒ *rat* ⇒ *int* **where**

```
  compute-mod-of-max-gso-norm first mn = log-base ^(log-ceiling log-base (max 2
(
     root-rat-ceiling 2 (mn * (rat-of-nat (bound-number first) + 3)) + 1)))

definition g-bnd-mode :: bool ⇒ rat ⇒ int vec list ⇒ bool where
  g-bnd-mode first b fs = (if first ∧ m ≠ 0 then sq-norm (gso fs 0) ≤ b else g-bnd
b fs)

definition d-of where d-of dmu i = (if i = 0 then 1 :: int else dmu $$ (i − 1, i
− 1))

definition compute-max-gso-norm :: bool ⇒ int mat ⇒ rat × nat where
  compute-max-gso-norm first dmu = (if m = 0 then (0,0) else
      case max-list-rats-with-index (map (λ i. (d-of dmu (Suc i), d-of dmu i, i)) [0
..< (if first then 1 else m)])
      of (num, denom, i) ⇒ (of-int num / of-int denom, i))


context
  fixes p :: int — the modulus
    and first :: bool — only compute first vector of reduced basis
begin

definition basis-reduction-mod-add-row ::
  int vec list ⇒ int mat ⇒ nat ⇒ nat ⇒ (int vec list × int mat)  where
  basis-reduction-mod-add-row mfs dmu i j =
    (let c = round-num-denom (dmu $$ (i,j)) (d-of dmu (Suc j)) in
      (if c = 0 then (mfs, dmu)
        else (mfs[ i := (map-vec (λ x. x symmod p)) (mfs ! i − c ·v mfs ! j)],
            mat m m (λ(i',j'). (if (i' = i ∧ j' ≤ j)
              then (if j'=j then (dmu $$ (i,j') − c * dmu $$ (j,j'))
                  else (dmu $$ (i,j') − c * dmu $$ (j,j'))
                      symmod (p * d-of dmu j' * d-of dmu (Suc j')))
              else (dmu $$ (i',j')))))))

fun basis-reduction-mod-add-rows-loop where
  basis-reduction-mod-add-rows-loop mfs dmu i 0 = (mfs, dmu)
| basis-reduction-mod-add-rows-loop mfs dmu i (Suc j) = (
     let (mfs', dmu') = basis-reduction-mod-add-row mfs dmu i j
      in basis-reduction-mod-add-rows-loop mfs' dmu' i j)

definition basis-reduction-mod-swap-dmu-mod :: int mat ⇒ nat ⇒ int mat where
  basis-reduction-mod-swap-dmu-mod dmu k = mat m m (λ(i, j). (
    if j < i ∧ (j = k ∨ j = k − 1) then
        dmu $$ (i, j) symmod (p * d-of dmu j * d-of dmu (Suc j))
    else dmu $$ (i, j)))

definition basis-reduction-mod-swap where
  basis-reduction-mod-swap mfs dmu k =
```

```
    (mfs[k := mfs ! (k − 1), k − 1 := mfs ! k],
     basis-reduction-mod-swap-dmu-mod (mat m m (λ(i,j). (
     if j < i then
       if i = k − 1 then
          dmu $$ (k, j)
       else if i = k ∧ j ≠ k − 1 then
            dmu $$ (k − 1, j)
       else if i > k ∧ j = k then
         ((d-of dmu (Suc k)) * dmu $$ (i, k − 1) − dmu $$ (k, k − 1) * dmu $$
(i, j))
              div (d-of dmu k)
       else if i > k ∧ j = k − 1 then
         (dmu $$ (k, k − 1) * dmu $$ (i, j) + dmu $$ (i, k) * (d-of dmu (k−1)))
            div (d-of dmu k)
       else dmu $$ (i, j)
     else if i = j then
       if i = k − 1 then
         ((d-of dmu (Suc k)) * (d-of dmu (k−1)) + dmu $$ (k, k − 1) * dmu $$
(k, k − 1))
           div (d-of dmu k)
       else (d-of dmu (Suc i))
     else dmu $$ (i, j))
   )) k)

fun basis-reduction-adjust-mod where
  basis-reduction-adjust-mod mfs dmu =
    (let (b,g-idx) = compute-max-gso-norm first dmu;
         p' = compute-mod-of-max-gso-norm first b
       in if p' < p then
         let mfs' = map (map-vec (λx. x symmod p')) mfs;
             d-vec = vec (Suc m) (λ i. d-of dmu i);
             dmu' = mat m m (λ (i,j). if j < i then dmu $$ (i,j)
               symmod (p' * d-vec $ j * d-vec $ (Suc j)) else
               dmu $$ (i,j))
           in (p', mfs', dmu', g-idx)
         else (p, mfs, dmu, g-idx))

definition basis-reduction-adjust-swap-add-step where
  basis-reduction-adjust-swap-add-step mfs dmu g-idx i = (
    let i1 = i − 1;
       (mfs1, dmu1) = basis-reduction-mod-add-row mfs dmu i i1;
       (mfs2, dmu2) = basis-reduction-mod-swap mfs1 dmu1 i
     in if i1 = g-idx then basis-reduction-adjust-mod mfs2 dmu2
       else (p, mfs2, dmu2, g-idx))


definition basis-reduction-mod-step where
  basis-reduction-mod-step mfs dmu g-idx i (j :: int) = (if i = 0 then (p, mfs, dmu,
g-idx, Suc i, j)
```

```
    else let di = d-of dmu i;
            (num, denom) = quotient-of α
     in if di * di * denom ≤ num * d-of dmu (i − 1) * d-of dmu (Suc i) then
        (p, mfs, dmu, g-idx, Suc i, j)
      else let (p', mfs', dmu', g-idx') = basis-reduction-adjust-swap-add-step mfs
dmu g-idx i
        in (p', mfs', dmu', g-idx', i − 1, j + 1))

primrec basis-reduction-mod-add-rows-outer-loop where
  basis-reduction-mod-add-rows-outer-loop mfs dmu 0 = (mfs, dmu) |
  basis-reduction-mod-add-rows-outer-loop mfs dmu (Suc i) =
   (let (mfs', dmu') = basis-reduction-mod-add-rows-outer-loop mfs dmu i in
    basis-reduction-mod-add-rows-loop mfs' dmu' (Suc i) (Suc i))
end
```

the main loop of the normal Storjohann algorithm

```
partial-function (tailrec) basis-reduction-mod-main where
  basis-reduction-mod-main p first mfs dmu g-idx i (j :: int) = (
   (if i < m
     then
       case basis-reduction-mod-step p first mfs dmu g-idx i j
       of (p', mfs', dmu', g-idx', i', j') ⇒
         basis-reduction-mod-main p' first mfs' dmu' g-idx' i' j'
     else
       (p, mfs, dmu)))

definition compute-max-gso-quot:: int mat ⇒ (int * int * nat) where
  compute-max-gso-quot dmu = max-list-rats-with-index
   (map (λi. ((d-of dmu (i+1)) * (d-of dmu (i+1)), (d-of dmu (i+2)) * (d-of dmu
i), Suc i)) [0..<(m−1)])
```

the main loop of Storjohann's algorithm with improved swap order

```
partial-function (tailrec) basis-reduction-iso-main where
  basis-reduction-iso-main p first mfs dmu g-idx (j :: int) = (
   (if m > 1 then
     (let (max-gso-num, max-gso-denum, indx) = compute-max-gso-quot dmu;
       (num, denum) = quotient-of α in
       (if (max-gso-num * denum  > num * max-gso-denum) then
           case basis-reduction-adjust-swap-add-step p first mfs dmu g-idx indx of
             (p', mfs', dmu', g-idx') ⇒
         basis-reduction-iso-main p' first mfs' dmu' g-idx' (j + 1)
        else
          (p, mfs, dmu)))
    else (p, mfs, dmu)))

definition compute-initial-mfs where
  compute-initial-mfs p = map (map-vec (λx. x symmod p)) fs-init

definition compute-initial-dmu where
```

```
  compute-initial-dmu p dmu = mat m m (λ(i',j'). if j' < i'
       then dmu $$ (i', j') symmod (p * d-of dmu j' * d-of dmu (Suc j'))
       else dmu $$ (i', j'))

definition dmu-initial = (let dmu = dμ-impl fs-init
   in mat m m (λ (i,j).
   if j ≤ i then dμ-impl fs-init !! i !! j else 0))

definition compute-initial-state first =
  (let dmu = dmu-initial;
       (b, g-idx) = compute-max-gso-norm first dmu;
       p = compute-mod-of-max-gso-norm first b
     in (p, compute-initial-mfs p, compute-initial-dmu p dmu, g-idx))
```

Storjohann's algorithm

```
definition reduce-basis-mod :: int vec list where
  reduce-basis-mod = (
     let first = False;
         (p0, mfs0, dmu0, g-idx) = compute-initial-state first;
         (p', mfs', dmu') = basis-reduction-mod-main p0 first mfs0 dmu0 g-idx 0 0;
           (mfs'', dmu'') = basis-reduction-mod-add-rows-outer-loop p' mfs' dmu'
(m-1)
      in mfs'')
```

Storjohann's algorithm with improved swap order

```
definition reduce-basis-iso :: int vec list where
  reduce-basis-iso = (
     let first = False;
         (p0, mfs0, dmu0, g-idx) = compute-initial-state first;
         (p', mfs', dmu') = basis-reduction-iso-main p0 first mfs0 dmu0 g-idx 0;
           (mfs'', dmu'') = basis-reduction-mod-add-rows-outer-loop p' mfs' dmu'
(m-1)
      in mfs'')
```

Storjohann's algorithm for computing a short vector

```
definition
  short-vector-mod = (
     let first = True;
         (p0, mfs0, dmu0, g-idx) = compute-initial-state first;
         (p', mfs', dmu') = basis-reduction-mod-main p0 first mfs0 dmu0 g-idx 0 0
      in hd mfs')
```

Storjohann's algorithm (iso-variant) for computing a short vector

```
definition
  short-vector-iso = (
     let first = True;
         (p0, mfs0, dmu0, g-idx) = compute-initial-state first;
         (p', mfs', dmu') = basis-reduction-iso-main p0 first mfs0 dmu0 g-idx 0
```

in hd mfs')
end

4.2 Towards soundness of Storjohann's algorithm

lemma *max-list-rats-with-index-in-set*:
assumes *max*: *max-list-rats-with-index xs* = (*nm*, *dm*, *im*)
and *len*: *length xs* ≥ *1*
shows (*nm*, *dm*, *im*) ∈ *set xs*
using *assms*
proof (*induct xs rule*: *max-list-rats-with-index.induct*)
case (*2 n1 d1 i1 n2 d2 i2 xs*)
have *1* ≤ *length* ((*if n1* ∗ *d2* ≤ *n2* ∗ *d1 then* (*n2*, *d2*, *i2*) *else* (*n1*, *d1*, *i1*)) # *xs*) **by** *simp*
moreover have *max-list-rats-with-index* ((*if n1* ∗ *d2* ≤ *n2* ∗ *d1 then* (*n2*, *d2*, *i2*) *else* (*n1*, *d1*, *i1*)) # *xs*)
= (*nm*, *dm*, *im*) **using** *2* **by** *simp*
moreover have (*if n1* ∗ *d2* ≤ *n2* ∗ *d1 then* (*n2*, *d2*, *i2*) *else* (*n1*, *d1*, *i1*)) ∈
set ((*n1*, *d1*, *i1*) # (*n2*, *d2*, *i2*) # *xs*) **by** *simp*
moreover then have *set* ((*if n1* ∗ *d2* ≤ *n2* ∗ *d1 then* (*n2*, *d2*, *i2*) *else* (*n1*, *d1*, *i1*)) # *xs*) ⊆
set ((*n1*, *d1*, *i1*) # (*n2*, *d2*, *i2*) # *xs*) **by** *auto*
ultimately show *?case* **using** *2*(*1*) **by** *auto*
qed *auto*

lemma *max-list-rats-with-index*: **assumes** ⋀ *n d i*. (*n*,*d*,*i*) ∈ *set xs* ⟹ *d* > *0*
and *max*: *max-list-rats-with-index xs* = (*nm*, *dm*, *im*)
and (*n*,*d*,*i*) ∈ *set xs*
shows *rat-of-int n* / *of-int d* ≤ *of-int nm* / *of-int dm*
using *assms*
proof (*induct xs arbitrary*: *n d i rule*: *max-list-rats-with-index.induct*)
case (*2 n1 d1 i1 n2 d2 i2 xs n d i*)
let *?r* = *rat-of-int*
from *2*(*2*) **have** *d1* > *0 d2* > *0* **by** *auto*
hence *d*: *?r d1* > *0 ?r d2* > *0* **by** *auto*
have (*n1* ∗ *d2* ≤ *n2* ∗ *d1*) = (*?r n1* ∗ *?r d2* ≤ *?r n2* ∗ *?r d1*)
unfolding *of-int-mult*[*symmetric*] **by** *presburger*
also have ... = (*?r n1* / *?r d1* ≤ *?r n2* / *?r d2*) **using** *d*
by (*smt divide-strict-right-mono leD le-less-linear mult.commute nonzero-mult-div-cancel-left*
not-less-iff-gr-or-eq times-divide-eq-right)
finally have *id*: (*n1* ∗ *d2* ≤ *n2* ∗ *d1*) = (*?r n1* / *?r d1* ≤ *?r n2* / *?r d2*) **.**
obtain *n' d' i'* **where** *new*: (*if n1* ∗ *d2* ≤ *n2* ∗ *d1 then* (*n2*, *d2*, *i2*) *else* (*n1*, *d1*, *i1*)) = (*n'*,*d'*,*i'*)
by *force*
have *nd'*: (*n'*,*d'*,*i'*) ∈ {(*n1*,*d1*,*i1*), (*n2*, *d2*, *i2*)} **using** *new*[*symmetric*] **by** *auto*
from *2*(*3*) **have** *res*: *max-list-rats-with-index* ((*n'*,*d'*,*i'*) # *xs*) = (*nm*, *dm*, *im*)
using *new* **by** *auto*
note *2* = *2*[*unfolded new*]

```
  show ?case
  proof (cases (n,d,i) ∈ set xs)
    case True
    show ?thesis
      by (rule 2(1)[of n d, OF 2(2) res], insert True nd', force+)
  next
    case False
    with 2(4) have n = n1 ∧ d = d1 ∨ n = n2 ∧ d = d2 by auto
    hence ?r n / ?r d ≤ ?r n' / ?r d' using new[unfolded id]
      by (metis linear prod.inject)
    also have ?r n' / ?r d' ≤ ?r nm / ?r dm
      by (rule 2(1)[of n' d', OF 2(2) res], insert nd', force+)
    finally show ?thesis .
  qed
qed auto

context LLL
begin

lemma log-base: log-base ≥ 2 unfolding log-base-def by auto

definition LLL-invariant-weak' :: nat ⇒ int vec list ⇒ bool where
  LLL-invariant-weak' i fs = (
    gs.lin-indpt-list (RAT fs) ∧
    lattice-of fs = L ∧
    weakly-reduced fs i ∧
    i ≤ m ∧
    length fs = m
  )

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

lemma *LLL-invI-weak*: **assumes**
set fs $\subseteq$ *carrier-vec n*
length fs = *m*
lattice-of fs = *L*
i $\leq$ *m*
lin-indep fs
weakly-reduced fs i
shows *LLL-invariant-weak′ i fs*
unfolding *LLL-invariant-weak′-def Let-def* **using** *assms* **by** *auto*

lemma *LLL-invw′-imp-w*: *LLL-invariant-weak′ i fs* $\Longrightarrow$ *LLL-invariant-weak fs*
unfolding *LLL-invariant-weak′-def LLL-invariant-weak-def* **by** *auto*

lemma *basis-reduction-add-row-weak*:
assumes *Linvw*: *LLL-invariant-weak′ i fs*
and *i*: *i* < *m* **and** *j*: *j* < *i*
and *fs′*: *fs′* = *fs*[*i* := *fs* ! *i* − *c* ·$_v$ *fs* ! *j*]
shows *LLL-invariant-weak′ i fs′*
g-bnd B fs $\Longrightarrow$ *g-bnd B fs′*
proof (*atomize*(*full*), *goal-cases*)
case *1*
note *Linv* = *LLL-invw′-imp-w*[*OF Linvw*]
note *main* = *basis-reduction-add-row-main*[*OF Linv i j fs′*]
have *bnd*: *g-bnd B fs* $\Longrightarrow$ *g-bnd B fs′* **using** *main*(*6*) **unfolding** *g-bnd-def* **by** *auto*
note *new* = *LLL-inv-wD*[*OF main*(*1*)]
note *old* = *LLL-invD-weak*[*OF Linvw*]
have *red*: *weakly-reduced fs′ i* **using** ‹*weakly-reduced fs i*› *main*(*6*) ‹*i* < *m*›
unfolding *gram-schmidt-fs.weakly-reduced-def* **by** *auto*
have *inv*: *LLL-invariant-weak′ i fs′* **using** *LLL-inv-wD*[*OF main*(*1*)] ‹*i* < *m*›
by (*intro LLL-invI-weak*, *auto intro*: *red*)
show *?case* **using** *inv red main bnd* **by** *auto*
qed

lemma *LLL-inv-weak-m-impl-i*:
assumes *inv*: *LLL-invariant-weak′ m fs*
and *i*: *i* $\leq$ *m*
shows *LLL-invariant-weak′ i fs*
proof −
have *weakly-reduced fs i* **using** *LLL-invD-weak*(*8*)[*OF inv*]
by (*meson assms*(*2*) *gram-schmidt-fs.weakly-reduced-def le-trans less-imp-le-nat linorder-not-less*)
then show *?thesis*
using *LLL-invI-weak*[*of fs i*, *OF LLL-invD-weak*(*3*,*6*,*7*)[*OF inv*] - *LLL-invD-weak*(*1*)[*OF inv*]]
LLL-invD-weak(*2*,*4*,*5*,*8*−)[*OF inv*] *i* **by** *simp*
qed

definition *mod-invariant* **where**
mod-invariant b p first = (*b* ≤ *rat-of-int* (*p* − *1*)^*2* / (*rat-of-nat* (*bound-number first*) + *3*)
∧ (∃ *e*. *p* = *log-base* ^ *e*))

lemma *compute-mod-of-max-gso-norm*: **assumes** *mn*: *mn* ≥ *0*
and *m*: *m* = *0* ⟹ *mn* = *0*
and *p*: *p* = *compute-mod-of-max-gso-norm first mn*
shows
p > *1*
mod-invariant mn p first
proof −
let *?m* = *bound-number first*
define *p'* **where** *p'* = *root-rat-ceiling 2* (*mn* ∗ (*rat-of-nat ?m* + *3*)) + *1*
define *p''* **where** *p''* = *max 2 p'*
define *q* **where** *q* = *real-of-rat* (*mn* ∗ (*rat-of-nat ?m* + *3*))
have ∗: −*1* < (*0* :: *real*) **by** *simp*
also have *0* ≤ *root 2* (*real-of-rat* (*mn* ∗ (*rat-of-nat ?m* + *3*))) **using** *mn* **by** *auto*
finally have *p'* ≥ *0* + *1* **unfolding** *p'-def*
by (*intro plus-left-mono*, *simp*)
hence *p'*: *p'* > *0* **by** *auto*
have *p''*: *p''* > *1* **unfolding** *p''-def* **by** *auto*
have *pp''*: *p* ≥ *p''* **unfolding** *compute-mod-of-max-gso-norm-def p p'-def*[*symmetric*] *p''-def*[*symmetric*]
using *log-base p'' log-ceiling-sound* **by** *auto*
hence *pp'*: *p* ≥ *p'* **unfolding** *p''-def* **by** *auto*
show *p* > *1* **using** *pp'' p''* **by** *auto*

have *q0*: *q* ≥ *0* **unfolding** *q-def* **using** *mn m* **by** *auto*
have (*mn* ≤ *rat-of-int* (*p'* − *1*)^*2* / (*rat-of-nat ?m* + *3*))
= (*real-of-rat mn* ≤ *real-of-rat* (*rat-of-int* (*p'* − *1*)^*2* / (*rat-of-nat ?m* + *3*)))
using *of-rat-less-eq* **by** *blast*
also have ... = (*real-of-rat mn* ≤ *real-of-rat* (*rat-of-int* (*p'* − *1*)^*2*) / *real-of-rat* (*rat-of-nat ?m* + *3*)) **by** (*simp add*: *of-rat-divide*)
also have ... = (*real-of-rat mn* ≤ ((*real-of-int* (*p'* − *1*))^*2*) / *real-of-rat* (*rat-of-nat ?m* + *3*))
by (*metis of-rat-of-int-eq of-rat-power*)
also have ... = (*real-of-rat mn* ≤ (*real-of-int* ⌈*sqrt q*⌉)^*2* / *real-of-rat* (*rat-of-nat ?m* + *3*))
unfolding *p'-def sqrt-def q-def* **by** *simp*
also have ...
proof −
have *real-of-rat mn* ≤ *q* / *real-of-rat* (*rat-of-nat ?m* + *3*) **unfolding** *q-def* **using** *m*
by (*auto simp*: *of-rat-mult*)
also have ... ≤ (*real-of-int* ⌈*sqrt q*⌉)^*2* / *real-of-rat* (*rat-of-nat ?m* + *3*)
proof (*rule divide-right-mono*)
have *q* = (*sqrt q*)^*2* **using** *q0* **by** *simp*

```
    also have ... ≤ (real-of-int ⌈sqrt q⌉)^2
      by (rule power-mono, auto simp: q0)
    finally show q ≤ (real-of-int ⌈sqrt q⌉)^2 .
  qed auto
  finally show ?thesis .
 qed
 finally have mn ≤ rat-of-int (p' − 1)^2 / (rat-of-nat ?m + 3) .
 also have ... ≤ rat-of-int (p − 1)^2 / (rat-of-nat ?m + 3)
  unfolding power2-eq-square
  by (intro divide-right-mono mult-mono, insert p' pp', auto)
 finally have mn ≤ rat-of-int (p − 1)^2 / (rat-of-nat ?m + 3) .
 moreover have ∃ e. p = log-base ^ e unfolding p compute-mod-of-max-gso-norm-def
by auto
  ultimately show mod-invariant mn p first unfolding mod-invariant-def by
auto
qed

lemma g-bnd-mode-cong: assumes ⋀ i. i < m ⟹ gso fs i = gso fs' i
 shows g-bnd-mode first b fs = g-bnd-mode first b fs'
 using assms unfolding g-bnd-mode-def g-bnd-def by auto

definition LLL-invariant-mod :: int vec list ⇒ int vec list ⇒ int mat ⇒ int ⇒
bool ⇒ rat ⇒ nat ⇒ bool where
 LLL-invariant-mod fs mfs dmu p first b i = (
   length fs = m ∧
   length mfs = m ∧
   i ≤ m ∧
   lattice-of fs = L ∧
   gs.lin-indpt-list (RAT fs) ∧
   weakly-reduced fs i ∧
   (map (map-vec (λx. x symmod p)) fs = mfs) ∧
   (∀ i' < m. ∀ j' < i'. |dμ fs i' j'| < p * d fs j' * d fs (Suc j')) ∧
   (∀ i' < m. ∀ j' < m. dμ fs i' j' = dmu $$ (i',j')) ∧
   p > 1 ∧
   g-bnd-mode first b fs ∧
   mod-invariant b p first
)

lemma LLL-invD-mod: assumes LLL-invariant-mod fs mfs dmu p first b i
shows
 length mfs = m
 i ≤ m
 length fs = m
 lattice-of fs = L
 gs.lin-indpt-list (RAT fs)
 weakly-reduced fs i
 (map (map-vec (λx. x symmod p)) fs = mfs)
 (∀ i' < m. ∀ j' < i'. |dμ fs i' j'| < p * d fs j' * d fs (Suc j'))
 (∀ i' < m. ∀ j' < m. dμ fs i' j' = dmu $$ (i',j'))
```

```
  ⋀ i. i < m ⟹ fs ! i ∈ carrier-vec n
  set fs ⊆ carrier-vec n
  ⋀ i. i < m ⟹ gso fs i ∈ carrier-vec n
  ⋀ i. i < m ⟹ mfs ! i ∈ carrier-vec n
  set mfs ⊆ carrier-vec n
  p > 1
  g-bnd-mode first b fs
  mod-invariant b p first
proof (atomize (full), goal-cases)
  case 1
  interpret gs': gram-schmidt-fs-lin-indpt n RAT fs
    using assms LLL-invariant-mod-def gs.lin-indpt-list-def
   by (meson gram-schmidt-fs-Rn.intro gram-schmidt-fs-lin-indpt.intro gram-schmidt-fs-lin-indpt-axioms.intro)
  have allfs: ∀ i < m. fs ! i ∈ carrier-vec n using assms gs'.f-carrier
    by (simp add: LLL.LLL-invariant-mod-def)
  then have setfs: set fs ⊆ carrier-vec n by (metis LLL-invariant-mod-def assms
in-set-conv-nth subsetI)
  have allgso: (∀ i < m. gso fs i ∈ carrier-vec n) using assms gs'.gso-carrier
    by (simp add: LLL.LLL-invariant-mod-def)
  show ?case
    using assms gs'.fs-carrier gs'.f-carrier gs'.gso-carrier allfs allgso
      LLL-invariant-mod-def gram-schmidt-fs.reduced-def in-set-conv-nth setfs by
fastforce
qed

lemma LLL-invI-mod: assumes
  length mfs = m
  i ≤ m
  length fs = m
  lattice-of fs = L
  gs.lin-indpt-list (RAT fs)
  weakly-reduced fs i
  map (map-vec (λx. x symmod p)) fs = mfs
  (∀ i' < m. ∀ j' < i'. |dμ fs i' j'| < p * d fs j' * d fs (Suc j'))
  (∀ i' < m. ∀ j' < m. dμ fs i' j' = dmu $$ (i',j'))
  p > 1
  g-bnd-mode first b fs
  mod-invariant b p first
shows LLL-invariant-mod fs mfs dmu p first b i
  unfolding LLL-invariant-mod-def using assms by blast

definition LLL-invariant-mod-weak :: int vec list ⇒ int vec list ⇒ int mat ⇒ int
⇒ bool ⇒ rat ⇒ bool where
  LLL-invariant-mod-weak fs mfs dmu p first b = (
    length fs = m ∧
    length mfs = m ∧
    lattice-of fs = L ∧
    gs.lin-indpt-list (RAT fs) ∧
    (map (map-vec (λx. x symmod p)) fs = mfs) ∧
```

$(\forall\, i' < m.\ \forall\ j' < i'.\ |d\mu\ fs\ i'\ j'| < p * d\ fs\ j' * d\ fs\ (Suc\ j')) \wedge$
$(\forall\, i' < m.\ \forall\, j' < m.\ d\mu\ fs\ i'\ j' = dmu\ \$\$\ (i',j')) \wedge$
$p > 1 \wedge$
g-bnd-mode first b fs $\wedge$
mod-invariant b p first
)

lemma *LLL-invD-modw*: **assumes** *LLL-invariant-mod-weak fs mfs dmu p first b*
shows
length mfs = m
length fs = m
lattice-of fs = L
gs.lin-indpt-list (RAT fs)
(map (map-vec (λx. x symmod p)) fs = mfs)
$(\forall\, i' < m.\ \forall\, j' < i'.\ |d\mu\ fs\ i'\ j'| < p * d\ fs\ j' * d\ fs\ (Suc\ j'))$
$(\forall\, i' < m.\ \forall\, j' < m.\ d\mu\ fs\ i'\ j' = dmu\ \$\$\ (i',j'))$
$\bigwedge\ i.\ i < m \Longrightarrow fs\ !\ i \in carrier\text{-}vec\ n$
set fs $\subseteq$ *carrier-vec n*
$\bigwedge\ i.\ i < m \Longrightarrow gso\ fs\ i \in carrier\text{-}vec\ n$
$\bigwedge\ i.\ i < m \Longrightarrow mfs\ !\ i \in carrier\text{-}vec\ n$
set mfs $\subseteq$ *carrier-vec n*
p > 1
g-bnd-mode first b fs
mod-invariant b p first
proof (*atomize* (*full*), *goal-cases*)
case *1*
interpret *gs'*: *gram-schmidt-fs-lin-indpt n RAT fs*
using *assms LLL-invariant-mod-weak-def gs.lin-indpt-list-def*
by (*meson gram-schmidt-fs-Rn.intro gram-schmidt-fs-lin-indpt.intro gram-schmidt-fs-lin-indpt-axioms.intro*)
have *allfs*: $\forall\, i < m.$ *fs ! i* $\in$ *carrier-vec n* **using** *assms gs'.f-carrier*
by (*simp add*: *LLL.LLL-invariant-mod-weak-def*)
then have *setfs*: *set fs* $\subseteq$ *carrier-vec n* **by** (*metis LLL-invariant-mod-weak-def assms in-set-conv-nth subsetI*)
have *allgso*: $(\forall\, i < m.$ *gso fs i* $\in$ *carrier-vec n*) **using** *assms gs'.gso-carrier*
by (*simp add*: *LLL.LLL-invariant-mod-weak-def*)
show *?case*
using *assms gs'.fs-carrier gs'.f-carrier gs'.gso-carrier allfs allgso*
LLL-invariant-mod-weak-def gram-schmidt-fs.reduced-def in-set-conv-nth setfs
by *fastforce*
qed

lemma *LLL-invI-modw*: **assumes**
length mfs = m
length fs = m
lattice-of fs = L
gs.lin-indpt-list (RAT fs)
map (map-vec (λx. x symmod p)) fs = mfs
$(\forall\, i' < m.\ \forall\, j' < i'.\ |d\mu\ fs\ i'\ j'| < p * d\ fs\ j' * d\ fs\ (Suc\ j'))$
$(\forall\, i' < m.\ \forall\, j' < m.\ d\mu\ fs\ i'\ j' = dmu\ \$\$\ (i',j'))$

```
  p > 1
  g-bnd-mode first b fs
  mod-invariant b p first
shows LLL-invariant-mod-weak fs mfs dmu p first b
  unfolding LLL-invariant-mod-weak-def using assms by blast

lemma ddμ:
  assumes i: i < m
  shows d fs (Suc i) = dμ fs i i
proof−
  have μ fs i i = 1 using i by (simp add: gram-schmidt-fs.μ.simps)
  then show ?thesis using dμ-def by simp
qed

lemma d-of-main: assumes (∀ i' < m. dμ fs i' i' = dmu $$ (i',i'))
  and i ≤ m
shows d-of dmu i = d fs i
proof (cases i = 0)
  case False
  with assms have i − 1 < m by auto
  from assms(1)[rule-format, OF this] ddμ[OF this, of fs] False
  show ?thesis by (simp add: d-of-def)
next
  case True
  thus ?thesis unfolding d-of-def True d-def by simp
qed

lemma d-of: assumes inv: LLL-invariant-mod fs mfs dmu p b first j
  and i ≤ m
shows d-of dmu i = d fs i
  by (rule d-of-main[OF - assms(2)], insert LLL-invD-mod(9)[OF inv], auto)

lemma d-of-weak: assumes inv: LLL-invariant-mod-weak fs mfs dmu p first b
  and i ≤ m
shows d-of dmu i = d fs i
  by (rule d-of-main[OF - assms(2)], insert LLL-invD-modw(7)[OF inv], auto)

lemma compute-max-gso-norm: assumes dmu: (∀ i' < m. dμ fs i' i' = dmu $$
(i',i'))
  and Linv: LLL-invariant-weak fs
shows g-bnd-mode first (fst (compute-max-gso-norm first dmu)) fs
  fst (compute-max-gso-norm first dmu) ≥ 0
  m = 0 ⟹ fst (compute-max-gso-norm first dmu) = 0
proof −
  show gbnd: g-bnd-mode first (fst (compute-max-gso-norm first dmu)) fs
  proof (cases first ∧ m ≠ 0)
    case False
    have ?thesis = (g-bnd (fst (compute-max-gso-norm first dmu)) fs) unfolding
g-bnd-mode-def using False by auto
```

```
    also have ... unfolding g-bnd-def
    proof (intro allI impI)
      fix i
      assume i: i < m
      have id: (if first then 1 else m) = m using False i by auto
      define list where list = map (λ i. (d-of dmu (Suc i), d-of dmu i, i)) [0 ..<
m ]
     obtain num denom j where ml: max-list-rats-with-index list = (num, denom,
j)
        by (metis prod-cases3)
      have dpos: d fs i > 0 using LLL-d-pos[OF Linv, of i]  i by auto
      have pos: (n, d, i) ∈ set list ⟹ 0 < d for n d i
        using LLL-d-pos[OF Linv] unfolding list-def using d-of-main[OF dmu]
by auto
      from i have list ! i ∈ set list using i unfolding list-def by auto
     also have list ! i = (d-of dmu (Suc i), d-of dmu i, i) unfolding list-def using
i by auto
      also have ... = (d fs (Suc i), d fs i, i) using d-of-main[OF dmu] i by auto
      finally have (d fs (Suc i), d fs i, i) ∈ set list .
      from max-list-rats-with-index[OF pos ml this]
      have of-int (d fs (Suc i)) / of-int (d fs i) ≤ fst (compute-max-gso-norm first
dmu)
        unfolding compute-max-gso-norm-def list-def[symmetric] ml id split using
i by auto
      also have of-int (d fs (Suc i)) / of-int (d fs i) = sq-norm (gso fs i)
        using LLL-d-Suc[OF Linv i] dpos by auto
      finally show sq-norm (gso fs i) ≤ fst (compute-max-gso-norm first dmu) .
    qed
    finally show ?thesis .
  next
    case True
   thus ?thesis unfolding g-bnd-mode-def compute-max-gso-norm-def using d-of-main[OF
dmu]
      LLL-d-Suc[OF Linv, of 0] LLL-d-pos[OF Linv, of 0] LLL-d-pos[OF Linv, of
1] by auto
  qed
  show fst (compute-max-gso-norm first dmu) ≥ 0
  proof (cases m = 0)
    case True
    thus ?thesis unfolding compute-max-gso-norm-def by simp
  next
    case False
    hence 0: 0 < m by simp
    have 0 ≤ sq-norm (gso fs 0) by blast
    also have ... ≤ fst (compute-max-gso-norm first dmu)
      using gbnd[unfolded g-bnd-mode-def g-bnd-def] using 0 by metis
    finally show ?thesis .
  qed
qed (auto simp: LLL.compute-max-gso-norm-def)
```

lemma *increase-i-mod*:
assumes *Linv*: *LLL-invariant-mod fs mfs dmu p first b i*
and *i*: $i < m$
and *red-i*: $i \neq 0 \Longrightarrow$ *sq-norm* (*gso fs* ($i - 1$)) $\leq \alpha *$ *sq-norm* (*gso fs i*)
shows *LLL-invariant-mod fs mfs dmu p first b* (*Suc i*) *LLL-measure i fs* > *LLL-measure* (*Suc i*) *fs*
proof −
note *inv* = *LLL-invD-mod*[*OF Linv*]
from *inv* **have** *red*: *weakly-reduced fs i* **by** (*auto*)
from *red red-i i* **have** *red*: *weakly-reduced fs* (*Suc i*)
unfolding *gram-schmidt-fs.weakly-reduced-def*
by (*intro allI impI*, *rename-tac ii*, *case-tac Suc ii* = *i*, *auto*)
show *LLL-invariant-mod fs mfs dmu p first b* (*Suc i*)
by (*intro LLL-invI-mod*, *insert inv red i*, *auto*)
show *LLL-measure i fs* > *LLL-measure* (*Suc i*) *fs* **unfolding** *LLL-measure-def*
using *i* **by** *auto*
qed

lemma *basis-reduction-mod-add-row-main*:
assumes *Linvmw*: *LLL-invariant-mod-weak fs mfs dmu p first b*
and *i*: $i < m$ **and** *j*: $j < i$
and *c*: *c* = *round* (μ *fs i j*)
and *mfs'*: *mfs'* = *mfs*[*i* := (*map-vec* (λ *x*. *x symmod p*)) (*mfs* ! *i* − *c* ·$_v$ *mfs* ! *j*)]
and *dmu'*: *dmu'* = *mat m m* (λ(*i'*,*j'*). (*if* ($i' = i \wedge j' \leq j$)
then (*if j'*=*j then* (*dmu* $$ (*i*,*j'*) − *c* * *dmu* $$ (*j*,*j'*))
else (*dmu* $$ (*i*,*j'*) − *c* * *dmu* $$ (*j*,*j'*))
symmod (*p* * (*d-of dmu j'*) * (*d-of dmu* (*Suc j'*))))
else (*dmu* $$ (*i'*,*j'*))))
shows ($\exists$ *fs'*. *LLL-invariant-mod-weak fs' mfs' dmu' p first b* $\wedge$
LLL-measure i fs' = *LLL-measure i fs*
$\wedge$ (μ-*small-row i fs* (*Suc j*) $\longrightarrow$ μ-*small-row i fs' j*)
$\wedge$ ($\forall$ *k* < *m*. *gso fs' k* = *gso fs k*)
$\wedge$ ($\forall$ *ii* $\leq$ *m*. *d fs' ii* = *d fs ii*)
$\wedge$ |μ *fs' i j*| $\leq$ *1 / 2*
$\wedge$ ($\forall$ *i' j'*. *i'* < *i* $\longrightarrow$ *j'* $\leq$ *i'* $\longrightarrow$ μ *fs' i' j'* = μ *fs i' j'*)
$\wedge$ (*LLL-invariant-mod fs mfs dmu p first b i* $\longrightarrow$ *LLL-invariant-mod fs' mfs' dmu' p first b i*))
proof −
define *fs'* **where** *fs'* = *fs*[*i* := *fs* ! *i* − *c* ·$_v$ *fs* ! *j*]
from *LLL-invD-modw*[*OF Linvmw*] **have** *gbnd*: *g-bnd-mode first b fs* **and** *p1*: *p* > *1* **and** *pgtz*: *p* > *0* **by** *auto*
have *Linvww*: *LLL-invariant-weak fs* **using** *LLL-invD-modw*[*OF Linvmw*] *LLL-invariant-weak-def* **by** *simp*
have
Linvw': *LLL-invariant-weak fs'* **and**
01: *c* = *round* (μ *fs i j*) $\Longrightarrow$ μ-*small-row i fs* (*Suc j*) $\Longrightarrow$ μ-*small-row i fs' j*
and

02: *LLL-measure i fs′ = LLL-measure i fs* **and**
03: $\bigwedge$ *i*. *i* < *m* ⟹ *gso fs′ i* = *gso fs i* **and**
04: $\bigwedge$ *i′ j′*. *i′* < *m* ⟹ *j′* < *m* ⟹
μ fs′ i′ j′ = (*if i′* = *i* ∧ *j′* ≤ *j then μ fs i j′* − *of-int c* ∗ *μ fs j j′ else μ fs i′ j′*) **and**
05: $\bigwedge$ *ii*. *ii* ≤ *m* ⟹ *d fs′ ii* = *d fs ii* **and**
06: |*μ fs′ i j*| ≤ *1* / *2* **and**
061: (∀ *i′ j′*. *i′* < *i* ⟶ *j′* ≤ *i′* ⟶ *μ fs i′ j′* = *μ fs′ i′ j′*)
using *basis-reduction-add-row-main*[*OF Linvww i j fs′-def*] *c i* **by** *auto*
have *07*: *lin-indep fs′* **and**
08: *length fs′* = *m* **and**
09: *lattice-of fs′* = *L* **using** *LLL-inv-wD Linvw′* **by** *auto*
have *091*: *fs-int-indpt n fs′* **using** *07* **using** *Gram-Schmidt-2.fs-int-indpt.intro* **by** *simp*
define *I* **where** *I* = {(*i′*,*j′*). *i′* = *i* ∧ *j′* < *j*}
have *10*: *I* ⊆ {(*i′*,*j′*). *i′* < *m* ∧ *j′* < *i′*} (*i*,*j*)∉ *I* ∀ *j′* ≥ *j*. (*i*,*j′*) ∉ *I* **using** *I-def i j* **by** *auto*
obtain *fs″* **where**
11: *lattice-of fs″* = *L* **and**
12: *map* (*map-vec* (λ *x*. *x symmod p*)) *fs″* = *map* (*map-vec* (λ *x*. *x symmod p*)) *fs′* **and**
13: *lin-indep fs″* **and**
14: *length fs″* = *m* **and**
15: (∀ *k* < *m*. *gso fs″ k* = *gso fs′ k*) **and**
16: (∀ *k* ≤ *m*. *d fs″ k* = *d fs′ k*) **and**
17: (∀ *i′* < *m*. ∀ *j′* < *m*. *dμ fs″ i′ j′* =
(*if* (*i′*,*j′*) ∈ *I then dμ fs′ i′ j′ symmod* (*p* ∗ *d fs′ j′* ∗ *d fs′* (*Suc j′*)) *else dμ fs′ i′ j′*))
using *mod-finite-set*[*OF 07 08 10*(*1*) *09 pgtz*] **by** *blast*
have *171*: (∀ *i′ j′*. *i′* < *i* ⟶ *j′* ≤ *i′* ⟶ *μ fs″ i′ j′* = *μ fs′ i′ j′*)
proof −
{
fix *i′ j′*
assume *i′j′*: *i′* < *i j′* ≤ *i′*
have *rat-of-int* (*dμ fs″ i′ j′*) = *rat-of-int* (*dμ fs′ i′ j′*) **using** *17 I-def i i′j′* **by** *auto*
then have *rat-of-int* (*int-of-rat* (*rat-of-int* (*d fs″* (*Suc j′*)) ∗ *μ fs″ i′ j′*)) =
rat-of-int (*int-of-rat* (*rat-of-int* (*d fs′* (*Suc j′*)) ∗ *μ fs′ i′ j′*))
using *dμ-def i′j′ j* **by** *auto*
then have *rat-of-int* (*d fs″* (*Suc j′*)) ∗ *μ fs″ i′ j′* =
rat-of-int (*d fs′* (*Suc j′*)) ∗ *μ fs′ i′ j′*
by (*smt 08 091 13 14 d-def dual-order.strict-trans fs-int.d-def*
fs-int-indpt.fs-int-mu-d-Z fs-int-indpt.intro i i′j′(*1*) *i′j′*(*2*) *int-of-rat*(*2*))
then have *μ fs″ i′ j′* = *μ fs′ i′ j′* **by** (*smt 16*
LLL-d-pos[*OF Linvw′*] *Suc-leI int-of-rat*(*1*)
dual-order.strict-trans fs′-def i i′j′ j
le-neq-implies-less nonzero-mult-div-cancel-left of-int-hom.hom-zero)
}
then show *?thesis* **by** *simp*

qed
then have *172*: $(\forall\, i'\, j'.\ i' < i \longrightarrow j' \leq i' \longrightarrow \mu\ fs''\ i'\ j' = \mu\ fs\ i'\ j')$ **using** *061* **by** *simp*
have *18*: *LLL-measure i fs'' = LLL-measure i fs'* **using** *16 LLL-measure-def logD-def D-def* **by** *simp*
have *19*: $(\forall\, k < m.\ gso\ fs''\ k = gso\ fs\ k)$ **using** *03 15* **by** *simp*
have $\forall\, j' \in \{j..(m-1)\}.\ j' < m$ **using** *j i* **by** *auto*
then have *20*: $\forall\, j' \in \{j..(m-1)\}.\ d\mu\ fs''\ i\ j' = d\mu\ fs'\ i\ j'$
using *10(3) 17 Suc-lessD less-trans-Suc* **by** (*meson atLeastAtMost-iff i*)
have *21*: $\forall\, j' \in \{j..(m-1)\}.\ \mu\ fs''\ i\ j' = \mu\ fs'\ i\ j'$
proof −
{
fix j'
assume j': $j' \in \{j..(m-1)\}$
define μ'' :: *rat* **where** $\mu'' = \mu\ fs''\ i\ j'$
define μ' :: *rat* **where** $\mu' = \mu\ fs'\ i\ j'$
have *rat-of-int* $(d\mu\ fs''\ i\ j') =$ *rat-of-int* $(d\mu\ fs'\ i\ j')$ **using** *20* j' **by** *simp*
moreover have $j' <$ *length fs'* **using** *i j' 08* **by** *auto*
ultimately have *rat-of-int* $(d\ fs'\ (Suc\ j'))$ $*$ *gram-schmidt-fs.μ n* (*map of-int-hom.vec-hom fs'*) *i j'*
$=$ *rat-of-int* $(d\ fs''\ (Suc\ j'))$ $*$ *gram-schmidt-fs.μ n* (*map of-int-hom.vec-hom fs''*) *i j'*
using *20 08 091 13 14 fs-int-indpt.dμ-def fs-int.d-def fs-int-indpt.dμ dμ-def d-def i fs-int-indpt.intro j'*
by *metis*
then have *rat-of-int* $(d\ fs'\ (Suc\ j'))$ $*$ $\mu'' =$ *rat-of-int* $(d\ fs'\ (Suc\ j'))$ $*$ μ'
using *16 i j' μ'-def μ''-def* **unfolding** *dμ-def* **by** *auto*
moreover have $0 <$ *d fs'* $(Suc\ j')$ **using** *LLL-d-pos*[*OF Linvw'*, *of Suc j'*] *i j'* **by** *auto*
ultimately have $\mu\ fs''\ i\ j' = \mu\ fs'\ i\ j'$ **using** *μ'-def μ''-def* **by** *simp*
}
then show *?thesis* **by** *simp*
qed
then have *22*: $\mu\ fs''\ i\ j = \mu\ fs'\ i\ j$ **using** *i j* **by** *simp*
then have *23*: $|\mu\ fs''\ i\ j| \leq 1\ /\ 2$ **using** *06* **by** *simp*
have *24*: *LLL-measure i fs'' = LLL-measure i fs* **using** *02 18* **by** *simp*
have *25*: $(\forall\ k \leq m.\ d\ fs''\ k = d\ fs\ k)$ **using** *16 05* **by** *simp*
have *26*: $(\forall\ k < m.\ gso\ fs''\ k = gso\ fs\ k)$ **using** *15 03* **by** *simp*
have *27*: *μ-small-row i fs* $(Suc\ j) \Longrightarrow$ *μ-small-row i fs'' j*
using *21 01 μ-small-row-def i j c* **by** *auto*
have *28*: *length fs = m length mfs = m* **using** *LLL-invD-modw*[*OF Linvmw*] **by** *auto*
have *29*: *map* (*map-vec* $(\lambda x.\ x\ symmod\ p)$) *fs = mfs* **using** *assms LLL-invD-modw* **by** *simp*
have *30*: $\bigwedge\ i.\ i < m \Longrightarrow fs\ !\ i \in$ *carrier-vec n* $\bigwedge\ i.\ i < m \Longrightarrow mfs\ !\ i \in$ *carrier-vec n*
using *LLL-invD-modw*[*OF Linvmw*] **by** *auto*
have *31*: $\bigwedge\ i.\ i < m \Longrightarrow$ *fs' ! i* $\in$ *carrier-vec n* **using** *fs'-def 30(1)*
using *08 091 fs-int-indpt.f-carrier* **by** *blast*

have *32*: $\bigwedge i.\ i < m \Longrightarrow mfs' ! i \in$ *carrier-vec n* **unfolding** *mfs'* **using** *30(2) 28(2)*
by (*metis* (*no-types*, *lifting*) *Suc-lessD j less-trans-Suc map-carrier-vec minus-carrier-vec*
nth-list-update-eq nth-list-update-neq smult-closed)
have *33*: *length mfs'* = *m* **using** *28(2) mfs'* **by** *simp*
then have *34*: *map* (*map-vec* (λ*x*. *x symmod p*)) *fs'* = *mfs'*
proof −
{
fix *i' j'*
have *j2*: *j* < *m* **using** *j i* **by** *auto*
assume *i'*: *i'* < *m*
assume *j'*: *j'* < *n*
then have *fsij*: (*fs* ! *i'* $ *j'*) *symmod p* = *mfs* ! *i'* $ *j'* **using** *30 i' j' 28 29* **by** *fastforce*
have *mfs'* ! *i* $ *j'* = (*mfs* ! *i* $ *j'*− (*c* ·$_v$ *mfs* ! *j*) $ *j'*) *symmod p*
unfolding *mfs'* **using** *30(2) j' 28 j2*
by (*metis* (*no-types*, *lifting*) *carrier-vecD i index-map-vec(1) index-minus-vec(1)*
index-minus-vec(2) index-smult-vec(2) nth-list-update-eq)
then have *mfs'ij*: *mfs'* ! *i* $ *j'* = (*mfs* ! *i* $ *j'*− *c* ∗ *mfs* ! *j* $ *j'*) *symmod p*
unfolding *mfs'* **using** *30(2) i' j' 28 j2* **by** *fastforce*
have (*fs'* ! *i'* $ *j'*) *symmod p* = *mfs'* ! *i'* $ *j'*
proof(*cases i'* = *i*)
case *True*
show *?thesis* **using** *fs'-def mfs' True 28 fsij*
proof −
have *fs'* ! *i'* $ *j'* = (*fs* ! *i'* − *c* ·$_v$ *fs* ! *j*) $ *j'* **using** *fs'-def True i' j' 28(1)* **by** *simp*
also have ... = *fs* ! *i'* $ *j'* − (*c* ·$_v$ *fs* ! *j*) $ *j'* **using** *i' j' 30(1)*
by (*metis Suc-lessD carrier-vecD i index-minus-vec(1) index-smult-vec(2) j less-trans-Suc*)
finally have *fs'* ! *i'* $ *j'* = *fs* ! *i'* $ *j'* − (*c* ·$_v$ *fs* ! *j*) $ *j'* **by** *auto*
then have (*fs'* ! *i'* $ *j'*) *symmod p* = (*fs* ! *i'* $ *j'* − (*c* ·$_v$ *fs* ! *j*) $ *j'*) *symmod p* **by** *auto*
also have ... = ((*fs* ! *i'* $ *j'*) *symmod p* − ((*c* ·$_v$ *fs* ! *j*) $ *j'*) *symmod p*) *symmod p*
by (*simp add*: *sym-mod-diff-eq*)
also have (*c* ·$_v$ *fs* ! *j*) $ *j'* = *c* ∗ (*fs* ! *j* $ *j'*)
using *i' j' True 28 30(1) j*
by (*metis Suc-lessD carrier-vecD index-smult-vec(1) less-trans-Suc*)
also have ((*fs* ! *i'* $ *j'*) *symmod p* − (*c* ∗ (*fs* ! *j* $ *j'*)) *symmod p*) *symmod p* =
((*fs* ! *i'* $ *j'*) *symmod p* − *c* ∗ ((*fs* ! *j* $ *j'*) *symmod p*)) *symmod p*
using *i' j' True 28 30(1) j* **by** (*metis sym-mod-diff-right-eq sym-mod-mult-right-eq*)
also have ((*fs* ! *j* $ *j'*) *symmod p*) = *mfs* ! *j* $ *j'* **using** *30 i' j' 28 29 j2* **by** *fastforce*
also have ((*fs* ! *i'* $ *j'*) *symmod p* − *c* ∗ *mfs* ! *j* $ *j'*) *symmod p* =
(*mfs* ! *i'* $ *j'* − *c* ∗ *mfs* ! *j* $ *j'*) *symmod p* **using** *fsij* **by** *simp*

```
      finally show ?thesis using mfs'ij by (simp add: True)
     qed
    next
     case False
     show ?thesis using fs'-def mfs' False 28 fsij by simp
    qed
   }
   then have ∀ i' < m. (map-vec (λx. x symmod p)) (fs' ! i') = mfs' ! i'
    using 31 32 33 08 by fastforce
   then show ?thesis using 31 32 33 08 by (simp add: map-nth-eq-conv)
  qed
  then have 35: map (map-vec (λx. x symmod p)) fs'' = mfs' using 12 by simp
  have 36: lin-indep fs'' using 13 by simp
  have Linvw'': LLL-invariant-weak fs'' using LLL-invariant-weak-def 11 13 14
by simp
  have 39: (∀ i' < m. ∀ j' < i'. |dμ fs'' i' j'| < p * d fs'' j' * d fs'' (Suc j'))
  proof −
   {
    fix i' j'
    assume i': i' < m
    assume j': j' < i'
    define pdd where pdd = (p * d fs'' j' * d fs'' (Suc j'))
    then have pddgtz: pdd > 0
     using pgtz j' LLL-d-pos[OF Linvw', of Suc j'] LLL-d-pos[OF Linvw', of j']
j' i' 16 by simp
    have |dμ fs'' i' j'| < p * d fs'' j' * d fs'' (Suc j')
    proof(cases i' = i)
     case i'i: True
     then show ?thesis
     proof (cases j' < j)
      case True
       then have eq'': dμ fs'' i' j' = dμ fs' i' j' symmod (p * d fs'' j' * d fs''
(Suc j'))
        using 16 17 10 I-def True i' j' i'i by simp
       have 0 < pdd using pddgtz by simp
       then show ?thesis unfolding eq'' unfolding pdd-def[symmetric] using
sym-mod-abs by blast
     next
      case fls: False
      then have (i',j') ∉ I using I-def i'i by simp
      then have dmufs''fs': dμ fs'' i' j' = dμ fs' i' j' using 17 i' j' by simp
      show ?thesis
      proof (cases j' = j)
       case True
       define μ'' where μ'' = μ fs'' i' j'
       define d'' where d'' = d fs'' (Suc j')
       have pge1: p ≥ 1 using pgtz by simp
       have lh: |μ''| ≤ 1 / 2 using 23 True i'i μ''-def by simp
        moreover have eq: dμ fs'' i' j' = μ'' * d'' using dμ-def i' j' μ''-def
```

d''-def
by (*smt 14 36 LLL.d-def Suc-lessD fs-int.d-def fs-int-indpt.dμ fs-int-indpt.intro*
int-of-rat(1) less-trans-Suc mult-of-int-commute of-rat-mult of-rat-of-int-eq)
moreover have *Sj'*: *Suc j' ≤ m j' ≤ m* **using** *True j' i i'* **by** *auto*
moreover then have *gtz*: *0 < d''* **using** *LLL-d-pos*[*OF Linvw''*] *d''-def* **by** *simp*
moreover have *rat-of-int* |*dμ fs'' i' j'*| = |*μ'' ∗* (*rat-of-int d''*)|
using *eq* **by** (*metis of-int-abs of-rat-hom.injectivity of-rat-mult of-rat-of-int-eq*)
moreover then have |*μ'' ∗ rat-of-int d''* | = |*μ''*| ∗ *rat-of-int* |*d''*|
by (*metis* (*mono-tags*, *opaque-lifting*) *abs-mult of-int-abs*)
moreover have ... = |*μ''*| ∗ *rat-of-int d''* **using** *gtz* **by** *simp*
moreover have ... < *rat-of-int d''* **using** *lh gtz* **by** *simp*
ultimately have *rat-of-int* |*dμ fs'' i' j'*| < *rat-of-int d''* **by** *simp*
then have |*dμ fs'' i' j'*| < *d fs''* (*Suc j'*) **using** *d''-def* **by** *simp*
then have |*dμ fs'' i' j'*| < *p ∗ d fs''* (*Suc j'*) **using** *pge1*
by (*smt mult-less-cancel-right2*)
then show *?thesis* **using** *pge1 LLL-d-pos*[*OF Linvw'' Sj'*(*2*)] *gtz* **unfolding** *d''-def*
by (*smt mult-less-cancel-left2 mult-right-less-imp-less*)
next
case *False*
have *j' < m* **using** *i' j'* **by** *simp*
moreover have *j' > j* **using** *False fls* **by** *simp*
ultimately have *μ fs' i' j' = μ fs i' j'* **using** *i' 04 i* **by** *simp*
then have *dμ fs' i' j' = dμ fs i' j'* **using** *dμ-def i' j' 05* **by** *simp*
then have *dμ fs'' i' j' = dμ fs i' j'* **using** *dmufs''fs'* **by** *simp*
then show *?thesis* **using** *LLL-invD-modw*[*OF Linvmw*] *i' j' 25* **by** *simp*
qed
qed
next
case *False*
then have (*i',j'*) ∉ *I* **using** *I-def* **by** *simp*
then have *dmufs''fs'*: *dμ fs'' i' j' = dμ fs' i' j'* **using** *17 i' j'* **by** *simp*
have *μ fs' i' j' = μ fs i' j'* **using** *i' 04 j' False* **by** *simp*
then have *dμ fs' i' j' = dμ fs i' j'* **using** *dμ-def i' j' 05* **by** *simp*
moreover then have *dμ fs'' i' j' = dμ fs i' j'* **using** *dmufs''fs'* **by** *simp*
then show *?thesis* **using** *LLL-invD-modw*[*OF Linvmw*] *i' j' 25* **by** *simp*
qed
}
then show *?thesis* **by** *simp*
qed
have *40*: (∀ *i' < m*. ∀ *j' < m*. *i' ≠ i* ∨ *j' > j* ⟶ *dμ fs' i' j' = dmu* $$ (*i',j'*))
proof −
{
fix *i' j'*
assume *i'*: *i' < m* **and** *j'*: *j' < m*

```
    assume assm: i' ≠ i ∨ j' > j
    have dμ fs' i' j' = dmu $$ (i',j')
    proof (cases i' ≠ i)
      case True
      then show ?thesis using fs'-def LLL-invD-modw[OF Linvmw] dμ-def i i' j
j'
        04 28(1) LLL-invI-weak basis-reduction-add-row-main(8)[OF Linvww] by
auto
    next
      case False
      then show ?thesis
        using 05 LLL-invD-modw[OF Linvmw] dμ-def i j j' 04 assm by simp
    qed
  }
  then show ?thesis by simp
 qed
 have 41: ∀ j' ≤ j. dμ fs' i j' = dmu $$ (i,j') − c ∗ dmu $$ (j,j')
 proof −
  {
    let ?oi = of-int :: - ⇒ rat
    fix j'
    assume j': j' ≤ j
    define dj' μi μj where dj' = d fs (Suc j') and μi = μ fs i j' and μj = μ fs
j j'
    have ?oi (dμ fs' i j') = ?oi (d fs (Suc j')) ∗ (μ fs i j' − ?oi c ∗ μ fs j j')
      using j' 04 dμ-def
      by (smt 05 08 091 Suc-leI d-def diff-diff-cancel fs-int.d-def
        fs-int-indpt.fs-int-mu-d-Z i int-of-rat(2) j less-imp-diff-less less-imp-le-nat)
    also have ... = (?oi dj') ∗ (μi − of-int c ∗ μj)
      using dj'-def μi-def μj-def by (simp add: of-rat-mult)
     also have ... = (rat-of-int dj') ∗ μi − of-int c ∗ (rat-of-int dj') ∗ μj by
algebra
   also have ... = rat-of-int (dμ fs i j') − ?oi c ∗ rat-of-int (dμ fs j j') unfolding
dj'-def μi-def μj-def
      using i j j' dμ-def
    using 28(1) LLL.LLL-invD-modw(4) Linvmw d-def fs-int.d-def fs-int-indpt.fs-int-mu-d-Z
fs-int-indpt.intro by auto
   also have ... = rat-of-int (dmu $$ (i,j')) − ?oi c ∗ rat-of-int (dmu $$ (j,j'))
      using LLL-invD-modw(7)[OF Linvmw] dμ-def j' i j by auto
    finally have ?oi (dμ fs' i j') = rat-of-int (dmu $$ (i,j')) − ?oi c ∗ rat-of-int
(dmu $$ (j,j')) by simp
    then have dμ fs' i j' = dmu $$ (i,j') − c ∗ dmu $$ (j,j')
      using of-int-eq-iff by fastforce
  }
  then show ?thesis by simp
 qed
 have 42: (∀ i' < m. ∀ j' < m. dμ fs'' i' j' = dmu' $$ (i',j'))
 proof −
  {
```

fix i' j'
assume i': $i' < m$ **and** j': $j' < m$
have $d\mu$ fs'' i' $j' = dmu'$ \$\$ (i',j')
proof (*cases* $i' = i$)
case $i'i$: *True*
then show *?thesis*
proof (*cases* $j' > j$)
case *True*
then have (i',j')$\notin I$ **using** *I-def* **by** *simp*
moreover then have $d\mu$ fs' i' $j' = d\mu$ fs i' j' **using** *04 05 True Suc-leI* $d\mu$*-def* i' j' **by** *simp*
moreover have dmu' \$\$ (i',j') = dmu \$\$ (i',j') **using** dmu' *True* i' j' **by** *simp*
ultimately show *?thesis* **using** *17 40 True* i' j' **by** *auto*
next
case *False*
then have $j'lej$: $j' \leq j$ **by** *simp*
then have eq': $d\mu$ fs' i $j' = dmu$ \$\$ (i,j') $-$ $c * dmu$ \$\$ (j,j') **using** *41* **by** *simp*
have *id*: *d-of dmu* $j' = d$ fs j' *d-of dmu* (*Suc* j') = d fs (*Suc* j')
using *d-of-weak*[*OF Linvmw*] ‹$j' < m$› **by** *auto*
show *?thesis*
proof (*cases* $j' \neq j$)
case *True*
then have $j'ltj$: $j' < j$ **using** *True False* **by** *simp*
then have (i',j') $\in I$ **using** *I-def True* $i'i$ **by** *simp*
then have $d\mu$ fs'' i' $j' =$
(*dmu* \$\$ (i,j') $-$ $c * dmu$ \$\$ (j,j')) *symmod* ($p * d$ fs' $j' * d$ fs' (*Suc* j'))
using *17* i' *41* $j'lej$ **by** (*simp add*: j' $i'i$)
also have ... = (*dmu* \$\$ (i,j') $-$ $c * dmu$ \$\$ (j,j')) *symmod* ($p * d$ *fs* $j' * d$ *fs* (*Suc* j'))
using *05* i $j'ltj$ j **by** *simp*
also have ... = dmu' \$\$ (i,j')
unfolding dmu' *index-mat*(*1*)[*OF* ‹$i < m$› ‹$j' < m$›] *split id* **using** $j'lej$ *True* **by** *auto*
finally show *?thesis* **using** $i'i$ **by** *simp*
next
case *False*
then have $j'j$: $j' = j$ **by** *simp*
then have $d\mu$ fs'' i $j' = d\mu$ fs' i j' **using** *20* j' **by** *simp*
also have ... = dmu \$\$ (i,j') $-$ $c * dmu$ \$\$ (j,j') **using** eq' **by** *simp*
also have ... = dmu' \$\$ (i,j') **using** dmu' $j'j$ i j' **by** *simp*
finally show *?thesis* **using** $i'i$ **by** *simp*
qed
qed
next
case *False*
then have (i',j')$\notin I$ **using** *I-def* **by** *simp*
moreover then have $d\mu$ fs' i' $j' = d\mu$ fs i' j' **by** (*simp add*: *04 05 False*

```
Suc-leI dμ-def i' j')
      moreover then have dmu' $$ (i',j') = dmu $$ (i',j') using dmu' False i'
j' by simp
      ultimately show ?thesis using 17 40 False i' j' by auto
    qed
   }
   then show ?thesis by simp
  qed
  from gbnd 26 have gbnd: g-bnd-mode first b fs'' using g-bnd-mode-cong[of fs''
fs] by simp
  {
   assume Linv: LLL-invariant-mod fs mfs dmu p first b i
   have Linvw: LLL-invariant-weak' i fs using Linv LLL-invD-mod LLL-invI-weak
by simp
   note Linvww = LLL-invw'-imp-w[OF Linvw]
   have 00: LLL-invariant-weak' i fs' using Linvw basis-reduction-add-row-weak[OF
Linvw i j fs'-def] by auto
   have 37: weakly-reduced fs'' i using 15 LLL-invD-weak(8)[OF 00] gram-schmidt-fs.weakly-reduced-def

     by (smt Suc-lessD i less-trans-Suc)
   have 38: LLL-invariant-weak' i fs''
     using 00 11 14 36 37 i 31 12  LLL-invariant-weak'-def by blast
   have LLL-invariant-mod fs'' mfs' dmu' p first b i
   using LLL-invI-mod[OF 33 - 14 11 13 37 35 39 42 p1 gbnd LLL-invD-mod(17)[OF
Linv]] i by simp
  }
  moreover have LLL-invariant-mod-weak fs'' mfs' dmu' p first b
   using LLL-invI-modw[OF 33 14 11 13 35 39 42 p1 gbnd LLL-invD-modw(15)[OF
Linvmw]] by simp
  ultimately show ?thesis using 27 23 24 25 26 172 by auto
qed

definition D-mod :: int mat ⇒ nat where D-mod dmu = nat (∏ i < m. d-of
dmu i)

definition logD-mod :: int mat ⇒ nat
  where logD-mod dmu = (if α = 4/3 then (D-mod dmu) else nat (floor (log (1
/ of-rat reduction) (D-mod dmu))))
end

locale fs-int'-mod =
  fixes n m fs-init α i fs mfs dmu p first b
  assumes LLL-inv-mod: LLL.LLL-invariant-mod n m fs-init α fs mfs dmu p first
b i

context LLL-with-assms
begin

lemma basis-reduction-swap-weak': assumes Linvw: LLL-invariant-weak' i fs
```

```
  and i: i < m
  and i0: i ≠ 0
  and mu-F1-i: |μ fs i (i−1)| ≤ 1 / 2
  and norm-ineq: sq-norm (gso fs (i − 1)) > α ∗ sq-norm (gso fs i)
  and fs'-def: fs' = fs[i := fs ! (i − 1), i − 1 := fs ! i]
shows LLL-invariant-weak' (i − 1) fs'
proof −
  note inv = LLL-invD-weak[OF Linvw]
  note invw = LLL-invw'-imp-w[OF Linvw]
 note main = basis-reduction-swap-main[OF invw disjI2[OF mu-F1-i] i i0 norm-ineq
fs'-def]
  note inv' = LLL-inv-wD[OF main(1)]
  from ‹weakly-reduced fs i› have weakly-reduced fs (i − 1)
    unfolding gram-schmidt-fs.weakly-reduced-def by auto
  also have weakly-reduced fs (i − 1) = weakly-reduced fs' (i − 1)
    unfolding gram-schmidt-fs.weakly-reduced-def
    by (intro all-cong, insert i0 i main(5), auto)
  finally have red: weakly-reduced fs' (i − 1) .
  show LLL-invariant-weak' (i − 1) fs' using i
    by (intro LLL-invI-weak red inv', auto)
qed

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

lemma LLL-invariant-mod-to-weak-m-to-i: assumes
  inv: LLL-invariant-mod fs mfs dmu p first b m
  and i: i ≤ m
shows LLL-invariant-mod fs mfs dmu p first b i
  LLL-invariant-weak' m fs
  LLL-invariant-weak' i fs
proof −
  show LLL-invariant-mod fs mfs dmu p first b i
  proof −
    have LLL-invariant-weak' m fs using LLL-invD-mod[OF inv] LLL-invI-weak
by simp
    then have LLL-invariant-weak' i fs using LLL-inv-weak-m-impl-i i by simp
    then have weakly-reduced fs i using i LLL-invD-weak(8) by simp
    then show ?thesis using LLL-invD-mod[OF inv] LLL-invI-mod i by simp
```

qed
then show *fsinvwi*: *LLL-invariant-weak' i fs* **using** *LLL-invD-mod LLL-invI-weak*
by *simp*
show *LLL-invariant-weak' m fs* **using** *LLL-invD-mod*[*OF inv*] *LLL-invI-weak*
by *simp*
qed

lemma *basis-reduction-mod-swap-main*:
assumes *Linvmw*: *LLL-invariant-mod-weak fs mfs dmu p first b*
and *k*: $k < m$
and *k0*: $k \neq 0$
and *mu-F1-i*: $|\mu\ fs\ k\ (k-1)| \leq 1\ /\ 2$
and *norm-ineq*: *sq-norm* (*gso fs* $(k - 1)$) $> \alpha *$ *sq-norm* (*gso fs k*)
and *mfs'-def*: $mfs' = mfs[k := mfs\ !\ (k - 1),\ k - 1 := mfs\ !\ k]$
and *dmu'-def*: $dmu' = (mat\ m\ m\ (\lambda(i,j).$ (
if $j < i$ *then*
if $i = k - 1$ *then*
dmu $$ (k, j)
else if $i = k \wedge j \neq k - 1$ *then*
dmu $$ $(k - 1, j)$
else if $i > k \wedge j = k$ *then*
((*d-of dmu* (*Suc k*)) $*$ *dmu* $$ $(i, k - 1)$ $-$ *dmu* $$ $(k, k - 1) *$ *dmu* $$ (i, j))
div (*d-of dmu k*)
else if $i > k \wedge j = k - 1$ *then*
(*dmu* $$ $(k, k - 1) *$ *dmu* $$ $(i, j) +$ *dmu* $$ $(i, k) *$ (*d-of dmu* $(k-1)$))
div (*d-of dmu k*)
else dmu $$ (i, j)
else if $i = j$ *then*
if $i = k - 1$ *then*
((*d-of dmu* (*Suc k*)) $*$ (*d-of dmu* $(k-1)$) $+$ *dmu* $$ $(k, k - 1) *$ *dmu* $$ $(k, k - 1)$)
div (*d-of dmu k*)
else (*d-of dmu* (*Suc i*))
else dmu $$ (i, j))
))
and *dmu'-mod-def*: $dmu'\text{-}mod = mat\ m\ m\ (\lambda(i, j).$ (
if $j < i \wedge (j = k \vee j = k - 1)$ *then*
dmu' $$ (i, j) *symmod* $(p *$ (*d-of dmu' j*) $*$ (*d-of dmu'* (*Suc j*)))
else dmu' $$ (i, j))))
shows $(\exists fs'.$ *LLL-invariant-mod-weak fs' mfs' dmu'-mod p first b* $\wedge$
LLL-measure $(k-1)$ *fs'* $<$ *LLL-measure k fs* $\wedge$
(*LLL-invariant-mod fs mfs dmu p first b k* $\longrightarrow$ *LLL-invariant-mod fs' mfs' dmu'-mod p first b* $(k-1)$))
proof $-$
define *fs'* **where** $fs' = fs[k := fs\ !\ (k - 1),\ k - 1 := fs\ !\ k]$
have *pgtz*: $p > 0$ **and** *p1*: $p > 1$ **using** *LLL-invD-modw*[*OF Linvmw*] **by** *auto*
have *invw*: *LLL-invariant-weak fs* **using** *LLL-invD-modw*[*OF Linvmw*] *LLL-invariant-weak-def*
by *simp*

note *swap-main* = *basis-reduction-swap-main*(3−)[*OF invw disjI2*[*OF mu-F1-i*] *k k0 norm-ineq fs'-def*]
note *ddμ-swap* = *d-dμ-swap*[*OF invw disjI2*[*OF mu-F1-i*] *k k0 norm-ineq fs'-def*]
have *invw'*: *LLL-invariant-weak fs'* **using** *fs'-def assms invw basis-reduction-swap-main*(1) **by** *simp*
have *02*: *LLL-measure k fs* > *LLL-measure* (*k* − 1) *fs'* **by** *fact*
have *03*: ⋀ *i j*. *i* < *m* ⟹ *j* < *i* ⟹
dμ fs' i j = (
if i = *k* − 1 *then*
dμ fs k j
else if i = *k* ∧ *j* ≠ *k* − 1 *then*
dμ fs (*k* − 1) *j*
else if i > *k* ∧ *j* = *k then*
(*d fs* (*Suc k*) ∗ *dμ fs i* (*k* − 1) − *dμ fs k* (*k* − 1) ∗ *dμ fs i j*) *div d fs k*
else if i > *k* ∧ *j* = *k* − 1 *then*
(*dμ fs k* (*k* − 1) ∗ *dμ fs i j* + *dμ fs i k* ∗ *d fs* (*k* − 1)) *div d fs k*
else dμ fs i j)
using *ddμ-swap* **by** *auto*
have *031*: ⋀*i*. *i* < *k*−1 ⟹ *gso fs' i* = *gso fs i*
using *swap-main*(2) *k k0* **by** *auto*
have *032*: ⋀ *ii*. *ii* ≤ *m* ⟹ *of-int* (*d fs' ii*) = (*if ii* = *k then*
sq-norm (*gso fs'* (*k* − 1)) / *sq-norm* (*gso fs* (*k* − 1)) ∗ *of-int* (*d fs k*)
else of-int (*d fs ii*))
by *fact*
have *gbnd*: *g-bnd-mode first b fs'*
proof (*cases first* ∧ *m* ≠ 0)
case *True*
have *sq-norm* (*gso fs'* 0) ≤ *sq-norm* (*gso fs* 0)
proof (*cases k* − 1 = 0)
case *False*
thus *?thesis* **using** *031*[*of 0*] **by** *simp*
next
case ∗: *True*
have *k-1*: *k* − 1 < *m* **using** *k* **by** *auto*
from ∗ *k0* **have** *k1*: *k* = 1 **by** *simp*

have *sq-norm* (*gso fs'* 0) ≤ *abs* (*sq-norm* (*gso fs'* 0)) **by** *simp*
also have ... = *abs* (*sq-norm* (*gso fs* 1) + *μ fs* 1 0 ∗ *μ fs* 1 0 ∗ *sq-norm* (*gso fs* 0))
by (*subst swap-main*(3)[*OF k-1*, *unfolded* ∗], *auto simp*: *k1*)
also have ... ≤ *sq-norm* (*gso fs* 1) + *abs* (*μ fs* 1 0) ∗ *abs* (*μ fs* 1 0) ∗ *sq-norm* (*gso fs* 0)
by (*simp add*: *sq-norm-vec-ge-0*)
also have ... ≤ *sq-norm* (*gso fs* 1) + (1 / 2) ∗ (1 / 2) ∗ *sq-norm* (*gso fs* 0)
using *mu-F1-i*[*unfolded k1*]
by (*intro plus-right-mono mult-mono*, *auto*)
also have ... < 1 / *α* ∗ *sq-norm* (*gso fs* 0) + (1 / 2) ∗ (1 / 2) ∗ *sq-norm* (*gso fs* 0)
by (*intro add-strict-right-mono*, *insert norm-ineq*[*unfolded mult.commute*[*of*

```
α],
        THEN mult-imp-less-div-pos[OF α0(1)]] k1, auto)
    also have ... = reduction * sq-norm (gso fs 0) unfolding reduction-def
      using α0 by (simp add: ring-distribs add-divide-distrib)
    also have ... ≤ 1 * sq-norm (gso fs 0) using reduction(2)
      by (intro mult-right-mono, auto)
    finally show ?thesis by simp
  qed
  thus ?thesis using LLL-invD-modw(14)[OF Linvmw] True
    unfolding g-bnd-mode-def by auto
 next
  case False
    from LLL-invD-modw(14)[OF Linvmw] False have g-bnd b fs unfolding
g-bnd-mode-def by auto
  hence g-bnd b fs' using g-bnd-swap[OF k k0 invw mu-F1-i norm-ineq fs'-def]
by simp
  thus ?thesis using False unfolding g-bnd-mode-def by auto
 qed
 note d-of = d-of-weak[OF Linvmw]
 have 033: ⋀ i. i < m ⟹ dμ fs' i i = (
          if i = k − 1 then
          ((d-of dmu (Suc k)) * (d-of dmu (k−1)) + dmu $$ (k, k − 1) * dmu $$
(k, k − 1))
              div (d-of dmu k)
          else (d-of dmu (Suc i)))
 proof −
  fix i
  assume i: i < m
  have dμ fs' i i = d fs' (Suc i) using ddμ i by simp
  also have ... = (if i = k − 1 then
       (d fs (Suc k) * d fs (k − 1) + dμ fs k (k − 1) * dμ fs k (k − 1)) div d fs
k
      else d fs (Suc i))
    by (subst ddμ-swap, insert ddμ k0 i, auto)
  also have ... = (if i = k − 1 then
      ((d-of dmu (Suc k)) * (d-of dmu (k−1)) + dmu $$ (k, k − 1) * dmu $$ (k,
k − 1))
        div (d-of dmu k)
      else (d-of dmu (Suc i))) (is - = ?r)
    using d-of i k LLL-invD-modw(7)[OF Linvmw] by auto
  finally show dμ fs' i i = ?r .
 qed
 have 04: lin-indep fs' length fs' = m lattice-of fs' = L using LLL-inv-wD[OF
invw'] by auto
 define I where I = {(i, j). i < m ∧ j < i ∧ (j = k ∨ j = k − 1)}
 then have Isubs: I ⊆ {(i,j). i < m ∧ j < i} using k k0 by auto
 obtain fs'' where
  05: lattice-of fs'' = L and
  06: map (map-vec (λ x. x symmod p)) fs'' = map (map-vec (λ x. x symmod p))
```

```
fs' and
    07: lin-indep fs'' and
    08: length fs'' = m and
    09: (∀ k < m. gso fs'' k = gso fs' k) and
    10: (∀ k ≤ m. d fs'' k = d fs' k) and
    11: (∀ i' < m. ∀ j' < m. dμ fs'' i' j' =
          (if (i',j') ∈ I then dμ fs' i' j' symmod (p * d fs' j' * d fs' (Suc j')) else
dμ fs' i' j'))
    using mod-finite-set[OF 04(1) 04(2) Isubs 04(3) pgtz] by blast
  have 13: length mfs' = m using mfs'-def LLL-invD-modw(1)[OF Linvmw] by
simp
  have 14: map (map-vec (λ x. x symmod p)) fs'' = mfs'
    using 06 fs'-def k k0 04(2) LLL-invD-modw(5)[OF Linvmw]
      by (metis (no-types, lifting) length-list-update less-imp-diff-less map-update
mfs'-def nth-map)
  have LLL-measure (k − 1) fs'' = LLL-measure (k − 1) fs' using 10 LLL-measure-def
logD-def D-def by simp
  then have 15: LLL-measure (k − 1) fs'' < LLL-measure k fs using 02 by simp

  {
    fix i' j'
    assume i'j': i'<m j'<i'
      and neq: j' ≠ k j' ≠ k − 1
    hence j'k: j' ≠ k Suc j' ≠ k using k0 by auto
    hence d fs'' j' = d fs j' d fs'' (Suc j') = d fs (Suc j')
      using ‹k < m› i'j' k0
        10[rule-format, of j'] 032[rule-format, of j']
        10[rule-format, of Suc j'] 032[rule-format, of Suc j']
      by auto
  } note d-id = this

  have 16: ∀ i'<m. ∀ j'<i'. |dμ fs'' i' j'| < p * d fs'' j' * d fs'' (Suc j')
  proof −
    {
      fix i' j'
      assume i'j': i'<m j'<i'
      have |dμ fs'' i' j'| < p * d fs'' j' * d fs'' (Suc j')
      proof (cases (i',j') ∈ I)
        case True
        define pdd where pdd = (p * d fs' j' * d fs' (Suc j'))
       have pdd-pos: pdd > 0 using pgtz i'j' LLL-d-pos[OF invw'] pdd-def by simp
        have dμ fs'' i' j' = dμ fs' i' j' symmod pdd using True 11 i'j' pdd-def by
simp
       then have |dμ fs'' i' j'| < pdd using True 11 i'j' pdd-pos sym-mod-abs by
simp
        then show ?thesis unfolding pdd-def using 10 i'j' by simp
      next
        case False
        from False[unfolded I-def] i'j' have neg: j' ≠ k j' ≠ k − 1 by auto
```

```
      consider (1) i' = k − 1 ∨ i' = k | (2) ¬ (i' = k − 1 ∨ i' = k)
        using False i'j' unfolding I-def by linarith
      thus ?thesis
      proof cases
        case **: 1
        let ?i'' = if i' = k − 1 then k else k −1
        from ** neg i'j' have i'': ?i'' < m j' < ?i'' using k0 k by auto
        have dμ fs'' i' j' = dμ fs' i' j' using 11 False i'j' by simp
        also have ... = dμ fs ?i'' j' unfolding 03[OF ‹i' < m› ‹j' < i'›]
          using ** neg by auto
        finally show ?thesis using LLL-invD-modw(6)[OF Linvmw, rule-format,
OF i''] unfolding d-id[OF i'j' neg] by auto
      next
        case **: 2
        hence neq: j' ≠ k j' ≠ k − 1 using False k k0 i'j' unfolding I-def by
auto
        have dμ fs'' i' j' = dμ fs' i' j' using 11 False i'j' by simp
        also have ... = dμ fs i' j' unfolding 03[OF ‹i' < m› ‹j' < i'›] using **
neq by auto
        finally show ?thesis using LLL-invD-modw(6)[OF Linvmw, rule-format,
OF i'j'] using d-id[OF i'j' neq] by auto
      qed
    qed
  }
  then show ?thesis by simp
 qed
 have 17: ∀ i'<m. ∀ j'<m. dμ fs'' i' j' = dmu'-mod $$ (i', j')
 proof −
  {
    fix i' j'
    assume i'j': i'<m j'<i'
    have d'dmu': ∀ j' < m. d fs' (Suc j') = dmu' $$ (j', j') using ddμ dmu'-def
033 by simp
    have eq': dμ fs' i' j' = dmu' $$ (i', j')
    proof −
      have t00: dμ fs k j' = dmu $$ (k, j') and
        t01: dμ fs (k − 1) j' =  dmu $$ (k − 1, j') and
        t04: dμ fs k (k − 1) = dmu $$ (k, k − 1) and
        t05: dμ fs i' k = dmu $$ (i', k)
        using LLL-invD-modw(7)[OF Linvmw] i'j' k ddμ k0 by auto
      have t03: d fs k = dμ fs (k−1) (k−1) using k0 k by (metis LLL.ddμ
Suc-diff-1 lessI not-gr-zero)
      have t06: d fs (k − 1) = (d-of dmu (k−1)) using d-of k by auto
      have t07: d fs k = (d-of dmu k) using d-of k by auto
      have j': j' < m using i'j' by simp
      have dμ fs' i' j' = (if i' = k − 1 then
                dmu $$ (k, j')
              else if i' = k ∧ j' ≠ k − 1 then
```

```
            dmu $$ (k − 1, j′)
          else if i′ > k ∧ j′ = k then
            (dmu $$ (k, k) ∗ dmu $$ (i′, k − 1) − dmu $$ (k, k − 1) ∗ dmu
$$ (i′, j′)) div (d-of dmu k)
          else if i′ > k ∧ j′ = k − 1 then
            (dmu $$ (k, k − 1) ∗ dmu $$ (i′, j′) + dmu $$ (i′, k) ∗ d fs (k −
1)) div (d-of dmu k)
          else dmu $$ (i′, j′))
      using ddμ k t00 t01 t03 LLL-invD-modw(7)[OF Linvmw] k i′j′ j′ 03 t07
by simp
     then show ?thesis using dmu′-def i′j′ j′ t06 t07 by (simp add: d-of-def)
   qed
   have dμ fs′′ i′ j′ = dmu′-mod $$ (i′, j′)
   proof (cases (i′,j′) ∈ I)
    case i′j′I: True
    have j′: j′ < m using i′j′ by simp
    show ?thesis
    proof −
     have dmu′-mod $$ (i′,j′) = dmu′ $$ (i′,j′)
           symmod (p ∗ (d-of dmu′ j′) ∗ (d-of dmu′ (Suc j′)))
       using dmu′-mod-def i′j′ i′j′I I-def by simp
     also have d-of dmu′ j′ = d fs′ j′
       using j′ d′dmu′ d-def Suc-diff-1 less-imp-diff-less unfolding d-of-def
       by (cases j′, auto)
     finally have dmu′-mod $$ (i′,j′) = dmu′ $$ (i′,j′) symmod (p ∗ d fs′ j′ ∗
d fs′ (Suc j′))
       using ddμ[OF j′] d′dmu′ j′ by (auto simp: d-of-def)
     then show ?thesis using i′j′I 11 i′j′ eq′ by simp
    qed
   next
    case False
    have dμ fs′′ i′ j′ = dμ fs′ i′ j′ using False 11 i′j′ by simp
    also have ... = dmu′ $$ (i′, j′) unfolding eq′ ..
    finally show ?thesis unfolding dmu′-mod-def using False[unfolded I-def]
i′j′ by auto
   qed
  }
   moreover have ∀ i′ j′. i′ < m ⟶ j′ < m ⟶ i′ = j′ ⟶ dμ fs′′ i′ j′ =
dmu′-mod $$ (i′, j′)
   using ddμ dmu′-def 033 10 dmu′-mod-def 11 I-def by simp
  moreover {
   fix i′ j′
   assume i′j′′: i′ < m j′ < m i′ < j′
   then have μz: μ fs′′ i′ j′ = 0 by (simp add: gram-schmidt-fs.μ.simps)
   have dmu′-mod $$ (i′,j′) = dmu′ $$ (i′,j′) using dmu′-mod-def i′j′′ by auto
    also have ... = dμ fs i′ j′ using LLL-invD-modw(7)[OF Linvmw] i′j′′
dmu′-def by simp
   also have ... = 0 using dμ-def i′j′′ by (simp add: gram-schmidt-fs.μ.simps)
   finally have dμ fs′′ i′ j′ =  dmu′-mod $$ (i′,j′) using μz d-def i′j′′ dμ-def
```

by *simp*
}
ultimately show *?thesis* **by** (*meson nat-neq-iff*)
qed
from *gbnd 09* **have** *g-bnd*: *g-bnd-mode first b fs''* **using** *g-bnd-mode-cong*[*of fs' fs''*] **by** *auto*
{
assume *Linv*: *LLL-invariant-mod fs mfs dmu p first b k*
have *00*: *LLL-invariant-weak' k fs* **using** *LLL-invD-mod*[*OF Linv*] *LLL-invI-weak* **by** *simp*
note *swap-weak' = basis-reduction-swap-weak'*[*OF 00 k k0 mu-F1-i norm-ineq fs'-def*]
have *01*: *LLL-invariant-weak' (k − 1) fs'* **by** *fact*
have *12*: *weakly-reduced fs'' (k−1)*
using *031 09 k LLL-invD-weak*(*8*)[*OF 00*] **unfolding** *gram-schmidt-fs.weakly-reduced-def* **by** *simp*
have *LLL-invariant-mod fs'' mfs' dmu'-mod p first b (k−1)*
using *LLL-invI-mod*[*OF 13 - 08 05 07 12 14 16 17 p1 g-bnd LLL-invD-mod*(*17*)[*OF Linv*]] *k* **by** *simp*
}
moreover have *LLL-invariant-mod-weak fs'' mfs' dmu'-mod p first b*
using *LLL-invI-modw*[*OF 13 08 05 07 14 16 17 p1 g-bnd LLL-invD-modw*(*15*)[*OF Linvmw*]] **by** *simp*
ultimately show *?thesis* **using** *15* **by** *auto*
qed

lemma *dmu-quot-is-round-of-μ*:
assumes *Linv*: *LLL-invariant-mod fs mfs dmu p first b i'*
and *c*: *c = round-num-denom (dmu $$ (i,j)) (d-of dmu (Suc j))*
and *i*: *i < m*
and *j*: *j < i*
shows *c = round*(*μ fs i j*)
proof −
have *Linvw*: *LLL-invariant-weak' i' fs* **using** *LLL-invD-mod*[*OF Linv*] *LLL-invI-weak* **by** *simp*
have *j2*: *j < m* **using** *i j* **by** *simp*
then have *j3*: *Suc j ≤ m* **by** *simp*
have *μ1*: *μ fs j j = 1* **using** *i j* **by** (*meson gram-schmidt-fs.μ.elims less-irrefl-nat*)
have *inZ*: *rat-of-int (d fs (Suc j)) ∗ μ fs i j ∈ ℤ* **using** *fs-int-indpt.fs-int-mu-d-Z-m-m i j*
LLL-invD-mod(*5*)[*OF Linv*] *LLL-invD-weak*(*2*) *Linvw d-def fs-int.d-def fs-int-indpt.intro* **by** *auto*
have *c = round*(*rat-of-int (dμ fs i j) / rat-of-int (dμ fs j j)*) **using** *LLL-invD-mod*(*9*) *Linv i j c*
by (*simp add*: *round-num-denom d-of-def*)
then show *?thesis* **using** *LLL-d-pos*[*OF LLL-invw'-imp-w*[*OF Linvw*] *j3*] *j i inZ dμ-def μ1* **by** *simp*
qed

lemma *dmu-quot-is-round-of-μ-weak*:
assumes *Linv*: *LLL-invariant-mod-weak fs mfs dmu p first b*
and *c*: *c* = *round-num-denom* (*dmu* $$ (*i*,*j*)) (*d-of dmu* (*Suc j*))
and *i*: *i* < *m*
and *j*: *j* < *i*
shows *c* = *round*(*μ fs i j*)
proof −
have *Linvww*: *LLL-invariant-weak fs* **using** *LLL-invD-modw*[*OF Linv*] *LLL-invariant-weak-def*
by *simp*
have *j2*: *j* < *m* **using** *i j* **by** *simp*
then have *j3*: *Suc j* ≤ *m* **by** *simp*
have *μ1*: *μ fs j j* = *1* **using** *i j* **by** (*meson gram-schmidt-fs.μ.elims less-irrefl-nat*)
have *inZ*: *rat-of-int* (*d fs* (*Suc j*)) ∗ *μ fs i j* ∈ ℤ **using** *fs-int-indpt.fs-int-mu-d-Z-m-m*
i j
LLL-invD-modw[*OF Linv*] *d-def fs-int.d-def fs-int-indpt.intro* **by** *auto*
have *c* = *round*(*rat-of-int* (*dμ fs i j*) / *rat-of-int* (*dμ fs j j*)) **using** *LLL-invD-modw*(*7*)
Linv i j c
by (*simp add*: *round-num-denom d-of-def*)
then show *?thesis* **using** *LLL-d-pos*[*OF Linvww j3*] *j i inZ dμ-def μ1* **by** *simp*
qed

lemma *basis-reduction-mod-add-row*: **assumes**
Linv: *LLL-invariant-mod-weak fs mfs dmu p first b*
and *res*: *basis-reduction-mod-add-row p mfs dmu i j* = (*mfs*′, *dmu*′)
and *i*: *i* < *m*
and *j*: *j* < *i*
and *igtz*: *i* ≠ *0*
shows (∃ *fs*′. *LLL-invariant-mod-weak fs*′ *mfs*′ *dmu*′ *p first b* ∧
LLL-measure i fs′ = *LLL-measure i fs* ∧
(*μ-small-row i fs* (*Suc j*) ⟶ *μ-small-row i fs*′ *j*) ∧
|*μ fs*′ *i j*| ≤ *1* / *2* ∧
(∀ *i*′ *j*′. *i*′ < *i* ⟶ *j*′ ≤ *i*′ ⟶ *μ fs*′ *i*′ *j*′ = *μ fs i*′ *j*′) ∧
(*LLL-invariant-mod fs mfs dmu p first b i* ⟶ *LLL-invariant-mod fs*′ *mfs*′
dmu′ *p first b i*) ∧
(∀ *ii* ≤ *m*. *d fs*′ *ii* = *d fs ii*))
proof −
define *c* **where** *c* = *round-num-denom* (*dmu* $$ (*i*,*j*)) (*d-of dmu* (*Suc j*))
then have *c*: *c* = *round*(*μ fs i j*) **using** *dmu-quot-is-round-of-μ-weak*[*OF Linv*
c-def i j] **by** *simp*
show *?thesis*
proof (*cases c* = *0*)
case *True*
then have *pair-id*: (*mfs*′, *dmu*′) = (*mfs*, *dmu*)
using *res c-def* **unfolding** *basis-reduction-mod-add-row-def Let-def* **by** *auto*
moreover have |*μ fs i j*| ≤ *inverse 2* **using** *c*[*symmetric*, *unfolded True*]
by (*simp add*: *round-def*, *linarith*)
moreover then have (*μ-small-row i fs* (*Suc j*) ⟶ *μ-small-row i fs j*)
unfolding *μ-small-row-def* **using** *Suc-leI le-neq-implies-less* **by** *blast*
ultimately show *?thesis* **using** *Linv pair-id* **by** *auto*

next
case *False*
then have *pair-id*: (*mfs'*, *dmu'*) = (*mfs*[*i* := *map-vec* (λ*x*. *x symmod p*) (*mfs* ! *i* − *c* ·$_v$ *mfs* ! *j*)],
mat m m (λ(*i'*, *j'*). *if i'* = *i* ∧ *j'* ≤ *j*
then if j' = *j then dmu* $$ (*i*, *j'*) − *c* ∗ *dmu* $$ (*j*, *j'*)
else (*dmu* $$ (*i*,*j'*) − *c* ∗ *dmu* $$ (*j*,*j'*))
symmod (*p* ∗ (*d-of dmu j'*) ∗ (*d-of dmu* (*Suc j'*)))
else dmu $$ (*i'*, *j'*)))
using *res c-def* **unfolding** *basis-reduction-mod-add-row-def Let-def* **by** *auto*
then have *mfs'*: *mfs'* = *mfs*[*i* := *map-vec* (λ*x*. *x symmod p*) (*mfs* ! *i* − *c* ·$_v$ *mfs* ! *j*)]
and *dmu'*: *dmu'* = *mat m m* (λ(*i'*, *j'*). *if i'* = *i* ∧ *j'* ≤ *j*
then if j' = *j then dmu* $$ (*i*, *j'*) − *c* ∗ *dmu* $$ (*j*, *j'*)
else (*dmu* $$ (*i*,*j'*) − *c* ∗ *dmu* $$ (*j*,*j'*))
symmod (*p* ∗ (*d-of dmu j'*) ∗ (*d-of dmu* (*Suc j'*)))
else dmu $$ (*i'*, *j'*)) **by** *auto*
show *?thesis* **using** *basis-reduction-mod-add-row-main*[*OF Linv i j c mfs' dmu'*]
by *blast*
qed
qed

lemma *basis-reduction-mod-swap*: **assumes**
Linv: *LLL-invariant-mod-weak fs mfs dmu p first b*
and *mu*: |*μ fs k* (*k*−1)| ≤ *1 / 2*
and *res*: *basis-reduction-mod-swap p mfs dmu k* = (*mfs'*, *dmu'-mod*)
and *cond*: *sq-norm* (*gso fs* (*k* − *1*)) > *α* ∗ *sq-norm* (*gso fs k*)
and *i*: *k* < *m k* ≠ *0*
shows (∃*fs'*. *LLL-invariant-mod-weak fs' mfs' dmu'-mod p first b* ∧
LLL-measure (*k* − *1*) *fs'* < *LLL-measure k fs* ∧
(*LLL-invariant-mod fs mfs dmu p first b k* ⟶ *LLL-invariant-mod fs' mfs' dmu'-mod p first b* (*k*−*1*)))
using *res*[*unfolded basis-reduction-mod-swap-def basis-reduction-mod-swap-dmu-mod-def*]

basis-reduction-mod-swap-main[*OF Linv i mu cond*] **by** *blast*

lemma *basis-reduction-adjust-mod*: **assumes**
Linv: *LLL-invariant-mod-weak fs mfs dmu p first b*
and *res*: *basis-reduction-adjust-mod p first mfs dmu* = (*p'*, *mfs'*, *dmu'*, *g-idx'*)
shows (∃*fs' b'*. (*LLL-invariant-mod fs mfs dmu p first b i* ⟶ *LLL-invariant-mod fs' mfs' dmu' p' first b' i*) ∧
LLL-invariant-mod-weak fs' mfs' dmu' p' first b' ∧
LLL-measure i fs' = *LLL-measure i fs*)
proof (*cases* ∃ *g-idx*. *basis-reduction-adjust-mod p first mfs dmu* = (*p*, *mfs*, *dmu*, *g-idx*))
case *True*
thus *?thesis* **using** *res Linv* **by** *auto*
next
case *False*

obtain b' *g-idx* **where** *norm*: *compute-max-gso-norm first dmu* = (b', *g-idx*) **by** *force*
define p'' **where** p'' = *compute-mod-of-max-gso-norm first* b'
define *d-vec* **where** *d-vec* = *vec* (*Suc m*) (λ*i*. *d-of dmu i*)
define *mfs''* **where** *mfs''* = *map* (*map-vec* (λ*x*. *x symmod* p'')) *mfs*
define *dmu''* **where** *dmu''* = *mat m m* (λ(*i*, *j*).
if j < *i then dmu* $$ (*i*, *j*) *symmod* (p'' ∗ *d-vec* $ *j* ∗ *d-vec* $ *Suc j*)
else dmu $$ (*i*, *j*))
note *res* = *res False*
note *res* = *res*[*unfolded basis-reduction-adjust-mod.simps Let-def norm split*,
folded p''*-def*, *folded d-vec-def mfs''-def*, *folded dmu''-def*]
from *res* **have** *pp'*: p'' < *p* **and** *id*: *dmu'* = *dmu''* *mfs'* = *mfs''* p' = p'' *g-idx'* = *g-idx*
by (*auto split*: *if-splits*)
define *I* **where** *I* = {(*i'*,*j'*). *i'* < *m* ∧ *j'* < *i'*}
note *inv* = *LLL-invD-modw*[*OF Linv*]
from *inv*(*4*) **have** *lin*: *gs.lin-indpt-list* (*RAT fs*) **.**
from *inv*(*3*) **have** *lat*: *lattice-of fs* = *L* **.**
from *inv*(*2*) **have** *len*: *length fs* = *m* **.**
have *weak*: *LLL-invariant-weak fs* **using** *Linv*
by (*auto simp*: *LLL-invariant-mod-weak-def LLL-invariant-weak-def*)
from *compute-max-gso-norm*[*OF - weak*, *of dmu first*, *unfolded norm*] *inv*(*7*)
have *bnd*: *g-bnd-mode first* b' *fs* **and** b': $b' \geq 0$ *m* = *0* ⟹ b' = *0* **by** *auto*
from *compute-mod-of-max-gso-norm*[*OF* b' p''*-def*]
have p'': *0* < p'' *1* < p'' *mod-invariant* b' p'' *first*
by *auto*
obtain *fs'* **where**
01: *lattice-of fs'* = *L* **and**
02: *map* (*map-vec* (λ *x*. *x symmod* p'')) *fs'* = *map* (*map-vec* (λ *x*. *x symmod* p'')) *fs* **and**
03: *lin-indep fs'* **and**
04: *length fs'* = *m* **and**
05: (∀ *k* < *m*. *gso fs' k* = *gso fs k*) **and**
06: (∀ *k* ≤ *m*. *d fs' k* = *d fs k*) **and**
07: (∀ *i'* < *m*. ∀ *j'* < *m*. *dμ fs' i' j'* =
(*if* (*i'*,*j'*) ∈ *I then dμ fs i' j' symmod* (p'' ∗ *d fs j'* ∗ *d fs* (*Suc j'*)) *else dμ fs i' j'*))
using *mod-finite-set*[*OF lin len - lat*, *of I*] *I-def* p'' **by** *blast*
from *bnd 05* **have** *bnd*: *g-bnd-mode first* b' *fs'* **using** *g-bnd-mode-cong*[*of fs fs'*] **by** *auto*
have *D*: *D fs* = *D fs'* **unfolding** *D-def* **using** *06* **by** *auto*

have *Linv'*: *LLL-invariant-mod-weak fs' mfs''* *dmu''* p'' *first* b'
proof (*intro LLL-invI-modw* p'' *04 03 01 bnd*)
{
have *mfs''* = *map* (*map-vec* (λ*x*. *x symmod* p'')) *mfs* **by** *fact*
also have ... = *map* (*map-vec* (λ*x*. *x symmod* p'')) (*map* (*map-vec* (λ*x*. *x symmod p*)) *fs*)

```
      using inv by simp
    also have ... = map (map-vec (λx. x symmod p symmod p'')) fs by auto
    also have (λ x. x symmod p symmod p'') = (λ x. x symmod p'')
    proof (intro ext)
      fix x
     from ‹mod-invariant b p first›[unfolded mod-invariant-def] obtain e where

        p: p = log-base ^ e by auto
      from p''[unfolded mod-invariant-def] obtain e' where
        p'': p'' = log-base ^ e' by auto
      from pp'[unfolded p p''] log-base have e' ≤ e by simp
    hence dvd: p'' dvd p unfolding p p'' using log-base by (metis le-imp-power-dvd)
      thus x symmod p symmod p'' = x symmod p''
        by (intro sym-mod-sym-mod-cancel)
    qed
    finally show map (map-vec (λx. x symmod p'')) fs' = mfs'' unfolding 02 ..
  }
  thus length mfs'' = m using 04 by auto
  show ∀ i'<m. ∀ j'<i'. |dμ fs' i' j'| < p'' * d fs' j' * d fs' (Suc j')
  proof −
    {
      fix i' j'
      assume i'j': i' < m j' < i'
      then have dμ fs' i' j' = dμ fs i' j' symmod (p'' * d fs' j' * d fs' (Suc j'))
        using 07 06 unfolding I-def by simp
      then have |dμ fs' i' j'| < p'' * d fs' j' * d fs' (Suc j')
        using sym-mod-abs p'' LLL-d-pos[OF weak] mult-pos-pos
        by (smt 06 i'j' less-imp-le-nat less-trans-Suc nat-SN.gt-trans)
    }
    then show ?thesis by simp
  qed
  from inv(7) have dmu: i' < m ⟹ j' < m ⟹ dmu $$ (i', j') = dμ fs i' j'
for i' j'
    by auto
  note d-of = d-of-weak[OF Linv]
  have dvec: i ≤ m ⟹ d-vec $ i = d fs i for i unfolding d-vec-def using d-of
by auto
  show ∀ i'<m. ∀ j'<m. dμ fs' i' j' = dmu'' $$ (i', j')
    using 07 unfolding dmu''-def I-def
    by (auto simp: dmu dvec)
 qed

 moreover
 {
  assume linv: LLL-invariant-mod fs mfs dmu p first b i
  note inv = LLL-invD-mod[OF linv]
  hence i: i ≤ m by auto
  have norm: j < m ⟹ ‖gso fs j‖² = ‖gso fs' j‖² for j
    using 05 by auto
```

```
    have weakly-reduced fs i = weakly-reduced fs' i
      unfolding gram-schmidt-fs.weakly-reduced-def using i
      by (intro all-cong arg-cong2[where f = (≤)] arg-cong[where f = λ x. - * x]
norm, auto)
    with inv have weakly-reduced fs' i by auto
    hence LLL-invariant-mod fs' mfs'' dmu'' p'' first b' i using inv
      by (intro LLL-invI-mod LLL-invD-modw[OF Linv'])
  }

  moreover have LLL-measure i fs' = LLL-measure i fs
    unfolding LLL-measure-def logD-def D ..
  ultimately show ?thesis unfolding id by blast
qed

lemma alpha-comparison: assumes
  Linv: LLL-invariant-mod-weak fs mfs dmu p first b
  and alph: quotient-of α = (num, denom)
  and i: i < m
  and i0: i ≠ 0
shows (d-of dmu i * d-of dmu i * denom ≤ num * d-of dmu (i − 1) * d-of dmu
(Suc i))
  = (sq-norm (gso fs (i − 1)) ≤ α * sq-norm (gso fs i))
proof −
  note inv = LLL-invD-modw[OF Linv]
  interpret fs-indep: fs-int-indpt n fs
    by (unfold-locales, insert inv, auto)
  from inv(2) i have ifs: i < length fs by auto
  note d-of-fs = d-of-weak[OF Linv]
  show ?thesis
    unfolding fs-indep.d-sq-norm-comparison[OF alph ifs i0, symmetric]
    by (subst (1 2 3 4) d-of-fs, use i d-def fs-indep.d-def in auto)
qed

lemma basis-reduction-adjust-swap-add-step: assumes
  Linv: LLL-invariant-mod-weak fs mfs dmu p first b
  and res: basis-reduction-adjust-swap-add-step p first mfs dmu g-idx i = (p', mfs',
dmu', g-idx')
  and alph: quotient-of α = (num, denom)
  and ineq: ¬ (d-of dmu i * d-of dmu i * denom
          ≤ num * d-of dmu (i − 1) * d-of dmu (Suc i))
  and i: i < m
  and i0: i ≠ 0
shows ∃fs' b'. LLL-invariant-mod-weak fs' mfs' dmu' p' first b' ∧
      LLL-measure (i − 1) fs' < LLL-measure i fs ∧
      LLL-measure (m − 1) fs' < LLL-measure (m − 1) fs ∧
      (LLL-invariant-mod fs mfs dmu p first b i ⟶
       LLL-invariant-mod fs' mfs' dmu' p' first b' (i − 1))
proof −
  obtain mfs0 dmu0 where add: basis-reduction-mod-add-row p mfs dmu i (i−1)
```

= (*mfs0*, *dmu0*) **by** *force*
obtain *mfs1 dmu1* **where** *swap*: *basis-reduction-mod-swap p mfs0 dmu0 i* = (*mfs1*, *dmu1*) **by** *force*
note *res* = *res*[*unfolded basis-reduction-adjust-swap-add-step-def Let-def add split swap*]
from *i0* **have** *ii*: $i - 1 < i$ **by** *auto*
from *basis-reduction-mod-add-row*[*OF Linv add i ii i0*]
obtain *fs0* **where** *Linv0*: *LLL-invariant-mod-weak fs0 mfs0 dmu0 p first b*
and *meas0*: *LLL-measure i fs0* = *LLL-measure i fs*
and *small*: $|\mu\ fs0\ i\ (i - 1)| \leq 1\ /\ 2$
and *Linv0'*: *LLL-invariant-mod fs mfs dmu p first b i* $\Longrightarrow$ *LLL-invariant-mod fs0 mfs0 dmu0 p first b i*
by *blast*
{
have *id*: *d-of dmu0 i* = *d-of dmu i d-of dmu0* $(i - 1)$ = *d-of dmu* $(i - 1)$ *d-of dmu0* (*Suc i*) = *d-of dmu* (*Suc i*)
using *i i0 add*[*unfolded basis-reduction-mod-add-row-def Let-def*]
by (*auto split*: *if-splits simp*: *d-of-def*)
from *ineq*[*folded id*, *unfolded alpha-comparison*[*OF Linv0 alph i i0*]]
have $\|gso\ fs0\ (i - 1)\|^2 > \alpha * \|gso\ fs0\ i\|^2$ **by** *simp*
} **note** *ineq* = *this*
from *Linv* **have** *LLL-invariant-weak fs*
by (*auto simp*: *LLL-invariant-weak-def LLL-invariant-mod-weak-def*)
from *basis-reduction-mod-swap*[*OF Linv0 small swap ineq i i0*, *unfolded meas0*] *Linv0'*
obtain *fs1* **where** *Linv1*: *LLL-invariant-mod-weak fs1 mfs1 dmu1 p first b*
and *meas1*: *LLL-measure* $(i - 1)$ *fs1* < *LLL-measure i fs*
and *Linv1'*: *LLL-invariant-mod fs mfs dmu p first b i* $\Longrightarrow$ *LLL-invariant-mod fs1 mfs1 dmu1 p first b* $(i - 1)$
by *auto*
show *?thesis*
proof (*cases* $i - 1$ = *g-idx*)
case *False*
with *res* **have** *id*: $p' = p$ $mfs' = mfs1$ $dmu' = dmu1$ $g\text{-}idx' = g\text{-}idx$ **by** *auto*
show *?thesis* **unfolding** *id* **using** *Linv1' meas1 Linv1* **by** (*intro exI*[*of - fs1*] *exI*[*of - b*], *auto simp*: *LLL-measure-def*)
next
case *True*
with *res* **have** *adjust*: *basis-reduction-adjust-mod p first mfs1 dmu1* = (p', mfs', dmu', $g\text{-}idx'$) **by** *simp*
from *basis-reduction-adjust-mod*[*OF Linv1 adjust*, *of* $i - 1$] *Linv1'*
obtain fs' b' **where** *Linvw*: *LLL-invariant-mod-weak* fs' mfs' dmu' p' *first* b'
and *Linv*: *LLL-invariant-mod fs mfs dmu p first b i* $\Longrightarrow$ *LLL-invariant-mod* fs' mfs' dmu' p' *first* b' $(i - 1)$
and *meas*: *LLL-measure* $(i - 1)$ fs' = *LLL-measure* $(i - 1)$ *fs1*
by *blast*
note *meas* = *meas1*[*folded meas*]
from *meas* **have** *meas'*: *LLL-measure* $(m - 1)$ fs' < *LLL-measure* $(m - 1)$ *fs*
unfolding *LLL-measure-def* **using** *i* **by** *auto*

```
    show ?thesis
      by (intro exI conjI impI, rule Linvw, rule meas, rule meas', rule Linv)
  qed
qed


lemma basis-reduction-mod-step: assumes
  Linv: LLL-invariant-mod fs mfs dmu p first b i
  and res: basis-reduction-mod-step p first mfs dmu g-idx i j = (p', mfs', dmu',
g-idx', i', j')
  and i: i < m
shows ∃fs' b'. LLL-measure i' fs' < LLL-measure i fs ∧ LLL-invariant-mod fs'
mfs' dmu' p' first b' i'
proof −
  note res = res[unfolded basis-reduction-mod-step-def Let-def]
  from Linv have Linvw: LLL-invariant-mod-weak fs mfs dmu p first b
    by (auto simp: LLL-invariant-mod-weak-def LLL-invariant-mod-def)
  show ?thesis
  proof (cases i = 0)
    case True
    then have ids: mfs' = mfs dmu' = dmu i' = Suc i p' = p using res by auto
    have LLL-measure i' fs < LLL-measure i fs ∧ LLL-invariant-mod fs mfs' dmu'
p first b i'
      using increase-i-mod[OF Linv i] True res ids inv by simp
    then show ?thesis using res ids inv by auto
  next
    case False
    hence id: (i = 0) = False by auto
    obtain num denom where alph: quotient-of α = (num, denom) by force
    note res = res[unfolded id if-False alph split]
    let ?comp = d-of dmu i ∗ d-of dmu i ∗ denom ≤ num ∗ d-of dmu (i − 1) ∗
d-of dmu (Suc i)
    show ?thesis
    proof (cases ?comp)
      case False
      hence id: ?comp = False by simp
      note res = res[unfolded id if-False]
      let ?step = basis-reduction-adjust-swap-add-step p first mfs dmu g-idx i
      from res have step: ?step = (p', mfs', dmu', g-idx')
        and i': i' = i − 1
        by (cases ?step, auto)+
      from basis-reduction-adjust-swap-add-step[OF Linvw step alph False i ‹i ≠
0›] Linv
      show ?thesis unfolding i' by blast
    next
      case True
      hence id: ?comp = True by simp
      note res = res[unfolded id if-True]
      from res have ids: p' = p mfs' = mfs dmu' = dmu i' = Suc i by auto
```

from *True alpha-comparison*[*OF Linvw alph i False*]
have *ineq*: *sq-norm* (*gso fs* (*i* − *1*)) ≤ *α* ∗ *sq-norm* (*gso fs i*) **by** *simp*
from *increase-i-mod*[*OF Linv i ineq*]
show *?thesis* **unfolding** *ids* **by** *auto*
qed
qed
qed

lemma *basis-reduction-mod-main*: **assumes** *LLL-invariant-mod fs mfs dmu p first b i*
and *res*: *basis-reduction-mod-main p first mfs dmu g-idx i j* = (*p′*, *mfs′*, *dmu′*)
shows ∃*fs′ b′*. *LLL-invariant-mod fs′ mfs′ dmu′ p′ first b′ m*
using *assms*
proof (*induct LLL-measure i fs arbitrary*: *i mfs dmu j p b fs g-idx rule*: *less-induct*)
case (*less i fs mfs dmu j p b g-idx*)
hence *fsinv*: *LLL-invariant-mod fs mfs dmu p first b i* **by** *auto*
note *res* = *less*(*3*)[*unfolded basis-reduction-mod-main.simps*[*of p first mfs dmu g-idx i j*]]
note *inv* = *less*(*2*)
note *IH* = *less*(*1*)
show *?case*
proof (*cases i* < *m*)
case *i*: *True*
obtain *p′ mfs′ dmu′ g-idx′ i′ j′* **where** *step*: *basis-reduction-mod-step p first mfs dmu g-idx i j* = (*p′*, *mfs′*, *dmu′*, *g-idx′*, *i′*, *j′*)
(**is** *?step* = -) **by** (*cases ?step*, *auto*)
then obtain *fs′ b′* **where** *Linv*: *LLL-invariant-mod fs′ mfs′ dmu′ p′ first b′ i′*
and *decr*: *LLL-measure i′ fs′* < *LLL-measure i fs*
using *basis-reduction-mod-step*[*OF fsinv step i*] *i fsinv* **by** *blast*
note *res* = *res*[*unfolded step split*]
from *res i* **show** *?thesis* **using** *IH*[*OF decr Linv*] **by** *auto*
next
case *False*
with *LLL-invD-mod*[*OF fsinv*] *res* **have** *i*: *i* = *m p′* = *p* **by** *auto*
then obtain *fs′ b′* **where** *LLL-invariant-mod fs′ mfs′ dmu′ p first b′ m* **using** *False res fsinv* **by** *simp*
then show *?thesis* **using** *i* **by** *auto*
qed
qed

lemma *compute-max-gso-quot-alpha*:
assumes *inv*: *LLL-invariant-mod-weak fs mfs dmu p first b*
and *max*: *compute-max-gso-quot dmu* = (*msq-num*, *msq-denum*, *idx*)
and *alph*: *quotient-of α* = (*num*, *denum*)
and *cmp*: (*msq-num* ∗ *denum* > *num* ∗ *msq-denum*) = *cmp*
and *m*: *m* > *1*
shows *cmp* ⟹ *idx* ≠ *0* ∧ *idx* < *m* ∧ ¬ (*d-of dmu idx* ∗ *d-of dmu idx* ∗ *denum*
≤ *num* ∗ *d-of dmu* (*idx* − *1*) ∗ *d-of dmu* (*Suc idx*))
and ¬ *cmp* ⟹ *LLL-invariant-mod fs mfs dmu p first b m*

proof −
from *inv*
have *fsinv*: *LLL-invariant-weak fs*
by (*simp add*: *LLL-invariant-mod-weak-def LLL-invariant-weak-def*)
define *qt* **where** *qt* = (λ*i*. ((*d-of dmu* (*i* + *1*)) ∗ (*d-of dmu* (*i* + *1*)),
(*d-of dmu* (*i* + *2*)) ∗ (*d-of dmu i*), *Suc i*))
define *lst* **where** *lst* = (*map* (λ*i*. *qt i*) [*0*..<(*m*−*1*)])
have *msqlst*: (*msq-num*, *msq-denum*, *idx*) = *max-list-rats-with-index lst*
using *max lst-def qt-def* **unfolding** *compute-max-gso-quot-def* **by** *simp*
have *nz*: ⋀*n d i*. (*n*, *d*, *i*) ∈ *set lst* ⟹ *d* > *0*
unfolding *lst-def qt-def* **using** *d-of-weak*[*OF inv*] *LLL-d-pos*[*OF fsinv*] **by** *auto*
have *geq*: ∀ (*n*, *d*, *i*) ∈ *set lst*. *rat-of-int msq-num* / *of-int msq-denum* ≥ *rat-of-int n* / *of-int d*
using *max-list-rats-with-index*[*of lst*] *nz msqlst* **by** (*metis* (*no-types*, *lifting*) *case-prodI2*)
have *len*: *length lst* ≥ *1* **using** *m* **unfolding** *lst-def* **by** *simp*
have *inset*: (*msq-num*, *msq-denum*, *idx*) ∈ *set lst*
using *max-list-rats-with-index-in-set*[*OF msqlst*[*symmetric*] *len*] *nz* **by** *simp*
then have *idxm*: *idx* ∈ {*1*..<*m*} **using** *lst-def*[*unfolded qt-def*] **by** *auto*
then have *idx0*: *idx* ≠ *0* **and** *idx*: *idx* < *m* **by** *auto*
have *00*: (*msq-num*, *msq-denum*, *idx*) = *qt* (*idx* − *1*) **using** *lst-def inset qt-def* **by** *auto*
then have *id-qt*: *msq-num* = *d-of dmu idx* ∗ *d-of dmu idx msq-denum* = *d-of dmu* (*Suc idx*) ∗ *d-of dmu* (*idx* − *1*)
unfolding *qt-def* **by** *auto*
have *msq-denum* = (*d-of dmu* (*idx* + *1*)) ∗ (*d-of dmu* (*idx* − *1*))
using *00* **unfolding** *qt-def* **by** *simp*
then have *dengt0*: *msq-denum* > *0* **using** *d-of-weak*[*OF inv*] *idxm LLL-d-pos*[*OF fsinv*] **by** *auto*
have *αdengt0*: *denum* > *0* **using** *alph* **by** (*metis quotient-of-denom-pos*)
from *cmp*[*unfolded id-qt*]
have *cmp*: *cmp* = (¬ (*d-of dmu idx* ∗ *d-of dmu idx* ∗ *denum* ≤ *num* ∗ *d-of dmu* (*idx* − *1*) ∗ *d-of dmu* (*Suc idx*)))
by (*auto simp*: *ac-simps*)
{
assume *cmp*
from *this*[*unfolded cmp*]
show *idx* ≠ *0* ∧ *idx* < *m* ∧ ¬ (*d-of dmu idx* ∗ *d-of dmu idx* ∗ *denum*
≤ *num* ∗ *d-of dmu* (*idx* − *1*) ∗ *d-of dmu* (*Suc idx*)) **using** *idx0 idx* **by** *auto*
}
{
assume ¬ *cmp*
from *this*[*unfolded cmp*] **have** *small*: *d-of dmu idx* ∗ *d-of dmu idx* ∗ *denum* ≤ *num* ∗ *d-of dmu* (*idx* − *1*) ∗ *d-of dmu* (*Suc idx*) **by** *auto*
note *d-pos* = *LLL-d-pos*[*OF fsinv*]
have *gso*: *k* < *m* ⟹ *sq-norm* (*gso fs k*) = *of-int* (*d fs* (*Suc k*)) / *of-int* (*d fs k*) **for** *k* **using**
LLL-d-Suc[*OF fsinv*, *of k*] *d-pos*[*of k*] **by** *simp*

have *gso-pos*: $k < m \Longrightarrow$ *sq-norm* (*gso fs k*) $>$ *0* **for** *k*
using *gso*[*of k*] *d-pos*[*of k*] *d-pos*[*of Suc k*] **by** *auto*
from *small*[*unfolded alpha-comparison*[*OF inv alph idx idx0*]]
have *alph*: *sq-norm* (*gso fs* (*idx* − *1*)) $\leq \alpha *$ *sq-norm* (*gso fs idx*) **.**
with *gso-pos*[*OF idx*] **have** *alph*: *sq-norm* (*gso fs* (*idx* − *1*)) / *sq-norm* (*gso fs idx*) $\leq \alpha$
by (*metis mult-imp-div-pos-le*)
have *weak*: *weakly-reduced fs m* **unfolding** *gram-schmidt-fs.weakly-reduced-def*
proof (*intro allI impI*, *goal-cases*)
case (*1 i*)
from *idx* **have** *idx1*: *idx* − *1* $<$ *m* **by** *auto*
from *geq*[*unfolded lst-def*]
have *mem*: (*d-of dmu* (*Suc i*) $*$ *d-of dmu* (*Suc i*),
d-of dmu (*Suc* (*Suc i*)) $*$ *d-of dmu i*, *Suc i*) $\in$ *set lst*
unfolding *lst-def qt-def* **using** *1* **by** *auto*
have *sq-norm* (*gso fs i*) / *sq-norm* (*gso fs* (*Suc i*)) =
of-int (*d-of dmu* (*Suc i*) $*$ *d-of dmu* (*Suc i*)) / *of-int* (*d-of dmu* (*Suc* (*Suc i*)) $*$ *d-of dmu i*)
using *gso idx0 d-of-weak*[*OF inv*] *1* **by** *auto*
also have ... $\leq$ *rat-of-int msq-num* / *rat-of-int msq-denum*
using *geq*[*rule-format*, *OF mem*, *unfolded split*] **by** *auto*
also have ... = *sq-norm* (*gso fs* (*idx* − *1*)) / *sq-norm* (*gso fs idx*)
unfolding *id-qt gso*[*OF idx*] *gso*[*OF idx1*] **using** *idx0 d-of-weak*[*OF inv*] *idx* **by** *auto*
also have ... $\leq \alpha$ **by** *fact*
finally show *sq-norm* (*gso fs i*) $\leq \alpha *$ *sq-norm* (*gso fs* (*Suc i*)) **using** *gso-pos*[*OF 1*]
using *pos-divide-le-eq* **by** *blast*
qed
with *inv* **show** *LLL-invariant-mod fs mfs dmu p first b m*
by (*auto simp*: *LLL-invariant-mod-weak-def LLL-invariant-mod-def*)
}
qed

lemma *small-m*:
assumes *inv*: *LLL-invariant-mod-weak fs mfs dmu p first b*
and *m*: *m* $\leq$ *1*
shows *LLL-invariant-mod fs mfs dmu p first b m*
proof −
have *weak*: *weakly-reduced fs m* **unfolding** *gram-schmidt-fs.weakly-reduced-def* **using** *m*
by *auto*
with *inv* **show** *LLL-invariant-mod fs mfs dmu p first b m*
by (*auto simp*: *LLL-invariant-mod-weak-def LLL-invariant-mod-def*)
qed

lemma *basis-reduction-iso-main*: **assumes** *LLL-invariant-mod-weak fs mfs dmu p first b*

and *res*: *basis-reduction-iso-main p first mfs dmu g-idx j* = (*p'*, *mfs'*, *dmu'*)
shows ∃*fs' b'*. *LLL-invariant-mod fs' mfs' dmu' p' first b' m*
using *assms*
proof (*induct LLL-measure* (*m*−*1*) *fs arbitrary*: *fs mfs dmu j p b g-idx rule*: *less-induct*)
case (*less fs mfs dmu j p b g-idx*)
have *inv*: *LLL-invariant-mod-weak fs mfs dmu p first b* **using** *less* **by** *auto*
hence *fsinv*: *LLL-invariant-weak fs*
by (*simp add*: *LLL-invariant-mod-weak-def LLL-invariant-weak-def*)
note *res* = *less*(*3*)[*unfolded basis-reduction-iso-main.simps*[*of p first mfs dmu g-idx j*]]
note *IH* = *less*(*1*)
obtain *msq-num msq-denum idx* **where** *max*: *compute-max-gso-quot dmu* = (*msq-num*, *msq-denum*, *idx*)
by (*metis prod-cases3*)
obtain *num denum* **where** *alph*: *quotient-of* α = (*num*, *denum*) **by** *force*
note *res* = *res*[*unfolded max alph Let-def split*]
consider (*small*) *m* ≤ *1* | (*final*) *m* > *1* ¬ (*num* ∗ *msq-denum* < *msq-num* ∗ *denum*) | (*step*) *m* > *1 num* ∗ *msq-denum* < *msq-num* ∗ *denum*
by *linarith*
thus *?case*
proof *cases*
case ∗: *step*
obtain *p1 mfs1 dmu1 g-idx1* **where** *step*: *basis-reduction-adjust-swap-add-step p first mfs dmu g-idx idx* = (*p1*, *mfs1*, *dmu1*, *g-idx1*)
by (*metis prod-cases4*)
from *res*[*unfolded step split*] ∗ **have** *res*: *basis-reduction-iso-main p1 first mfs1 dmu1 g-idx1* (*j* + *1*) = (*p'*, *mfs'*, *dmu'*) **by** *auto*
from *compute-max-gso-quot-alpha*(*1*)[*OF inv max alph refl* ∗]
have *idx0*: *idx* ≠ *0* **and** *idx*: *idx* < *m* **and** *cmp*: ¬ *d-of dmu idx* ∗ *d-of dmu idx* ∗ *denum* ≤ *num* ∗ *d-of dmu* (*idx* − *1*) ∗ *d-of dmu* (*Suc idx*) **by** *auto*
from *basis-reduction-adjust-swap-add-step*[*OF inv step alph cmp idx idx0*] **obtain** *fs1 b1*
where *inv1*: *LLL-invariant-mod-weak fs1 mfs1 dmu1 p1 first b1* **and** *meas*: *LLL-measure* (*m* − *1*) *fs1* < *LLL-measure* (*m* − *1*) *fs*
by *auto*
from *IH*[*OF meas inv1 res*] **show** *?thesis* **.**
next
case *small*
with *res small-m*[*OF inv*] **show** *?thesis* **by** *auto*
next
case *final*
from *compute-max-gso-quot-alpha*(*2*)[*OF inv max alph refl final*]
final **show** *?thesis* **using** *res* **by** *auto*
qed
qed

lemma *basis-reduction-mod-add-rows-loop-inv'*: **assumes**
fsinv: *LLL-invariant-mod fs mfs dmu p first b m*

and *res*: *basis-reduction-mod-add-rows-loop p mfs dmu i i* = (*mfs'*, *dmu'*)
and *i*: *i* < *m*
shows ∃*fs'*. *LLL-invariant-mod fs' mfs' dmu' p first b m* ∧
(∀ *i' j'*. *i'* < *i* ⟶ *j'* ≤ *i'* ⟶ *μ fs i' j'* = *μ fs' i' j'*) ∧
μ-small fs' i
proof −
{
fix *j*
assume *j*: *j* ≤ *i* **and** *mu-small*: *μ-small-row i fs j*
and *resj*: *basis-reduction-mod-add-rows-loop p mfs dmu i j* = (*mfs'*, *dmu'*)
have ∃*fs'*. *LLL-invariant-mod fs' mfs' dmu' p first b m* ∧
(∀ *i' j'*. *i'* < *i* ⟶ *j'* ≤ *i'* ⟶ *μ fs i' j'* = *μ fs' i' j'*) ∧
(*μ-small fs' i*)
proof (*insert fsinv mu-small resj i j*, *induct j arbitrary*: *fs mfs dmu mfs' dmu'*)
case (*0 fs*)
then have (*mfs'*, *dmu'*) = (*mfs*, *dmu*) **by** *simp*
then show *?case*
using *LLL-invariant-mod-to-weak-m-to-i*(*3*) *basis-reduction-add-row-done-weak 0* **by** *auto*
next
case (*Suc j*)
hence *j*: *j* < *i* **by** *auto*
have *in0*: *i* ≠ *0* **using** *Suc*(*6*) **by** *simp*
define *c* **where** *c* = *round-num-denom* (*dmu* $$ (*i*,*j*)) (*d-of dmu* (*Suc j*))
have *c2*: *c* = *round* (*μ fs i j*) **using** *dmu-quot-is-round-of-μ*[*OF - - i j*] *c-def Suc* **by** *simp*
define *mfs''* **where** *mfs''* = (*if c=0 then mfs else mfs*[*i* := (*map-vec* (λ *x*. *x symmod p*)) (*mfs* ! *i* − *c* ·$_v$ *mfs* ! *j*)])
define *dmu''* **where** *dmu''* = (*if c=0 then dmu else mat m m* (λ(*i'*,*j'*). (*if* (*i'* = *i* ∧ *j'* ≤ *j*)
then (*if j'=j then* (*dmu* $$ (*i*,*j'*) − *c* ∗ *dmu* $$ (*j*,*j'*))
else (*dmu* $$ (*i*,*j'*) − *c* ∗ *dmu* $$ (*j*,*j'*)) *symmod* (*p* ∗ (*d-of dmu j'*) ∗ (*d-of dmu* (*Suc j'*))))
else (*dmu* $$ (*i'*,*j'*)))))
have *00*: *basis-reduction-mod-add-row p mfs dmu i j* = (*mfs''*, *dmu''*)
using *mfs''-def dmu''-def* **unfolding** *basis-reduction-mod-add-row-def c-def*[*symmetric*] **by** *simp*
then have *01*: *basis-reduction-mod-add-rows-loop p mfs'' dmu'' i j* = (*mfs'*, *dmu'*)
using *basis-reduction-mod-add-rows-loop.simps*(*2*)[*of p mfs dmu i j*] *Suc* **by** *simp*
have *fsinvi*: *LLL-invariant-mod fs mfs dmu p first b i* **using** *LLL-invariant-mod-to-weak-m-to-i*[*OF Suc*(*2*)] *i* **by** *simp*
then have *fsinvmw*: *LLL-invariant-mod-weak fs mfs dmu p first b* **using** *LLL-invD-mod LLL-invI-modw* **by** *simp*
obtain *fs''* **where** *fs''invi*: *LLL-invariant-mod fs'' mfs'' dmu'' p first b i* **and**
μ-small': (*μ-small-row i fs* (*Suc j*) ⟶ *μ-small-row i fs'' j*) **and**
μs: (∀ *i' j'*. *i'* < *i* ⟶ *j'* ≤ *i'* ⟶ *μ fs'' i' j'* = *μ fs i' j'*)

```
      using Suc basis-reduction-mod-add-row[OF fsinvmw 00 i j] fsinvi by auto
    moreover then have μsm: μ-small-row i fs'' j using Suc by simp
   have fs''invwi: LLL-invariant-weak' i fs'' using LLL-invD-mod[OF fs''invi]
LLL-invI-weak by simp
     have fsinvwi: LLL-invariant-weak' i fs using LLL-invD-mod[OF fsinvi]
LLL-invI-weak by simp
    note invw = LLL-invw'-imp-w[OF fsinvwi]
    note invw'' = LLL-invw'-imp-w[OF fs''invwi]
    have LLL-invariant-mod fs'' mfs'' dmu'' p first b m
    proof −
      have (∀ l. Suc l < m ⟶ sq-norm (gso fs'' l) ≤ α ∗ sq-norm (gso fs''
(Suc l)))
      proof −
       {
        fix l
        assume l: Suc l < m
        have sq-norm (gso fs'' l) ≤ α ∗ sq-norm (gso fs'' (Suc l))
        proof (cases i ≤ Suc l)
         case True
         have deq: ⋀k. k < m ⟹ d fs (Suc k) = d fs'' (Suc k)
           using ddμ LLL-invD-mod(9)[OF fs''invi] LLL-invD-mod(9)[OF
Suc(2)] dmu''-def j by simp
         {
          fix k
          assume k: k < m
          then have d fs (Suc k) = d fs'' (Suc k)
            using ddμ LLL-invD-mod(9)[OF fs''invi] LLL-invD-mod(9)[OF
Suc(2)] dmu''-def j by simp
          have d fs 0 = 1 d fs'' 0 = 1 using d-def by auto
          moreover have sqid: sq-norm (gso fs'' k) = rat-of-int (d fs'' (Suc
k)) / rat-of-int (d fs'' k)
            using LLL-d-Suc[OF invw''] LLL-d-pos[OF invw''] k
          by (smt One-nat-def Suc-less-eq Suc-pred le-imp-less-Suc mult-eq-0-iff
less-imp-le-nat
            nonzero-mult-div-cancel-right of-int-0-less-iff of-int-hom.hom-zero)
           moreover have sq-norm (gso fs k) = rat-of-int (d fs (Suc k)) /
rat-of-int (d fs k)
            using LLL-d-Suc[OF invw] LLL-d-pos[OF invw] k
          by (smt One-nat-def Suc-less-eq Suc-pred le-imp-less-Suc mult-eq-0-iff
less-imp-le-nat
            nonzero-mult-div-cancel-right of-int-0-less-iff of-int-hom.hom-zero)
           ultimately have sq-norm (gso fs k) = sq-norm (gso fs'' k) using
k deq
             LLL-d-pos[OF invw] LLL-d-pos[OF invw'']
            by (metis (no-types, lifting) Nat.lessE Suc-lessD old.nat.inject
zero-less-Suc)
         }
          then show ?thesis using LLL-invD-mod(6)[OF Suc(2)] by (simp
add: gram-schmidt-fs.weakly-reduced-def l)
```

```
      next
        case False
      then show ?thesis using LLL-invD-mod(6)[OF fs''invi] gram-schmidt-fs.weakly-reduced-def
          by (metis less-or-eq-imp-le nat-neq-iff)
      qed
     }
     then show ?thesis by simp
    qed
   then have weakly-reduced fs'' m using gram-schmidt-fs.weakly-reduced-def
by blast
    then show ?thesis using LLL-invD-mod[OF fs''invi] LLL-invI-mod by
simp
  qed
  then show ?case using 01 Suc.hyps i j less-imp-le-nat μsm μs by metis
 qed
}
then show ?thesis using μ-small-row-refl res by auto
qed

lemma basis-reduction-mod-add-rows-outer-loop-inv:
 assumes inv: LLL-invariant-mod fs mfs dmu p first b m
 and (mfs', dmu') = basis-reduction-mod-add-rows-outer-loop p mfs dmu i
 and i: i < m
shows (∃fs'. LLL-invariant-mod fs' mfs' dmu' p first b m ∧
 (∀j. j ≤ i ⟶ μ-small fs' j))
proof(insert assms, induct i arbitrary: fs mfs dmu mfs' dmu')
 case (0 fs)
 then show ?case using μ-small-def by auto
next
 case (Suc i fs mfs dmu mfs' dmu')
 obtain mfs'' dmu'' where mfs''dmu'': (mfs'', dmu'')
  = basis-reduction-mod-add-rows-outer-loop p mfs dmu i by (metis surj-pair)
 then obtain fs'' where fs'': LLL-invariant-mod fs'' mfs'' dmu'' p first b m
  and 00: (∀j. j ≤ i ⟶ μ-small fs'' j) using Suc by fastforce
 have (mfs', dmu') = basis-reduction-mod-add-rows-loop p mfs'' dmu'' (Suc i)
(Suc i)
  using Suc(3,4) mfs''dmu'' by (smt basis-reduction-mod-add-rows-outer-loop.simps(2)
case-prod-conv)
 then obtain fs' where 01: LLL-invariant-mod fs' mfs' dmu' p first b m
  and 02: ∀i' j'. i' < (Suc i) ⟶ j' ≤ i' ⟶ μ fs'' i' j' = μ fs' i' j' and 03:
μ-small fs' (Suc i)
  using fs'' basis-reduction-mod-add-rows-loop-inv' Suc by metis
 moreover have ∀j. j ≤ (Suc i) ⟶ μ-small fs' j using 02 00 03 μ-small-def
by (simp add: le-Suc-eq)
 ultimately show ?case by blast
qed

lemma basis-reduction-mod-fs-bound:
 assumes Linv: LLL-invariant-mod fs mfs dmu p first b k
```

```
  and mu-small: μ-small fs i
  and i: i < m
  and nFirst: ¬ first
shows fs ! i = mfs ! i
proof −
  from LLL-invD-mod(16−17)[OF Linv] nFirst g-bnd-mode-def
  have gbnd: g-bnd b fs and bp: b ≤ (rat-of-int (p − 1))² / (rat-of-nat m + 3)
    by (auto simp: mod-invariant-def bound-number-def)
 have Linvw: LLL-invariant-weak' k fs using LLL-invD-mod[OF Linv] LLL-invI-weak
by simp
 have fs-int-indpt n fs using LLL-invD-mod(5)[OF Linv] Gram-Schmidt-2.fs-int-indpt.intro
by simp
  then interpret fs: fs-int-indpt n fs
    using fs-int-indpt.sq-norm-fs-via-sum-mu-gso by simp
  have ‖gso fs 0‖² ≤ b using gbnd i unfolding g-bnd-def by blast
  then have b0: 0 ≤ b using sq-norm-vec-ge-0 dual-order.trans by auto
  have 00: of-int ‖fs ! i‖² = (∑ j←[0..<Suc i]. (μ fs i j)² * ‖gso fs j‖²)
   using fs.sq-norm-fs-via-sum-mu-gso LLL-invD-mod[OF Linv] Gram-Schmidt-2.fs-int-indpt.intro
i by simp
  have 01: ∀ j < i. (μ fs i j)² * ‖gso fs j‖² ≤ (1 / rat-of-int 4) * ‖gso fs j‖²
  proof −
    {
      fix j
      assume j: j < i
      then have |fs.gs.μ i j| ≤ 1 / (rat-of-int 2)
         using mu-small Power.linordered-idom-class.abs-square-le-1 j unfolding
μ-small-def by simp
      moreover have |μ fs i j| ≥ 0 by simp
      ultimately have |μ fs i j|² ≤ (1 / rat-of-int 2)²
        using Power.linordered-idom-class.abs-le-square-iff by fastforce
      also have ... = 1 / (rat-of-int 4) by (simp add: field-simps)
      finally have |μ fs i j|² ≤ 1 / rat-of-int 4 by simp
    }
    then show ?thesis using fs.gs.μ.simps by (metis mult-right-mono power2-abs
sq-norm-vec-ge-0)
  qed
  then have 0111: ⋀j. j ∈ set [0..<i] ⟹ (μ fs i j)² * ‖gso fs j‖² ≤ (1 / rat-of-int
4) * ‖gso fs j‖²
    by simp
  {
    fix j
    assume j: j < n
    have 011: (μ fs i i)² * ‖gso fs i‖² = 1 * ‖gso fs i‖²
      using fs.gs.μ.simps by simp
    have 02: ∀ j < Suc i. ‖gso fs j‖² ≤ b
      using gbnd i unfolding g-bnd-def by simp
    have 03: length [0..<Suc i] = (Suc i) by simp
    have of-int ‖fs ! i‖² = (∑ j←[0..<i]. (μ fs i j)² * ‖gso fs j‖²) + ‖gso fs i‖²
      unfolding 00 using 011 by simp
```

also have $(\sum j{\leftarrow}[0..{<}i].\ (\mu\ fs\ i\ j)^2 * \|gso\ fs\ j\|^2) \leq (\sum j{\leftarrow}[0..{<}i].\ ((1\ /\ rat\text{-}of\text{-}int\ 4) * \|gso\ fs\ j\|^2))$
using *Groups-List.sum-list-mono*[*OF 0111*] **by** *fast*
finally have *of-int* $\|fs\ !\ i\|^2 \leq (\sum j{\leftarrow}[0..{<}i].\ ((1\ /\ rat\text{-}of\text{-}int\ 4) * \|gso\ fs\ j\|^2)) + \|gso\ fs\ i\|^2$
by *simp*
also have $(\sum j{\leftarrow}[0..{<}i].\ ((1\ /\ rat\text{-}of\text{-}int\ 4) * \|gso\ fs\ j\|^2)) \leq (\sum j{\leftarrow}[0..{<}i].\ (1\ /\ rat\text{-}of\text{-}int\ 4) * b)$
by (*intro sum-list-mono*, *insert 02*, *auto*)
also have $\|gso\ fs\ i\|^2 \leq b$ **using** *02* **by** *simp*
finally have *of-int* $\|fs\ !\ i\|^2 \leq (\sum j{\leftarrow}[0..{<}i].\ (1\ /\ rat\text{-}of\text{-}int\ 4) * b) + b$ **by** *simp*
also have $\ldots = (rat\text{-}of\text{-}nat\ i) * ((1\ /\ rat\text{-}of\text{-}int\ 4) *\ b) + b$
using *03 sum-list-triv*[*of* $(1\ /\ rat\text{-}of\text{-}int\ 4) * b\ [0..{<}i]$] **by** *simp*
also have $\ldots = (rat\text{-}of\text{-}nat\ i)\ /\ 4 * b + b$ **by** *simp*
also have $\ldots = ((rat\text{-}of\text{-}nat\ i)\ /\ 4 + 1) * b$ **by** *algebra*
also have $\ldots = (rat\text{-}of\text{-}nat\ i + 4)\ /\ 4 * b$ **by** *simp*
finally have *of-int* $\|fs\ !\ i\|^2 \leq (rat\text{-}of\text{-}nat\ i + 4)\ /\ 4 * b$ **by** *simp*
also have $\ldots \leq (rat\text{-}of\text{-}nat\ (m + 3))\ /\ 4 * b$ **using** *i b0 times-left-mono* **by** *fastforce*
finally have *of-int* $\|fs\ !\ i\|^2 \leq rat\text{-}of\text{-}nat\ (m{+}3)\ /\ 4 * b$ **by** *simp*
moreover have $|fs\ !\ i\ \$\ j|^2 \leq \|fs\ !\ i\|^2$ **using** *vec-le-sq-norm LLL-invD-mod*(*10*)[*OF Linv*] *i j* **by** *blast*
ultimately have *04*: *of-int* $(|fs\ !\ i\ \$\ j|^2) \leq rat\text{-}of\text{-}nat\ (m{+}3)\ /\ 4 * b$ **using** *ge-trans i* **by** *linarith*
then have *05*: *real-of-int* $(|fs\ !\ i\ \$\ j|^2) \leq real\text{-}of\text{-}rat\ (rat\text{-}of\text{-}nat\ (m{+}3)\ /\ 4 * b)$
proof −
from *j* **have** *rat-of-int* $(|fs\ !\ i\ \$\ j|^2) \leq rat\text{-}of\text{-}nat\ (m{+}3)\ /\ 4 * b$ **using** *04* **by** *simp*
then have *real-of-int* $(|fs\ !\ i\ \$\ j|^2) \leq real\text{-}of\text{-}rat\ (rat\text{-}of\text{-}nat\ (m{+}3)\ /\ 4 * b)$
using *j of-rat-less-eq* **by** (*metis of-rat-of-int-eq*)
then show *?thesis* **by** *simp*
qed
define *rhs* **where** $rhs = real\text{-}of\text{-}rat\ (rat\text{-}of\text{-}nat\ (m{+}3)\ /\ 4 * b)$
have *rhs0*: $rhs \geq 0$ **using** *b0 i rhs-def* **by** *simp*
have *fsij*: *real-of-int* $|fs\ !\ i\ \$\ j| \geq 0$ **by** *simp*
have *real-of-int* $(|fs\ !\ i\ \$\ j|^2) = (real\text{-}of\text{-}int\ |fs\ !\ i\ \$\ j|)^2$ **by** *simp*
then have $(real\text{-}of\text{-}int\ |fs\ !\ i\ \$\ j|)^2 \leq rhs$ **using** *05 j rhs-def* **by** *simp*
then have *g1*: *real-of-int* $|fs\ !\ i\ \$\ j| \leq sqrt\ rhs$ **using** *NthRoot.real-le-rsqrt* **by** *simp*
have *pbnd*: $2 * |fs\ !\ i\ \$\ j| < p$
proof −
have *rat-of-nat* $(m{+}3)\ /\ 4 * b \leq (rat\text{-}of\text{-}nat\ (m\ {+}3)\ /\ 4) * (rat\text{-}of\text{-}int\ (p - 1))^2\ /\ (rat\text{-}of\text{-}nat\ m{+}3)$
using *bp b0 i times-left-mono SN-Orders.of-nat-ge-zero gs.m-comm times-divide-eq-right*
by (*smt gs.l-null le-divide-eq-numeral1*(*1*))
also have $\ldots = (rat\text{-}of\text{-}int\ (p - 1))^2\ /\ 4 * (rat\text{-}of\text{-}nat\ (m + 3)\ /\ rat\text{-}of\text{-}nat$

```
(m + 3))
    by (metis (no-types, lifting) gs.m-comm of-nat-add of-nat-numeral times-divide-eq-left)
    finally have rat-of-nat (m+3) / 4 * b ≤ (rat-of-int (p - 1))^2 / 4 by simp
    then have sqrt rhs ≤ sqrt (real-of-rat ((rat-of-int (p - 1))^2 / 4))
      unfolding rhs-def using of-rat-less-eq by fastforce
    then have two-ineq:
      2 * |fs ! i $ j| ≤ 2 * sqrt (real-of-rat ((rat-of-int (p - 1))^2 / 4))
      using g1 by linarith
    have 2 * sqrt (real-of-rat ((rat-of-int (p - 1))^2 / 4)) =
    sqrt (real-of-rat (4 * ((rat-of-int (p - 1))^2 / 4)))
    by (metis (no-types, opaque-lifting) real-sqrt-mult of-int-numeral of-rat-hom.hom-mult

        of-rat-of-int-eq real-sqrt-four times-divide-eq-right)
    also have ... = sqrt (real-of-rat ((rat-of-int (p - 1))^2)) using i by simp
    also have (real-of-rat ((rat-of-int (p - 1))^2)) = (real-of-rat (rat-of-int (p -
1)))^2
      using Rat.of-rat-power by blast
    also have sqrt ((real-of-rat (rat-of-int (p - 1)))^2) = real-of-rat (rat-of-int (p
- 1))
      using LLL-invD-mod(15)[OF Linv] by simp
    finally have 2 * sqrt (real-of-rat ((rat-of-int (p - 1))^2 / 4)) =
    real-of-rat (rat-of-int (p - 1)) by simp
    then have 2 * |fs ! i $ j| ≤ real-of-rat (rat-of-int (p - 1))
      using two-ineq by simp
    then show ?thesis by (metis of-int-le-iff of-rat-of-int-eq zle-diff1-eq)
  qed
  have p1: p > 1 using LLL-invD-mod[OF Linv] by blast
  interpret pm: poly-mod-2 p
    by (unfold-locales, rule p1)
    from LLL-invD-mod[OF Linv] have len: length fs = m and fs: set fs ⊆
carrier-vec n by auto
   from pm.inv-M-rev[OF pbnd, unfolded pm.M-def] have pm.inv-M (fs ! i $ j
mod p) = fs ! i $ j .
  also have pm.inv-M (fs ! i $ j mod p) = mfs ! i $ j unfolding LLL-invD-mod(7)[OF
Linv, symmetric] sym-mod-def
    using i j len fs by auto
  finally have fs ! i $ j = mfs ! i $ j ..
 }
 thus fs ! i = mfs ! i using LLL-invD-mod(10,13)[OF Linv i] by auto
qed

lemma basis-reduction-mod-fs-bound-first:
 assumes Linv: LLL-invariant-mod fs mfs dmu p first b k
 and m0: m > 0
 and first: first
shows fs ! 0 = mfs ! 0
proof -
 from LLL-invD-mod(16-17)[OF Linv] first g-bnd-mode-def m0
 have gbnd: sq-norm (gso fs 0) ≤ b and bp: b ≤ (rat-of-int (p - 1))^2 / 4
```

```
    by (auto simp: mod-invariant-def bound-number-def)
  from LLL-invD-mod[OF Linv] have p1: p > 1 by blast
  have Linvw: LLL-invariant-weak' k fs using LLL-invD-mod[OF Linv] LLL-invI-weak
by simp
  have fs-int-indpt n fs using LLL-invD-mod(5)[OF Linv] Gram-Schmidt-2.fs-int-indpt.intro
by simp
  then interpret fs: fs-int-indpt n fs
    using fs-int-indpt.sq-norm-fs-via-sum-mu-gso by simp
  from gbnd have b0: 0 ≤ b using sq-norm-vec-ge-0 dual-order.trans by auto
  have of-int ‖fs ! 0‖² = (μ fs 0 0)² * ‖gso fs 0‖²
   using fs.sq-norm-fs-via-sum-mu-gso LLL-invD-mod[OF Linv] Gram-Schmidt-2.fs-int-indpt.intro
m0 by simp
  also have ... = ‖gso fs 0‖² unfolding fs.gs.μ.simps by (simp add: gs.μ.simps)
  also have ... ≤ (rat-of-int (p − 1))² / 4 using gbnd bp by auto
  finally have one: of-int (sq-norm (fs ! 0)) ≤ (rat-of-int (p − 1))² / 4 .
  {
    fix j
    assume j: j < n
    have leq: |fs ! 0 $ j|² ≤ ‖fs ! 0‖² using vec-le-sq-norm LLL-invD-mod(10)[OF
Linv] m0 j by blast
    have rat-of-int ((2 * |fs ! 0 $ j|)^2) = rat-of-int (4 * |fs ! 0 $ j|²) by simp
    also have ... ≤ 4 * of-int ‖fs ! 0‖² using leq by simp
    also have ... ≤ 4 * (rat-of-int (p − 1))² / 4 using one by simp
    also have ... = (rat-of-int (p − 1))² by simp
    also have ... = rat-of-int ((p − 1)²) by simp
    finally have (2 * |fs ! 0 $ j|)^2 ≤ (p − 1)² by linarith
    hence 2 * |fs ! 0 $ j| ≤ p − 1 using p1
      by (smt power-mono-iff zero-less-numeral)
    hence pbnd: 2 * |fs ! 0 $ j| < p by simp
    interpret pm: poly-mod-2 p
      by (unfold-locales, rule p1)
    from LLL-invD-mod[OF Linv] m0 have len: length fs = m length mfs = m
      and fs: fs ! 0 ∈ carrier-vec n mfs ! 0 ∈ carrier-vec n by auto
    from pm.inv-M-rev[OF pbnd, unfolded pm.M-def] have pm.inv-M (fs ! 0 $ j
mod p) = fs ! 0 $ j .
   also have pm.inv-M (fs ! 0 $ j mod p) = mfs ! 0 $ j unfolding LLL-invD-mod(7)[OF
Linv, symmetric] sym-mod-def
      using m0 j len fs by auto
    finally have mfs ! 0 $ j = fs ! 0 $ j .
  }
  thus fs ! 0 = mfs ! 0 using LLL-invD-mod(10,13)[OF Linv m0] by auto
qed

lemma dmu-initial: dmu-initial = mat m m (λ (i,j). dμ fs-init i j)
proof −
  interpret fs: fs-int-indpt n fs-init
    by (unfold-locales, intro lin-dep)
  show ?thesis unfolding dmu-initial-def Let-def
  proof (intro cong-mat refl refl, unfold split, goal-cases)
```

case (*1 i j*)
show *?case*
proof (*cases* $j \leq i$)
case *False*
thus *?thesis* **by** (*auto simp*: *dμ-def gs.μ.simps*)
next
case *True*
hence *id*: *dμ-impl fs-init* !! *i* !! *j* = *fs.dμ i j* **unfolding** *fs.dμ-impl*
by (*subst of-fun-nth*, *use 1 len* **in** *force*, *subst of-fun-nth*, *insert True*, *auto*)
also have ... = *dμ fs-init i j* **unfolding** *fs.dμ-def dμ-def fs.d-def d-def* **by** *simp*
finally show *?thesis* **using** *True* **by** *auto*
qed
qed
qed

lemma *LLL-initial-invariant-mod*: **assumes** *res*: *compute-initial-state first* = (*p*, *mfs*, *dmu'*, *g-idx*)
shows ∃*fs b*. *LLL-invariant-mod fs mfs dmu' p first b 0*
proof −
from *dmu-initial* **have** *dmu*: (∀ $i' < m$. ∀ $j' < m$. *dμ fs-init i' j'* = *dmu-initial* $$ (*i'*,*j'*)) **by** *auto*
obtain *b g-idx* **where** *norm*: *compute-max-gso-norm first dmu-initial* = (*b*,*g-idx*) **by** *force*
note *res* = *res*[*unfolded compute-initial-state-def Let-def norm split*]
from *res* **have** *p*: *p* = *compute-mod-of-max-gso-norm first b* **by** *auto*
then have *p0*: *p* > *0* **unfolding** *compute-mod-of-max-gso-norm-def* **using** *log-base* **by** *simp*
then have *p1*: $p \geq 1$ **by** *simp*
note *res* = *res*[*folded p*]
from *res*[*unfolded compute-initial-mfs-def*]
have *mfs*: *mfs* = *map* (*map-vec* (λ*x*. *x symmod p*)) *fs-init* **by** *auto*
from *res*[*unfolded compute-initial-dmu-def*]
have *dmu'*: *dmu'* = *mat m m* (λ(*i'*,*j'*). *if j'* < *i'*
then dmu-initial $$ (*i'*, *j'*) *symmod* (*p* ∗ *d-of dmu-initial j'* ∗ *d-of dmu-initial* (*Suc j'*))
else dmu-initial $$ (*i'*,*j'*)) **by** *auto*
have *lat*: *lattice-of fs-init* = *L* **by** (*auto simp*: *L-def*)
define *I* **where** *I* = {(*i'*,*j'*). $i' < m \wedge j' < i'$}
obtain *fs* **where**
01: *lattice-of fs* = *L* **and**
02: *map* (*map-vec* (λ *x*. *x symmod p*)) *fs* = *map* (*map-vec* (λ *x*. *x symmod p*)) *fs-init* **and**
03: *lin-indep fs* **and**
04: *length fs* = *m* **and**
05: (∀ $k < m$. *gso fs k* = *gso fs-init k*) **and**
06: (∀ $k \leq m$. *d fs k* = *d fs-init k*) **and**
07: (∀ $i' < m$. ∀ $j' < m$. *dμ fs i' j'* =
(*if* (*i'*,*j'*) ∈ *I then dμ fs-init i' j' symmod* (*p* ∗ *d fs-init j'* ∗ *d fs-init* (*Suc j'*))

else dμ fs-init i′ j′))
using *mod-finite-set*[*OF lin-dep len - lat p0*, *of I*] *I-def* **by** *blast*
have *inv*: *LLL-invariant-weak fs-init*
by (*intro LLL-inv-wI lat len lin-dep fs-init*)
have ∀ *i′*<*m*. *dμ fs-init i′ i′* = *dmu-initial* $$ (*i′*, *i′*) **unfolding** *dmu-initial* **by** *auto*
from *compute-max-gso-norm*[*OF this inv*, *of first*, *unfolded norm*] **have** *gbnd*: *g-bnd-mode first b fs-init*
and *b0*: *0* ≤ *b* **and** *mb0*: *m* = *0* ⟹ *b* = *0* **by** *auto*
from *gbnd 05* **have** *gbnd*: *g-bnd-mode first b fs* **using** *g-bnd-mode-cong*[*of fs fs-init*] **by** *auto*
have *dμdmu′*: ∀ *i′*<*m*. ∀ *j′*<*m*. *dμ fs i′ j′* = *dmu′* $$ (*i′*, *j′*) **using** *07 dmu d-of-main*[*of fs-init dmu-initial*]
unfolding *I-def dmu′* **by** *simp*
have *wred*: *weakly-reduced fs 0* **by** (*simp add*: *gram-schmidt-fs.weakly-reduced-def*)
have *fs-carr*: *set fs* ⊆ *carrier-vec n* **using** *03* **unfolding** *gs.lin-indpt-list-def* **by** *force*
have *m0*: *m* ≥ *0* **using** *len* **by** *auto*
have *Linv*: *LLL-invariant-weak′ 0 fs*
by (*intro LLL-invI-weak 03 04 01 wred fs-carr m0*)
note *Linvw* = *LLL-invw′-imp-w*[*OF Linv*]
from *compute-mod-of-max-gso-norm*[*OF b0 mb0 p*]
have *p*: *mod-invariant b p first p* > *1* **by** *auto*
from *len mfs* **have** *len′*: *length mfs* = *m* **by** *auto*
have *modbnd*: ∀ *i′*<*m*. ∀ *j′*<*i′*. |*dμ fs i′ j′*| < *p* ∗ *d fs j′* ∗ *d fs* (*Suc j′*)
proof −
have ∀ *i′* < *m*. ∀ *j′* < *i′*. *dμ fs i′ j′* = *dμ fs i′ j′ symmod* (*p* ∗ *d fs j′* ∗ *d fs* (*Suc j′*))
using *I-def 07 06* **by** *simp*
moreover have ∀ *j′* < *m*. *p* ∗ *d fs j′* ∗ *d fs* (*Suc j′*) > *0* **using** *p*(*2*) *LLL-d-pos*[*OF Linvw*] **by** *simp*
ultimately show *?thesis* **using** *sym-mod-abs*
by *auto* (*smt* (*verit*, *del-insts*) *Suc-lessD less-trans-Suc*)
qed
have *LLL-invariant-mod fs mfs dmu′ p first b 0*
using *LLL-invI-mod*[*OF len′ m0 04 01 03 wred - modbnd dμdmu′ p*(*2*) *gbnd p*(*1*)] *02 mfs* **by** *simp*
then show *?thesis* **by** *auto*
qed

4.3 Soundness of Storjohann's algorithm

For all of these abstract algorithms, we actually formulate their soundness proofs by linking to the LLL-invariant (which implies that *fs* is reduced (*LLL-invariant True m fs*) or that the first vector of *fs* is short (*LLL-invariant-weak fs* ∧ *gram-schmidt-fs.weakly-reduced n* (*map of-int-hom.vec-hom fs*) *α m*).

Soundness of Storjohann's algorithm

lemma *reduce-basis-mod-inv*: **assumes** *res*: *reduce-basis-mod* = *fs*
shows *LLL-invariant True m fs*
proof (*cases m* = *0*)
case *True*
from *True* **have** ∗: *fs-init* = [] **using** *len* **by** *simp*
moreover have *fs* = [] **using** *res basis-reduction-mod-add-rows-outer-loop.simps*(*1*)
unfolding *reduce-basis-mod-def Let-def basis-reduction-mod-main.simps*[*of - - - - - 0*]
compute-initial-mfs-def compute-initial-state-def compute-initial-dmu-def
unfolding *True* ∗ **by** (*auto split*: *prod.splits*)
ultimately show *?thesis* **using** *True LLL-inv-initial-state* **by** *blast*
next
case *False*
let *?first* = *False*
obtain *p mfs0 dmu0 g-idx0* **where** *init*: *compute-initial-state ?first* = (*p*, *mfs0*, *dmu0*, *g-idx0*) **by** (*metis prod-cases4*)
from *LLL-initial-invariant-mod*[*OF init*]
obtain *fs0 b* **where** *fs0*: *LLL-invariant-mod fs0 mfs0 dmu0 p ?first b 0* **by** *blast*
note *res* = *res*[*unfolded reduce-basis-mod-def init Let-def split*]
obtain *p1 mfs1 dmu1* **where** *mfs1dmu1*: (*p1*, *mfs1*, *dmu1*) = *basis-reduction-mod-main p ?first mfs0 dmu0 g-idx0 0 0*
by (*metis prod.exhaust*)
obtain *fs1 b1* **where** *Linv1*: *LLL-invariant-mod fs1 mfs1 dmu1 p1 ?first b1 m*
using *basis-reduction-mod-main*[*OF fs0 mfs1dmu1*[*symmetric*]] **by** *auto*
obtain *mfs2 dmu2* **where** *mfs2dmu2*:
(*mfs2*, *dmu2*) = *basis-reduction-mod-add-rows-outer-loop p1 mfs1 dmu1* (*m*−*1*)
by (*metis old.prod.exhaust*)
obtain *fs2* **where** *fs2*: *LLL-invariant-mod fs2 mfs2 dmu2 p1 ?first b1 m*
and *μs*: ((∀ *j*. *j* < *m* ⟶ *μ-small fs2 j*))
using *basis-reduction-mod-add-rows-outer-loop-inv*[*OF - mfs2dmu2*, *of fs1 ?first b1*] *Linv1 False* **by** *auto*
have *rbd*: *LLL-invariant-weak′ m fs2* ∀ *j* < *m*. *μ-small fs2 j*
using *LLL-invD-mod*[*OF fs2*] *LLL-invI-weak μs* **by** *auto*
have *redfs2*: *reduced fs2 m* **using** *rbd LLL-invD-weak*(*8*) *gram-schmidt-fs.reduced-def μ-small-def* **by** *blast*
have *fs*: *fs* = *mfs2*
using *res*[*folded mfs1dmu1*, *unfolded Let-def split*, *folded mfs2dmu2*, *unfolded split*] **..**
have ∀ *i* < *m*. *fs2* ! *i* = *fs* ! *i*
proof (*intro allI impI*)
fix *i*
assume *i*: *i* < *m*
then have *fs2i*: *LLL-invariant-mod fs2 mfs2 dmu2 p1 ?first b1 i*
using *fs2 LLL-invariant-mod-to-weak-m-to-i* **by** *simp*
have *μsi*: *μ-small fs2 i* **using** *μs i* **by** *simp*
show *fs2* ! *i* = *fs* ! *i*
using *basis-reduction-mod-fs-bound*(*1*)[*OF fs2i μsi i*] *fs* **by** *simp*
qed
then have *fs2* = *fs*

```
    using LLL-invD-mod(1,3,10,13)[OF fs2] fs by (metis nth-equalityI)
  then show ?thesis using redfs2 fs rbd(1) reduce-basis-def res LLL-invD-weak
      LLL-invariant-def by simp
qed
```

Soundness of Storjohann's algorithm for computing a short vector.

```
lemma short-vector-mod-inv: assumes res: short-vector-mod = v
  and m: m > 0
  shows ∃ fs. LLL-invariant-weak fs ∧ weakly-reduced fs m ∧ v = hd fs
proof −
  let ?first = True
  obtain p mfs0 dmu0 g-idx0 where init: compute-initial-state ?first = (p, mfs0,
dmu0, g-idx0) by (metis prod-cases4)
  from LLL-initial-invariant-mod[OF init]
  obtain fs0 b where fs0: LLL-invariant-mod fs0 mfs0 dmu0 p ?first b 0 by blast
  obtain p1 mfs1 dmu1 where main: basis-reduction-mod-main p ?first mfs0 dmu0
g-idx0 0 0 = (p1, mfs1, dmu1)
    by (metis prod.exhaust)
  obtain fs1 b1 where Linv1: LLL-invariant-mod fs1 mfs1 dmu1 p1 ?first b1 m
    using basis-reduction-mod-main[OF fs0 main] by auto
  have v = hd mfs1 using res[unfolded short-vector-mod-def Let-def init split main]
..
    with basis-reduction-mod-fs-bound-first[OF Linv1 m] LLL-invD-mod(1,3)[OF
Linv1] m
  have v: v = hd fs1 by (cases fs1; cases mfs1; auto)
  from Linv1 have Linv1: LLL-invariant-weak fs1 and red: weakly-reduced fs1 m

    unfolding LLL-invariant-mod-def LLL-invariant-weak-def by auto
  show ?thesis
    by (intro exI[of - fs1] conjI Linv1 red v)
qed
```

Soundness of Storjohann's algorithm with improved swap order

```
lemma reduce-basis-iso-inv: assumes res: reduce-basis-iso = fs
  shows LLL-invariant True m fs
proof (cases m = 0)
  case True
  then have ∗: fs-init = [] using len by simp
 moreover have fs = [] using res basis-reduction-mod-add-rows-outer-loop.simps(1)
    unfolding reduce-basis-iso-def Let-def basis-reduction-iso-main.simps[of - - - -
- 0]
      compute-initial-mfs-def compute-initial-state-def compute-initial-dmu-def
    unfolding True ∗ by (auto split: prod.splits)
  ultimately show ?thesis using True LLL-inv-initial-state by blast
next
  case False
  let ?first = False
  obtain p mfs0 dmu0 g-idx0 where init: compute-initial-state ?first = (p, mfs0,
dmu0, g-idx0) by (metis prod-cases4)
```

from *LLL-initial-invariant-mod*[*OF init*]
obtain *fs0 b* **where** *fs0*: *LLL-invariant-mod fs0 mfs0 dmu0 p ?first b 0* **by** *blast*
have *fs0w*: *LLL-invariant-mod-weak fs0 mfs0 dmu0 p ?first b* **using** *LLL-invD-mod*[*OF fs0*] *LLL-invI-modw* **by** *simp*
note *res* = *res*[*unfolded reduce-basis-iso-def init Let-def split*]
obtain *p1 mfs1 dmu1* **where** *mfs1dmu1*: (*p1*, *mfs1*, *dmu1*) = *basis-reduction-iso-main p ?first mfs0 dmu0 g-idx0 0*
by (*metis prod.exhaust*)
obtain *fs1 b1* **where** *Linv1*: *LLL-invariant-mod fs1 mfs1 dmu1 p1 ?first b1 m*
using *basis-reduction-iso-main*[*OF fs0w mfs1dmu1*[*symmetric*]] **by** *auto*
obtain *mfs2 dmu2* **where** *mfs2dmu2*:
(*mfs2*, *dmu2*) = *basis-reduction-mod-add-rows-outer-loop p1 mfs1 dmu1* (*m*−*1*)
by (*metis old.prod.exhaust*)
obtain *fs2* **where** *fs2*: *LLL-invariant-mod fs2 mfs2 dmu2 p1 ?first b1 m*
and *μs*: ((∀ *j*. *j* < *m* ⟶ *μ-small fs2 j*))
using *basis-reduction-mod-add-rows-outer-loop-inv*[*OF - mfs2dmu2*, *of fs1 ?first b1*] *Linv1 False* **by** *auto*
have *rbd*: *LLL-invariant-weak′ m fs2* ∀ *j* < *m*. *μ-small fs2 j*
using *LLL-invD-mod*[*OF fs2*] *LLL-invI-weak μs* **by** *auto*
have *redfs2*: *reduced fs2 m* **using** *rbd LLL-invD-weak*(*8*) *gram-schmidt-fs.reduced-def μ-small-def* **by** *blast*
have *fs*: *fs* = *mfs2*
using *res*[*folded mfs1dmu1*, *unfolded Let-def split*, *folded mfs2dmu2*, *unfolded split*] **..**
have ∀ *i* < *m*. *fs2* ! *i* = *fs* ! *i*
proof (*intro allI impI*)
fix *i*
assume *i*: *i* < *m*
then have *fs2i*: *LLL-invariant-mod fs2 mfs2 dmu2 p1 ?first b1 i*
using *fs2 LLL-invariant-mod-to-weak-m-to-i* **by** *simp*
have *μsi*: *μ-small fs2 i* **using** *μs i* **by** *simp*
show *fs2* ! *i* = *fs* ! *i*
using *basis-reduction-mod-fs-bound*(*1*)[*OF fs2i μsi i*] *fs* **by** *simp*
qed
then have *fs2* = *fs*
using *LLL-invD-mod*(*1*,*3*,*10*,*13*)[*OF fs2*] *fs* **by** (*metis nth-equalityI*)
then show *?thesis* **using** *redfs2 fs rbd*(*1*) *reduce-basis-def res LLL-invD-weak LLL-invariant-def* **by** *simp*
qed

Soundness of Storjohann's algorithm to compute short vectors with improved swap order

lemma *short-vector-iso-inv*: **assumes** *res*: *short-vector-iso* = *v*
and *m*: *m* > *0*
shows ∃ *fs*. *LLL-invariant-weak fs* ∧ *weakly-reduced fs m* ∧ *v* = *hd fs*
proof −
let *?first* = *True*
obtain *p mfs0 dmu0 g-idx0* **where** *init*: *compute-initial-state ?first* = (*p*, *mfs0*, *dmu0*, *g-idx0*) **by** (*metis prod-cases4*)

from *LLL-initial-invariant-mod*[*OF init*]
obtain *fs0 b* **where** *fs0*: *LLL-invariant-mod fs0 mfs0 dmu0 p ?first b 0* **by** *blast*
have *fs0w*: *LLL-invariant-mod-weak fs0 mfs0 dmu0 p ?first b* **using** *LLL-invD-mod*[*OF fs0*] *LLL-invI-modw* **by** *simp*
obtain *p1 mfs1 dmu1* **where** *main*: *basis-reduction-iso-main p ?first mfs0 dmu0 g-idx0 0* = (*p1*, *mfs1*, *dmu1*)
by (*metis prod.exhaust*)
obtain *fs1 b1* **where** *Linv1*: *LLL-invariant-mod fs1 mfs1 dmu1 p1 ?first b1 m*
using *basis-reduction-iso-main*[*OF fs0w main*] **by** *auto*
have *v* = *hd mfs1* **using** *res*[*unfolded short-vector-iso-def Let-def init split main*]
..
with *basis-reduction-mod-fs-bound-first*[*OF Linv1 m*] *LLL-invD-mod*(*1*,*3*)[*OF Linv1*] *m*
have *v*: *v* = *hd fs1* **by** (*cases fs1*; *cases mfs1*; *auto*)
from *Linv1* **have** *Linv1*: *LLL-invariant-weak fs1* **and** *red*: *weakly-reduced fs1 m*

unfolding *LLL-invariant-mod-def LLL-invariant-weak-def* **by** *auto*
show *?thesis*
by (*intro exI*[*of - fs1*] *conjI Linv1 red v*)
qed

end

From the soundness results of these abstract versions of the algorithms, one just needs to derive actual implementations that may integrate low-level optimizations.

end

5 Storjohann's basis reduction algorithm (concrete implementation)

We refine the abstract algorithm into a more efficient executable one.

theory *Storjohann-Impl*
imports
Storjohann
begin

5.1 Implementation

We basically store four components:

- The f-basis (as list, all values taken modulo p)
- The $d\mu$-matrix (as nested arrays, all values taken modulo $d_i d_{i+1} p$)
- The d-values (as array)

- The modulo-values $d_i d_{i+1} p$ (as array)

type-synonym *state-impl* = *int vec list* × *int iarray iarray* × *int iarray* × *int iarray*

fun *di-of* :: *state-impl* ⇒ *int iarray* **where**
di-of (*mfsi*, *dmui*, *di*, *mods*) = *di*

context *LLL*
begin

fun *state-impl-inv* :: - ⇒ - ⇒ - ⇒ *state-impl* ⇒ *bool* **where**
state-impl-inv p mfs dmu (*mfsi*, *dmui*, *di*, *mods*) = (*mfsi* = *mfs* ∧ *di* = *IArray.of-fun* (*d-of dmu*) (*Suc m*)
∧ *dmui* = *IArray.of-fun* (λ *i*. *IArray.of-fun* (λ *j*. *dmu* $$ (*i*,*j*)) *i*) *m*
∧ *mods* = *IArray.of-fun* (λ *j*. *p* ∗ *di* !! *j* ∗ *di* !! (*Suc j*)) (*m* − *1*))

definition *state-iso-inv* :: (*int* × *int*) *iarray* ⇒ *int iarray* ⇒ *bool* **where**
state-iso-inv prods di = (*prods* = *IArray.of-fun*
(λ *i*. (*di* !! (*i*+*1*) ∗ *di* !! (*i*+*1*), *di* !! (*i*+*2*) ∗ *di* !! *i*)) (*m* − *1*))

definition *perform-add-row* :: *int* ⇒ *state-impl* ⇒ *nat* ⇒ *nat* ⇒ *int* ⇒ *int iarray* ⇒ *int* ⇒ *int* ⇒ *state-impl* **where**
perform-add-row p state i j c rowi muij dij1 = (*let*
(*mfsi*, *dmui*, *di*, *mods*) = *state*;
fsj = *mfsi* ! *j*;
rowj = *dmui* !! *j*
in
(*case split-at i mfsi of* (*start*, *fsi* # *end*) ⇒ *start* @ *vec n* (λ *k*. (*fsi* $ *k* − *c* ∗ *fsj* $ *k*) *symmod p*) # *end*,
IArray.of-fun (λ *ii*. *if i* = *ii then*
IArray.of-fun (λ *jj*. *if jj* < *j then*
(*rowi* !! *jj* − *c* ∗ *rowj* !! *jj*) *symmod* (*mods* !! *jj*)
else if jj = *j then muij* − *c* ∗ *dij1*
else rowi !! *jj*) *i*
else dmui !! *ii*) *m*,
di, *mods*))

definition *LLL-add-row* :: *int* ⇒ *state-impl* ⇒ *nat* ⇒ *nat* ⇒ *state-impl* **where**
LLL-add-row p state i j = (*let*
(-, *dmui*, *di*, -) = *state*;
rowi = *dmui* !! *i*;
dij1 = *di* !! (*Suc j*);
muij = *rowi* !! *j*;
c = *round-num-denom muij dij1*
in if c = *0 then state*
else perform-add-row p state i j c rowi muij dij1)

definition *LLL-swap-row* :: *int* ⇒ *state-impl* ⇒ *nat* ⇒ *state-impl* **where**
LLL-swap-row p state k = (*case state of* (*mfsi*, *dmui*, *di*, *mods*) ⇒ *let*
k1 = *k* − *1*;
kS1 = *Suc k*;
muk = *dmui* !! *k*;
muk1 = *dmui* !! *k1*;
mukk1 = *muk* !! *k1*;
dk1 = *di* !! *k1*;
dkS1 = *di* !! *kS1*;
dk = *di* !! *k*;
dk′ = (*dkS1* ∗ *dk1* + *mukk1* ∗ *mukk1*) *div dk*;
mod1 = *p* ∗ *dk1* ∗ *dk′*;
modk = *p* ∗ *dk′* ∗ *dkS1*
in
(*case split-at k1 mfsi*
of (*start*, *fsk1* # *fsk* # *end*) ⇒ *start* @ *fsk* # *fsk1* # *end*,
IArray.of-fun (λ *i*.
if i < *k1 then dmui* !! *i*
else if i > *k then*
let row-i = *dmui* !! *i*; *muik* = *row-i* !! *k*; *muik1* = *row-i* !! *k1 in*
IArray.of-fun
(λ *j*. *if j* = *k1 then* ((*mukk1* ∗ *muik1* + *muik* ∗ *dk1*) *div dk*) *symmod*
mod1
else if j = *k then* ((*dkS1* ∗ *muik1* − *mukk1* ∗ *muik*) *div dk*)
symmod modk
else row-i !! *j*) *i*
else if i = *k then IArray.of-fun* (λ *j*. *if j* = *k1 then mukk1 symmod mod1*
else muk1 !! *j*) *i*
else IArray.of-fun ((!!) *muk*) *i*
) *m*,
IArray.of-fun (λ *i*. *if i* = *k then dk′ else di* !! *i*) (*Suc m*),
IArray.of-fun (λ *j*. *if j* = *k1 then mod1 else if j* = *k then modk else mods* !!
j) (*m* − *1*)))

definition *perform-swap-add* **where** *perform-swap-add p state k k1 c row-k mukk1*
dk =
(*let* (*fs*, *dmu*, *dd*, *mods*) = *state*;
row-k1 = *dmu* !! *k1*;
kS1 = *Suc k*;
mukk1′ = *mukk1* − *c* ∗ *dk*;
dk1 = *dd* !! *k1*;
dkS1 = *dd* !! *kS1*;
dk′ = (*dkS1* ∗ *dk1* + *mukk1′* ∗ *mukk1′*) *div dk*;
mod1 = *p* ∗ *dk1* ∗ *dk′*;
modk = *p* ∗ *dk′* ∗ *dkS1*
in
(*case split-at k1 fs of* (*start*, *fsk1* # *fsk* # *end*) ⇒
start @ *vec n* (λ*k*. (*fsk* $ *k* − *c* ∗ *fsk1* $ *k*) *symmod p*) # *fsk1* # *end*,
IArray.of-fun

```
      (λi. if i < k1
            then dmu !! i
            else if k < i
                 then let row-i = dmu !! i;
                          muik1 = row-i !! k1;
                          muik = row-i !! k
                      in IArray.of-fun
                          (λj. if j = k1 then (mukk1' * muik1 + muik * dk1) div dk
symmod mod1
                               else if j = k then (dkS1 * muik1 − mukk1' * muik) div
dk symmod modk
                                else row-i !! j)
                          i
                 else if i = k then IArray.of-fun (λj. if j = k1 then mukk1' symmod
mod1 else row-k1 !! j) k
                 else IArray.of-fun (λj. (row-k !! j − c * row-k1 !! j) symmod mods
!! j) i)
       m,
      IArray.of-fun (λi. if i = k then dk' else dd !! i) (Suc m),
      IArray.of-fun (λj. if j = k1 then mod1 else if j = k then modk else mods !! j)
(m − 1)))


definition LLL-swap-add where
  LLL-swap-add p state i = (let
    i1 = i − 1;
    (-, dmui, di, -) = state;
    rowi = dmui !! i;
    dii = di !! i;
    muij = rowi !! i1;
    c = round-num-denom muij dii
    in if c = 0 then LLL-swap-row p state i
     else perform-swap-add p state i i1 c rowi muij dii)

definition LLL-max-gso-norm-di :: bool ⇒ int iarray ⇒ rat × nat where
  LLL-max-gso-norm-di first di =
     (if first then (of-int (di !! 1), 0)
      else case max-list-rats-with-index (map (λ i. (di !! (Suc i), di !! i, i)) [0 ..<
m ])
     of (num, denom, i) ⇒ (of-int num / of-int denom, i))

definition LLL-max-gso-quot:: (int * int) iarray ⇒ (int * int * nat) where
  LLL-max-gso-quot di-prods = max-list-rats-with-index
    (map (λi. case di-prods !! i of (l,r) ⇒ (l, r, Suc i)) [0..<(m−1)])


definition LLL-max-gso-norm :: bool ⇒ state-impl ⇒ rat × nat where
 LLL-max-gso-norm first state = (case state of (-, -, di, mods) ⇒ LLL-max-gso-norm-di
first di)
```

definition *perform-adjust-mod* :: *int* ⇒ *state-impl* ⇒ *state-impl* **where**
perform-adjust-mod p state = (*case state of* (*mfsi*, *dmui*, *di*, -) ⇒
let mfsi′ = *map* (*map-vec* (λ*x*. *x symmod p*)) *mfsi*;
mods = *IArray.of-fun* (λ *j*. *p* ∗ *di* !! *j* ∗ *di* !! (*Suc j*)) (*m* − *1*);
dmui′ = *IArray.of-fun* (λ *i*. *let row* = *dmui* !! *i in IArray.of-fun* (λ *j*. *row* !! *j symmod* (*mods* !! *j*)) *i*) *m*
in
((*mfsi′*, *dmui′*, *di*, *mods*)))

definition *mod-of-gso-norm* :: *bool* ⇒ *rat* ⇒ *int* **where**
mod-of-gso-norm first mn = *log-base* ^ (*log-ceiling log-base* (*max 2* (
root-rat-ceiling 2 (*mn* ∗ (*rat-of-nat* (*if first then 4 else m* + *3*))) + *1*)))

definition *LLL-adjust-mod* :: *int* ⇒ *bool* ⇒ *state-impl* ⇒ *int* × *state-impl* × *nat* **where**
LLL-adjust-mod p first state = (
let (*b′*, *g-idx*) = *LLL-max-gso-norm first state*;
p′ = *mod-of-gso-norm first b′*
in if p′ < *p then* (*p′*, *perform-adjust-mod p′ state*, *g-idx*)
else (*p*, *state*, *g-idx*)
)

definition *LLL-adjust-swap-add* **where**
LLL-adjust-swap-add p first state g-idx i = (
let state1 = *LLL-swap-add p state i*
in if i − *1* = *g-idx then*
LLL-adjust-mod p first state1 else (*p*, *state1*, *g-idx*))

definition *LLL-step* :: *int* ⇒ *bool* ⇒ *state-impl* ⇒ *nat* ⇒ *nat* ⇒ *int* ⇒ (*int* × *state-impl* × *nat*) × *nat* × *int* **where**
LLL-step p first state g-idx i j = (*if i* = *0 then* ((*p*, *state*, *g-idx*), *Suc i*, *j*)
else let
i1 = *i* − *1*;
iS = *Suc i*;
(-, -, *di*, -) = *state*;
(*num*, *denom*) = *quotient-of* α;
d-i = *di* !! *i*;
d-i1 = *di* !! *i1*;
d-Si = *di* !! *iS*
in if d-i ∗ *d-i* ∗ *denom* ≤ *num* ∗ *d-i1* ∗ *d-Si then*
((*p*, *state*, *g-idx*), *iS*, *j*)
else (*LLL-adjust-swap-add p first state g-idx i*, *i1*, *j* + *1*))

partial-function (*tailrec*) *LLL-main* :: *int* ⇒ *bool* ⇒ *state-impl* ⇒ *nat* ⇒ *nat* ⇒ *int* ⇒ *int* × *state-impl*
where
LLL-main p first state g-idx i (*j* :: *int*) = (

```
    (if i < m
     then case LLL-step p first state g-idx i j of
         ((p', state', g-idx'), i', j') ⇒
         LLL-main p' first state' g-idx' i' j'
       else
         (p, state)))

partial-function (tailrec) LLL-iso-main-inner where
  LLL-iso-main-inner p first state di-prods g-idx (j :: int) = (
      case state of (-, -, di, -) ⇒
    (
      (let (max-gso-num, max-gso-denum, indx) = LLL-max-gso-quot di-prods;
        (num, denum) = quotient-of α in
        (if max-gso-num * denum  > num * max-gso-denum then
           case LLL-adjust-swap-add p first state g-idx indx of
             (p', state', g-idx') ⇒ case state' of (-, -, di', -) ⇒
             let di-prods' = IArray.of-fun (λ i. case di-prods !! i of lr ⇒
                   if i > indx ∨ i + 2 < indx then lr
                    else case lr of (l,r)
                  ⇒ if i + 1 = indx then let d-idx = di' !! indx in (d-idx * d-idx, r)
else (l, di' !! (i + 2) * di' !! i)) (m - 1)
               in LLL-iso-main-inner p' first state' di-prods' g-idx' (j + 1)
         else
           (p, state)))))

definition LLL-iso-main where
  LLL-iso-main p first state g-idx j = (if m > 1 then
     case state of (-, -, di, -) ⇒
     let di-prods = IArray.of-fun (λ i. (di !! (i+1) * di !! (i+1), di !! (i+2) * di !!
i)) (m - 1)
      in LLL-iso-main-inner p first state di-prods g-idx j else (p,state))


definition LLL-initial :: bool ⇒ int × state-impl × nat where
  LLL-initial first = (let init = dμ-impl fs-init;
      di = IArray.of-fun (λ i. if i = 0 then 1 else let i1 = i - 1 in init !! i1 !! i1)
(Suc m);
      (b,g-idx) = LLL-max-gso-norm-di first di;
      p = mod-of-gso-norm first b;
      mods = IArray.of-fun (λ j. p * di !! j * di !! (Suc j)) (m - 1);
       dmui = IArray.of-fun (λ i. let row = init !! i in IArray.of-fun (λ j. row !! j
symmod (mods !! j)) i) m
      in (p, (compute-initial-mfs p, dmui, di, mods), g-idx))

fun LLL-add-rows-loop where
  LLL-add-rows-loop p state i 0 = state
| LLL-add-rows-loop p state i (Suc j) = (
     let state' = LLL-add-row p state i j
      in LLL-add-rows-loop p state' i j)
```

```
primrec LLL-add-rows-outer-loop where
  LLL-add-rows-outer-loop p state 0 = state |
  LLL-add-rows-outer-loop p state (Suc i) =
    (let state' = LLL-add-rows-outer-loop p state i in
      LLL-add-rows-loop p state' (Suc i) (Suc i))

definition
  LLL-reduce-basis = (if m = 0 then [] else
    let first = False;
        (p0, state0, g-idx0) = LLL-initial first;
        (p, state) = LLL-main p0 first state0 g-idx0 0 0;
        (mfs,-,-,-) = LLL-add-rows-outer-loop p state (m - 1)
     in mfs)

definition
  LLL-reduce-basis-iso = (if m = 0 then [] else
    let first = False;
        (p0, state0, g-idx0) = LLL-initial first;
        (p, state) = LLL-iso-main p0 first state0 g-idx0 0;
        (mfs,-,-,-) = LLL-add-rows-outer-loop p state (m - 1)
     in mfs)

definition
  LLL-short-vector = (
    let first = True;
        (p0, state0, g-idx0) = LLL-initial first;
        (p, (mfs,-,-,-)) = LLL-main p0 first state0 g-idx0 0 0
     in hd mfs)

definition
  LLL-short-vector-iso = (
    let first = True;
        (p0, state0, g-idx0) = LLL-initial first;
        (p, (mfs,-,-,-)) = LLL-iso-main p0 first state0 g-idx0 0
     in hd mfs)

end

declare LLL.LLL-short-vector-def[code]
declare LLL.LLL-short-vector-iso-def[code]
declare LLL.LLL-reduce-basis-def[code]
declare LLL.LLL-reduce-basis-iso-def[code]
declare LLL.LLL-iso-main-def[code]
declare LLL.LLL-iso-main-inner.simps[code]
declare LLL.LLL-add-rows-outer-loop.simps[code]
declare LLL.LLL-add-rows-loop.simps[code]
declare LLL.LLL-initial-def[code]
declare LLL.LLL-main.simps[code]
```

declare *LLL.LLL-adjust-mod-def*[*code*]
declare *LLL.LLL-max-gso-norm-def*[*code*]
declare *LLL.perform-adjust-mod-def*[*code*]
declare *LLL.LLL-max-gso-norm-di-def*[*code*]
declare *LLL.LLL-max-gso-quot-def*[*code*]
declare *LLL.LLL-step-def*[*code*]
declare *LLL.LLL-add-row-def*[*code*]
declare *LLL.perform-add-row-def*[*code*]
declare *LLL.LLL-swap-row-def*[*code*]
declare *LLL.LLL-swap-add-def*[*code*]
declare *LLL.LLL-adjust-swap-add-def*[*code*]
declare *LLL.perform-swap-add-def*[*code*]
declare *LLL.mod-of-gso-norm-def*[*code*]
declare *LLL.compute-initial-mfs-def*[*code*]
declare *LLL.log-base-def*[*code*]

5.2 Towards soundness proof of implementation

context *LLL*
begin
lemma *perform-swap-add*: **assumes** *k*: $k \neq 0$ $k < m$ **and** *fs*: *length fs* = *m*
shows *LLL-swap-row p* (*perform-add-row p* (*fs*, *dmu*, *di*, *mods*) *k* (*k* − *1*) *c* (*dmu* !! *k*) (*dmu* !! *k* !! (*k* − *1*)) (*di* !! *k*)) *k*
= *perform-swap-add p* (*fs*, *dmu*, *di*, *mods*) *k* (*k* − *1*) *c* (*dmu* !! *k*) (*dmu* !! *k* !! (*k* − *1*)) (*di* !! *k*)
proof −
from *k*[*folded fs*]
have *drop*: *drop k fs* = *fs* ! *k* # *drop* (*Suc k*) *fs*
by (*simp add*: *Cons-nth-drop-Suc*)
obtain *v* **where** *v*: *vec n* (λ*ka*. (*fs* ! *k* $ *ka* − *c* ∗ *fs* ! (*k* − *1*) $ *ka*) *symmod p*) = *v* **by** *auto*
from *k*[*folded fs*]
have *drop1*: *drop* (*k* − *1*) (*take k fs* @ *v* # *drop* (*Suc k*) *fs*) = *fs* ! (*k* − *1*) # *v* # *drop* (*Suc k*) *fs*
by (*simp add*: *Cons-nth-drop-Suc*)
(*smt Cons-nth-drop-Suc Suc-diff-Suc Suc-less-eq Suc-pred diff-Suc-less diff-self-eq-0 drop-take less-SucI take-Suc-Cons take-eq-Nil*)
from *k*[*folded fs*]
have *drop2*: *drop* (*k* − *1*) *fs* = *fs* ! (*k* − *1*) # *fs* ! *k* # *drop* (*Suc k*) *fs*
by (*metis Cons-nth-drop-Suc One-nat-def Suc-less-eq Suc-pred less-SucI neq0-conv*)
have *take*: *take* (*k* − *1*) (*take k fs* @ *xs*) = *take* (*k* − *1*) *fs* **for** *xs* **using** *k*[*folded fs*] **by** *auto*
obtain *rowk* **where** *rowk*: *IArray.of-fun*
(λ*jj*. *if jj* < *k* − *1 then* (*dmu* !! *k* !! *jj* − *c* ∗ *dmu* !! (*k* − *1*) !! *jj*) *symmod mods* !! *jj*
else if jj = *k* − *1 then dmu* !! *k* !! (*k* − *1*) − *c* ∗ *di* !! *k else dmu* !! *k* !! *jj*) *k* = *rowk*
by *auto*
obtain *mukk1'* **where** *mukk1'*: (*di* !! *Suc k* ∗ *di* !! (*k* − *1*) + *rowk* !! (*k* − *1*) ∗

```
rowk !! (k − 1)) div di !! k = mukk1′
    by auto
  have kk1: k − 1 < k using k by auto
  have mukk1′′: (di !! Suc k ∗ di !! (k − 1) +
          (dmu !! k !! (k − 1) − c ∗ di !! k) ∗ (dmu !! k !! (k − 1) − c ∗ di !! k))
div
          di !! k = mukk1′
    unfolding mukk1′[symmetric] rowk[symmetric] IArray.of-fun-nth[OF kk1] by
auto
  have id: (k = k) = True by simp
  have rowk1: dmu !! k !! (k − 1) − c ∗ di !! k = rowk !! (k − 1)
    unfolding rowk[symmetric] IArray.of-fun-nth[OF kk1] by simp
  show ?thesis
   unfolding perform-swap-add-def split perform-add-row-def Let-def split LLL-swap-row-def
split-at-def
   unfolding drop list.simps v drop1 take prod.inject drop2 rowk IArray.of-fun-nth[OF
‹k < m›] id if-True
    unfolding rowk1
  proof (intro conjI refl iarray-cong, unfold rowk1[symmetric], goal-cases)
    case i: (1 i)
   show ?case unfolding IArray.of-fun-nth[OF i] IArray.of-fun-nth[OF ‹k < m›]
id if-True mukk1′ mukk1′′
      rowk1[symmetric]
    proof (intro if-cong[OF refl], force, goal-cases)
      case 3
      hence i: i = k − 1 by auto
    show ?case unfolding i by (intro iarray-cong[OF refl], unfold rowk[symmetric],
        subst IArray.of-fun-nth, insert k, auto)
    next
      case ki: 1
      hence id: (k = i) = False by auto
      show ?case unfolding id if-False rowk
        by (intro iarray-cong if-cong refl)
    next
      case 2
      show ?case unfolding 2
        by (intro iarray-cong if-cong refl, subst IArray.of-fun-nth, insert k, auto)
    qed
  qed
qed


lemma LLL-swap-add-eq: assumes i: i ≠ 0 i < m and fs: length fs = m
  shows LLL-swap-add p (fs,dmu,di,mods) i = (LLL-swap-row p (LLL-add-row p
(fs,dmu,di,mods) i (i − 1)) i)
proof −
  define c where c = round-num-denom (dmu !! i !! (i − 1)) (di !! i)
  from i have si1: Suc (i − 1) = i by auto
   note res1 = LLL-swap-add-def[of p (fs,dmu,di,mods) i, unfolded split Let-def
```

c-def[*symmetric*]]
show *?thesis*
proof (*cases c = 0*)
case *True*
thus *?thesis* **using** *i* **unfolding** *res1 LLL-add-row-def split id c-def Let-def* **by** *auto*
next
case *False*
hence *c*: (*c = 0*) = *False* **by** *simp*
have *add*: *LLL-add-row p* (*fs*, *dmu*, *di*, *mods*) *i* (*i − 1*) =
perform-add-row p (*fs*, *dmu*, *di*, *mods*) *i* (*i − 1*) *c* (*dmu !! i*) (*dmu !! i !! (i − 1*)) (*di !! i*)
unfolding *LLL-add-row-def Let-def split si1 c-def*[*symmetric*] *c* **by** *auto*
show *?thesis* **unfolding** *res1 c if-False add*
by (*subst perform-swap-add*[*OF assms*]) *simp*
qed
qed
end

context *LLL-with-assms*
begin

lemma *LLL-mod-inv-to-weak*: *LLL-invariant-mod fs mfs dmu p first b i* ⟹ *LLL-invariant-mod-weak fs mfs dmu p first b*
unfolding *LLL-invariant-mod-def LLL-invariant-mod-weak-def* **by** *auto*

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

lemma *LLL-swap-row*: **assumes** *impl*: *state-impl-inv p mfs dmu state*
and *Linv*: *LLL-invariant-mod-weak fs mfs dmu p first b*
and *res*: *basis-reduction-mod-swap p mfs dmu k* = (*mfs′*, *dmu′*)
and *res′*: *LLL-swap-row p state k* = *state′*
and *k*: *k* < *m k* ≠ *0*
shows *state-impl-inv p mfs′ dmu′ state′*
proof −
note *inv* = *LLL-invD-modw*[*OF Linv*]
obtain *fsi dmui di mods* **where** *state*: *state* = (*fsi*, *dmui*, *di*, *mods*) **by** (*cases state*, *auto*)
obtain *fsi′ dmui′ di′ mods′* **where** *state′*: *state′* = (*fsi′*, *dmui′*, *di′*, *mods′*) **by** (*cases state′*, *auto*)
from *impl*[*unfolded state*, *simplified*]
have *id*: *fsi* = *mfs*
di = *IArray.of-fun* (*d-of dmu*) (*Suc m*)
dmui = *IArray.of-fun* (λ*i*. *IArray.of-fun* (λ*j*. *dmu* $$ (*i*, *j*)) *i*) *m*
mods = *IArray.of-fun* (λ*j*. *p* ∗ *di !! j* ∗ *di !! Suc j*) (*m − 1*)
by *auto*
have *kk1*: *dmui !! k !! (k − 1)* = *dmu* $$ (*k*, *k − 1*) **using** *k* **unfolding** *id*
IArray.of-fun-nth[*OF k*(*1*)]

```
  by (subst IArray.of-fun-nth, auto)
 have di: i ≤ m ⟹ di !! i = d-of dmu i for i
  unfolding id by (subst IArray.of-fun-nth, auto)
 have dS1: di !! Suc k = d-of dmu (Suc k) using di k by auto
 have d1: di !! (k − 1) = d-of dmu (k − 1) using di k by auto
 have dk: di !! k = d-of dmu k using di k by auto
 define dk′ where dk′ = (d-of dmu (Suc k) ∗ d-of dmu (k − 1) + dmu $$ (k, k
− 1) ∗ dmu $$ (k, k − 1)) div d-of dmu k
 define mod1 where mod1 = p ∗ d-of dmu (k − 1) ∗ dk′
 define modk where modk = p ∗ dk′ ∗ d-of dmu (Suc k)
 define dmu′′ where dmu′′ = (mat m m
   (λ(i, j).
     if j < i
     then if i = k − 1 then dmu $$ (k, j)
       else if i = k ∧ j ≠ k − 1 then dmu $$ (k − 1, j)
         else if k < i ∧ j = k then (d-of dmu (Suc k) ∗ dmu $$ (i, k − 1)
− dmu $$ (k, k − 1) ∗ dmu $$ (i, j)) div d-of dmu k
           else if k < i ∧ j = k − 1 then (dmu $$ (k, k − 1) ∗ dmu $$
(i, j) + dmu $$ (i, k) ∗ d-of dmu (k − 1)) div d-of dmu k else dmu $$ (i, j)
     else if i = j then if i = k − 1 then (d-of dmu (Suc k) ∗ d-of dmu (k − 1)
+ dmu $$ (k, k − 1) ∗ dmu $$ (k, k − 1)) div d-of dmu k else d-of dmu (Suc i)
       else dmu $$ (i, j)))
 have drop: drop (k − 1) fsi = mfs ! (k − 1) # mfs ! k # drop (Suc k) mfs
unfolding id using ‹length mfs = m› k
  by (metis Cons-nth-drop-Suc One-nat-def Suc-less-eq Suc-pred less-SucI linorder-neqE-nat
not-less0)
 have dk′: dk′ = d-of dmu′′ k unfolding dk′-def d-of-def dmu′′-def using k by
auto
 have mod1: mod1 = p ∗ d-of dmu′′ (k − 1) ∗ d-of dmu′′ k unfolding mod1-def
dk′ using k
  by (auto simp: dmu′′-def d-of-def)
 have modk: modk = p ∗ d-of dmu′′ k ∗ d-of dmu′′ (Suc k) unfolding modk-def
dk′ using k
  by (auto simp: dmu′′-def d-of-def)
 note res = res[unfolded basis-reduction-mod-swap-def, folded dmu′′-def, symmet-
ric]
 note res′ = res′[unfolded state state′ split-at-def drop list.simps split LLL-swap-row-def
Let-def kk1 dS1 d1 dk,
    folded dk′-def mod1-def modk-def, symmetric]
 from res′ have fsi′: fsi′ = take (k − 1) mfs @ mfs ! k # mfs ! (k − 1) # drop
(Suc k) mfs unfolding id by simp
 from res′ have di′: di′ = IArray.of-fun (λii. if ii = k then dk′ else di !! ii) (Suc
m) by simp
 from res′ have dmui′: dmui′ = IArray.of-fun
  (λi. if i < k − 1 then dmui !! i
    else if k < i then IArray.of-fun
      (λj. if j = k − 1
        then (dmu $$ (k, k − 1) ∗ dmui !! i !! (k − 1) + dmui !! i !!
k ∗ d-of dmu (k − 1))
```

```
                    div d-of dmu k symmod mod1
                else if j = k
                    then (d-of dmu (Suc k) * dmui !! i !! (k - 1) - dmu $$
(k, k - 1) * dmui !! i !! k)
                     div d-of dmu k symmod modk
                    else dmui !! i !! j)
            i
        else if i = k then IArray.of-fun (λj. if j = k - 1 then dmu $$ (k, k -
1) symmod mod1
      else dmui !! (k - 1) !! j) i else IArray.of-fun ((!!) (dmui !! k)) i)
  m by auto
 from res' have mods': mods' = IArray.of-fun (λjj. if jj = k - 1 then mod1 else
if jj = k then modk else mods !! jj) (m - 1)
   by auto
 from res have dmu': dmu' = basis-reduction-mod-swap-dmu-mod p dmu'' k by
auto
 show ?thesis unfolding state' state-impl-inv.simps
 proof (intro conjI)
  from res have mfs': mfs' = mfs[k := mfs ! (k - 1), k - 1 := mfs ! k] by simp
  show fsi' = mfs' unfolding fsi' mfs' using ‹length mfs = m› k
  proof (intro nth-equalityI, force, goal-cases)
    case (1 j)
    have choice: j = k - 1 ∨ j = k ∨ j < k - 1 ∨ j > k by linarith
    have min (length mfs) (k - 1) = k - 1 using 1 by auto
    with 1 choice show ?case by (auto simp: nth-append)
  qed
  show di' = IArray.of-fun (d-of dmu') (Suc m) unfolding di'
  proof (intro iarray-cong refl, goal-cases)
    case i: (1 i)
   hence d-of dmu' i = d-of dmu'' i unfolding dmu' basis-reduction-mod-swap-dmu-mod-def
d-of-def
      by (intro if-cong, auto)
    also have ... = ((if i = k then dk' else di !! i))
    proof (cases i = k)
      case False
      hence d-of dmu'' i = d-of dmu i unfolding dmu''-def d-of-def using i k
        by (intro if-cong refl, auto)
      thus ?thesis using False i k unfolding id by (metis iarray-of-fun-sub)
    next
      case True
      thus ?thesis using dk' by auto
    qed
    finally show ?case by simp
  qed
  have dkS1: d-of dmu (Suc k) = d-of dmu'' (Suc k)
    unfolding dmu''-def d-of-def using k by auto
  have dk1: d-of dmu (k - 1) = d-of dmu'' (k - 1)
    unfolding dmu''-def d-of-def using k by auto
  show dmui' = IArray.of-fun (λi. IArray.of-fun (λj. dmu' $$ (i, j)) i) m
```

```
    unfolding dmui'
  proof (intro iarray-cong refl, goal-cases)
    case i: (1 i)
    consider (1) i < k - 1 | (2) i = k - 1 | (3) i = k | (4) i > k by linarith
    thus ?case
    proof (cases)
      case 1
      hence *: (i < k - 1) = True by simp
      show ?thesis unfolding * if-True id IArray.of-fun-nth[OF i] using i k 1
     by (intro iarray-cong refl, auto simp: dmu' basis-reduction-mod-swap-dmu-mod-def,
auto simp: dmu''-def)
    next
      case 2
      hence *: (i < k - 1) = False (k < i) = False (i = k) = False using k by
auto
    show ?thesis unfolding * if-False id using i k 2 unfolding IArray.of-fun-nth[OF
k(1)]
          by (intro iarray-cong refl, subst IArray.of-fun-nth, auto simp: dmu'
basis-reduction-mod-swap-dmu-mod-def dmu''-def)
    next
      case 3
      hence *: (i < k - 1) = False (k < i) = False (i = k) = True using k by
auto
      show ?thesis unfolding * if-False if-True id IArray.of-fun-nth[OF k(1)]
      proof (intro iarray-cong refl, goal-cases)
        case j: (1 j)
        show ?case
        proof (cases j = k - 1)
          case False
          hence *: (j = k - 1) = False by auto
          show ?thesis unfolding * if-False using False j k i 3
            by (subst IArray.of-fun-nth, force, subst IArray.of-fun-nth, force, auto
simp: dmu' basis-reduction-mod-swap-dmu-mod-def dmu''-def)
        next
          case True
          hence *: (j = k - 1) = True by auto
          show ?thesis unfolding * if-True unfolding True 3 using k
            by (auto simp: basis-reduction-mod-swap-dmu-mod-def dmu' dk' mod1
dmu''-def)
        qed
      qed
    next
      case 4
      hence *: (i < k - 1) = False (k < i) = True using k by auto
       show ?thesis unfolding * if-False if-True id IArray.of-fun-nth[OF k(1)]
IArray.of-fun-nth[OF ‹i < m›]
      proof (intro iarray-cong refl, goal-cases)
        case j: (1 j)
        from 4 have k1: k - 1 < i by auto
```

```
        show ?case unfolding IArray.of-fun-nth[OF j] IArray.of-fun-nth[OF 4]
IArray.of-fun-nth[OF k1]
          unfolding mod1 modk dmu' basis-reduction-mod-swap-dmu-mod-def
using i j 4 k
          by (auto intro!: arg-cong[of - - λ x. x symmod -], auto simp: dmu''-def)
      qed
    qed
  qed
  show mods' = IArray.of-fun (λj. p * di' !! j * di' !! Suc j) (m - 1)
    unfolding mods' di' dk' mod1 modk
  proof (intro iarray-cong refl, goal-cases)
    case (1 j)
    hence j: j < Suc m Suc j < Suc m by auto
    show ?case unfolding
      IArray.of-fun-nth[OF 1]
      IArray.of-fun-nth[OF j(1)]
      IArray.of-fun-nth[OF j(2)] id(4) using k di dk1 dkS1
      by auto
  qed
 qed
qed


lemma LLL-add-row: assumes impl: state-impl-inv p mfs dmu state
 and Linv: LLL-invariant-mod-weak fs mfs dmu p first b
 and res: basis-reduction-mod-add-row p mfs dmu i j = (mfs', dmu')
 and res': LLL-add-row p state i j = state'
 and i: i < m
 and j: j < i
shows state-impl-inv p mfs' dmu' state'
proof -
 note inv = LLL-invD-modw[OF Linv]
 obtain fsi dmui di mods where state: state = (fsi, dmui, di, mods) by (cases
state, auto)
 obtain fsi' dmui' di' mods' where state': state' = (fsi', dmui', di', mods') by
(cases state', auto)
 from impl[unfolded state, simplified]
 have id: fsi = mfs
   di = IArray.of-fun (d-of dmu) (Suc m)
   dmui = IArray.of-fun (λi. IArray.of-fun (λj. dmu $$ (i, j)) i) m
   mods = IArray.of-fun (λj. p * di !! j * di !! Suc j) (m - 1)
   by auto
 let ?c = round-num-denom (dmu $$ (i, j)) (d-of dmu (Suc j))
 let ?c' = round-num-denom (dmui !! i !! j) (di !! Suc j)
 obtain c where c: ?c = c by auto
 have c': ?c' = c unfolding id c[symmetric] using i j
   by (subst (1 2) IArray.of-fun-nth, (force+)[2],
     subst IArray.of-fun-nth, force+)
 have drop: drop i fsi = mfs ! i # drop (Suc i) mfs unfolding id using ‹length
```

```
mfs = m› i
    by (metis Cons-nth-drop-Suc)
  note res = res[unfolded basis-reduction-mod-add-row-def Let-def c, symmetric]
  note res' = res'[unfolded state state' split LLL-add-row-def Let-def c', symmetric]
  show ?thesis
  proof (cases c = 0)
    case True
     from res[unfolded True] res'[unfolded True] show ?thesis unfolding state'
using id by auto
  next
    case False
    hence False: (c = 0) = False by simp
    note res = res[unfolded Let-def False if-False]
    from res have mfs': mfs' = mfs[i := map-vec (λx. x symmod p) (mfs ! i − c
·v mfs ! j)] by auto
    from res have dmu': dmu' = mat m m (λ(i', j').
        if i' = i ∧ j' ≤ j
        then if j' = j then dmu $$ (i, j') − c ∗ dmu $$ (j, j')
             else (dmu $$ (i, j') − c ∗ dmu $$ (j, j')) symmod (p ∗ d-of dmu j' ∗
d-of dmu (Suc j'))
        else dmu $$ (i', j')) by auto
   note res' = res'[unfolded Let-def False if-False perform-add-row-def drop list.simps
split-at-def split]
    from res' have fsi': fsi' = take i fsi @ vec n (λk. (mfs ! i $ k − c ∗ mfs ! j $
k) symmod p) # drop (Suc i) mfs
      by (auto simp: id)
    from res' have di': di' = di and mods': mods' = mods by auto
    from res' have dmui': dmui' = IArray.of-fun (λii. if i = ii
          then IArray.of-fun
              (λjj. if jj < j then (dmui !! i !! jj − c ∗ dmui !! j !! jj) symmod (mods
!! jj)
                    else if jj = j then dmui !! i !! j − c ∗ di !! (Suc j) else dmui !! i
!! jj)
               i
          else dmui !! ii) m by auto
    show ?thesis unfolding state' state-impl-inv.simps
    proof (intro conjI)
      from inv(11) i j have vec: mfs ! i ∈ carrier-vec n mfs ! j ∈ carrier-vec n by
auto
       hence id': map-vec (λx. x symmod p) (mfs ! i − c ·v mfs ! j) = vec n (λk.
(mfs ! i $ k − c ∗ mfs ! j $ k) symmod p)
        by (intro eq-vecI, auto)
      show mods' = IArray.of-fun (λj. p ∗ di' !! j ∗ di' !! Suc j) (m − 1) using id
unfolding mods' di' by auto
      show fsi' = mfs' unfolding fsi' mfs' id unfolding id' using ‹length mfs =
m› i
        by (simp add: upd-conv-take-nth-drop)
      show di' = IArray.of-fun (d-of dmu') (Suc m)
        unfolding dmu' di' id d-of-def
```

```
      by (intro iarray-cong if-cong refl, insert i j, auto)
    show dmui' = IArray.of-fun (λi. IArray.of-fun (λj. dmu' $$ (i, j)) i) m
      unfolding dmui'
    proof (intro iarray-cong refl)
      fix ii
      assume ii: ii < m
      show (if i = ii
        then IArray.of-fun
            (λjj. if jj < j then (dmui !! i !! jj − c ∗ dmui !! j !! jj) symmod (mods
!! jj)
                  else if jj = j then dmui !! i !! j − c ∗ di !! (Suc j) else dmui !! i
!! jj)
             i
        else dmui !! ii) =
        IArray.of-fun (λj. dmu' $$ (ii, j)) ii
      proof (cases i = ii)
        case False
        hence ∗: (i = ii) = False by auto
        show ?thesis unfolding ∗ if-False id dmu' using False i j ii
          unfolding IArray.of-fun-nth[OF ii]
          by (intro iarray-cong refl, auto)
      next
        case True
        hence ∗: (i = ii) = True by auto
        from i j have j < m by simp
         show ?thesis unfolding ∗ if-True dmu' id IArray.of-fun-nth[OF i] IAr-
ray.of-fun-nth[OF ‹j < m›]
          unfolding True[symmetric]
        proof (intro iarray-cong refl, goal-cases)
          case jj: (1 jj)
          consider (1) jj < j | (2) jj = j | (3) jj > j by linarith
          thus ?case
          proof cases
            case 1
            thus ?thesis using jj i j unfolding id(4)
              by (subst (1 2 3 4 5 6) IArray.of-fun-nth, auto)
          next
            case 2
            thus ?thesis using jj i j
              by (subst (5 6) IArray.of-fun-nth, auto simp: d-of-def)
          next
            case 3
            thus ?thesis using jj i j
              by (subst (7) IArray.of-fun-nth, auto simp: d-of-def)
          qed
        qed
      qed
    qed
  qed
```

```
  qed
qed


lemma LLL-max-gso-norm-di: assumes di: di = IArray.of-fun (d-of dmu) (Suc
m)
  and m: m ≠ 0
shows LLL-max-gso-norm-di first di = compute-max-gso-norm first dmu
proof −
  have di: j ≤ m ⟹ di !! j = d-of dmu j for j unfolding di
    by (subst IArray.of-fun-nth, auto)
  have id: (m = 0) = False using m by auto
  show ?thesis
  proof (cases first)
    case False
    hence id': first = False by auto
    show ?thesis unfolding LLL-max-gso-norm-di-def compute-max-gso-norm-def
id id' if-False
      by (intro if-cong refl arg-cong[of - - λ xs. case max-list-rats-with-index xs of
(num, denom, i) ⇒ (rat-of-int num / rat-of-int denom, i)],
          unfold map-eq-conv, intro ballI, subst (1 2) di, auto)
  next
    case True
    hence id': first = True by auto
    show ?thesis unfolding LLL-max-gso-norm-di-def compute-max-gso-norm-def
id id' if-False if-True
      using m di[of 1]
      by (simp add: d-of-def)
  qed
qed

lemma LLL-max-gso-quot: assumes di: di = IArray.of-fun (d-of dmu) (Suc m)
  and prods: state-iso-inv di-prods di
shows LLL-max-gso-quot di-prods = compute-max-gso-quot dmu
proof −
  have di: j ≤ m ⟹ di !! j = d-of dmu j for j unfolding di
    by (subst IArray.of-fun-nth, auto)
 show ?thesis unfolding LLL-max-gso-quot-def compute-max-gso-quot-def prods[unfolded
state-iso-inv-def]
   by (intro if-cong refl arg-cong[of - - max-list-rats-with-index], unfold map-eq-conv
Let-def, intro ballI,
     subst IArray.of-fun-nth, force, unfold split,
     subst (1 2 3 4) di, auto)
qed

lemma LLL-max-gso-norm: assumes impl: state-impl-inv p mfs dmu state
  and m: m ≠ 0
shows LLL-max-gso-norm first state = compute-max-gso-norm first dmu
proof −
```

obtain *mfsi dmui di mods* **where** *state*: *state* = (*mfsi*, *dmui*, *di*,*mods*)
by (*metis prod-cases3*)
from *impl*[*unfolded state state-impl-inv.simps*]
have *di*: *di* = *IArray.of-fun* (*d-of dmui*) (*Suc m*) **by** *auto*
show *?thesis* **using** *LLL-max-gso-norm-di*[*OF di m*] **unfolding** *LLL-max-gso-norm-def state split* **.**
qed

lemma *mod-of-gso-norm*: $m \neq 0 \Longrightarrow$ *mod-of-gso-norm first mn* =
compute-mod-of-max-gso-norm first mn
unfolding *mod-of-gso-norm-def compute-mod-of-max-gso-norm-def bound-number-def*
by *auto*

lemma *LLL-adjust-mod*: **assumes** *impl*: *state-impl-inv p mfs dmu state*
and *res*: *basis-reduction-adjust-mod p first mfs dmu* = (*p'*, *mfs'*, *dmu'*, *g-idx*)
and *res'*: *LLL-adjust-mod p first state* = (*p''*, *state'*, *g-idx'*)
and *m*: $m \neq 0$
shows *state-impl-inv p' mfs' dmu' state'* $\wedge$ $p'' = p' \wedge$ *g-idx'* = *g-idx*
proof −
from *LLL-max-gso-norm*[*OF impl m*]
have *id*: *LLL-max-gso-norm first state* = *compute-max-gso-norm first dmu* **by** *auto*
obtain *b gi* **where** *norm*: *compute-max-gso-norm first dmu* = (*b*, *gi*) **by** *force*
obtain *P* **where** *P*: *compute-mod-of-max-gso-norm first b* = *P* **by** *auto*
note *res* = *res*[*unfolded basis-reduction-adjust-mod.simps Let-def P norm split*]
note *res'* = *res'*[*unfolded LLL-adjust-mod-def id Let-def P norm split mod-of-gso-norm*[*OF m*]]
show *?thesis*
proof (*cases P* < *p*)
case *False*
thus *?thesis* **using** *res res' impl* **by** (*auto split*: *if-splits*)
next
case *True*
hence *id*: (*P* < *p*) = *True* **by** *auto*
obtain *fsi dmui di mods* **where** *state*: *state* = (*fsi*, *dmui*, *di*, *mods*) **by** (*metis prod-cases3*)
from *impl*[*unfolded state state-impl-inv.simps*]
have *impl*: *fsi* = *mfs di* = *IArray.of-fun* (*d-of dmui*) (*Suc m*) *dmui* = *IArray.of-fun* (λ*i*. *IArray.of-fun* (λ*j*. *dmui* $$ (*i*, *j*)) *i*) *m* **by** *auto*
note *res* = *res*[*unfolded id if-True*]
from *res* **have** *mfs'*: *mfs'* = *map* (*map-vec* (λ*x*. *x symmod P*)) *mfs*
and *p'*: *p'* = *P*
and *dmu'*: *dmu'* = *mat m m* (λ(*i*, *j*). *if j* < *i then dmu* $$ (*i*, *j*) *symmod* (*P* ∗ *vec* (*Suc m*) (*d-of dmu*) $ *j* ∗ *vec* (*Suc m*) (*d-of dmu*) $ *Suc j*) *else dmu* $$ (*i*, *j*))
and *gidx*: *g-idx* = *gi*
by *auto*
let *?mods* = *IArray.of-fun* (λ*j*. *P* ∗ *di* !! *j* ∗ *di* !! *Suc j*) (*m* − *1*)
let *?dmu* = *IArray.of-fun* (λ*i*. *IArray.of-fun* (λ*j*. *dmui* !! *i* !! *j symmod ?mods*

```
!! j) i) m
    note res' = res'[unfolded id if-True state split impl(1) perform-adjust-mod-def
Let-def]
   from res' have p'': p'' = P and state': state' = (map (map-vec (λx. x symmod
P)) mfs, ?dmu, di, ?mods)
       and gidx': g-idx' = gi by auto
    show ?thesis unfolding state' state-impl-inv.simps mfs' p'' p' gidx gidx'
    proof (intro conjI refl)
      show di = IArray.of-fun (d-of dmu') (Suc m) unfolding impl
        by (intro iarray-cong refl, auto simp: dmu' d-of-def)
      show ?dmu = IArray.of-fun (λi. IArray.of-fun (λj. dmu' $$ (i, j)) i) m
      proof (intro iarray-cong refl, goal-cases)
        case (1 i j)
        hence j < m Suc j < Suc m j < Suc m j < m − 1 by auto
          show ?case unfolding dmu' impl IArray.of-fun-nth[OF ‹i < m›] IAr-
ray.of-fun-nth[OF ‹j < i›]
            IArray.of-fun-nth[OF ‹j < m›] IArray.of-fun-nth[OF ‹Suc j < Suc m›]
            IArray.of-fun-nth[OF ‹j < Suc m›] IArray.of-fun-nth[OF ‹j < m − 1›]
using 1 by auto
      qed
    qed
  qed
qed

lemma LLL-adjust-swap-add: assumes impl: state-impl-inv p mfs dmu state
  and Linv: LLL-invariant-mod-weak fs mfs dmu p first b
  and res: basis-reduction-adjust-swap-add-step p first mfs dmu g-idx k = (p', mfs',
dmu', g-idx')
  and res': LLL-adjust-swap-add p first state g-idx k = (p'',state', G-idx')
  and k: k < m and k0: k ≠ 0
shows state-impl-inv p' mfs' dmu' state' p'' = p' G-idx' = g-idx'
  i ≤ m ⟹ i ≠ k ⟹ di-of state' !! i = di-of state !! i
proof (atomize(full), goal-cases)
  case 1
  from k have m: m ≠ 0 by auto
  obtain mfsi dmui di mods where state: state = (mfsi, dmui, di, mods)
    by (metis prod-cases3)
  obtain state'' where add': LLL-add-row p state k (k − 1) = state'' by blast
  obtain mfs'' dmu'' where add: basis-reduction-mod-add-row p mfs dmu k (k −
1) = (mfs'', dmu'') by force
   obtain mfs3 dmu3 where swap: basis-reduction-mod-swap p mfs'' dmu'' k =
(mfs3, dmu3) by force
  obtain state3 where swap': LLL-swap-row p state'' k = state3 by blast
   obtain mfsi2 dmui2 di2 mods2 where state2: state'' = (mfsi2, dmui2, di2,
mods2) by (cases state'', auto)
   obtain mfsi3 dmui3 di3 mods3 where state3: state3 = (mfsi3, dmui3, di3,
mods3) by (cases state3, auto)
  have length mfsi = m using impl[unfolded state state-impl-inv.simps] LLL-invD-modw[OF
Linv] by auto
```

note *res'* = *res'*[*unfolded state LLL-adjust-swap-add-def LLL-swap-add-eq*[*OF k0 k this*], *folded state*, *unfolded add' swap' Let-def*]
note *res* = *res*[*unfolded basis-reduction-adjust-swap-add-step-def Let-def add split swap*]
from *LLL-add-row*[*OF impl Linv add add' k*] *k0*
have *impl'*: *state-impl-inv p mfs'' dmu'' state''* **by** *auto*
from *basis-reduction-mod-add-row*[*OF Linv add k - k0*] *k0*
obtain *fs''* **where** *Linv'*: *LLL-invariant-mod-weak fs'' mfs'' dmu'' p first b* **by** *auto*
from *LLL-swap-row*[*OF impl' Linv' swap swap' k k0*]
have *impl3*: *state-impl-inv p mfs3 dmu3 state3* **.**
have *di2*: *di2* = *di* **using** *add'*[*unfolded state LLL-add-row-def Let-def split perform-add-row-def state2*]
by (*auto split*: *if-splits*)
have *di3*: *di3* = *IArray.of-fun* (λ*i*. *if i* = *k then* (*di2* !! *Suc k* ∗ *di2* !! (*k* − *1*) + *dmui2* !! *k* !! (*k* − *1*) ∗ *dmui2* !! *k* !! (*k* − *1*)) *div di2* !! *k else di2* !! *i*) (*Suc m*)
using *swap'*[*unfolded state2 state3*]
unfolding *LLL-swap-row-def Let-def* **by** *simp*
have *di3*: *i* ≤ *m* ⟹ *i* ≠ *k* ⟹ *di3* !! *i* = *di* !! *i*
unfolding *di2*[*symmetric*] *di3*
by (*subst IArray.of-fun-nth*, *auto*)
show *?case*
proof (*cases k* − *1* = *g-idx*)
case *True*
hence *id*: (*k* − *1* = *g-idx*) = *True* **by** *simp*
note *res* = *res*[*unfolded id if-True*]
note *res'* = *res'*[*unfolded id if-True*]
obtain *mfsi4 dmui4 di4 mods4* **where** *state'*: *state'* = (*mfsi4*, *dmui4*, *di4*, *mods4*) **by** (*cases state'*, *auto*)
from *res'*[*unfolded state3 state' LLL-adjust-mod-def Let-def perform-adjust-mod-def*] **have** *di4*: *di4* = *di3*
by (*auto split*: *if-splits prod.splits*)
from *LLL-adjust-mod*[*OF impl3 res res' m*] *di3 state state' di4 res'*
show *?thesis* **by** *auto*
next
case *False*
hence *id*: (*k* − *1* = *g-idx*) = *False* **by** *simp*
note *res* = *res*[*unfolded id if-False*]
note *res'* = *res'*[*unfolded id if-False*]
from *impl3 res res' di3 state state3* **show** *?thesis* **by** *auto*
qed
qed

lemma *LLL-step*: **assumes** *impl*: *state-impl-inv p mfs dmu state*
and *Linv*: *LLL-invariant-mod-weak fs mfs dmu p first b*
and *res*: *basis-reduction-mod-step p first mfs dmu g-idx k j* = (*p'*, *mfs'*, *dmu'*, *g-idx'*, *k'*, *j'*)

and *res'*: *LLL-step p first state g-idx k j* = ((*p''*,*state'*, *g-idx''*), *k''*, *j''*)
and *k*: *k* < *m*
shows *state-impl-inv p' mfs' dmu' state'* ∧ *k''* = *k'* ∧ *p''* = *p'* ∧ *j''* = *j'* ∧ *g-idx''* = *g-idx'*
proof (*cases k* = *0*)
case *True*
thus *?thesis* **using** *res res' impl* **unfolding** *LLL-step-def basis-reduction-mod-step-def* **by** *auto*
next
case *k0*: *False*
hence *id*: (*k* = *0*) = *False* **by** *simp*
note *res* = *res*[*unfolded basis-reduction-mod-step-def id if-False*]
obtain *num denom* **where** *alph*: *quotient-of* α = (*num*,*denom*) **by** *force*
obtain *mfsi dmui di mods* **where** *state*: *state* = (*mfsi*, *dmui*, *di*, *mods*)
by (*metis prod-cases3*)
note *res'* = *res'*[*unfolded LLL-step-def id if-False Let-def state split alph*, *folded state*]
from *k0* **have** *kk1*: *k* − *1* < *k* **by** *auto*
note *res* = *res*[*unfolded Let-def alph split*]
obtain *state''* **where** *addi*: *LLL-swap-add p state k* = *state''* **by** *auto*
from *impl*[*unfolded state state-impl-inv.simps*]
have *di*: *di* = *IArray.of-fun* (*d-of dmu*) (*Suc m*) **by** *auto*
have *id*: *di* !! *k* = *d-of dmu k*
di !! (*Suc k*) = *d-of dmu* (*Suc k*)
di !! (*k* − *1*) = *d-of dmu* (*k* − *1*)
unfolding *di* **using** *k*
by (*subst IArray.of-fun-nth*, *force*, *force*)+
have *length mfsi* = *m* **using** *impl*[*unfolded state state-impl-inv.simps*] *LLL-invD-modw*[*OF Linv*] **by** *auto*
note *res'* = *res'*[*unfolded id*]
let *?cond* = *d-of dmu k* ∗ *d-of dmu k* ∗ *denom* ≤ *num* ∗ *d-of dmu* (*k* − *1*) ∗ *d-of dmu* (*Suc k*)
show *?thesis*
proof (*cases ?cond*)
case *True*
from *True res res' state* **show** *?thesis* **using** *impl* **by** *auto*
next
case *False*
hence *cond*: *?cond* = *False* **by** *simp*
note *res* = *res*[*unfolded cond if-False*]
note *res'* = *res'*[*unfolded cond if-False*]
let *?step* = *basis-reduction-adjust-swap-add-step p first mfs dmu g-idx k*
let *?step'* = *LLL-adjust-swap-add p first state g-idx k*
from *res* **have** *step*: *?step* = (*p'*, *mfs'*, *dmu'*, *g-idx'*) **by** (*cases ?step*, *auto*)
note *res* = *res*[*unfolded step split*]
from *res'* **have** *step'*: *?step'* = (*p''*,*state'*, *g-idx''*) **by** *auto*
note *res'* = *res'*[*unfolded step'*]
from *LLL-adjust-swap-add*[*OF impl Linv step step' k k0*]
show *?thesis* **using** *res res'* **by** *auto*

qed
qed

lemma *LLL-main*: **assumes** *impl*: *state-impl-inv p mfs dmu state*
and *Linv*: *LLL-invariant-mod fs mfs dmu p first b i*
and *res*: *basis-reduction-mod-main p first mfs dmu g-idx i k* = (*p'*, *mfs'*, *dmu'*)
and *res'*: *LLL-main p first state g-idx i k* = (*pi'*, *state'*)
shows *state-impl-inv p' mfs' dmu' state'* ∧ *pi'* = *p'*
using *assms*
proof (*induct LLL-measure i fs arbitrary*: *mfs dmu state fs p b k i g-idx rule*: *less-induct*)
case (*less fs i mfs dmu state p b k g-idx*)
note *impl* = *less*(*2*)
note *Linv* = *less*(*3*)
note *res* = *less*(*4*)
note *res'* = *less*(*5*)
note *IH* = *less*(*1*)
note *res* = *res*[*unfolded basis-reduction-mod-main.simps*[*of - - - - - - k*]]
note *res'* = *res'*[*unfolded LLL-main.simps*[*of - - - - - k*]]
note *Linvw* = *LLL-mod-inv-to-weak*[*OF Linv*]
show *?case*
proof (*cases i* < *m*)
case *False*
thus *?thesis* **using** *res res' impl* **by** *auto*
next
case *i*: *True*
hence *id*: (*i* < *m*) = *True* **by** *simp*
obtain *P'' state'' I'' K'' G-idx''* **where** *step'*: *LLL-step p first state g-idx i k* = ((*P''*, *state''*, *G-idx''*), *I''*, *K''*)
by (*metis prod-cases3*)
obtain *p'' mfs'' dmu'' i'' k'' g-idx''* **where** *step*: *basis-reduction-mod-step p first mfs dmu g-idx i k* = (*p''*, *mfs''*, *dmu''*, *g-idx''*, *i''*, *k''*)
by (*metis prod-cases3*)
from *LLL-step*[*OF impl Linvw step step' i*]
have *impl''*: *state-impl-inv p'' mfs'' dmu'' state''* **and** *ID*: *I''* = *i''* *K''* = *k''* *P''* = *p''* *G-idx''* = *g-idx''* **by** *auto*
from *basis-reduction-mod-step*[*OF Linv step i*] **obtain**
fs'' b'' **where**
Linv'': *LLL-invariant-mod fs'' mfs'' dmu'' p'' first b'' i''* **and**
decr: *LLL-measure i'' fs''* < *LLL-measure i fs* **by** *auto*
note *res* = *res*[*unfolded id if-True step split*]
note *res'* = *res'*[*unfolded id if-True step' split ID*]
show *?thesis*
by (*rule IH*[*OF decr impl'' Linv'' res res'*])
qed
qed

lemma *LLL-iso-main-inner*: **assumes** *impl*: *state-impl-inv p mfs dmu state*

and *di-prods*: *state-iso-inv di-prods* (*di-of state*)
and *Linv*: *LLL-invariant-mod-weak fs mfs dmu p first b*
and *res*: *basis-reduction-iso-main p first mfs dmu g-idx k* = (*p'*, *mfs'*, *dmu'*)
and *res'*: *LLL-iso-main-inner p first state di-prods g-idx k* = (*pi'*, *state'*)
and *m*: *m* > *1*
shows *state-impl-inv p' mfs' dmu' state'* ∧ *pi'* = *p'*
using *assms*(*1*−*5*)
proof (*induct LLL-measure* (*m* − *1*) *fs arbitrary*: *mfs dmu state fs p b k di-prods g-idx rule*: *less-induct*)
case (*less fs mfs dmu state p b k di-prods g-idx*)
note *impl* = *less*(*2*)
note *di-prods* = *less*(*3*)
note *Linv* = *less*(*4*)
note *res* = *less*(*5*)
note *res'* = *less*(*6*)
note *IH* = *less*(*1*)
obtain *mfsi dmui di mods* **where** *state*: *state* = (*mfsi*, *dmui*, *di*, *mods*)
by (*metis prod-cases4*)
from *di-prods state* **have** *di-prods*: *state-iso-inv di-prods di* **by** *auto*
obtain *num denom idx* **where** *quot'*: *LLL-max-gso-quot di-prods* = (*num*, *denom*, *idx*)
by (*metis prod-cases3*)
note *inv* = *LLL-invD-modw*[*OF Linv*]
obtain *na da* **where** *alph*: *quotient-of* α = (*na*,*da*) **by** *force*
from *impl*[*unfolded state*] **have** *di*: *di* = *IArray.of-fun* (*d-of dmu*) (*Suc m*) **by** *auto*
from *LLL-max-gso-quot*[*OF di di-prods*] **have** *quot*: *compute-max-gso-quot dmu* = *LLL-max-gso-quot di-prods* **..**
obtain *cmp* **where** *cmp*: (*na* ∗ *denom* < *num* ∗ *da*) = *cmp* **by** *force*
have (*m* > *1*) = *True* **using** *m* **by** *auto*
note *res* = *res*[*unfolded basis-reduction-iso-main.simps*[*of - - - - - k*] *this if-True Let-def quot quot' split alph cmp*]
note *res'* = *res'*[*unfolded LLL-iso-main-inner.simps*[*of - - - - - k*] *state split Let-def quot' alph cmp*, *folded state*]
note *cmp* = *compute-max-gso-quot-alpha*[*OF Linv quot*[*unfolded quot'*] *alph cmp m*]
show *?case*
proof (*cases cmp*)
case *False*
thus *?thesis* **using** *res res' impl* **by** *auto*
next
case *True*
hence *id*: *cmp* = *True* **by** *simp*
note *cmp* = *cmp*(*1*)[*OF True*]
obtain *state''* *P''* *G-idx''* **where** *step'*: *LLL-adjust-swap-add p first state g-idx idx* = (*P''*,*state''*, *G-idx''*)
by (*metis prod.exhaust*)
obtain *mfs''* *dmu''* *p''* *g-idx''* **where** *step*: *basis-reduction-adjust-swap-add-step p first mfs dmu g-idx idx* = (*p''*, *mfs''*, *dmu''*, *g-idx''*)

by (*metis prod-cases3*)
obtain *mfsi2 dmui2 di2 mods2* **where** *state2*: *state''* = (*mfsi2*, *dmui2*, *di2*, *mods2*) **by** (*cases state''*, *auto*)
note *res* = *res*[*unfolded id if-True step split*]
note *res'* = *res'*[*unfolded id if-True step' state2 split*, *folded state2*]
from *cmp* **have** *idx0*: *idx* ≠ *0* **and** *idx*: *idx* < *m* **and** *ineq*: ¬ *d-of dmu idx* ∗ *d-of dmu idx* ∗ *da* ≤ *na* ∗ *d-of dmu* (*idx* − *1*) ∗ *d-of dmu* (*Suc idx*)
by *auto*
from *basis-reduction-adjust-swap-add-step*[*OF Linv step alph ineq idx idx0*]
obtain *fs''* *b''* **where** *Linv''*: *LLL-invariant-mod-weak fs'' mfs'' dmu'' p'' first b''* **and**
meas: *LLL-measure* (*m* − *1*) *fs''* < *LLL-measure* (*m* − *1*) *fs* **by** *auto*
from *LLL-adjust-swap-add*[*OF impl Linv step step' idx idx0*]
have *impl''*: *state-impl-inv p'' mfs'' dmu'' state''* **and** *P''*: *P''* = *p''* *G-idx''* = *g-idx''*
and *di-prod-upd*: ⋀ *i*. *i* ≤ *m* ⟹ *i* ≠ *idx* ⟹ *di2* !! *i* = *di* !! *i*
using *state state2* **by** *auto*
have *di-prods*: *state-iso-inv* (*IArray.of-fun*
(λ*i*. *if idx* < *i* ∨ *i* + *2* < *idx then di-prods* !! *i*
else case di-prods !! *i of* (*l*, *r*) ⇒ *if i* + *1* = *idx then* (*di2* !! *idx* ∗ *di2* !! *idx*, *r*) *else* (*l*, *di2* !! (*i* + *2*) ∗ *di2* !! *i*))
(*m* − *1*)) *di2* **unfolding** *state-iso-inv-def*
by (*intro iarray-cong'*, *insert di-prod-upd*, *unfold di-prods*[*unfolded state-iso-inv-def*], *subst* (*1 2*) *IArray.of-fun-nth*, *auto*)
show *?thesis*
by (*rule IH*[*OF meas impl'' - Linv'' res res'*[*unfolded step' P''*]], *insert di-prods state2*, *auto*)
qed
qed

lemma *LLL-iso-main*: **assumes** *impl*: *state-impl-inv p mfs dmu state*
and *Linv*: *LLL-invariant-mod-weak fs mfs dmu p first b*
and *res*: *basis-reduction-iso-main p first mfs dmu g-idx k* = (*p'*, *mfs'*, *dmu'*)
and *res'*: *LLL-iso-main p first state g-idx k* = (*pi'*, *state'*)
shows *state-impl-inv p' mfs' dmu' state'* ∧ *pi'* = *p'*
proof (*cases m* > *1*)
case *True*
from *LLL-iso-main-inner*[*OF impl - Linv res - True*, *unfolded state-iso-inv-def*, *OF refl*, *of pi' state'*] *res' True*
show *?thesis* **unfolding** *LLL-iso-main-def* **by** (*cases state*, *auto*)
next
case *False*
thus *?thesis* **using** *res res' impl* **unfolding** *LLL-iso-main-def*
basis-reduction-iso-main.simps[*of - - - - - k*] **by** *auto*
qed

lemma *LLL-initial*: **assumes** *res*: *compute-initial-state first* = (*p*, *mfs*, *dmu*, *g-idx*)

and *res'*: *LLL-initial first* = (*p'*, *state*, *g-idx'*)

and m: $m \neq 0$
shows *state-impl-inv p mfs dmu state* $\wedge$ $p' = p \wedge$ *g-idx'* = *g-idx*
proof −
obtain *b gi* **where** *norm*: *compute-max-gso-norm first dmu-initial* = (*b*,*gi*) **by** *force*
obtain *P* **where** *P*: *compute-mod-of-max-gso-norm first b* = *P* **by** *auto*
define *di* **where** *di* = *IArray.of-fun* (λi. *if i* = *0 then 1 else d*μ*-impl fs-init* !! (*i* − *1*) !! (*i* − *1*)) (*Suc m*)
note *res* = *res*[*unfolded compute-initial-state-def Let-def P norm split*]
have *di*: *di* = *IArray.of-fun* (*d-of dmu-initial*) (*Suc m*)
unfolding *di-def dmu-initial-def Let-def d-of-def*
by (*intro iarray-cong refl if-cong*, *auto*)
note *norm'* = *LLL-max-gso-norm-di*[*OF di m*, *of first*, *unfolded norm*]
note *res'* = *res'*[*unfolded LLL-initial-def Let-def*, *folded di-def*, *unfolded norm' P split mod-of-gso-norm*[*OF m*]]
from *res* **have** *p*: *p* = *P* **and** *mfs*: *mfs* = *compute-initial-mfs p* **and** *dmu*: *dmu* = *compute-initial-dmu P dmu-initial*
and *g-idx*: *g-idx* = *gi*
by *auto*
let *?mods* = *IArray.of-fun* (λj. *P* ∗ *di* !! *j* ∗ *di* !! *Suc j*) (*m* − *1*)
have *di'*: *di* = *IArray.of-fun* (*d-of* (*compute-initial-dmu P dmu-initial*)) (*Suc m*)
unfolding *di*
by (*intro iarray-cong refl*, *auto simp*: *compute-initial-dmu-def d-of-def*)
from *res'* **have** *p'*: *p'* = *P* **and** *g-idx'*: *g-idx'* = *gi* **and** *state*:
state = (*compute-initial-mfs P*, *IArray.of-fun* (λi. *IArray.of-fun* (λj. *d*μ*-impl fs-init* !! *i* !! *j symmod ?mods* !! *j*) *i*) *m*, *di*, *?mods*)
by *auto*
show *?thesis* **unfolding** *mfs p state p' dmu state-impl-inv.simps g-idx' g-idx*
proof (*intro conjI refl di' iarray-cong*, *goal-cases*)
case (*1 i j*)
hence *j* < *m Suc j* < *Suc m j* < *Suc m j* < *m* − *1* **by** *auto*
thus *?case* **unfolding** *compute-initial-dmu-def di*
IArray.of-fun-nth[*OF* ‹*j* < *m*›]
IArray.of-fun-nth[*OF* ‹*Suc j* < *Suc m*›]
IArray.of-fun-nth[*OF* ‹*j* < *Suc m*›]
IArray.of-fun-nth[*OF* ‹*j* < *m* − *1*›]
unfolding *dmu-initial-def Let-def* **using** *1* **by** *auto*
qed
qed

lemma *LLL-add-rows-loop*: **assumes** *impl*: *state-impl-inv p mfs dmu state*
and *Linv*: *LLL-invariant-mod fs mfs dmu p b first i*
and *res*: *basis-reduction-mod-add-rows-loop p mfs dmu i j* = (*mfs'*, *dmu'*)
and *res'*: *LLL-add-rows-loop p state i j* = *state'*
and *j*: *j* ≤ *i*
and *i*: *i* < *m*
shows *state-impl-inv p mfs' dmu' state'*
using *assms*(*1*−*5*)

proof (*induct j arbitrary*: *fs mfs dmu state*)
case (*Suc j*)
note *impl* = *Suc*(*2*)
note *Linv* = *Suc*(*3*)
note *res* = *Suc*(*4*)
note *res′* = *Suc*(*5*)
note *IH* = *Suc*(*1*)
from *Suc* **have** *j*: $j < i$ **and** *ji*: $j \leq i$ **by** *auto*
obtain *mfs1 dmu1* **where** *add*: *basis-reduction-mod-add-row p mfs dmu i j* = (*mfs1*, *dmu1*) **by** *force*
note *res* = *res*[*unfolded basis-reduction-mod-add-rows-loop.simps Let-def add split*]
obtain *state1* **where** *add′*: *LLL-add-row p state i j* = *state1* **by** *auto*
note *res′* = *res′*[*unfolded LLL-add-rows-loop.simps Let-def add′*]
note *Linvw* = *LLL-mod-inv-to-weak*[*OF Linv*]
from *LLL-add-row*[*OF impl Linvw add add′ i j*]
have *impl1*: *state-impl-inv p mfs1 dmu1 state1* **.**
from *basis-reduction-mod-add-row*[*OF Linvw add i j*] *Linv j*
obtain *fs1* **where** *Linv1*: *LLL-invariant-mod fs1 mfs1 dmu1 p b first i* **by** *auto*
show *?case* **using** *IH*[*OF impl1 Linv1 res res′ ji*] **.**
qed *auto*

lemma *LLL-add-rows-outer-loop*: **assumes** *impl*: *state-impl-inv p mfs dmu state*
and *Linv*: *LLL-invariant-mod fs mfs dmu p first b m*
and *res*: *basis-reduction-mod-add-rows-outer-loop p mfs dmu i* = (*mfs′*, *dmu′*)
and *res′*: *LLL-add-rows-outer-loop p state i* = *state′*
and *i*: $i \leq m - 1$
shows *state-impl-inv p mfs′ dmu′ state′*
using *assms*
proof (*induct i arbitrary*: *fs mfs dmu state mfs′ dmu′ state′*)
case (*Suc i*)
note *impl* = *Suc*(*2*)
note *Linv* = *Suc*(*3*)
note *res* = *Suc*(*4*)
note *res′* = *Suc*(*5*)
note *i* = *Suc*(*6*)
note *IH* = *Suc*(*1*)
from *i* **have** *im*: $i < m$ $i \leq m - 1$ *Suc i* < *m* **by** *auto*
obtain *mfs1 dmu1* **where** *add*: *basis-reduction-mod-add-rows-outer-loop p mfs dmu i* = (*mfs1*, *dmu1*) **by** *force*
note *res* = *res*[*unfolded basis-reduction-mod-add-rows-outer-loop.simps Let-def add split*]
obtain *state1* **where** *add′*: *LLL-add-rows-outer-loop p state i* = *state1* **by** *auto*
note *res′* = *res′*[*unfolded LLL-add-rows-outer-loop.simps Let-def add′*]
from *IH*[*OF impl Linv add add′ im*(*2*)]
have *impl1*: *state-impl-inv p mfs1 dmu1 state1* **.**
from *basis-reduction-mod-add-rows-outer-loop-inv*[*OF Linv add*[*symmetric*] *im*(*1*)]
obtain *fs1* **where** *Linv1*: *LLL-invariant-mod fs1 mfs1 dmu1 p first b m* **by** *auto*
from *basis-reduction-mod-add-rows-loop-inv′*[*OF Linv1 res im*(*3*)] **obtain** *fs′*
where

Linv': *LLL-invariant-mod fs' mfs' dmu' p first b m* **by** *auto*
from *LLL-add-rows-loop*[*OF impl1 LLL-invariant-mod-to-weak-m-to-i*(*1*)[*OF Linv1*] *res res' le-refl im*(*3*)] *i*
show *?case* **by** *auto*
qed *auto*

5.3 Soundness of implementation

We just prove that the concrete implementations have the same input-output-behaviour as the abstract versions of Storjohann's algorithms.

lemma *LLL-reduce-basis*: *LLL-reduce-basis* = *reduce-basis-mod*
proof (*cases m* = *0*)
case *True*
from *LLL-invD*[*OF reduce-basis-mod-inv*[*OF refl*]] *True*
have *reduce-basis-mod* = [] **by** *auto*
thus *?thesis* **using** *True* **unfolding** *LLL-reduce-basis-def* **by** *auto*
next
case *False*
hence *idm*: (*m* = *0*) = *False* **by** *auto*
let *?first* = *False*
obtain *p1 mfs1 dmu1 g-idx1* **where** *init*: *compute-initial-state ?first* = (*p1*, *mfs1*, *dmu1*,*g-idx1*)
by (*metis prod-cases3*)
obtain *p1' state1 g-idx1'* **where** *init'*: *LLL-initial ?first* = (*p1'*, *state1*, *g-idx1'*)

by (*metis prod.exhaust*)
from *LLL-initial*[*OF init init' False*]
have *impl1*: *state-impl-inv p1 mfs1 dmu1 state1* **and** *id*: *p1'* = *p1 g-idx1'* = *g-idx1* **by** *auto*
from *LLL-initial-invariant-mod*[*OF init*] **obtain** *fs1 b1* **where**
inv1: *LLL-invariant-mod fs1 mfs1 dmu1 p1 ?first b1 0* **by** *auto*
obtain *p2 mfs2 dmu2* **where** *main*: *basis-reduction-mod-main p1 ?first mfs1 dmu1 g-idx1 0 0* = (*p2*, *mfs2*, *dmu2*)
by (*metis prod-cases3*)
from *basis-reduction-mod-main*[*OF inv1 main*] **obtain** *fs2 b2* **where**
inv2: *LLL-invariant-mod fs2 mfs2 dmu2 p2 ?first b2 m* **by** *auto*
obtain *p2' state2* **where** *main'*: *LLL-main p1 ?first state1 g-idx1 0 0* = (*p2'*, *state2*)
by (*metis prod.exhaust*)
from *LLL-main*[*OF impl1 inv1 main*, *unfolded id*, *OF main'*]
have *impl2*: *state-impl-inv p2 mfs2 dmu2 state2* **and** *p2*: *p2'* = *p2* **by** *auto*
obtain *mfs3 dmu3* **where** *outer*: *basis-reduction-mod-add-rows-outer-loop p2 mfs2 dmu2* (*m* − *1*) = (*mfs3*, *dmu3*) **by** *force*
obtain *mfsi3 dmui3 di3 mods3* **where** *outer'*: *LLL-add-rows-outer-loop p2 state2* (*m* − *1*) = (*mfsi3*, *dmui3*, *di3*, *mods3*)
by (*metis prod-cases4*)
from *LLL-add-rows-outer-loop*[*OF impl2 inv2 outer outer' le-refl*]
have *state-impl-inv p2 mfs3 dmu3* (*mfsi3*, *dmui3*, *di3*, *mods3*) .
hence *identity*: *mfs3* = *mfsi3* **unfolding** *state-impl-inv.simps* **by** *auto*

note *res* = *reduce-basis-mod-def*[*unfolded init main split Let-def outer*]
note *res'* = *LLL-reduce-basis-def*[*unfolded init' Let-def main' id split p2 outer' idm if-False*]
show *?thesis* **unfolding** *res res' identity* **..**
qed

lemma *LLL-reduce-basis-iso*: *LLL-reduce-basis-iso* = *reduce-basis-iso*
proof (*cases m* = *0*)
case *True*
from *LLL-invD*[*OF reduce-basis-iso-inv*[*OF refl*]] *True*
have *reduce-basis-iso* = [] **by** *auto*
thus *?thesis* **using** *True* **unfolding** *LLL-reduce-basis-iso-def* **by** *auto*
next
case *False*
hence *idm*: (*m* = *0*) = *False* **by** *auto*
let *?first* = *False*
obtain *p1 mfs1 dmu1 g-idx1* **where** *init*: *compute-initial-state ?first* = (*p1*, *mfs1*, *dmu1*, *g-idx1*)
by (*metis prod-cases3*)
obtain *p1' state1 g-idx1'* **where** *init'*: *LLL-initial ?first* = (*p1'*, *state1*, *g-idx1'*)

by (*metis prod.exhaust*)
from *LLL-initial*[*OF init init' False*]
have *impl1*: *state-impl-inv p1 mfs1 dmu1 state1* **and** *id*: *p1'* = *p1 g-idx1'* = *g-idx1* **by** *auto*
from *LLL-initial-invariant-mod*[*OF init*] **obtain** *fs1 b1* **where**
inv1: *LLL-invariant-mod-weak fs1 mfs1 dmu1 p1 ?first b1*
by (*auto simp*: *LLL-invariant-mod-weak-def LLL-invariant-mod-def*)
obtain *p2 mfs2 dmu2* **where** *main*: *basis-reduction-iso-main p1 ?first mfs1 dmu1 g-idx1 0* = (*p2*, *mfs2*, *dmu2*)
by (*metis prod-cases3*)
from *basis-reduction-iso-main*[*OF inv1 main*] **obtain** *fs2 b2* **where**
inv2: *LLL-invariant-mod fs2 mfs2 dmu2 p2 ?first b2 m* **by** *auto*
obtain *p2' state2* **where** *main'*: *LLL-iso-main p1 ?first state1 g-idx1 0* = (*p2'*, *state2*)
by (*metis prod.exhaust*)
from *LLL-iso-main*[*OF impl1 inv1 main*, *unfolded id*, *OF main'*]
have *impl2*: *state-impl-inv p2 mfs2 dmu2 state2* **and** *p2*: *p2'* = *p2* **by** *auto*
obtain *mfs3 dmu3* **where** *outer*: *basis-reduction-mod-add-rows-outer-loop p2 mfs2 dmu2* (*m* − *1*) = (*mfs3*, *dmu3*) **by** *force*
obtain *mfsi3 dmui3 di3 mods3* **where** *outer'*: *LLL-add-rows-outer-loop p2 state2* (*m* − *1*) = (*mfsi3*, *dmui3*, *di3*, *mods3*)
by (*metis prod-cases4*)
from *LLL-add-rows-outer-loop*[*OF impl2 inv2 outer outer' le-refl*]
have *state-impl-inv p2 mfs3 dmu3* (*mfsi3*, *dmui3*, *di3*, *mods3*) **.**
hence *identity*: *mfs3* = *mfsi3* **unfolding** *state-impl-inv.simps* **by** *auto*
note *res* = *reduce-basis-iso-def*[*unfolded init main split Let-def outer*]
note *res'* = *LLL-reduce-basis-iso-def*[*unfolded init' Let-def main' id split p2 outer' idm if-False*]

show *?thesis* **unfolding** *res res' identity* **..**
qed

lemma *LLL-short-vector*: **assumes** *m*: $m \neq 0$
shows *LLL-short-vector* = *short-vector-mod*
proof −
let *?first* = *True*
obtain *p1 mfs1 dmu1 g-idx1* **where** *init*: *compute-initial-state ?first* = (*p1*, *mfs1*, *dmu1*,*g-idx1*)
by (*metis prod-cases3*)
obtain *p1' state1 g-idx1'* **where** *init'*: *LLL-initial ?first* = (*p1'*, *state1*, *g-idx1'*)

by (*metis prod.exhaust*)
from *LLL-initial*[*OF init init' m*]
have *impl1*: *state-impl-inv p1 mfs1 dmu1 state1* **and** *id*: *p1'* = *p1 g-idx1'* = *g-idx1* **by** *auto*
from *LLL-initial-invariant-mod*[*OF init*] **obtain** *fs1 b1* **where**
inv1: *LLL-invariant-mod fs1 mfs1 dmu1 p1 ?first b1 0* **by** *auto*
obtain *p2 mfs2 dmu2* **where** *main*: *basis-reduction-mod-main p1 ?first mfs1 dmu1 g-idx1 0 0* = (*p2*, *mfs2*, *dmu2*)
by (*metis prod-cases3*)
from *basis-reduction-mod-main*[*OF inv1 main*] **obtain** *fs2 b2* **where**
inv2: *LLL-invariant-mod fs2 mfs2 dmu2 p2 ?first b2 m* **by** *auto*
obtain *p2' mfsi2 dmui2 di2 mods2* **where** *main'*: *LLL-main p1 ?first state1 g-idx1 0 0* = (*p2'*, (*mfsi2*, *dmui2*, *di2*, *mods2*))
by (*metis prod.exhaust*)
from *LLL-main*[*OF impl1 inv1 main*, *unfolded id*, *OF main'*]
have *impl2*: *state-impl-inv p2 mfs2 dmu2* (*mfsi2*, *dmui2*, *di2*, *mods2*) **and** *p2*: *p2'* = *p2* **by** *auto*
hence *identity*: *mfs2* = *mfsi2* **unfolding** *state-impl-inv.simps* **by** *auto*
note *res* = *short-vector-mod-def*[*unfolded init main split Let-def*]
note *res'* = *LLL-short-vector-def*[*unfolded init' Let-def main' id split p2*]
show *?thesis* **unfolding** *res res' identity* **..**
qed

lemma *LLL-short-vector-iso*: **assumes** *m*: $m \neq 0$
shows *LLL-short-vector-iso* = *short-vector-iso*
proof −
let *?first* = *True*
obtain *p1 mfs1 dmu1 g-idx1* **where** *init*: *compute-initial-state ?first* = (*p1*, *mfs1*, *dmu1*,*g-idx1*)
by (*metis prod-cases3*)
obtain *p1' state1 g-idx1'* **where** *init'*: *LLL-initial ?first* = (*p1'*, *state1*, *g-idx1'*)

by (*metis prod.exhaust*)
from *LLL-initial*[*OF init init' m*]
have *impl1*: *state-impl-inv p1 mfs1 dmu1 state1* **and** *id*: *p1'* = *p1 g-idx1'* = *g-idx1* **by** *auto*
from *LLL-initial-invariant-mod*[*OF init*] **obtain** *fs1 b1* **where**

inv1: *LLL-invariant-mod-weak fs1 mfs1 dmu1 p1 ?first b1*
by (*auto simp*: *LLL-invariant-mod-weak-def LLL-invariant-mod-def*)
obtain *p2 mfs2 dmu2* **where** *main*: *basis-reduction-iso-main p1 ?first mfs1 dmu1 g-idx1 0* = (*p2*, *mfs2*, *dmu2*)
by (*metis prod-cases3*)
from *basis-reduction-iso-main*[*OF inv1 main*] **obtain** *fs2 b2* **where**
inv2: *LLL-invariant-mod fs2 mfs2 dmu2 p2 ?first b2 m* **by** *auto*
obtain *p2′ mfsi2 dmui2 di2 mods2* **where** *main′*: *LLL-iso-main p1 ?first state1 g-idx1 0* = (*p2′*, (*mfsi2*, *dmui2*, *di2*, *mods2*))
by (*metis prod.exhaust*)
from *LLL-iso-main*[*OF impl1 inv1 main*, *unfolded id*, *OF main′*]
have *impl2*: *state-impl-inv p2 mfs2 dmu2* (*mfsi2*, *dmui2*, *di2*, *mods2*) **and** *p2*: *p2′* = *p2* **by** *auto*
hence *identity*: *mfs2* = *mfsi2* **unfolding** *state-impl-inv.simps* **by** *auto*
note *res* = *short-vector-iso-def*[*unfolded init main split Let-def*]
note *res′* = *LLL-short-vector-iso-def*[*unfolded init′ Let-def main′ id split p2*]
show *?thesis* **unfolding** *res res′ identity* **..**
qed

end

end

6 Generalization of the statement about the uniqueness of the Hermite normal form

theory *Uniqueness-Hermite*
imports *Hermite.Hermite*
begin

instance *int* :: *bezout-ring-div*
proof qed

lemma *map-matrix-rat-of-int-mult*:
shows *map-matrix rat-of-int* (*A*∗∗*B*) = (*map-matrix rat-of-int A*)∗∗(*map-matrix rat-of-int B*)
unfolding *map-matrix-def matrix-matrix-mult-def* **by** *auto*

lemma *det-map-matrix*:
fixes *A* :: *int*^′*n*::*mod-type*^′*n*::*mod-type*
shows *det* (*map-matrix rat-of-int A*) = *rat-of-int* (*det A*)
unfolding *map-matrix-def* **unfolding** *Determinants.det-def* **by** *auto*

lemma *inv-Z-imp-inv-Q*:
fixes *A* :: *int*^′*n*::*mod-type*^′*n*::*mod-type*
assumes *inv-A*: *invertible A*

shows *invertible* (*map-matrix rat-of-int A*)
proof −
have *is-unit* (*det A*) **using** *inv-A invertible-iff-is-unit* **by** *blast*
hence *is-unit* (*det* (*map-matrix rat-of-int A*))
by (*simp add*: *det-map-matrix dvd-if-abs-eq*)
thus *?thesis* **using** *invertible-iff-is-unit* **by** *blast*
qed

lemma *upper-triangular-Z-eq-Q*:
upper-triangular (*map-matrix rat-of-int A*) = *upper-triangular A*
unfolding *upper-triangular-def* **by** *auto*

lemma *invertible-and-upper-diagonal-not0*:
fixes *H* :: *int*^'*n*::*mod-type*^'*n*::*mod-type*
assumes *inv-H*: *invertible* (*map-matrix rat-of-int H*) **and** *up-H*: *upper-triangular H*
shows *H* $ *i* $ *i* ≠ *0*
proof −
let *?RAT-H* = (*map-matrix rat-of-int H*)
have *up-RAT-H*: *upper-triangular ?RAT-H*
using *up-H* **unfolding** *upper-triangular-def* **by** *auto*
have *is-unit* (*det ?RAT-H*) **using** *inv-H* **using** *invertible-iff-is-unit* **by** *blast*
hence *?RAT-H* $ *i* $ *i* ≠ *0* **using** *inv-H up-RAT-H is-unit-diagonal*
by (*metis not-is-unit-0*)
thus *?thesis* **by** *auto*
qed

lemma *diagonal-least-nonzero*:
fixes *H* :: *int*^'*n*::*mod-type*^'*n*::*mod-type*
assumes *H*: *Hermite associates residues H*
and *inv-H*: *invertible* (*map-matrix rat-of-int H*) **and** *up-H*: *upper-triangular H*
shows (*LEAST n*. *H* $ *i* $ *n* ≠ *0*) = *i*
proof (*rule Least-equality*)
show *H* $ *i* $ *i* ≠ *0* **by** (*rule invertible-and-upper-diagonal-not0*[*OF inv-H up-H*])
fix *y*
assume *Hiy*: *H* $ *i* $ *y* ≠ *0*
show *i* ≤ *y*
using *up-H* **unfolding** *upper-triangular-def*
by (*metis* (*poly-guards-query*) *Hiy not-less*)
qed

lemma *diagonal-in-associates*:
fixes *H* :: *int*^'*n*::*mod-type*^'*n*::*mod-type*
assumes *H*: *Hermite associates residues H*
and *inv-H*: *invertible* (*map-matrix rat-of-int H*) **and** *up-H*: *upper-triangular H*
shows *H* $ *i* $ *i* ∈ *associates*
proof −
have *H* $ *i* $ *i* ≠ *0* **by** (*rule invertible-and-upper-diagonal-not0*[*OF inv-H up-H*])
hence ¬ *is-zero-row i H* **unfolding** *is-zero-row-def is-zero-row-upt-k-def ncols-def*

by *auto*
thus *?thesis* **using** *H* **unfolding** *Hermite-def* **unfolding** *diagonal-least-nonzero*[*OF H inv-H up-H*]
by *auto*
qed

lemma *above-diagonal-in-residues*:
fixes *H* :: *int*^'*n*::*mod-type*^'*n*::*mod-type*
assumes *H*: *Hermite associates residues H*
and *inv-H*: *invertible* (*map-matrix rat-of-int H*) **and** *up-H*: *upper-triangular H*
and *j-i*: *j*<*i*
shows *H* $ *j* $ (*LEAST n*. *H* $ *i* $ *n* ≠ *0*) ∈ *residues* (*H* $ *i* $ (*LEAST n*. *H* $ *i* $ *n* ≠ *0*))
proof −
have *H* $ *i* $ *i* ≠ *0* **by** (*rule invertible-and-upper-diagonal-not0*[*OF inv-H up-H*])
hence ¬ *is-zero-row i H* **unfolding** *is-zero-row-def is-zero-row-upt-k-def ncols-def*
by *auto*
thus *?thesis* **using** *H j-i* **unfolding** *Hermite-def* **unfolding** *diagonal-least-nonzero*[*OF H inv-H up-H*]
by *auto*
qed

lemma *Hermite-unique-generalized*:
fixes *K*::*int*^'*n*::*mod-type*^'*n*::*mod-type*
assumes *A-PH*: *A* = *P* ** *H*
and *A-QK*: *A* = *Q* ** *K*
and *inv-A*: *invertible* (*map-matrix rat-of-int A*)
and *inv-P*: *invertible P*
and *inv-Q*: *invertible Q*
and *H*: *Hermite associates residues H*
and *K*: *Hermite associates residues K*
shows *H* = *K*
proof −
let *?RAT* = *map-matrix rat-of-int*
have *cs-residues*: *Complete-set-residues residues* **using** *H* **unfolding** *Hermite-def*
by *simp*
have *inv-H*: *invertible* (*?RAT H*)
proof −
have *?RAT A* = *?RAT P* ** *?RAT H* **using** *A-PH map-matrix-rat-of-int-mult*
by *blast*
thus *?thesis*
by (*metis inv-A invertible-left-inverse matrix-inv*(*1*) *matrix-mul-assoc*)
qed
have *inv-K*: *invertible* (*?RAT K*)
proof −
have *?RAT A* = *?RAT Q* ** *?RAT K* **using** *A-QK map-matrix-rat-of-int-mult*
by *blast*
thus *?thesis*

by (*metis inv-A invertible-left-inverse matrix-inv*(*1*) *matrix-mul-assoc*)
qed
define *U* **where** *U* = (*matrix-inv P*)∗∗*Q*
have *inv-U*: *invertible U*
by (*metis U-def inv-P inv-Q invertible-def invertible-mult matrix-inv-left matrix-inv-right*)
have *H-UK*: *H* = *U* ∗∗ *K* **using** *A-PH A-QK inv-P*
by (*metis U-def matrix-inv-left matrix-mul-assoc matrix-mul-lid*)
have *Determinants.det K* ∗*k U* = *H* ∗∗ *adjugate K*
unfolding *H-UK matrix-mul-assoc*[*symmetric*] *mult-adjugate-det matrix-mul-mat*
..
have *upper-triangular-H*: *upper-triangular H*
by (*metis H Hermite-def echelon-form-imp-upper-triagular*)
have *upper-triangular-K*: *upper-triangular K*
by (*metis K Hermite-def echelon-form-imp-upper-triagular*)
have *upper-triangular-U*: *upper-triangular U*
proof −
have *U-H-K*: *?RAT U* = (*?RAT H*) ∗∗ (*matrix-inv* (*?RAT K*))
by (*metis H-UK inv-K map-matrix-rat-of-int-mult matrix-inv*(*2*) *matrix-mul-assoc matrix-mul-rid*)
have *up-inv-RAT-K*: *upper-triangular* (*matrix-inv* (*?RAT K*)) **using** *upper-triangular-inverse*
by (*simp add*: *upper-triangular-inverse inv-K upper-triangular-K upper-triangular-Z-eq-Q*)
have *upper-triangular* (*?RAT U*) **unfolding** *U-H-K*
by (*rule upper-triangular-mult*[*OF - up-inv-RAT-K*],
auto simp add: *upper-triangular-H upper-triangular-Z-eq-Q*)
thus *?thesis* **using** *upper-triangular-Z-eq-Q* **by** *auto*
qed
have *unit-det-U*: *is-unit* (*det U*) **by** (*metis inv-U invertible-iff-is-unit*)
have *is-unit-diagonal-U*: (∀ *i*. *is-unit* (*U* $ *i* $ *i*))
by (*rule is-unit-diagonal*[*OF upper-triangular-U unit-det-U*])
have *Uii-1*: (∀ *i*. (*U* $ *i* $ *i*) = *1*) **and** *Hii-Kii*: (∀ *i*. (*H* $ *i* $ *i*) = (*K* $ *i* $ *i*))
proof (*auto*)
fix *i*
have *Hii*: *H* $ *i* $ *i* ∈ *associates*
by (*rule diagonal-in-associates*[*OF H inv-H upper-triangular-H*])
have *Kii*: *K* $ *i* $ *i* ∈ *associates*
by (*rule diagonal-in-associates*[*OF K inv-K upper-triangular-K*])
have *ass-Hii-Kii*: *normalize* (*H* $ *i* $ *i*) = *normalize* (*K* $ *i* $ *i*)
by (*metis H-UK is-unit-diagonal-U normalize-mult-unit-left upper-triangular-K upper-triangular-U upper-triangular-mult-diagonal*)
show *Hii-eq-Kii*: *H* $ *i* $ *i* = *K* $ *i* $ *i*
by (*metis Hermite-def Hii K Kii ass-Hii-Kii in-Ass-not-associated*)
have *H* $ *i* $ *i* = *U* $ *i* $ *i* ∗ *K* $ *i* $ *i*
by (*metis H-UK upper-triangular-K upper-triangular-U upper-triangular-mult-diagonal*)
thus *U* $ *i* $ *i* = *1* **unfolding** *Hii-eq-Kii mult-cancel-right1*
using *inv-K invertible-and-upper-diagonal-not0 upper-triangular-K* **by** *blast*
qed
have *zero-above*: ∀ *j s*. *j*≥*1* ∧ *j* < *ncols A* − *to-nat s* ⟶ *U* $ *s* $ (*s* + *from-nat j*) = *0*

```
 proof (clarify)
  fix j s assume  1 ≤ j and j < ncols A − (to-nat (s::'n))
  thus U $ s $ (s + from-nat j) = 0
  proof (induct j rule: less-induct)
   fix p
   assume induct-step: (⋀y. y < p ⟹ 1 ≤ y ⟹ y < ncols A − to-nat s ⟹
U $ s $ (s + from-nat y) = 0)
    and p1: 1 ≤ p and p2: p < ncols A − to-nat s
   have s-less: s < s + from-nat p using p1 p2 unfolding ncols-def
   by (metis One-nat-def add.commute add-diff-cancel-right' add-lessD1 add-to-nat-def

     from-nat-to-nat-id less-diff-conv neq-iff not-le
     to-nat-from-nat-id to-nat-le zero-less-Suc)
   show U $ s $ (s + from-nat p) = 0
   proof −
    have UNIV-rw: UNIV = insert s (UNIV−{s}) by auto
    have UNIV-s-rw: UNIV−{s} = insert (s + from-nat p) ((UNIV−{s}) −
{s + from-nat p})
     using p1 p2 s-less unfolding ncols-def by (auto simp: algebra-simps)
    have sum-rw: (∑ k∈UNIV−{s}. U $ s $ k * K $ k $ (s + from-nat p))
     = U $ s $ (s + from-nat p) * K $ (s + from-nat p) $ (s + from-nat p)
       + (∑ k∈(UNIV−{s})−{s + from-nat p}. U $ s $ k * K $ k $ (s +
from-nat p))
      using UNIV-s-rw sum.insert by (metis (erased, lifting) Diff-iff finite
singletonI)
    have sum-0: (∑ k∈(UNIV−{s})−{s + from-nat p}. U $ s $ k * K $ k $
(s + from-nat p)) = 0
    proof (rule sum.neutral, rule)
     fix x assume x: x ∈ UNIV − {s} − {s + from-nat p}
     show U $ s $ x * K $ x $ (s + from-nat p) = 0
     proof (cases x<s)
      case True
      thus ?thesis using upper-triangular-U unfolding upper-triangular-def
       by auto
     next
      case False
      hence x-g-s: x>s using x by (metis Diff-iff neq-iff singletonI)
      show ?thesis
      proof (cases x<s+from-nat p)
       case True
       define a where a = to-nat x − to-nat s
       from x-g-s have to-nat s < to-nat x by (rule to-nat-mono)
       hence xa: x=s+(from-nat a) unfolding a-def add-to-nat-def
              by (simp add: less-imp-diff-less to-nat-less-card algebra-simps
to-nat-from-nat-id)
       have U $ s $ x =0
       proof (unfold xa, rule induct-step)
        show a-p: a<p unfolding a-def using p2 unfolding ncols-def
        proof −
```

have $x < \textit{from-nat}\ (\textit{to-nat}\ s + \textit{to-nat}\ (\textit{from-nat}\ p::'n))$
by (*metis* (*no-types*) *True add-to-nat-def*)
hence $\textit{to-nat}\ x - \textit{to-nat}\ s < \textit{to-nat}\ (\textit{from-nat}\ p::'n)$
by (*simp add*: *add.commute less-diff-conv2 less-imp-le to-nat-le x-g-s*)
thus $\textit{to-nat}\ x - \textit{to-nat}\ s < p$
by (*metis* (*no-types*) *from-nat-eq-imp-eq from-nat-to-nat-id le-less-trans less-imp-le not-le to-nat-less-card*)
qed
show $1 \leq a$
by (*auto simp add*: *a-def p1 p2*) (*metis Suc-leI to-nat-mono x-g-s zero-less-diff*)
show $a < \textit{ncols}\ A - \textit{to-nat}\ s$ **using** *a-p p2* **by** *auto*
qed
thus *?thesis* **by** *simp*
next
case *False*
hence $x > s + \textit{from-nat}\ p$ **using** *x-g-s x* **by** *auto*
thus *?thesis* **using** *upper-triangular-K* **unfolding** *upper-triangular-def*
by *auto*
qed
qed
qed
have $H\ \$\ s\ \$\ (s + \textit{from-nat}\ p) = (\sum k{\in}\textit{UNIV}.\ U\ \$\ s\ \$\ k * K\ \$\ k\ \$\ (s + \textit{from-nat}\ p))$
unfolding *H-UK matrix-matrix-mult-def* **by** *auto*
also have $\ldots = (\sum k{\in}\textit{insert}\ s\ (\textit{UNIV}-\{s\}).\ U\ \$\ s\ \$\ k * K\ \$\ k\ \$\ (s + \textit{from-nat}\ p))$
using *UNIV-rw* **by** *simp*
also have $\ldots = U\ \$\ s\ \$\ s * K\ \$\ s\ \$\ (s + \textit{from-nat}\ p)$
$+ (\sum k{\in}\textit{UNIV}-\{s\}.\ U\ \$\ s\ \$\ k * K\ \$\ k\ \$\ (s + \textit{from-nat}\ p))$
by (*rule sum.insert*, *simp-all*)
also have $\ldots = U\ \$\ s\ \$\ s * K\ \$\ s\ \$\ (s + \textit{from-nat}\ p)$
$+ U\ \$\ s\ \$\ (s + \textit{from-nat}\ p) * K\ \$\ (s + \textit{from-nat}\ p)\ \$\ (s + \textit{from-nat}\ p)$
unfolding *sum-rw sum-0* **by** *simp*
finally have *H-s-sp*: $H\ \$\ s\ \$\ (s + \textit{from-nat}\ p)$
$= U\ \$\ s\ \$\ (s + \textit{from-nat}\ p) * K\ \$\ (s + \textit{from-nat}\ p)\ \$\ (s + \textit{from-nat}\ p) + K\ \$\ s\ \$\ (s + \textit{from-nat}\ p)$
using *Uii-1* **by** *auto*
hence *cong-HK*: $\textit{cong}\ (H\ \$\ s\ \$\ (s + \textit{from-nat}\ p))\ (K\ \$\ s\ \$\ (s + \textit{from-nat}\ p))\ (K\ \$\ (s + \textit{from-nat}\ p)\ \$\ (s + \textit{from-nat}\ p))$
unfolding *cong-def* **by** *auto*
have *H-s-sp-residues*: $(H\ \$\ s\ \$\ (s + \textit{from-nat}\ p)) \in \textit{residues}\ (K\ \$\ (s + \textit{from-nat}\ p)\ \$\ (s + \textit{from-nat}\ p))$
using *above-diagonal-in-residues*[*OF H inv-H upper-triangular-H s-less*]
unfolding *diagonal-least-nonzero*[*OF H inv-H upper-triangular-H*]
by (*metis Hii-Kii*)
have *K-s-sp-residues*: $(K\ \$\ s\ \$\ (s + \textit{from-nat}\ p)) \in \textit{residues}\ (K\ \$\ (s + \textit{from-nat}$

```
p) $ (s + from-nat p))
        using above-diagonal-in-residues[OF K inv-K upper-triangular-K s-less]
        unfolding diagonal-least-nonzero[OF K inv-K upper-triangular-K] .
      have Hs-sp-Ks-sp: (H $ s $ (s + from-nat p)) = (K $ s $ (s + from-nat p))

            using cong-HK in-Res-not-congruent[OF cs-residues H-s-sp-residues
K-s-sp-residues]
        by fast
       have K $ (s + from-nat p) $ (s + from-nat p) ≠ 0
       using inv-K invertible-and-upper-diagonal-not0 upper-triangular-K by blast
        thus ?thesis unfolding from-nat-1 using H-s-sp unfolding Hs-sp-Ks-sp
by auto
     qed
   qed
  qed
  have U = mat 1
  proof (unfold mat-def vec-eq-iff, auto)
    fix ia show U $ ia $ ia = 1 using Uii-1 by simp
    fix i assume i-ia: i ≠ ia
    show U $ i $ ia = 0
    proof (cases ia<i)
      case True
       thus ?thesis using upper-triangular-U unfolding upper-triangular-def by
auto
    next
      case False
      hence i-less-ia: i<ia using i-ia by auto
      define a where a = to-nat ia − to-nat i
      have ia-eq: ia = i + from-nat a unfolding a-def
     by (metis i-less-ia a-def add-to-nat-def dual-order.strict-iff-order from-nat-to-nat-id

            le-add-diff-inverse less-imp-diff-less to-nat-from-nat-id to-nat-less-card
to-nat-mono)
      have 1 ≤ a unfolding a-def
        by (metis diff-is-0-eq i-less-ia less-one not-less to-nat-mono)
      moreover have a < ncols A − to-nat i
        unfolding a-def ncols-def
        by (metis False diff-less-mono not-less to-nat-less-card to-nat-mono')
      ultimately show ?thesis using zero-above unfolding ia-eq by blast
    qed
  qed
  thus ?thesis using H-UK matrix-mul-lid by fast
qed

end
```

7 Uniqueness of Hermite normal form in JNF

This theory contains the proof of the uniqueness theorem of the Hermite normal form in JNF, moved from HOL Analysis.

theory *Uniqueness-Hermite-JNF*
imports
Hermite.Hermite
Uniqueness-Hermite
Smith-Normal-Form.SNF-Missing-Lemmas
Smith-Normal-Form.Mod-Type-Connect
Smith-Normal-Form.Finite-Field-Mod-Type-Connection
begin

hide-const (**open**) *residues*

We first define some properties that currently exist in HOL Analysis, but not in JNF, namely a predicate for being in echelon form, another one for being in Hermite normal form, definition of a row of zeros up to a concrete position, and so on.

definition *is-zero-row-upt-k-JNF* :: *nat* => *nat* =>'*a*::{*zero*} *mat* => *bool*
where *is-zero-row-upt-k-JNF i k A* = (∀ *j*. *j* < *k* ⟶ *A* $$ (*i*,*j*) = *0*)

definition *is-zero-row-JNF* :: *nat* =>'*a*::{*zero*} *mat* => *bool*
where *is-zero-row-JNF i A* = (∀ *j*<*dim-col A*. *A* $$ (*i*, *j*) = *0*)

lemma *echelon-form-def'*:
echelon-form A = (
(∀ *i*. *is-zero-row i A* ⟶ ¬ (∃ *j*. *j*>*i* ∧ ¬ *is-zero-row j A*))
∧
(∀ *i j*. *i*<*j* ∧ ¬ (*is-zero-row i A*) ∧ ¬ (*is-zero-row j A*)
⟶ ((*LEAST n*. *A* $ *i* $ *n* ≠ *0*) < (*LEAST n*. *A* $ *j* $ *n* ≠ *0*))))
unfolding *echelon-form-def* *echelon-form-upt-k-def* **unfolding** *is-zero-row-def*
by *auto*

definition
echelon-form-JNF :: '*a*::{*bezout-ring*} *mat* ⇒ *bool*
where
echelon-form-JNF A = (
(∀ *i*<*dim-row A*. *is-zero-row-JNF i A* ⟶ ¬ (∃ *j*. *j* < *dim-row A* ∧ *j*>*i* ∧ ¬ *is-zero-row-JNF j A*))
∧
(∀ *i j*. *i*<*j* ∧ *j*<*dim-row A* ∧ ¬ (*is-zero-row-JNF i A*) ∧ ¬ (*is-zero-row-JNF j A*)
⟶ ((*LEAST n*. *A* $$ (*i*, *n*) ≠ *0*) < (*LEAST n*. *A* $$ (*j*, *n*) ≠ *0*))))

Now, we connect the existing definitions in HOL Analysis to the ones just defined in JNF by means of transfer rules.

context includes *lifting-syntax*

begin

lemma *HMA-is-zero-row-mod-type*[*transfer-rule*]:
(*Mod-Type-Connect.HMA-I*) ===> (*Mod-Type-Connect.HMA-M* :: - ⇒ 'a :: *comm-ring-1* ^ 'n :: *mod-type* ^ 'm :: *mod-type* ⇒ -)
===> (=)) *is-zero-row-JNF is-zero-row*
proof (*intro rel-funI*, *goal-cases*)
case (*1 i i' A A'*)
note *ii'* = *1*(*1*)[*transfer-rule*]
note *AA'* = *1*(*2*)[*transfer-rule*]
have (∀ *j*<*dim-col A*. *A* $$ (*i*, *j*) = *0*) = (∀ *j*. *A'* $*h i'* $*h j* = *0*)
proof (*rule*;*rule*+)
fix *j'*::'n **assume** *Aij-0*: ∀ *j*<*dim-col A*. *A* $$ (*i*, *j*) = *0*
define *j* **where** *j* = *mod-type-class.to-nat j'*
have [*transfer-rule*]: *Mod-Type-Connect.HMA-I j j'* **unfolding** *Mod-Type-Connect.HMA-I-def j-def* **by** *auto*
have *A-ij0'*: *A* $$ (*i*,*j*) = *0* **using** *Aij-0* **unfolding** *j-def*
by (*metis AA' Mod-Type-Connect.HMA-M-def Mod-Type-Connect.from-hma*$_m$*-def*

dim-col-mat(*1*) *mod-type-class.to-nat-less-card*)
hence *index-hma A' i' j'* = *0* **by** *transfer*
thus *A'* $*h i'* $*h j'* = *0* **unfolding** *index-hma-def* **by** *simp*
next
fix *j* **assume** *1*: ∀ *j'*. *A'* $*h i'* $*h j'* = *0* **and** *2*: *j* < *dim-col A*
define *j'*::'n **where** *j'* = *mod-type-class.from-nat j*
have [*transfer-rule*]: *Mod-Type-Connect.HMA-I j j'* **unfolding** *Mod-Type-Connect.HMA-I-def j'-def*
using *Mod-Type.to-nat-from-nat-id*[*of j*, **where** *?'a* = *'n*] *2*
using *AA' Mod-Type-Connect.dim-col-transfer-rule* **by** *force*
have *A'* $*h i'* $*h j'* = *0* **using** *1* **by** *auto*
hence *index-hma A' i' j'* = *0* **unfolding** *index-hma-def* **by** *simp*
thus *A* $$ (*i*, *j*) = *0* **by** *transfer*
qed
thus *?case* **unfolding** *is-zero-row-def' is-zero-row-JNF-def* **by** *auto*
qed

lemma *HMA-echelon-form-mod-type*[*transfer-rule*]:
(*Mod-Type-Connect.HMA-M* :: - ⇒ 'a ::*bezout-ring* ^ 'n :: *mod-type* ^ 'm :: *mod-type* ⇒ -) ===> (=))
echelon-form-JNF echelon-form
proof (*intro rel-funI*, *goal-cases*)
case (*1 A A'*)
note *AA'* = *1*(*1*)[*transfer-rule*]
have *1*: (∀ *i*<*dim-row A*. *is-zero-row-JNF i A* ⟶ ¬ (∃ *j* < *dim-row A*. *j*>*i* ∧ ¬ *is-zero-row-JNF j A*))
= (∀ *i*. *is-zero-row i A'* ⟶ ¬ (∃ *j*>*i*. ¬ *is-zero-row j A'*))
proof (*auto*)
fix *i' j'* **assume** *1*: ∀ *i*<*dim-row A*. *is-zero-row-JNF i A* ⟶ (∀ *j*>*i*. *j* < *dim-row*

$A \longrightarrow$ *is-zero-row-JNF j A*)
and *2*: *is-zero-row i′ A′* **and** *3*: $i' < j'$
let *?i* = *Mod-Type.to-nat i′*
let *?j* = *Mod-Type.to-nat j′*
have *ii′*[*transfer-rule*]: *Mod-Type-Connect.HMA-I ?i i′* **and** *jj′*[*transfer-rule*]: *Mod-Type-Connect.HMA-I ?j j′*
unfolding *Mod-Type-Connect.HMA-I-def* **by** *auto*
have *is-zero-row-JNF ?i A* **using** *2* **by** *transfer′*
hence *is-zero-row-JNF ?j A* **using** *1 3 to-nat-mono*
by (*metis AA′ Mod-Type-Connect.HMA-M-def Mod-Type-Connect.from-hma*$_m$*-def dim-row-mat*(*1*) *mod-type-class.to-nat-less-card*)
thus *is-zero-row j′ A′* **by** *transfer′*
next
fix *i j* **assume** *1*: $\forall i'$. *is-zero-row i′ A′* $\longrightarrow$ ($\forall j' > i'$. *is-zero-row j′ A′*)
and *2*: *is-zero-row-JNF i A* **and** *3*: $i < j$ **and** *4*: $j <$ *dim-row A*
let *?i′* = *Mod-Type.from-nat i*::′*m*
let *?j′* = *Mod-Type.from-nat j*::′*m*
have *ii′*[*transfer-rule*]: *Mod-Type-Connect.HMA-I i ?i′*
unfolding *Mod-Type-Connect.HMA-I-def* **using** *Mod-Type.to-nat-from-nat-id*[*of i*]
using *3 4 AA′ Mod-Type-Connect.dim-row-transfer-rule less-trans* **by** *fastforce*
have *jj′*[*transfer-rule*]: *Mod-Type-Connect.HMA-I j ?j′*
unfolding *Mod-Type-Connect.HMA-I-def* **using** *Mod-Type.to-nat-from-nat-id*[*of j*]
using *3 4 AA′ Mod-Type-Connect.dim-row-transfer-rule less-trans* **by** *fastforce*
have *is-zero-row ?i′ A′* **using** *2* **by** *transfer*
moreover have $?i' < ?j'$ **using** *3 4 AA′ Mod-Type-Connect.dim-row-transfer-rule from-nat-mono* **by** *fastforce*
ultimately have *is-zero-row ?j′ A′* **using** *1 3* **by** *auto*
thus *is-zero-row-JNF j A* **by** *transfer*
qed
have *2*: (($\forall i\ j.\ i<j \wedge \neg$ (*is-zero-row i A′*) $\wedge \neg$ (*is-zero-row j A′*)
$\longrightarrow$ ((*LEAST n. A′ $h i $h n* $\neq 0$) $<$ (*LEAST n. A′ $h j $h n* $\neq 0$))))
$=$ ($\forall i\ j.\ i<j \wedge j <$ *dim-row A* $\wedge \neg$ (*is-zero-row-JNF i A*) $\wedge \neg$ (*is-zero-row-JNF j A*)
$\longrightarrow$ ((*LEAST n. A $$ (i, n)* $\neq 0$) $<$ (*LEAST n. A $$ (j, n)* $\neq 0$)))
proof (*auto*)
fix *i j* **assume** *1*: $\forall i'\ j'.\ i' < j' \wedge \neg$ *is-zero-row i′ A′* $\wedge \neg$ *is-zero-row j′ A′*
$\longrightarrow$ (*LEAST n′. A′ $h i′ $h n′* $\neq 0$) $<$ (*LEAST n′. A′ $h j′ $h n′* $\neq 0$)
and *ij*: $i < j$ **and** *j*: $j <$ *dim-row A* **and** *i0*: $\neg$ *is-zero-row-JNF i A*
and *j0*: $\neg$ *is-zero-row-JNF j A*
let *?i′* = *Mod-Type.from-nat i*::′*m*
let *?j′* = *Mod-Type.from-nat j*::′*m*
have *ii′*[*transfer-rule*]: *Mod-Type-Connect.HMA-I i ?i′*
unfolding *Mod-Type-Connect.HMA-I-def* **using** *Mod-Type.to-nat-from-nat-id*[*of i*]
using *ij j AA′ Mod-Type-Connect.dim-row-transfer-rule less-trans* **by** *fastforce*
have *jj′*[*transfer-rule*]: *Mod-Type-Connect.HMA-I j ?j′*
unfolding *Mod-Type-Connect.HMA-I-def* **using** *Mod-Type.to-nat-from-nat-id*[*of*

```
j]
    using ij j AA' Mod-Type-Connect.dim-row-transfer-rule less-trans by fastforce
   have i'0: ¬ is-zero-row ?i' A' using i0 by transfer
   have j'0: ¬ is-zero-row ?j' A' using j0 by transfer
   have i'j': ?i' < ?j'
     using AA' Mod-Type-Connect.dim-row-transfer-rule from-nat-mono ij j by
fastforce
   have l1l2: (LEAST n'. A' $h ?i' $h n' ≠ 0) < (LEAST n'. A' $h ?j' $h n' ≠
0)
    using 1 i'0 j'0 i'j' by auto
   define l1 where l1 = (LEAST n'. A' $h ?i' $h n' ≠ 0)
   define l2 where l2 = (LEAST n'. A' $h ?j' $h n' ≠ 0)
   let ?least-n1 = Mod-Type.to-nat l1
   let ?least-n2 = Mod-Type.to-nat l2
  have l1[transfer-rule]: Mod-Type-Connect.HMA-I ?least-n1 l1 and [transfer-rule]:
Mod-Type-Connect.HMA-I ?least-n2 l2
    unfolding Mod-Type-Connect.HMA-I-def by auto
   have (LEAST n. A $$ (i, n) ≠ 0) = ?least-n1
   proof (rule Least-equality)
    obtain n' where n'1: A $$ (i,n') ≠ 0 and n'2: n'<dim-col A
     using i0 unfolding is-zero-row-JNF-def by auto
    let ?n' = Mod-Type.from-nat n'::'n
    have n'n'[transfer-rule]: Mod-Type-Connect.HMA-I n' ?n'
    unfolding Mod-Type-Connect.HMA-I-def using Mod-Type.to-nat-from-nat-id
n'2
     using AA' Mod-Type-Connect.dim-col-transfer-rule by fastforce
    have index-hma A' ?i' ?n' ≠ 0 using n'1 by transfer
    hence A'i'n': A' $h ?i' $h ?n' ≠ 0 unfolding index-hma-def by simp
     have least-le-n': (LEAST n. A $$ (i, n) ≠ 0) ≤ n' by (simp add: Least-le
n'1)
    have l1-le-n': l1 ≤ ?n' by (simp add: A'i'n' Least-le l1-def)
    have A $$ (i, ?least-n1) = index-hma A' ?i' l1 by (transfer, simp)
   also have ... = A' $h mod-type-class.from-nat i $h l1 unfolding index-hma-def
by simp
    also have ... ≠ 0 unfolding l1-def by (metis (mono-tags, lifting) LeastI i'0
is-zero-row-def')
    finally show A $$ (i, mod-type-class.to-nat l1) ≠ 0 .
    fix y assume Aiy: A $$ (i, y) ≠ 0
    let ?y' = Mod-Type.from-nat y::'n
    show Mod-Type.to-nat l1 ≤ y
    proof (cases y≤n')
     case True
     hence y: y < dim-col A using n'2 by auto
    have yy'[transfer-rule]: Mod-Type-Connect.HMA-I y ?y' unfolding Mod-Type-Connect.HMA-I-def
       apply (rule Mod-Type.to-nat-from-nat-id[symmetric])
       using y Mod-Type-Connect.dim-col-transfer-rule[OF AA'] by auto
     have Mod-Type.to-nat l1 ≤ Mod-Type.to-nat ?y'
     proof (rule to-nat-mono')
      have index-hma A' ?i' ?y' ≠ 0 using Aiy by transfer
```

```
        hence A' $h ?i' $h ?y' ≠ 0 unfolding index-hma-def by simp
        thus l1 ≤ ?y' unfolding l1-def by (simp add: Least-le)
      qed
        then show ?thesis by (metis Mod-Type-Connect.HMA-I-def yy')
      next
        case False
        hence n' < y by auto
        then show ?thesis
          by (metis False Mod-Type-Connect.HMA-I-def dual-order.trans l1-le-n'
linear n'n' to-nat-mono')
      qed
    qed
    moreover have (LEAST n. A $$ (j, n) ≠ 0) = ?least-n2
    proof (rule Least-equality)
      obtain n' where n'1: A $$ (j,n') ≠ 0 and n'2: n'<dim-col A
      using j0 unfolding is-zero-row-JNF-def by auto
    let ?n' = Mod-Type.from-nat n'::'n
    have n'n'[transfer-rule]: Mod-Type-Connect.HMA-I n' ?n'
    unfolding Mod-Type-Connect.HMA-I-def using Mod-Type.to-nat-from-nat-id
n'2
      using AA' Mod-Type-Connect.dim-col-transfer-rule by fastforce
    have index-hma A' ?j' ?n' ≠ 0 using n'1 by transfer
    hence A'i'n': A' $h ?j' $h ?n' ≠ 0 unfolding index-hma-def by simp
     have least-le-n': (LEAST n. A $$ (j, n) ≠ 0) ≤ n' by (simp add: Least-le
n'1)
    have l1-le-n': l2 ≤ ?n' by (simp add: A'i'n' Least-le l2-def)
    have A $$ (j, ?least-n2) = index-hma A' ?j' l2 by (transfer, simp)
    also have ... = A' $h ?j' $h l2 unfolding index-hma-def by simp
    also have ... ≠ 0 unfolding l2-def by (metis (mono-tags, lifting) LeastI j'0
is-zero-row-def')
    finally show A $$ (j, mod-type-class.to-nat l2) ≠ 0 .
    fix y assume Aiy: A $$ (j, y) ≠ 0
    let ?y' = Mod-Type.from-nat y::'n
    show Mod-Type.to-nat l2 ≤ y
    proof (cases y≤n')
      case True
      hence y: y < dim-col A using n'2 by auto
    have yy'[transfer-rule]: Mod-Type-Connect.HMA-I y ?y' unfolding Mod-Type-Connect.HMA-I-def
        apply (rule Mod-Type.to-nat-from-nat-id[symmetric])
        using y Mod-Type-Connect.dim-col-transfer-rule[OF AA'] by auto
      have Mod-Type.to-nat l2 ≤ Mod-Type.to-nat ?y'
      proof (rule to-nat-mono')
        have index-hma A' ?j' ?y' ≠ 0 using Aiy by transfer
        hence A' $h ?j' $h ?y' ≠ 0 unfolding index-hma-def by simp
        thus l2 ≤ ?y' unfolding l2-def by (simp add: Least-le)
      qed
        then show ?thesis by (metis Mod-Type-Connect.HMA-I-def yy')
      next
        case False
```

hence $n' < y$ **by** *auto*
then show *?thesis*
by (*metis False Mod-Type-Connect.HMA-I-def dual-order.trans l1-le-n' linear n'n' to-nat-mono'*)
qed
qed
ultimately show (*LEAST n. A $$ (i, n) ≠ 0*) < (*LEAST n. A $$ (j, n) ≠ 0*)
using *l1l2* **unfolding** *l1-def l2-def* **by** (*simp add*: *to-nat-mono*)
next
fix *i' j'* **assume** *1*: ∀ *i j. i < j ∧ j < dim-row A ∧ ¬ is-zero-row-JNF i A ∧ ¬ is-zero-row-JNF j A*
⟶ (*LEAST n. A $$ (i, n) ≠ 0*) < (*LEAST n. A $$ (j, n) ≠ 0*)
and *i'j'*: *i' < j'* **and** *i'*: ¬ *is-zero-row i' A'* **and** *j'*: ¬ *is-zero-row j' A'*
let *?i = Mod-Type.to-nat i'*
let *?j = Mod-Type.to-nat j'*
have [*transfer-rule*]: *Mod-Type-Connect.HMA-I ?i i'*
and [*transfer-rule*]: *Mod-Type-Connect.HMA-I ?j j'*
unfolding *Mod-Type-Connect.HMA-I-def* **by** *auto*
have *i*: ¬ *is-zero-row-JNF ?i A* **using** *i'* **by** *transfer'*
have *j*: ¬ *is-zero-row-JNF ?j A* **using** *j'* **by** *transfer'*
have *ij*: *?i < ?j* **using** *i'j' to-nat-mono* **by** *blast*
have *j-dim-row*: *?j < dim-row A*
using *AA' Mod-Type-Connect.dim-row-transfer-rule mod-type-class.to-nat-less-card* **by** *fastforce*
have *least-ij*: (*LEAST n. A $$ (?i, n) ≠ 0*) < (*LEAST n. A $$ (?j, n) ≠ 0*)
using *i j ij j-dim-row 1* **by** *auto*
define *l1* **where** *l1* = (*LEAST n'. A $$ (?i, n') ≠ 0*)
define *l2* **where** *l2* = (*LEAST n'. A $$ (?j, n') ≠ 0*)
let *?least-n1 = Mod-Type.from-nat l1*::*'n*
let *?least-n2 = Mod-Type.from-nat l2*::*'n*
have *l1-dim-col*: *l1 < dim-col A*
by (*smt is-zero-row-JNF-def j l1-def leI le-less-trans least-ij less-trans not-less-Least*)
have *l2-dim-col*: *l2 < dim-col A*
by (*metis* (*mono-tags*, *lifting*) *Least-le is-zero-row-JNF-def j l2-def le-less-trans*)
have [*transfer-rule*]: *Mod-Type-Connect.HMA-I l1 ?least-n1* **unfolding** *Mod-Type-Connect.HMA-I-def*
using *AA' Mod-Type-Connect.dim-col-transfer-rule l1-dim-col Mod-Type.to-nat-from-nat-id*
by *fastforce*
have [*transfer-rule*]: *Mod-Type-Connect.HMA-I l2 ?least-n2* **unfolding** *Mod-Type-Connect.HMA-I-def*
using *AA' Mod-Type-Connect.dim-col-transfer-rule l2-dim-col Mod-Type.to-nat-from-nat-id*
by *fastforce*
have (*LEAST n. A' $h i' $h n ≠ 0*) = *?least-n1*
proof (*rule Least-equality*)
obtain *n'* **where** *n'1*: *A' $h i' $h n' ≠ 0* **using** *i'* **unfolding** *is-zero-row-def'* **by** *auto*
have *A' $h i' $h ?least-n1 = index-hma A' i' ?least-n1* **unfolding** *index-hma-def* **by** *simp*
also have ... = *A$$ (?i, l1)* **by** (*transfer*, *simp*)

also have ... $\neq$ *0* **by** (*metis* (*mono-tags*, *lifting*) *LeastI i is-zero-row-JNF-def l1-def*)
finally show *A' $h i' $h ?least-n1* $\neq$ *0* **.**
next
fix *y* **assume** *y*: *A' $h i' $h y* $\neq$ *0*
let *?y'* = *Mod-Type.to-nat y*
have [*transfer-rule*]: *Mod-Type-Connect.HMA-I ?y' y* **unfolding** *Mod-Type-Connect.HMA-I-def* **by** *simp*
have *?least-n1* $\leq$ *Mod-Type.from-nat ?y'*
proof (*unfold l1-def*, *rule from-nat-mono'*)
show *Mod-Type.to-nat y* < *CARD('n)* **by** (*simp add*: *mod-type-class.to-nat-less-card*)
have $*$: *A $$* (*mod-type-class.to-nat i'*, *mod-type-class.to-nat y*) $\neq$ *0*
using *y*[*unfolded index-hma-def*[*symmetric*]] **by** *transfer'*
show (*LEAST n'*. *A $$* (*mod-type-class.to-nat i'*, *n'*) $\neq$ *0*) $\leq$ *mod-type-class.to-nat y*
by (*rule Least-le*, *simp add*: $*$)
qed
also have ... = *y* **by** *simp*
finally show *?least-n1* $\leq$ *y* **.**
qed
moreover have (*LEAST n*. *A' $h j' $h n* $\neq$ *0*) = *?least-n2*
proof (*rule Least-equality*)
obtain *n'* **where** *n'1*: *A' $h j' $h n'* $\neq$ *0* **using** *j'* **unfolding** *is-zero-row-def'* **by** *auto*
have *A' $h j' $h ?least-n2* = *index-hma A' j' ?least-n2* **unfolding** *index-hma-def* **by** *simp*
also have ... = *A$$* (*?j*, *l2*) **by** (*transfer*, *simp*)
also have ... $\neq$ *0* **by** (*metis* (*mono-tags*, *lifting*) *LeastI j is-zero-row-JNF-def l2-def*)
finally show *A' $h j' $h ?least-n2* $\neq$ *0* **.**
next
fix *y* **assume** *y*: *A' $h j' $h y* $\neq$ *0*
let *?y'* = *Mod-Type.to-nat y*
have [*transfer-rule*]: *Mod-Type-Connect.HMA-I ?y' y* **unfolding** *Mod-Type-Connect.HMA-I-def* **by** *simp*
have *?least-n2* $\leq$ *Mod-Type.from-nat ?y'*
proof (*unfold l2-def*, *rule from-nat-mono'*)
show *Mod-Type.to-nat y* < *CARD('n)* **by** (*simp add*: *mod-type-class.to-nat-less-card*)
have $*$: *A $$* (*mod-type-class.to-nat j'*, *mod-type-class.to-nat y*) $\neq$ *0*
using *y*[*unfolded index-hma-def*[*symmetric*]] **by** *transfer'*
show (*LEAST n'*. *A $$* (*mod-type-class.to-nat j'*, *n'*) $\neq$ *0*) $\leq$ *mod-type-class.to-nat y*
by (*rule Least-le*, *simp add*: $*$)
qed
also have ... = *y* **by** *simp*
finally show *?least-n2* $\leq$ *y* **.**
qed
ultimately show (*LEAST n*. *A' $h i' $h n* $\neq$ *0*) < (*LEAST n*. *A' $h j' $h n* $\neq$ *0*) **using** *least-ij*

unfolding *l1-def l2-def*
using *AA′ Mod-Type-Connect.dim-col-transfer-rule from-nat-mono l2-def l2-dim-col*
by *fastforce*
qed
show *?case* **unfolding** *echelon-form-JNF-def echelon-form-def′* **using** *1 2* **by** *auto*
qed

definition *Hermite-JNF* :: *'a*::{*bezout-ring-div*,*normalization-semidom*} *set* ⇒ (*'a* ⇒ *'a set*) ⇒ *'a mat* ⇒ *bool*
where *Hermite-JNF associates residues A* = (
Complete-set-non-associates associates ∧ (*Complete-set-residues residues*) ∧ *echelon-form-JNF A*
∧ (∀ *i*<*dim-row A*. ¬ *is-zero-row-JNF i A* ⟶ *A* $$ (*i*, *LEAST n*. *A* $$ (*i*, *n*) ≠ *0*) ∈ *associates*)
∧ (∀ *i*<*dim-row A*. ¬ *is-zero-row-JNF i A* ⟶ (∀ *j*. *j*<*i* ⟶ *A* $$ (*j*, (*LEAST n*. *A* $$ (*i*, *n*) ≠ *0*))
∈ *residues* (*A* $$ (*i*,(*LEAST n*. *A* $$ (*i*,*n*) ≠ *0*)))
)))

lemma *HMA-LEAST*[*transfer-rule*]:
assumes *AA′*: (*Mod-Type-Connect.HMA-M* :: - ⇒ *'a* :: *comm-ring-1* ^ *'n* :: *mod-type* ^ *'m* :: *mod-type* ⇒ -) *A A′*
and *ii′*: *Mod-Type-Connect.HMA-I i i′* **and** *zero-i*: ¬ *is-zero-row-JNF i A*
shows *Mod-Type-Connect.HMA-I* (*LEAST n*. *A* $$ (*i*, *n*) ≠ *0*) (*LEAST n*. *index-hma A′ i′ n* ≠ *0*)
proof −
define *l* **where** *l* = (*LEAST n′*. *A′* $*h i′* $*h n′* ≠ *0*)
let *?least-n2* = *Mod-Type.to-nat l*
note *AA′*[*transfer-rule*] *ii′*[*transfer-rule*]
have [*transfer-rule*]: *Mod-Type-Connect.HMA-I ?least-n2 l*
by (*simp add*: *Mod-Type-Connect.HMA-I-def*)
have *zero-i′*: ¬ *is-zero-row i′ A′* **using** *zero-i* **by** *transfer*
have (*LEAST n*. *A* $$ (*i*, *n*) ≠ *0*) = *?least-n2*
proof (*rule Least-equality*)
obtain *n′* **where** *n′1*: *A* $$ (*i*,*n′*) ≠ *0* **and** *n′2*: *n′*<*dim-col A*
using *zero-i* **unfolding** *is-zero-row-JNF-def* **by** *auto*
let *?n′* = *Mod-Type.from-nat n′*::*'n*
have *n′n′*[*transfer-rule*]: *Mod-Type-Connect.HMA-I n′ ?n′*
unfolding *Mod-Type-Connect.HMA-I-def* **using** *Mod-Type.to-nat-from-nat-id n′2*
using *AA′ Mod-Type-Connect.dim-col-transfer-rule* **by** *fastforce*
have *index-hma A′ i′ ?n′* ≠ *0* **using** *n′1* **by** *transfer*
hence *A′i′n′*: *A′* $*h i′* $*h ?n′* ≠ *0* **unfolding** *index-hma-def* **by** *simp*
have *least-le-n′*: (*LEAST n*. *A* $$ (*i*, *n*) ≠ *0*) ≤ *n′* **by** (*simp add*: *Least-le n′1*)

```
    have l1-le-n': l ≤ ?n' by (simp add: A'i'n' Least-le l-def)
    have A $$ (i, ?least-n2) = index-hma A' i' l by (transfer, simp)
    also have ... = A' $h i' $h l unfolding index-hma-def by simp
    also have ... ≠ 0 unfolding l-def by (metis (mono-tags) A'i'n' LeastI)
    finally show A $$ (i, mod-type-class.to-nat l) ≠ 0 .
    fix y assume Aiy: A $$ (i, y) ≠ 0
    let ?y' = Mod-Type.from-nat y::'n
    show Mod-Type.to-nat l ≤ y
    proof (cases y≤n')
      case True
      hence y: y < dim-col A using n'2 by auto
     have yy'[transfer-rule]: Mod-Type-Connect.HMA-I y ?y' unfolding Mod-Type-Connect.HMA-I-def
        apply (rule Mod-Type.to-nat-from-nat-id[symmetric])
        using y Mod-Type-Connect.dim-col-transfer-rule[OF AA'] by auto
      have Mod-Type.to-nat l ≤ Mod-Type.to-nat ?y'
      proof (rule to-nat-mono')
        have index-hma A' i' ?y' ≠ 0 using Aiy by transfer
        hence A' $h i' $h ?y' ≠ 0 unfolding index-hma-def by simp
        thus l ≤ ?y' unfolding l-def by (simp add: Least-le)
      qed
        then show ?thesis by (metis Mod-Type-Connect.HMA-I-def yy')
      next
        case False
        hence n' < y by auto
        then show ?thesis
          by (metis False Mod-Type-Connect.HMA-I-def dual-order.trans l1-le-n'
linear n'n' to-nat-mono')
      qed
     qed
      thus ?thesis unfolding Mod-Type-Connect.HMA-I-def l-def index-hma-def
by auto
qed


lemma element-least-not-zero-eq-HMA-JNF:
  fixes A'::'a::comm-ring-1 ^'n::mod-type ^'m::mod-type
  assumes AA': Mod-Type-Connect.HMA-M A A' and jj': Mod-Type-Connect.HMA-I
j j'
    and ii': Mod-Type-Connect.HMA-I i i' and zero-i': ¬ is-zero-row i' A'
  shows A $$ (j, LEAST n. A $$ (i, n) ≠ 0) = A' $h j' $h (LEAST n. A' $h i'
$h n ≠ 0)
proof −
  note AA'[transfer-rule] jj'[transfer-rule] ii'[transfer-rule]
   have [transfer-rule]: Mod-Type-Connect.HMA-I (LEAST n. A $$ (i, n) ≠ 0)
(LEAST n. index-hma A' i' n ≠ 0)
    by (rule HMA-LEAST[OF AA' ii'], insert zero-i', transfer, simp)
  have A' $h j' $h (LEAST n. A' $h i' $h n ≠ 0) = index-hma A' j' (LEAST n.
index-hma A' i' n ≠ 0)
    unfolding index-hma-def by simp
```

```
  also have ... = A $$ (j, LEAST n. A $$ (i, n) ≠ 0) by (transfer', simp)
  finally show ?thesis by simp
qed


lemma HMA-Hermite[transfer-rule]:
 shows ((Mod-Type-Connect.HMA-M :: - ⇒ 'a :: {bezout-ring-div,normalization-semidom}
^ 'n :: mod-type ^ 'm :: mod-type ⇒ -) ===> (=))
  (Hermite-JNF associates residues) (Hermite associates residues)
proof (intro rel-funI, goal-cases)
  case (1 A A')
  note AA' = 1(1)[transfer-rule]
  have 1: echelon-form A' = echelon-form-JNF A by (transfer, simp)
  have 2: (∀ i<dim-row A. ¬ is-zero-row-JNF i A ⟶ A $$ (i, LEAST n. A $$
(i, n) ≠ 0) ∈ associates) =
  (∀ i. ¬ is-zero-row i A' ⟶ A' $h i $h (LEAST n. A' $h i $h n ≠ 0) ∈ associates)
(is ?lhs = ?rhs)
  proof
    assume lhs: ?lhs
    show ?rhs
    proof (rule allI, rule impI)
      fix i' assume zero-i': ¬ is-zero-row i' A'
      let ?i = Mod-Type.to-nat i'
     have ii'[transfer-rule]: Mod-Type-Connect.HMA-I ?i i' unfolding Mod-Type-Connect.HMA-I-def
by simp
      have [simp]: ?i < dim-row A using Mod-Type.to-nat-less-card[of i']
        using AA' Mod-Type-Connect.dim-row-transfer-rule by fastforce
      have zero-i: ¬ is-zero-row-JNF ?i A using zero-i' by transfer
       have [transfer-rule]: Mod-Type-Connect.HMA-I (LEAST n. A $$ (?i, n) ≠
0) (LEAST n. index-hma A' i' n ≠ 0)
        by (rule HMA-LEAST[OF AA' ii'], insert zero-i', transfer, simp)
      have A' $h i' $h (LEAST n. A' $h i' $h n ≠ 0) = A $$ (?i, LEAST n. A $$
(?i, n) ≠ 0)
       by (rule element-least-not-zero-eq-HMA-JNF[OF AA' ii' ii' zero-i', symmet-
ric])
      also have ... ∈ associates using lhs zero-i by simp
      finally show A' $h i' $h (LEAST n. A' $h i' $h n ≠ 0) ∈ associates .
    qed
  next
    assume rhs: ?rhs
    show ?lhs
    proof (rule allI, rule impI, rule impI)
      fix i assume zero-i: ¬ is-zero-row-JNF i A and i: i < dim-row A
      let ?i' = Mod-Type.from-nat i :: 'm
     have ii'[transfer-rule]: Mod-Type-Connect.HMA-I i ?i' unfolding Mod-Type-Connect.HMA-I-def
      using Mod-Type.to-nat-from-nat-id AA' Mod-Type-Connect.dim-row-transfer-rule
i by fastforce
      have zero-i': ¬ is-zero-row ?i' A' using zero-i by transfer
      have A $$ (i, LEAST n. A $$ (i, n) ≠ 0) = A' $h ?i' $h (LEAST n. A' $h
```

```
?i' $h n ≠ 0)
      by (rule element-least-not-zero-eq-HMA-JNF[OF AA' ii' ii' zero-i'])
    also have ... ∈ associates using rhs zero-i' i by simp
    finally show A $$ (i, LEAST n. A $$ (i, n) ≠ 0) ∈ associates .
   qed
  qed
  have 3: (∀ i<dim-row A. ¬ is-zero-row-JNF i A ⟶ (∀ j<i. A $$ (j, LEAST n.
A $$ (i, n) ≠ 0)
          ∈ residues (A $$ (i, LEAST n. A $$ (i, n) ≠ 0)))) =
          (∀ i. ¬ is-zero-row i A' ⟶ (∀ j<i. A' $h j $h (LEAST n. A' $h i $h n
≠ 0)
          ∈ residues (A' $h i $h (LEAST n. A' $h i $h n ≠ 0)))) (is ?lhs = ?rhs)
  proof
   assume lhs: ?lhs
   show ?rhs
   proof (rule allI, rule impI, rule allI, rule impI)
    fix i' j' :: 'm
    assume zero-i': ¬ is-zero-row i' A' and j'i': j' < i'
    let ?i = Mod-Type.to-nat i'
   have ii'[transfer-rule]: Mod-Type-Connect.HMA-I ?i i' unfolding Mod-Type-Connect.HMA-I-def
by simp
    have i: ?i < dim-row A
    using AA' Mod-Type-Connect.dim-row-transfer-rule mod-type-class.to-nat-less-card
     by fastforce
    have zero-i: ¬ is-zero-row-JNF ?i A using zero-i' by transfer'
    let ?j = Mod-Type.to-nat j'
   have jj'[transfer-rule]: Mod-Type-Connect.HMA-I ?j j' unfolding Mod-Type-Connect.HMA-I-def
by simp
    have ji: ?j<?i using j'i' to-nat-mono by blast
    have eq1: A $$ (?j, LEAST n. A $$ (?i, n) ≠ 0) = A' $h j' $h (LEAST n.
A' $h i' $h n ≠ 0)
      by (rule element-least-not-zero-eq-HMA-JNF[OF AA' jj' ii' zero-i'])
    have eq2: A $$ (?i, LEAST n. A $$ (?i, n) ≠ 0) = A' $h i' $h (LEAST n.
A' $h i' $h n ≠ 0)
      by (rule element-least-not-zero-eq-HMA-JNF[OF AA' ii' ii' zero-i'])
     show A' $h j' $h (LEAST n. A' $h i' $h n ≠ 0) ∈ residues (A' $h i' $h
(LEAST n. A' $h i' $h n ≠ 0))
      using lhs eq1 eq2 ji i zero-i by fastforce
   qed
  next
   assume rhs: ?rhs
   show ?lhs
   proof (safe)
    fix i j assume i: i < dim-row A and zero-i: ¬ is-zero-row-JNF i A and ji: j
< i
    let ?i' = Mod-Type.from-nat i :: 'm
   have ii'[transfer-rule]: Mod-Type-Connect.HMA-I i ?i' unfolding Mod-Type-Connect.HMA-I-def
    using Mod-Type.to-nat-from-nat-id AA' Mod-Type-Connect.dim-row-transfer-rule
i by fastforce
```

```
    have zero-i': ¬ is-zero-row ?i' A' using zero-i by transfer
    let ?j' = Mod-Type.from-nat j :: 'm
      have j'i': ?j' < ?i' using AA' Mod-Type-Connect.dim-row-transfer-rule
from-nat-mono i ji
      by fastforce
   have jj'[transfer-rule]: Mod-Type-Connect.HMA-I j ?j' unfolding Mod-Type-Connect.HMA-I-def
      using Mod-Type.to-nat-from-nat-id[of j, where ?'a='m] AA'
        Mod-Type-Connect.dim-row-transfer-rule[OF AA'] j'i' i ji by auto
    have zero-i': ¬ is-zero-row ?i' A' using zero-i by transfer
    have eq1: A $$ (j, LEAST n. A $$ (i, n) ≠ 0) = A' $h ?j' $h (LEAST n.
A' $h ?i' $h n ≠ 0)
      by (rule element-least-not-zero-eq-HMA-JNF[OF AA' jj' ii' zero-i'])
    have eq2: A $$ (i, LEAST n. A $$ (i, n) ≠ 0) = A' $h ?i' $h (LEAST n.
A' $h ?i' $h n ≠ 0)
      by (rule element-least-not-zero-eq-HMA-JNF[OF AA' ii' ii' zero-i'])
    show A $$ (j, LEAST n. A $$ (i, n) ≠ 0) ∈ residues (A $$ (i, LEAST n. A
$$ (i, n) ≠ 0))
      using rhs eq1 eq2 j'i' i zero-i' by fastforce
   qed
  qed
  show Hermite-JNF associates residues A = Hermite associates residues A'
   unfolding Hermite-def Hermite-JNF-def
   using 1 2 3 by auto
qed


corollary HMA-Hermite2[transfer-rule]:
  shows ((=) ===> (=) ===> (Mod-Type-Connect.HMA-M :: -
 ⇒ 'a :: {bezout-ring-div,normalization-semidom} ^'n :: mod-type ^'m :: mod-type
⇒ -) ===> (=))
  (Hermite-JNF) (Hermite)
  by (simp add: HMA-Hermite rel-funI)
```

Once the definitions of both libraries are connected, we start to move the theorem about the uniqueness of the Hermite normal form (stated in HOL Analysis, named *Hermite-unique*) to JNF.

Using the previous transfer rules, we get an statement in JNF. However, the matrices have *CARD*('n::*mod-type*) rows and columns. We want to get rid of that type variable and just state that they are of dimension $n \times n$ (expressed via the predicate *carrier-mat*

```
lemma Hermite-unique-JNF':
 fixes A::'a::{bezout-ring-div,normalization-euclidean-semiring,unique-euclidean-ring}
mat
  assumes A ∈ carrier-mat CARD('n::mod-type) CARD('n::mod-type)
    P ∈ carrier-mat CARD('n::mod-type) CARD('n::mod-type)
    H ∈ carrier-mat CARD('n::mod-type) CARD('n::mod-type)
    Q ∈ carrier-mat CARD('n::mod-type) CARD('n::mod-type)
    K ∈ carrier-mat CARD('n::mod-type) CARD('n::mod-type)
```

assumes $A = P * H$
and $A = Q * K$ **and** *invertible-mat* A **and** *invertible-mat* P
and *invertible-mat* Q **and** *Hermite-JNF associates res* H **and** *Hermite-JNF associates res* K
shows $H = K$
proof −
define A' **where** $A' = (Mod\text{-}Type\text{-}Connect.to\text{-}hma_m\ A :: 'a\ ^{'}n :: mod\text{-}type\ ^{'}n :: mod\text{-}type)$
define P' **where** $P' = (Mod\text{-}Type\text{-}Connect.to\text{-}hma_m\ P :: 'a\ ^{'}n :: mod\text{-}type\ ^{'}n :: mod\text{-}type)$
define H' **where** $H' = (Mod\text{-}Type\text{-}Connect.to\text{-}hma_m\ H :: 'a\ ^{'}n :: mod\text{-}type\ ^{'}n :: mod\text{-}type)$
define Q' **where** $Q' = (Mod\text{-}Type\text{-}Connect.to\text{-}hma_m\ Q :: 'a\ ^{'}n :: mod\text{-}type\ ^{'}n :: mod\text{-}type)$
define K' **where** $K' = (Mod\text{-}Type\text{-}Connect.to\text{-}hma_m\ K :: 'a\ ^{'}n :: mod\text{-}type\ ^{'}n :: mod\text{-}type)$
have AA'[*transfer-rule*]: *Mod-Type-Connect.HMA-M A* A' **unfolding** *Mod-Type-Connect.HMA-M-def* **using** *assms* A'*-def* **by** *auto*
have PP'[*transfer-rule*]: *Mod-Type-Connect.HMA-M P* P' **unfolding** *Mod-Type-Connect.HMA-M-def* **using** *assms* P'*-def* **by** *auto*
have HH'[*transfer-rule*]: *Mod-Type-Connect.HMA-M H* H' **unfolding** *Mod-Type-Connect.HMA-M-def* **using** *assms* H'*-def* **by** *auto*
have QQ'[*transfer-rule*]: *Mod-Type-Connect.HMA-M Q* Q' **unfolding** *Mod-Type-Connect.HMA-M-def* **using** *assms* Q'*-def* **by** *auto*
have KK'[*transfer-rule*]: *Mod-Type-Connect.HMA-M K* K' **unfolding** *Mod-Type-Connect.HMA-M-def* **using** *assms* K'*-def* **by** *auto*
have *A-PH*: $A' = P' ** H'$ **using** *assms* **by** *transfer*
moreover have *A-QK*: $A' = Q' ** K'$ **using** *assms* **by** *transfer*
moreover have *inv-A*: *invertible* A' **using** *assms* **by** *transfer*
moreover have *inv-P*: *invertible* P' **using** *assms* **by** *transfer*
moreover have *inv-Q*: *invertible* Q' **using** *assms* **by** *transfer*
moreover have *H*: *Hermite associates res* H' **using** *assms* **by** *transfer*
moreover have *K*: *Hermite associates res* K' **using** *assms* **by** *transfer*
ultimately have $H' = K'$ **using** *Hermite-unique* **by** *blast*
thus *H=K* **by** *transfer*
qed

Since the *mod-type* restriction relies on many things, the shortcut is to use the *mod-ring* typedef developed in the Berlekamp-Zassenhaus development. This type definition allows us to apply local type definitions easily. Since *mod-ring* is just an instance of *mod-type*, it is straightforward to obtain the following lemma, where $CARD('n::mod\text{-}type)$ has now been substituted by $CARD('n::nontriv\ mod\text{-}ring)$

corollary *Hermite-unique-JNF-with-nontriv-mod-ring*:
fixes $A::'a::\{bezout\text{-}ring\text{-}div, normalization\text{-}euclidean\text{-}semiring, unique\text{-}euclidean\text{-}ring\}$ *mat*
assumes $A \in carrier\text{-}mat\ CARD('n)\ CARD('n::nontriv\ mod\text{-}ring)$
$P \in carrier\text{-}mat\ CARD('n)\ CARD('n)$
$H \in carrier\text{-}mat\ CARD('n)\ CARD('n)$

$Q \in$ *carrier-mat CARD('n) CARD('n)*
$K \in$ *carrier-mat CARD('n) CARD('n)*
assumes $A = P * H$
and $A = Q * K$ **and** *invertible-mat A* **and** *invertible-mat P*
and *invertible-mat Q* **and** *Hermite-JNF associates res H* **and** *Hermite-JNF associates res K*
shows $H = K$ **using** *Hermite-unique-JNF' assms* **by** (*smt CARD-mod-ring*)

Now, we assume in a context that there exists a type text $'b$ of cardinality n and we prove inside this context the lemma.

context
fixes *n*::*nat*
assumes *local-typedef*: $\exists$(*Rep* :: ('b $\Rightarrow$ *int*)) *Abs. type-definition Rep Abs* {*0..<n* :: *int*}
and *p*: *n>1*
begin

private lemma *type-to-set*:
shows *class.nontriv TYPE('b)* (**is** *?a*) **and** *n=CARD('b)* (**is** *?b*)
proof −
from *local-typedef* **obtain** *Rep*::('b $\Rightarrow$ *int*) **and** *Abs*
where *t*: *type-definition Rep Abs* {*0..<n* :: *int*} **by** *auto*
have *card* (*UNIV* :: *'b set*) = *card* {*0..<n*} **using** *t type-definition.card* **by** *fastforce*
also have ... = *n* **by** *auto*
finally show *?b* **..**
then show *?a* **unfolding** *class.nontriv-def* **using** *p* **by** *auto*
qed

lemma *Hermite-unique-JNF-aux*:
fixes *A*::'*a*::{*bezout-ring-div*,*normalization-euclidean-semiring*,*unique-euclidean-ring*} *mat*
assumes $A \in$ *carrier-mat n n*
$P \in$ *carrier-mat n n*
$H \in$ *carrier-mat n n*
$Q \in$ *carrier-mat n n*
$K \in$ *carrier-mat n n*
assumes $A = P * H$
and $A = Q * K$ **and** *invertible-mat A* **and** *invertible-mat P*
and *invertible-mat Q* **and** *Hermite-JNF associates res H* **and** *Hermite-JNF associates res K*
shows $H = K$
using *Hermite-unique-JNF-with-nontriv-mod-ring*[*unfolded CARD-mod-ring*, *internalize-sort 'n*::*nontriv*, **where** *?'a='b*]
unfolding *type-to-set*(*2*)[*symmetric*] **using** *type-to-set*(*1*) *assms* **by** *blast*
end

Now, we cancel the local type definition of the previous context. Since the

mod-type restriction imposes the type to have cardinality greater than 1, the cases $n = 0$ and $n = 1$ must be proved separately (they are trivial)

lemma *Hermite-unique-JNF*:
fixes *A*::'*a*::{*bezout-ring-div*,*normalization-euclidean-semiring*,*unique-euclidean-ring*} *mat*
assumes *A*: *A* ∈ *carrier-mat n n* **and** *P*: *P* ∈ *carrier-mat n n* **and** *H*: *H* ∈ *carrier-mat n n*
and *Q*: *Q* ∈ *carrier-mat n n* **and** *K*: *K* ∈ *carrier-mat n n*
assumes *A-PH*: *A* = *P* ∗ *H* **and** *A-QK*: *A* = *Q* ∗ *K*
and *inv-A*: *invertible-mat A* **and** *inv-P*: *invertible-mat P* **and** *inv-Q*: *invertible-mat Q*
and *HNF-H*: *Hermite-JNF associates res H* **and** *HNF-K*: *Hermite-JNF associates res K*
shows *H* = *K*
proof (*cases n=0* ∨ *n=1*)
case *True* **note** *zero-or-one* = *True*
show *?thesis*
proof (*cases n=0*)
case *True*
then show *?thesis* **using** *assms* **by** *auto*
next
case *False*
have *CS-A*: *Complete-set-non-associates associates* **using** *HNF-H* **unfolding** *Hermite-JNF-def* **by** *simp*
have *H*: *H* ∈ *carrier-mat 1 1* **and** *K*: *K*∈ *carrier-mat 1 1* **using** *False zero-or-one assms* **by** *auto*
have *det-P-dvd-1*: *Determinant.det P dvd 1* **using** *invertible-iff-is-unit-JNF inv-P P* **by** *blast*
have *det-Q-dvd-1*: *Determinant.det Q dvd 1* **using** *invertible-iff-is-unit-JNF inv-Q Q* **by** *blast*
have *PH-QK*: *Determinant.det P* ∗ *Determinant.det H* = *Determinant.det Q* ∗ *Determinant.det K*
using *Determinant.det-mult assms* **by** *metis*
hence *Determinant.det P* ∗ *H* $$ (*0*,*0*) = *Determinant.det Q* ∗ *K* $$ (*0*,*0*)
by (*metis H K determinant-one-element*)
obtain *u* **where** *uH-K*: *u* ∗ *H* $$(*0*,*0*) = *K* $$ (*0*,*0*) **and** *unit-u*: *is-unit u*
by (*metis* (*no-types*, *opaque-lifting*) *H K PH-QK algebraic-semidom-class.dvd-mult-unit-iff det-P-dvd-1*
det-Q-dvd-1 det-singleton dvdE dvd-mult-cancel-left mult.commute mult.right-neutral one-dvd)
have *H00-not-0*: *H* $$ (*0*,*0*) ≠ *0*
by (*metis A A-PH Determinant.det-mult False H P determinant-one-element inv-A*
invertible-iff-is-unit-JNF mult-not-zero not-is-unit-0 zero-or-one)
hence *LEAST-H*: (*LEAST n. H* $$ (*0*,*n*) ≠ *0*) = *0* **by** *simp*
have *H00*: *H* $$ (*0*,*0*) ∈ *associates* **using** *HNF-H LEAST-H H H00-not-0*
unfolding *Hermite-JNF-def is-zero-row-JNF-def* **by** *auto*
have *K00-not-0*: *K* $$ (*0*,*0*) ≠ *0*
by (*metis A A-QK Determinant.det-mult False K Q determinant-one-element*

inv-A
invertible-iff-is-unit-JNF mult-not-zero not-is-unit-0 zero-or-one)
hence *LEAST-K*: (*LEAST n. K $$ (0,n) ≠ 0*) = *0* **by** *simp*
have *K00*: *K $$ (0,0) ∈ associates* **using** *HNF-K LEAST-K K K00-not-0*
unfolding *Hermite-JNF-def is-zero-row-JNF-def* **by** *auto*
have *ass-H00-K00*: *normalize* (*H $$ (0,0)*) = *normalize* (*K $$ (0,0)*)
by (*metis normalize-mult-unit-left uH-K unit-u*)
have *H00-eq-K00*: *H $$ (0,0) = K $$ (0,0)*
using *in-Ass-not-associated*[*OF CS-A H00 K00*] *ass-H00-K00* **by** *auto*
show *?thesis* **by** (*rule eq-matI*, *insert H K H00-eq-K00*, *auto*)
qed
next
case *False*
hence {*0..<int n*} ≠ {} **by** *auto*
moreover have *n>1* **using** *False* **by** *simp*
ultimately show *?thesis* **using** *Hermite-unique-JNF-aux*[*cancel-type-definition*] *assms* **by** *metis*
qed

end

From here on, we apply the same approach to move the new generalized statement about the uniqueness Hermite normal form, i.e., the version restricted to integer matrices, but imposing invertibility over the rationals.

lemma *HMA-map-matrix* [*transfer-rule*]:
((=) ===> *Mod-Type-Connect.HMA-M* ===> *Mod-Type-Connect.HMA-M*) *map-mat map-matrix*
unfolding *map-vector-def map-matrix-def*[*abs-def*] *map-mat-def*[*abs-def*]
Mod-Type-Connect.HMA-M-def Mod-Type-Connect.from-hma$_m$*-def*
by *auto*

lemma *Hermite-unique-generalized-JNF'*:
fixes *A*::*int mat*
assumes *A ∈ carrier-mat CARD('n::mod-type) CARD('n::mod-type)*
P ∈ carrier-mat CARD('n::mod-type) CARD('n::mod-type)
H ∈ carrier-mat CARD('n::mod-type) CARD('n::mod-type)
Q ∈ carrier-mat CARD('n::mod-type) CARD('n::mod-type)
K ∈ carrier-mat CARD('n::mod-type) CARD('n::mod-type)
assumes *A = P * H*
and *A = Q * K* **and** *invertible-mat* (*map-mat rat-of-int A*) **and** *invertible-mat P*
and *invertible-mat Q* **and** *Hermite-JNF associates res H* **and** *Hermite-JNF associates res K*
shows *H = K*
proof −
define *A'* **where** *A' = (Mod-Type-Connect.to-hma*$_m$ *A :: int ^'n :: mod-type ^'n :: mod-type)*

define P' **where** $P' = (Mod\text{-}Type\text{-}Connect.to\text{-}hma_m\ P :: int\ \widehat{}'n :: mod\text{-}type\ \widehat{}'n :: mod\text{-}type)$
define H' **where** $H' = (Mod\text{-}Type\text{-}Connect.to\text{-}hma_m\ H :: int\ \widehat{}'n :: mod\text{-}type\ \widehat{}'n :: mod\text{-}type)$
define Q' **where** $Q' = (Mod\text{-}Type\text{-}Connect.to\text{-}hma_m\ Q :: int\ \widehat{}'n :: mod\text{-}type\ \widehat{}'n :: mod\text{-}type)$
define K' **where** $K' = (Mod\text{-}Type\text{-}Connect.to\text{-}hma_m\ K :: int\ \widehat{}'n :: mod\text{-}type\ \widehat{}'n :: mod\text{-}type)$
have AA'[*transfer-rule*]: *Mod-Type-Connect.HMA-M A A'* **unfolding** *Mod-Type-Connect.HMA-M-def* **using** *assms A'-def* **by** *auto*
have PP'[*transfer-rule*]: *Mod-Type-Connect.HMA-M P P'* **unfolding** *Mod-Type-Connect.HMA-M-def* **using** *assms P'-def* **by** *auto*
have HH'[*transfer-rule*]: *Mod-Type-Connect.HMA-M H H'* **unfolding** *Mod-Type-Connect.HMA-M-def* **using** *assms H'-def* **by** *auto*
have QQ'[*transfer-rule*]: *Mod-Type-Connect.HMA-M Q Q'* **unfolding** *Mod-Type-Connect.HMA-M-def* **using** *assms Q'-def* **by** *auto*
have KK'[*transfer-rule*]: *Mod-Type-Connect.HMA-M K K'* **unfolding** *Mod-Type-Connect.HMA-M-def* **using** *assms K'-def* **by** *auto*
have *A-PH*: $A' = P' ** H'$ **using** *assms* **by** *transfer*
moreover have *A-QK*: $A' = Q' ** K'$ **using** *assms* **by** *transfer*
moreover have *inv-A*: *invertible* (*map-matrix rat-of-int A'*) **using** *assms* **by** *transfer*
moreover have *invertible* (*Finite-Cartesian-Product.map-matrix rat-of-int A'*)
using *inv-A* **unfolding** *Finite-Cartesian-Product.map-matrix-def map-matrix-def map-vector-def*
by *simp*
moreover have *inv-P*: *invertible P'* **using** *assms* **by** *transfer*
moreover have *inv-Q*: *invertible Q'* **using** *assms* **by** *transfer*
moreover have *H*: *Hermite associates res H'* **using** *assms* **by** *transfer*
moreover have *K*: *Hermite associates res K'* **using** *assms* **by** *transfer*
ultimately have $H' = K'$ **using** *Hermite-unique-generalized* **by** *blast*
thus $H{=}K$ **by** *transfer*
qed

corollary *Hermite-unique-generalized-JNF-with-nontriv-mod-ring*:
fixes *A*::*int mat*
assumes $A \in carrier\text{-}mat\ CARD('n)\ CARD('n::nontriv\ mod\text{-}ring)$
$P \in carrier\text{-}mat\ CARD('n)\ CARD('n)$
$H \in carrier\text{-}mat\ CARD('n)\ CARD('n)$
$Q \in carrier\text{-}mat\ CARD('n)\ CARD('n)$
$K \in carrier\text{-}mat\ CARD('n)\ CARD('n)$
assumes $A = P * H$
and $A = Q * K$ **and** *invertible-mat* (*map-mat rat-of-int A*) **and** *invertible-mat P*
and *invertible-mat Q* **and** *Hermite-JNF associates res H* **and** *Hermite-JNF associates res K*
shows $H = K$ **using** *Hermite-unique-generalized-JNF' assms* **by** (*smt CARD-mod-ring*)

context
fixes *p*::*nat*
assumes *local-typedef*: ∃(*Rep* :: ('*b* ⇒ *int*)) *Abs*. *type-definition Rep Abs* {*0*..<*p* :: *int*}
and *p*: *p*>*1*
begin

private lemma *type-to-set2*:
shows *class.nontriv TYPE*('*b*) (**is** *?a*) **and** *p*=*CARD*('*b*) (**is** *?b*)
proof −
from *local-typedef* **obtain** *Rep*::('*b* ⇒ *int*) **and** *Abs*
where *t*: *type-definition Rep Abs* {*0*..<*p* :: *int*} **by** *auto*
have *card* (*UNIV* :: '*b set*) = *card* {*0*..<*p*} **using** *t type-definition.card* **by** *fastforce*
also have ... = *p* **by** *auto*
finally show *?b* **..**
then show *?a* **unfolding** *class.nontriv-def* **using** *p* **by** *auto*
qed

lemma *Hermite-unique-generalized-JNF-aux*:
fixes *A*::*int mat*
assumes *A* ∈ *carrier-mat p p*
P ∈ *carrier-mat p p*
H ∈ *carrier-mat p p*
Q ∈ *carrier-mat p p*
K ∈ *carrier-mat p p*
assumes *A* = *P* ∗ *H*
and *A* = *Q* ∗ *K* **and** *invertible-mat* (*map-mat rat-of-int A*) **and** *invertible-mat P*
and *invertible-mat Q* **and** *Hermite-JNF associates res H* **and** *Hermite-JNF associates res K*
shows *H* = *K*
using *Hermite-unique-generalized-JNF-with-nontriv-mod-ring*[*unfolded CARD-mod-ring*, *internalize-sort* '*n*::*nontriv*, **where** *?*'*a*='*b*]
unfolding *type-to-set2*(*2*)[*symmetric*] **using** *type-to-set2*(*1*) *assms* **by** *blast*
end

lemma *HNF-unique-generalized-JNF*:
fixes *A*::*int mat*
assumes *A*: *A* ∈ *carrier-mat n n* **and** *P*: *P* ∈ *carrier-mat n n* **and** *H*: *H* ∈ *carrier-mat n n*
and *Q*: *Q* ∈ *carrier-mat n n* **and** *K*: *K* ∈ *carrier-mat n n*
assumes *A-PH*: *A* = *P* ∗ *H* **and** *A-QK*: *A* = *Q* ∗ *K*
and *inv-A*: *invertible-mat* (*map-mat rat-of-int A*) **and** *inv-P*: *invertible-mat P*

and *inv-Q*: *invertible-mat Q*
and *HNF-H*: *Hermite-JNF associates res H* **and** *HNF-K*: *Hermite-JNF associates res K*
shows *H = K*
proof (*cases n=0 ∨ n=1*)
case *True* **note** *zero-or-one = True*
show *?thesis*
proof (*cases n=0*)
case *True*
then show *?thesis* **using** *assms* **by** *auto*
next
let *?RAT = map-mat rat-of-int*
case *False*
hence *n*: *n=1* **using** *zero-or-one* **by** *auto*
have *CS-A*: *Complete-set-non-associates associates* **using** *HNF-H* **unfolding** *Hermite-JNF-def* **by** *simp*
have *H*: *H ∈ carrier-mat 1 1* **and** *K*: *K∈ carrier-mat 1 1* **using** *False zero-or-one assms* **by** *auto*
have *det-P-dvd-1*: *Determinant.det P dvd 1* **using** *invertible-iff-is-unit-JNF inv-P P* **by** *blast*
have *det-Q-dvd-1*: *Determinant.det Q dvd 1* **using** *invertible-iff-is-unit-JNF inv-Q Q* **by** *blast*
have *PH-QK*: *Determinant.det P ∗ Determinant.det H = Determinant.det Q ∗ Determinant.det K*
using *Determinant.det-mult assms* **by** *metis*
hence *Determinant.det P ∗ H $$ (0,0) = Determinant.det Q ∗ K $$ (0,0)*
by (*metis H K determinant-one-element*)
obtain *u* **where** *uH-K*: *u ∗ H $$(0,0) = K $$ (0,0)* **and** *unit-u*: *is-unit u*
by (*metis* (*no-types*, *opaque-lifting*) *H K PH-QK algebraic-semidom-class.dvd-mult-unit-iff det-P-dvd-1*
det-Q-dvd-1 det-singleton dvdE dvd-mult-cancel-left mult.commute mult.right-neutral one-dvd)
have *H00-not-0*: *H $$ (0,0) ≠ 0*
proof −
have *?RAT A = ?RAT P ∗ ?RAT H* **using** *A-PH*
using *P H n of-int-hom.mat-hom-mult* **by** *blast*
hence *det* (*?RAT H*) *≠ 0*
by (*metis A Determinant.det-mult False H P inv-A invertible-iff-is-unit-JNF*
map-carrier-mat mult-eq-0-iff not-is-unit-0 zero-or-one)
thus *?thesis*
using *H determinant-one-element* **by** *force*
qed
hence *LEAST-H*: (*LEAST n. H $$ (0,n) ≠ 0*) *= 0* **by** *simp*
have *H00*: *H $$ (0,0) ∈ associates* **using** *HNF-H LEAST-H H H00-not-0*
unfolding *Hermite-JNF-def is-zero-row-JNF-def* **by** *auto*
have *K00-not-0*: *K $$ (0,0) ≠ 0*
proof −
have *?RAT A = ?RAT Q ∗ ?RAT K* **using** *A-QK*

```
      using Q K n of-int-hom.mat-hom-mult by blast
    hence det (?RAT K) ≠ 0
     by (metis A Determinant.det-mult False Q K inv-A invertible-iff-is-unit-JNF

         map-carrier-mat mult-eq-0-iff not-is-unit-0 zero-or-one)
    thus ?thesis
      using K determinant-one-element by force
  qed
  hence LEAST-K: (LEAST n. K $$ (0,n) ≠ 0) = 0 by simp
  have K00: K $$ (0,0) ∈ associates using HNF-K LEAST-K K K00-not-0
    unfolding Hermite-JNF-def is-zero-row-JNF-def by auto
  have ass-H00-K00: normalize (H $$ (0,0)) = normalize (K $$ (0,0))
    by (metis normalize-mult-unit-left uH-K unit-u)
  have H00-eq-K00: H $$ (0,0) = K $$ (0,0)
    using in-Ass-not-associated[OF CS-A H00 K00] ass-H00-K00 by auto
  show ?thesis by (rule eq-matI, insert H K H00-eq-K00, auto)
 qed
next
 case False
 hence {0..<int n} ≠ {} by auto
 moreover have n>1 using False by simp
 ultimately show ?thesis
    using Hermite-unique-generalized-JNF-aux[cancel-type-definition] assms by
metis
qed

end
```

8 Formalization of an efficient Hermite normal form algorithm

We formalize a version of the Hermite normal form algorithm based on reductions modulo the determinant. This avoids the growth of the intermediate coefficients.

8.1 Implementation of the algorithm using generic modulo operation

Exception on generic modulo: currently in Hermite-reduce-above, ordinary div/mod is used, since that is our choice for the complete set of residues.

```
theory HNF-Mod-Det-Algorithm
 imports
   Jordan-Normal-Form.Gauss-Jordan-IArray-Impl
   Show.Show-Instances
   Jordan-Normal-Form.Determinant-Impl
   Jordan-Normal-Form.Show-Matrix
```

LLL-Basis-Reduction.LLL-Certification
Smith-Normal-Form.SNF-Algorithm-Euclidean-Domain
Smith-Normal-Form.SNF-Missing-Lemmas
Uniqueness-Hermite-JNF
Matrix-Change-Row
begin

8.1.1 Echelon form algorithm

fun *make-first-column-positive* :: *int mat* ⇒ *int mat* **where**
make-first-column-positive A = (
Matrix.mat (*dim-row A*) (*dim-col A*) — Create a matrix of the same dimensions
(λ(*i*,*j*). *if A* $$(*i*,*0*) < *0 then* − *A* $$(*i*,*j*) *else A* $$(*i*,*j*)
)
)

locale *mod-operation* =
fixes *generic-mod* :: *int* ⇒ *int* ⇒ *int* (**infixl** ‹*gmod*› *70*)
and *generic-div* :: *int* ⇒ *int* ⇒ *int* (**infixl** ‹*gdiv*› *70*)
begin

Version for reducing all elements

fun *reduce* :: *nat* ⇒ *nat* ⇒ *int* ⇒ *int mat* ⇒ *int mat* **where**
reduce a b D A = (*let Aaj* = *A*$$(*a*,*0*); *Abj* = *A* $$ (*b*,*0*)
in
if Aaj = *0 then A else*
case euclid-ext2 Aaj Abj of (*p*,*q*,*u*,*v*,*d*) ⇒ — p*Aaj + q * Abj = d, u = - Abj/d, v = Aaj/d
Matrix.mat (*dim-row A*) (*dim-col A*) — Create a matrix of the same dimensions
(λ(*i*,*k*). *if i* = *a then let r* = (*p*∗*A*$$(*a*,*k*) + *q*∗*A*$$(*b*,*k*)) *in*
if k = *0 then if D dvd r then D else r else r gmod D* — Row a is multiplied by p and added row b multiplied by q, modulo D
else if i = *b then let r* = *u* ∗ *A*$$(*a*,*k*) + *v* ∗ *A*$$(*b*,*k*) *in*
if k = *0 then r else r gmod D* — Row b is multiplied by v and added row a multiplied by u, modulo D
else A$$(*i*,*k*) — All the other rows remain unchanged
)
)

Version for reducing, with abs-checking

fun *reduce-abs* :: *nat* ⇒ *nat* ⇒ *int* ⇒ *int mat* ⇒ *int mat* **where**
reduce-abs a b D A = (*let Aaj* = *A*$$(*a*,*0*); *Abj* = *A* $$ (*b*,*0*)
in
if Aaj = *0 then A else*
case euclid-ext2 Aaj Abj of (*p*,*q*,*u*,*v*,*d*) ⇒ — p*Aaj + q * Abj = d, u = - Abj/d, v = Aaj/d
Matrix.mat (*dim-row A*) (*dim-col A*) — Create a matrix of the same dimensions
(λ(*i*,*k*). *if i* = *a then let r* = (*p*∗*A*$$(*a*,*k*) + *q*∗*A*$$(*b*,*k*)) *in*

```
                    if abs r > D then if k = 0 ∧ D dvd r then D else r gmod D
else r
                else if i = b then let r = u * A$$(a,k) + v * A$$(b,k) in
                        if abs r > D then r gmod D else r
                else A$$(i,k)  — All the other rows remain unchanged
          )
  )

definition reduce-impl :: nat ⇒ nat ⇒ int ⇒ int mat ⇒ int mat where
  reduce-impl a b D A = (let
    row-a = Matrix.row A a;
    Aaj = row-a $v 0
  in
  if Aaj = 0 then A else let
    row-b = Matrix.row A b;
    Abj = row-b $v 0 in
  case euclid-ext2 Aaj Abj of (p,q,u,v,d) ⇒
    let row-a' = (λ k ak. let r = (p * ak + q * row-b $v k) in
            if k = 0 then if D dvd r then D else r else r gmod D);
        row-b' = (λ k bk. let r = u * row-a $v k + v * bk in
                        if k = 0 then r else r gmod D)
     in change-row a row-a' (change-row b row-b' A)
  )

definition reduce-abs-impl :: nat ⇒ nat ⇒ int ⇒ int mat ⇒ int mat where
  reduce-abs-impl a b D A = (let
    row-a = Matrix.row A a;
    Aaj = row-a $v 0
  in
  if Aaj = 0 then A else let
    row-b = Matrix.row A b;
    Abj = row-b $v 0 in
  case euclid-ext2 Aaj Abj of (p,q,u,v,d) ⇒
    let row-a' = (λ k ak. let r = (p * ak + q * row-b $v k) in
            if abs r > D then if k = 0 ∧ D dvd r then D else r gmod D else r);
        row-b' = (λ k bk. let r = u * row-a $v k + v * bk in
                        if abs r > D then r gmod D else r)
     in change-row a row-a' (change-row b row-b' A)
  )

lemma reduce-impl: a < nr ⟹ b < nr ⟹ 0 < nc ⟹ a ≠ b ⟹ A ∈ carrier-mat
nr nc
  ⟹ reduce-impl a b D A = reduce a b D A
  unfolding reduce-impl-def reduce.simps Let-def
  apply (intro if-cong[OF - refl], force)
  apply (intro prod.case-cong refl, force)
  apply (intro eq-matI, auto)
  done
```

lemma *reduce-abs-impl*: $a < nr \Longrightarrow b < nr \Longrightarrow 0 < nc \Longrightarrow a \neq b \Longrightarrow A \in$ *carrier-mat nr nc*
$\Longrightarrow$ *reduce-abs-impl a b D A* = *reduce-abs a b D A*
unfolding *reduce-abs-impl-def reduce-abs.simps Let-def*
apply (*intro if-cong*[*OF - refl*], *force*)
apply (*intro prod.case-cong refl*, *force*)
apply (*intro eq-matI*, *auto*)
done

fun *reduce-below* :: *nat* $\Rightarrow$ *nat list* $\Rightarrow$ *int* $\Rightarrow$ *int mat* $\Rightarrow$ *int mat*
where *reduce-below a* [] *D A* = *A*
| *reduce-below a* (*x* # *xs*) *D A* = *reduce-below a xs D* (*reduce a x D A*)

fun *reduce-below-impl* :: *nat* $\Rightarrow$ *nat list* $\Rightarrow$ *int* $\Rightarrow$ *int mat* $\Rightarrow$ *int mat*
where *reduce-below-impl a* [] *D A* = *A*
| *reduce-below-impl a* (*x* # *xs*) *D A* = *reduce-below-impl a xs D* (*reduce-impl a x D A*)

lemma *reduce-impl-carrier*[*simp*,*intro*]: *A* $\in$ *carrier-mat m n* $\Longrightarrow$ *reduce-impl a b D A* $\in$ *carrier-mat m n*
unfolding *reduce-impl-def Let-def* **by** (*auto split*: *prod.splits*)

lemma *reduce-below-impl*: $a < nr \Longrightarrow 0 < nc \Longrightarrow (\bigwedge b.\ b \in set\ bs \Longrightarrow b < nr)$
$\Longrightarrow a \notin set\ bs$
$\Longrightarrow A \in$ *carrier-mat nr nc* $\Longrightarrow$ *reduce-below-impl a bs D A* = *reduce-below a bs D A*
proof (*induct bs arbitrary*: *A*)
case (*Cons b bs A*)
show *?case* **by** (*simp del*: *reduce.simps*,
subst reduce-impl[*of - nr - nc*],
(*insert Cons*, *auto simp del*: *reduce.simps*)[*5*],
rule Cons(*1*), *insert Cons*(*2*$-$), *auto simp*: *Let-def split*: *prod.splits*)
qed *simp*

fun *reduce-below-abs* :: *nat* $\Rightarrow$ *nat list* $\Rightarrow$ *int* $\Rightarrow$ *int mat* $\Rightarrow$ *int mat*
where *reduce-below-abs a* [] *D A* = *A*
| *reduce-below-abs a* (*x* # *xs*) *D A* = *reduce-below-abs a xs D* (*reduce-abs a x D A*)

fun *reduce-below-abs-impl* :: *nat* $\Rightarrow$ *nat list* $\Rightarrow$ *int* $\Rightarrow$ *int mat* $\Rightarrow$ *int mat*
where *reduce-below-abs-impl a* [] *D A* = *A*
| *reduce-below-abs-impl a* (*x* # *xs*) *D A* = *reduce-below-abs-impl a xs D* (*reduce-abs-impl a x D A*)

lemma *reduce-abs-impl-carrier*[*simp*,*intro*]: $A \in$ *carrier-mat m n* $\Longrightarrow$ *reduce-abs-impl a b D A* $\in$ *carrier-mat m n*
unfolding *reduce-abs-impl-def Let-def* **by** (*auto split*: *prod.splits*)

lemma *reduce-abs-below-impl*: $a <$ *nr* $\Longrightarrow$ $0 <$ *nc* $\Longrightarrow$ $(\bigwedge$ *b*. *b* $\in$ *set bs* $\Longrightarrow$ *b* $<$ *nr*) $\Longrightarrow$ *a* $\notin$ *set bs*
$\Longrightarrow$ *A* $\in$ *carrier-mat nr nc* $\Longrightarrow$ *reduce-below-abs-impl a bs D A* = *reduce-below-abs a bs D A*
proof (*induct bs arbitrary*: *A*)
case (*Cons b bs A*)
show *?case* **by** (*simp del*: *reduce-abs.simps*,
subst reduce-abs-impl[*of - nr - nc*],
(*insert Cons*, *auto simp del*: *reduce-abs.simps*)[*5*],
rule Cons(*1*), *insert Cons*(*2*−), *auto simp*: *Let-def split*: *prod.splits*)
qed *simp*

This function outputs a matrix in echelon form via reductions modulo the determinant

function *FindPreHNF* :: *bool* ⇒ *int* ⇒ *int mat* ⇒ *int mat*
where *FindPreHNF abs-flag D A* =
(*let m* = *dim-row A*; *n* = *dim-col A in*
if m < *2* ∨ *n* = *0 then A else* — No operations are carried out if m = 1
let non-zero-positions = *filter* (λ*i*. *A* $$ (*i*,*0*) ≠ *0*) [*1*..<*dim-row A*];
A′ = (*if A*$$(*0*,*0*) ≠ *0 then A*
else let i = *non-zero-positions* ! *0* — Select the first non-zero position below the first element
in swaprows 0 i A
);
Reduce = (*if abs-flag then reduce-below-abs else reduce-below*)
in
if n < *2 then Reduce 0 non-zero-positions D A′* — If n = 1, then we have to reduce the column
else
let
(*A-UL*,*A-UR*,*A-DL*,*A-DR*) = *split-block* (*Reduce 0 non-zero-positions D* (*make-first-column-positive A′*)) *1 1*;
sub-PreHNF = *FindPreHNF abs-flag D A-DR in*
four-block-mat A-UL A-UR A-DL sub-PreHNF)
by *pat-completeness auto*

termination
proof (*relation Wellfounded.measure* (λ(*abs-flag*,*D*,*A*). *dim-col A*))
show *wf* (*Wellfounded.measure* (λ(*abs-flag*,*D*, *A*). *dim-col A*)) **by** *auto*
fix *abs-flag D A m n nz A′ R xd A′-UL y A′-UR ya A′-DL A′-DR*
assume *m*: *m* = *dim-row A* **and** *n*:*n* = *dim-col A*
and *m2*: ¬ (*m* < *2* ∨ *n* = *0*) **and** *nz-def*: *nz* = *filter* (λ*i*. *A* $$ (*i*, *0*) ≠ *0*) [*1*..<*dim-row A*]
and *A′-def*: *A′* = (*if A* $$ (*0*, *0*) ≠ *0 then A else let i* = *nz* ! *0 in swaprows 0 i A*)

and *R-def*: $R = (\textit{if abs-flag then reduce-below-abs else reduce-below})$
and *n2*: $\neg\ n < 2$ **and** $xd = \textit{split-block}\ (R\ 0\ nz\ D\ (\textit{make-first-column-positive}\ A'))\ 1\ 1$
and $(A'\text{-}UL,\ y) = xd$ **and** $(A'\text{-}UR,\ ya) = y$ **and** $(A'\text{-}DL,\ A'\text{-}DR) = ya$
hence *A'-split*: $(A'\text{-}UL,\ A'\text{-}UR,\ A'\text{-}DL,\ A'\text{-}DR)$
$= \textit{split-block}\ (R\ 0\ nz\ D\ (\textit{make-first-column-positive}\ A'))\ 1\ 1$ **by** *force*
have *dr-mk1*: *dim-row* (*make-first-column-positive A*) = *dim-row A* **for** *A* **by** *auto*
have *dr-mk2*: *dim-col* (*make-first-column-positive A*) = *dim-col A* **for** *A* **by** *auto*

have *r1*: *reduce-below a xs D A* ∈ *carrier-mat m n* **if** *A* ∈ *carrier-mat m n* **for** *A a xs*
using *that* **by** (*induct a xs D A rule*: *reduce-below.induct*, *auto simp add*: *Let-def euclid-ext2-def*)
hence *R*: (*reduce-below 0 nz D* (*make-first-column-positive A'*)) ∈ *carrier-mat m n*
using *A'-def m n*
by (*metis carrier-matI index-mat-swaprows*(*2*,*3*) *dr-mk1 dr-mk2*)
have *reduce-below-abs a xs D A* ∈ *carrier-mat m n* **if** *A* ∈ *carrier-mat m n* **for** *A a xs*
using *that* **by** (*induct a xs D A rule*: *reduce-below-abs.induct*, *auto simp add*: *Let-def euclid-ext2-def*)
hence *R2*: (*reduce-below-abs 0 nz D* (*make-first-column-positive A'*)) ∈ *carrier-mat m n*
using *A'-def m n*
by (*metis carrier-matI index-mat-swaprows*(*2*,*3*) *dr-mk1 dr-mk2*)

have *A'-DR* ∈ *carrier-mat* (*m*−*1*) (*n*−*1*)
by (*cases abs-flag*; *rule split-block*(*4*)[*OF A'-split*[*symmetric*]],*insert m2 n2 m n R-def R R2*, *auto*)
thus ((*abs-flag*, *D*, *A'-DR*),*abs-flag*, *D*, *A*) ∈ *Wellfounded.measure* (λ(*abs-flag*,*D*, *A*). *dim-col A*) **using** *n2 m2 n m* **by** *auto*
qed

lemma *FindPreHNF-code*: *FindPreHNF abs-flag D A* =
(*let m* = *dim-row A*; *n* = *dim-col A in*
if m < *2* ∨ *n* = *0 then A else*
let non-zero-positions = *filter* (λ*i*. *A* $$ (*i*,*0*) ≠ *0*) [*1*..<*dim-row A*];
A' = (*if A*$$(*0*,*0*) ≠ *0 then A*
else let i = *non-zero-positions* ! *0 in swaprows 0 i A*
);
Reduce-impl = (*if abs-flag then reduce-below-abs-impl else reduce-below-impl*)
in
if n < *2 then Reduce-impl 0 non-zero-positions D A'*
else
let
(*A-UL*,*A-UR*,*A-DL*,*A-DR*) = *split-block* (*Reduce-impl 0 non-zero-positions D* (*make-first-column-positive A'*)) *1 1*;
sub-PreHNF = *FindPreHNF abs-flag D A-DR in*

```
      four-block-mat A-UL A-UR A-DL sub-PreHNF)  (is ?lhs = ?rhs)
proof −
  let ?f = λR. (if dim-row A < 2 ∨ dim-col A = 0 then A else if dim-col A < 2
        then R 0 (filter (λi. A $$ (i, 0) ≠ 0) [1..<dim-row A]) D
   (if A $$ (0, 0) ≠ 0 then A else swaprows 0 (filter (λi. A $$ (i, 0) ≠ 0)
[1..<dim-row A] ! 0) A)
  else case split-block (R 0 (filter (λi. A $$ (i, 0) ≠ 0) [1..<dim-row A]) D
    (make-first-column-positive (if A $$ (0, 0) ≠ 0 then A else
      swaprows 0 (filter (λi. A $$ (i, 0) ≠ 0) [1..<dim-row A] ! 0) A))) 1 1 of
  (A-UL, A-UR, A-DL, A-DR) ⇒ four-block-mat A-UL A-UR A-DL (FindPreHNF
abs-flag D A-DR))
  have M-carrier: make-first-column-positive (if A $$ (0, 0) ≠ 0 then A
    else swaprows 0 (filter (λi. A $$ (i, 0) ≠ 0) [1..<dim-row A] ! 0) A)
    ∈ carrier-mat (dim-row A) (dim-col A)
   by (smt (verit) index-mat-swaprows(2) index-mat-swaprows(3) make-first-column-positive.simps
mat-carrier)
  have ∗: 0 ∉ set (filter (λi. A $$ (i, 0) ≠ 0) [1..<dim-row A]) by simp
  have ?lhs = ?f (if abs-flag then reduce-below-abs else reduce-below)
    unfolding FindPreHNF.simps[of abs-flag D A] Let-def by presburger
  also have ... = ?rhs
  proof (cases abs-flag)
    case True
   have ?f (if abs-flag then reduce-below-abs else reduce-below) = ?f reduce-below-abs

      using True by presburger
    also have ... = ?f reduce-below-abs-impl
      by ((intro if-cong refl prod.case-cong arg-cong[of - - λ x. split-block x 1 1];
       (subst reduce-abs-below-impl[where nr = dim-row A and nc = dim-col A])),
(auto)[9])
       (insert M-carrier ∗, blast+)
   also have ... = ?f (if abs-flag then reduce-below-abs-impl else reduce-below-impl)

      using True by presburger
    finally show ?thesis using True unfolding FindPreHNF.simps[of abs-flag D
A] Let-def by blast
  next
    case False
    have ?f (if abs-flag then reduce-below-abs else reduce-below) = ?f reduce-below
      using False by presburger
    also have ... = ?f reduce-below-impl
      by ((intro if-cong refl prod.case-cong arg-cong[of - - λ x. split-block x 1 1];
        (subst reduce-below-impl[where nr = dim-row A and nc = dim-col A])),
(auto)[9])
       (insert M-carrier ∗, blast+)
   also have ... = ?f (if abs-flag then reduce-below-abs-impl else reduce-below-impl)

      using False by presburger
    finally show ?thesis using False unfolding FindPreHNF.simps[of abs-flag D
A] Let-def by blast
```

```
  qed
  finally show ?thesis by blast
qed
end

declare mod-operation.FindPreHNF-code[code]
declare mod-operation.reduce-below-impl.simps[code]
declare mod-operation.reduce-impl-def[code]
declare mod-operation.reduce-below-abs-impl.simps[code]
declare mod-operation.reduce-abs-impl-def[code]
```

8.1.2 From echelon form to Hermite normal form

From here on, we define functions to transform a matrix in echelon form into its Hermite normal form. Essentially, we are defining the functions that are available in the AFP entry Hermite (which uses HOL Analysis + mod-type) in the JNF matrix representation.

```
definition find-fst-non0-in-row :: nat ⇒ int mat ⇒ nat option where
  find-fst-non0-in-row l A = (let is = [l ..< dim-col A];
    Ais = filter (λj. A $$ (l, j) ≠ 0) is
    in case Ais of [] ⇒ None | - ⇒ Some (Ais!0))

primrec Hermite-reduce-above
where Hermite-reduce-above (A::int mat) 0 i j = A
    | Hermite-reduce-above A (Suc n) i j = (let
    Aij = A $$ (i,j);
    Anj = A $$ (n,j)
    in
    Hermite-reduce-above (addrow (− (Anj div Aij)) n i A) n i j)

definition Hermite-of-row-i :: int mat ⇒ nat ⇒ int mat
  where Hermite-of-row-i A i = (
  case find-fst-non0-in-row i A of None ⇒ A | Some j ⇒
    let Aij = A $$(i,j) in
    if Aij < 0 then Hermite-reduce-above (multrow i (−1) A) i i j
    else Hermite-reduce-above A i i j)


primrec Hermite-of-list-of-rows
  where
 Hermite-of-list-of-rows A [] = A |
 Hermite-of-list-of-rows A (a#xs) = Hermite-of-list-of-rows (Hermite-of-row-i A
a) xs
```

We combine the previous functions to assemble the algorithm

```
definition (in mod-operation) Hermite-mod-det abs-flag A =
  (let m = dim-row A; n = dim-col A;
   D = abs(det-int A);
```

```
  A' = A @_r D ·_m 1_m n;
  E = FindPreHNF abs-flag D A';
  H = Hermite-of-list-of-rows E [0..<m+n]
 in mat-of-rows n (map (Matrix.row H) [0..<m]))
```

8.1.3 Some examples of execution

```
declare mod-operation.Hermite-mod-det-def[code]

value let B = mat-of-rows-list 4 ([[0,3,1,4],[7,1,0,0],[8,0,19,16],[2,0,0,3::int]])
in
  show (mod-operation.Hermite-mod-det (mod) True B)




value let B = mat-of-rows-list 7 ([
[ 1, 17, -41, -1,  1,  0,  0],
[ 0, -1,  2,  0, -6,  2,  1],
[ 9,  2,  1,  1, -2,  2, -5],
[-1, -3, -1,  0, -9,  0,  0],
[ 9, -1, -9,  0,  0,  0,  1],
[ 1, -1,  1,  0,  1, -8,  0],
[ 1, -1,  0, -2, -1, -1,  0::int]]) in
  show (mod-operation.Hermite-mod-det (mod) True B)



end
```

8.2 Soundness of the algorithm

```
theory HNF-Mod-Det-Soundness
  imports
    HNF-Mod-Det-Algorithm
    Signed-Modulo
begin

hide-const(open) Determinants.det Determinants2.upper-triangular
  Finite-Cartesian-Product.row Finite-Cartesian-Product.rows
  Finite-Cartesian-Product.vec
```

8.2.1 Results connecting lattices and Hermite normal form

The following results will also be useful for proving the soundness of the certification approach.

```
lemma of-int-mat-hom-int-id[simp]:
  fixes A::int mat
  shows of-int-hom.mat-hom A = A unfolding map-mat-def by auto
```

definition *is-sound-HNF algorithm associates res*
  = (∀ *A*. *let* (*P*,*H*) = *algorithm A*; *m* = *dim-row A*; *n* = *dim-col A in*
    P ∈ *carrier-mat m m* ∧ *H* ∈ *carrier-mat m n* ∧ *invertible-mat P* ∧ *A* = *P* ∗ *H*
    ∧ *Hermite-JNF associates res H*)

lemma *HNF-A-eq-HNF-PA*:
 fixes *A*::'*a*::{*bezout-ring-div*,*normalization-euclidean-semiring*,*unique-euclidean-ring*} *mat*
 assumes *A*: *A* ∈ *carrier-mat n n* **and** *inv-A*: *invertible-mat A*
  and *inv-P*: *invertible-mat P* **and** *P*: *P* ∈ *carrier-mat n n*
  and *sound-HNF*: *is-sound-HNF HNF associates res*
  and *P1-H1*: (*P1*,*H1*) = *HNF* (*P*∗*A*)
  and *P2-H2*: (*P2*,*H2*) = *HNF A*
 shows *H1* = *H2*
proof −
  obtain *inv-P* **where** *P-inv-P*: *inverts-mat P inv-P* **and** *inv-P-P*: *inverts-mat inv-P P*
   and *inv-P*: *inv-P* ∈ *carrier-mat n n*
   using *P inv-P obtain-inverse-matrix* **by** *blast*
 have *P1*: *P1* ∈ *carrier-mat n n*
    using *P1-H1 sound-HNF* **unfolding** *is-sound-HNF-def Let-def*
   by (*metis* (*no-types*, *lifting*) *P carrier-matD*(*1*) *index-mult-mat*(*2*) *old.prod.case*)
  have *H1*: *H1* ∈ *carrier-mat n n* **using** *P1-H1 sound-HNF* **unfolding** *is-sound-HNF-def Let-def*
 by (*metis* (*no-types*, *lifting*) *A P carrier-matD*(*1*) *carrier-matD*(*2*) *case-prodD index-mult-mat*(*2*,*3*))
 have *invertible-inv-P*: *invertible-mat inv-P*
    using *P-inv-P inv-P inv-P-P invertible-mat-def square-mat.simps* **by** *blast*
  have *P-A-P1-H1*: *P* ∗ *A* = *P1* ∗ *H1* **using** *P1-H1 sound-HNF* **unfolding** *is-sound-HNF-def Let-def*
   by (*metis* (*mono-tags*, *lifting*) *case-prod-conv*)
 hence *A* = *inv-P* ∗ (*P1* ∗ *H1*)
  by (*smt* (*verit*) *A P inv-P-P inv-P assoc-mult-mat carrier-matD*(*1*) *inverts-mat-def left-mult-one-mat*)
 hence *A-inv-P-P1-H1*: *A* = (*inv-P* ∗ *P1*) ∗ *H1*
   using *H1 P1 inv-P* **by** *fastforce*
 have *A-P2-H2*: *A* = *P2* ∗ *H2* **using** *P2-H2 sound-HNF* **unfolding** *is-sound-HNF-def Let-def*
   by (*metis* (*mono-tags*, *lifting*) *case-prod-conv*)
 have *invertible-inv-P-P1*: *invertible-mat* (*inv-P* ∗ *P1*)
 proof (*rule invertible-mult-JNF*[*OF inv-P P1 invertible-inv-P*])
   show *invertible-mat P1*
    by (*smt* (*verit*) *P1-H1 is-sound-HNF-def prod.sel*(*1*) *sound-HNF split-beta*)
 qed
 show *?thesis*
 proof (*rule Hermite-unique-JNF*[*OF A - H1 - - A-inv-P-P1-H1 A-P2-H2 inv-A*

invertible-inv-P-P1])
show *inv-P* ∗ *P1* ∈ *carrier-mat n n*
by (*metis carrier-matD*(*1*) *carrier-matI index-mult-mat*(*2*) *inv-P invertible-inv-P-P1 invertible-mat-def square-mat.simps*)
show *P2* ∈ *carrier-mat n n*
by (*smt* (*verit*) *A P2-H2 carrier-matD*(*1*) *is-sound-HNF-def prod.sel*(*1*) *sound-HNF split-beta*)
show *H2* ∈ *carrier-mat n n*
by (*smt* (*verit*) *A P2-H2 carrier-matD*(*1*) *carrier-matD*(*2*) *is-sound-HNF-def prod.sel*(*2*) *sound-HNF split-beta*)
show *invertible-mat P2*
by (*smt* (*verit*) *P2-H2 is-sound-HNF-def prod.sel*(*1*) *sound-HNF split-beta*)
show *Hermite-JNF associates res H1*
by (*smt* (*verit*) *P1-H1 is-sound-HNF-def prod.sel*(*2*) *sound-HNF split-beta*)
show *Hermite-JNF associates res H2*
by (*smt* (*verit*) *P2-H2 is-sound-HNF-def prod.sel*(*2*) *sound-HNF split-beta*)
qed
qed

context *vec-module*
begin

lemma *mat-mult-invertible-lattice-eq*:
assumes *fs*: *set fs* ⊆ *carrier-vec n*
and *gs*: *set gs* ⊆ *carrier-vec n*
and *P*: *P* ∈ *carrier-mat m m* **and** *invertible-P*: *invertible-mat P*
and *length-fs*: *length fs* = *m* **and** *length-gs*: *length gs* = *m*
and *prod*: *mat-of-rows n fs* = (*map-mat of-int P*) ∗ *mat-of-rows n gs*
shows *lattice-of fs* = *lattice-of gs*
proof thm *mat-mult-sub-lattice*
show *lattice-of fs* ⊆ *lattice-of gs*
by (*rule mat-mult-sub-lattice*[*OF fs gs - prod*],*simp add*: *length-fs length-gs P*)
next
obtain *inv-P* **where** *P-inv-P*: *inverts-mat P inv-P* **and** *inv-P-P*: *inverts-mat inv-P P*
and *inv-P*: *inv-P* ∈ *carrier-mat m m*
using *P invertible-P obtain-inverse-matrix* **by** *blast*
have *of-int-hom.mat-hom* (*inv-P*) ∗ *mat-of-rows n fs*
= *of-int-hom.mat-hom* (*inv-P*) ∗ ((*map-mat of-int P*) ∗ *mat-of-rows n gs*)
using *prod* **by** *auto*
also have ... = *of-int-hom.mat-hom* (*inv-P*) ∗ (*map-mat of-int P*) ∗ *mat-of-rows n gs*
by (*smt* (*verit*) *P assoc-mult-mat inv-P length-gs map-carrier-mat mat-of-rows-carrier*(*1*))
also have ... = *of-int-hom.mat-hom* (*inv-P* ∗ *P*) ∗ *mat-of-rows n gs*
by (*metis P inv-P of-int-hom.mat-hom-mult*)
also have ... = *mat-of-rows n gs*
by (*metis carrier-matD*(*1*) *inv-P inv-P-P inverts-mat-def left-mult-one-mat′ length-gs mat-of-rows-carrier*(*2*) *of-int-hom.mat-hom-one*)

finally have *prod*: *mat-of-rows n gs* = *of-int-hom.mat-hom* (*inv-P*) ∗ *mat-of-rows n fs* **..**
show *lattice-of gs* ⊆ *lattice-of fs*
by (*rule mat-mult-sub-lattice*[*OF gs fs - prod*], *simp add*: *length-fs length-gs inv-P*)
qed

end

context
fixes *n* :: *nat*
begin

interpretation *vec-module TYPE*(*int*) **.**

lemma *lattice-of-HNF*:
assumes *sound-HNF*: *is-sound-HNF HNF associates res*
and *P1-H1*: (*P*,*H*) = *HNF* (*mat-of-rows n fs*)
and *fs*: *set fs* ⊆ *carrier-vec n* **and** *len*: *length fs* = *m*
shows *lattice-of fs* = *lattice-of* (*rows H*)
proof (*rule mat-mult-invertible-lattice-eq*[*OF fs*])
have *H*: *H* ∈ *carrier-mat m n* **using** *sound-HNF P1-H1* **unfolding** *is-sound-HNF-def Let-def*
by (*metis* (*mono-tags*, *lifting*) *assms*(*4*) *mat-of-rows-carrier*(*2*) *mat-of-rows-carrier*(*3*) *prod.sel*(*2*) *split-beta*)
have *H-rw*: *mat-of-rows n* (*Matrix.rows H*) = *H* **using** *mat-of-rows-rows H* **by** *fast*
have *PH-fs-init*: *mat-of-rows n fs* = *P* ∗ *H* **using** *sound-HNF P1-H1* **unfolding** *is-sound-HNF-def Let-def*
by (*metis* (*mono-tags*, *lifting*) *case-prodD*)
show *mat-of-rows n fs* = *of-int-hom.mat-hom P* ∗ *mat-of-rows n* (*Matrix.rows H*)
unfolding *H-rw of-int-mat-hom-int-id* **using** *PH-fs-init* **by** *simp*
show *set* (*Matrix.rows H*) ⊆ *carrier-vec n* **using** *H rows-carrier* **by** *blast*
show *P* ∈ *carrier-mat m m* **using** *sound-HNF P1-H1* **unfolding** *is-sound-HNF-def Let-def*
by (*metis* (*no-types*, *lifting*) *len case-prodD mat-of-rows-carrier*(*2*))
show *invertible-mat P* **using** *sound-HNF P1-H1* **unfolding** *is-sound-HNF-def Let-def*
by (*metis* (*no-types*, *lifting*) *case-prodD*)
show *length fs* = *m* **using** *len* **by** *simp*
show *length* (*Matrix.rows H*) = *m* **using** *H* **by** *auto*
qed
end

context *LLL-with-assms*
begin

lemma *certification-via-eq-HNF*:
assumes *sound-HNF*: *is-sound-HNF HNF associates res*
and *P1-H1*: (*P1*,*H1*) = *HNF* (*mat-of-rows n fs-init*)
and *P2-H2*: (*P2*,*H2*) = *HNF* (*mat-of-rows n gs*)
and *H1-H2*: *H1* = *H2*
and *gs*: *set gs* ⊆ *carrier-vec n* **and** *len-gs*: *length gs* = *m*
shows *lattice-of gs* = *lattice-of fs-init LLL-with-assms n m gs* α
proof −
have *lattice-of fs-init* = *lattice-of* (*rows H1*)
by (*rule lattice-of-HNF*[*OF sound-HNF P1-H1 fs-init*], *simp add*: *len*)
also have ... = *lattice-of* (*rows H2*) **using** *H1-H2* **by** *auto*
also have ... = *lattice-of gs*
by (*rule lattice-of-HNF*[*symmetric*, *OF sound-HNF P2-H2 gs len-gs*])
finally show *lattice-of gs* = *lattice-of fs-init* **..**
have *invertible-P1*: *invertible-mat P1*
using *sound-HNF P1-H1* **unfolding** *is-sound-HNF-def*
by (*metis* (*mono-tags*, *lifting*) *case-prodD*)
have *invertible-P2*: *invertible-mat P2*
using *sound-HNF P2-H2* **unfolding** *is-sound-HNF-def*
by (*metis* (*mono-tags*, *lifting*) *case-prodD*)
have *P2*: *P2* ∈ *carrier-mat m m*
using *sound-HNF P2-H2* **unfolding** *is-sound-HNF-def*
by (*metis* (*no-types*, *lifting*) *len-gs case-prodD mat-of-rows-carrier*(*2*))
obtain *inv-P2* **where** *P2-inv-P2*: *inverts-mat P2 inv-P2* **and** *inv-P2-P2*: *inverts-mat inv-P2 P2*
and *inv-P2*: *inv-P2* ∈ *carrier-mat m m*
using *P2 invertible-P2 obtain-inverse-matrix* **by** *blast*
have *P1*: *P1* ∈ *carrier-mat m m*
using *sound-HNF P1-H1* **unfolding** *is-sound-HNF-def*
by (*metis* (*no-types*, *lifting*) *len case-prodD mat-of-rows-carrier*(*2*))
have *H1*: *H1* ∈ *carrier-mat m n*
using *sound-HNF P1-H1* **unfolding** *is-sound-HNF-def*
by (*metis* (*no-types*, *lifting*) *case-prodD len mat-of-rows-carrier*(*2*) *mat-of-rows-carrier*(*3*))
have *H2*: *H2* ∈ *carrier-mat m n*
using *sound-HNF P2-H2* **unfolding** *is-sound-HNF-def*
by (*metis* (*no-types*, *lifting*) *len-gs case-prodD mat-of-rows-carrier*(*2*) *mat-of-rows-carrier*(*3*))
have *P2-H2*: *P2* ∗ *H2* = *mat-of-rows n gs*
by (*smt* (*verit*) *P2-H2 sound-HNF case-prodD is-sound-HNF-def*)
have *P1-H1-fs*: *P1* ∗ *H1* = *mat-of-rows n fs-init*
by (*smt* (*verit*) *P1-H1 sound-HNF case-prodD is-sound-HNF-def*)
obtain *inv-P1* **where** *P1-inv-P1*: *inverts-mat P1 inv-P1* **and** *inv-P1-P1*: *inverts-mat inv-P1 P1*
and *inv-P1*: *inv-P1* ∈ *carrier-mat m m*
using *P1 invertible-P1 obtain-inverse-matrix* **by** *blast*
show *LLL-with-assms n m gs* α
proof (*rule LLL-change-basis*(*2*)[*OF gs len-gs*])
show *P1* ∗ *inv-P2* ∈ *carrier-mat m m* **using** *P1 inv-P2* **by** *auto*

have *mat-of-rows n fs-init = P1 * H1* **using** *sound-HNF P2-H2* **unfolding** *is-sound-HNF-def*
by (*metis* (*mono-tags*, *lifting*) *P1-H1 case-prodD*)
also have ... = *P1 * inv-P2 * P2 * H1*
by (*smt* (*verit*) *P1 P2 assoc-mult-mat carrier-matD*(1) *inv-P2 inv-P2-P2 inverts-mat-def right-mult-one-mat*)
also have ... = *P1 * inv-P2 * P2 * H2* **using** *H1-H2* **by** *blast*
also have ... = *P1 * inv-P2 * (P2 * H2)*
using *H2 P2* ‹*P1 * inv-P2* ∈ *carrier-mat m m*› *assoc-mult-mat* **by** *blast*
also have ... = *P1 * (inv-P2 * P2 * H2)*
by (*metis H2* ‹*P1 * H1 = P1 * inv-P2 * P2 * H1*› ‹*P1 * inv-P2 * P2 * H2 = P1 * inv-P2 * (P2 * H2)*›
H1-H2 carrier-matD(1) *inv-P2 inv-P2-P2 inverts-mat-def left-mult-one-mat*)
also have ... = *P1 * (inv-P2 * (P2 * H2))* **using** *H2 P2 inv-P2* **by** *auto*
also have ... = *P1 * inv-P2 * mat-of-rows n gs*
using *P2-H2* ‹*P1 * (inv-P2 * P2 * H2) = P1 * (inv-P2 * (P2 * H2))*›
‹*P1 * inv-P2 * (P2 * H2) = P1 * (inv-P2 * P2 * H2)*› **by** *auto*
finally show *mat-of-rows n fs-init = P1 * inv-P2 * mat-of-rows n gs* **.**
show *P2 * inv-P1* ∈ *carrier-mat m m*
using *P2 inv-P1* **by** *auto*
have *mat-of-rows n gs = P2 * H2* **using** *sound-HNF P2-H2* **unfolding** *is-sound-HNF-def* **by** *metis*
also have ... = *P2 * inv-P1 * P1 * H2*
by (*smt* (*verit*) *P1 P2 assoc-mult-mat carrier-matD*(1) *inv-P1 inv-P1-P1 inverts-mat-def right-mult-one-mat*)
also have ... = *P2 * inv-P1 * P1 * H1* **using** *H1-H2* **by** *blast*
also have ... = *P2 * inv-P1 * (P1 * H1)*
using *H1 P1* ‹*P2 * inv-P1* ∈ *carrier-mat m m*› *assoc-mult-mat* **by** *blast*
also have ... = *P2 * (inv-P1 * P1 * H1)*
by (*metis H2* ‹*P2 * H2 = P2 * inv-P1 * P1 * H2*› ‹*P2 * inv-P1 * P1 * H1 = P2 * inv-P1 * (P1 * H1)*›
H1-H2 carrier-matD(1) *inv-P1 inv-P1-P1 inverts-mat-def left-mult-one-mat*)
also have ... = *P2 * (inv-P1 * (P1 * H1))* **using** *H1 P1 inv-P1* **by** *auto*
also have ... = *P2 * inv-P1 * mat-of-rows n fs-init*
using *P1-H1-fs* ‹*P2 * (inv-P1 * P1 * H1) = P2 * (inv-P1 * (P1 * H1))*›
‹*P2 * inv-P1 * (P1 * H1) = P2 * (inv-P1 * P1 * H1)*› **by** *auto*
finally show *mat-of-rows n gs = P2 * inv-P1 * mat-of-rows n fs-init* **.**
qed
qed

end

context *vec-space*
begin

lemma *lin-indpt-cols-imp-det-not-0*:
fixes *A*::*'a mat*
assumes *A*: *A* ∈ *carrier-mat n n* **and** *li*: *lin-indpt* (*set* (*cols A*)) **and** *d*: *distinct* (*cols A*)

shows *det A ≠ 0*
using *A li d det-rank-iff lin-indpt-full-rank* **by** *blast*

corollary *lin-indpt-rows-imp-det-not-0*:
fixes *A*::*'a mat*
assumes *A*: *A ∈ carrier-mat n n* **and** *li*: *lin-indpt* (*set* (*rows A*)) **and** *d*: *distinct* (*rows A*)
shows *det A ≠ 0*
using *A li d det-rank-iff lin-indpt-full-rank*
by (*metis* (*full-types*) *Determinant.det-transpose cols-transpose transpose-carrier-mat*)
end

context *LLL*
begin

lemma *eq-lattice-imp-mat-mult-invertible-cols*:
assumes *fs*: *set fs ⊆ carrier-vec n*
and *gs*: *set gs ⊆ carrier-vec n* **and** *ind-fs*: *lin-indep fs*
and *length-fs*: *length fs = n* **and** *length-gs*: *length gs = n*
and *l*: *lattice-of fs = lattice-of gs*
shows ∃ *Q ∈ carrier-mat n n*. *invertible-mat Q ∧ mat-of-cols n fs = mat-of-cols n gs ∗ Q*
proof (*cases n=0*)
case *True*
show *?thesis*
by (*rule bexI*[*of - 1*$_m$ *0*], *insert True assms*, *auto*)
next
case *False*
hence *n*: *0<n* **by** *simp*
have *ind-RAT-fs*: *gs.lin-indpt* (*set* (*RAT fs*)) **using** *ind-fs*
by (*simp add*: *cof-vec-space.lin-indpt-list-def*)
have *fs-carrier*: *mat-of-cols n fs ∈ carrier-mat n n* **by** (*simp add*: *length-fs carrier-matI*)
let *?f* = (*λi*. *SOME x*. *x∈carrier-vec* (*length gs*) *∧* (*mat-of-cols n gs*) *∗*$_v$ *x = fs ! i*)
let *?cols-Q* = *map ?f* [*0..<length fs*]
let *?Q* = *mat-of-cols n ?cols-Q*
show *?thesis*
proof (*rule bexI*[*of - ?Q*], *rule conjI*)
show *Q*: *?Q ∈ carrier-mat n n* **using** *length-fs* **by** *auto*
show *fs-gs-Q*: *mat-of-cols n fs = mat-of-cols n gs ∗ ?Q*
proof (*rule mat-col-eqI*)
fix *j* **assume** *j*: *j < dim-col* (*mat-of-cols n gs ∗ ?Q*)
have *j2*: *j<n* **using** *j Q length-gs* **by** *auto*
have *fs-j-in-gs*: *fs ! j ∈ lattice-of gs* **using** *fs l basis-in-latticeI j* **by** *auto*
have *fs-j-carrier-vec*: *fs ! j ∈ carrier-vec n*
using *fs-j-in-gs gs lattice-of-as-mat-mult* **by** *blast*
let *?x* = *SOME x*. *x∈carrier-vec* (*length gs*) *∧* (*mat-of-cols n gs*) *∗*$_v$ *x = fs ! j*
have *?x∈carrier-vec* (*length gs*) *∧* (*mat-of-cols n gs*) *∗*$_v$ *?x = fs ! j*

```
      by (rule someI-ex, insert fs-j-in-gs lattice-of-as-mat-mult[OF gs], auto)
    hence x: ?x ∈ carrier-vec (length gs)
      and gs-x: (mat-of-cols n gs) *v ?x = fs ! j by blast+
    have col ?Q j = ?cols-Q ! j
    proof (rule col-mat-of-cols)
      show j < length (map ?f [0..<length fs]) using length-fs j2 by auto
      have map ?f [0..<length fs] ! j = ?f ([0..<length fs] ! j)
        by (rule nth-map, insert j2 length-fs, auto)
      also have ... = ?f j by (simp add: length-fs j2)
      also have ... ∈ carrier-vec n using x length-gs by auto
      finally show map ?f [0..<length fs] ! j ∈ carrier-vec n .
    qed
    also have ... = ?f ([0..<length fs] ! j)
      by (rule nth-map, insert j2 length-fs, auto)
    also have ... = ?x by (simp add: length-fs j2)
    finally have col-Qj-x: col ?Q j = ?x .
    have col (mat-of-cols n fs) j = fs ! j
     by (metis (no-types, lifting) j Q fs length-fs carrier-matD(2) cols-mat-of-cols
cols-nth
         index-mult-mat(3) mat-of-cols-carrier(3))
    also have ... = (mat-of-cols n gs) *v ?x using gs-x by auto
    also have ... = (mat-of-cols n gs) *v (col ?Q j) unfolding col-Qj-x by simp
    also have ... = col (mat-of-cols n gs * ?Q) j
      by (rule col-mult2[symmetric, OF - Q j2], insert length-gs mat-of-cols-def,
auto)
    finally show col (mat-of-cols n fs) j = col (mat-of-cols n gs * ?Q) j .
  qed (insert length-gs gs, auto)
  show invertible-mat ?Q

  proof −
    let ?f' = (λi. SOME x. x∈carrier-vec (length fs) ∧ (mat-of-cols n fs) *v x =
gs ! i)
    let ?cols-Q' = map ?f' [0..<length gs]
    let ?Q' = mat-of-cols n ?cols-Q'
    have Q': ?Q' ∈ carrier-mat n n using length-gs by auto
    have gs-fs-Q': mat-of-cols n gs = mat-of-cols n fs * ?Q'
    proof (rule mat-col-eqI)
      fix j assume j: j < dim-col (mat-of-cols n fs * ?Q')
      have j2: j<n using j Q length-gs by auto
      have gs-j-in-fs: gs ! j ∈ lattice-of fs using gs l basis-in-latticeI j by auto
      have gs-j-carrier-vec: gs ! j ∈ carrier-vec n
        using gs-j-in-fs fs lattice-of-as-mat-mult by blast
      let ?x = SOME x. x∈carrier-vec (length fs) ∧ (mat-of-cols n fs) *v x = gs
! j
      have ?x∈carrier-vec (length fs) ∧ (mat-of-cols n fs) *v ?x = gs ! j
        by (rule someI-ex, insert gs-j-in-fs lattice-of-as-mat-mult[OF fs], auto)
      hence x: ?x ∈ carrier-vec (length fs)
        and fs-x: (mat-of-cols n fs) *v ?x = gs ! j by blast+
      have col ?Q' j = ?cols-Q' ! j
```

```
      proof (rule col-mat-of-cols)
        show j < length (map ?f' [0..<length gs]) using length-gs j2 by auto
        have map ?f' [0..<length gs] ! j = ?f' ([0..<length gs] ! j)
          by (rule nth-map, insert j2 length-gs, auto)
        also have ... = ?f' j by (simp add: length-gs j2)
        also have ... ∈ carrier-vec n using x length-fs by auto
        finally show map ?f' [0..<length gs] ! j ∈ carrier-vec n .
      qed
      also have ... = ?f' ([0..<length gs] ! j)
        by (rule nth-map, insert j2 length-gs, auto)
      also have ... = ?x by (simp add: length-gs j2)
      finally have col-Qj-x: col ?Q' j = ?x .
    have col (mat-of-cols n gs) j = gs ! j by (simp add: length-gs gs-j-carrier-vec
j2)
      also have ... = (mat-of-cols n fs) *v ?x using fs-x by auto
    also have ... = (mat-of-cols n fs) *v (col ?Q' j) unfolding col-Qj-x by simp
      also have ... = col (mat-of-cols n fs * ?Q') j
       by (rule col-mult2[symmetric, OF - Q' j2], insert length-fs mat-of-cols-def,
auto)
      finally show col (mat-of-cols n gs) j = col (mat-of-cols n fs * ?Q') j .
    qed (insert length-fs fs, auto)

    have det-fs-not-zero: rat-of-int (det (mat-of-cols n fs)) ≠ 0
    proof −
      let ?A = (of-int-hom.mat-hom (mat-of-cols n fs)):: rat mat
      have rat-of-int (det (mat-of-cols n fs)) = det ?A
        by simp
      moreover have det ?A ≠ 0
      proof (rule gs.lin-indpt-cols-imp-det-not-0[of ?A])
        have c-eq: (set (cols ?A)) = set (RAT fs)
            by (metis assms(3) cof-vec-space.lin-indpt-list-def cols-mat-of-cols fs
mat-of-cols-map)
        show ?A ∈ carrier-mat n n by (simp add: fs-carrier)
        show gs.lin-indpt (set (cols ?A)) using ind-RAT-fs c-eq by auto
        show distinct (cols ?A)
              by (metis ind-fs cof-vec-space.lin-indpt-list-def cols-mat-of-cols fs
mat-of-cols-map)
      qed
      ultimately show ?thesis by auto
    qed
    have Q'Q: ?Q' * ?Q ∈ carrier-mat n n using Q Q' mult-carrier-mat by blast
    have fs-fs-Q'Q: mat-of-cols n fs = mat-of-cols n fs * ?Q' * ?Q using gs-fs-Q'
fs-gs-Q by presburger
  hence 0m n n = mat-of-cols n fs * ?Q' * ?Q − mat-of-cols n fs using length-fs
by auto
    also have ... = mat-of-cols n fs * ?Q' * ?Q − mat-of-cols n fs * 1m n
      using fs-carrier by auto
    also have ... = mat-of-cols n fs * (?Q' * ?Q) − mat-of-cols n fs * 1m n
      using Q Q' fs-carrier by auto
```

also have ... = *mat-of-cols n fs* ∗ (*?Q′* ∗ *?Q* − 1_m *n*)
by (*rule mult-minus-distrib-mat*[*symmetric*, *OF fs-carrier Q′Q*], *auto*)
finally have *mat-of-cols n fs* ∗ (*?Q′* ∗ *?Q* − 1_m *n*) = 0_m *n n* **..**
have *det* (*?Q′* ∗ *?Q*) = *1*
by (*smt* (*verit*) *Determinant.det-mult Q Q′ Q′Q fs-fs-Q′Q assoc-mult-mat det-fs-not-zero*
fs-carrier mult-cancel-left2 of-int-code(*2*))
hence *det-Q′-Q-1*: *det ?Q* ∗ *det ?Q′* = *1*
by (*metis* (*no-types*, *lifting*) *Determinant.det-mult Groups.mult-ac*(*2*) *Q Q′*)
hence *det ?Q* = *1* ∨ *det ?Q* = −*1* **by** (*rule pos-zmult-eq-1-iff-lemma*)
thus *?thesis* **using** *invertible-iff-is-unit-JNF*[*OF Q*] **by** *fastforce*
qed
qed
qed

corollary *eq-lattice-imp-mat-mult-invertible-rows*:
assumes *fs*: *set fs* ⊆ *carrier-vec n*
and *gs*: *set gs* ⊆ *carrier-vec n* **and** *ind-fs*: *lin-indep fs*
and *length-fs*: *length fs* = *n* **and** *length-gs*: *length gs* = *n*
and *l*: *lattice-of fs* = *lattice-of gs*
shows ∃ *P* ∈ *carrier-mat n n*. *invertible-mat P* ∧ *mat-of-rows n fs* = *P* ∗ *mat-of-rows n gs*
proof −
obtain *Q* **where** *Q*: *Q* ∈ *carrier-mat n n* **and** *inv-Q*: *invertible-mat Q*
and *fs-gs-Q*: *mat-of-cols n fs* = *mat-of-cols n gs* ∗ *Q*
using *eq-lattice-imp-mat-mult-invertible-cols*[*OF assms*] **by** *auto*
have *invertible-mat* Q^T **by** (*simp add*: *inv-Q invertible-mat-transpose*)
moreover have *mat-of-rows n fs* = Q^T ∗ *mat-of-rows n gs* **using** *fs-gs-Q*
by (*metis Matrix.transpose-mult Q length-gs mat-of-cols-carrier*(*1*) *transpose-mat-of-cols*)
moreover have Q^T ∈ *carrier-mat n n* **using** *Q* **by** *auto*
ultimately show *?thesis* **by** *blast*
qed
end

8.2.2 Missing results

This is a new definition for upper triangular matrix, valid for rectangular matrices. This definition will allow us to prove that echelon form implies upper triangular for any matrix.

definition *upper-triangular′ A* = (∀ *i* < *dim-row A*. ∀ *j*<*dim-col A*. *j* < *i* ⟶ *A* $$ (*i*,*j*) = *0*)

lemma *upper-triangular′D*[*elim*] :
upper-triangular′ A ⟹ *j*<*dim-col A* ⟹ *j* < *i* ⟹ *i* < *dim-row A* ⟹ *A* $$ (*i*,*j*) = *0*
unfolding *upper-triangular′-def* **by** *auto*

lemma *upper-triangular′I*[*intro*] :

(⋀*i j*. *j*<*dim-col A* ⟹ *j* < *i* ⟹ *i* < *dim-row A* ⟹ *A* $$ (*i*,*j*) = *0*) ⟹ *upper-triangular' A*
unfolding *upper-triangular'-def* **by** *auto*

lemma *prod-list-abs*:
fixes *xs*:: *int list*
shows *prod-list* (*map abs xs*) = *abs* (*prod-list xs*)
by (*induct xs*, *auto simp add*: *abs-mult*)

lemma *euclid-ext2-works*:
assumes *euclid-ext2 a b* = (*p*,*q*,*u*,*v*,*d*)
shows *p*∗*a*+*q*∗*b* = *d* **and** *d* = *gcd a b* **and** *gcd a b* ∗ *u* = −*b* **and** *gcd a b* ∗ *v* = *a*
and *u* = −*b div gcd a b* **and** *v* = *a div gcd a b*
using *assms* **unfolding** *euclid-ext2-def*
by (*auto simp add*: *bezout-coefficients-fst-snd*)

lemma *res-function-euclidean2*:
res-function (λ*b n*::'*a*::{*unique-euclidean-ring*}. *n mod b*)
proof−
have *n mod b* = *n* **if** *b*=*0* **for** *n b*::'*a* :: *unique-euclidean-ring* **using** *that* **by** *auto*
hence *res-function-euclidean* = (λ*b n*::'*a*. *n mod b*)
by (*unfold fun-eq-iff res-function-euclidean-def*, *auto*)
thus *?thesis* **using** *res-function-euclidean* **by** *auto*
qed

lemma *mult-row-1-id*:
fixes *A*:: '*a*::*semiring-1*^'*n*^'*m*
shows *mult-row A b 1* = *A* **unfolding** *mult-row-def* **by** *vector*

Results about appending rows

lemma *row-append-rows1*:
assumes *A*: *A* ∈ *carrier-mat m n*
and *B*: *B* ∈ *carrier-mat p n*
assumes *i*: *i* < *dim-row A*
shows *Matrix.row* (*A* @$_r$ *B*) *i* = *Matrix.row A i*
proof (*rule eq-vecI*)
have *AB-carrier*[*simp*]: (*A* @$_r$ *B*) ∈ *carrier-mat* (*m*+*p*) *n* **by** (*rule carrier-append-rows*[*OF A B*])
thus *dim-vec* (*Matrix.row* (*A* @$_r$ *B*) *i*) = *dim-vec* (*Matrix.row A i*)
using *A B* **by** (*auto*, *insert carrier-matD*(*2*), *blast*)
fix *j* **assume** *j*: *j* < *dim-vec* (*Matrix.row A i*)
have *Matrix.row* (*A* @$_r$ *B*) *i* $*v j* = (*A* @$_r$ *B*) $$ (*i*, *j*)
by (*metis AB-carrier Matrix.row-def j A carrier-matD*(*2*) *index-row*(*2*) *index-vec*)
also have ... = (*if i* < *dim-row A then A* $$ (*i*, *j*) *else B* $$ (*i* − *m*, *j*))
by (*rule append-rows-nth*, *insert assms j*, *auto*)
also have ... = *A*$$ (*i*,*j*) **using** *i* **by** *simp*
finally show *Matrix.row* (*A* @$_r$ *B*) *i* $*v j* = *Matrix.row A i* $*v j* **using** *i j* **by**

simp
qed

lemma *row-append-rows2*:
assumes *A*: $A \in$ *carrier-mat m n*
and *B*: $B \in$ *carrier-mat p n*
assumes *i*: $i \in \{m..<m+p\}$
shows *Matrix.row* $(A \mathbin{@_r} B)$ i = *Matrix.row* B $(i - m)$
proof (*rule eq-vecI*)
have *AB-carrier*[*simp*]: $(A \mathbin{@_r} B) \in$ *carrier-mat* $(m+p)$ n **by** (*rule carrier-append-rows*[*OF A B*])
thus *dim-vec* (*Matrix.row* $(A \mathbin{@_r} B)$ i) = *dim-vec* (*Matrix.row* B $(i-m)$)
using *A B* **by** (*auto*, *insert carrier-matD*(*2*), *blast*)
fix *j* **assume** *j*: $j <$ *dim-vec* (*Matrix.row* B $(i-m)$)
have *Matrix.row* $(A \mathbin{@_r} B)$ i \$*v* j = $(A \mathbin{@_r} B)$ \$\$ (i, j)
by (*metis AB-carrier Matrix.row-def j B carrier-matD*(*2*) *index-row*(*2*) *index-vec*)
also have ... = (*if* $i <$ *dim-row A then A* \$\$ (i, j) *else B* \$\$ $(i - m, j)$)
by (*rule append-rows-nth*, *insert assms j*, *auto*)
also have ... = B \$\$ $(i - m, j)$ **using** i A **by** *simp*
finally show *Matrix.row* $(A \mathbin{@_r} B)$ i \$*v* j = *Matrix.row* B $(i-m)$ \$*v* j **using** i j A B **by** *auto*
qed

lemma *rows-append-rows*:
assumes *A*: $A \in$ *carrier-mat m n*
and *B*: $B \in$ *carrier-mat p n*
shows *Matrix.rows* $(A \mathbin{@_r} B)$ = *Matrix.rows A* @ *Matrix.rows B*
proof −
have *AB-carrier*: $(A \mathbin{@_r} B) \in$ *carrier-mat* $(m+p)$ n
by (*rule carrier-append-rows*, *insert A B*, *auto*)
hence *1*: *dim-row* $(A \mathbin{@_r} B)$ = *dim-row A* + *dim-row B* **using** *A B* **by** *blast*
moreover have *Matrix.row* $(A \mathbin{@_r} B)$ i = (*Matrix.rows A* @ *Matrix.rows B*) ! i
if *i*: $i <$ *dim-row* $(A \mathbin{@_r} B)$ **for** i
proof (*cases i*<*dim-row A*)
case *True*
have *Matrix.row* $(A \mathbin{@_r} B)$ i = *Matrix.row A i* **using** *A True B row-append-rows1* **by** *blast*
also have ... = *Matrix.rows A* ! i **unfolding** *Matrix.rows-def* **using** *True* **by** *auto*
also have ... = (*Matrix.rows A* @ *Matrix.rows B*) ! i **using** *True* **by** (*simp add*: *nth-append*)
finally show *?thesis* **.**
next
case *False*
have *i-mp*: $i < m + p$ **using** *AB-carrier A B i* **by** *fastforce*
have *Matrix.row* $(A \mathbin{@_r} B)$ i = *Matrix.row* B $(i-m)$ **using** *A False B i row-append-rows2 i-mp*

by (*smt* (*verit*) *AB-carrier atLeastLessThan-iff carrier-matD*(*1*) *le-add1 linordered-semidom-class.add-diff-inverse row-append-rows2*)
also have ... = *Matrix.rows B* ! (*i*−*m*) **unfolding** *Matrix.rows-def* **using** *False i A 1* **by** *auto*
also have ... = (*Matrix.rows A* @ *Matrix.rows B*) ! (*i*−*m*+*m*)
by (*metis add-diff-cancel-right' A carrier-matD*(*1*) *length-rows not-add-less2 nth-append*)
also have ... = (*Matrix.rows A* @ *Matrix.rows B*) ! *i* **using** *False A* **by** *auto*
finally show *?thesis* **.**
qed
ultimately show *?thesis* **unfolding** *list-eq-iff-nth-eq* **by** *auto*
qed

lemma *append-rows-nth2*:
assumes *A'*: *A'* ∈ *carrier-mat m n*
and *B*: *B* ∈ *carrier-mat p n*
and *A-def*: *A* = (*A'* @$_r$ *B*)
and *a*: *a*<*m* **and** *ap*: *a* < *p* **and** *j*: *j*<*n*
shows *A* $$ (*a* + *m*, *j*) = *B* $$ (*a*,*j*)
proof −
have *A* $$ (*a* + *m*, *j*) = (*if a* + *m* < *dim-row A' then A'* $$ (*a* + *m*, *j*) *else B* $$ (*a* + *m* − *m*, *j*))
unfolding *A-def* **by** (*rule append-rows-nth*[*OF A' B - j*], *insert ap a*, *auto*)
also have ... = *B* $$ (*a*,*j*) **using** *ap a A'* **by** *auto*
finally show *?thesis* **.**
qed

lemma *append-rows-nth3*:
assumes *A'*: *A'* ∈ *carrier-mat m n*
and *B*: *B* ∈ *carrier-mat p n*
and *A-def*: *A* = (*A'* @$_r$ *B*)
and *a*: *a*≥*m* **and** *ap*: *a* < *m* + *p* **and** *j*: *j*<*n*
shows *A* $$ (*a*, *j*) = *B* $$ (*a*−*m*,*j*)
proof −
have *A* $$ (*a*, *j*) = (*if a* < *dim-row A' then A'* $$ (*a*, *j*) *else B* $$ (*a* − *m*, *j*))
unfolding *A-def* **by** (*rule append-rows-nth*[*OF A' B - j*], *insert ap a*, *auto*)
also have ... = *B* $$ (*a*−*m*,*j*) **using** *ap a A'* **by** *auto*
finally show *?thesis* **.**
qed

Results about submatrices

lemma *pick-first-id*: **assumes** *i*: *i*<*n* **shows** *pick* {*0*..<*n*} *i* = *i*
proof −
have *i* = (*card* {*a* ∈ {*0*..<*n*}. *a* < *i*}) **using** *i*
by (*auto*, *smt* (*verit*) *Collect-cong card-Collect-less-nat nat-SN.gt-trans*)
thus *?thesis* **using** *pick-card-in-set i*

```
    by (metis atLeastLessThan-iff zero-order(1))
qed

lemma submatrix-index-id:
  assumes H: H ∈ carrier-mat m n and i: i<k1 and j: j<k2
  and k1: k1≤m and k2: k2≤n
  shows (submatrix H {0..<k1} {0..<k2}) $$ (i,j) = H $$ (i,j)
proof −
  let ?I = {0..<k1}
  let ?J = {0..<k2}
  let ?H = submatrix H ?I ?J
  have km: k1≤m and kn: k2≤n using k1 k2 by simp+
  then have {i. i < m ∧ i < k1} = {..<k1} {i. i < n ∧ i < k2} = {..<k2} by
auto
  then have card-mk: card {i. i < m ∧ i < k1} = k1 and card-nk: card {i. i <
n ∧ i < k2} = k2
    by auto
  show ?thesis
  proof−
    have pick-j: pick ?J j = j by (rule pick-first-id[OF j])
    have pick-i: pick ?I i = i by (rule pick-first-id[OF i])
    have submatrix H ?I ?J $$ (i, j) = H $$ (pick ?I i, pick ?J j)
      by (rule submatrix-index, insert H i j card-mk card-nk, auto)
    also have ... = H $$ (i,j) using pick-i pick-j by simp
    finally show ?thesis .
  qed
qed

lemma submatrix-carrier-first:
  assumes H: H ∈ carrier-mat m n
  and k1: k1 ≤ m and k2: k2 ≤ n
  showssubmatrix H {0..<k1} {0..<k2} ∈ carrier-mat k1 k2
proof −
  have km: k1≤m and kn: k2≤n using k1 k2 by simp+
  then have {i. i < m ∧ i < k1} = {..<k1} {i. i < n ∧ i < k2} = {..<k2} by
auto
  then have card-mk: card {i. i < m ∧ i < k1} = k1 and card-nk: card {i. i <
n ∧ i < k2} = k2
    by auto
  show ?thesis
   by (smt (verit) Collect-cong H atLeastLessThan-iff card-mk card-nk carrier-matD

      carrier-matI dim-submatrix zero-order(1))
qed



lemma Units-eq-invertible-mat:
  assumes A ∈ carrier-mat n n
```

shows $A \in$ *Group.Units* (*ring-mat TYPE*('*a*::*comm-ring-1*) *n b*) = *invertible-mat A* (**is** *?lhs* = *?rhs*)
proof −
interpret *m*: *ring ring-mat TYPE*('*a*) *n b* **by** (*rule ring-mat*)
show *?thesis*
proof
assume *?lhs* **thus** *?rhs*
unfolding *Group.Units-def*
by (*insert assms*, *auto simp add*: *ring-mat-def invertible-mat-def inverts-mat-def*)
next
assume *?rhs*
from *this* **obtain** *B* **where** *AB*: $A * B = 1_m\ n$ **and** *BA*: $B * A = 1_m\ n$ **and** *B*: $B \in$ *carrier-mat n n*
by (*metis assms carrier-matD*(*1*) *inverts-mat-def obtain-inverse-matrix*)
hence $\exists x{\in}$*carrier* (*ring-mat TYPE*('*a*) *n b*). $x \otimes_{\textit{ring-mat TYPE}('a)\ n\ b} A = \mathbf{1}_{\textit{ring-mat TYPE}('a)\ n\ b}$
$\wedge\ A \otimes_{\textit{ring-mat TYPE}('a)\ n\ b} x = \mathbf{1}_{\textit{ring-mat TYPE}('a)\ n\ b}$
unfolding *ring-mat-def* **by** *auto*
thus *?lhs* **unfolding** *Group.Units-def* **using** *assms* **unfolding** *ring-mat-def* **by** *auto*
qed
qed

lemma *map-first-rows-index*:
assumes $A \in$ *carrier-mat M n* **and** $m \leq M$ **and** $i{<}m$ **and** $ja{<}n$
shows *map* (*Matrix.row A*) [*0*..<*m*] ! *i* $v *ja* = *A* $$ (*i*, *ja*)
using *assms* **by** *auto*

lemma *matrix-append-rows-eq-if-preserves*:
assumes *A*: $A \in$ *carrier-mat* $(m{+}p)$ *n* **and** *B*: $B \in$ *carrier-mat p n*
and *eq*: $\forall\, i{\in}\{m..{<}m{+}p\}.\forall\, j{<}n.\ A\$\$(i,j) = B\ \$\$\ (i{-}m,j)$
shows A = *mat-of-rows n* [*Matrix.row A i*. *i* ← [*0*..<*m*]] $@_r$ *B* (**is** - = *?A'* $@_r$ -)
proof (*rule eq-matI*)
have *A'*: *?A'* ∈ *carrier-mat m n* **by** (*simp add*: *mat-of-rows-def*)
hence *A'B*: *?A'* $@_r$ *B* ∈ *carrier-mat* $(m{+}p)$ *n* **using** *B* **by** *blast*
show *dim-row A* = *dim-row* (*?A'* $@_r$ *B*) **and** *dim-col A* = *dim-col* (*?A'* $@_r$ *B*)
using *A'B A* **by** *auto*
fix *i j* **assume** *i*: *i* < *dim-row* (*?A'* $@_r$ *B*)
and *j*: *j* < *dim-col* (*?A'* $@_r$ *B*)
have *jn*: *j*<*n* **using** *A*
by (*metis append-rows-def dim-col-mat*(*1*) *index-mat-four-block*(*3*) *index-zero-mat*(*3*)
j mat-of-rows-def nat-arith.rule0)
let *?xs* = (*map* (*Matrix.row A*) [*0*..<*m*])
show *A* $$ (*i*, *j*) = (*?A'* $@_r$ *B*) $$ (*i*, *j*)
proof (*cases i*<*m*)
case *True*
have (*?A'* $@_r$ *B*) $$ (*i*, *j*) = *?A'* $$ (*i*,*j*)

```
    by (metis (no-types, lifting) Nat.add-0-right True append-rows-def diff-zero i
      index-mat-four-block index-zero-mat(3) j length-map length-upt mat-of-rows-carrier(2))
  also have ... = ?xs ! i $v j
    by (rule mat-of-rows-index, insert i True j, auto simp add: append-rows-def)
  also have ... = A $$ (i,j)
    by (rule map-first-rows-index, insert assms A True i jn, auto)
  finally show ?thesis ..
 next
  case False
  have (?A' @_r B) $$ (i, j) = B $$ (i−m,j)
   by (smt (verit) A' carrier-matD(1) False append-rows-def i index-mat-four-block
j jn length-map
      length-upt mat-of-rows-carrier(2,3))
  also have ... = A $$ (i,j)
    by (metis False append-rows-def B eq atLeastLessThan-iff carrier-matD(1)
diff-zero i
      index-mat-four-block(2) index-zero-mat(2) jn le-add1 length-map length-upt

      linordered-semidom-class.add-diff-inverse mat-of-rows-carrier(2))
  finally show ?thesis ..
 qed
qed

lemma invertible-mat-first-column-not0:
 fixes A::'a :: comm-ring-1 mat
 assumes A: A ∈ carrier-mat n n and inv-A: invertible-mat A and n0: 0<n
 shows col A 0 ≠ (0_v n)
proof (rule ccontr)
 assume ¬ col A 0 ≠ 0_v n hence col-A0: col A 0 = 0_v n by simp
 have (det A dvd 1) using inv-A invertible-iff-is-unit-JNF[OF A] by auto
 hence 1: det A ≠ 0 by auto
 have det A = (∑ i<n. A $$ (i, 0) * Determinant.cofactor A i 0)
  by (rule laplace-expansion-column[OF A n0])
 also have ... = 0
  by (rule sum.neutral, insert col-A0 n0 A, auto simp add: col-def,
     metis Matrix.zero-vec-def index-vec mult-zero-left)
 finally show False using 1 by contradiction
qed

lemma invertible-mat-mult-int:
 assumes A = P * B
  and P ∈ carrier-mat n n
  and B ∈ carrier-mat n n
  and invertible-mat P
  and invertible-mat (map-mat rat-of-int B)
 shows invertible-mat (map-mat rat-of-int A)
 by (metis (no-types, opaque-lifting) assms dvd-field-iff
     invertible-iff-is-unit-JNF invertible-mult-JNF map-carrier-mat not-is-unit-0
     of-int-hom.hom-0 of-int-hom.hom-det of-int-hom.mat-hom-mult)
```

lemma *echelon-form-JNF-intro*:
assumes ($\forall$ *i*<*dim-row A*. *is-zero-row-JNF i A* $\longrightarrow$ $\neg$ ($\exists$ *j*. *j* < *dim-row A* $\wedge$ *j*>*i* $\wedge$ $\neg$ *is-zero-row-JNF j A*))
and ($\forall$ *i j*. *i*<*j* $\wedge$ *j*<*dim-row A* $\wedge$ $\neg$ (*is-zero-row-JNF i A*) $\wedge$ $\neg$ (*is-zero-row-JNF j A*)
$\longrightarrow$ ((*LEAST n*. *A* $$ (*i*, *n*) $\neq$ *0*) < (*LEAST n*. *A* $$ (*j*, *n*) $\neq$ *0*)))
shows *echelon-form-JNF A* **using** *assms* **unfolding** *echelon-form-JNF-def* **by** *simp*

lemma *echelon-form-submatrix*:
assumes *ef-H*: *echelon-form-JNF H* **and** *H*: *H* $\in$ *carrier-mat m n*
and *k*: *k* $\leq$ *min m n*
shows *echelon-form-JNF* (*submatrix H* {*0*..<*k*} {*0*..<*k*})
proof −
let *?I* = {*0*..<*k*}
let *?H* = *submatrix H ?I ?I*
have *km*: *k*≤*m* **and** *kn*: *k*≤*n* **using** *k* **by** *simp*+
then have {*i*. *i* < *m* $\wedge$ *i* < *k*} = {..<*k*} {*i*. *i* < *n* $\wedge$ *i* < *k*} = {..<*k*} **by** *auto*
then have *card-mk*: *card* {*i*. *i* < *m* $\wedge$ *i* < *k*} = *k* **and** *card-nk*: *card* {*i*. *i* < *n* $\wedge$ *i* < *k*} = *k*
by *auto*
have *H-ij*: *H* $$ (*i*,*j*) = (*submatrix H ?I ?I*) $$ (*i*,*j*) **if** *i*: *i*<*k* **and** *j*: *j*<*k* **for** *i j*
proof−
have *pick-j*: *pick ?I j* = *j* **by** (*rule pick-first-id*[*OF j*])
have *pick-i*: *pick ?I i* = *i* **by** (*rule pick-first-id*[*OF i*])
have *submatrix H ?I ?I* $$ (*i*, *j*) = *H* $$ (*pick ?I i*, *pick ?I j*)
by (*rule submatrix-index*, *insert H i j card-mk card-nk*, *auto*)
also have ... = *H* $$ (*i*,*j*) **using** *pick-i pick-j* **by** *simp*
finally show *?thesis* **..**
qed
have *H'*[*simp*]: *?H* $\in$ *carrier-mat k k*
using *H dim-submatrix*[*of H* {*0*..<*k*} {*0*..<*k*}] *card-mk card-nk* **by** *auto*
show *?thesis*
proof (*rule echelon-form-JNF-intro*, *auto*)
fix *i j* **assume** *iH'-0*: *is-zero-row-JNF i ?H* **and** *ij*: *i* < *j* **and** *j*: *j* < *dim-row ?H*
have *jm*: *j*<*m*
by (*metis H' carrier-matD*(*1*) *j km le-eq-less-or-eq nat-SN.gt-trans*)
show *is-zero-row-JNF j ?H*
proof (*rule ccontr*)
assume *j-not0-H'*: $\neg$ *is-zero-row-JNF j ?H*
define *a* **where** *a* = (*LEAST n*. *?H* $$ (*j*,*n*) $\neq$ *0*)
have *H'-ja*: *?H* $$ (*j*,*a*) $\neq$ *0*
by (*metis* (*mono-tags*) *LeastI j-not0-H' a-def is-zero-row-JNF-def*)
have *a*: *a* < *dim-col ?H*
by (*smt* (*verit*) *j-not0-H' a-def is-zero-row-JNF-def linorder-neqE-nat*

not-less-Least order-trans-rules(*19*))
have *j-not0-H*: ¬ *is-zero-row-JNF j H*
by (*metis H′ H′-ja H-ij a assms*(*2*) *basic-trans-rules*(*19*) *carrier-matD is-zero-row-JNF-def j kn le-eq-less-or-eq*)
hence *i-not0-H*: ¬ *is-zero-row-JNF i H* **using** *ef-H j ij* **unfolding** *echelon-form-JNF-def*
by (*metis H′* ‹¬ *is-zero-row-JNF j H*› *assms*(*2*) *carrier-matD*(*1*) *ij j km not-less-iff-gr-or-eq order.strict-trans order-trans-rules*(*21*))
hence *least-ab*: (*LEAST n. H* $$ (*i*, *n*) ≠ *0*) < (*LEAST n. H* $$ (*j*, *n*) ≠ *0*) **using** *jm*
using *j-not0-H assms*(*2*) *echelon-form-JNF-def ef-H ij* **by** *blast*
define *b* **where** *b* = (*LEAST n. H* $$ (*i*, *n*) ≠ *0*)
have *H-ib*: *H* $$ (*i*, *b*) ≠ *0*
by (*metis* (*mono-tags*, *lifting*) *LeastI b-def i-not0-H is-zero-row-JNF-def*)
have *b*: *b* < *dim-col ?H* **using** *least-ab a* **unfolding** *a-def b-def*
by (*metis* (*mono-tags*, *lifting*) *H′ H′-ja H-ij a-def carrier-matD dual-order.strict-trans j nat-neq-iff not-less-Least*)
have *H′-ib*: *?H* $$ (*i*,*b*) ≠ *0* **using** *H-ib b H-ij H′ ij j*
by (*metis H′ carrier-matD dual-order.strict-trans ij j*)
hence ¬ *is-zero-row-JNF i ?H* **using** *b is-zero-row-JNF-def* **by** *blast*
thus *False* **using** *iH′-0* **by** *contradiction*
qed
next
fix *i j* **assume** *ij*: *i* < *j* **and** *j*: *j* < *dim-row ?H*
have *jm*: *j*<*m*
by (*metis H′ carrier-matD*(*1*) *j km le-eq-less-or-eq nat-SN.gt-trans*)
assume *not0-iH′*: ¬ *is-zero-row-JNF i ?H*
and *not0-jH′*: ¬ *is-zero-row-JNF j ?H*
define *a* **where** *a* = (*LEAST n. ?H* $$ (*i*, *n*) ≠ *0*)
define *b* **where** *b* = (*LEAST n. ?H* $$ (*j*, *n*) ≠ *0*)
have *H′-ia*: *?H* $$ (*i*,*a*) ≠ *0*
by (*metis* (*mono-tags*) *LeastI-ex a-def is-zero-row-JNF-def not0-iH′*)
have *H′-jb*: *?H* $$ (*j*,*b*) ≠ *0*
by (*metis* (*mono-tags*) *LeastI-ex b-def is-zero-row-JNF-def not0-jH′*)
have *a*: *a* < *dim-row ?H*
by (*smt* (*verit*) *H′ a-def carrier-matD is-zero-row-JNF-def less-trans linorder-neqE-nat not0-iH′ not-less-Least*)
have *b*: *b* < *dim-row ?H*
by (*smt* (*verit*) *H′ b-def carrier-matD is-zero-row-JNF-def less-trans linorder-neqE-nat not0-jH′ not-less-Least*)
have *a-eq*: *a* = (*LEAST n. H* $$ (*i*, *n*) ≠ *0*)
by (*smt* (*verit*) *H′ H′-ia H-ij LeastI-ex a a-def carrier-matD*(*1*) *ij j linorder-neqE-nat not-less-Least order-trans-rules*(*19*))
have *b-eq*: *b* = (*LEAST n. H* $$ (*j*, *n*) ≠ *0*)
by (*smt* (*verit*) *H′ H′-jb H-ij LeastI-ex b b-def carrier-matD*(*1*) *ij j linorder-neqE-nat not-less-Least order-trans-rules*(*19*))
have *not0-iH*: ¬ *is-zero-row-JNF i H*
by (*metis H′ H′-ia H-ij a H carrier-matD ij is-zero-row-JNF-def j kn le-eq-less-or-eq order.strict-trans*)

```
    have not0-jH: ¬ is-zero-row-JNF j H
    by (metis H' H'-jb H-ij b H carrier-matD is-zero-row-JNF-def j kn le-eq-less-or-eq
order.strict-trans)
    show (LEAST n. ?H $$ (i, n) ≠ 0) < (LEAST n. ?H $$ (j, n) ≠ 0)
     unfolding a-def[symmetric] b-def[symmetric] a-eq b-eq using not0-iH not0-jH
ef-H ij jm H
      unfolding echelon-form-JNF-def by auto
  qed
qed


lemma HNF-submatrix:
  assumes HNF-H: Hermite-JNF associates res H and H: H ∈ carrier-mat m n
  and k: k ≤ min m n
  shows Hermite-JNF associates res (submatrix H {0..<k} {0..<k})
proof −
  let ?I = {0..<k}
  let ?H = submatrix H ?I ?I
  have km: k≤m and kn: k≤n using k by simp+
  then have {i. i < m ∧ i < k} = {..<k} {i. i < n ∧ i < k} = {..<k} by auto
  then have card-mk: card {i. i < m ∧ i < k} = k and card-nk: card {i. i < n
∧ i < k} = k
    by auto
  have H-ij: H $$ (i,j) = (submatrix H ?I ?I) $$ (i,j)  if i: i<k and j: j<k for i j
  proof−
    have pick-j: pick ?I j = j by (rule pick-first-id[OF j])
    have pick-i: pick ?I i = i by (rule pick-first-id[OF i])
    have submatrix H ?I ?I $$ (i, j) = H $$ (pick ?I i, pick ?I j)
      by (rule submatrix-index, insert H i j card-mk card-nk, auto)
    also have ... = H $$ (i,j) using pick-i pick-j by simp
    finally show ?thesis ..
  qed
  have H'[simp]: ?H ∈ carrier-mat k k
    using H dim-submatrix[of H {0..<k} {0..<k}] card-mk card-nk by auto
  have CS-ass: Complete-set-non-associates associates using HNF-H unfolding
Hermite-JNF-def by simp
  moreover have CS-res: Complete-set-residues res  using HNF-H unfolding
Hermite-JNF-def by simp
  have ef-H: echelon-form-JNF H using HNF-H unfolding Hermite-JNF-def by
auto
  have ef-H': echelon-form-JNF ?H
    by (rule echelon-form-submatrix[OF ef-H H k])
  have HNF1: ?H $$ (i, LEAST n. ?H $$ (i, n) ≠ 0) ∈ associates
    and HNF2: (∀ j<i. ?H $$ (j, LEAST n. ?H $$ (i, n) ≠ 0)
              ∈ res (?H $$ (i, LEAST n. ?H $$ (i, n) ≠ 0)))
    if i: i<dim-row ?H and not0-iH': ¬ is-zero-row-JNF i ?H for i
  proof −
    define a where a = (LEAST n. ?H $$ (i, n) ≠ 0)
    have im: i<m
```

```
    by (metis H' carrier-matD(1) km order.strict-trans2 that(1))
  have H'-ia: ?H $$ (i,a) ≠ 0
    by (metis (mono-tags) LeastI-ex a-def is-zero-row-JNF-def not0-iH')
  have a: a < dim-row ?H
  by (smt (verit) H' a-def carrier-matD is-zero-row-JNF-def less-trans linorder-neqE-nat
not0-iH' not-less-Least)
  have a-eq: a = (LEAST n. H $$ (i, n) ≠ 0)
  by (smt (verit) H' H'-ia H-ij LeastI-ex a a-def carrier-matD(1) i linorder-neqE-nat
not-less-Least order-trans-rules(19))
   have H'-ia-H-ia: ?H $$ (i, a) = H $$ (i, a)  by (metis H' H-ij a car-
rier-matD(1) i)
  have not'-iH: ¬ is-zero-row-JNF i H
    by (metis H' H'-ia H'-ia-H-ia a assms(2) carrier-matD(1) carrier-matD(2)
is-zero-row-JNF-def kn order.strict-trans2)
  thus ?H $$ (i, LEAST n. ?H $$ (i, n) ≠ 0) ∈ associates using im
   by (metis H'-ia-H-ia Hermite-JNF-def a-def a-eq HNF-H H carrier-matD(1))
  show (∀ j<i. ?H $$ (j, LEAST n. ?H $$ (i, n) ≠ 0)
          ∈ res (?H $$ (i, LEAST n. ?H $$ (i, n) ≠ 0)))
  proof −
    { fix nn :: nat
  have ff1: ∀ n. ?H $$ (n, a) = H $$ (n, a) ∨ ¬ n < k
    by (metis (no-types) H' H-ij a carrier-matD(1))
    have ff2: i < k
  by (metis H' carrier-matD(1) that(1))
  then have H $$ (nn, a) ∈ res (H $$ (i, a)) ⟶ H $$ (nn, a) ∈ res (?H $$
(i, a))
  using ff1 by (metis (no-types))
    moreover
    { assume H $$ (nn, a) ∈ res (?H $$ (i, a))
     then have ?H $$ (nn, a) = H $$ (nn, a) ⟶ ?H $$ (nn, a) ∈ res (?H $$
(i, a))
        by presburger
       then have ¬ nn < i ∨ ?H $$ (nn, LEAST n. ?H $$ (i, n) ≠ 0) ∈ res
(?H $$ (i, LEAST n. ?H $$ (i, n) ≠ 0))
        using ff2 ff1 a-def order.strict-trans by blast }
      ultimately have ¬ nn < i ∨ ?H $$ (nn, LEAST n. ?H $$ (i, n) ≠ 0) ∈
res (?H $$ (i, LEAST n. ?H $$ (i, n) ≠ 0))
       using Hermite-JNF-def a-eq assms(1) assms(2) im not'-iH by blast }
    then show ?thesis
      by meson
  qed
 qed
 show ?thesis using HNF1 HNF2 ef-H' CS-res CS-ass unfolding Hermite-JNF-def
by blast
qed

lemma HNF-of-HNF-id:
 fixes H :: int mat
 assumes HNF-H: Hermite-JNF associates res H
```

and *H*: $H \in$ *carrier-mat n n*
and *H-P1-H1*: $H = P1 * H1$
and *inv-P1*: *invertible-mat P1*
and *H1*: $H1 \in$ *carrier-mat n n*
and *P1*: $P1 \in$ *carrier-mat n n*
and *HNF-H1*: *Hermite-JNF associates res H1*
and *inv-H*: *invertible-mat* (*map-mat rat-of-int H*)
shows $H1 = H$
proof (*rule HNF-unique-generalized-JNF*[*OF H P1 H1 - H H-P1-H1*])
show $H = (1_m\ n) * H$ **using** *H* **by** *auto*
qed (*insert assms*, *auto*)

context
fixes *n* :: *nat*
begin

interpretation *vec-module TYPE*(*int*) **.**

lemma *lattice-is-monotone*:
fixes *S T*
assumes *S*: *set S* $\subseteq$ *carrier-vec n*
assumes *T*: *set T* $\subseteq$ *carrier-vec n*
assumes *subs*: *set S* $\subseteq$ *set T*
shows *lattice-of S* $\subseteq$ *lattice-of T*
proof −
have $\exists$*fa*. *lincomb fa* (*set T*) = *lincomb f* (*set S*) **for** *f*
proof −
let *?f* = λ*i*. *if i* $\in$ *set T* − *set S then 0 else f i*
have *set-T-eq*: *set T* = *set S* $\cup$ (*set T* − *set S*) **using** *subs* **by** *blast*
have *l0*: *lincomb ?f* (*set T* − *set S*) = 0_v *n* **by** (*rule lincomb-zero*, *insert T*, *auto*)
have *lincomb ?f* (*set T*) = *lincomb ?f* (*set S* $\cup$ (*set T* − *set S*)) **using** *set-T-eq* **by** *simp*
also have ... = *lincomb ?f* (*set S*) + *lincomb ?f* (*set T* − *set S*)
by (*rule lincomb-union*, *insert S T subs*, *auto*)
also have ... = *lincomb ?f* (*set S*) **using** *l0* **by** (*auto simp add*: *S*)
also have ... = *lincomb f* (*set S*) **using** *S* **by** *fastforce*
finally show *?thesis* **by** *blast*
qed
thus *?thesis* **unfolding** *lattice-of-altdef-lincomb*[*OF S*] *lattice-of-altdef-lincomb*[*OF T*]
by *auto*
qed

lemma *lattice-of-append*:
assumes *fs*: *set fs* $\subseteq$ *carrier-vec n*

assumes *gs*: *set gs* ⊆ *carrier-vec n*
shows *lattice-of* (*fs* @ *gs*) = *lattice-of* (*gs* @ *fs*)
proof −
have *fsgs*: *set* (*fs* @ *gs*) ⊆ *carrier-vec n* **using** *fs gs* **by** *auto*
have *gsfs*: *set* (*gs* @ *fs*) ⊆ *carrier-vec n* **using** *fs gs* **by** *auto*
show *?thesis*
unfolding *lattice-of-altdef-lincomb*[*OF fsgs*] *lattice-of-altdef-lincomb*[*OF gsfs*]
by *auto* (*metis Un-commute*)+
qed

lemma *lattice-of-append-cons*:
assumes *fs*: *set fs* ⊆ *carrier-vec n* **and** *v*: *v* ∈ *carrier-vec n*
shows *lattice-of* (*v* # *fs*) = *lattice-of* (*fs* @ [*v*])
proof −
have *v-fs*: *set* (*v* # *fs*) ⊆ *carrier-vec n* **using** *fs v* **by** *auto*
hence *fs-v*: *set* (*fs* @ [*v*]) ⊆ *carrier-vec n* **by** *simp*
show *?thesis*
unfolding *lattice-of-altdef-lincomb*[*OF v-fs*] *lattice-of-altdef-lincomb*[*OF fs-v*]
by *auto*
qed

lemma *already-in-lattice-subset*:
assumes *fs*: *set fs* ⊆ *carrier-vec n* **and** *inlattice*: *v* ∈ *lattice-of fs*
and *v*: *v* ∈ *carrier-vec n*
shows *lattice-of* (*v* # *fs*) ⊆ *lattice-of fs*
proof (*cases v*∈*set fs*)
case *True*
then show *?thesis*
by (*metis fs lattice-is-monotone set-ConsD subset-code*(*1*))
next
case *False* **note** *v-notin-fs* = *False*
obtain *g* **where** *v-g*: *lincomb g* (*set fs*) = *v*
using *lattice-of-altdef-lincomb*[*OF fs*] *inlattice* **by** *auto*
have *v-fs*: *set* (*v* # *fs*) ⊆ *carrier-vec n* **using** *v fs* **by** *auto*
have ∃*fa*. *lincomb fa* (*set fs*) = *lincomb f* (*insert v* (*set fs*)) **for** *f*
proof −
have *smult-rw*: *f v* ·$_v$ (*lincomb g* (*set fs*)) = *lincomb* (λ*w*. *f v* ∗ *g w*) (*set fs*)
by (*rule lincomb-smult*[*symmetric*, *OF fs*])
have *lincomb f* (*insert v* (*set fs*)) = *f v* ·$_v$ *v* + *lincomb f* (*set fs*)
by (*rule lincomb-insert2*[*OF - fs - v-notin-fs v*], *auto*)
also have ... = *f v* ·$_v$ (*lincomb g* (*set fs*)) + *lincomb f* (*set fs*) **using** *v-g* **by** *simp*
also have ... = *lincomb* (λ*w*. *f v* ∗ *g w*) (*set fs*) + *lincomb f* (*set fs*)
unfolding *smult-rw* **by** *auto*
also have ... = *lincomb* (λ*w*. (λ*w*. *f v* ∗ *g w*) *w* + *f w*) (*set fs*)
by (*rule lincomb-sum*[*symmetric*, *OF - fs*], *simp*)
finally show *?thesis* **by** *auto*
qed
thus *?thesis* **unfolding** *lattice-of-altdef-lincomb*[*OF v-fs*] *lattice-of-altdef-lincomb*[*OF*

fs] **by** *auto*
qed

lemma *already-in-lattice*:
assumes *fs*: *set fs* ⊆ *carrier-vec n* **and** *inlattice*: *v* ∈ *lattice-of fs*
and *v*: *v* ∈ *carrier-vec n*
shows *lattice-of fs* = *lattice-of* (*v* # *fs*)
proof −
have *dir1*: *lattice-of fs* ⊆ *lattice-of* (*v* # *fs*)
by (*intro lattice-is-monotone*, *insert fs v*, *auto*)
moreover have *dir2*: *lattice-of* (*v* # *fs*) ⊆ *lattice-of fs*
by (*rule already-in-lattice-subset*[*OF assms*])
ultimately show *?thesis* **by** *auto*
qed

lemma *already-in-lattice-append*:
assumes *fs*: *set fs* ⊆ *carrier-vec n* **and** *inlattice*: *lattice-of gs* ⊆ *lattice-of fs*
and *gs*: *set gs* ⊆ *carrier-vec n*
shows *lattice-of fs* = *lattice-of* (*fs* @ *gs*)
using *assms*
proof (*induct gs arbitrary*: *fs*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons a gs*)
note *fs* = *Cons.prems*(*1*)
note *inlattice* = *Cons.prems*(*2*)
note *gs* = *Cons.prems*(*3*)
have *gs-in-fs*: *lattice-of gs* ⊆ *lattice-of fs*
by (*meson basic-trans-rules*(*23*) *gs lattice-is-monotone local.Cons*(*3*) *set-subset-Cons*)
have *a*: *a* ∈ *lattice-of* (*fs* @ *gs*)
using *basis-in-latticeI fs gs gs-in-fs local.Cons*(*1*) *local.Cons*(*3*) **by** *auto*
have *lattice-of* (*fs* @ *a* # *gs*) = *lattice-of* ((*a* # *gs*) @ *fs*)
by (*rule lattice-of-append*, *insert fs gs*, *auto*)
also have ... = *lattice-of* (*a* # (*gs* @ *fs*)) **by** *auto*
also have ... = *lattice-of* (*a* # (*fs* @ *gs*))
by (*rule lattice-of-eq-set*, *insert gs fs*, *auto*)
also have ... = *lattice-of* (*fs* @ *gs*)
by (*rule already-in-lattice*[*symmetric*, *OF - a*], *insert fs gs*, *auto*)
also have ... = *lattice-of fs* **by** (*rule Cons.hyps*[*symmetric*, *OF fs gs-in-fs*], *insert gs*, *auto*)
finally show *?case* **..**
qed

lemma *zero-in-lattice*:
assumes *fs-carrier*: *set fs* ⊆ *carrier-vec n*
shows 0_v *n* ∈ *lattice-of fs*

proof −
have $\forall f.$ *lincomb* $(\lambda v.\ 0 * f\ v)$ (*set fs*) $= 0_v\ n$
using *fs-carrier lincomb-closed lincomb-smult lmult-0* **by** *presburger*
hence *lincomb* $(\lambda i.\ 0)$ (*set fs*) $= 0_v\ n$ **by** *fastforce*
thus *?thesis* **unfolding** *lattice-of-altdef-lincomb*[*OF fs-carrier*] **by** *auto*
qed

lemma *lattice-zero-rows-subset*:
assumes *H*: $H \in$ *carrier-mat a n*
shows *lattice-of* (*Matrix.rows* $(0_m\ m\ n)$) $\subseteq$ *lattice-of* (*Matrix.rows H*)
proof
let *?fs* = *Matrix.rows* $(0_m\ m\ n)$
let *?gs* = *Matrix.rows H*
have *fs-carrier*: *set ?fs* $\subseteq$ *carrier-vec n* **unfolding** *Matrix.rows-def* **by** *auto*
have *gs-carrier*: *set ?gs* $\subseteq$ *carrier-vec n* **using** *H* **unfolding** *Matrix.rows-def* **by** *auto*
fix *x* **assume** *x*: $x \in$ *lattice-of* (*Matrix.rows* $(0_m\ m\ n)$)
obtain *f* **where** *fx*: *lincomb* (*of-int* $\circ$ *f*) (*set* (*Matrix.rows* $(0_m\ m\ n)$)) $= x$
using *x lattice-of-altdef-lincomb*[*OF fs-carrier*] **by** *blast*
have *lincomb* (*of-int* $\circ$ *f*) (*set* (*Matrix.rows* $(0_m\ m\ n)$)) $= 0_v\ n$
unfolding *lincomb-def* **by** (*rule M.finsum-all0*, *unfold Matrix.rows-def*, *auto*)
hence $x = 0_v\ n$ **using** *fx* **by** *auto*
thus $x \in$ *lattice-of* (*Matrix.rows H*) **using** *zero-in-lattice*[*OF gs-carrier*] **by** *auto*

qed

lemma *lattice-of-append-zero-rows*:
assumes *H'*: $H' \in$ *carrier-mat m n*
and *H*: $H = H' @_r (0_m\ m\ n)$
shows *lattice-of* (*Matrix.rows H*) $=$ *lattice-of* (*Matrix.rows H'*)
proof −
have *Matrix.rows H* $=$ *Matrix.rows H'* @ *Matrix.rows* $(0_m\ m\ n)$
by (*unfold H*, *rule rows-append-rows*[*OF H'*], *auto*)
also have *lattice-of ...* $=$ *lattice-of* (*Matrix.rows H'*)
proof (*rule already-in-lattice-append*[*symmetric*])
show *lattice-of* (*Matrix.rows* $(0_m\ m\ n)$) $\subseteq$ *lattice-of* (*Matrix.rows H'*)
by (*rule lattice-zero-rows-subset*[*OF H'*])
qed (*insert H'*, *auto simp add*: *Matrix.rows-def*)
finally show *?thesis* **.**
qed
end

Lemmas about echelon form

lemma *echelon-form-JNF-1xn*:
assumes *A*$\in$*carrier-mat m n* **and** *m*$<$*2*
shows *echelon-form-JNF A*
using *assms* **unfolding** *echelon-form-JNF-def is-zero-row-JNF-def* **by** *fastforce*

lemma *echelon-form-JNF-mx1*:
assumes *A*∈*carrier-mat m n* **and** *n*<*2*
and ∀ *i* ∈ {*1*..<*m*}. *A*$$(*i*,*0*) = *0*
shows *echelon-form-JNF A*
using *assms* **unfolding** *echelon-form-JNF-def is-zero-row-JNF-def*
using *atLeastLessThan-iff less-2-cases* **by** *fastforce*

lemma *echelon-form-mx0*:
assumes *A* ∈ *carrier-mat m 0*
shows *echelon-form-JNF A* **using** *assms* **unfolding** *echelon-form-JNF-def is-zero-row-JNF-def*
by *auto*

lemma *echelon-form-JNF-first-column-0*:
assumes *eA*: *echelon-form-JNF A* **and** *A*: *A* ∈ *carrier-mat m n*
and *i0*: *0*<*i* **and** *im*: *i*<*m* **and** *n0*: *0*<*n*
shows *A* $$ (*i*,*0*) =*0*
proof (*rule ccontr*)
assume *Ai0*: *A* $$ (*i*, *0*) ≠ *0*
hence *nz-iA*: ¬ *is-zero-row-JNF i A* **using** *n0 A* **unfolding** *is-zero-row-JNF-def*
by *auto*
hence *nz-0A*: ¬ *is-zero-row-JNF 0 A* **using** *eA A* **unfolding** *echelon-form-JNF-def*
using *i0 im* **by** *auto*
have (*LEAST n. A* $$ (*0*, *n*) ≠ *0*) < (*LEAST n. A* $$ (*i*, *n*) ≠ *0*)
using *nz-iA nz-0A eA A* **unfolding** *echelon-form-JNF-def* **using** *i0 im* **by**
blast
moreover have (*LEAST n. A* $$ (*i*, *n*) ≠ *0*) = *0* **using** *Ai0* **by** *simp*
ultimately show *False* **by** *auto*
qed

lemma *is-zero-row-JNF-multrow*[*simp*]:
fixes *A*::*'a*::*comm-ring-1 mat*
assumes *i*<*dim-row A*
shows *is-zero-row-JNF i* (*multrow j* (− *1*) *A*) = *is-zero-row-JNF i A*
using *assms* **unfolding** *is-zero-row-JNF-def* **by** *auto*

lemma *echelon-form-JNF-multrow*:
assumes *A* : *carrier-mat m n* **and** *i*<*m* **and** *eA*: *echelon-form-JNF A*
shows *echelon-form-JNF* (*multrow i* (− *1*) *A*)
proof (*rule echelon-form-JNF-intro*)
have *A* $$ (*j*, *ja*) = *0* **if** ∀ *j'*<*dim-col A*. *A* $$ (*ia*, *j'*) = *0*
and *iaj*: *ia* < *j* **and** *j*: *j* < *dim-row A* **and** *ja*: *ja* < *dim-col A* **for** *ia j ja*
using *assms that* **unfolding** *echelon-form-JNF-def is-zero-row-JNF-def*
by (*meson order.strict-trans*)
thus ∀ *ia*<*dim-row* (*multrow i* (− *1*) *A*). *is-zero-row-JNF ia* (*multrow i* (− *1*)
A)

$\longrightarrow \neg$ ($\exists j$<*dim-row* (*multrow i* (− *1*) *A*). *ia* < *j* ∧ ¬ *is-zero-row-JNF j* (*multrow i* (− *1*) *A*))
unfolding *is-zero-row-JNF-def* **by** *simp*
have *Least-eq*: (*LEAST n. multrow i* (− *1*) *A* $$ (*ia*, *n*) ≠ *0*) = (*LEAST n. A* $$ (*ia*, *n*) ≠ *0*)
if *ia*: *ia* < *dim-row A* **and** *nz-ia-mrA*: ¬ *is-zero-row-JNF ia* (*multrow i* (− *1*) *A*) **for** *ia*
proof (*rule Least-equality*)
have *nz-ia-A*: ¬ *is-zero-row-JNF ia A* **using** *nz-ia-mrA ia* **by** *auto*
have *Least-Aian-n*: (*LEAST n. A* $$ (*ia*, *n*) ≠ *0*) < *dim-col A*
by (*smt* (*verit*) *dual-order.strict-trans is-zero-row-JNF-def not-less-Least not-less-iff-gr-or-eq nz-ia-A*)
show *multrow i* (− *1*) *A* $$ (*ia*, *LEAST n. A* $$ (*ia*, *n*) ≠ *0*) ≠ *0*
by (*smt* (*verit*) *LeastI Least-Aian-n class-cring.cring-simprules*(*22*) *equation-minus-iff ia*
index-mat-multrow(*1*) *is-zero-row-JNF-def mult-minus1 nz-ia-A*)
show ⋀*y*. *multrow i* (− *1*) *A* $$ (*ia*, *y*) ≠ *0* ⟹ (*LEAST n. A* $$ (*ia*, *n*) ≠ *0*) ≤ *y*
by (*metis* (*mono-tags*, *lifting*) *Least-Aian-n class-cring.cring-simprules*(*22*) *ia*
index-mat-multrow(*1*) *leI mult-minus1 order.strict-trans wellorder-Least-lemma*(*2*))
qed
have (*LEAST n. multrow i* (− *1*) *A* $$ (*ia*, *n*) ≠ *0*) < (*LEAST n. multrow i* (− *1*) *A* $$ (*j*, *n*) ≠ *0*)
if *ia-j*: *ia* < *j* **and**
j: *j* < *dim-row A*
and *nz-ia-A*: ¬ *is-zero-row-JNF ia A*
and *nz-j-A*: ¬ *is-zero-row-JNF j A*
for *ia j*
proof −
have *ia*: *ia* < *dim-row A* **using** *ia-j j* **by** *auto*
show *?thesis* **using** *Least-eq*[*OF ia*] *Least-eq*[*OF j*] *nz-ia-A nz-j-A*
is-zero-row-JNF-multrow[*OF ia*] *is-zero-row-JNF-multrow*[*OF j*] *eA ia-j j*
unfolding *echelon-form-JNF-def* **by** *simp*
qed
thus ∀ *ia j*.
ia < *j* ∧ *j* < *dim-row* (*multrow i* (− *1*) *A*) ∧ ¬ *is-zero-row-JNF ia* (*multrow i* (− *1*) *A*)
∧ ¬ *is-zero-row-JNF j* (*multrow i* (− *1*) *A*) ⟶
(*LEAST n. multrow i* (− *1*) *A* $$ (*ia*, *n*) ≠ *0*) < (*LEAST n. multrow i* (− *1*) *A* $$ (*j*, *n*) ≠ *0*)
by *auto*
qed

thm *echelon-form-imp-upper-triagular*

lemma *echelon-form-JNF-least-position-ge-diagonal*:
assumes *eA*: *echelon-form-JNF A*
and *A*: *A*: *carrier-mat m n*
and *nz-iA*: ¬ *is-zero-row-JNF i A*
and *im*: *i*<*m*
shows *i*≤(*LEAST n. A $$ (i,n) ≠ 0*)
using *nz-iA im*
proof (*induct i rule*: *less-induct*)
case (*less i*)
note *nz-iA* = *less.prems*(*1*)
note *im* = *less.prems*(*2*)
show *?case*
proof (*cases i=0*)
case *True* **show** *?thesis* **using** *True* **by** *blast*
next
case *False*
show *?thesis*
proof (*rule ccontr*)
assume ¬ *i* ≤ (*LEAST n. A $$ (i, n) ≠ 0*)
hence *i-least*: *i* > (*LEAST n. A $$ (i, n) ≠ 0*) **by** *auto*
have *nz-i1A*: ¬ *is-zero-row-JNF* (*i*−*1*) *A*
using *nz-iA im False A eA* **unfolding** *echelon-form-JNF-def*
by (*metis Num.numeral-nat*(*7*) *Suc-pred carrier-matD*(*1*) *gr-implies-not0*
lessI linorder-neqE-nat order.strict-trans)
have *i*−*1*≤(*LEAST n. A $$ (i*−*1,n) ≠ 0*) **by** (*rule less.hyps*, *insert im nz-i1A False*, *auto*)
moreover have (*LEAST n. A $$ (i,n) ≠ 0*) > (*LEAST n. A $$ (i*−*1,n) ≠ 0*)
using *nz-i1A nz-iA im False A eA* **unfolding** *echelon-form-JNF-def* **by** *auto*
ultimately show *False* **using** *i-least* **by** *auto*
qed
qed
qed

lemma *echelon-form-JNF-imp-upper-triangular*:
assumes *eA*: *echelon-form-JNF A*
shows *upper-triangular A*
proof
fix *i j* **assume** *ji*: *j*<*i* **and** *i*: *i*<*dim-row A*
have *A*: *A* ∈ *carrier-mat* (*dim-row A*) (*dim-col A*) **by** *auto*
show *A $$ (i,j)* = *0*
proof (*cases is-zero-row-JNF i A*)
case *False*
have *i*≤ (*LEAST n. A $$(i,n) ≠ 0*)
by (*rule echelon-form-JNF-least-position-ge-diagonal*[*OF eA A False i*])
then show *?thesis*
using *ji not-less-Least order.strict-trans2* **by** *blast*

next
case *True*

then show *?thesis* **unfolding** *is-zero-row-JNF-def* **oops**

lemma *echelon-form-JNF-imp-upper-triangular*:
assumes *eA*: *echelon-form-JNF A*
shows *upper-triangular' A*
proof
fix *i j* **assume** *ji*: *j<i* **and** *i*: *i<dim-row A* **and** *j*: *j<dim-col A*
have *A*: *A* ∈ *carrier-mat* (*dim-row A*) (*dim-col A*) **by** *auto*
show *A* $$ (*i*,*j*) = *0*
proof (*cases is-zero-row-JNF i A*)
case *False*
have *i*≤ (*LEAST n. A* $$(*i*,*n*) ≠ *0*)
by (*rule echelon-form-JNF-least-position-ge-diagonal*[*OF eA A False i*])
then show *?thesis*
using *ji not-less-Least order.strict-trans2* **by** *blast*
next
case *True*
then show *?thesis* **unfolding** *is-zero-row-JNF-def* **using** *j* **by** *auto*
qed
qed

lemma *upper-triangular-append-zero*:
assumes *uH*: *upper-triangular' H*
and *H*: *H* ∈ *carrier-mat* (*m*+*m*) *n* **and** *mn*: *n*≤*m*
shows *H* = *mat-of-rows n* (*map* (*Matrix.row H*) [*0*..<*m*]) @$_r$ *0*$_m$ *m n* (**is** - = *?H'* @$_r$ *0*$_m$ *m n*)
proof
have *H'*: *?H'* ∈ *carrier-mat m n* **using** *H uH* **by** *auto*
have *H'0*: (*?H'* @$_r$ *0*$_m$ *m n*) ∈ *carrier-mat* (*m*+*m*) *n* **by** (*simp add*: *H'*)
thus *dr*: *dim-row H* = *dim-row* (*?H'* @$_r$ *0*$_m$ *m n*) **using** *H H'* **by** (*simp add*: *append-rows-def*)
show *dc*: *dim-col H* = *dim-col* (*?H'* @$_r$ *0*$_m$ *m n*) **using** *H H'* **by** (*simp add*: *append-rows-def*)
fix *i j* **assume** *i*: *i* < *dim-row* (*?H'* @$_r$ *0*$_m$ *m n*) **and** *j*: *j* < *dim-col* (*?H'* @$_r$ *0*$_m$ *m n*)
show *H* $$ (*i*, *j*) = (*?H'* @$_r$ *0*$_m$ *m n*) $$ (*i*, *j*)
proof (*cases i<m*)
case *True*
have *H* $$ (*i*, *j*) = *?H'* $$ (*i*,*j*)
by (*metis True H' append-rows-def H carrier-matD index-mat-four-block*(*3*) *index-zero-mat*(*3*) *j*
le-add1 map-first-rows-index mat-of-rows-carrier(*2*) *mat-of-rows-index nat-arith.rule0*)

then show *?thesis*
by (*metis* (*mono-tags*, *lifting*) *H′ True add.comm-neutral append-rows-def carrier-matD*(*1*) *i index-mat-four-block index-zero-mat*(*3*) *j*)
next
case *False*
have *imn*: *i*<*m*+*m* **using** *i dr H* **by** *auto*
have *jn*: *j*<*n* **using** *j dc H* **by** *auto*
have *ji*: *j*<*i* **using** *j i False mn jn* **by** *linarith*
hence *H* $$ (*i*, *j*) = *0* **using** *uH* **unfolding** *upper-triangular′-def dr imn* **using** *i jn*
by (*simp add*: *dc j*)
also have ... = (*?H′* @$_r$ *0*$_m$ *m n*) $$ (*i*, *j*)
by (*smt* (*verit*) *False H′ append-rows-def assms*(*2*) *carrier-matD*(*1*) *carrier-matD*(*2*) *dc imn*
index-mat-four-block(*1*,*3*) *index-zero-mat j less-diff-conv2 linorder-not-less*)
finally show *?thesis* **.**
qed
qed

8.2.3 The algorithm is sound

lemma *find-fst-non0-in-row*:
assumes *A*: *A* ∈ *carrier-mat m n*
and *res*: *find-fst-non0-in-row l A* = *Some j*
shows *A* $$ (*l*,*j*) ≠ *0 l* ≤ *j j* < *dim-col A*
proof −
let *?xs* = *filter* (λ*j*. *A* $$ (*l*, *j*) ≠ *0*) [*l* ..< *dim-col A*]
from *res*[*unfolded find-fst-non0-in-row-def Let-def*]
have *xs*: *?xs* ≠ [] **by** (*cases ?xs*, *auto*)
have *j-in-xs*: *j* ∈ *set ?xs* **using** *res* **unfolding** *find-fst-non0-in-row-def Let-def*
by (*metis* (*no-types*, *lifting*) *length-greater-0-conv list.case*(*2*) *list.exhaust nth-mem option.simps*(*1*) *xs*)
show *A* $$ (*l*,*j*) ≠ *0 l* ≤ *j j* < *dim-col A* **using** *j-in-xs* **by** *auto*+
qed

lemma *find-fst-non0-in-row-zero-before*:
assumes *A*: *A* ∈ *carrier-mat m n*
and *res*: *find-fst-non0-in-row l A* = *Some j*
shows ∀ *j′*∈{*l*..<*j*}. *A* $$ (*l*,*j′*) = *0*
proof −
let *?xs* = *filter* (λ*j*. *A* $$ (*l*, *j*) ≠ *0*) [*l* ..< *dim-col A*]
from *res*[*unfolded find-fst-non0-in-row-def Let-def*]
have *xs*: *?xs* ≠ [] **by** (*cases ?xs*, *auto*)
have *j-in-xs*: *j* ∈ *set ?xs* **using** *res* **unfolding** *find-fst-non0-in-row-def Let-def*
by (*metis* (*no-types*, *lifting*) *length-greater-0-conv list.case*(*2*) *list.exhaust nth-mem option.simps*(*1*) *xs*)
have *j-xs0*: *j* = *?xs* ! *0*
by (*smt* (*verit*) *res*[*unfolded find-fst-non0-in-row-def Let-def*] *list.case*(*2*) *list.exhaust*

```
option.inject xs)
  show ∀j'∈{l..<j}. A $$ (l,j') = 0
  proof (rule+, rule ccontr)
    fix j' assume j': j' : {l..<j} and Alj': A $$ (l, j') ≠ 0
    have j'j: j'<j using j' by auto
    have j'-in-xs: j' ∈ set ?xs
     by (metis (mono-tags, lifting) A Set.member-filter j' Alj' res atLeastLessThan-iff
filter-set
         find-fst-non0-in-row(3) nat-SN.gt-trans set-upt)
    have l-rw: [l..<dim-col A] = [l ..<j] @[j..<dim-col A]
      using assms(1) assms(2) find-fst-non0-in-row(3) j' upt-append by auto
    have xs-rw: ?xs = filter (λj. A $$ (l, j) ≠ 0) ([l ..<j] @[j..<dim-col A])
      using l-rw by auto
    hence filter (λj. A $$ (l, j) ≠ 0) [l ..<j] = [] using j-xs0
     by (metis (no-types, lifting) Set.member-filter atLeastLessThan-iff filter-append
filter-set
         length-greater-0-conv nth-append nth-mem order-less-irrefl set-upt)
    thus False using j-xs0 j' j-xs0
      by (metis Set.member-filter filter-empty-conv filter-set j'-in-xs set-upt)
  qed
qed


corollary find-fst-non0-in-row-zero-before':
  assumes A: A ∈ carrier-mat m n
  and res: find-fst-non0-in-row l A = Some j
  and j' ∈ {l..<j}
  shows A $$ (l,j') = 0 using find-fst-non0-in-row-zero-before assms by auto

lemma find-fst-non0-in-row-LEAST:
  assumes A: A ∈ carrier-mat m n
  and ut-A: upper-triangular' A
  and res: find-fst-non0-in-row l A = Some j
  and lm: l<m
shows j = (LEAST n. A $$ (l,n) ≠ 0)
proof (rule Least-equality[symmetric])
  show  A $$ (l, j) ≠ 0 using res find-fst-non0-in-row(1) by blast
  show ⋀y. A $$ (l, y) ≠ 0 ⟹ j ≤ y
  proof (rule ccontr)
    fix y assume Aly: A $$ (l, y) ≠ 0 and jy: ¬ j ≤ y
    have yn: y < n
      by (metis A jy carrier-matD(2) find-fst-non0-in-row(3) leI less-imp-le-nat
nat-SN.compat res)
    have A $$(l,y) = 0
    proof (cases y∈{l..<j})
      case True
      show ?thesis by (rule find-fst-non0-in-row-zero-before'[OF A res True])
    next
      case False hence y<l using jy by auto
```

thus *?thesis* **using** *ut-A A lm* **unfolding** *upper-triangular'-def* **using** *yn* **by** *blast*
qed
thus *False* **using** *Aly* **by** *contradiction*
qed
qed

lemma *find-fst-non0-in-row-None'*:
assumes *A*: $A \in$ *carrier-mat m n*
and *lm*: $l<m$
shows (*find-fst-non0-in-row l A* = *None*) = ($\forall j \in \{l..<$*dim-col A*$\}$. *A* $$ $(l,j) = 0$) (**is** *?lhs* = *?rhs*)
proof
assume *res*: *?lhs*
let *?xs* = *filter* (λj. *A* $$ $(l, j) \neq 0$) [l ..< *dim-col A*]
from *res*[*unfolded find-fst-non0-in-row-def Let-def*]
have *xs*: *?xs* = [] **by** (*cases ?xs*, *auto*)
have *A* $$ $(l, j) = 0$ **if** *j*: $j \in \{l..<$*dim-col A*$\}$ **for** *j*
using *xs* **by** (*metis* (*mono-tags*, *lifting*) *empty-filter-conv j set-upt*)
thus *?rhs* **by** *blast*
next
assume *rhs*: *?rhs*
show *?lhs*
proof (*rule ccontr*)
assume *find-fst-non0-in-row l A* $\neq$ *None*
from *this* **obtain** *j* **where** *r*: *find-fst-non0-in-row l A* = *Some j* **by** *blast*
hence *A* $$ $(l,j) \neq 0$ **and** $l \leq j$ **and** $j<$*dim-col A* **using** *find-fst-non0-in-row*[*OF A r*] **by** *blast+*
thus *False* **using** *rhs* **by** *auto*
qed
qed

lemma *find-fst-non0-in-row-None*:
assumes *A*: $A \in$ *carrier-mat m n*
and *ut-A*: *upper-triangular' A*
and *lm*: $l<m$
shows (*find-fst-non0-in-row l A* = *None*) = (*is-zero-row-JNF l A*) (**is** *?lhs* = *?rhs*)
proof
assume *res*: *?lhs*
let *?xs* = *filter* (λj. *A* $$ $(l, j) \neq 0$) [l ..< *dim-col A*]
from *res*[*unfolded find-fst-non0-in-row-def Let-def*]
have *xs*: *?xs* = [] **by** (*cases ?xs*, *auto*)
have *A* $$ $(l, j) = 0$ **if** *j*: $j <$ *dim-col A* **for** *j*
proof (*cases* $j<l$)
case *True*
then show *?thesis* **using** *ut-A A lm j* **unfolding** *upper-triangular'-def* **by**

blast
next
case *False*
hence *j-ln*: $j \in \{l..<dim\text{-}col\ A\}$ **using** *A lm j* **by** *simp*
then show *?thesis* **using** *xs* **by** (*metis* (*mono-tags*, *lifting*) *empty-filter-conv set-upt*)
qed
thus *?rhs* **unfolding** *is-zero-row-JNF-def* **by** *blast*
next
assume *rhs*: *?rhs*
show *?lhs*
proof (*rule ccontr*)
assume *find-fst-non0-in-row l A* $\neq$ *None*
from *this* **obtain** *j* **where** *r*: *find-fst-non0-in-row l A* = *Some j* **by** *blast*
hence *A* $$ (*l*,*j*) $\neq$ *0* **and** *j*<*dim-col A* **using** *find-fst-non0-in-row*[*OF A r*] **by** *blast*+
hence ¬ *is-zero-row-JNF l A* **unfolding** *is-zero-row-JNF-def* **using** *lm A* **by** *auto*
thus *False* **using** *rhs* **by** *contradiction*
qed
qed

lemma *make-first-column-positive-preserves-dimensions*:
shows [*simp*]: *dim-row* (*make-first-column-positive A*) = *dim-row A*
and [*simp*]: *dim-col* (*make-first-column-positive A*) = *dim-col A*
by (*auto*)

lemma *make-first-column-positive-works*:
assumes *A*∈*carrier-mat m n* **and** *i*: *i*<*m* **and** *0*<*n*
shows *make-first-column-positive A* $$ (*i*,*0*) $\geq$ *0*
and *j*<*n* $\Longrightarrow$ *A* $$ (*i*,*0*) < *0* $\Longrightarrow$ (*make-first-column-positive A*) $$ (*i*,*j*) = − *A* $$ (*i*,*j*)
and *j*<*n* $\Longrightarrow$ *A* $$ (*i*,*0*) $\geq$ *0* $\Longrightarrow$ (*make-first-column-positive A*) $$ (*i*,*j*) = *A* $$ (*i*,*j*)
using *assms* **by** *auto*

lemma *make-first-column-positive-invertible*:
shows ∃ *P*. *invertible-mat P* ∧ *P* ∈ *carrier-mat* (*dim-row A*) (*dim-row A*)
∧ *make-first-column-positive A* = *P* ∗ *A*
proof −
let *?P* = *Matrix.mat* (*dim-row A*) (*dim-row A*)
(λ(*i*,*j*). *if i* = *j then if A* $$(*i*,*0*) < *0 then* − *1 else 1 else 0*::*int*)
have *invertible-mat ?P*
proof −
have (*map abs* (*diag-mat ?P*)) = *replicate* (*length* ((*map abs* (*diag-mat ?P*)))) *1*
by (*rule replicate-length-same*[*symmetric*], *auto simp add*: *diag-mat-def*)

hence *m-rw*: (*map abs* (*diag-mat ?P*)) = *replicate* (*dim-row A*) *1* **by** (*auto simp add*: *diag-mat-def*)
have *Determinant.det ?P* = *prod-list* (*diag-mat ?P*) **by** (*rule det-upper-triangular*, *auto*)
also have *abs ...* = *prod-list* (*map abs* (*diag-mat ?P*)) **unfolding** *prod-list-abs* **by** *blast*
also have *...* = *prod-list* (*replicate* (*dim-row A*) *1*) **using** *m-rw* **by** *simp*
also have *...* = *1* **by** *auto*
finally have |*Determinant.det ?P*| = *1* **by** *blast*
hence *Determinant.det ?P dvd 1* **by** *fastforce*
thus *?thesis* **using** *invertible-iff-is-unit-JNF mat-carrier* **by** *blast*
qed
moreover have *make-first-column-positive A* = *?P* ∗ *A* (**is** *?M* = -)
proof (*rule eq-matI*)
show *dim-row ?M* = *dim-row* (*?P* ∗ *A*) **and** *dim-col ?M* = *dim-col* (*?P* ∗ *A*) **by** *auto*
fix *i j* **assume** *i*: *i* < *dim-row* (*?P* ∗ *A*) **and** *j*: *j* < *dim-col* (*?P* ∗ *A*)
have *set-rw*: {*0*..<*dim-row A*} = *insert i* ({*0*..<*dim-row A*} − {*i*}) **using** *i* **by** *auto*
have *rw0*: (∑ *ia* ∈ {*0*..<*dim-row A* } − {*i*}. *Matrix.row ?P i* $*v ia* ∗ *col A j* $*v ia*) = *0*
by (*rule sum.neutral*, *insert i*, *auto*)
have (*?P* ∗ *A*) $$ (*i*, *j*) = *Matrix.row ?P i* · *col A j* **using** *i j* **by** *auto*
also have *...* = (∑ *ia* = *0*..<*dim-row A*. *Matrix.row ?P i* $*v ia* ∗ *col A j* $*v ia*)
unfolding *scalar-prod-def* **by** *auto*
also have *...* = (∑ *ia* ∈ *insert i* ({*0*..<*dim-row A*} − {*i*}). *Matrix.row ?P i* $*v ia* ∗ *col A j* $*v ia*)
using *set-rw* **by** *argo*
also have *...* = *Matrix.row ?P i* $*v i* ∗ *col A j* $*v i*
+ (∑ *ia* ∈ {*0*..<*dim-row A* } − {*i*}. *Matrix.row ?P i* $*v ia* ∗ *col A j* $*v ia*)
by (*rule sum.insert*, *auto*)
also have *...* = *Matrix.row ?P i* $*v i* ∗ *col A j* $*v i* **unfolding** *rw0* **by** *simp*
finally have ∗: (*?P* ∗ *A*) $$ (*i*, *j*) = *Matrix.row ?P i* $*v i* ∗ *col A j* $*v i* **.**
also have *...* = *?M* $$ (*i*,*j*)
by (*cases A* $$ (*i*, *0*) < *0*, *insert i j*, *auto simp add*: *col-def*)
finally show *?M* $$ (*i*, *j*) = (*?P* ∗ *A*) $$ (*i*, *j*) **..**
qed
moreover have *?P* ∈ *carrier-mat* (*dim-row A*) (*dim-row A*) **by** *auto*
ultimately show *?thesis* **by** *blast*
qed

locale *proper-mod-operation* = *mod-operation* +
assumes *dvd-gdiv-mult-right*[*simp*]: *b* > *0* ⟹ *b dvd a* ⟹ (*a gdiv b*) ∗ *b* = *a*
and *gmod-gdiv*: *y* > *0* ⟹ *x gmod y* = *x* − *x gdiv y* ∗ *y*
and *dvd-imp-gmod-0*: *0* < *a* ⟹ *a dvd b* ⟹ *b gmod a* = *0*
and *gmod-0-imp-dvd*: *a gmod b* = *0* ⟹ *b dvd a*
and *gmod-0*[*simp*]: *n gmod 0* = *n n* > *0* ⟹ *0 gmod n* = *0*
begin
lemma *reduce-alt-def-not0*:

assumes A $$ $(a,0) \neq 0$ **and** *pquvd*: $(p,q,u,v,d) =$ *euclid-ext2* $(A\$\$(a,0))$ $(A$ $$ $(b,0))$
shows *reduce a b D A* =
Matrix.mat (*dim-row A*) (*dim-col A*)
($\lambda(i,k)$. *if* $i = a$ *then let* $r = (p*A\$\$(a,k) + q*A\$\$(b,k))$ *in*
if $k = 0$ *then if D dvd r then D else r else r gmod D*
else if $i = b$ *then let* $r = u * A\$\$(a,k) + v * A\$\$(b,k)$ *in*
if $k = 0$ *then r else r gmod D*
else $A\$\$(i,k)$) (**is** - = *?rhs*)
and
reduce-abs a b D A =
Matrix.mat (*dim-row A*) (*dim-col A*)
($\lambda(i,k)$. *if* $i = a$ *then let* $r = (p*A\$\$(a,k) + q*A\$\$(b,k))$ *in*
if abs r $>$ *D then if* $k = 0 \wedge$ *D dvd r then D else r gmod D else r*
else if $i = b$ *then let* $r = u * A\$\$(a,k) + v * A\$\$(b,k)$ *in*
if abs r $>$ *D then r gmod D else r*
else $A\$\$(i,k)$) (**is** - = *?rhs-abs*)
proof −
have *reduce a b D A* =
(*case euclid-ext2* $(A\$\$(a,0))$ $(A$ $$ $(b,0))$ *of* $(p,q,u,v,d) \Rightarrow$
Matrix.mat (*dim-row A*) (*dim-col A*)
($\lambda(i,k)$. *if* $i = a$ *then let* $r = (p*A\$\$(a,k) + q*A\$\$(b,k))$ *in*
if $k = 0$ *then if D dvd r then D else r else r gmod D*
else if $i = b$ *then let* $r = u * A\$\$(a,k) + v * A\$\$(b,k)$ *in*
if $k = 0$ *then r else r gmod D*
else $A\$\$(i,k)$
)) **using** *assms* **by** *auto*
also have ... = *?rhs* **unfolding** *reduce.simps Let-def*
by (*rule eq-matI*, *insert pquvd*) (*metis* (*no-types*, *lifting*) *split-conv*)+
finally show *reduce a b D A = ?rhs* **.**
have *reduce-abs a b D A* =
(*case euclid-ext2* $(A\$\$(a,0))$ $(A$ $$ $(b,0))$ *of* $(p,q,u,v,d) \Rightarrow$
Matrix.mat (*dim-row A*) (*dim-col A*)
($\lambda(i,k)$. *if* $i = a$ *then let* $r = (p*A\$\$(a,k) + q*A\$\$(b,k))$ *in*
if abs r $>$ *D then if* $k = 0 \wedge$ *D dvd r then D else r gmod D else r*
else if $i = b$ *then let* $r = u * A\$\$(a,k) + v * A\$\$(b,k)$ *in*
if abs r $>$ *D then r gmod D else r*
else $A\$\$(i,k)$
)) **using** *assms* **by** *auto*
also have ... = *?rhs-abs* **unfolding** *reduce.simps Let-def*
by (*rule eq-matI*, *insert pquvd*) (*metis* (*no-types*, *lifting*) *split-conv*)+
finally show *reduce-abs a b D A = ?rhs-abs* **.**
qed

lemma *reduce-preserves-dimensions*:
shows [*simp*]: *dim-row* (*reduce a b D A*) = *dim-row A*
and [*simp*]: *dim-col* (*reduce a b D A*) = *dim-col A*

and [*simp*]: *dim-row* (*reduce-abs a b D A*) = *dim-row A*
and [*simp*]: *dim-col* (*reduce-abs a b D A*) = *dim-col A*
by (*auto simp add*: *Let-def split-beta*)

lemma *reduce-carrier*:
assumes *A* ∈ *carrier-mat m n*
shows (*reduce a b D A*) ∈ *carrier-mat m n*
and (*reduce-abs a b D A*) ∈ *carrier-mat m n*
by (*insert assms*, *auto simp add*: *Let-def split-beta*)

lemma *reduce-gcd*:
assumes *A*: *A* ∈ *carrier-mat m n* **and** *a*: *a*<*m* **and** *j*: *0*<*n*
and *Aaj*: *A* $$ (*a*,*0*) ≠ *0*
shows (*reduce a b D A*) $$ (*a*,*0*) = (*let r* = *gcd* (*A*$$(*a*,*0*)) (*A*$$(*b*,*0*)) *in if D dvd r then D else r*) (**is** *?lhs* = *?rhs*)
and (*reduce-abs a b D A*) $$ (*a*,*0*) = (*let r* = *gcd* (*A*$$(*a*,*0*)) (*A*$$(*b*,*0*)) *in if D* < *r then*
if D dvd r then D else r gmod D else r) (**is** *?lhs-abs* = *?rhs-abs*)
proof −
obtain *p q u v d* **where** *pquvd*: *euclid-ext2* (*A*$$(*a*,*0*)) (*A*$$(*b*,*0*)) = (*p*,*q*,*u*,*v*,*d*)
using *prod-cases5* **by** *blast*
have *p* ∗ *A* $$ (*a*, *0*) + *q* ∗ *A* $$ (*b*, *0*) = *d*
using *Aaj pquvd is-bezout-ext-euclid-ext2* **unfolding** *is-bezout-ext-def*
by (*smt* (*verit*) *Pair-inject bezout-coefficients-fst-snd euclid-ext2-def*)
also have ... = *gcd* (*A*$$(*a*,*0*)) (*A*$$(*b*,*0*)) **by** (*metis euclid-ext2-def pquvd prod.sel*(*2*))
finally have *pAaj-qAbj-gcd*: *p* ∗ *A* $$ (*a*, *0*) + *q* ∗ *A* $$ (*b*, *0*) = *gcd* (*A*$$(*a*,*0*)) (*A*$$(*b*,*0*)) **.**
let *?f* = (λ(*i*, *k*). *if i* = *a then let r* = *p* ∗ *A* $$ (*a*, *k*) + *q* ∗ *A* $$ (*b*, *k*) *in if k* = *0 then if D dvd r then D else r else r gmod D*
else if i = *b then let r* = *u* ∗ *A* $$ (*a*, *k*) + *v* ∗ *A* $$ (*b*, *k*) *in if k* = *0 then r else r gmod D else A* $$ (*i*, *k*))
have (*reduce a b D A*) $$ (*a*,*0*) = *Matrix.mat* (*dim-row A*) (*dim-col A*) *?f* $$ (*a*, *0*)
using *Aaj pquvd* **by** *auto*
also have ... = (*let r* = *p* ∗ *A* $$ (*a*, *0*) + *q* ∗ *A* $$ (*b*, *0*) *in if* (*0*::*nat*) = *0 then if D dvd r then D else r else r gmod D*)
using *A a j* **by** *auto*
also have ... = (*if D dvd gcd* (*A*$$(*a*,*0*)) (*A*$$(*b*,*0*)) *then D else*
gcd (*A*$$(*a*,*0*)) (*A*$$(*b*,*0*)))
by (*simp add*: *pAaj-qAbj-gcd*)
finally show *?lhs* = *?rhs* **by** *auto*
let *?g* = (λ(*i*, *k*). *if i* = *a then let r* = *p* ∗ *A* $$ (*a*, *k*) + *q* ∗ *A* $$ (*b*, *k*) *in*
if D < |*r*| *then if k* = *0* ∧ *D dvd r then D else r gmod D else r*
else if i = *b then let r* = *u* ∗ *A* $$ (*a*, *k*) + *v* ∗ *A* $$ (*b*, *k*) *in*
if D < |*r*| *then r gmod D else r else A* $$ (*i*, *k*))
have (*reduce-abs a b D A*) $$ (*a*,*0*) = *Matrix.mat* (*dim-row A*) (*dim-col A*) *?g* $$ (*a*, *0*)
using *Aaj pquvd* **by** *auto*

also have ... = (*let* $r = p * A \$\$ (a, 0) + q * A \$\$ (b, 0)$ *in if* $D < |r|$ *then*
if $(0::nat) = 0 \wedge D$ *dvd* r *then* D *else* r *gmod* D *else* r)
using A a j **by** *auto*
also have ... = (*if* $D < |gcd\ (A\$\$(a,0))\ (A\$\$(b,0))|$ *then if* D *dvd gcd* $(A\$\$(a,0))$ $(A\$\$(b,0))$ *then* D *else*
gcd $(A\$\$(a,0))\ (A\$\$(b,0))$ *gmod* D *else gcd* $(A\$\$(a,0))\ (A\$\$(b,0))$)
by (*simp add*: *pAaj-qAbj-gcd*)
finally show *?lhs-abs* = *?rhs-abs* **by** *auto*
qed

lemma *reduce-preserves*:
assumes A: $A \in$ *carrier-mat* m n **and** j: $j<n$
and *Aaj*: $A \$\$ (a,0) \neq 0$ **and** *ib*: $i \neq b$ **and** *ia*: $i \neq a$ **and** *im*: $i<m$
shows (*reduce* a b D A) $\$\$ (i,j) = A \$\$ (i,j)$ (**is** *?thesis1*)
and (*reduce-abs* a b D A) $\$\$ (i,j) = A \$\$ (i,j)$ (**is** *?thesis2*)
proof −
obtain p q u v d **where** *pquvd*: $(p,q,u,v,d) =$ *euclid-ext2* $(A\$\$(a,0))\ (A\$\$(b,0))$
using *prod-cases5* **by** *metis*
show *?thesis1* **unfolding** *reduce-alt-def-not0*[*OF Aaj pquvd*] **using** *ia im j A ib* **by** *auto*
show *?thesis2* **unfolding** *reduce-alt-def-not0*[*OF Aaj pquvd*] **using** *ia im j A ib* **by** *auto*
qed

lemma *reduce-0*:
assumes A: $A \in$ *carrier-mat* m n **and** a: $a<m$ **and** j: $0<n$ **and** b: $b<m$ **and** *ab*: $a \neq b$
and *Aaj*: $A \$\$ (a,0) \neq 0$
and D: $D \geq 0$
shows (*reduce* a b D A) $\$\$ (b,0) = 0$ (**is** *?thesis1*)
and (*reduce-abs* a b D A) $\$\$ (b,0) = 0$ (**is** *?thesis2*)
proof −
obtain p q u v d **where** *pquvd*: *euclid-ext2* $(A\$\$(a,0))\ (A\$\$(b,0)) = (p,q,u,v,d)$
using *prod-cases5* **by** *blast*
hence u: $u = -\ (A\$\$(b,0))$ *div gcd* $(A\$\$(a,0))\ (A\$\$(b,0))$
using *euclid-ext2-works*[*OF pquvd*] **by** *auto*
have v: $v = A\$\$(a,0)$ *div gcd* $(A\$\$(a,0))\ (A\$\$(b,0))$ **using** *euclid-ext2-works*[*OF pquvd*] **by** *auto*
have *uv0*: $u * A\$\$(a,0) + v * A\$\$(b,0) = 0$ **using** u v
proof −
have $\forall\, i\ ia.\ gcd\ (ia::int)\ i * (ia\ div\ gcd\ ia\ i) = ia$
by (*meson dvd-mult-div-cancel gcd-dvd1*)
then have $v * -\ A \$\$ (b, 0) = u * A \$\$ (a, 0)$
by (*metis* (*no-types*) *dvd-minus-iff dvd-mult-div-cancel gcd-dvd2 minus-minus mult.assoc mult.commute u v*)

```
    then show ?thesis
      by simp
  qed
  let ?f = (λ(i, k). if i = a then let r = p * A $$ (a, k) + q * A $$ (b, k) in
            if k = 0 then if D dvd r then D else r else r gmod D
              else if i = b then let r = u * A $$ (a, k) + v * A $$ (b, k) in
                  if k = 0 then r else r gmod D else A $$ (i, k))
  have (reduce a b D A) $$ (b,0) = Matrix.mat (dim-row A) (dim-col A) ?f $$ (b,
0)
    using Aaj pquvd by auto
  also have ... = (let r = u * A$$(a,0) + v * A$$(b,0) in r)
    using A a j ab b by auto
  also have ... = 0 using uv0 D
    by (smt (verit) gmod-0(1) gmod-0(2))
  finally show ?thesis1 .
  let ?g = (λ(i, k). if i = a then let r = p * A $$ (a, k) + q * A $$ (b, k) in
          if D < |r| then if k = 0 ∧ D dvd r then D else r gmod D else r
              else if i = b then let r = u * A $$ (a, k) + v * A $$ (b, k) in
                  if D < |r| then r gmod D else r else A $$ (i, k))
  have (reduce-abs a b D A) $$ (b,0) = Matrix.mat (dim-row A) (dim-col A) ?g
$$ (b, 0)
    using Aaj pquvd by auto
  also have ... = (let r = u * A$$(a,0) + v * A$$(b,0) in if D < |r| then r gmod
D else r)
    using A a j ab b by auto
  also have ... = 0 using uv0 D by simp
  finally show ?thesis2 .
qed
end
```

Let us show the key lemma: operations modulo determinant don't modify the (integer) row span.

```
context LLL-with-assms
begin

lemma lattice-of-kId-subset-fs-init:
  assumes k-det: k = Determinant.det (mat-of-rows n fs-init)
  and mn: m=n
  shows lattice-of (Matrix.rows (k ·m (1m m))) ⊆ lattice-of fs-init
proof −
  let ?Z = (mat-of-rows n fs-init)
  let ?RAT = of-int-hom.mat-hom :: int mat ⇒ rat mat
  have RAT-fs-init: ?RAT (mat-of-rows n fs-init) ∈ carrier-mat n n
     using len map-carrier-mat mat-of-rows-carrier(1) mn by blast
  have det-RAT-fs-init: Determinant.det (?RAT ?Z) ≠ 0
  proof (rule gs.lin-indpt-rows-imp-det-not-0[OF RAT-fs-init])
    have rw: Matrix.rows (?RAT (mat-of-rows n fs-init)) = RAT fs-init
    by (metis cof-vec-space.lin-indpt-list-def fs-init lin-dep mat-of-rows-map rows-mat-of-rows)
    thus gs.lin-indpt (set (Matrix.rows (?RAT (mat-of-rows n fs-init))))
```

by (*insert lin-dep*, *simp add*: *cof-vec-space.lin-indpt-list-def*)
show *distinct* (*Matrix.rows* (*?RAT* (*mat-of-rows n fs-init*)))
using *rw cof-vec-space.lin-indpt-list-def lin-dep* **by** *auto*
qed
obtain *inv-Z* **where** *inverts-Z*: *inverts-mat* (*?RAT ?Z*) *inv-Z* **and** *inv-Z*: *inv-Z* ∈ *carrier-mat m m*
by (*metis mn det-RAT-fs-init dvd-field-iff invertible-iff-is-unit-JNF*
len map-carrier-mat mat-of-rows-carrier(*1*) *obtain-inverse-matrix*)
have *det-rat-Z-k*: *Determinant.det* (*?RAT ?Z*) = *rat-of-int k*
using *k-det of-int-hom.hom-det* **by** *blast*
have *?RAT ?Z* ∗ *adj-mat* (*?RAT ?Z*) = *Determinant.det* (*?RAT ?Z*) ·$_m$ *1*$_m$ *n*
by (*rule adj-mat*[*OF RAT-fs-init*])
hence *inv-Z* ∗ (*?RAT ?Z* ∗ *adj-mat* (*?RAT ?Z*)) = *inv-Z* ∗ (*Determinant.det* (*?RAT ?Z*) ·$_m$ *1*$_m$ *n*) **by** *simp*
hence *k-inv-Z-eq-adj*: (*rat-of-int k*) ·$_m$ *inv-Z* = *adj-mat* (*?RAT ?Z*)
by (*smt* (*verit*) *Determinant.mat-mult-left-right-inverse RAT-fs-init adj-mat*(*1*,*3*) *mn*
carrier-matD det-RAT-fs-init det-rat-Z-k gs.det-nonzero-congruence inv-Z inverts-Z
inverts-mat-def mult-smult-assoc-mat smult-carrier-mat)
have *adj-mat-Z*: *adj-mat* (*?RAT ?Z*) $$ (*i*,*j*) ∈ ℤ **if** *i*: *i*<*m* **and** *j*: *j*<*n* **for** *i j*
proof −
have *det-mat-delete-Z*: *Determinant.det* (*mat-delete* (*?RAT ?Z*) *j i*) ∈ ℤ
proof (*rule Ints-det*)
fix *ia ja*
assume *ia*: *ia* < *dim-row* (*mat-delete* (*?RAT ?Z*) *j i*)
and *ja*: *ja* < *dim-col* (*mat-delete* (*?RAT ?Z*) *j i*)
have (*mat-delete* (*?RAT ?Z*) *j i*) $$ (*ia*, *ja*) = (*?RAT ?Z*) $$ (*insert-index j ia*, *insert-index i ja*)
by (*rule mat-delete-index*[*symmetric*], *insert i j mn len ia ja RAT-fs-init*, *auto*)
also have ... = *rat-of-int* (*?Z* $$ (*insert-index j ia*, *insert-index i ja*))
by (*rule index-map-mat*, *insert i j ia ja*, *auto simp add*: *insert-index-def*)
also have ... ∈ ℤ **using** *Ints-of-int* **by** *blast*
finally show (*mat-delete* (*?RAT ?Z*) *j i*) $$ (*ia*, *ja*) ∈ ℤ **.**
qed
have *adj-mat* (*?RAT ?Z*) $$ (*i*,*j*) = *Determinant.cofactor* (*?RAT ?Z*) *j i*
unfolding *adj-mat-def*
by (*simp add*: *len i j*)
also have ... = (− *1*) ^ (*j* + *i*) ∗ *Determinant.det* (*mat-delete* (*?RAT ?Z*) *j i*)
unfolding *Determinant.cofactor-def* **by** *auto*
also have ... ∈ ℤ **using** *det-mat-delete-Z* **by** *auto*
finally show *?thesis* **.**
qed
have *kinvZ-in-Z*: ((*rat-of-int k*) ·$_m$ *inv-Z*) $$ (*i*,*j*) ∈ ℤ **if** *i*: *i*<*m* **and** *j*: *j*<*n* **for** *i j*
using *k-inv-Z-eq-adj* **by** (*simp add*: *adj-mat-Z i j*)
have *?RAT* (*k* ·$_m$ (*1*$_m$ *m*)) = *Determinant.det* (*?RAT ?Z*) ·$_m$ (*inv-Z* ∗ *?RAT*

?Z) (**is** *?lhs = ?rhs*)
proof −
have (*inv-Z* ∗ *?RAT ?Z*) = (*1*$_m$ *m*)
by (*metis Determinant.mat-mult-left-right-inverse RAT-fs-init mn carrier-matD*(*1*) *inv-Z inverts-Z inverts-mat-def*)
from *this* **have** *?rhs = rat-of-int k* ·$_m$ (*1*$_m$ *m*) **using** *det-rat-Z-k* **by** *auto*
also have *... = ?lhs* **by** *auto*
finally show *?thesis* **..**
qed
also have *... = (Determinant.det* (*?RAT ?Z*) ·$_m$ *inv-Z*) ∗ *?RAT ?Z*
by (*metis RAT-fs-init mn inv-Z mult-smult-assoc-mat*)
also have *... = ((rat-of-int k)* ·$_m$ *inv-Z*) ∗ *?RAT ?Z* **by** (*simp add*: *k-det*)
finally have *r'*: *?RAT* (*k* ·$_m$ (*1*$_m$ *m*)) = ((*rat-of-int k*) ·$_m$ *inv-Z*) ∗ *?RAT ?Z* **.**
have *r*: (*k* ·$_m$ (*1*$_m$ *m*)) = ((*map-mat int-of-rat* ((*rat-of-int k*) ·$_m$ *inv-Z*))) ∗ *?Z*
proof −
have *?RAT* ((*map-mat int-of-rat* ((*rat-of-int k*) ·$_m$ *inv-Z*))) = ((*rat-of-int k*) ·$_m$ *inv-Z*)
proof (*rule eq-matI*, *auto*)
fix *i j* **assume** *i*: *i* < *dim-row inv-Z* **and** *j*: *j* < *dim-col inv-Z*
have ((*rat-of-int k*) ·$_m$ *inv-Z*) $$ (*i*,*j*) = (*rat-of-int k* ∗ *inv-Z* $$ (*i*, *j*))
using *index-smult-mat i j* **by** *auto*
hence *kinvZ-in-Z'*: *...* ∈ ℤ **using** *kinvZ-in-Z i j inv-Z mn* **by** *simp*
show *rat-of-int* (*int-of-rat* (*rat-of-int k* ∗ *inv-Z* $$ (*i*, *j*))) = *rat-of-int k* ∗ *inv-Z* $$ (*i*, *j*)
by (*rule int-of-rat*, *insert kinvZ-in-Z'*, *auto*)
qed
hence *?RAT* (*k* ·$_m$ (*1*$_m$ *m*)) = *?RAT* ((*map-mat int-of-rat* ((*rat-of-int k*) ·$_m$ *inv-Z*))) ∗ *?RAT ?Z*
using *r'* **by** *simp*
also have *... = ?RAT* ((*map-mat int-of-rat* ((*rat-of-int k*) ·$_m$ *inv-Z*)) ∗ *?Z*)
by (*metis RAT-fs-init adj-mat*(*1*) *k-inv-Z-eq-adj map-carrier-mat of-int-hom.mat-hom-mult*)
finally show *?thesis* **by** (*rule of-int-hom.mat-hom-inj*)
qed
show *?thesis*
proof (*rule mat-mult-sub-lattice*[*OF - fs-init*])
have *rw*: *of-int-hom.mat-hom* (*map-mat int-of-rat* ((*rat-of-int k*) ·$_m$ *inv-Z*))
= *map-mat int-of-rat* ((*rat-of-int k*) ·$_m$ *inv-Z*) **by** *auto*
have *mat-of-rows n* (*Matrix.rows* (*k* ·$_m$ *1*$_m$ *m*)) = (*k* ·$_m$ (*1*$_m$ *m*))
by (*metis mn index-one-mat*(*3*) *index-smult-mat*(*3*) *mat-of-rows-rows*)
also have *... = of-int-hom.mat-hom* (*map-mat int-of-rat* ((*rat-of-int k*) ·$_m$ *inv-Z*)) ∗ *mat-of-rows n fs-init*
using *r rw* **by** *auto*
finally show *mat-of-rows n* (*Matrix.rows* (*k* ·$_m$ *1*$_m$ *m*))
= *of-int-hom.mat-hom* (*map-mat int-of-rat* ((*rat-of-int k*) ·$_m$ *inv-Z*)) ∗ *mat-of-rows n fs-init* **.**
show *set* (*Matrix.rows* (*k* ·$_m$ *1*$_m$ *m*)) ⊆ *carrier-vec n***using** *mn* **unfolding** *Matrix.rows-def* **by** *auto*
show *map-mat int-of-rat* (*rat-of-int k* ·$_m$ *inv-Z*) ∈ *carrier-mat* (*length* (*Matrix.rows* (*k* ·$_m$ *1*$_m$ *m*))) (*length fs-init*)

```
    using len fs-init by (simp add: inv-Z)
  qed
qed

end

context LLL-with-assms
begin


lemma lattice-of-append-det-preserves:
  assumes k-det: k = abs (Determinant.det (mat-of-rows n fs-init))
  and mn: m = n
  and A: A = (mat-of-rows n fs-init) @_r (k ·_m (1_m m))
shows lattice-of (Matrix.rows A) = lattice-of fs-init
proof −
  have Matrix.rows (mat-of-rows n fs-init @_r k ·_m 1_m m) = (Matrix.rows (mat-of-rows
n fs-init) @ Matrix.rows (k ·_m (1_m m)))
    by (rule rows-append-rows, insert fs-init len mn, auto)
  also have ... = (fs-init @ Matrix.rows (k ·_m (1_m m))) by (simp add: fs-init)
  finally have rw: Matrix.rows (mat-of-rows n fs-init @_r k ·_m 1_m m)
    = (fs-init @ Matrix.rows (k ·_m (1_m m))) .
  have lattice-of (Matrix.rows A) = lattice-of (fs-init @ Matrix.rows (k ·_m (1_m
m)))
    by (rule arg-cong[of - - lattice-of], auto simp add: A rw)
  also have ... = lattice-of fs-init
  proof (cases k = Determinant.det (mat-of-rows n fs-init))
    case True
    then show ?thesis
    by (rule already-in-lattice-append[symmetric, OF fs-init
                lattice-of-kId-subset-fs-init[OF - mn]], insert mn, auto simp add:
Matrix.rows-def)
  next
    case False
    hence k2: k = −Determinant.det (mat-of-rows n fs-init) using k-det by auto
    have l: lattice-of (Matrix.rows (− k ·_m 1_m m)) ⊆ lattice-of fs-init
      by (rule lattice-of-kId-subset-fs-init[OF - mn], insert k2, auto)
    have l2: lattice-of (Matrix.rows (− k ·_m 1_m m)) = lattice-of (Matrix.rows (k
·_m 1_m m))
    proof (rule mat-mult-invertible-lattice-eq)
      let ?P = (− 1::int) ·_m 1_m m
      show P: ?P ∈ carrier-mat m m by simp
       have det ?P = 1 ∨ det ?P = −1 unfolding det-smult by (auto simp add:
minus-1-power-even)
      hence det ?P dvd 1 by (smt (verit) minus-dvd-iff one-dvd)
      thus  invertible-mat ?P unfolding invertible-iff-is-unit-JNF[OF P] .
      have (− k ·_m 1_m m) = ?P ∗ (k ·_m 1_m m)
        unfolding mat-diag-smult[symmetric] unfolding mat-diag-diag by auto
      thus  mat-of-rows n (Matrix.rows (− k ·_m 1_m m)) = of-int-hom.mat-hom ?P
```

```
∗ mat-of-rows n (Matrix.rows (k ·m 1m m))
          by (metis mn index-one-mat(3) index-smult-mat(3) mat-of-rows-rows
of-int-mat-hom-int-id)
      show  set (Matrix.rows (− k ·m 1m m)) ⊆ carrier-vec n
        and set (Matrix.rows (k ·m 1m m)) ⊆ carrier-vec n
       using assms(2) one-carrier-mat set-rows-carrier smult-carrier-mat by blast+
    qed (insert mn, auto)
    hence l2: lattice-of (Matrix.rows (k ·m 1m m)) ⊆ lattice-of fs-init using l by
auto
    show ?thesis by (rule already-in-lattice-append[symmetric, OF fs-init l2],
          insert mn one-carrier-mat set-rows-carrier smult-carrier-mat, blast)
  qed
  finally show ?thesis .
qed
```

This is another key lemma. Here, A is the initial matrix (*mat-of-rows n fs-init*) augmented with m rows $(k, 0, \ldots, 0), (0, k, 0, \ldots, 0), \ldots, (0, \ldots, 0, k)$ where k is the determinant of (*mat-of-rows n fs-init*). With the algorithm of the article, we obtain $H = H' \mathbin{@_r} (0_m\ m\ n)$ by means of an invertible matrix P (which is computable). Then, H is the HNF of A. The lemma shows that H' is the HNF of (*mat-of-rows n fs-init*) and that there exists an invertible matrix to carry out the transformation.

```
lemma Hermite-append-det-id:
  assumes k-det: k = abs (Determinant.det (mat-of-rows n fs-init))
  and mn: m = n
  and A: A = (mat-of-rows n fs-init) @r (k ·m (1m m))
  and H': H'∈ carrier-mat m n
  and H-append: H = H' @r (0m m n)
  and P: P ∈ carrier-mat (m+m) (m+m)
  and inv-P: invertible-mat P
  and A-PH: A = P ∗ H
  and HNF-H: Hermite-JNF associates res H
shows Hermite-JNF associates res H'
  and (∃ P'. invertible-mat P' ∧ P' ∈ carrier-mat m m ∧ (mat-of-rows n fs-init)
= P' ∗ H')
proof −
  have A-carrier: A ∈ carrier-mat (m+m) n using A mn len by auto
  let ?A' = (mat-of-rows n fs-init)
  let ?H' = submatrix H {0..<m} {0..<n}
  have nm: n≤m by (simp add: mn)
  have H: H ∈ carrier-mat (m + m) n using H-append H' by auto
  have submatrix-carrier: submatrix H {0..<m} {0..<n} ∈ carrier-mat m n
    by (rule submatrix-carrier-first[OF H], auto)
  have H'-eq: H' = ?H'
  proof (rule eq-matI)
    fix i j assume i: i < dim-row ?H' and j: j < dim-col ?H'
    have im: i<m and jn: j<n using i j submatrix-carrier by auto
    have ?H' $$ (i,j) = H $$ (i,j)
```

by (*rule submatrix-index-id*[*OF H*], *insert i j submatrix-carrier*, *auto*)
also have ... = (*if i* < *dim-row H′ then H′* $$ (*i*, *j*) *else* (*0*$_m$ *m n*) $$ (*i* − *m*, *j*))
unfolding *H-append* **by** (*rule append-rows-nth*[*OF H′*], *insert im jn*, *auto*)
also have ... = *H′* $$ (*i*,*j*) **using** *H′ im jn* **by** *simp*
finally show *H′* $$ (*i*, *j*) = *?H′* $$ (*i*, *j*) **..**
qed (*insert H′ submatrix-carrier*, *auto*)
show *HNF-H′*: *Hermite-JNF associates res H′*
unfolding *H′-eq mn* **by** (*rule HNF-submatrix*[*OF HNF-H H*], *insert nm*, *simp*)
have *L-fs-init-A*: *lattice-of* (*fs-init*) = *lattice-of* (*Matrix.rows A*)
by (*rule lattice-of-append-det-preserves*[*symmetric*, *OF k-det mn A*])
have *L-H′-H*: *lattice-of* (*Matrix.rows H′*) = *lattice-of* (*Matrix.rows H*)
using *H-append H′ lattice-of-append-zero-rows* **by** *blast*
have *L-A-H*: *lattice-of* (*Matrix.rows A*) = *lattice-of* (*Matrix.rows H*)
proof (*rule mat-mult-invertible-lattice-eq*[*OF - - P inv-P*])
show *set* (*Matrix.rows A*) ⊆ *carrier-vec n* **using** *A-carrier set-rows-carrier* **by** *blast*
show *set* (*Matrix.rows H*) ⊆ *carrier-vec n* **using** *H set-rows-carrier* **by** *blast*
show *length* (*Matrix.rows A*) = *m* + *m* **using** *A-carrier* **by** *auto*
show *length* (*Matrix.rows H*) = *m* + *m* **using** *H* **by** *auto*
show *mat-of-rows n* (*Matrix.rows A*) = *of-int-hom.mat-hom P* ∗ *mat-of-rows n* (*Matrix.rows H*)
by (*metis A-carrier H A-PH carrier-matD*(*2*) *mat-of-rows-rows of-int-mat-hom-int-id*)
qed
have *L-fs-init-H′*: *lattice-of fs-init* = *lattice-of* (*Matrix.rows H′*)
using *L-fs-init-A L-A-H L-H′-H* **by** *auto*
have *exists-P2*:
∃ *P2*. *P2* ∈ *carrier-mat n n* ∧ *invertible-mat P2* ∧ *mat-of-rows n* (*Matrix.rows H′*) = *P2* ∗ *H′*
by (*rule exI*[*of - 1*$_m$ *n*], *insert H′ mn*, *auto*)
have *exist-P′*: ∃ *P′*∈*carrier-mat n n*. *invertible-mat P′*
∧ *mat-of-rows n fs-init* = *P′* ∗ *mat-of-rows n* (*Matrix.rows H′*)
by (*rule eq-lattice-imp-mat-mult-invertible-rows*[*OF fs-init - lin-dep len*[*unfolded mn*] - *L-fs-init-H′*],
insert H′ mn set-rows-carrier, *auto*)
thus ∃ *P′*. *invertible-mat P′* ∧ *P′* ∈ *carrier-mat m m* ∧ (*mat-of-rows n fs-init*) = *P′* ∗ *H′*
by (*metis mn H′ carrier-matD*(*2*) *mat-of-rows-rows*)
qed
end

context *proper-mod-operation*
begin

definition *reduce-element-mod-D* (*A*::*int mat*) *a j D m* =
(*if j* = *0 then if D dvd A*$$(*a*,*j*) *then addrow* (−((*A*$$(*a*,*j*) *gdiv D*)) + *1*) *a* (*j* +

m) *A else A*
else addrow (−((*A*$$(*a*,*j*) *gdiv D*))) *a* (*j* + *m*) *A*)

definition *reduce-element-mod-D-abs* (*A*::*int mat*) *a j D m* =
(*if j* = *0* ∧ *D dvd A*$$(*a*,*j*) *then addrow* (−((*A*$$(*a*,*j*) *gdiv D*)) + *1*) *a* (*j* + *m*)
A
else addrow (−((*A*$$(*a*,*j*) *gdiv D*))) *a* (*j* + *m*) *A*)

lemma *reduce-element-mod-D-preserves-dimensions*:
shows [*simp*]: *dim-row* (*reduce-element-mod-D A a j D m*) = *dim-row A*
and [*simp*]: *dim-col* (*reduce-element-mod-D A a j D m*) = *dim-col A*
and [*simp*]: *dim-row* (*reduce-element-mod-D-abs A a j D m*) = *dim-row A*
and [*simp*]: *dim-col* (*reduce-element-mod-D-abs A a j D m*) = *dim-col A*
by (*auto simp add*: *reduce-element-mod-D-def reduce-element-mod-D-abs-def Let-def split-beta*)

lemma *reduce-element-mod-D-carrier*:
shows *reduce-element-mod-D A a j D m* ∈ *carrier-mat* (*dim-row A*) (*dim-col A*)

and *reduce-element-mod-D-abs A a j D m* ∈ *carrier-mat* (*dim-row A*) (*dim-col A*) **by** *auto*

lemma *reduce-element-mod-D-invertible-mat*:
assumes *A-def*: *A* = *A*′ @$_r$ (*D* ·$_m$ (*1*$_m$ *n*))
and *A*′: *A*′ ∈ *carrier-mat m n* **and** *a*: *a*<*m* **and** *j*: *j*<*n* **and** *mn*: *m*≥*n*
shows ∃ *P*. *P* ∈ *carrier-mat* (*m*+*n*) (*m*+*n*) ∧ *invertible-mat P* ∧
reduce-element-mod-D A a j D m = *P* ∗ *A* (**is** *?thesis1*)
and ∃ *P*. *P* ∈ *carrier-mat* (*m*+*n*) (*m*+*n*) ∧ *invertible-mat P* ∧
reduce-element-mod-D-abs A a j D m = *P* ∗ *A* (**is** *?thesis2*)
unfolding *atomize-conj*
proof (*rule conjI*; *cases j* = *0* ∧ *D dvd A*$$(*a*,*j*))
case *True*
let *?P* = *addrow-mat* (*m*+*n*) (− (*A* $$ (*a*, *j*) *gdiv D*) + *1*) *a* (*j* + *m*)
have *A*: *A* ∈ *carrier-mat* (*m* + *n*) *n* **using** *A-def A*′ *mn* **by** *auto*
have *reduce-element-mod-D A a j D m* = *addrow* (− (*A* $$ (*a*, *j*) *gdiv D*) + *1*)
a (*j* + *m*) *A*
unfolding *reduce-element-mod-D-def* **using** *True* **by** *auto*
also have ... = *?P* ∗ *A* **by** (*rule addrow-mat*[*OF A*], *insert j mn*, *auto*)
finally have *reduce-element-mod-D A a j D m* = *?P* ∗ *A* **.**
moreover have *P*: *?P* ∈ *carrier-mat* (*m*+*n*) (*m*+*n*) **by** *simp*
moreover have *inv-P*: *invertible-mat ?P*
by (*metis addrow-mat-carrier a det-addrow-mat dvd-mult-right*
invertible-iff-is-unit-JNF mult.right-neutral not-add-less2 semiring-gcd-class.gcd-dvd1)
ultimately show *?thesis1* **by** *blast*
have *reduce-element-mod-D-abs A a j D m* = *addrow* (− (*A* $$ (*a*, *j*) *gdiv D*) +
1) *a* (*j* + *m*) *A*
unfolding *reduce-element-mod-D-abs-def* **using** *True* **by** *auto*
also have ... = *?P* ∗ *A* **by** (*rule addrow-mat*[*OF A*], *insert j mn*, *auto*)

finally have *reduce-element-mod-D-abs A a j D m = ?P * A* .
thus *?thesis2* **using** *P inv-P* **by** *blast*
next
case *False* **note** *nc1 = False*
let *?P = addrow-mat (m+n) (− (A $$ (a, j) gdiv D)) a (j + m)*
have *A*: *A ∈ carrier-mat (m + n) n* **using** *A-def A′ mn* **by** *auto*
have *P*: *?P ∈ carrier-mat (m+n) (m+n)* **by** *simp*
have *inv-P*: *invertible-mat ?P*
by (*metis addrow-mat-carrier a det-addrow-mat dvd-mult-right*
invertible-iff-is-unit-JNF mult.right-neutral not-add-less2 semiring-gcd-class.gcd-dvd1)
show *?thesis1*
proof (*cases j = 0*)
case *True*
have *reduce-element-mod-D A a j D m = A*
unfolding *reduce-element-mod-D-def* **using** *True nc1* **by** *auto*
thus *?thesis1*
by (*metis A-def A′ carrier-append-rows invertible-mat-one*
left-mult-one-mat one-carrier-mat smult-carrier-mat)
next
case *False*
have *reduce-element-mod-D A a j D m = addrow (− (A $$ (a, j) gdiv D)) a (j + m) A*
unfolding *reduce-element-mod-D-def* **using** *False* **by** *auto*
also have *... = ?P * A* **by** (*rule addrow-mat*[*OF A*], *insert j mn*, *auto*)
finally have *reduce-element-mod-D A a j D m = ?P * A* .
thus *?thesis* **using** *P inv-P* **by** *blast*
qed
have *reduce-element-mod-D-abs A a j D m = addrow (− (A $$ (a, j) gdiv D)) a (j + m) A*
unfolding *reduce-element-mod-D-abs-def* **using** *False* **by** *auto*
also have *... = ?P * A* **by** (*rule addrow-mat*[*OF A*], *insert j mn*, *auto*)
finally have *reduce-element-mod-D-abs A a j D m = ?P * A* .
thus *?thesis2* **using** *P inv-P* **by** *blast*
qed

lemma *reduce-element-mod-D-append*:
assumes *A-def*: *A = A′ @$_r$ (D ·$_m$ (1$_m$ n))*
and *A′*: *A′ ∈ carrier-mat m n* **and** *a*: *a<m* **and** *j*: *j<n* **and** *mn*: *m≥n*
shows *reduce-element-mod-D A a j D m*
= mat-of-rows n [Matrix.row (reduce-element-mod-D A a j D m) i. i ← [0..<m]] @$_r$ (D ·$_m$ (1$_m$ n)) (**is** *?lhs = ?A′ @$_r$?D*)
and *reduce-element-mod-D-abs A a j D m*
= mat-of-rows n [Matrix.row (reduce-element-mod-D-abs A a j D m) i. i ← [0..<m]] @$_r$ (D ·$_m$ (1$_m$ n)) (**is** *?lhs-abs = ?A′-abs @$_r$?D*)
unfolding *atomize-conj*
proof (*rule conjI*; *rule eq-matI*)
let *?xs = (map (Matrix.row (reduce-element-mod-D A a j D m)) [0..<m])*
let *?xs-abs = (map (Matrix.row (reduce-element-mod-D-abs A a j D m)) [0..<m])*

have *lhs-carrier*: *?lhs* ∈ *carrier-mat* (*m*+*n*) *n*
and *lhs-carrier-abs*: *?lhs-abs* ∈ *carrier-mat* (*m*+*n*) *n*
by (*metis* (*no-types*, *lifting*) *add.comm-neutral append-rows-def A-def A′ carrier-matD*
carrier-mat-triv index-mat-four-block(*2*,*3*) *index-one-mat*(*2*) *index-smult-mat*(*2*) *index-zero-mat*(*2*,*3*)
reduce-element-mod-D-preserves-dimensions)+
have *map-A-carrier*[*simp*]: *?A′* ∈ *carrier-mat m n*
and *map-A-carrier-abs*[*simp*]: *?A′-abs* ∈ *carrier-mat m n*
by (*simp add*: *mat-of-rows-def*)+
have *AD-carrier*[*simp*]: *?A′* @$_r$ *?D* ∈ *carrier-mat* (*m*+*n*) *n*
and *AD-carrier-abs*[*simp*]: *?A′-abs* @$_r$ *?D* ∈ *carrier-mat* (*m*+*n*) *n*
by (*rule carrier-append-rows*, *insert lhs-carrier mn*, *auto*)
show *dim-row* (*?lhs*) = *dim-row* (*?A′* @$_r$ *?D*) **and** *dim-col* (*?lhs*) = *dim-col* (*?A′* @$_r$ *?D*)
dim-row (*?lhs-abs*) = *dim-row* (*?A′-abs* @$_r$ *?D*) **and** *dim-col* (*?lhs-abs*) = *dim-col* (*?A′-abs* @$_r$ *?D*)
using *lhs-carrier lhs-carrier-abs AD-carrier AD-carrier-abs* **unfolding** *carrier-mat-def* **by** *simp*+
show *?lhs* $$ (*i*, *ja*) = (*?A′* @$_r$ *?D*) $$ (*i*, *ja*) **if** *i*: *i* < *dim-row* (*?A′* @$_r$ *?D*) **and** *ja*: *ja* < *dim-col* (*?A′* @$_r$ *?D*) **for** *i ja*
proof (*cases i*<*m*)
case *True*
have *ja-n*: *ja* < *n*
by (*metis Nat.add-0-right append-rows-def index-mat-four-block*(*3*) *index-zero-mat*(*3*) *ja mat-of-rows-carrier*(*3*))
have (*?A′* @$_r$ *?D*) $$ (*i*, *ja*) = *?A′* $$ (*i*,*ja*)
by (*metis* (*no-types*, *lifting*) *Nat.add-0-right True append-rows-def diff-zero i index-mat-four-block index-zero-mat*(*3*) *ja length-map length-upt mat-of-rows-carrier*(*2*))
also have ... = *?xs* ! *i* $*v ja*
by (*rule mat-of-rows-index*, *insert i True ja* , *auto simp add*: *append-rows-def*)
also have ... = *?lhs* $$ (*i*,*ja*)
by (*rule map-first-rows-index*, *insert assms lhs-carrier True i ja-n*, *auto*)
finally show *?thesis* **..**
next
case *False*
have *ja-n*: *ja* < *n*
by (*metis Nat.add-0-right append-rows-def index-mat-four-block*(*3*) *index-zero-mat*(*3*) *ja mat-of-rows-carrier*(*3*))
have (*?A′* @$_r$ *?D*) $$ (*i*, *ja*) =*?D* $$ (*i*−*m*,*ja*)
by (*smt* (*verit*) *False Nat.add-0-right map-A-carrier append-rows-def carrier-matD i*
index-mat-four-block index-zero-mat(*3*) *ja-n*)
also have ... = *?lhs* $$ (*i*,*ja*)
by (*metis* (*no-types*, *lifting*) *False Nat.add-0-right map-A-carrier append-rows-def A-def A′ a*
carrier-matD i index-mat-addrow(*1*) *index-mat-four-block*(*1*,*2*) *index-zero-mat*(*3*) *ja-n*
lhs-carrier reduce-element-mod-D-def reduce-element-mod-D-preserves-dimensions)

```
    finally show ?thesis ..
  qed
  fix i ja assume i: i < dim-row (?A'-abs @r ?D) and ja: ja < dim-col (?A'-abs
@r ?D)
  have ja-n: ja < n
   by (metis Nat.add-0-right append-rows-def index-mat-four-block(3) index-zero-mat(3)
ja mat-of-rows-carrier(3))
  show ?lhs-abs $$ (i, ja) = (?A'-abs @r ?D) $$ (i, ja)
  proof (cases i<m)
    case True
    have (?A'-abs @r ?D) $$ (i, ja) = ?A'-abs $$ (i,ja)
      by (metis (no-types, lifting) Nat.add-0-right True append-rows-def diff-zero i
       index-mat-four-block index-zero-mat(3) ja length-map length-upt mat-of-rows-carrier(2))
    also have ... = ?xs-abs ! i $v ja
     by (rule mat-of-rows-index, insert i True ja , auto simp add: append-rows-def)
    also have ... = ?lhs-abs $$ (i,ja)
      by (rule map-first-rows-index, insert assms lhs-carrier-abs True i ja-n, auto)
    finally show ?thesis ..
  next
    case False
    have (?A'-abs @r ?D) $$ (i, ja) = ?D $$ (i−m,ja)
        by (smt (verit) False Nat.add-0-right map-A-carrier-abs append-rows-def
carrier-matD i
          index-mat-four-block index-zero-mat(3) ja-n)
    also have ... = ?lhs-abs $$ (i,ja)
        by (metis (no-types, lifting) False Nat.add-0-right map-A-carrier-abs ap-
pend-rows-def A-def A' a
       carrier-matD i index-mat-addrow(1) index-mat-four-block(1,2) index-zero-mat(3)
ja-n
       lhs-carrier-abs reduce-element-mod-D-abs-def reduce-element-mod-D-preserves-dimensions)
    finally show ?thesis ..
  qed
qed


lemma reduce-append-rows-eq:
  assumes A': A' ∈ carrier-mat m n
    and A-def: A = A' @r (D ·m (1m n)) and a: a<m and xm: x<m and 0<n
  and Aaj: A $$ (a,0) ≠ 0
  shows reduce a x D A
  = mat-of-rows n [Matrix.row ((reduce a x D A)) i. i ← [0..<m]] @r D ·m 1m n
(is ?thesis1)
  and reduce-abs a x D A
  = mat-of-rows n [Matrix.row ((reduce-abs a x D A)) i. i ← [0..<m]] @r D ·m
1m n (is ?thesis2)
  unfolding atomize-conj
proof (rule conjI; rule matrix-append-rows-eq-if-preserves)
  let ?reduce-ax = reduce a x D A
  let ?reduce-abs = reduce-abs a x D A
```

obtain *p q u v d* **where** *pquvd*: $(p,q,u,v,d) = euclid\text{-}ext2\ (A\$\$(a,0))\ (A\$\$(x,0))$
by (*metis prod-cases5*)
have *A*: *A*: *carrier-mat* $(m+n)\ n$ **by** (*simp add*: *A-def A'*)
show *D1*: $D \cdot_m 1_m\ n \in$ *carrier-mat n n* **and** $D \cdot_m 1_m\ n \in$ *carrier-mat n n* **by** *simp+*
show *?reduce-ax* $\in$ *carrier-mat* $(m + n)\ n$ *?reduce-abs* $\in$ *carrier-mat* $(m + n)\ n$
by (*metis Nat.add-0-right append-rows-def A' A-def carrier-matD carrier-mat-triv index-mat-four-block*(*2*,*3*) *index-one-mat*(*2*) *index-smult-mat*(*2*) *index-zero-mat*(*2*) *index-zero-mat*(*3*) *reduce-preserves-dimensions*)+
show $\forall\, i{\in}\{m..{<}m + n\}.\ \forall\, ja{<}n.$ *?reduce-ax* \$\$ $(i,\ ja) = (D \cdot_m 1_m\ n)$ \$\$ $(i - m,\ ja)$
and $\forall\, i{\in}\{m..{<}m + n\}.\ \forall\, ja{<}n.$ *?reduce-abs* \$\$ $(i,\ ja) = (D \cdot_m 1_m\ n)$ \$\$ $(i - m,\ ja)$
unfolding *atomize-conj*
proof (*rule conjI*; *rule+*)
fix *i ja* **assume** *i*: $i \in \{m..{<}m + n\}$ **and** *ja*: $ja < n$
have *ja-dc*: $ja <$ *dim-col A* **and** *i-dr*: $i <$ *dim-row A* **using** *i ja A* **by** *auto*
have *i-not-a*: $i \neq a$ **using** *i a* **by** *auto*
have *i-not-x*: $i \neq x$ **using** *i xm* **by** *auto*
have *?reduce-ax* \$\$ $(i,ja) = A$ \$\$ (i,ja)
unfolding *reduce-alt-def-not0*[*OF Aaj pquvd*] **using** *ja-dc i-dr i-not-a i-not-x* **by** *auto*
also have $\ldots = (if\ i < dim\text{-}row\ A'\ then\ A'\ \$\$(i,ja)\ else\ (D \cdot_m (1_m\ n))\$\$(i{-}m,ja))$
by (*unfold A-def*, *rule append-rows-nth*[*OF A' D1 - ja*], *insert A i-dr*, *simp*)
also have $\ldots = (D \cdot_m 1_m\ n)$ \$\$ $(i - m,\ ja)$ **using** *i A'* **by** *auto*
finally show *?reduce-ax* \$\$ $(i,ja) = (D \cdot_m 1_m\ n)$ \$\$ $(i - m,\ ja)$ **.**
have *?reduce-abs* \$\$ $(i,ja) = A$ \$\$ (i,ja)
unfolding *reduce-alt-def-not0*[*OF Aaj pquvd*] **using** *ja-dc i-dr i-not-a i-not-x* **by** *auto*
also have $\ldots = (if\ i < dim\text{-}row\ A'\ then\ A'\ \$\$(i,ja)\ else\ (D \cdot_m (1_m\ n))\$\$(i{-}m,ja))$
by (*unfold A-def*, *rule append-rows-nth*[*OF A' D1 - ja*], *insert A i-dr*, *simp*)
also have $\ldots = (D \cdot_m 1_m\ n)$ \$\$ $(i - m,\ ja)$ **using** *i A'* **by** *auto*
finally show *?reduce-abs* \$\$ $(i,ja) = (D \cdot_m 1_m\ n)$ \$\$ $(i - m,\ ja)$ **.**
qed
qed

fun *reduce-row-mod-D*
where *reduce-row-mod-D A a* [] *D m* $= A$ |
reduce-row-mod-D A a $(x \#\ xs)$ *D m* $=$ *reduce-row-mod-D* (*reduce-element-mod-D A a x D m*) *a xs D m*

fun *reduce-row-mod-D-abs*
where *reduce-row-mod-D-abs A a* [] *D m* $= A$ |
reduce-row-mod-D-abs A a $(x \#\ xs)$ *D m* $=$ *reduce-row-mod-D-abs* (*reduce-element-mod-D-abs A a x D m*) *a xs D m*

lemma *reduce-row-mod-D-preserves-dimensions*:
shows [*simp*]: *dim-row* (*reduce-row-mod-D A a xs D m*) = *dim-row A*
and [*simp*]: *dim-col* (*reduce-row-mod-D A a xs D m*) = *dim-col A*
by (*induct A a xs D m rule*: *reduce-row-mod-D.induct*, *auto*)

lemma *reduce-row-mod-D-preserves-dimensions-abs*:
shows [*simp*]: *dim-row* (*reduce-row-mod-D-abs A a xs D m*) = *dim-row A*
and [*simp*]: *dim-col* (*reduce-row-mod-D-abs A a xs D m*) = *dim-col A*
by (*induct A a xs D m rule*: *reduce-row-mod-D-abs.induct*, *auto*)

lemma *reduce-row-mod-D-invertible-mat*:
assumes *A-def*: $A = A' \mathbin{@_r} (D \cdot_m (1_m\ n))$
and *A'*: $A' \in$ *carrier-mat m n* **and** *a*: $a<m$ **and** *j*: $\forall j \in$ *set xs*. $j<n$ **and** *mn*: $m \geq n$
shows $\exists P.\ P \in$ *carrier-mat* $(m+n)\ (m+n) \wedge$ *invertible-mat P* $\wedge$
reduce-row-mod-D A a xs D m = $P * A$
using *assms*
proof (*induct A a xs D m arbitrary*: *A' rule*: *reduce-row-mod-D.induct*)
case (*1 A a D m*)
show *?case* **by** (*rule exI*[*of* - $1_m\ (m+n)$], *insert 1.prems*, *auto simp add*: *append-rows-def*)
next
case (*2 A a x xs D m*)
let *?reduce-xs* = (*reduce-element-mod-D A a x D m*)
have *1*: *reduce-row-mod-D A a* (*x # xs*) *D m*
= *reduce-row-mod-D ?reduce-xs a xs D m* **by** *simp*
have $\exists P.\ P \in$ *carrier-mat* $(m+n)\ (m+n) \wedge$ *invertible-mat P* $\wedge$
reduce-element-mod-D A a x D m = $P * A$
by (*rule reduce-element-mod-D-invertible-mat*, *insert 2.prems*, *auto*)
from *this* **obtain** *P* **where** *P*: $P \in$ *carrier-mat* $(m+n)\ (m+n)$ **and** *inv-P*: *invertible-mat P*
and *R-P*: *reduce-element-mod-D A a x D m* = $P * A$ **by** *auto*
have $\exists P.\ P \in$ *carrier-mat* $(m + n)\ (m + n) \wedge$ *invertible-mat P* $\wedge$ *reduce-row-mod-D ?reduce-xs a xs D m* = $P *$ *?reduce-xs*
proof (*rule 2.hyps*)
let *?A'* = *mat-of-rows n* [*Matrix.row* (*reduce-element-mod-D A a x D m*) *i*. *i* $\leftarrow$ [*0*..<*m*]]
show *reduce-element-mod-D A a x D m* = *?A'* $\mathbin{@_r} (D \cdot_m (1_m\ n))$
by (*rule reduce-element-mod-D-append*, *insert 2.prems*, *auto*)
qed (*insert 2.prems*, *auto*)
from *this* **obtain** *P2* **where** *P2*: $P2 \in$ *carrier-mat* $(m + n)\ (m + n)$ **and** *inv-P2*: *invertible-mat P2*
and *R-P2*: *reduce-row-mod-D ?reduce-xs a xs D m* = *P2* $*$ *?reduce-xs*
by *auto*
have *invertible-mat* (*P2* $*$ *P*) **using** *P P2 inv-P inv-P2 invertible-mult-JNF* **by** *blast*
moreover have (*P2* $*$ *P*) $\in$ *carrier-mat* $(m+n)\ (m+n)$ **using** *P2 P* **by** *auto*
moreover have *reduce-row-mod-D A a* (*x # xs*) *D m* = (*P2* $*$ *P*) $*$ *A*
by (*smt* (*verit*) *P P2 R-P R-P2 1 assoc-mult-mat carrier-matD carrier-mat-triv*

index-mult-mat reduce-row-mod-D-preserves-dimensions)
ultimately show *?case* **by** *blast*
qed

lemma *reduce-row-mod-D-abs-invertible-mat*:
assumes *A-def*: $A = A' \mathbin{@_r} (D \cdot_m (1_m\ n))$
and *A'*: $A' \in$ *carrier-mat m n* **and** *a*: $a<m$ **and** *j*: $\forall j \in$ *set xs*. $j<n$ **and** *mn*: $m \geq n$
shows $\exists P.\ P \in$ *carrier-mat* $(m+n)\ (m+n) \wedge$ *invertible-mat P* $\wedge$
reduce-row-mod-D-abs A a xs D m $= P * A$
using *assms*
proof (*induct A a xs D m arbitrary*: *A' rule*: *reduce-row-mod-D-abs.induct*)
case (*1 A a D m*)
show *?case* **by** (*rule exI*[*of* - $1_m\ (m+n)$], *insert 1.prems*, *auto simp add*: *append-rows-def*)
next
case (*2 A a x xs D m*)
let *?reduce-xs* = (*reduce-element-mod-D-abs A a x D m*)
have *1*: *reduce-row-mod-D-abs A a* $(x \# xs)$ *D m*
$=$ *reduce-row-mod-D-abs ?reduce-xs a xs D m* **by** *simp*
have $\exists P.\ P \in$ *carrier-mat* $(m+n)\ (m+n) \wedge$ *invertible-mat P* $\wedge$
reduce-element-mod-D-abs A a x D m $= P * A$
by (*rule reduce-element-mod-D-invertible-mat*, *insert 2.prems*, *auto*)
from *this* **obtain** *P* **where** *P*: $P \in$ *carrier-mat* $(m+n)\ (m+n)$ **and** *inv-P*: *invertible-mat P*
and *R-P*: *reduce-element-mod-D-abs A a x D m* $= P * A$ **by** *auto*
have $\exists P.\ P \in$ *carrier-mat* $(m+n)\ (m+n) \wedge$ *invertible-mat P* $\wedge$ *reduce-row-mod-D-abs ?reduce-xs a xs D m* $= P *$ *?reduce-xs*
proof (*rule 2.hyps*)
let *?A'* = *mat-of-rows n* [*Matrix.row* (*reduce-element-mod-D-abs A a x D m*) *i*. $i \leftarrow [0..<m]$]
show *reduce-element-mod-D-abs A a x D m* $=$ *?A'* $\mathbin{@_r} (D \cdot_m (1_m\ n))$
by (*rule reduce-element-mod-D-append*, *insert 2.prems*, *auto*)
qed (*insert 2.prems*, *auto*)
from *this* **obtain** *P2* **where** *P2*: $P2 \in$ *carrier-mat* $(m+n)\ (m+n)$ **and** *inv-P2*: *invertible-mat P2*
and *R-P2*: *reduce-row-mod-D-abs ?reduce-xs a xs D m* $= P2 *$ *?reduce-xs*
by *auto*
have *invertible-mat* $(P2 * P)$ **using** *P P2 inv-P inv-P2 invertible-mult-JNF* **by** *blast*
moreover have $(P2 * P) \in$ *carrier-mat* $(m+n)\ (m+n)$ **using** *P2 P* **by** *auto*
moreover have *reduce-row-mod-D-abs A a* $(x \# xs)$ *D m* $= (P2 * P) * A$
by (*smt* (*verit*) *P P2 R-P R-P2 1 assoc-mult-mat carrier-matD carrier-mat-triv index-mult-mat reduce-row-mod-D-preserves-dimensions-abs*)
ultimately show *?case* **by** *blast*
qed
end

context *proper-mod-operation*
begin
lemma *dvd-gdiv-mult-left*[*simp*]: **assumes** $b > 0$ b *dvd* a **shows** $b * (a$ *gdiv* $b) = a$
using *dvd-gdiv-mult-right*[*OF assms*] **by** (*auto simp*: *ac-simps*)

lemma *reduce-element-mod-D*:
assumes *A-def*: $A = A' \,@_r\ (D \cdot_m (1_m\ n))$
and A': $A' \in$ *carrier-mat m n* **and** a: $a \leq m$ **and** j: $j < n$ **and** mn: $m \geq n$
and D: $D > 0$
shows *reduce-element-mod-D A a j D m* = *Matrix.mat* (*dim-row A*) (*dim-col A*)
($\lambda(i,k)$. *if* $i = a \wedge k = j$ *then if* $j = 0$ *then if D dvd A*$$(i,k)$
then D else A$$(i,k) *else A*$$(i,k) *gmod D else A*$$(i,k)) (**is** - = *?A*)
and *reduce-element-mod-D-abs A a j D m* = *Matrix.mat* (*dim-row A*) (*dim-col A*)
($\lambda(i,k)$. *if* $i = a \wedge k = j$ *then if* $j = 0 \wedge D$ *dvd A*$$(i,k) *then D else A*$$(i,k) *gmod D else A*$$(i,k)) (**is** - = *?A-abs*)
unfolding *atomize-conj*
proof (*rule conjI*; *rule eq-matI*)
have *A*: $A \in$ *carrier-mat* $(m+n)$ n **using** *A-def A'* **by** *simp*
have *dr*: *dim-row ?A* = *dim-row ?A-abs* **and** *dc*: *dim-col ?A* = *dim-col ?A-abs* **by** *auto*
have *1*: *reduce-element-mod-D A a j D m* $$ (*i*, *ja*) = *?A* $$ (*i*, *ja*) (**is** *?thesis1*)
and *2*: *reduce-element-mod-D-abs A a j D m* $$ (*i*, *ja*) = *?A-abs* $$ (*i*, *ja*) (**is** *?thesis2*)
if *i*: $i <$ *dim-row ?A* **and** *ja*: *ja* $<$ *dim-col ?A* **for** *i ja*
unfolding *atomize-conj*
proof (*rule conjI*; *cases i*=*a*)
case *False*
have *reduce-element-mod-D A a j D m* = (*if j* = *0 then if D dvd A*$$(*a*,*j*) *then addrow* $(-((A\$\$(a,j)\ gdiv\ D)) + 1)\ a\ (j + m)\ A$
else A
else addrow $(-((A\$\$(a,j)\ gdiv\ D)))\ a\ (j + m)\ A$)
unfolding *reduce-element-mod-D-def* **by** *simp*
also have ... $$ (*i*,*ja*) = *A* $$ (*i*, *ja*) **unfolding** *mat-addrow-def* **using** *False ja i* **by** *auto*
also have ... = *?A* $$ (*i*,*ja*) **using** *False* **using** *i ja* **by** *auto*
finally show *?thesis1* **.**
have *reduce-element-mod-D-abs A a j D m* $$ (*i*,*ja*) = *A* $$ (*i*, *ja*)
unfolding *reduce-element-mod-D-abs-def mat-addrow-def* **using** *False ja i* **by** *auto*
also have ... = *?A-abs* $$ (*i*,*ja*) **using** *False* **using** *i ja* **by** *auto*
finally show *?thesis2* **.**
next
case *True* **note** *ia* = *True*
have *reduce-element-mod-D A a j D m*
= (*if j* = *0 then if D dvd A*$$(*a*,*j*) *then addrow* $(-((A\$\$(a,j)\ gdiv\ D)) + 1)$ *a* $(j + m)$ *A else A*

```
      else addrow (−((A$$(a,j) gdiv D))) a (j + m) A)
    unfolding reduce-element-mod-D-def by simp
  also have ... $$ (i,ja) = ?A $$ (i,ja)
  proof (cases ja = j)
    case True note ja-j = True
    have A $$ (j + m, ja) = (D ·m (1m n)) $$ (j,ja)
      by (rule append-rows-nth2[OF A' - A-def ], insert j ja A mn, auto)
    also have ... = D ∗ (1m n) $$ (j,ja) by (rule index-smult-mat, insert ja j A
mn, auto)
    also have ... = D by (simp add: True j mn)
    finally have A-ja-jaD: A $$ (j + m, ja) = D .
    show ?thesis
    proof (cases j=0 ∧ D dvd A$$(a,j))
      case True
      have 1: reduce-element-mod-D A a j D m = addrow (−((A$$(a,j) gdiv D))
+ 1) a (j + m) A
        using True ia ja-j unfolding reduce-element-mod-D-def by auto
      also have ... $$(i,ja) = (− (A $$ (a, j) gdiv D) + 1) ∗ A $$ (j + m, ja) +
A $$ (i, ja)
        unfolding mat-addrow-def using True ja-j ia
        using A i j by auto
      also have ... = D
      proof −
        have A $$ (i, ja) + D ∗ − (A $$ (i, ja) gdiv D) = 0
          using True ia ja-j D by force
        then show ?thesis
            by (metis A-ja-jaD ab-semigroup-add-class.add-ac(1) add.commute
add-right-imp-eq ia int-distrib(2)
            ja-j more-arith-simps(3) mult.commute mult-cancel-right1)
      qed
      also have ... = ?A $$ (i,ja) using True ia A i j ja-j by auto
      finally show ?thesis
        using True 1 by auto
    next
      case False
      show ?thesis
      proof (cases ja=0)
        case True
        then show ?thesis
          using False i ja ja-j by force
      next
        case False
      have ?A $$ (i,ja) = A $$ (i, ja) gmod D using True ia A i j False by auto
      also have ... = A $$ (i, ja) − ((A $$ (i, ja) gdiv D) ∗ D)
        by (subst gmod-gdiv[OF D], auto)
      also have ... = − (A $$ (a, j) gdiv D) ∗ A $$ (j + m, ja) + A $$ (i, ja)
        unfolding A-ja-jaD by (simp add: True ia)
      finally show ?thesis
        using A False True i ia j by auto
```

```
    qed
  qed
 next
    case False
    have A $$ (j + m, ja) = (D ·_m (1_m n)) $$ (j,ja)
      by (rule append-rows-nth2[OF A' - A-def ], insert j mn ja A, auto)
    also have ... = D * (1_m n) $$ (j,ja) by (rule index-smult-mat, insert ja j A
mn, auto)
    also have ... = 0 using False using A a mn ja j by force
    finally have A-am-ja0: A $$ (j + m, ja) = 0 .
    then show ?thesis using False i ja by fastforce
  qed
  finally show ?thesis1 .
  have reduce-element-mod-D-abs A a j D m
    = (if j = 0 ∧ D dvd A$$(a,j) then addrow (−((A$$(a,j) gdiv D)) + 1) a (j
+ m) A
      else addrow (−((A$$(a,j) gdiv D))) a (j + m) A)
    unfolding reduce-element-mod-D-abs-def by simp
  also have ... $$ (i,ja) = ?A-abs $$ (i,ja)
  proof (cases ja = j)
    case True note ja-j = True
    have A $$ (j + m, ja) = (D ·_m (1_m n)) $$ (j,ja)
      by (rule append-rows-nth2[OF A' - A-def ], insert j ja A mn, auto)
    also have ... = D * (1_m n) $$ (j,ja) by (rule index-smult-mat, insert ja j A
mn, auto)
    also have ... = D by (simp add: True j mn)
    finally have A-ja-jaD: A $$ (j + m, ja) = D .
    show ?thesis
    proof (cases j=0 ∧ D dvd A$$(a,j))
      case True
      have 1: reduce-element-mod-D-abs A a j D m = addrow (−((A$$(a,j) gdiv
D)) + 1) a (j + m) A
        using True ia ja-j unfolding reduce-element-mod-D-abs-def by auto
      also have ... $$(i,ja) = (− (A $$ (a, j) gdiv D) + 1) * A $$ (j + m, ja) +
A $$ (i, ja)
        unfolding mat-addrow-def using True ja-j ia
        using A i j by auto
      also have ... = D
      proof −
        have A $$ (i, ja) + D * − (A $$ (i, ja) gdiv D) = 0
          using True ia ja-j D by force
        then show ?thesis
            by (metis A-ja-jaD ab-semigroup-add-class.add-ac(1) add.commute
add-right-imp-eq ia int-distrib(2)
            ja-j more-arith-simps(3) mult.commute mult-cancel-right1)
      qed
      also have ... = ?A-abs $$ (i,ja) using True ia A i j ja-j by auto
      finally show ?thesis
        using True 1 by auto
```

```
      next
        case False
        have i: i<dim-row ?A-abs and ja: ja<dim-col ?A-abs using i ja by auto
        have ?A-abs $$ (i,ja) = A $$ (i, ja) gmod D using True ia A i j False by
auto
        also have ... = A $$ (i, ja) − ((A $$ (i, ja) gdiv D) ∗ D)
          by (subst gmod-gdiv[OF D], auto)
        also have ... =  − (A $$ (a, j) gdiv D) ∗ A $$ (j + m, ja) + A $$ (i, ja)
          unfolding A-ja-jaD by (simp add: True ia)
        finally show ?thesis
          using A False True i ia j by auto
      qed
  next
      case False
      have A $$ (j + m, ja) = (D ·m (1m n)) $$ (j,ja)
        by (rule append-rows-nth2[OF A' - A-def ], insert j mn ja A, auto)
      also have ... = D ∗ (1m n) $$ (j,ja) by (rule index-smult-mat, insert ja j A
mn, auto)
      also have ... = 0 using False using A a mn ja j by force
      finally have A-am-ja0: A $$ (j + m, ja) = 0 .
      then show ?thesis using False i ja by fastforce
    qed
    finally show ?thesis2 .
  qed
  from this
  show ⋀i ja. i<dim-row ?A ⟹ ja < dim-col ?A ⟹ reduce-element-mod-D A a
j D m $$ (i, ja) = ?A $$ (i, ja)
   and ⋀i ja. i<dim-row ?A-abs ⟹ ja < dim-col ?A-abs ⟹ reduce-element-mod-D-abs
A a j D m $$ (i, ja) = ?A-abs $$ (i, ja)
    using dr dc by auto
next
  show dim-row (reduce-element-mod-D A a j D m) = dim-row ?A
    and dim-col (reduce-element-mod-D A a j D m) = dim-col ?A
    dim-row (reduce-element-mod-D-abs A a j D m) = dim-row ?A-abs
    and dim-col (reduce-element-mod-D-abs A a j D m) = dim-col ?A-abs
    by auto
qed


lemma reduce-row-mod-D:
  assumes A-def: A = A' @r (D ·m (1m n))
    and A': A' ∈ carrier-mat m n and a: a<m and j: ∀j∈set xs. j<n
    and d: distinct xs and m≥n
    and D > 0
  shows reduce-row-mod-D A a xs D m = Matrix.mat (dim-row A) (dim-col A)
        (λ(i,k). if i = a ∧ k ∈ set xs then if k = 0 then if D dvd A$$(i,k)
         then D else A$$(i,k) else A$$(i,k) gmod D else A$$(i,k))
  using assms
proof (induct A a xs D m arbitrary: A' rule: reduce-row-mod-D.induct)
```

```
  case (1 A a D m)
  then show ?case by force
next
  case (2 A a x xs D m)
  let ?reduce-xs = (reduce-element-mod-D A a x D m)
  have 1: reduce-row-mod-D A a (x # xs) D m
    = reduce-row-mod-D ?reduce-xs a xs D m by simp
  have 2: reduce-element-mod-D A a j D m = Matrix.mat (dim-row A) (dim-col
A)
        (λ(i,k). if i = a ∧ k = j then if j = 0 then if D dvd A$$(i,k)
        then D else A$$(i,k) else A$$(i,k) gmod D else A$$(i,k)) if j<n for j
    by (rule reduce-element-mod-D, insert 2.prems that, auto)
  have reduce-row-mod-D ?reduce-xs a xs D m =
    Matrix.mat (dim-row ?reduce-xs) (dim-col ?reduce-xs) (λ(i,k). if i = a ∧ k ∈
set xs then
    if k=0 then if D dvd ?reduce-xs $$ (i, k) then D else ?reduce-xs $$ (i, k)
    else ?reduce-xs $$ (i, k) gmod D else ?reduce-xs $$ (i, k))
  proof (rule 2.hyps)
    let ?A' = mat-of-rows n [Matrix.row (reduce-element-mod-D A a x D m) i. i
← [0..<m]]
    show reduce-element-mod-D A a x D m = ?A' @r (D ·m (1m n))
      by (rule reduce-element-mod-D-append, insert 2.prems, auto)
  qed (insert 2.prems, auto)
  also have ... = Matrix.mat (dim-row A) (dim-col A)
        (λ(i,k). if i = a ∧ k ∈ set (x # xs) then if k = 0 then if D dvd A$$(i,k)
        then D else A$$(i,k) else A$$(i,k) gmod D else A$$(i,k)) (is ?lhs = ?rhs)
  proof (rule eq-matI)
    show dim-row ?lhs = dim-row ?rhs and dim-col ?lhs = dim-col ?rhs by auto
    fix i j assume i: i<dim-row ?rhs and j: j < dim-col ?rhs
    have jn: j<n using j 2.prems by (simp add: append-rows-def)
    have xn: x < n by (simp add: 2.prems(4))
    show ?lhs $$ (i,j) = ?rhs $$ (i,j)
    proof (cases i=a ∧ j ∈ set xs)
      case True note ia-jxs = True
      have j-not-x: j≠x
        using 2.prems(5) True by auto
      show ?thesis
      proof (cases j=0 ∧ D dvd ?reduce-xs $$(i,j))
        case True
        have ?lhs $$ (i,j) = D
          using True i j ia-jxs by auto
        also have ... = ?rhs $$ (i,j) using i j j-not-x
         by (smt (verit) 2 calculation dim-col-mat(1) dim-row-mat(1) index-mat(1)
insert-iff list.set(2) prod.simps(2) xn)
        finally show ?thesis .
      next
        case False note nc1 = False
        show ?thesis
        proof (cases j=0)
```

```
        case True
        then show ?thesis
         by (smt (verit) 2 False case-prod-conv dim-col-mat(1) dim-row-mat(1) i
index-mat(1) j j-not-x xn)
       next
        case False
     have ?lhs $$ (i,j) = ?reduce-xs $$ (i, j) gmod D
      using True False i j by auto
     also have ... = A $$ (i,j) gmod D using 2[OF xn] j-not-x i j by auto
     also have ... = ?rhs $$ (i,j) using i j j-not-x ‹D > 0›
     using False True dim-col-mat(1) dim-row-mat(1) index-mat(1) list.set-intros(2)
old.prod.case
       by auto
     finally show ?thesis .
   qed
  qed
  next
     case False
     show ?thesis using 2 i j xn
      by (smt (verit) False dim-col-mat(1) dim-row-mat(1) index-mat(1) insert-iff
list.set(2) prod.simps(2))
   qed
  qed
  finally show ?case using 1 by simp
qed



lemma reduce-row-mod-D-abs:
  assumes A-def: A = A' @_r (D ·_m (1_m n))
    and A': A' ∈ carrier-mat m n and a: a<m and j: ∀j∈set xs. j<n
    and d: distinct xs and m≥n
    and D > 0
  shows reduce-row-mod-D-abs A a xs D m = Matrix.mat (dim-row A) (dim-col
A)
            (λ(i,k). if i = a ∧ k ∈ set xs then if k = 0 ∧ D dvd A$$(i,k)
             then D else A$$(i,k) gmod D else A$$(i,k))
  using assms
proof (induct A a xs D m arbitrary: A' rule: reduce-row-mod-D-abs.induct)
  case (1 A a D m)
  then show ?case by force
next
  case (2 A a x xs D m)
  let ?reduce-xs = (reduce-element-mod-D-abs A a x D m)
  have 1: reduce-row-mod-D-abs A a (x # xs) D m
    = reduce-row-mod-D-abs ?reduce-xs a xs D m by simp
 have 2: reduce-element-mod-D-abs A a j D m = Matrix.mat (dim-row A) (dim-col
A)
```

```
  (λ(i,k). if i = a ∧ k = j then if j = 0 ∧ D dvd A$$(i,k) then D
    else A$$(i,k) gmod D else A$$(i,k)) if j<n for j
    by (rule reduce-element-mod-D, insert 2.prems that, auto)
  have reduce-row-mod-D-abs ?reduce-xs a xs D m =
     Matrix.mat (dim-row ?reduce-xs) (dim-col ?reduce-xs) (λ(i,k). if i = a ∧ k ∈
set xs then
    if k=0 ∧ D dvd ?reduce-xs $$ (i, k) then D
    else ?reduce-xs $$ (i, k) gmod D else ?reduce-xs $$ (i, k))
  proof (rule 2.hyps)
    let ?A' = mat-of-rows n [Matrix.row (reduce-element-mod-D-abs A a x D m)
i. i ← [0..<m]]
    show reduce-element-mod-D-abs A a x D m = ?A' @r (D ·m (1m n))
      by (rule reduce-element-mod-D-append, insert 2.prems, auto)
  qed (insert 2.prems, auto)
  also have ... = Matrix.mat (dim-row A) (dim-col A)
          (λ(i,k). if i = a ∧ k ∈ set (x # xs) then if k = 0 ∧ D dvd A$$(i,k)
          then D else A$$(i,k) gmod D else A$$(i,k)) (is ?lhs = ?rhs)
  proof (rule eq-matI)
    show dim-row ?lhs = dim-row ?rhs and dim-col ?lhs = dim-col ?rhs by auto
    fix i j assume i: i<dim-row ?rhs and j: j < dim-col ?rhs
    have jn: j<n using j 2.prems by (simp add: append-rows-def)
    have xn: x < n by (simp add: 2.prems(4))
    show ?lhs $$ (i,j) = ?rhs $$ (i,j)
    proof (cases i=a ∧ j ∈ set xs)
      case True note ia-jxs = True
      have j-not-x: j≠x
        using 2.prems(5) True by auto
      show ?thesis
      proof (cases j=0 ∧ D dvd ?reduce-xs $$(i,j))
        case True
        have ?lhs $$ (i,j) = D
          using True i j ia-jxs by auto
        also have ... = ?rhs $$ (i,j) using i j j-not-x
         by (smt (verit) 2 calculation dim-col-mat(1) dim-row-mat(1) index-mat(1)
insert-iff list.set(2) prod.simps(2) xn)
        finally show ?thesis .
      next
        case False
      have ?lhs $$ (i,j) = ?reduce-xs $$ (i, j) gmod D
        using True False i j by auto
      also have ... = A $$ (i,j) gmod D using 2[OF xn] j-not-x i j by auto
      also have ... = ?rhs $$ (i,j) using i j j-not-x ‹D > 0›
      using 2 False True dim-col-mat(1) dim-row-mat(1) index-mat(1) list.set-intros(2)

          old.prod.case xn by auto
      finally show ?thesis .
    qed
  next
      case False
```

```
    show ?thesis using 2 i j xn
     by (smt (verit) False dim-col-mat(1) dim-row-mat(1) index-mat(1) insert-iff
list.set(2) prod.simps(2))
   qed
  qed
  finally show ?case using 1 by simp
qed
end
```

Now, we prove some transfer rules to connect Bézout matrices in HOL Analysis and JNF

```
lemma HMA-bezout-matrix[transfer-rule]:
  shows ((Mod-Type-Connect.HMA-M :: - ⇒ 'a :: {bezout-ring} ^ 'n :: mod-type
^ 'm :: mod-type ⇒ -)
 ===> (Mod-Type-Connect.HMA-I :: - ⇒ 'm ⇒ -) ===> (Mod-Type-Connect.HMA-I
:: - ⇒ 'm ⇒ -)
 ===> (Mod-Type-Connect.HMA-I :: - ⇒ 'n ⇒ -) ===> (=) ===> (Mod-Type-Connect.HMA-M))

  (bezout-matrix-JNF) (bezout-matrix)
proof (intro rel-funI, goal-cases)
  case (1 A A' a a' b b' j j' bezout bezout')
  note HMA-AA'[transfer-rule] = 1(1)
  note HMI-aa'[transfer-rule] = 1(2)
  note HMI-bb'[transfer-rule] = 1(3)
  note HMI-jj'[transfer-rule] = 1(4)
  note eq-bezout'[transfer-rule] = 1(5)
 show ?case unfolding Mod-Type-Connect.HMA-M-def Mod-Type-Connect.from-hma_m-def

  proof (rule eq-matI)
    let ?A = Matrix.mat CARD('m) CARD('m) (λ(i, j). bezout-matrix A' a' b' j'
bezout'
        $h mod-type-class.from-nat i $h mod-type-class.from-nat j)
    show dim-row (bezout-matrix-JNF A a b j bezout) = dim-row ?A
      and dim-col (bezout-matrix-JNF A a b j bezout) = dim-col ?A
      using Mod-Type-Connect.dim-row-transfer-rule[OF HMA-AA'] 
      unfolding bezout-matrix-JNF-def by auto
    fix i ja assume i: i < dim-row ?A and ja: ja < dim-col ?A
    let ?i = mod-type-class.from-nat i :: 'm
    let ?ja = mod-type-class.from-nat ja :: 'm
    have i-A: i < dim-row A
      using HMA-AA' Mod-Type-Connect.dim-row-transfer-rule i by fastforce
    have ja-A: ja < dim-row A
     using Mod-Type-Connect.dim-row-transfer-rule[OF HMA-AA'] ja by fastforce
    have HMA-I-ii'[transfer-rule]: Mod-Type-Connect.HMA-I i ?i
      unfolding Mod-Type-Connect.HMA-I-def using from-nat-not-eq i by auto
    have HMA-I-ja'[transfer-rule]: Mod-Type-Connect.HMA-I ja ?ja
      unfolding Mod-Type-Connect.HMA-I-def using from-nat-not-eq ja by auto
   have Aaj: A' $h a' $h j' = A $$ (a,j) unfolding index-hma-def[symmetric] by
(transfer, simp)
```

```
    have Abj: A' $h b' $h j' = A $$ (b, j) unfolding index-hma-def[symmetric]
by (transfer, simp)
    have ?A $$ (i, ja) = bezout-matrix A' a' b' j' bezout' $h ?i $h ?ja using i ja
by auto
    also have ... = (let (p, q, u, v, d) = bezout' (A' $h a' $h j') (A' $h b' $h j')
          in if ?i = a' ∧ ?ja = a' then p else if ?i = a' ∧ ?ja = b' then q else if ?i
= b' ∧ ?ja = a'
           then u else if ?i = b' ∧ ?ja = b' then v else if ?i = ?ja then 1 else 0)
      unfolding bezout-matrix-def by auto
    also have ... =  (let
       (p, q, u, v, d) = bezout (A $$ (a, j)) (A $$ (b, j))
      in
        if i = a ∧ ja = a then p else
        if i = a ∧ ja = b then q else
        if i = b ∧ ja = a then u else
        if i = b ∧ ja = b then v else
        if i = ja then 1 else 0) unfolding eq-bezout' Aaj Abj by (transfer, simp)
    also have ... = bezout-matrix-JNF A a b j bezout $$ (i,ja)
      unfolding bezout-matrix-JNF-def using i-A ja-A by auto
    finally show bezout-matrix-JNF A a b j bezout $$ (i, ja) = ?A $$ (i, ja) ..
  qed
qed



context
begin

private lemma invertible-bezout-matrix-JNF-mod-type:
  fixes A::'a::{bezout-ring-div} mat
  assumes A ∈ carrier-mat CARD('m::mod-type) CARD('n::mod-type)
  assumes ib: is-bezout-ext bezout
  and a-less-b: a < b and b: b<CARD('m) and j: j<CARD('n)
  and aj: A $$ (a, j) ≠ 0
shows invertible-mat (bezout-matrix-JNF A a b j bezout)
proof −
  define A' where A' = (Mod-Type-Connect.to-hma_m A :: 'a ^'n :: mod-type ^'m
:: mod-type)
  define a' where a' = (Mod-Type.from-nat a :: 'm)
  define b' where b' = (Mod-Type.from-nat b :: 'm)
  define j' where j' = (Mod-Type.from-nat j :: 'n)
  have AA'[transfer-rule]: Mod-Type-Connect.HMA-M A A'
    unfolding Mod-Type-Connect.HMA-M-def using assms A'-def by auto
  have aa'[transfer-rule]: Mod-Type-Connect.HMA-I a a'
    unfolding Mod-Type-Connect.HMA-I-def a'-def using assms
    using from-nat-not-eq order.strict-trans by blast
  have bb'[transfer-rule]: Mod-Type-Connect.HMA-I b b'
    unfolding Mod-Type-Connect.HMA-I-def b'-def using assms
    using from-nat-not-eq order.strict-trans by blast
```

```
  have jj'[transfer-rule]: Mod-Type-Connect.HMA-I j j'
    unfolding Mod-Type-Connect.HMA-I-def j'-def using assms
    using from-nat-not-eq order.strict-trans by blast
  have [transfer-rule]: bezout = bezout ..
   have [transfer-rule]: Mod-Type-Connect.HMA-M (bezout-matrix-JNF A a b j
bezout)
      (bezout-matrix A' a' b' j' bezout)
    by transfer-prover
  have invertible (bezout-matrix A' a' b' j' bezout)
  proof (rule invertible-bezout-matrix[OF ib])
    show a' < b' using a-less-b by (simp add: a'-def b b'-def from-nat-mono)
     show A' $h a' $h j' ≠ 0 unfolding index-hma-def[symmetric] using aj by
(transfer, simp)
  qed
  thus ?thesis by (transfer, simp)
qed

private lemma invertible-bezout-matrix-JNF-nontriv-mod-ring:
  fixes A::'a::{bezout-ring-div} mat
 assumes A ∈ carrier-mat CARD('m::nontriv mod-ring) CARD('n::nontriv mod-ring)
  assumes ib: is-bezout-ext bezout
  and a-less-b: a < b and b: b<CARD('m) and j: j<CARD('n)
  and aj: A $$ (a, j) ≠ 0
shows invertible-mat (bezout-matrix-JNF A a b j bezout)
 using assms invertible-bezout-matrix-JNF-mod-type by (smt (verit) CARD-mod-ring)




lemmas invertible-bezout-matrix-JNF-internalized =
  invertible-bezout-matrix-JNF-nontriv-mod-ring[unfolded CARD-mod-ring,
      internalize-sort 'm::nontriv, internalize-sort 'c::nontriv]

context
  fixes m::nat and n::nat
  assumes local-typedef1: ∃(Rep :: ('b ⇒ int)) Abs. type-definition Rep Abs {0..<m
:: int}
  assumes local-typedef2: ∃(Rep :: ('c ⇒ int)) Abs. type-definition Rep Abs {0..<n
:: int}
  and m: m>1
  and n: n>1
begin

lemma type-to-set1:
  shows class.nontriv TYPE('b) (is ?a) and m=CARD('b) (is ?b)
proof −
  from local-typedef1 obtain Rep::('b ⇒ int) and Abs
    where t: type-definition Rep Abs {0..<m :: int} by auto
   have card (UNIV :: 'b set) = card {0..<m} using t type-definition.card by
```

fastforce
also have ... = *m* **by** *auto*
finally show *?b* **..**
then show *?a* **unfolding** *class.nontriv-def* **using** *m* **by** *auto*
qed

lemma *type-to-set2*:
shows *class.nontriv TYPE('c)* (**is** *?a*) **and** *n=CARD('c)* (**is** *?b*)
proof −
from *local-typedef2* **obtain** *Rep*::('*c* ⇒ *int*) **and** *Abs*
where *t*: *type-definition Rep Abs* {*0*..<*n* :: *int*} **by** *blast*
have *card* (*UNIV* :: '*c set*) = *card* {*0*..<*n*} **using** *t type-definition.card* **by** *force*
also have ... = *n* **by** *auto*
finally show *?b* **..**
then show *?a* **unfolding** *class.nontriv-def* **using** *n* **by** *auto*
qed

lemma *invertible-bezout-matrix-JNF-nontriv-mod-ring-aux*:
fixes *A*::'*a*::{*bezout-ring-div*} *mat*
assumes *A* ∈ *carrier-mat m n*
assumes *ib*: *is-bezout-ext bezout*
and *a-less-b*: *a* < *b* **and** *b*: *b*<*m* **and** *j*: *j*<*n*
and *aj*: *A* $$ (*a*, *j*) ≠ *0*
shows *invertible-mat* (*bezout-matrix-JNF A a b j bezout*)
using *invertible-bezout-matrix-JNF-internalized*[*OF type-to-set2*(*1*) *type-to-set*(*1*),
where *?'aa* = '*b*]
using *assms*
using *type-to-set1*(*2*) *type-to-set2*(*2*) *local-typedef1 m* **by** *blast*
end

context
begin

private lemma *invertible-bezout-matrix-JNF-cancelled-first*:
∃ *Rep Abs. type-definition Rep Abs* {*0*..<*int n*} ⟹ {*0*..<*int m*} ≠ {} ⟹
1 < *m* ⟹ *1* < *n* ⟹
(*A*::'*a*::*bezout-ring-div mat*) ∈ *carrier-mat m n* ⟹ *is-bezout-ext bezout*
⟹ *a* < *b* ⟹ *b* < *m* ⟹ *j* < *n* ⟹ *A* $$ (*a*, *j*) ≠ *0* ⟹ *invertible-mat*
(*bezout-matrix-JNF A a b j bezout*)
using *invertible-bezout-matrix-JNF-nontriv-mod-ring-aux*[*cancel-type-definition*]
by *blast*

private lemma *invertible-bezout-matrix-JNF-cancelled-both*:
{*0*..<*int n*} ≠ {} ⟹ {*0*..<*int m*} ≠ {} ⟹ *1* < *m* ⟹ *1* < *n* ⟹

$1 < m \Longrightarrow 1 < n \Longrightarrow$
$(A::'a::bezout\text{-}ring\text{-}div\ mat) \in carrier\text{-}mat\ m\ n \Longrightarrow is\text{-}bezout\text{-}ext\ bezout$
$\Longrightarrow a < b \Longrightarrow b < m \Longrightarrow j < n \Longrightarrow A\ \$\$\ (a,\ j) \neq 0 \Longrightarrow invertible\text{-}mat$
$(bezout\text{-}matrix\text{-}JNF\ A\ a\ b\ j\ bezout)$
using *invertible-bezout-matrix-JNF-cancelled-first*[*cancel-type-definition*] **by** *blast*

lemma *invertible-bezout-matrix-JNF'*:
fixes $A::'a::\{bezout\text{-}ring\text{-}div\}\ mat$
assumes $A \in carrier\text{-}mat\ m\ n$
assumes *ib*: *is-bezout-ext bezout*
and *a-less-b*: $a < b$ **and** *b*: $b<m$ **and** *j*: $j<n$
and $n>1$
and *aj*: $A\ \$\$\ (a,\ j) \neq 0$
shows *invertible-mat* (*bezout-matrix-JNF A a b j bezout*)
using *invertible-bezout-matrix-JNF-cancelled-both assms* **by** *auto*

lemma *invertible-bezout-matrix-JNF-n1*:
fixes $A::'a::\{bezout\text{-}ring\text{-}div\}\ mat$
assumes *A*: $A \in carrier\text{-}mat\ m\ n$
assumes *ib*: *is-bezout-ext bezout*
and *a-less-b*: $a < b$ **and** *b*: $b<m$ **and** *j*: $j<n$
and *n1*: $n=1$
and *aj*: $A\ \$\$\ (a,\ j) \neq 0$
shows *invertible-mat* (*bezout-matrix-JNF A a b j bezout*)
proof −
let $?A = A\ @_c\ (0_m\ m\ n)$
have $(A\ @_c\ 0_m\ m\ n)\ \$\$\ (a,\ j) =\ (if\ j < dim\text{-}col\ A\ then\ A\ \$\$\ (a,\ j)\ else\ (0_m\ m\ n)\ \$\$\ (a,\ j - n))$
by (*rule append-cols-nth*[*OF A*], *insert assms*, *auto*)
also have ... = $A\ \$\$\ (a,j)$ **using** *assms* **by** *auto*
finally have *Aaj*: $(A\ @_c\ 0_m\ m\ n)\ \$\$\ (a,\ j) =\ A\ \$\$\ (a,j)$ **.**
have $(A\ @_c\ 0_m\ m\ n)\ \$\$\ (b,\ j) =\ (if\ j < dim\text{-}col\ A\ then\ A\ \$\$\ (b,\ j)\ else\ (0_m\ m\ n)\ \$\$\ (b,\ j - n))$
by (*rule append-cols-nth*[*OF A*], *insert assms*, *auto*)
also have ... = $A\ \$\$\ (b,j)$ **using** *assms* **by** *auto*
finally have *Abj*: $(A\ @_c\ 0_m\ m\ n)\ \$\$\ (b,\ j) = A\ \$\$\ (b,\ j)$ **.**
have *dr*: *dim-row A* = *dim-row ?A* **by** (*simp add*: *append-cols-def*)
have *dc*: *dim-col ?A* = *2*
by (*metis Suc-1 append-cols-def A n1 carrier-matD*(*2*) *index-mat-four-block*(*3*)
index-zero-mat(*3*) *plus-1-eq-Suc*)
have *bz-eq*: *bezout-matrix-JNF A a b j bezout* = *bezout-matrix-JNF ?A a b j bezout*
unfolding *bezout-matrix-JNF-def Aaj Abj dr* **by** *auto*
have *invertible-mat* (*bezout-matrix-JNF ?A a b j bezout*)
by (*rule invertible-bezout-matrix-JNF'*, *insert assms Aaj Abj dr dc*, *auto*)
thus *?thesis* **using** *bz-eq* **by** *simp*

qed

corollary *invertible-bezout-matrix-JNF*:
fixes *A*::'*a*::{*bezout-ring-div*} *mat*
assumes *A* ∈ *carrier-mat m n*
assumes *ib*: *is-bezout-ext bezout*
and *a-less-b*: *a* < *b* **and** *b*: *b*<*m* **and** *j*: *j*<*n*
and *aj*: *A* $$ (*a*, *j*) ≠ *0*
shows *invertible-mat* (*bezout-matrix-JNF A a b j bezout*)
using *invertible-bezout-matrix-JNF-n1 invertible-bezout-matrix-JNF' assms*
by (*metis One-nat-def gr-implies-not0 less-Suc0 not-less-iff-gr-or-eq*)

end
end

We continue with the soundness of the algorithm

lemma *bezout-matrix-JNF-mult-eq*:
assumes *A'*: *A'* ∈ *carrier-mat m n* **and** *a*: *a*≤*m* **and** *b*: *b*≤*m* **and** *ab*: *a* ≠ *b*
and *A-def*: *A* = *A'* @$_r$ *B* **and** *B*: *B* ∈ *carrier-mat n n*
assumes *pquvd*: (*p*,*q*,*u*,*v*,*d*) = *euclid-ext2* (*A*$$(*a*,*j*)) (*A*$$(*b*,*j*))
shows *Matrix.mat* (*dim-row A*) (*dim-col A*)
(λ(*i*,*k*). *if i* = *a then* (*p*∗*A*$$(*a*,*k*) + *q*∗*A*$$(*b*,*k*))
else if i = *b then u* ∗ *A*$$(*a*,*k*) + *v* ∗ *A*$$(*b*,*k*)
else A$$(*i*,*k*)
) = (*bezout-matrix-JNF A a b j euclid-ext2*) ∗ *A* (**is** *?A* = *?BM* ∗ *A*)
proof (*rule eq-matI*)
have *A*: *A* ∈ *carrier-mat* (*m*+*n*) *n* **using** *A-def A' B* **by** *simp*
hence *A-carrier*: *?A* ∈ *carrier-mat* (*m*+*n*) *n* **by** *auto*
show *dr*: *dim-row ?A* = *dim-row* (*?BM* ∗ *A*) **and** *dc*: *dim-col ?A* = *dim-col* (*?BM* ∗ *A*)
unfolding *bezout-matrix-JNF-def* **by** *auto*
fix *i ja* **assume** *i*: *i* < *dim-row* (*?BM* ∗ *A*) **and** *ja*: *ja* < *dim-col* (*?BM* ∗ *A*)
let *?f* = λ*ia*. (*bezout-matrix-JNF A a b j euclid-ext2*) $$ (*i*,*ia*) ∗ *A* $$ (*ia*,*ja*)
have *dv*: *dim-vec* (*col A ja*) = *m*+*n* **using** *A* **by** *auto*
have *i-dr*: *i*<*dim-row A* **using** *i A* **unfolding** *bezout-matrix-JNF-def* **by** *auto*
have *a-dr*: *a*<*dim-row A* **using** *A a ja* **by** *auto*
have *b-dr*: *b*<*dim-row A* **using** *A b ja* **by** *auto*
show *?A* $$ (*i*,*ja*) = (*?BM* ∗ *A*) $$ (*i*,*ja*)
proof −
have (*?BM* ∗ *A*) $$ (*i*,*ja*) = *Matrix.row ?BM i* · *col A ja*
by (*rule index-mult-mat*, *insert i ja*, *auto*)
also have ... = (∑ *ia* = *0*..<*dim-vec* (*col A ja*).
Matrix.row (*bezout-matrix-JNF A a b j euclid-ext2*) *i* $*v ia* ∗ *col A ja* $*v ia*)
by (*simp add*: *scalar-prod-def*)
also have ... = (∑ *ia* = *0*..<*m*+*n*. *?f ia*)
by (*rule sum.cong*, *insert A i dr dc*, *auto*) (*smt* (*verit*) *bezout-matrix-JNF-def carrier-matD*(*1*)

```
        dim-col-mat(1) index-col index-mult-mat(3) index-row(1) ja)
    also have ... = (∑ ia ∈ ({a,b} ∪ ({0..<m+n} − {a,b})). ?f ia)
      by (rule sum.cong, insert a a-dr b A ja, auto)
    also have ... = sum ?f {a,b} + sum ?f ({0..<m+n} − {a,b})
      by (rule sum.union-disjoint, auto)
     finally have BM-A-ija-eq: (?BM * A) $$ (i,ja) = sum ?f {a,b} + sum ?f
({0..<m+n} − {a,b}) by auto
    show ?thesis
    proof (cases i = a)
      case True
      have sum0: sum ?f ({0..<m+n} − {a,b}) = 0
      proof (rule sum.neutral, rule)
        fix x assume x: x ∈ {0..<m + n} − {a, b}
        hence xm: x < m+n by auto
        have x-not-i: x ≠ i using True x by blast
        have x-dr: x < dim-row A using x A by auto
        have bezout-matrix-JNF A a b j euclid-ext2 $$ (i, x) = 0
          unfolding bezout-matrix-JNF-def
          unfolding index-mat(1)[OF i-dr x-dr] using x-not-i x by auto
       thus bezout-matrix-JNF A a b j euclid-ext2 $$ (i, x) * A $$ (x, ja) = 0 by
auto
      qed
      have fa: bezout-matrix-JNF A a b j euclid-ext2 $$ (i, a) = p
        unfolding bezout-matrix-JNF-def index-mat(1)[OF i-dr a-dr] using True
pquvd
        by (auto, metis split-conv)
      have fb: bezout-matrix-JNF A a b j euclid-ext2 $$ (i, b) = q
        unfolding bezout-matrix-JNF-def index-mat(1)[OF i-dr b-dr] using True
pquvd ab
        by (auto, metis split-conv)
       have sum ?f {a,b} + sum ?f ({0..<m+n} − {a,b}) = ?f a + ?f b using
sum0 by (simp add: ab)
      also have ... = p * A $$ (a, ja) + q * A $$ (b, ja) unfolding fa fb by simp
      also have ... = ?A $$ (i,ja) using A True dr i ja by auto
      finally show ?thesis using BM-A-ija-eq by simp
    next
      case False note i-not-a = False
      show ?thesis
      proof (cases i=b)
        case True
        have sum0: sum ?f ({0..<m+n} − {a,b}) = 0
        proof (rule sum.neutral, rule)
          fix x assume x: x ∈ {0..<m + n} − {a, b}
          hence xm: x < m+n by auto
          have x-not-i: x ≠ i using True x by blast
          have x-dr: x < dim-row A using x A by auto
          have bezout-matrix-JNF A a b j euclid-ext2 $$ (i, x) = 0
            unfolding bezout-matrix-JNF-def
            unfolding index-mat(1)[OF i-dr x-dr] using x-not-i x by auto
```

thus *bezout-matrix-JNF A a b j euclid-ext2* $$ (*i*, *x*) ∗ *A* $$ (*x*, *ja*) = *0* **by** *auto*
qed
have *fa*: *bezout-matrix-JNF A a b j euclid-ext2* $$ (*i*, *a*) = *u*
unfolding *bezout-matrix-JNF-def index-mat*(*1*)[*OF i-dr a-dr*] **using** *True i-not-a pquvd*
by (*auto*, *metis split-conv*)
have *fb*: *bezout-matrix-JNF A a b j euclid-ext2* $$ (*i*, *b*) = *v*
unfolding *bezout-matrix-JNF-def index-mat*(*1*)[*OF i-dr b-dr*] **using** *True i-not-a pquvd ab*
by (*auto*, *metis split-conv*)
have *sum ?f* {*a*,*b*} + *sum ?f* ({*0*..<*m*+*n*} − {*a*,*b*}) = *?f a* + *?f b* **using** *sum0* **by** (*simp add*: *ab*)
also have ... = *u* ∗ *A* $$ (*a*, *ja*) + *v* ∗ *A* $$ (*b*, *ja*) **unfolding** *fa fb* **by** *simp*
also have ... = *?A* $$ (*i*,*ja*) **using** *A True i-not-a dr i ja* **by** *auto*
finally show *?thesis* **using** *BM-A-ija-eq* **by** *simp*
next
case *False* **note** *i-not-b* = *False*
have *sum0*: *sum ?f* ({*0*..<*m*+*n*} − {*a*,*b*} − {*i*}) = *0*
proof (*rule sum.neutral*, *rule*)
fix *x* **assume** *x*: *x* ∈ {*0*..<*m* + *n*} − {*a*, *b*} − {*i*}
hence *xm*: *x* < *m*+*n* **by** *auto*
have *x-not-i*: *x* ≠ *i* **using** *x* **by** *blast*
have *x-dr*: *x* < *dim-row A* **using** *x A* **by** *auto*
have *bezout-matrix-JNF A a b j euclid-ext2* $$ (*i*, *x*) = *0*
unfolding *bezout-matrix-JNF-def*
unfolding *index-mat*(*1*)[*OF i-dr x-dr*] **using** *x-not-i x* **by** *auto*
thus *bezout-matrix-JNF A a b j euclid-ext2* $$ (*i*, *x*) ∗ *A* $$ (*x*, *ja*) = *0* **by** *auto*
qed
have *fa*: *bezout-matrix-JNF A a b j euclid-ext2* $$ (*i*, *a*) = *0*
unfolding *bezout-matrix-JNF-def index-mat*(*1*)[*OF i-dr a-dr*] **using** *False i-not-a pquvd*
by *auto*
have *fb*: *bezout-matrix-JNF A a b j euclid-ext2* $$ (*i*, *b*) = *0*
unfolding *bezout-matrix-JNF-def index-mat*(*1*)[*OF i-dr b-dr*] **using** *False i-not-a pquvd*
by *auto*
have *sum ?f* ({*0*..<*m*+*n*} − {*a*,*b*}) = *sum ?f* (*insert i* ({*0*..<*m*+*n*} − {*a*,*b*} − {*i*}))
by (*rule sum.cong*, *insert i-dr A i-not-a i-not-b*, *auto*)
also have ... = *?f i* + *sum ?f* ({*0*..<*m*+*n*} − {*a*,*b*} − {*i*}) **by** (*rule sum.insert*, *auto*)
also have ... = *?f i* **using** *sum0* **by** *simp*
also have ... = *?A* $$ (*i*,*ja*)
unfolding *bezout-matrix-JNF-def* **using** *i-not-a i-not-b A dr i ja* **by** *fastforce*
finally show *?thesis* **unfolding** *BM-A-ija-eq* **by** (*simp add*: *ab fa fb*)
qed

```
  qed
 qed
qed


context proper-mod-operation
begin

lemma reduce-invertible-mat:
 assumes A': A' ∈ carrier-mat m n and a: a<m and j: 0<n and b: b<m and
ab: a ≠ b
 and A-def: A = A' @_r (D ·_m (1_m n))
 and Aaj: A $$ (a,0) ≠ 0
 and a-less-b: a < b
 and mn: m≥n
 and D-ge0: D > 0
shows ∃P. invertible-mat P ∧ P ∈ carrier-mat (m+n) (m+n) ∧ (reduce a b D
A) = P * A (is ?thesis1)
proof −
 obtain p q u v d where pquvd: (p,q,u,v,d) = euclid-ext2 (A$$(a,0)) (A$$(b,0))
  by (metis prod-cases5)
 let ?A = Matrix.mat (dim-row A) (dim-col A)
       (λ(i,k). if i = a then (p*A$$(a,k) + q*A$$(b,k))
             else if i = b then u * A$$(a,k) + v * A$$(b,k)
             else A$$(i,k)
        )
 have D: D ·_m 1_m n ∈ carrier-mat n n by auto
 have A: A ∈ carrier-mat (m+n) n using A-def A' by simp
 hence A-carrier: ?A ∈ carrier-mat (m+n) n by auto

 let ?BM = bezout-matrix-JNF A a b 0 euclid-ext2
 have A'-BZ-A: ?A = ?BM * A
  by (rule bezout-matrix-JNF-mult-eq[OF A' - - ab A-def D pquvd], insert a b,
auto)
 have invertible-bezout: invertible-mat ?BM
  by (rule invertible-bezout-matrix-JNF[OF A is-bezout-ext-euclid-ext2 a-less-b -
j Aaj],
     insert a-less-b b, auto)
 have BM: ?BM ∈ carrier-mat (m+n) (m+n) unfolding bezout-matrix-JNF-def
using A by auto

 define xs where xs = [0..<n]
 let ?reduce-a = reduce-row-mod-D ?A a xs D m
 let ?A' = mat-of-rows n [Matrix.row ?A i. i ← [0..<m]]
 have A-A'-D: ?A = ?A' @_r D ·_m 1_m n
 proof (rule matrix-append-rows-eq-if-preserves[OF A-carrier D], rule+)
  fix i j assume i: i ∈ {m..<m + n} and j: j < n
```

have *?A $$ (i,j) = A $$ (i,j)* **using** *i a b A j* **by** *auto*
also have *... = (if i < dim-row A' then A' $$(i,j) else (D ·$_m$ (1$_m$ n))$$(i−m,j))*
by (*unfold A-def*, *rule append-rows-nth*[*OF A' D - j*], *insert i*, *auto*)
also have *... = (D ·$_m$ 1$_m$ n) $$ (i − m, j)* **using** *i A'* **by** *auto*
finally show *?A $$ (i,j) = (D ·$_m$ 1$_m$ n) $$ (i − m, j)* **.**
qed
have *reduce-a-eq*: *?reduce-a = Matrix.mat (dim-row ?A) (dim-col ?A)*
(λ(i, k). if i = a ∧ k ∈ set xs then if k = 0 then if D dvd ?A$$(i,k) then D
else ?A $$ (i, k) else ?A $$ (i, k) gmod D else ?A $$ (i, k))
by (*rule reduce-row-mod-D*[*OF A-A'-D - a -*], *insert xs-def mn D-ge0*, *auto*)
have *reduce-a*: *?reduce-a ∈ carrier-mat (m+n) n* **using** *reduce-a-eq A* **by** *auto*
have *∃ P. P ∈ carrier-mat (m + n) (m + n) ∧ invertible-mat P ∧ ?reduce-a = P ∗ ?A*
by (*rule reduce-row-mod-D-invertible-mat*[*OF A-A'-D - a*], *insert xs-def mn*, *auto*)
from *this* **obtain** *P* **where** *P*: *P ∈ carrier-mat (m + n) (m + n)* **and** *inv-P*: *invertible-mat P*
and *reduce-a-PA*: *?reduce-a = P ∗ ?A* **by** *blast*
define *ys* **where** *ys = [1..<n]*
let *?reduce-b = reduce-row-mod-D ?reduce-a b ys D m*
let *?B' = mat-of-rows n [Matrix.row ?reduce-a i. i ← [0..<m]]*
have *reduce-a-B'-D*: *?reduce-a = ?B' @$_r$ D ·$_m$ 1$_m$ n*
proof (*rule matrix-append-rows-eq-if-preserves*[*OF reduce-a D*], *rule+*)
fix *i ja* **assume** *i*: *i ∈ {m..<m + n}* **and** *ja*: *ja < n*
have *i-not-a*:*i≠a* **and** *i-not-b*: *i≠b* **using** *i a b* **by** *auto*
have *?reduce-a $$ (i,ja) = ?A $$ (i, ja)*
unfolding *reduce-a-eq* **using** *i i-not-a i-not-b ja A* **by** *auto*
also have *... = A $$ (i,ja)* **using** *i i-not-a i-not-b ja A* **by** *auto*
also have *... = (D ·$_m$ 1$_m$ n) $$ (i − m, ja)*
by (*smt* (*verit*) *D append-rows-nth A' A-def atLeastLessThan-iff*
carrier-matD(*1*) *i ja less-irrefl-nat nat-SN.compat*)
finally show *?reduce-a $$ (i,ja) = (D ·$_m$ 1$_m$ n) $$ (i − m, ja)* **.**
qed
have *reduce-b-eq*: *?reduce-b = Matrix.mat (dim-row ?reduce-a) (dim-col ?reduce-a)*

(λ(i, k). if i = b ∧ k ∈ set ys then if k = 0 then if D dvd ?reduce-a$$(i,k) then D else ?reduce-a $$ (i, k)
else ?reduce-a $$ (i, k) gmod D else ?reduce-a $$ (i, k))
by (*rule reduce-row-mod-D*[*OF reduce-a-B'-D - b - - mn*], *unfold ys-def*, *insert D-ge0*, *auto*)
have *∃ P. P ∈ carrier-mat (m + n) (m + n) ∧ invertible-mat P ∧ ?reduce-b = P ∗ ?reduce-a*
by (*rule reduce-row-mod-D-invertible-mat*[*OF reduce-a-B'-D - b - mn*], *insert ys-def*, *auto*)
from *this* **obtain** *Q* **where** *Q*: *Q ∈ carrier-mat (m + n) (m + n)* **and** *inv-Q*: *invertible-mat Q*
and *reduce-b-Q-reduce*: *?reduce-b = Q ∗ ?reduce-a* **by** *blast*
have *reduce-b-eq-reduce*: *?reduce-b = (reduce a b D A)*
proof (*rule eq-matI*)

```
  show dr-eq: dim-row ?reduce-b = dim-row (reduce a b D A)
    and dc-eq: dim-col ?reduce-b = dim-col (reduce a b D A)
    using reduce-preserves-dimensions by auto
  fix i ja assume i: i<dim-row (reduce a b D A) and ja: ja< dim-col (reduce a
b D A)
  have im: i<m+n using A i reduce-preserves-dimensions(1) by auto
  have ja-n: ja<n using A ja reduce-preserves-dimensions(2) by auto
  show ?reduce-b $$ (i,ja) = (reduce a b D A) $$ (i,ja)
  proof (cases (i≠a ∧ i≠b))
    case True
    have ?reduce-b $$ (i,ja) = ?reduce-a $$ (i,ja) unfolding reduce-b-eq
        by (smt (verit) True dr-eq dc-eq i index-mat(1) ja prod.simps(2) re-
duce-row-mod-D-preserves-dimensions)
    also have ... = ?A $$ (i,ja)
      by (smt (verit) A True carrier-matD(2) dim-col-mat(1) dim-row-mat(1) i
index-mat(1) ja-n
          reduce-a-eq reduce-preserves-dimensions(1) split-conv)
    also have ... = A $$ (i,ja) using A True im ja-n by auto
    also have ... = (reduce a b D A) $$ (i,ja) unfolding reduce-alt-def-not0[OF
Aaj pquvd]
      using im ja-n A True by auto
    finally show ?thesis .
  next
    case False note a-or-b = False
    show ?thesis
    proof (cases i=a)
      case True note ia = True
      hence i-not-b: i≠b using ab by auto
      show ?thesis
      proof −
        have ja-in-xs: ja ∈ set xs
          unfolding xs-def using True ja-n im a A unfolding set-filter by auto
         have 1: ?reduce-b $$ (i,ja) = ?reduce-a $$ (i,ja) unfolding reduce-b-eq

            by (smt (verit) ab dc-eq dim-row-mat(1) dr-eq i ia index-mat(1) ja
prod.simps(2)
              reduce-b-eq reduce-row-mod-D-preserves-dimensions(2))
        show ?thesis
        proof (cases ja = 0 ∧ D dvd p∗A$$(a,ja) + q∗A$$(b,ja))
          case True
          have ?reduce-a $$ (i,ja) = D
           unfolding reduce-a-eq using True ab a-or-b i-not-b ja-n im a A ja-in-xs
False by auto
          also have ... = (reduce a b D A) $$ (i,ja)
            unfolding reduce-alt-def-not0[OF Aaj pquvd]
            using True a-or-b i-not-b ja-n im A False
            by auto
          finally show ?thesis using 1 by simp
        next
```

```
          case False note nc1 = False
          show ?thesis
          proof (cases ja=0)
            case True
            then show ?thesis
           by (smt (verit) 1 A assms(3) assms(7) dim-col-mat(1) dim-row-mat(1)
euclid-ext2-works i ia im index-mat(1)
              ja ja-in-xs old.prod.case pquvd reduce-gcd reduce-preserves-dimensions
reduce-a-eq)
          next
            case False
            have ?reduce-a $$ (i,ja) = ?A $$ (i, ja) gmod D
                unfolding reduce-a-eq using True ab a-or-b i-not-b ja-n im a A
ja-in-xs False by auto
            also have ... = (reduce a b D A) $$ (i,ja)
            unfolding reduce-alt-def-not0[OF Aaj pquvd] using True a-or-b i-not-b
ja-n im A False by auto
            finally show ?thesis using 1 by simp
        qed
      qed
    qed
    next
      case False note i-not-a = False
      have i-drb: i<dim-row ?reduce-b
        and i-dra: i<dim-row ?reduce-a
        and ja-drb: ja < dim-col ?reduce-b
       and ja-dra: ja < dim-col ?reduce-a using reduce-carrier[OF A] i ja A dr-eq
dc-eq by auto
        have ib: i=b using False a-or-b by auto
      show ?thesis
      proof (cases ja ∈ set ys)
        case True note ja-in-ys = True
        hence ja-not0: ja ≠ 0 unfolding ys-def by auto
        have ?reduce-b $$ (i,ja) = (if ja = 0 then if D dvd ?reduce-a$$(i,ja) then
D
            else ?reduce-a $$ (i, ja) else ?reduce-a $$ (i, ja) gmod D)
          unfolding reduce-b-eq using i-not-a True ja ja-in-ys
          by (smt (verit) i-dra ja-dra a-or-b index-mat(1) prod.simps(2))
        also have ... = (if ja = 0 then if D dvd ?reduce-a$$(i,ja) then D else ?A
$$ (i, ja) else ?A $$ (i, ja) gmod D)
         unfolding reduce-a-eq using True ab a-or-b ib False ja-n im a A ja-in-ys
by auto
        also have ... = (reduce a b D A) $$ (i,ja)
          unfolding reduce-alt-def-not0[OF Aaj pquvd] using True ja-not0 False
a-or-b ib ja-n im A
          using i-not-a by auto
        finally show ?thesis .
      next
        case False
```

```
        hence ja0:ja = 0 using ja-n unfolding ys-def by auto
        have rw0: u * A $$ (a, ja) + v * A $$ (b, ja) = 0
          unfolding euclid-ext2-works[OF pquvd[symmetric]] ja0
       by (smt (verit) euclid-ext2-works[OF pquvd[symmetric]] more-arith-simps(11)
mult.commute mult-minus-left)
          have ?reduce-b $$ (i,ja) = ?reduce-a $$ (i,ja) unfolding reduce-b-eq

         by (smt (verit) False a-or-b dc-eq dim-row-mat(1) dr-eq i index-mat(1)
ja
             prod.simps(2) reduce-b-eq reduce-row-mod-D-preserves-dimensions(2))
        also have ... = ?A $$ (i, ja)
          unfolding reduce-a-eq using False ab a-or-b i-not-a ja-n im a A  by
auto
        also have ... = u * A $$ (a, ja) + v * A $$ (b, ja)
        by (smt (verit, ccfv-SIG) A ‹ja = 0› assms(3) assms(5) carrier-matD(2)
i ib index-mat(1)
            old.prod.case reduce-preserves-dimensions(1))
        also have ... = (reduce a b D A) $$ (i,ja)
          unfolding reduce-alt-def-not0[OF Aaj pquvd]
          using False a-or-b i-not-a ja-n im A ja0 by auto
        finally show ?thesis .
      qed
    qed
  qed
  have inv-QPBM: invertible-mat (Q * P * ?BM)
   by (meson BM P Q inv-P inv-Q invertible-bezout invertible-mult-JNF mult-carrier-mat)
  moreover have (Q*P*?BM) ∈ carrier-mat (m + n) (m + n) using BM P Q
by auto
  moreover have (reduce a b D A) = (Q*P*?BM) * A
  proof −
    have ?BM * A = ?A using A'-BZ-A by auto
    hence P * (?BM * A) = ?reduce-a using reduce-a-PA by auto
    hence Q * (P * (?BM * A)) = ?reduce-b using reduce-b-Q-reduce by auto
    thus ?thesis using reduce-b-eq-reduce
    by (smt (verit) A A'-BZ-A A-carrier BM P Q assoc-mult-mat mn mult-carrier-mat
reduce-a-PA)
  qed
  ultimately show ?thesis by blast
qed


lemma reduce-abs-invertible-mat:
  assumes A': A' ∈ carrier-mat m n and a: a<m and j: 0<n and b: b<m and
ab: a ≠ b
  and A-def: A = A' @r (D ·m (1m n))
  and Aaj: A $$ (a,0) ≠ 0
  and a-less-b: a < b
  and mn: m≥n
```

and *D-ge0*: $D > 0$
shows $\exists P.$ *invertible-mat* $P \wedge P \in$ *carrier-mat* $(m+n)$ $(m+n) \wedge$ (*reduce-abs a b D A*) $= P * A$ (**is** *?thesis1*)
proof −
obtain *p q u v d* **where** *pquvd*: $(p,q,u,v,d) =$ *euclid-ext2* (*A$$(a,0)*) (*A$$(b,0)*)
by (*metis prod-cases5*)
let *?A* = *Matrix.mat* (*dim-row A*) (*dim-col A*)
(λ(*i*,*k*). *if i* = *a then* (*p*∗*A$$(a,k)* + *q*∗*A$$(b,k)*)
else if i = *b then u* ∗ *A$$(a,k)* + *v* ∗ *A$$(b,k)*
else A$$(i,k)
)
have *D*: $D \cdot_m 1_m\ n \in$ *carrier-mat n n* **by** *auto*
have *A*: $A \in$ *carrier-mat* $(m+n)$ *n* **using** *A-def A′* **by** *simp*
hence *A-carrier*: *?A* ∈ *carrier-mat* $(m+n)$ *n* **by** *auto*

let *?BM* = *bezout-matrix-JNF A a b 0 euclid-ext2*
have *A′-BZ-A*: *?A* = *?BM* ∗ *A*
by (*rule bezout-matrix-JNF-mult-eq*[*OF A′ - - ab A-def D pquvd*], *insert a b*, *auto*)
have *invertible-bezout*: *invertible-mat ?BM*
by (*rule invertible-bezout-matrix-JNF*[*OF A is-bezout-ext-euclid-ext2 a-less-b - j Aaj*],
insert a-less-b b, *auto*)
have *BM*: *?BM* ∈ *carrier-mat* $(m+n)$ $(m+n)$ **unfolding** *bezout-matrix-JNF-def* **using** *A* **by** *auto*

define *xs* **where** *xs* = *filter* (λ*i*. *abs* (*?A $$ (a,i)*) > *D*) [*0*..<*n*]
let *?reduce-a* = *reduce-row-mod-D-abs ?A a xs D m*
let *?A′* = *mat-of-rows n* [*Matrix.row ?A i*. *i* ← [*0*..<*m*]]
have *A-A′-D*: *?A* = *?A′* @$_r$ $D \cdot_m 1_m\ n$
proof (*rule matrix-append-rows-eq-if-preserves*[*OF A-carrier D*], *rule*+)
fix *i j* **assume** *i*: *i* ∈ {*m*..<*m* + *n*} **and** *j*: *j* < *n*
have *?A $$ (i,j)* = *A $$ (i,j)* **using** *i a b A j* **by** *auto*
also have ... = (*if i* < *dim-row A′ then A′ $$(i,j) else* ($D \cdot_m (1_m\ n)$)$$(*i*−*m*,*j*))
by (*unfold A-def*, *rule append-rows-nth*[*OF A′ D - j*], *insert i*, *auto*)
also have ... = ($D \cdot_m 1_m\ n$) $$ (*i* − *m*, *j*) **using** *i A′* **by** *auto*
finally show *?A $$ (i,j)* = ($D \cdot_m 1_m\ n$) $$ (*i* − *m*, *j*) **.**
qed
have *reduce-a-eq*: *?reduce-a* = *Matrix.mat* (*dim-row ?A*) (*dim-col ?A*)
(λ(*i*, *k*). *if i* = *a* ∧ *k* ∈ *set xs then*
if k = *0* ∧ *D dvd ?A$$(i,k) then D else ?A $$ (i*, *k) gmod D else ?A $$ (i*, *k*))
by (*rule reduce-row-mod-D-abs*[*OF A-A′-D - a -*], *insert xs-def mn D-ge0*, *auto*)

have *reduce-a*: *?reduce-a* ∈ *carrier-mat* $(m+n)$ *n* **using** *reduce-a-eq A* **by** *auto*
have $\exists P.$ $P \in$ *carrier-mat* $(m + n)$ $(m + n) \wedge$ *invertible-mat* $P \wedge$ *?reduce-a* = P ∗ *?A*
by (*rule reduce-row-mod-D-abs-invertible-mat*[*OF A-A′-D - a*], *insert xs-def mn*, *auto*)

from *this* **obtain** P **where** P: $P \in$ *carrier-mat* $(m + n)$ $(m + n)$ **and** *inv-P*: *invertible-mat P*
and *reduce-a-PA*: *?reduce-a* $= P *$ *?A* **by** *blast*
define *ys* **where** *ys* $=$ *filter* $(\lambda i.$ *abs* $($*?A* \$\$ $(b,i)) > D)$ $[0..<n]$
let *?reduce-b* $=$ *reduce-row-mod-D-abs ?reduce-a b ys D m*
let *?B'* $=$ *mat-of-rows n* $[$*Matrix.row ?reduce-a i*. $i \leftarrow [0..<m]]$
have *reduce-a-B'-D*: *?reduce-a* $=$ *?B'* $@_r$ $D \cdot_m 1_m$ n
proof (*rule matrix-append-rows-eq-if-preserves*[*OF reduce-a D*], *rule*+)
fix *i ja* **assume** *i*: $i \in \{m..<m + n\}$ **and** *ja*: $ja < n$
have *i-not-a*:$i \neq a$ **and** *i-not-b*: $i \neq b$ **using** *i a b* **by** *auto*
have *?reduce-a* \$\$ $(i,ja) =$ *?A* \$\$ $(i,$ *ja*$)$
unfolding *reduce-a-eq* **using** *i i-not-a i-not-b ja A* **by** *auto*
also have ... $=$ *A* \$\$ (i,ja) **using** *i i-not-a i-not-b ja A* **by** *auto*
also have ... $= (D \cdot_m 1_m n)$ \$\$ $(i - m,$ *ja*$)$
by (*smt* (*verit*) *D append-rows-nth A' A-def atLeastLessThan-iff carrier-matD*(*1*) *i ja less-irrefl-nat nat-SN.compat*)
finally show *?reduce-a* \$\$ $(i,ja) = (D \cdot_m 1_m n)$ \$\$ $(i - m,$ *ja*$)$ **.**
qed
have *reduce-b-eq*: *?reduce-b* $=$ *Matrix.mat* (*dim-row ?reduce-a*) (*dim-col ?reduce-a*)
$(\lambda(i,$ *k*$).$ *if* $i = b \wedge k \in$ *set ys then if* $k = 0 \wedge D$ *dvd ?reduce-a*\$\$$(i,k)$ *then D else ?reduce-a* \$\$ $(i,$ *k*$)$ *gmod D else ?reduce-a* \$\$ $(i,$ *k*$))$
by (*rule reduce-row-mod-D-abs*[*OF reduce-a-B'-D - b - - mn*], *unfold ys-def*, *insert D-ge0*, *auto*)
have $\exists P.$ $P \in$ *carrier-mat* $(m + n)$ $(m + n) \wedge$ *invertible-mat* $P \wedge$ *?reduce-b* $= P *$ *?reduce-a*
by (*rule reduce-row-mod-D-abs-invertible-mat*[*OF reduce-a-B'-D - b - mn*], *insert ys-def*, *auto*)
from *this* **obtain** Q **where** Q: $Q \in$ *carrier-mat* $(m + n)$ $(m + n)$ **and** *inv-Q*: *invertible-mat Q*
and *reduce-b-Q-reduce*: *?reduce-b* $= Q *$ *?reduce-a* **by** *blast*
have *reduce-b-eq-reduce*: *?reduce-b* $=$ (*reduce-abs a b D A*)
proof (*rule eq-matI*)
show *dr-eq*: *dim-row ?reduce-b* $=$ *dim-row* (*reduce-abs a b D A*)
and *dc-eq*: *dim-col ?reduce-b* $=$ *dim-col* (*reduce-abs a b D A*)
using *reduce-preserves-dimensions* **by** *auto*
fix *i ja* **assume** *i*: $i <$ *dim-row* (*reduce-abs a b D A*) **and** *ja*: $ja <$ *dim-col* (*reduce-abs a b D A*)
have *im*: $i < m+n$ **using** *A i reduce-preserves-dimensions*(*3*) **by** *auto*
have *ja-n*: $ja < n$ **using** *A ja reduce-preserves-dimensions*(*4*) **by** *auto*
show *?reduce-b* \$\$ $(i,ja) =$ (*reduce-abs a b D A*) \$\$ (i,ja)
proof (*cases* $(i \neq a \wedge i \neq b)$)
case *True*
have *?reduce-b* \$\$ $(i,ja) =$ *?reduce-a* \$\$ (i,ja) **unfolding** *reduce-b-eq*
by (*smt* (*verit*) *True dr-eq dc-eq i index-mat*(*1*) *ja prod.simps*(*2*) *reduce-row-mod-D-preserves-dimensions-abs*)
also have ... $=$ *?A* \$\$ (i,ja)
by (*smt* (*verit*) *A True carrier-matD*(*2*) *dim-col-mat*(*1*) *dim-row-mat*(*1*) *i index-mat*(*1*) *ja-n*

```
        reduce-a-eq reduce-preserves-dimensions(3) split-conv)
    also have ... = A $$ (i,ja) using A True im ja-n by auto
   also have ... = (reduce-abs a b D A) $$ (i,ja) unfolding reduce-alt-def-not0[OF
Aaj pquvd]
      using im ja-n A True by auto
    finally show ?thesis .
  next
    case False note a-or-b = False
    show ?thesis
    proof (cases i=a)
      case True note ia = True
      hence i-not-b: i≠b using ab by auto
      show ?thesis
      proof (cases abs((p∗A$$(a,ja) + q∗A$$(b,ja))) > D)
        case True note ge-D = True
        have ja-in-xs: ja ∈ set xs
          unfolding xs-def using True ja-n im a A unfolding set-filter by auto
         have 1: ?reduce-b $$ (i,ja) = ?reduce-a $$ (i,ja) unfolding reduce-b-eq

           by (smt (verit) ab dc-eq dim-row-mat(1) dr-eq i ia index-mat(1) ja
prod.simps(2)
             reduce-b-eq reduce-row-mod-D-preserves-dimensions-abs(2))
        show ?thesis
        proof (cases ja = 0 ∧ D dvd p∗A$$(a,ja) + q∗A$$(b,ja))
          case True
          have ?reduce-a $$ (i,ja) = D
           unfolding reduce-a-eq using True ab a-or-b i-not-b ja-n im a A ja-in-xs
False by auto
          also have ... = (reduce-abs a b D A) $$ (i,ja)
            unfolding reduce-alt-def-not0[OF Aaj pquvd]
            using True a-or-b i-not-b ja-n im A False ge-D
            by auto
          finally show ?thesis using 1 by simp
        next
          case False
          have ?reduce-a $$ (i,ja) = ?A $$ (i, ja) gmod D
           unfolding reduce-a-eq using True ab a-or-b i-not-b ja-n im a A ja-in-xs
False by auto
          also have ... = (reduce-abs a b D A) $$ (i,ja)
           unfolding reduce-alt-def-not0[OF Aaj pquvd] using True a-or-b i-not-b
ja-n im A False by auto
          finally show ?thesis using 1 by simp
        qed
      next
        case False
        have ja-in-xs: ja ∉ set xs
          unfolding xs-def using False ja-n im a A unfolding set-filter by auto
           have ?reduce-b $$ (i,ja) = ?reduce-a $$ (i,ja) unfolding reduce-b-eq
```

```
            by (smt (verit) ab dc-eq dim-row-mat(1) dr-eq i ia index-mat(1) ja
prod.simps(2)
              reduce-b-eq reduce-row-mod-D-preserves-dimensions-abs(2))
          also have ... = ?A $$ (i, ja)
           unfolding reduce-a-eq using False ab a-or-b i-not-b ja-n im a A ja-in-xs
by auto
          also have ... = (reduce-abs a b D A) $$ (i,ja)
           unfolding reduce-alt-def-not0[OF Aaj pquvd] using False a-or-b i-not-b
ja-n im A by auto
          finally show ?thesis .
        qed
      next
        case False note i-not-a = False
        have i-drb: i<dim-row ?reduce-b
          and i-dra: i<dim-row ?reduce-a
          and ja-drb: ja < dim-col ?reduce-b
         and ja-dra: ja < dim-col ?reduce-a using reduce-carrier[OF A] i ja A dr-eq
dc-eq by auto
          have ib: i=b using False a-or-b by auto
        show ?thesis
        proof (cases abs((u*A$$(a,ja) + v * A$$(b,ja))) > D)
          case True note ge-D = True
          have ja-in-ys: ja ∈ set ys
           unfolding ys-def using True False ib ja-n im a b A unfolding set-filter
by auto
           have ?reduce-b $$ (i,ja) = (if ja = 0 ∧ D dvd ?reduce-a$$(i,ja) then D
else ?reduce-a $$ (i, ja) gmod D)
            unfolding reduce-b-eq using i-not-a True  ja ja-in-ys
            by (smt (verit) i-dra ja-dra a-or-b index-mat(1) prod.simps(2))
          also have ... = (if ja = 0 ∧ D dvd ?reduce-a$$(i,ja) then D else ?A $$ (i,
ja) gmod D)
           unfolding reduce-a-eq using True ab a-or-b ib False ja-n im a A ja-in-ys
by auto
          also have ... = (reduce-abs a b D A) $$ (i,ja)
          proof (cases ja = 0 ∧ D dvd ?reduce-a$$(i,ja))
            case True
            have ja0: ja=0 using True by auto
            have u * A $$ (a, ja) + v * A $$ (b, ja) = 0
              unfolding euclid-ext2-works[OF pquvd[symmetric]] ja0
          by (smt (verit) euclid-ext2-works[OF pquvd[symmetric]] more-arith-simps(11)
mult.commute mult-minus-left)
            hence abs-0: abs((u*A$$(a,ja) + v * A$$(b,ja))) = 0 by auto
            show ?thesis using abs-0 D-ge0 ge-D by linarith
          next
            case False
            then show ?thesis
              unfolding reduce-alt-def-not0[OF Aaj pquvd] using True ge-D False
a-or-b ib ja-n im A
              using i-not-a by auto
```

```
        qed
        finally show ?thesis .
      next
        case False
        have ja-in-ys: ja ∉ set ys
        unfolding ys-def using i-not-a False ib ja-n im a b A unfolding set-filter
by auto
          have ?reduce-b $$ (i,ja) = ?reduce-a $$ (i,ja) unfolding reduce-b-eq

          using i-dra ja-dra ja-in-ys by auto
        also have ... = ?A $$ (i, ja)
          unfolding reduce-a-eq using False ab a-or-b i-not-a ja-n im a A  by
auto
        also have ... = u * A $$ (a, ja) + v * A $$ (b, ja)
         unfolding reduce-a-eq using False ab a-or-b i-not-a ja-n im a A ja-in-ys
by auto
        also have ... = (reduce-abs a b D A) $$ (i,ja)
          unfolding reduce-alt-def-not0[OF Aaj pquvd]
          using False a-or-b i-not-a ja-n im A by auto
        finally show ?thesis .
      qed
    qed
  qed
  qed
  have inv-QPBM: invertible-mat (Q * P * ?BM)
   by (meson BM P Q inv-P inv-Q invertible-bezout invertible-mult-JNF mult-carrier-mat)
  moreover have (Q*P*?BM) ∈ carrier-mat (m + n) (m + n) using BM P Q
by auto
  moreover have (reduce-abs a b D A) = (Q*P*?BM) * A
  proof −
    have ?BM * A = ?A using A'-BZ-A by auto
    hence P * (?BM * A) = ?reduce-a using reduce-a-PA by auto
    hence Q * (P * (?BM * A)) = ?reduce-b using reduce-b-Q-reduce by auto
    thus ?thesis using reduce-b-eq-reduce
    by (smt (verit) A A'-BZ-A A-carrier BM P Q assoc-mult-mat mn mult-carrier-mat
reduce-a-PA)
  qed
  ultimately show ?thesis by blast
qed
```

```
lemma reduce-element-mod-D-case-m':
  assumes A-def: A = A' @_r  B and B: B∈carrier-mat n n
  and A': A' ∈ carrier-mat m n and a: a≤m and j: j<n
  and mn: m>=n and B1: B $$ (j, j) = D and B2: (∀ j'∈{0..<n}−{j}. B $$ (j,
j') = 0)
  and D0: D > 0
```

```
  shows reduce-element-mod-D A a j D m = Matrix.mat (dim-row A) (dim-col A)
        (λ(i,k). if i = a ∧ k = j then if j = 0 then if D dvd A$$(i,k) then D
              else A$$(i,k) else A$$(i,k) gmod D else A$$(i,k)) (is - = ?A)
proof (rule eq-matI)
  have jm: j<m using mn j by auto
  have A: A ∈ carrier-mat (m+n) n using A-def A' B mn by simp
  fix i ja assume i: i < dim-row ?A and ja: ja < dim-col ?A
  show reduce-element-mod-D A a j D m $$ (i, ja) = ?A $$ (i, ja)
 proof (cases i=a)
    case False
    have reduce-element-mod-D A a j D m = (if j = 0 then if D dvd A$$(a,j)
       then addrow (−((A$$(a,j) gdiv D)) + 1) a (j + m) A else A
       else addrow (−((A$$(a,j) gdiv D))) a (j + m) A)
      unfolding reduce-element-mod-D-def by simp
    also have ... $$ (i,ja) = A $$ (i, ja) unfolding mat-addrow-def using False
ja i by auto
    also have ... = ?A $$ (i,ja) using False using i ja by auto
    finally show ?thesis .
  next
    case True note ia = True
    have reduce-element-mod-D A a j D m
      = (if j = 0 then if D dvd A$$(a,j) then addrow (−((A$$(a,j) gdiv D)) + 1)
a (j + m) A else A
       else addrow (−((A$$(a,j) gdiv D))) a (j + m) A)
      unfolding reduce-element-mod-D-def by simp
    also have ... $$ (i,ja) = ?A $$ (i,ja)
    proof (cases ja = j)
      case True note ja-j = True
      have A $$ (j + m, ja) = B $$ (j,ja)
        by (rule append-rows-nth2[OF A' - A-def ], insert j ja A B mn, auto)
      also have ... = D using True j mn B1 B2 B by auto
      finally have A-ja-jaD: A $$ (j + m, ja) = D .

      show ?thesis
      proof (cases j=0 ∧ D dvd A$$(a,j))
        case True
        have 1: reduce-element-mod-D A a j D m = addrow (−((A$$(a,j) gdiv D))
+ 1) a (j + m) A
          using True ia ja-j unfolding reduce-element-mod-D-def by auto
       also have ... $$(i,ja) = (− (A $$ (a, j) gdiv D) + 1) ∗ A $$ (j + m, ja) +
A $$ (i, ja)
          unfolding mat-addrow-def using True ja-j ia
          using A i j by auto
       also have ... = D
       proof −
         have A $$ (i, ja) + D ∗ − (A $$ (i, ja) gdiv D) = 0
           using True ia ja-j using D0 by force
         then show ?thesis
              by (metis A-ja-jaD ab-semigroup-add-class.add-ac(1) add.commute
```

add-right-imp-eq ia int-distrib(2)
ja-j more-arith-simps(3) mult.commute mult-cancel-right1)
qed
also have ... = *?A* $$ (*i*,*ja*) **using** *True ia A i j ja-j* **by** *auto*
finally show *?thesis*
using *True 1* **by** *auto*
next
case *False*
show *?thesis*
proof (*cases j=0*)
case *True*
then show *?thesis*
using *False i ja* **by** *auto*
next
case *False*
have *?A* $$ (*i*,*ja*) = *A* $$ (*i*, *ja*) *gmod D* **using** *True ia A i j False* **by** *auto*
also have ... = *A* $$ (*i*, *ja*) − ((*A* $$ (*i*, *ja*) *gdiv D*) ∗ *D*)
by (*subst gmod-gdiv*[*OF D0*], *auto*)
also have ... = − (*A* $$ (*a*, *j*) *gdiv D*) ∗ *A* $$ (*j* + *m*, *ja*) + *A* $$ (*i*, *ja*)
unfolding *A-ja-jaD* **by** (*simp add*: *True ia*)
finally show *?thesis*
using *A False True i ia j* **by** *auto*
qed
qed
next
case *False*
have *A* $$ (*j* + *m*, *ja*) = *B* $$ (*j*,*ja*)
by (*rule append-rows-nth2*[*OF A′ - A-def*], *insert j mn ja A B*, *auto*)
also have ... = *0* **using** *False* **using** *A a mn ja j B2* **by** *force*
finally have *A-am-ja0*: *A* $$ (*j* + *m*, *ja*) = *0* **.**
then show *?thesis* **using** *False i ja* **by** *fastforce*
qed
finally show *?thesis* **.**
qed
next
show *dim-row* (*reduce-element-mod-D A a j D m*) = *dim-row ?A*
and *dim-col* (*reduce-element-mod-D A a j D m*) = *dim-col ?A*
using *reduce-element-mod-D-def* **by** *auto*
qed

lemma *reduce-element-mod-D-abs-case-m′*:
assumes *A-def*: *A* = *A′* @$_r$ *B* **and** *B*: *B*∈*carrier-mat n n*
and *A′*: *A′* ∈ *carrier-mat m n* **and** *a*: *a*≤*m* **and** *j*: *j*<*n*
and *mn*: *m*>=*n* **and** *B1*: *B* $$ (*j*, *j*) = *D* **and** *B2*: (∀ *j′*∈{*0*..<*n*}−{*j*}. *B* $$ (*j*, *j′*) = *0*)

```
  and D0: D > 0
  shows reduce-element-mod-D-abs A a j D m = Matrix.mat (dim-row A) (dim-col
A)
          (λ(i,k). if i = a ∧ k = j then if j = 0 ∧ D dvd A$$(i,k) then D else
A$$(i,k) gmod D else A$$(i,k)) (is - = ?A)
proof (rule eq-matI)
  have jm: j<m using mn j by auto
  have A: A ∈ carrier-mat (m+n) n using A-def A' B mn by simp
  fix i ja assume i: i < dim-row ?A and ja: ja < dim-col ?A
  show reduce-element-mod-D-abs A a j D m $$ (i, ja) = ?A $$ (i, ja)
 proof (cases i=a)
    case False
    have reduce-element-mod-D-abs A a j D m = (if j = 0 ∧ D dvd A$$(a,j)
       then addrow (−((A$$(a,j) gdiv D)) + 1) a (j + m) A
       else addrow (−((A$$(a,j) gdiv D))) a (j + m) A)
      unfolding reduce-element-mod-D-abs-def by simp
    also have ... $$ (i,ja) = A $$ (i, ja) unfolding mat-addrow-def using False
ja i by auto
    also have ... = ?A $$ (i,ja) using False using i ja by auto
    finally show ?thesis .
  next
    case True note ia = True
    have reduce-element-mod-D-abs A a j D m
      = (if j = 0 ∧ D dvd A$$(a,j) then addrow (−((A$$(a,j) gdiv D)) + 1) a (j
+ m) A
       else addrow (−((A$$(a,j) gdiv D))) a (j + m) A)
      unfolding reduce-element-mod-D-abs-def by simp
    also have ... $$ (i,ja) = ?A $$ (i,ja)
    proof (cases ja = j)
      case True note ja-j = True
      have A $$ (j + m, ja) = B $$ (j,ja)
        by (rule append-rows-nth2[OF A' - A-def ], insert j ja A B mn, auto)
      also have ... = D using True j mn B1 B2 B by auto
      finally have A-ja-jaD: A $$ (j + m, ja) = D .

      show ?thesis
      proof (cases j=0 ∧ D dvd A$$(a,j))
        case True
        have 1: reduce-element-mod-D-abs A a j D m = addrow (−((A$$(a,j) gdiv
D)) + 1) a (j + m) A
          using True ia ja-j unfolding reduce-element-mod-D-abs-def by auto
       also have ... $$(i,ja) = (− (A $$ (a, j) gdiv D) + 1) ∗ A $$ (j + m, ja) +
A $$ (i, ja)
          unfolding mat-addrow-def using True ja-j ia
          using A i j by auto
        also have ... = D
        proof −
          have A $$ (i, ja) + D ∗ − (A $$ (i, ja) gdiv D) = 0
            using True ia ja-j using D0 by force
```

```
        then show ?thesis
            by (metis A-ja-jaD ab-semigroup-add-class.add-ac(1) add.commute
add-right-imp-eq ia int-distrib(2)
            ja-j more-arith-simps(3) mult.commute mult-cancel-right1)
       qed
       also have ... = ?A $$ (i,ja) using True ia A i j ja-j by auto
       finally show ?thesis
        using True 1 by auto
     next
       case False
        have ?A $$ (i,ja) = A $$ (i, ja) gmod D using True ia A i j False by
auto
        also have ... = A $$ (i, ja) − ((A $$ (i, ja) gdiv D) ∗ D)
          by (subst gmod-gdiv[OF D0], auto)
        also have ... =  − (A $$ (a, j) gdiv D) ∗ A $$ (j + m, ja) + A $$ (i, ja)
          unfolding A-ja-jaD by (simp add: True ia)
        finally show ?thesis
          using A False True i ia j by auto
       qed
   next
     case False
     have A $$ (j + m, ja) = B $$ (j,ja)
       by (rule append-rows-nth2[OF A' - A-def ], insert j mn ja A B, auto)
     also have ... = 0 using False using A a mn ja j B2 by force
     finally have A-am-ja0: A $$ (j + m, ja) = 0 .
     then show ?thesis using False i ja by fastforce
   qed
   finally show ?thesis .
 qed
next
 show dim-row (reduce-element-mod-D-abs A a j D m) = dim-row ?A
   and dim-col (reduce-element-mod-D-abs A a j D m) = dim-col ?A
   using reduce-element-mod-D-abs-def by auto
qed


lemma reduce-row-mod-D-case-m':
 assumes A-def: A = A' @r B and B ∈ carrier-mat n n
   and A': A' ∈ carrier-mat m n and a < m
   and j: ∀j∈set xs. j<n ∧ (B $$ (j, j) = D) ∧ (∀j'∈{0..<n}−{j}. B $$ (j, j')
= 0)
   and d: distinct xs and m≥n
   and D: D > 0
 shows reduce-row-mod-D A a xs D m = Matrix.mat (dim-row A) (dim-col A)
       (λ(i,k). if i = a ∧ k ∈ set xs then if k = 0 then if D dvd A$$(i,k) then D
           else A$$(i,k) else A$$(i,k) gmod D else A$$(i,k))
 using assms
proof (induct A a xs D m arbitrary: A' B rule: reduce-row-mod-D.induct)
 case (1 A a D m)
```

then show *?case* **by** *force*
next
case (*2 A a x xs D m*)
note *A-A'B = 2.prems(1)*
note *B = 2.prems(2)*
note *A' = 2.prems(3)*
note *a = 2.prems(4)*
note *j = 2.prems(5)*
note *mn = 2.prems(7)*
note *d = 2.prems(6)*
let *?reduce-xs = (reduce-element-mod-D A a x D m)*
have *reduce-xs-carrier*: *?reduce-xs ∈ carrier-mat (m + n) n*
by (*metis 2.prems(1) 2.prems(2) 2.prems(3) add.right-neutral append-rows-def*

carrier-matD carrier-mat-triv index-mat-four-block(2,3) index-zero-mat(2,3)
reduce-element-mod-D-preserves-dimensions)
have *1*: *reduce-row-mod-D A a (x # xs) D m*
= reduce-row-mod-D ?reduce-xs a xs D m **by** *simp*
have *2*: *reduce-element-mod-D A a j D m = Matrix.mat (dim-row A) (dim-col A)*
(λ(i,k). if i = a ∧ k = j then if j = 0 then if D dvd A$$(i,k)
then D else A$$(i,k) else A$$(i,k) gmod D else A$$(i,k)) **if** *j∈set (x#xs)* **for** *j*
by (*rule reduce-element-mod-D-case-m'[OF A-A'B B A']*, *insert 2.prems that*, *auto*)
have *reduce-row-mod-D ?reduce-xs a xs D m =*
Matrix.mat (dim-row ?reduce-xs) (dim-col ?reduce-xs) (λ(i,k). if i = a ∧ k ∈ set xs
then if k = 0 then if D dvd ?reduce-xs $$ (i, k) then D else ?reduce-xs $$ (i, k) else
?reduce-xs $$ (i, k) gmod D else ?reduce-xs $$ (i, k))
proof (*rule 2.hyps[OF - B - a - - mn]*)
let *?A' = mat-of-rows n [Matrix.row (reduce-element-mod-D A a x D m) i. i ← [0..<m]]*
show *reduce-element-mod-D A a x D m = ?A' @$_r$ B*
proof (*rule matrix-append-rows-eq-if-preserves[OF reduce-xs-carrier B]*)
show *∀ i∈{m..<m + n}. ∀ j<n. reduce-element-mod-D A a x D m $$ (i, j) = B $$ (i − m, j)*
by (*smt (verit) A-A'B A' B a Metric-Arith.nnf-simps(7) add-diff-cancel-left' atLeastLessThan-iff*
carrier-matD index-mat-addrow(1) index-row(1) le-add-diff-inverse2 less-diff-conv
reduce-element-mod-D-def reduce-element-mod-D-preserves-dimensions reduce-xs-carrier
row-append-rows2)
qed
qed (*insert 2.prems*, *auto simp add*: *mat-of-rows-def*)
also have *... = Matrix.mat (dim-row A) (dim-col A)*
(λ(i,k). if i = a ∧ k ∈ set (x # xs) then if k = 0 then if D dvd A$$(i,k)

```
        then D else A$$(i,k) else A$$(i,k) gmod D else A$$(i,k)) (is ?lhs = ?rhs)
  proof (rule eq-matI)
    show dim-row ?lhs = dim-row ?rhs and dim-col ?lhs = dim-col ?rhs by auto
    fix i j assume i: i<dim-row ?rhs and j: j < dim-col ?rhs
    have jn: j<n using j 2.prems by (simp add: append-rows-def)
    have xn: x < n
      by (simp add: 2.prems(5))
    show ?lhs $$ (i,j) = ?rhs $$ (i,j)
    proof (cases i=a ∧ j ∈ set xs)
      case True note ia-jxs = True
      have j-not-x: j≠x using d True by auto
      show ?thesis
      proof (cases j=0 ∧ D dvd ?reduce-xs $$(i,j))
        case True
        have ?lhs $$ (i,j) = D
          using True i j ia-jxs by auto
        also have ... = ?rhs $$ (i,j) using i j j-not-x
         by (smt (verit) 2 calculation dim-col-mat(1) dim-row-mat(1) index-mat(1)
insert-iff list.set(2) prod.simps(2) xn)
        finally show ?thesis .
      next
        case False
        show ?thesis
        proof (cases j=0)
          case True
          then show ?thesis
         by (smt (verit) 2 dim-col-mat(1) dim-row-mat(1) i index-mat(1) insert-iff
j list.set(2) old.prod.case)
        next
          case False
          have ?lhs $$ (i,j) = ?reduce-xs $$ (i, j) gmod D
            using True False i j by auto
          also have ... = A $$ (i,j) gmod D using 2[OF ] j-not-x i j by auto
          also have ... = ?rhs $$ (i,j) using i j j-not-x
            using False True dim-col-mat(1) dim-row-mat(1) index-mat(1)
              list.set-intros(2) old.prod.case by auto
          finally show ?thesis .
        qed
      qed
    next
      case False
      show ?thesis using 2 i j xn
       by (smt (verit) False dim-col-mat(1) dim-row-mat(1) index-mat(1) insert-iff
list.set(2) prod.simps(2))
    qed
  qed
  finally show ?case using 1 by simp
qed
```

lemma *reduce-row-mod-D-abs-case-m'*:
assumes *A-def*: $A = A' \ @_r \ B$ **and** $B \in$ *carrier-mat n n*
and *A'*: $A' \in$ *carrier-mat m n* **and** $a < m$
and *j*: $\forall j \in$ *set xs*. $j<n \wedge (B \$\$ (j, j) = D) \wedge (\forall j' \in \{0..<n\}-\{j\}.\ B \$\$ (j, j') = 0)$
and *d*: *distinct xs* **and** $m \geq n$
and *D*: $D > 0$
shows *reduce-row-mod-D-abs A a xs D m* = *Matrix.mat* (*dim-row A*) (*dim-col A*)
(λ(*i*,*k*). *if i* = *a* $\wedge$ *k* $\in$ *set xs then if k* = 0 $\wedge$ *D dvd A*$$(*i*,*k*) *then D*
else A$$(*i*,*k*) *gmod D else A*$$(*i*,*k*))
using *assms*
proof (*induct A a xs D m arbitrary*: *A' B rule*: *reduce-row-mod-D-abs.induct*)
case (*1 A a D m*)
then show *?case* **by** *force*
next
case (*2 A a x xs D m*)
note *A-A'B* = *2.prems*(*1*)
note *B* = *2.prems*(*2*)
note *A'* = *2.prems*(*3*)
note *a* = *2.prems*(*4*)
note *j* = *2.prems*(*5*)
note *mn* = *2.prems*(*7*)
note *d* = *2.prems*(*6*)
let *?reduce-xs* = (*reduce-element-mod-D-abs A a x D m*)
have *reduce-xs-carrier*: *?reduce-xs* $\in$ *carrier-mat* (*m* + *n*) *n*
by (*metis 2.prems*(*1*) *2.prems*(*2*) *2.prems*(*3*) *add.right-neutral append-rows-def*
carrier-matD carrier-mat-triv index-mat-four-block(*2*,*3*) *index-zero-mat*(*2*,*3*)
reduce-element-mod-D-preserves-dimensions)
have *1*: *reduce-row-mod-D-abs A a* (*x* # *xs*) *D m*
= *reduce-row-mod-D-abs ?reduce-xs a xs D m* **by** *simp*
have *2*: *reduce-element-mod-D-abs A a j D m* = *Matrix.mat* (*dim-row A*) (*dim-col A*)
(λ(*i*,*k*). *if i* = *a* $\wedge$ *k* = *j then if j* = 0 $\wedge$ *D dvd A*$$(*i*,*k*)
then D else A$$(*i*,*k*) *gmod D else A*$$(*i*,*k*)) **if** *j*$\in$*set* (*x*#*xs*) **for** *j*
by (*rule reduce-element-mod-D-abs-case-m'*[*OF A-A'B B A'*], *insert 2.prems that*, *auto*)
have *reduce-row-mod-D-abs ?reduce-xs a xs D m* =
Matrix.mat (*dim-row ?reduce-xs*) (*dim-col ?reduce-xs*) (λ(*i*,*k*). *if i* = *a* $\wedge$ *k* $\in$ *set xs*
then if k = 0 $\wedge$ *D dvd ?reduce-xs* $$ (*i*, *k*) *then D else*
?reduce-xs $$ (*i*, *k*) *gmod D else ?reduce-xs* $$ (*i*, *k*))
proof (*rule 2.hyps*[*OF* - *B* - *a* - - *mn*])
let *?A'* = *mat-of-rows n* [*Matrix.row* (*reduce-element-mod-D-abs A a x D m*) *i*. *i* $\leftarrow$ [*0*..<*m*]]

```
    show reduce-element-mod-D-abs A a x D m = ?A' @_r B
    proof (rule matrix-append-rows-eq-if-preserves[OF reduce-xs-carrier B])
      show ∀ i∈{m..<m + n}. ∀ j<n. reduce-element-mod-D-abs A a x D m $$ (i,
j) = B $$ (i − m, j)
       by (smt (verit) A-A'B A' B a Metric-Arith.nnf-simps(7) add-diff-cancel-left'
atLeastLessThan-iff
            carrier-matD index-mat-addrow(1) index-row(1) le-add-diff-inverse2
less-diff-conv
          reduce-element-mod-D-abs-def reduce-element-mod-D-preserves-dimensions
reduce-xs-carrier
           row-append-rows2)
    qed
  qed (insert 2.prems, auto simp add: mat-of-rows-def)
  also have ... = Matrix.mat (dim-row A) (dim-col A)
        (λ(i,k). if i = a ∧ k ∈ set (x # xs) then if k = 0 ∧ D dvd A$$(i,k)
        then D else A$$(i,k) gmod D else A$$(i,k)) (is ?lhs = ?rhs)
  proof (rule eq-matI)
    show dim-row ?lhs = dim-row ?rhs and dim-col ?lhs = dim-col ?rhs by auto
    fix i j assume i: i<dim-row ?rhs and j: j < dim-col ?rhs
    have jn: j<n using j 2.prems by (simp add: append-rows-def)
    have xn: x < n
      by (simp add: 2.prems(5))
    show ?lhs $$ (i,j) = ?rhs $$ (i,j)
    proof (cases i=a ∧ j ∈ set xs)
      case True note ia-jxs = True
      have j-not-x: j≠x using d True by auto
      show ?thesis
      proof (cases j=0 ∧ D dvd ?reduce-xs $$(i,j))
        case True
        have ?lhs $$ (i,j) = D
          using True i j ia-jxs by auto
        also have ... = ?rhs $$ (i,j) using i j j-not-x
         by (smt (verit) 2 calculation dim-col-mat(1) dim-row-mat(1) index-mat(1)
insert-iff list.set(2) prod.simps(2) xn)
        finally show ?thesis .
      next
        case False
        have ?lhs $$ (i,j) = ?reduce-xs $$ (i, j) gmod D
         using True False i j by auto
       also have ... = A $$ (i,j) gmod D using 2[OF ] j-not-x i j by auto
       also have ... = ?rhs $$ (i,j) using i j j-not-x
        by (smt (verit) False True ‹Matrix.mat (dim-row ?reduce-xs)
          (dim-col ?reduce-xs) (λ(i, k). if i = a ∧ k ∈ set xs
          then if k = 0 ∧ D dvd  ?reduce-xs $$ (i, k)
          then D else  ?reduce-xs $$ (i, k) gmod D
          else  ?reduce-xs $$ (i, k)) $$ (i, j) =  ?reduce-xs $$ (i, j) gmod D›
            calculation dim-col-mat(1) dim-row-mat(1) dvd-imp-gmod-0[OF ‹D >
0›] index-mat(1)
            insert-iff list.set(2) gmod-0-imp-dvd prod.simps(2))
```

finally show *?thesis* **.**
qed
next
case *False*
show *?thesis* **using** *2 i j xn*
by (*smt* (*verit*) *False dim-col-mat*(*1*) *dim-row-mat*(*1*) *index-mat*(*1*) *insert-iff list.set*(*2*) *prod.simps*(*2*))
qed
qed
finally show *?case* **using** *1* **by** *simp*
qed

lemma
assumes *A-def*: $A = A' \ @_r \ B$ **and** *B*: $B \in$ *carrier-mat n n*
and *A'*: $A' \in$ *carrier-mat m n* **and** *a*: $a<m$ **and** *j*: $j<n$ **and** *mn*: $m \geq n$
shows *reduce-element-mod-D-invertible-mat-case-m*:
$\exists P.\ P \in$ *carrier-mat* $(m+n)\ (m+n) \wedge$ *invertible-mat* $P \wedge$ *reduce-element-mod-D A a j D m* $= P * A$ (**is** *?thesis1*)
and *reduce-element-mod-D-abs-invertible-mat-case-m*:
$\exists P.\ P \in$ *carrier-mat* $(m+n)\ (m+n) \wedge$ *invertible-mat* $P \wedge$
reduce-element-mod-D-abs A a j D m $= P * A$ (**is** *?thesis2*)
unfolding *atomize-conj*
proof (*rule conjI*; *cases* $j = 0 \wedge D$ *dvd A$$(a,j)*)
case *True*
let *?P* = *addrow-mat* $(m+n)\ (-\ (A\ \$\$\ (a,\ j)\ gdiv\ D) + 1)\ a\ (j + m)$
have *A*: $A \in$ *carrier-mat* $(m + n)\ n$ **using** *A-def A' B mn* **by** *auto*
have *reduce-element-mod-D-abs A a j D m* = *addrow* $(-\ (A\ \$\$\ (a,\ j)\ gdiv\ D) + 1)\ a\ (j + m)\ A$
unfolding *reduce-element-mod-D-abs-def* **using** *True* **by** *auto*
also have ... = *?P* $* A$ **by** (*rule addrow-mat*[*OF A*], *insert j mn*, *auto*)
finally have *rw*: *reduce-element-mod-D-abs A a j D m* = *?P* $* A$ **.**
have *reduce-element-mod-D A a j D m* = *addrow* $(-\ (A\ \$\$\ (a,\ j)\ gdiv\ D) + 1)\ a\ (j + m)\ A$
unfolding *reduce-element-mod-D-def* **using** *True* **by** *auto*
also have ... = *?P* $* A$ **by** (*rule addrow-mat*[*OF A*], *insert j mn*, *auto*)
finally have *reduce-element-mod-D A a j D m* = *?P* $* A$ **.**
moreover have *?P* $\in$ *carrier-mat* $(m+n)\ (m+n)$ **by** *simp*
moreover have *invertible-mat ?P*
by (*metis addrow-mat-carrier a det-addrow-mat dvd-mult-right invertible-iff-is-unit-JNF mult.right-neutral not-add-less2 semiring-gcd-class.gcd-dvd1*)
ultimately show *?thesis1* **and** *?thesis2* **using** *rw* **by** *blast+*
next
case *False*
show *?thesis1*
proof (*cases j=0*)
case *True*
have *reduce-element-mod-D A a j D m* = *A* **unfolding** *reduce-element-mod-D-def*

using *False True* **by** *auto*
then show *?thesis*
by (*metis A-def assms(2) assms(3) carrier-append-rows invertible-mat-one left-mult-one-mat one-carrier-mat*)
next
case *False*
let *?P* = *addrow-mat* (*m*+*n*) (− (*A* $$ (*a*, *j*) *gdiv D*)) *a* (*j* + *m*)
have *A*: *A* ∈ *carrier-mat* (*m* + *n*) *n* **using** *A-def B A′ mn* **by** *auto*
have *reduce-element-mod-D A a j D m* = *addrow* (− (*A* $$ (*a*, *j*) *gdiv D*)) *a* (*j* + *m*) *A*
unfolding *reduce-element-mod-D-def* **using** *False* **by** *auto*
also have ... = *?P* ∗ *A* **by** (*rule addrow-mat*[*OF A*], *insert j mn*, *auto*)
finally have *reduce-element-mod-D A a j D m* = *?P* ∗ *A* **.**
moreover have *?P* ∈ *carrier-mat* (*m*+*n*) (*m*+*n*) **by** *simp*
moreover have *invertible-mat ?P*
by (*metis addrow-mat-carrier a det-addrow-mat dvd-mult-right invertible-iff-is-unit-JNF mult.right-neutral not-add-less2 semiring-gcd-class.gcd-dvd1*)
ultimately show *?thesis* **by** *blast*
qed
show *?thesis2*
proof −
let *?P* = *addrow-mat* (*m*+*n*) (− (*A* $$ (*a*, *j*) *gdiv D*)) *a* (*j* + *m*)
have *A*: *A* ∈ *carrier-mat* (*m* + *n*) *n* **using** *A-def B A′ mn* **by** *auto*
have *reduce-element-mod-D-abs A a j D m* = *addrow* (− (*A* $$ (*a*, *j*) *gdiv D*)) *a* (*j* + *m*) *A*
unfolding *reduce-element-mod-D-abs-def* **using** *False* **by** *auto*
also have ... = *?P* ∗ *A* **by** (*rule addrow-mat*[*OF A*], *insert j mn*, *auto*)
finally have *reduce-element-mod-D-abs A a j D m* = *?P* ∗ *A* **.**
moreover have *?P* ∈ *carrier-mat* (*m*+*n*) (*m*+*n*) **by** *simp*
moreover have *invertible-mat ?P*
by (*metis addrow-mat-carrier a det-addrow-mat dvd-mult-right invertible-iff-is-unit-JNF mult.right-neutral not-add-less2 semiring-gcd-class.gcd-dvd1*)
ultimately show *?thesis* **by** *blast*
qed
qed

lemma *reduce-row-mod-D-invertible-mat-case-m*:
assumes *A-def*: *A* = *A′* @$_r$ *B* **and** *B* ∈ *carrier-mat n n*
and *A′*: *A′* ∈ *carrier-mat m n* **and** *a*: *a* < *m*
and *j*: ∀ *j*∈*set xs*. *j*<*n* ∧ (*B* $$ (*j*, *j*) = *D*) ∧ (∀ *j′*∈{*0*..<*n*}−{*j*}. *B* $$ (*j*, *j′*) = *0*)
and *mn*: *m*≥*n*
shows ∃ *P*. *P* ∈ *carrier-mat* (*m*+*n*) (*m*+*n*) ∧ *invertible-mat P* ∧
reduce-row-mod-D A a xs D m = *P* ∗ *A*
using *assms*
proof (*induct A a xs D m arbitrary*: *A′ B rule*: *reduce-row-mod-D.induct*)
case (*1 A a D m*)
show *?case* **by** (*rule exI*[*of - 1*$_m$ (*m*+*n*)], *insert 1.prems*, *auto simp add*: *ap-*

pend-rows-def)
next
case (*2 A a x xs D m*)
note *A-def* = *2.prems*(*1*)
note *B* = *2.prems*(*2*)
note *A′* = *2.prems*(*3*)
note *a* = *2.prems*(*4*)
note *j* = *2.prems*(*5*)
note *mn* = *2.prems*(*6*)
let *?reduce-xs* = (*reduce-element-mod-D A a x D m*)
have *1*: *reduce-row-mod-D A a* (*x # xs*) *D m*
= *reduce-row-mod-D ?reduce-xs a xs D m* **by** *simp*
have ∃ *P*. *P* ∈ *carrier-mat* (*m*+*n*) (*m*+*n*) ∧ *invertible-mat P* ∧
reduce-element-mod-D A a x D m = *P* ∗ *A*
by (*rule reduce-element-mod-D-invertible-mat-case-m*, *insert 2.prems*, *auto*)
from *this* **obtain** *P* **where** *P*: *P* ∈ *carrier-mat* (*m*+*n*) (*m*+*n*) **and** *inv-P*: *invertible-mat P*
and *R-P*: *reduce-element-mod-D A a x D m* = *P* ∗ *A* **by** *auto*
have ∃ *P*. *P* ∈ *carrier-mat* (*m* + *n*) (*m* + *n*) ∧ *invertible-mat P*
∧ *reduce-row-mod-D ?reduce-xs a xs D m* = *P* ∗ *?reduce-xs*
proof (*rule 2.hyps*)
let *?A′* = *mat-of-rows n* [*Matrix.row ?reduce-xs i*. *i* ← [*0*..<*m*]]
let *?B′* = *mat-of-rows n* [*Matrix.row ?reduce-xs i*. *i* ← [*m*..<*m*+*n*]]

show *reduce-xs-A′B′*: *?reduce-xs* = *?A′* @$_r$ *?B′*
by (*smt* (*verit*) *2*(*2*) *2*(*4*) *P R-P add.comm-neutral append-rows-def append-rows-split carrier-matD*
index-mat-four-block(*3*) *index-mult-mat*(*2*) *index-zero-mat*(*3*) *le-add1 reduce-element-mod-D-preserves-dimensions*(*2*))
show ∀ *j*∈*set xs*. *j* < *n* ∧ *?B′* $$ (*j*, *j*) = *D* ∧ (∀ *j′*∈{*0*..<*n*} − {*j*}. *?B′* $$ (*j*, *j′*) = *0*)
proof
fix *j* **assume** *j-in-xs*: *j* ∈ *set xs*
have *jn*: *j*<*n* **using** *j-in-xs j* **by** *auto*
have *?B′* $$ (*j*, *j*) = *?reduce-xs* $$ (*m*+*j*,*j*)
using *2*(*7*) *add-diff-cancel-left′ jn length-map length-upt*
by (*smt* (*verit*, *ccfv-SIG*) *1 2*(*4*) *A-def B P inv-P R-P add-strict-left-mono carrier-append-rows carrier-matD*(*1*)
carrier-matD(*2*) *index-mult-mat*(*2*) *index-row*(*1*) *mat-of-rows-index nth-map-upt reduce-row-mod-D-preserves-dimensions*(*2*))
also have ... = *B* $$ (*j*,*j*)
by (*smt* (*verit*) *2*(*2*) *2*(*5*) *A′ P R-P add-diff-cancel-left′ append-rows-def carrier-matD*
group-cancel.rule0 index-mat-addrow(*1*) *index-mat-four-block*(*1*) *index-mat-four-block*(*2*,*3*)
index-mult-mat(*2*) *index-zero-mat*(*3*) *jn le-add1 linorder-not-less nat-SN.plus-gt-right-mono*

reduce-element-mod-D-def reduce-element-mod-D-preserves-dimensions(*1*))
also have ... = *D* **using** *j j-in-xs* **by** *auto*

finally have *B'-jj*: *?B'* $$ (*j*, *j*) = *D* **by** *auto*
moreover have ∀*j'*∈{*0*..<*n*} − {*j*}. *?B'* $$ (*j*, *j'*) = *0*
proof
fix *j'* **assume** *j'*: *j'* ∈{*0*..<*n*} − {*j*}
then have *?B'* $$ (*j*, *j'*) = *?reduce-xs* $$ (*m*+*j*,*j'*)
by (*smt* (*z3*) *mn Diff-iff add.commute add-diff-cancel-left'*
append-rows-nth2 atLeastLessThan-iff diff-zero jn length-map length-upt

mat-of-rows-carrier(*1*) *nat-SN.compat reduce-xs-A'B'*)
also have ... = *B* $$ (*j*,*j'*)
by (*smt* (*verit*) *2*(*2*) *2*(*5*) *A' Diff-iff P R-P j' add.commute add-diff-cancel-left'*

append-rows-def atLeastLessThan-iff carrier-matD group-cancel.rule0
index-mat-addrow(*1*)
index-mat-four-block index-mult-mat(*2*) *index-zero-mat*(*3*) *jn nat-SN.plus-gt-right-mono*

not-add-less2 reduce-element-mod-D-def reduce-element-mod-D-preserves-dimensions(*1*))
also have ... = *0* **using** *j j-in-xs j'* **by** *auto*
finally show *?B'* $$ (*j*, *j'*) = *0* **.**
qed
ultimately show *j* < *n* ∧ *?B'* $$ (*j*, *j*) = *D* ∧ (∀*j'*∈{*0*..<*n*} − {*j*}. *?B'* $$
(*j*, *j'*) = *0*)
using *jn* **by** *blast*
qed
show *?A'* : *carrier-mat m n* **by** *auto*
show *?B'* : *carrier-mat n n* **by** *auto*
show *a*<*m* **using** *2.prems* **by** *auto*
show *n*≤*m* **using** *2.prems* **by** *auto*
qed
from *this* **obtain** *P2* **where** *P2*: *P2* ∈ *carrier-mat* (*m* + *n*) (*m* + *n*) **and**
inv-P2: *invertible-mat P2*
and *R-P2*: *reduce-row-mod-D ?reduce-xs a xs D m* = *P2* ∗ *?reduce-xs*
by *auto*
have *invertible-mat* (*P2* ∗ *P*) **using** *P P2 inv-P inv-P2 invertible-mult-JNF* **by**
blast
moreover have (*P2* ∗ *P*) ∈ *carrier-mat* (*m*+*n*) (*m*+*n*) **using** *P2 P* **by** *auto*
moreover have *reduce-row-mod-D A a* (*x* # *xs*) *D m* = (*P2* ∗ *P*) ∗ *A*
by (*smt* (*verit*) *P P2 R-P R-P2 1 assoc-mult-mat carrier-matD carrier-mat-triv*
index-mult-mat reduce-row-mod-D-preserves-dimensions)
ultimately show *?case* **by** *blast*
qed

lemma *reduce-row-mod-D-abs-invertible-mat-case-m*:
assumes *A-def*: *A* = *A'* @$_r$ *B* **and** *B* ∈ *carrier-mat n n*
and *A'*: *A'* ∈ *carrier-mat m n* **and** *a*: *a* < *m*
and *j*: ∀*j*∈*set xs*. *j*<*n* ∧ (*B* $$ (*j*, *j*) = *D*) ∧ (∀*j'*∈{*0*..<*n*}−{*j*}. *B* $$ (*j*, *j'*)

= 0)
  and *mn*: *m≥n*
 shows ∃ *P*. *P* ∈ *carrier-mat* (*m+n*) (*m+n*) ∧ *invertible-mat P* ∧
  reduce-row-mod-D-abs A a xs D m = *P* ∗ *A*
 using *assms*
proof (*induct A a xs D m arbitrary*: *A′ B rule*: *reduce-row-mod-D-abs.induct*)
 case (*1 A a D m*)
 show *?case* **by** (*rule exI*[*of - 1*$_m$ (*m+n*)], *insert 1.prems*, *auto simp add*: *append-rows-def*)
next
 case (*2 A a x xs D m*)
 note *A-def* = *2.prems*(*1*)
 note *B* = *2.prems*(*2*)
 note *A′* = *2.prems*(*3*)
 note *a* = *2.prems*(*4*)
 note *j* = *2.prems*(*5*)
 note *mn* = *2.prems*(*6*)
 let *?reduce-xs* = (*reduce-element-mod-D-abs A a x D m*)
 have *1*: *reduce-row-mod-D-abs A a* (*x* # *xs*) *D m*
  = *reduce-row-mod-D-abs ?reduce-xs a xs D m* **by** *simp*
 have ∃ *P*. *P* ∈ *carrier-mat* (*m+n*) (*m+n*) ∧ *invertible-mat P* ∧
  reduce-element-mod-D-abs A a x D m = *P* ∗ *A*
  by (*rule reduce-element-mod-D-abs-invertible-mat-case-m*, *insert 2.prems*, *auto*)
  from *this* **obtain** *P* **where** *P*: *P* ∈ *carrier-mat* (*m+n*) (*m+n*) **and** *inv-P*: *invertible-mat P*
  and *R-P*: *reduce-element-mod-D-abs A a x D m* = *P* ∗ *A* **by** *auto*
 have ∃ *P*. *P* ∈ *carrier-mat* (*m* + *n*) (*m* + *n*) ∧ *invertible-mat P*
  ∧ *reduce-row-mod-D-abs ?reduce-xs a xs D m* = *P* ∗ *?reduce-xs*
 proof (*rule 2.hyps*)
  let *?A′* = *mat-of-rows n* [*Matrix.row ?reduce-xs i*. *i* ← [*0*..<*m*]]
  let *?B′* = *mat-of-rows n* [*Matrix.row ?reduce-xs i*. *i* ← [*m*..<*m+n*]]

  show *reduce-xs-A′B′*: *?reduce-xs* = *?A′* @$_r$ *?B′*
   by (*smt* (*verit*) *2*(*2*) *2*(*4*) *P R-P add.comm-neutral append-rows-def append-rows-split carrier-matD*
    index-mat-four-block(*3*) *index-mult-mat*(*2*) *index-zero-mat*(*3*) *le-add1 reduce-element-mod-D-preserves-dimensions*(*4*))
  show ∀ *j*∈*set xs*. *j* < *n* ∧ *?B′* $$ (*j*, *j*) = *D* ∧ (∀ *j′*∈{*0*..<*n*} − {*j*}. *?B′* $$ (*j*, *j′*) = *0*)
  proof
   fix *j* **assume** *j-in-xs*: *j* ∈ *set xs*
   have *jn*: *j*<*n* **using** *j-in-xs j* **by** *auto*
   have *?B′* $$ (*j*, *j*) = *?reduce-xs* $$ (*m+j*,*j*)
    by (*smt* (*z3*) *2*(*7*) *Groups.add-ac*(*2*) *jn reduce-xs-A′B′ add-diff-cancel-left′ append-rows-nth2*
     diff-zero length-map length-upt mat-of-rows-carrier(*1*) *nat-SN.compat*)
   also have ... = *B* $$ (*j*,*j*)
    by (*smt* (*verit*) *2*(*2*) *2*(*5*) *A′ P R-P add-diff-cancel-left′ append-rows-def carrier-matD*

group-cancel.rule0 index-mat-addrow(1) index-mat-four-block(1) index-mat-four-block(2,3)
index-mult-mat(2) index-zero-mat(3) jn le-add1 linorder-not-less nat-SN.plus-gt-right-mono

reduce-element-mod-D-abs-def reduce-element-mod-D-preserves-dimensions(3))
also have *... = D* **using** *j j-in-xs* **by** *auto*
finally have *B'-jj*: *?B' $$ (j, j) = D* **by** *auto*
moreover have $\forall j' \in \{0..<n\} - \{j\}$. *?B' $$ (j, j') = 0*
proof
fix *j'* **assume** *j'*: $j' \in \{0..<n\} - \{j\}$
then
have *?B' $$ (j, j') = ?reduce-xs $$ (m+j,j')*
apply *simp*
by (*metis* (*mono-tags*, *lifting*) *2(4) A-def B P R-P add-less-cancel-left carrier-append-rows carrier-matD(1) carrier-matD(2) diff-add-inverse index-mult-mat(2) index-mult-mat(3) index-row(1) jn length-map length-upt mat-of-rows-index nth-map-upt*)
also have *... = B $$ (j,j')*
by (*smt* (*verit*) *2(2) 2(5) A' Diff-iff P R-P j' add.commute add-diff-cancel-left'*

append-rows-def atLeastLessThan-iff carrier-matD group-cancel.rule0 index-mat-addrow(1)
index-mat-four-block index-mult-mat(2) index-zero-mat(3) jn nat-SN.plus-gt-right-mono

not-add-less2 reduce-element-mod-D-abs-def reduce-element-mod-D-preserves-dimensions(3))
also have *... = 0* **using** *j j-in-xs j'* **by** *auto*
finally show *?B' $$ (j, j') = 0* **.**
qed
ultimately show $j < n \wedge$ *?B' $$ (j, j) = D* $\wedge (\forall j' \in \{0..<n\} - \{j\}$. *?B' $$ (j, j') = 0)*
using *jn* **by** *blast*
qed
show *?A'* : *carrier-mat m n* **by** *auto*
show *?B'* : *carrier-mat n n* **by** *auto*
show $a<m$ **using** *2.prems* **by** *auto*
show $n \leq m$ **using** *2.prems* **by** *auto*
qed
from *this* **obtain** *P2* **where** *P2*: *P2* $\in$ *carrier-mat* $(m + n)$ $(m + n)$ **and** *inv-P2*: *invertible-mat P2*
and *R-P2*: *reduce-row-mod-D-abs ?reduce-xs a xs D m = P2 * ?reduce-xs*
by *auto*
have *invertible-mat* $(P2 * P)$ **using** *P P2 inv-P inv-P2 invertible-mult-JNF* **by** *blast*
moreover have $(P2 * P) \in$ *carrier-mat* $(m+n)$ $(m+n)$ **using** *P2 P* **by** *auto*
moreover have *reduce-row-mod-D-abs A a* $(x \# xs)$ *D m* $= (P2 * P) * A$
by (*smt* (*verit*) *P P2 R-P R-P2 1 assoc-mult-mat carrier-matD carrier-mat-triv index-mult-mat reduce-row-mod-D-preserves-dimensions-abs*)
ultimately show *?case* **by** *blast*
qed

lemma *reduce-row-mod-D-case-m''*:
assumes *A-def*: $A = A' \,@_r\, B$ **and** $B \in$ *carrier-mat n n*
and *A'*: $A' \in$ *carrier-mat m n* **and** $a \leq m$
and *j*: $\forall j{\in}$*set xs*. $j{<}n \wedge (B\ \$\$\ (j,\ j) = D) \wedge (\forall j'{\in}\{0..{<}n\}{-}\{j\}.\ B\ \$\$\ (j,\ j') = 0)$
and *d*: *distinct xs* **and** $m{\geq}n$ **and** $0 \notin$ *set xs*
and $D > 0$
shows *reduce-row-mod-D A a xs D m* = *Matrix.mat* (*dim-row A*) (*dim-col A*)
(λ(*i*,*k*). *if* $i = a \wedge k \in$ *set xs then if* $k = 0$ *then if D dvd A*$$(*i*,*k*) *then D else A*$$(*i*,*k*) *else A*$$(*i*,*k*) *gmod D else A*$$(*i*,*k*))
using *assms*
proof (*induct A a xs D m arbitrary*: *A' B rule*: *reduce-row-mod-D.induct*)
case (*1 A a D m*)
then show *?case* **by** *force*
next
case (*2 A a x xs D m*)
note *A-A'B* = *2.prems*(*1*)
note *B* = *2.prems*(*2*)
note *A'* = *2.prems*(*3*)
note *a* = *2.prems*(*4*)
note *j* = *2.prems*(*5*)
note *mn* = *2.prems*(*7*)
note *d* = *2.prems*(*6*)
note *zero-not-xs* = *2.prems*(*8*)
let *?reduce-xs* = (*reduce-element-mod-D A a x D m*)
have *reduce-xs-carrier*: *?reduce-xs* $\in$ *carrier-mat* $(m + n)$ *n*
by (*metis 2.prems*(*1*) *2.prems*(*2*) *2.prems*(*3*) *add.right-neutral append-rows-def carrier-matD carrier-mat-triv index-mat-four-block*(*2*,*3*) *index-zero-mat*(*2*,*3*) *reduce-element-mod-D-preserves-dimensions*)
have *A*: *A*:*carrier-mat* $(m{+}n)$ *n* **using** *A' B A-A'B* **by** *blast*
have *1*: *reduce-row-mod-D A a* (*x* # *xs*) *D m*
= *reduce-row-mod-D ?reduce-xs a xs D m* **by** *simp*
have *2*: *reduce-element-mod-D A a j D m* = *Matrix.mat* (*dim-row A*) (*dim-col A*)
(λ(*i*,*k*). *if* $i = a \wedge k = j$ *then if* $j = 0$ *then if D dvd A*$$(*i*,*k*) *then D else A*$$(*i*,*k*) *else A*$$(*i*,*k*) *gmod D else A*$$(*i*,*k*)) **if** *j*$\in$*set* (*x*#*xs*)
for *j*
by (*rule reduce-element-mod-D-case-m'*[*OF A-A'B B A'*], *insert 2.prems that*, *auto*)
have *reduce-row-mod-D ?reduce-xs a xs D m* =
Matrix.mat (*dim-row ?reduce-xs*) (*dim-col ?reduce-xs*) (λ(*i*,*k*). *if* $i = a \wedge k \in$ *set xs*
then if $k{=}0$ *then if D dvd ?reduce-xs* $$ (*i*, *k*) *then D else ?reduce-xs* $$ (*i*, *k*)
else ?reduce-xs $$ (*i*, *k*) *gmod D else ?reduce-xs* $$ (*i*, *k*))

proof (*rule 2.hyps*[*OF - - - a - - mn*])
let *?A′ = mat-of-rows n* [*Matrix.row* (*reduce-element-mod-D A a x D m*) *i*. *i* ← [*0..<m*]]
define *B′* **where** *B′ = mat-of-rows n* [*Matrix.row ?reduce-xs i*. *i* ← [*m..<dim-row A*]]
show *A″*: *?A′* : *carrier-mat m n* **by** *auto*
show *B′*: *B′* : *carrier-mat n n* **unfolding** *B′-def* **using** *mn A* **by** *auto*
show *reduce-split*: *?reduce-xs = ?A′* @$_r$ *B′*
by (*metis B′-def append-rows-split carrier-matD*
reduce-element-mod-D-preserves-dimensions(*1*) *reduce-xs-carrier le-add1*)
show ∀*j*∈*set xs*. *j*<*n* ∧ (*B′* $$ (*j*, *j*) = *D*) ∧ (∀*j′*∈{*0..<n*}−{*j*}. *B′* $$ (*j*, *j′*) = *0*)
proof
fix *j* **assume** *j-xs*: *j*∈*set xs*
have *B* $$ (*j*,*j′*) = *B′* $$ (*j*,*j′*) **if** *j′*: *j′*<*n* **for** *j′*
proof −
have *B* $$ (*j*,*j′*) = *A* $$ (*m*+*j*,*j′*)
by (*smt* (*verit*) *A-A′B A A′ Groups.add-ac*(*2*) *j-xs add-diff-cancel-left′ append-rows-def carrier-matD j′*
index-mat-four-block(*1*) *index-mat-four-block*(*2*,*3*) *insert-iff j less-diff-conv list.set*(*2*) *not-add-less1*)
also have ... = *?reduce-xs* $$ (*m*+*j*,*j′*)
by (*smt* (*verit*, *ccfv-threshold*) *A″ diff-add-zero index-mat-addrow*(*3*) *neq0-conv*
a j zero-not-xs A add.commute add-diff-cancel-left′ reduce-element-mod-D-def
cancel-comm-monoid-add-class.diff-cancel carrier-matD index-mat-addrow(*1*) *j′*
j-xs le-eq-less-or-eq less-diff-conv less-not-refl2 list.set-intros(*2*) *nat-SN.compat*)
also have ... = *B′*$$ (*j*,*j′*)
by (*smt* (*verit*) *B A A′ A-A′B B′ A″ reduce-split add.commute add-diff-cancel-left′ j′ not-add-less1*
append-rows-def carrier-matD index-mat-four-block j j-xs less-diff-conv list.set-intros(*2*))
finally show *?thesis* **.**
qed
thus *j* < *n* ∧ *B′* $$ (*j*, *j*) = *D* ∧ (∀*j′*∈{*0..<n*} − {*j*}. *B′* $$ (*j*, *j′*) = *0*) **using** *j*
by (*metis Diff-iff atLeastLessThan-iff insert-iff j-xs list.simps*(*15*))
qed
qed (*insert 2.prems*, *auto simp add*: *mat-of-rows-def*)
also have ... = *Matrix.mat* (*dim-row A*) (*dim-col A*)
(λ(*i*,*k*). *if i = a* ∧ *k* ∈ *set* (*x # xs*) *then if k = 0 then if D dvd A*$$(*i*,*k*)
then D else A$$(*i*,*k*) *else A*$$(*i*,*k*) *gmod D else A*$$(*i*,*k*)) (**is** *?lhs = ?rhs*)
proof (*rule eq-matI*)
show *dim-row ?lhs = dim-row ?rhs* **and** *dim-col ?lhs = dim-col ?rhs* **by** *auto*
fix *i j* **assume** *i*: *i*<*dim-row ?rhs* **and** *j*: *j* < *dim-col ?rhs*
have *jn*: *j*<*n* **using** *j 2.prems* **by** (*simp add*: *append-rows-def*)
have *xn*: *x* < *n*

```
      by (simp add: 2.prems(5))
    show ?lhs $$ (i,j) = ?rhs $$ (i,j)
    proof (cases i=a ∧ j ∈ set xs)
      case True note ia-jxs = True
      have j-not-x: j≠x using d True by auto
      show ?thesis
      proof (cases j=0 ∧ D dvd ?reduce-xs $$(i,j))
        case True
        have ?lhs $$ (i,j) = D
          using True i j ia-jxs by auto
        also have ... = ?rhs $$ (i,j) using i j j-not-x
          by (metis 2.prems(8) True ia-jxs list.set-intros(2))
        finally show ?thesis .
      next
        case False
        show ?thesis
       by (smt (verit) 2 2.prems(8) dim-col-mat(1) dim-row-mat(1) i index-mat(1)
insert-iff j j-not-x list.set(2) old.prod.case)
    qed
  next
      case False
      show ?thesis using 2 i j xn
          by (smt (verit) 2.prems(8) False carrier-matD(2) dim-row-mat(1) in-
dex-mat(1)
        insert-iff jn list.set(2) old.prod.case reduce-element-mod-D-preserves-dimensions(2)
reduce-xs-carrier)
    qed
  qed
  finally show ?case using 1 by simp
qed
```

```
lemma reduce-row-mod-D-abs-case-m'':
  assumes A-def: A = A' @_r B and B ∈ carrier-mat n n
    and A': A' ∈ carrier-mat m n and a ≤ m
    and j: ∀j∈set xs. j<n ∧ (B $$ (j, j) = D) ∧ (∀j'∈{0..<n}−{j}. B $$ (j, j')
= 0)
    and d: distinct xs and m≥n and 0 ∉ set xs
    and D > 0
  shows reduce-row-mod-D-abs A a xs D m = Matrix.mat (dim-row A) (dim-col
A)
          (λ(i,k). if i = a ∧ k ∈ set xs then if k = 0 ∧ D dvd A$$(i,k) then D
                else A$$(i,k) gmod D else A$$(i,k))
  using assms
proof (induct A a xs D m arbitrary: A' B rule: reduce-row-mod-D-abs.induct)
  case (1 A a D m)
```

then show *?case* **by** *force*
next
case (*2 A a x xs D m*)
note *A-A'B = 2.prems(1)*
note *B = 2.prems(2)*
note *A' = 2.prems(3)*
note *a = 2.prems(4)*
note *j = 2.prems(5)*
note *mn = 2.prems(7)*
note *d = 2.prems(6)*
note *zero-not-xs = 2.prems(8)*
let *?reduce-xs = (reduce-element-mod-D-abs A a x D m)*
have *reduce-xs-carrier*: *?reduce-xs ∈ carrier-mat (m + n) n*
by (*metis 2.prems(1) 2.prems(2) 2.prems(3) add.right-neutral append-rows-def*

carrier-matD carrier-mat-triv index-mat-four-block(2,3) index-zero-mat(2,3)
reduce-element-mod-D-preserves-dimensions)
have *A*: *A:carrier-mat (m+n) n* **using** *A' B A-A'B* **by** *blast*
have *1*: *reduce-row-mod-D-abs A a (x # xs) D m*
= reduce-row-mod-D-abs ?reduce-xs a xs D m **by** *simp*
have *2*: *reduce-element-mod-D-abs A a j D m = Matrix.mat (dim-row A) (dim-col A)*
(λ(i,k). if i = a ∧ k = j then if j = 0 ∧ D dvd A$$(i,k)
then D else A$$(i,k) gmod D else A$$(i,k)) **if** *j∈set (x#xs)* **for** *j*
by (*rule reduce-element-mod-D-abs-case-m'[OF A-A'B B A']*, *insert 2.prems that*, *auto*)
have *reduce-row-mod-D-abs ?reduce-xs a xs D m =*
Matrix.mat (dim-row ?reduce-xs) (dim-col ?reduce-xs) (λ(i,k). if i = a ∧ k ∈ set xs
then if k=0 ∧ D dvd ?reduce-xs $$ (i, k) then D
else ?reduce-xs $$ (i, k) gmod D else ?reduce-xs $$ (i, k))
proof (*rule 2.hyps*[*OF - - - a - - mn*])
let *?A' = mat-of-rows n [Matrix.row (reduce-element-mod-D-abs A a x D m) i. i ← [0..<m]]*
define *B'* **where** *B' = mat-of-rows n [Matrix.row ?reduce-xs i. i ← [m..<dim-row A]]*
show *A''*: *?A' : carrier-mat m n* **by** *auto*
show *B'*: *B' : carrier-mat n n* **unfolding** *B'-def* **using** *mn A* **by** *auto*
show *reduce-split*: *?reduce-xs = ?A' @$_r$ B'*
by (*metis B'-def append-rows-split carrier-matD*
reduce-element-mod-D-preserves-dimensions(3) reduce-xs-carrier le-add1)
show *∀j∈set xs. j<n ∧ (B' $$ (j, j) = D) ∧ (∀j'∈{0..<n}−{j}. B' $$ (j, j') = 0)*
proof
fix *j* **assume** *j-xs*: *j∈set xs*
have *B $$ (j,j') = B' $$ (j,j')* **if** *j'*: *j'<n* **for** *j'*
proof −
have *B $$ (j,j') = A $$ (m+j,j')*
by (*smt (verit) A-A'B A A' Groups.add-ac(2) j-xs add-diff-cancel-left'*

```
append-rows-def carrier-matD j'
        index-mat-four-block(1) index-mat-four-block(2,3) insert-iff j less-diff-conv
list.set(2) not-add-less1)
      also have ... = ?reduce-xs $$ (m+j,j')
          by (smt (verit, ccfv-threshold) A'' diff-add-zero index-mat-addrow(3)
neq0-conv
        a j zero-not-xs A add.commute add-diff-cancel-left' reduce-element-mod-D-abs-def
        cancel-comm-monoid-add-class.diff-cancel carrier-matD index-mat-addrow(1)
j'
                  j-xs le-eq-less-or-eq less-diff-conv less-not-refl2 list.set-intros(2)
nat-SN.compat)
      also have ... = B'$$ (j,j')
     by (smt (verit) B A A' A-A'B B' A'' reduce-split add.commute add-diff-cancel-left'
j' not-add-less1
          append-rows-def carrier-matD index-mat-four-block j j-xs less-diff-conv
list.set-intros(2))
      finally show ?thesis .
    qed
    thus j < n ∧ B' $$ (j, j) = D ∧ (∀ j'∈{0..<n} − {j}. B' $$ (j, j') = 0)
using j
      by (metis Diff-iff atLeastLessThan-iff insert-iff j-xs list.simps(15))
  qed
 qed (insert 2.prems, auto simp add: mat-of-rows-def)
 also have ... = Matrix.mat (dim-row A) (dim-col A)
        (λ(i,k). if i = a ∧ k ∈ set (x # xs) then if k = 0 then if D dvd A$$(i,k)
        then D else A$$(i,k) else A$$(i,k) gmod D else A$$(i,k)) (is ?lhs = ?rhs)
 proof (rule eq-matI)
  show dim-row ?lhs = dim-row ?rhs and dim-col ?lhs = dim-col ?rhs by auto
  fix i j assume i: i<dim-row ?rhs and j: j < dim-col ?rhs
  have jn: j<n using j 2.prems by (simp add: append-rows-def)
  have xn: x < n
   by (simp add: 2.prems(5))
  show ?lhs $$ (i,j) = ?rhs $$ (i,j)
  proof (cases i=a ∧ j ∈ set xs)
   case True note ia-jxs = True
   have j-not-x: j≠x using d True by auto
   show ?thesis
   proof (cases j=0 ∧ D dvd ?reduce-xs $$(i,j))
    case True
    have ?lhs $$ (i,j) = D
     using True i j ia-jxs by auto
    also have ... = ?rhs $$ (i,j) using i j j-not-x
     by (metis 2.prems(8) True ia-jxs list.set-intros(2))
    finally show ?thesis .
   next
    case False
    show ?thesis
   by (smt (verit) 2 2.prems(8) dim-col-mat(1) dim-row-mat(1) i index-mat(1)
insert-iff j j-not-x list.set(2) old.prod.case)
```

```
    qed
  next
    case False
    show ?thesis using 2 i j xn
      by (smt (verit) 2.prems(8) False carrier-matD(2) dim-row-mat(1) in-
dex-mat(1)
      insert-iff jn list.set(2) old.prod.case reduce-element-mod-D-preserves-dimensions(4)
reduce-xs-carrier)
    qed
  qed
  finally show ?case using 1
    by (smt (verit, ccfv-SIG) 2.prems(8) cong-mat split-conv)
qed



lemma
  assumes A-def: A = A' @_r B and B: B ∈ carrier-mat n n
  and A': A' ∈ carrier-mat m n and a: a≤m and j: j<n and mn: m≥n and j0:
j≠0
shows reduce-element-mod-D-invertible-mat-case-m':
  ∃ P. P ∈ carrier-mat (m+n) (m+n) ∧ invertible-mat P ∧ reduce-element-mod-D
A a j D m = P * A (is ?thesis1)
  and reduce-element-mod-D-abs-invertible-mat-case-m':
 ∃ P. P ∈ carrier-mat (m+n) (m+n) ∧ invertible-mat P ∧ reduce-element-mod-D-abs
A a j D m = P * A (is ?thesis2)
proof −
  let ?P = addrow-mat (m+n) (− (A $$ (a, j) gdiv D)) a (j + m)
  have jm: j+m ≠a using j0 a by auto
  have A: A ∈ carrier-mat (m + n) n using A-def A' B mn by auto
  have rw: reduce-element-mod-D A a j D m = reduce-element-mod-D-abs A a j D
m
    unfolding reduce-element-mod-D-def reduce-element-mod-D-abs-def using j0
by auto
  have reduce-element-mod-D A a j D m =  addrow (− (A $$ (a, j) gdiv D)) a (j
+ m) A
    unfolding reduce-element-mod-D-def using j0 by auto
  also have ... = ?P * A by (rule addrow-mat[OF A], insert j mn, auto)
  finally have reduce-element-mod-D A a j D m = ?P * A .
  moreover have ?P ∈ carrier-mat (m+n) (m+n) by simp
  moreover have invertible-mat ?P
    by (metis addrow-mat-carrier det-addrow-mat dvd-mult-right jm
      invertible-iff-is-unit-JNF mult.right-neutral semiring-gcd-class.gcd-dvd1)
  ultimately show ?thesis1 and ?thesis2 using rw by metis+
qed



lemma reduce-row-mod-D-invertible-mat-case-m':
  assumes A-def: A = A' @_r B and B ∈ carrier-mat n n
```

and A': $A' \in$ *carrier-mat m n* **and** a: $a \leq m$
and j: $\forall j{\in}$*set xs*. $j{<}n \wedge (B\ \$\$\ (j,\ j) = D) \wedge (\forall j'{\in}\{0..{<}n\}{-}\{j\}.\ B\ \$\$\ (j,\ j') = 0)$
and d: *distinct xs* **and** mn: $m{\geq}n$ **and** $0 \notin$ *set xs*
shows $\exists P.\ P \in$ *carrier-mat* $(m{+}n)\ (m{+}n) \wedge$ *invertible-mat* $P \wedge$
reduce-row-mod-D A a xs D m $= P * A$
using *assms*
proof (*induct A a xs D m arbitrary*: A' B *rule*: *reduce-row-mod-D.induct*)
case (*1 A a D m*)
show *?case* **by** (*rule exI*[*of - 1*$_m$ $(m{+}n)$], *insert 1.prems*, *auto simp add*: *append-rows-def*)
next
case (*2 A a x xs D m*)
note *A-A'B = 2.prems*(*1*)
note *B = 2.prems*(*2*)
note *A' = 2.prems*(*3*)
note *a = 2.prems*(*4*)
note *j = 2.prems*(*5*)
note *mn = 2.prems*(*7*)
note *d = 2.prems*(*6*)
note *zero-not-xs = 2.prems*(*8*)
let *?reduce-xs* = (*reduce-element-mod-D A a x D m*)
have *reduce-xs-carrier*: *?reduce-xs* $\in$ *carrier-mat* $(m + n)\ n$
by (*metis 2.prems*(*1*) *2.prems*(*2*) *2.prems*(*3*) *add.right-neutral append-rows-def carrier-matD carrier-mat-triv index-mat-four-block*(*2*,*3*) *index-zero-mat*(*2*,*3*) *reduce-element-mod-D-preserves-dimensions*)
have *A*: *A*:*carrier-mat* $(m{+}n)\ n$ **using** A' B *A-A'B* **by** *blast*
let *?reduce-xs* = (*reduce-element-mod-D A a x D m*)
have *1*: *reduce-row-mod-D A a* $(x \#\ xs)$ *D m*
$=$ *reduce-row-mod-D ?reduce-xs a xs D m* **by** *simp*
have $\exists P.\ P \in$ *carrier-mat* $(m{+}n)\ (m{+}n) \wedge$ *invertible-mat* $P \wedge$
reduce-element-mod-D A a x D m $= P * A$
by (*rule reduce-element-mod-D-invertible-mat-case-m'*[*OF A-A'B B A' a - mn*], *insert zero-not-xs j*, *auto*)
from *this* **obtain** P **where** P: $P \in$ *carrier-mat* $(m{+}n)\ (m{+}n)$ **and** *inv-P*: *invertible-mat P*
and *R-P*: *reduce-element-mod-D A a x D m* $= P * A$ **by** *auto*
have $\exists P.\ P \in$ *carrier-mat* $(m + n)\ (m + n) \wedge$ *invertible-mat P*
$\wedge$ *reduce-row-mod-D ?reduce-xs a xs D m* $= P *$ *?reduce-xs*
proof (*rule 2.hyps*)
let *?A'* = *mat-of-rows n* [*Matrix.row ?reduce-xs i*. $i \leftarrow [0..{<}m]$]
let *?B'* = *mat-of-rows n* [*Matrix.row ?reduce-xs i*. $i \leftarrow [m..{<}m{+}n]$]
show B': *?B'* $\in$ *carrier-mat n n* **by** *auto*
show A'': *?A'* : *carrier-mat m n* **by** *auto*
show *reduce-split*: *?reduce-xs* = *?A'* $@_r$ *?B'*
by (*smt* (*verit*) *2*(*2*) *2*(*4*) *P R-P add.comm-neutral append-rows-def append-rows-split carrier-matD index-mat-four-block*(*3*) *index-mult-mat*(*2*) *index-zero-mat*(*3*) *le-add1*

reduce-element-mod-D-preserves-dimensions(*2*))
show ∀*j*∈*set xs*. *j* < *n* ∧ *?B′* $$ (*j*, *j*) = *D* ∧ (∀*j′*∈{*0*..<*n*} − {*j*}. *?B′* $$ (*j*, *j′*) = *0*)
proof
fix *j* **assume** *j-xs*: *j*∈*set xs*
have *B* $$ (*j*,*j′*) = *?B′* $$ (*j*,*j′*) **if** *j′*: *j′*<*n* **for** *j′*
proof −
have *B* $$ (*j*,*j′*) = *A* $$ (*m*+*j*,*j′*)
by (*smt* (*verit*) *A-A′B A A′ Groups.add-ac*(*2*) *j-xs add-diff-cancel-left′ append-rows-def carrier-matD j′*
index-mat-four-block(*1*) *index-mat-four-block*(*2*,*3*) *insert-iff j less-diff-conv list.set*(*2*) *not-add-less1*)
also have ... = *?reduce-xs* $$ (*m*+*j*,*j′*)
by (*smt* (*verit*, *ccfv-SIG*) *not-add-less1*
a j zero-not-xs A add.commute add-diff-cancel-left′ reduce-element-mod-D-def
cancel-comm-monoid-add-class.diff-cancel carrier-matD index-mat-addrow(*1*) *j′*
j-xs le-eq-less-or-eq less-diff-conv less-not-refl2 list.set-intros(*2*) *nat-SN.compat*)
also have ... = *?B′*$$ (*j*,*j′*)
by (*smt* (*verit*) *B A A′ A-A′B B′ A″ reduce-split add.commute add-diff-cancel-left′ j′ not-add-less1*
append-rows-def carrier-matD index-mat-four-block j j-xs less-diff-conv list.set-intros(*2*))
finally show *?thesis* **.**
qed
thus *j* < *n* ∧ *?B′* $$ (*j*, *j*) = *D* ∧ (∀*j′*∈{*0*..<*n*} − {*j*}. *?B′* $$ (*j*, *j′*) = *0*) **using** *j*
by (*metis Diff-iff atLeastLessThan-iff insert-iff j-xs list.simps*(*15*))
qed
qed (*insert d zero-not-xs a mn*, *auto*)
from *this* **obtain** *P2* **where** *P2*: *P2* ∈ *carrier-mat* (*m* + *n*) (*m* + *n*) **and** *inv-P2*: *invertible-mat P2*
and *R-P2*: *reduce-row-mod-D ?reduce-xs a xs D m* = *P2* ∗ *?reduce-xs*
by *auto*
have *invertible-mat* (*P2* ∗ *P*) **using** *P P2 inv-P inv-P2 invertible-mult-JNF* **by** *blast*
moreover have (*P2* ∗ *P*) ∈ *carrier-mat* (*m*+*n*) (*m*+*n*) **using** *P2 P* **by** *auto*
moreover have *reduce-row-mod-D A a* (*x* # *xs*) *D m* = (*P2* ∗ *P*) ∗ *A*
by (*smt* (*verit*) *P P2 R-P R-P2 1 assoc-mult-mat carrier-matD carrier-mat-triv*
index-mult-mat reduce-row-mod-D-preserves-dimensions)
ultimately show *?case* **by** *blast*
qed

lemma *reduce-row-mod-D-abs-invertible-mat-case-m′*:
assumes *A-def*: *A* = *A′* @$_r$ *B* **and** *B* ∈ *carrier-mat n n*
and *A′*: *A′* ∈ *carrier-mat m n* **and** *a*: *a* ≤ *m*

and j: $\forall j{\in}set\ xs.\ j{<}n \wedge (B\ \$\$\ (j,\ j) = D) \wedge (\forall j'{\in}\{0..{<}n\}{-}\{j\}.\ B\ \$\$\ (j,\ j') = 0)$
and d: *distinct xs* **and** mn: $m{\geq}n$ **and** $0{\notin}\ set\ xs$
shows $\exists P.\ P \in carrier\text{-}mat\ (m{+}n)\ (m{+}n) \wedge invertible\text{-}mat\ P \wedge$
reduce-row-mod-D-abs A a xs D m $= P * A$
using *assms*
proof (*induct A a xs D m arbitrary*: A' B *rule*: *reduce-row-mod-D-abs.induct*)
case (*1 A a D m*)
show *?case* **by** (*rule exI*[*of* - 1_m $(m{+}n)$], *insert 1.prems*, *auto simp add*: *append-rows-def*)
next
case (*2 A a x xs D m*)
note *A-A'B = 2.prems*(*1*)
note *B = 2.prems*(*2*)
note *A' = 2.prems*(*3*)
note *a = 2.prems*(*4*)
note *j = 2.prems*(*5*)
note *mn = 2.prems*(*7*)
note *d = 2.prems*(*6*)
note *zero-not-xs = 2.prems*(*8*)
let *?reduce-xs* = (*reduce-element-mod-D-abs A a x D m*)
have *reduce-xs-carrier*: *?reduce-xs* $\in$ *carrier-mat* $(m + n)$ n
by (*metis 2.prems*(*1*) *2.prems*(*2*) *2.prems*(*3*) *add.right-neutral append-rows-def*
carrier-matD carrier-mat-triv index-mat-four-block(*2*,*3*) *index-zero-mat*(*2*,*3*)
reduce-element-mod-D-preserves-dimensions)
have *A*: *A*:*carrier-mat* $(m{+}n)$ n **using** A' B *A-A'B* **by** *blast*
let *?reduce-xs* = (*reduce-element-mod-D-abs A a x D m*)
have *1*: *reduce-row-mod-D-abs A a* $(x\ \#\ xs)$ *D m*
$=$ *reduce-row-mod-D-abs ?reduce-xs a xs D m* **by** *simp*
have $\exists P.\ P \in carrier\text{-}mat\ (m{+}n)\ (m{+}n) \wedge invertible\text{-}mat\ P \wedge$
reduce-element-mod-D-abs A a x D m $= P * A$
by (*rule reduce-element-mod-D-abs-invertible-mat-case-m'*[*OF A-A'B B A' a - mn*],
insert zero-not-xs j, *auto*)
from *this* **obtain** P **where** P: $P \in carrier\text{-}mat\ (m{+}n)\ (m{+}n)$ **and** *inv-P*: *invertible-mat P*
and *R-P*: *reduce-element-mod-D-abs A a x D m* $= P * A$ **by** *auto*
have $\exists P.\ P \in carrier\text{-}mat\ (m + n)\ (m + n) \wedge invertible\text{-}mat\ P$
$\wedge$ *reduce-row-mod-D-abs ?reduce-xs a xs D m* $= P *$ *?reduce-xs*
proof (*rule 2.hyps*)
let *?A'* = *mat-of-rows n* [*Matrix.row ?reduce-xs i*. $i \leftarrow [0..{<}m]$]
let *?B'* = *mat-of-rows n* [*Matrix.row ?reduce-xs i*. $i \leftarrow [m..{<}m{+}n]$]
show *B'*: *?B'* $\in$ *carrier-mat n n* **by** *auto*
show *A''*: *?A'* : *carrier-mat m n* **by** *auto*
show *reduce-split*: *?reduce-xs* = *?A'* $@_r$ *?B'*
by (*smt* (*verit*) *2*(*2*) *2*(*4*) *P R-P add.comm-neutral append-rows-def append-rows-split carrier-matD*
index-mat-four-block(*3*) *index-mult-mat*(*2*) *index-zero-mat*(*3*) *le-add1*

reduce-element-mod-D-preserves-dimensions(*4*))
show $\forall j \in$ *set xs*. $j < n \wedge$ *?B'* $$ $(j, j) = D \wedge (\forall j' \in \{0..<n\} - \{j\}$. *?B'* $$ $(j, j') = 0)$
proof
fix *j* **assume** *j-xs*: $j \in$ *set xs*
have *B* $$ $(j,j') =$ *?B'* $$ (j,j') **if** *j'*: $j'<n$ **for** *j'*
proof −
have *B* $$ $(j,j') =$ *A* $$ $(m+j,j')$
by (*smt* (*verit*) *A-A'B A A' Groups.add-ac*(*2*) *j-xs add-diff-cancel-left' append-rows-def carrier-matD j'*
index-mat-four-block(*1*) *index-mat-four-block*(*2*,*3*) *insert-iff j less-diff-conv list.set*(*2*) *not-add-less1*)
also have ... = *?reduce-xs* $$ $(m+j,j')$
by (*smt* (*verit*, *ccfv-SIG*) *not-add-less1*
a j zero-not-xs A add.commute add-diff-cancel-left' reduce-element-mod-D-abs-def
cancel-comm-monoid-add-class.diff-cancel carrier-matD index-mat-addrow(*1*) *j'*
j-xs le-eq-less-or-eq less-diff-conv less-not-refl2 list.set-intros(*2*) *nat-SN.compat*)
also have ... = *?B'*$$ (j,j')
by (*smt* (*verit*) *B A A' A-A'B B' A'' reduce-split add.commute add-diff-cancel-left' j' not-add-less1*
append-rows-def carrier-matD index-mat-four-block j j-xs less-diff-conv list.set-intros(*2*))
finally show *?thesis* **.**
qed
thus $j < n \wedge$ *?B'* $$ $(j, j) = D \wedge (\forall j' \in \{0..<n\} - \{j\}$. *?B'* $$ $(j, j') = 0)$ **using** *j*
by (*metis Diff-iff atLeastLessThan-iff insert-iff j-xs list.simps*(*15*))
qed
qed (*insert d zero-not-xs a mn*, *auto*)
from *this* **obtain** *P2* **where** *P2*: $P2 \in$ *carrier-mat* $(m + n)$ $(m + n)$ **and** *inv-P2*: *invertible-mat P2*
and *R-P2*: *reduce-row-mod-D-abs ?reduce-xs a xs D m* = *P2* $*$ *?reduce-xs*
by *auto*
have *invertible-mat* $(P2 * P)$ **using** *P P2 inv-P inv-P2 invertible-mult-JNF* **by** *blast*
moreover have $(P2 * P) \in$ *carrier-mat* $(m+n)$ $(m+n)$ **using** *P2 P* **by** *auto*
moreover have *reduce-row-mod-D-abs A a* $(x \# xs)$ *D m* = $(P2 * P) * A$
by (*smt* (*verit*) *P P2 R-P R-P2 1 assoc-mult-mat carrier-matD carrier-mat-triv*
index-mult-mat reduce-row-mod-D-preserves-dimensions-abs)
ultimately show *?case* **by** *blast*
qed

lemma *reduce-invertible-mat-case-m*:
assumes *A'*: $A' \in$ *carrier-mat m n* **and** *B*: $B \in$ *carrier-mat n n*
and *a*: $a<m$ **and** *ab*: $a \neq m$
and *A-def*: $A = A' \mathbin{@_r} B$

and *j*: $\forall j{\in}set\ xs.\ j{<}n \wedge (B\ \$\$\ (j,\ j) = D) \wedge (\forall j'{\in}\{0..{<}n\}{-}\{j\}.\ B\ \$\$\ (j,\ j') = 0)$
and *Aaj*: $A\ \$\$\ (a,0) \neq 0$
and *mn*: $m{\geq}n$
and *n0*: $0{<}n$
and *pquvd*: $(p,q,u,v,d) = euclid\text{-}ext2\ (A\$\$(a,0))\ (A\$\$(m,0))$
and *A2-def*: $A2 = Matrix.mat\ (dim\text{-}row\ A)\ (dim\text{-}col\ A)$
$(\lambda(i,k).\ if\ i = a\ then\ (p{*}A\$\$(a,k) + q{*}A\$\$(m,k))$
$else\ if\ i = m\ then\ u * A\$\$(a,k) + v * A\$\$(m,k)$
$else\ A\$\$(i,k)$
$)$
and *xs-def*: $xs = [1..{<}n]$
and *ys-def*: $ys = [1..{<}n]$
and *j-ys*: $\forall j{\in}set\ ys.\ j{<}n \wedge (B\ \$\$\ (j,\ j) = D) \wedge (\forall j'{\in}\{0..{<}n\}{-}\{j\}.\ B\ \$\$\ (j,\ j') = 0)$
and *D0*: $D > 0$
and *Am0-D*: $A\ \$\$\ (m,\ 0) \in \{0,D\}$
and *Am0-D2*: $A\ \$\$\ (m,\ 0) = 0 \longrightarrow A\ \$\$\ (a,\ 0) = D$
shows $\exists P.\ invertible\text{-}mat\ P \wedge P \in carrier\text{-}mat\ (m{+}n)\ (m{+}n) \wedge (reduce\ a\ m\ D\ A) = P * A$
proof −
let *?A* = $Matrix.mat\ (dim\text{-}row\ A)\ (dim\text{-}col\ A)$
$(\lambda(i,k).\ if\ i = a\ then\ (p{*}A\$\$(a,k) + q{*}A\$\$(m,k))$
$else\ if\ i = m\ then\ u * A\$\$(a,k) + v * A\$\$(m,k)$
$else\ A\$\$(i,k)$
$)$
have *D*: $D \cdot_m 1_m\ n \in carrier\text{-}mat\ n\ n$ **using** *mn* **by** *auto*
have *A*: $A \in carrier\text{-}mat\ (m{+}n)\ n$ **using** *A-def A′ B mn* **by** *simp*
hence *A-carrier*: $?A \in carrier\text{-}mat\ (m{+}n)\ n$ **by** *auto*

let *?BM* = *bezout-matrix-JNF A a m 0 euclid-ext2*

have *A′-BZ-A*: *?A* = *?BM* ∗ *A*
by (*rule bezout-matrix-JNF-mult-eq*[*OF A′ - - ab A-def B pquvd*], *insert a*, *auto*)

have *invertible-bezout*: *invertible-mat ?BM*
by (*rule invertible-bezout-matrix-JNF*[*OF A is-bezout-ext-euclid-ext2 a - - Aaj*], *insert a n0*, *auto*)
have *BM*: *?BM* ∈ *carrier-mat* (*m*+*n*) (*m*+*n*) **unfolding** *bezout-matrix-JNF-def* **using** *A* **by** *auto*
let *?reduce-a* = *reduce-row-mod-D ?A a xs D m*
define *A′1* **where** *A′1* = *mat-of-rows n* [*Matrix.row ?A i*. *i* ← [0..<*m*]]
define *A′2* **where** *A′2* = *mat-of-rows n* [*Matrix.row ?A i*. *i* ← [*m*..<*dim-row A*]]
have *A-A′-D*: *?A* = *A′1* @$_r$ *A′2* **using** *append-rows-split A*
by (*metis* (*no-types*, *lifting*) *A′1-def A′2-def A-carrier carrier-matD le-add1*)
have *j-A′1-A′2*: $\forall j{\in}set\ xs.\ j < n \wedge A'2\ \$\$\ (j,\ j) = D \wedge (\forall j'{\in}\{0..{<}n\} - \{j\}.\ A'2\ \$\$\ (j,\ j') = 0)$
proof (*rule ballI*)

fix *ja* **assume** *ja*: *ja*∈*set xs*
have *ja-n*: *ja* < *n* **using** *ja* **unfolding** *xs-def* **by** *auto*
have *ja2*: *ja* < *dim-row A* − *m* **using** *A mn ja-n* **by** *auto*
have *ja-m*: *ja* < *m* **using** *ja-n mn* **by** *auto*
have *ja-not-0*: *ja* ≠ *0* **using** *ja* **unfolding** *xs-def* **by** *auto*
show *ja* < *n* ∧ *A'2* $$ (*ja*, *ja*) = *D* ∧ (∀ *j'*∈{*0*..<*n*} − {*ja*}. *A'2* $$ (*ja*, *j'*) = *0*)
proof −
have *A'2* $$ (*ja*, *ja*) = [*Matrix.row ?A i*. *i* ← [*m*..<*dim-row A*]] ! *ja* $*v ja*
by (*metis* (*no-types*, *lifting*) *A A'2-def add-diff-cancel-left' carrier-matD*(*1*)
ja-n length-map length-upt mat-of-rows-index)
also have ... = *?A* $$ (*m* + *ja*, *ja*) **using** *A mn ja-n* **by** *auto*
also have ... = *A* $$ (*m*+*ja*, *ja*) **using** *A a mn ja-n ja-not-0* **by** *auto*
also have ... = (*A'* @$_r$ *B*) $$ (*m* + *ja*, *ja*) **unfolding** *A-def* **..**
also have ... = *B* $$ (*ja*, *ja*)
by (*metis B Groups.add-ac*(*2*) *append-rows-nth2 assms*(*1*) *ja-n mn nat-SN.compat*)
also have ... = *D* **using** *j ja* **by** *blast*
finally have *A2-D*: *A'2* $$ (*ja*, *ja*) = *D* **.**

moreover have (∀ *j'*∈{*0*..<*n*} − {*ja*}. *A'2* $$ (*ja*, *j'*) = *0*)
proof (*rule ballI*)
fix *j'* **assume** *j'*: *j'*: {*0*..<*n*} − {*ja*}
have *A'2* $$ (*ja*, *j'*) = [*Matrix.row ?A i*. *i* ← [*m*..<*dim-row A*]] ! *ja* $*v j'*
unfolding *A'2-def* **by** (*rule mat-of-rows-index*, *insert j' ja-n ja2*, *auto*)
also have ... = *?A* $$ (*m* + *ja*, *j'*) **using** *A mn ja-n j'* **by** *auto*
also have ... = *A* $$ (*m*+*ja*, *j'*) **using** *A a mn ja-n ja-not-0 j'* **by** *auto*
also have ... = (*A'* @$_r$ *B*) $$ (*ja* + *m*, *j'*) **unfolding** *A-def*
by (*simp add*: *add.commute*)
also have ... = *B* $$ (*ja*, *j'*)
by (*rule append-rows-nth2*[*OF A' B - ja-m ja-n*], *insert j'*, *auto*)
also have ... = *0* **using** *mn j' ja-n j ja* **by** *auto*
finally show *A'2* $$ (*ja*, *j'*) = *0* **.**
qed
ultimately show *?thesis* **using** *ja-n* **by** *simp*
qed
qed
have *reduce-a-eq*: *?reduce-a* = *Matrix.mat* (*dim-row ?A*) (*dim-col ?A*)
(λ(*i*, *k*). *if i* = *a* ∧ *k* ∈ *set xs then if k* = *0 then if D dvd ?A* $$ (*i*, *k*) *then D else ?A* $$ (*i*, *k*) *else ?A* $$ (*i*, *k*) *gmod D else ?A* $$ (*i*, *k*))
proof (*rule reduce-row-mod-D-case-m'*[*OF A-A'-D - - a j-A'1-A'2 - mn D0*])
show *A'2* ∈ *carrier-mat n n* **using** *A A'2-def* **by** *auto*
show *A'1* ∈ *carrier-mat m n* **by** (*simp add*: *A'1-def mat-of-rows-def*)
show *distinct xs* **using** *distinct-filter distinct-upt xs-def* **by** *blast*
qed
have *reduce-a*: *?reduce-a* ∈ *carrier-mat* (*m*+*n*) *n* **using** *reduce-a-eq A* **by** *auto*
have ∃ *P*. *P* ∈ *carrier-mat* (*m* + *n*) (*m* + *n*) ∧ *invertible-mat P* ∧ *?reduce-a* = *P* ∗ *?A*

by (*rule reduce-row-mod-D-invertible-mat-case-m*[*OF A-A′-D - - - j-A′1-A′2 mn*],
insert a A A′2-def A′1-def, *auto*)
from *this* **obtain** *P* **where** *P*: *P* ∈ *carrier-mat* (*m* + *n*) (*m* + *n*) **and** *inv-P*: *invertible-mat P*
and *reduce-a-PA*: *?reduce-a* = *P* ∗ *?A* **by** *blast*
let *?reduce-b* = *reduce-row-mod-D ?reduce-a m ys D m*
let *?B′* = *mat-of-rows n* [*Matrix.row ?reduce-a i*. *i* ← [*0*..<*m*]]
define *reduce-a1* **where** *reduce-a1* = *mat-of-rows* (*dim-col ?reduce-a*) [*Matrix.row ?reduce-a i*. *i* ← [*0*..<*m*]]
define *reduce-a2* **where** *reduce-a2* = *mat-of-rows* (*dim-col ?reduce-a*) [*Matrix.row ?reduce-a i*. *i* ← [*m*..<*dim-row ?reduce-a*]]
have *reduce-a-split*: *?reduce-a* = *reduce-a1* @$_r$ *reduce-a2*
by (*unfold reduce-a1-def reduce-a2-def*, *rule append-rows-split*, *insert mn A*, *auto*)
have *zero-notin-ys*: *0* ∉ *set ys*
proof −
have *m*: *m*<*dim-row A* **using** *A n0* **by** *auto*
have *?A* $$ (*m*,*0*) = *u* ∗ *A* $$ (*a*, *0*) + *v* ∗ *A* $$ (*m*, *0*) **using** *m n0 a A* **by** *auto*
also have ... = *0* **using** *pquvd*
by (*smt* (*verit*) *dvd-mult-div-cancel euclid-ext2-def euclid-ext2-works*(*3*) *more-arith-simps*(*11*) *mult.commute mult-minus-left prod.sel*(*1*) *prod.sel*(*2*) *semiring-gcd-class.gcd-dvd1*)
finally show *?thesis* **using** *D0* **unfolding** *ys-def* **by** *auto*
qed
have *reduce-a2*: *reduce-a2* ∈ *carrier-mat n n* **unfolding** *reduce-a2-def* **using** *A* **by** *auto*
have *reduce-a1*: *reduce-a1* ∈ *carrier-mat m n* **unfolding** *reduce-a1-def* **using** *A* **by** *auto*
have *j2*: ∀*j*∈*set ys*. *j* < *n* ∧ *reduce-a2* $$ (*j*, *j*) = *D* ∧ (∀*j′*∈{*0*..<*n*} − {*j*}. *reduce-a2* $$ (*j*, *j′*) = *0*)
proof
fix *j* **assume** *j-in-ys*: *j* ∈ *set ys*
have *a-jm*: *a* ≠ *j*+*m* **using** *a* **by** *auto*
have *m-not-jm*: *m* ≠ *j* + *m* **using** *zero-notin-ys j-in-ys* **by** *fastforce*
have *jm*: *j*+*m* < *dim-row ?A* **using** *A-carrier j-in-ys* **unfolding** *ys-def* **by** *auto*
have *jn*: *j* < *dim-col ?A* **using** *A-carrier j-in-ys* **unfolding** *ys-def* **by** *auto*
have *jm′*: *j*+*m* < *dim-row A* **using** *A-carrier j-in-ys* **unfolding** *ys-def* **by** *auto*
have *jn′*: *j* < *dim-col A* **using** *A-carrier j-in-ys* **unfolding** *ys-def* **by** *auto*
have *reduce-a2* $$ (*j*, *j′*) = *B* $$ (*j*,*j′*) **if** *j′*: *j′*<*n* **for** *j′*
proof −
have *reduce-a2* $$ (*j*, *j′*) = *?reduce-a* $$ (*j*+*m*,*j′*)
by (*rule append-rows-nth2*[*symmetric*, *OF reduce-a1 reduce-a2 reduce-a-split*], *insert j-in-ys mn j′*, *auto simp add*: *ys-def*)
also have ... = *?A* $$ (*j*+*m*, *j′*) **using** *reduce-a-eq jm jn a-jm j′ A-carrier* **by** *auto*
also have ... = *A* $$ (*j*+*m*, *j′*) **using** *a-jm m-not-jm jm′ jn′ j′ A-carrier* **by** *auto*

```
    also have ... = B $$ (j,j')
      using assms(1) assms(2) assms(5) assms(8,14) unfolding A-def
      by (meson append-rows-nth2 less-le-trans j' j-in-ys)
    finally show ?thesis .
  qed
  thus j < n ∧ reduce-a2 $$ (j, j) = D ∧ (∀ j'∈{0..<n} − {j}. reduce-a2 $$ (j,
j') = 0)
    using j-ys j-in-ys by auto
 qed
 have reduce-b-eq: ?reduce-b = Matrix.mat (dim-row ?reduce-a) (dim-col ?reduce-a)

  (λ(i, k). if i = m ∧ k ∈ set ys then if k = 0 then if D dvd ?reduce-a $$ (i, k)
then D
     else ?reduce-a $$ (i, k) else ?reduce-a $$ (i, k) gmod D else ?reduce-a $$ (i,
k))
  by (rule reduce-row-mod-D-case-m''[OF reduce-a-split reduce-a2 reduce-a1 - j2
- mn zero-notin-ys],
      insert D0, auto simp add: ys-def)
 have ∃ P. P ∈ carrier-mat (m + n) (m + n) ∧ invertible-mat P ∧ ?reduce-b =
P ∗ ?reduce-a
  by (rule reduce-row-mod-D-invertible-mat-case-m'[OF reduce-a-split reduce-a2
reduce-a1 - j2 - mn zero-notin-ys],
      auto simp add: ys-def)
 from this obtain Q where Q: Q ∈ carrier-mat (m + n) (m + n) and inv-Q:
invertible-mat Q
  and reduce-b-Q-reduce: ?reduce-b = Q ∗ ?reduce-a by blast
 have reduce-b-eq-reduce: ?reduce-b = (reduce a m D A)
 proof (rule eq-matI)
  show dr-eq: dim-row ?reduce-b = dim-row (reduce a m D A)
    and dc-eq: dim-col ?reduce-b = dim-col (reduce a m D A)
    using reduce-preserves-dimensions by auto
  fix i ja assume i: i<dim-row (reduce a m D A) and ja: ja< dim-col (reduce a
m D A)
  have im: i<m+n using A i reduce-preserves-dimensions(1) by auto
  have ja-n: ja<n using A ja reduce-preserves-dimensions(2) by auto
  show ?reduce-b $$ (i,ja) = (reduce a m D A) $$ (i,ja)
  proof (cases (i≠a ∧ i≠m))
    case True
    have ?reduce-b $$ (i,ja) = ?reduce-a $$ (i,ja) unfolding reduce-b-eq
      by (smt (verit) True dr-eq dc-eq i index-mat(1) ja prod.simps(2) re-
duce-row-mod-D-preserves-dimensions)
    also have ... = ?A $$ (i,ja)
      by (smt (verit) A True carrier-matD(2) dim-col-mat(1) dim-row-mat(1) i
index-mat(1) ja-n
        reduce-a-eq reduce-preserves-dimensions(1) split-conv)
    also have ... = A $$ (i,ja) using A True im ja-n by auto
   also have ... = (reduce a m D A) $$ (i,ja) unfolding reduce-alt-def-not0[OF
Aaj pquvd]
      using im ja-n A True by auto
```

```
    finally show ?thesis .
  next
    case False note a-or-b = False
    have gcd-pq: p * A $$ (a, 0) + q * A $$ (m, 0) = gcd (A $$ (a, 0)) (A $$
(m, 0))
      by (metis assms(10) euclid-ext2-works(1) euclid-ext2-works(2))
    have gcd-le-D: gcd (A $$ (a, 0)) (A $$ (m, 0)) ≤ D
      by (metis Am0-D D0 assms(17) empty-iff gcd-le1-int gcd-le2-int insert-iff)
    show ?thesis
    proof (cases i=a)
      case True note ia = True
      hence i-not-b: i≠m using ab by auto
        have 1: ?reduce-b $$ (i,ja) = ?reduce-a $$ (i,ja) unfolding reduce-b-eq

          by (smt (verit) ab dc-eq dim-row-mat(1) dr-eq i ia index-mat(1) ja
prod.simps(2)
            reduce-b-eq reduce-row-mod-D-preserves-dimensions(2))
      show ?thesis
      proof (cases ja=0)
        case True note ja0 = True
        hence ja-notin-xs: ja ∉ set xs unfolding xs-def by auto
        have ?reduce-a $$ (i,ja) = p * A $$ (a, 0) + q * A $$ (m, 0)
         unfolding reduce-a-eq using True ja0 ab a-or-b i-not-b ja-n im a A False
ja-notin-xs
          by auto
        also have ... = (reduce a m D A) $$ (i,ja)
          unfolding reduce-alt-def-not0[OF Aaj pquvd]
          using True a-or-b i-not-b ja-n im A False
          using gcd-le-D gcd-pq Am0-D Am0-D2 by auto
        finally show ?thesis using 1 by auto
      next
        case False
        hence ja-in-xs: ja ∈ set xs
          unfolding xs-def using True ja-n im a A unfolding set-filter by auto
        have ?reduce-a $$ (i,ja) = ?A $$ (i, ja) gmod D
          unfolding reduce-a-eq using True ab a-or-b i-not-b ja-n im a A ja-in-xs
False by auto
        also have ... = (reduce a m D A) $$ (i,ja)
          unfolding reduce-alt-def-not0[OF Aaj pquvd] using True a-or-b i-not-b
ja-n im A False by auto
        finally show ?thesis using 1 by simp
      qed
    next
      case False note i-not-a = False
      have i-drb: i<dim-row ?reduce-b
        and i-dra: i<dim-row ?reduce-a
        and ja-drb: ja < dim-col ?reduce-b
       and ja-dra: ja < dim-col ?reduce-a using i ja reduce-carrier[OF A] A ja-n
im by auto
```

```
      have ib: i=m using False a-or-b by auto
      show ?thesis
      proof (cases ja = 0)
        case True note ja0 = True
        have uv: u * A $$ (a, ja) + v * A $$ (m, ja) = 0
          unfolding euclid-ext2-works[OF pquvd[symmetric]] True
       by (smt (verit) euclid-ext2-works[OF pquvd[symmetric]] more-arith-simps(11)
mult.commute mult-minus-left)
        have ?reduce-b $$ (i,ja) = u * A $$ (a, ja) + v * A $$ (m, ja)
        by (smt (verit) A A-carrier True assms(4) carrier-matD i ib index-mat(1)
reduce-a-eq
                  ja-dra old.prod.case reduce-preserves-dimensions(1) zero-notin-ys
reduce-b-eq
              reduce-row-mod-D-preserves-dimensions)
        also have ... = 0 using uv by blast
        also have ... = (reduce a m D A) $$ (i,ja)
         unfolding reduce-alt-def-not0[OF Aaj pquvd] using True False a-or-b ib
ja-n im A
           using i-not-a uv by auto
        finally show ?thesis by auto
      next
        case False
        have ja-in-ys: ja ∈ set ys
          unfolding ys-def using False ib ja-n im a  A unfolding set-filter by
auto
        have ?reduce-b $$ (i,ja) = (if ja = 0 then if D dvd ?reduce-a$$(i,ja) then
D
                          else ?reduce-a $$ (i, ja) else ?reduce-a $$ (i, ja) gmod D)
          unfolding reduce-b-eq using i-not-a  ja ja-in-ys
          by (smt (verit) i-dra ja-dra a-or-b index-mat(1) prod.simps(2))
        also have ... = (if ja = 0 then if D dvd ?reduce-a$$(i,ja) then D
                      else ?A $$ (i, ja) else ?A $$ (i, ja) gmod D)
          unfolding reduce-a-eq using ab a-or-b ib False ja-n im a A ja-in-ys by
auto
        also have ... = (reduce a m D A) $$ (i,ja)
         unfolding reduce-alt-def-not0[OF Aaj pquvd] using  False a-or-b ib ja-n
im A
          using i-not-a  by auto
        finally show ?thesis .
      qed
    qed
  qed
  have r: ?reduce-a = (P*?BM) * A using A A'-BZ-A BM P reduce-a-PA by
auto
  have Q * P * ?BM : carrier-mat (m+n) (m+n) using P BM Q by auto
  moreover have invertible-mat (Q * P*?BM)
    using inv-P invertible-bezout BM P invertible-mult-JNF inv-Q Q by (metis
mult-carrier-mat)
```

moreover have (*reduce a m D A*) = (*Q* ∗ *P* ∗ *?BM*) ∗ *A* **using** *reduce-a-eq r reduce-b-eq-reduce*
by (*smt* (*verit*) *BM P Q assoc-mult-mat carrier-matD carrier-mat-triv dim-row-mat*(*1*) *index-mult-mat*(*2*,*3*) *reduce-b-Q-reduce*)
ultimately show *?thesis* **by** *auto*
qed

lemma *reduce-abs-invertible-mat-case-m*:
assumes *A′*: *A′* ∈ *carrier-mat m n* **and** *B*: *B* ∈ *carrier-mat n n*
and *a*: *a*<*m* **and** *ab*: *a* ≠ *m*
and *A-def*: *A* = *A′* @$_r$ *B*
and *j*: ∀ *j*∈*set xs*. *j*<*n* ∧ (*B* $$ (*j*, *j*) = *D*) ∧ (∀ *j′*∈{*0*..<*n*}−{*j*}. *B* $$ (*j*, *j′*) = *0*)
and *Aaj*: *A* $$ (*a*,*0*) ≠ *0*
and *mn*: *m*≥*n*
and *n0*: *0*<*n*
and *pquvd*: (*p*,*q*,*u*,*v*,*d*) = *euclid-ext2* (*A*$$(*a*,*0*)) (*A*$$(*m*,*0*))
and *A2-def*: *A2* = *Matrix.mat* (*dim-row A*) (*dim-col A*)
(λ(*i*,*k*). *if i* = *a then* (*p*∗*A*$$(*a*,*k*) + *q*∗*A*$$(*m*,*k*))
else if i = *m then u* ∗ *A*$$(*a*,*k*) + *v* ∗ *A*$$(*m*,*k*)
else A$$(*i*,*k*)
)
and *xs-def*: *xs* = *filter* (λ*i*. *abs* (*A2* $$ (*a*,*i*)) > *D*) [*0*..<*n*]
and *ys-def*: *ys* = *filter* (λ*i*. *abs* (*A2* $$ (*m*,*i*)) > *D*) [*0*..<*n*]
and *j-ys*: ∀ *j*∈*set ys*. *j*<*n* ∧ (*B* $$ (*j*, *j*) = *D*) ∧ (∀ *j′*∈{*0*..<*n*}−{*j*}. *B* $$ (*j*, *j′*) = *0*)
and *D0*: *D* > *0*
shows ∃ *P*. *invertible-mat P* ∧ *P* ∈ *carrier-mat* (*m*+*n*) (*m*+*n*) ∧ (*reduce-abs a m D A*) = *P* ∗ *A*
proof −
let *?A* = *Matrix.mat* (*dim-row A*) (*dim-col A*)
(λ(*i*,*k*). *if i* = *a then* (*p*∗*A*$$(*a*,*k*) + *q*∗*A*$$(*m*,*k*))
else if i = *m then u* ∗ *A*$$(*a*,*k*) + *v* ∗ *A*$$(*m*,*k*)
else A$$(*i*,*k*)
)
note *xs-def* = *xs-def*[*unfolded A2-def*]
note *ys-def* = *ys-def*[*unfolded A2-def*]
have *D*: *D* ·$_m$ *1*$_m$ *n* ∈ *carrier-mat n n* **using** *mn* **by** *auto*
have *A*: *A* ∈ *carrier-mat* (*m*+*n*) *n* **using** *A-def A′ B mn* **by** *simp*
hence *A-carrier*: *?A* ∈ *carrier-mat* (*m*+*n*) *n* **by** *auto*

let *?BM* = *bezout-matrix-JNF A a m 0 euclid-ext2*

have *A′-BZ-A*: *?A* = *?BM* ∗ *A*
by (*rule bezout-matrix-JNF-mult-eq*[*OF A′ - - ab A-def B pquvd*], *insert a*, *auto*)

have *invertible-bezout*: *invertible-mat ?BM*

by (*rule invertible-bezout-matrix-JNF*[*OF A is-bezout-ext-euclid-ext2 a - - Aaj*], *insert a n0*, *auto*)
have *BM*: *?BM* ∈ *carrier-mat* (*m*+*n*) (*m*+*n*) **unfolding** *bezout-matrix-JNF-def* **using** *A* **by** *auto*
let *?reduce-a* = *reduce-row-mod-D-abs ?A a xs D m*
define *A′1* **where** *A′1* = *mat-of-rows n* [*Matrix.row ?A i*. *i* ← [*0*..<*m*]]
define *A′2* **where** *A′2* = *mat-of-rows n* [*Matrix.row ?A i*. *i* ← [*m*..<*dim-row A*]]
have *A-A′-D*: *?A* = *A′1* @$_r$ *A′2* **using** *append-rows-split A*
by (*metis* (*no-types*, *lifting*) *A′1-def A′2-def A-carrier carrier-matD le-add1*)
have *j-A′1-A′2*: ∀*j*∈*set xs*. *j* < *n* ∧ *A′2* $$ (*j*, *j*) = *D* ∧ (∀*j′*∈{*0*..<*n*} − {*j*}. *A′2* $$ (*j*, *j′*) = *0*)
proof (*rule ballI*)
fix *ja* **assume** *ja*: *ja*∈*set xs*
have *ja-n*: *ja* < *n* **using** *ja* **unfolding** *xs-def* **by** *auto*
have *ja2*: *ja* < *dim-row A* − *m* **using** *A mn ja-n* **by** *auto*
have *ja-m*: *ja* < *m* **using** *ja-n mn* **by** *auto*
have *abs-A-a-ja-D*: |(*?A* $$ (*a*,*ja*))| > *D* **using** *ja* **unfolding** *xs-def* **by** *auto*
have *ja-not-0*: *ja* ≠ *0*
proof (*rule ccontr*, *simp*)
assume *ja-a*: *ja* = *0*
have *A-mja-D*: *A*$$(*m*,*ja*) = *D*
proof −
have *A*$$(*m*,*ja*) = (*A′* @$_r$ *B*) $$ (*m*, *ja*) **unfolding** *A-def* **..**
also have ... = *B* $$ (*m*−*m*,*ja*)
by (*metis B append-rows-nth A′ assms*(*9*) *carrier-matD*(*1*) *ja-a less-add-same-cancel1 less-irrefl-nat*)
also have ... = *B* $$ (*0*,*0*) **unfolding** *ja-a* **by** *auto*
also have ... = *D* **using** *mn* **unfolding** *ja-a* **using** *ja-n ja j ja-a* **by** *auto*
finally show *?thesis* **.**
qed
have *?A* $$ (*a*, *ja*) = *p*∗*A*$$(*a*,*ja*) + *q*∗*A*$$(*m*,*ja*) **using** *A-carrier ja-n a A* **by** *auto*
also have ... = *d* **using** *pquvd A assms*(*2*) *ja-n ja-a*
by (*simp add*: *bezout-coefficients-fst-snd euclid-ext2-def*)
also have ... = *gcd* (*A*$$(*a*,*ja*)) (*A*$$(*m*,*ja*))
by (*metis euclid-ext2-works*(*2*) *ja-a pquvd*)
also have *abs*(...) ≤ *D* **using** *A-mja-D* **by** (*simp add*: *D0*)
finally have *abs* (*?A* $$ (*a*, *ja*)) ≤ *D* **.**
thus *False* **using** *abs-A-a-ja-D* **by** *auto*
qed
show *ja* < *n* ∧ *A′2* $$ (*ja*, *ja*) = *D* ∧ (∀*j′*∈{*0*..<*n*} − {*ja*}. *A′2* $$ (*ja*, *j′*) = *0*)
proof −
have *A′2* $$ (*ja*, *ja*) = [*Matrix.row ?A i*. *i* ← [*m*..<*dim-row A*]] ! *ja* $*v ja*
by (*metis* (*no-types*, *lifting*) *A A′2-def add-diff-cancel-left′ carrier-matD*(*1*) *ja-n length-map length-upt mat-of-rows-index*)
also have ... = *?A* $$ (*m* + *ja*, *ja*) **using** *A mn ja-n* **by** *auto*

```
      also have ... = A $$ (m+ja, ja) using A a mn ja-n ja-not-0 by auto
      also have ... =  (A' @r B) $$ (m + ja, ja) unfolding A-def ..
      also have ... = B $$ (ja, ja)
            by (metis B Groups.add-ac(2) append-rows-nth2 assms(1) ja-n mn
nat-SN.compat)
      also have ... = D using j ja by blast
      finally have A2-D: A'2 $$ (ja, ja) = D .

      moreover have (∀j'∈{0..<n} − {ja}. A'2 $$ (ja, j') = 0)
      proof (rule ballI)
        fix j' assume j': j': {0..<n} − {ja}
        have A'2 $$ (ja, j') = [Matrix.row ?A i. i ← [m..<dim-row A]] ! ja $v j'
          unfolding A'2-def by (rule mat-of-rows-index, insert j' ja-n ja2, auto)
        also have ... = ?A $$ (m + ja, j') using A mn ja-n j' by auto
        also have ... = A $$ (m+ja, j') using A a mn ja-n ja-not-0 j' by auto
        also have ... =  (A' @r B) $$ (ja + m, j') unfolding A-def
          by (simp add: add.commute)
        also have ... = B $$ (ja, j')
          by (rule append-rows-nth2[OF A' B - ja-m ja-n], insert j', auto)
        also have ... = 0 using mn j' ja-n j ja by auto
        finally show A'2 $$ (ja, j') = 0 .
      qed
      ultimately show ?thesis using ja-n by simp
    qed
  qed
  have reduce-a-eq: ?reduce-a = Matrix.mat (dim-row ?A) (dim-col ?A)
    (λ(i, k). if i = a ∧ k ∈ set xs then if k = 0 ∧ D dvd ?A $$ (i, k) then D else
?A $$ (i, k) gmod D else ?A $$ (i, k))
  proof (rule reduce-row-mod-D-abs-case-m'[OF A-A'-D - - a j-A'1-A'2 - mn D0])

    show A'2 ∈ carrier-mat n n using A A'2-def by auto
    show A'1 ∈ carrier-mat m n by (simp add: A'1-def mat-of-rows-def)
    show distinct xs using distinct-filter distinct-upt xs-def by blast
  qed
  have reduce-a: ?reduce-a ∈ carrier-mat (m+n) n using reduce-a-eq A by auto
  have ∃P. P ∈ carrier-mat (m + n) (m + n) ∧ invertible-mat P ∧ ?reduce-a =
P ∗ ?A
    by (rule reduce-row-mod-D-abs-invertible-mat-case-m[OF A-A'-D - - - j-A'1-A'2
mn],
        insert a A A'2-def A'1-def, auto)
  from this obtain P where P: P ∈ carrier-mat (m + n) (m + n) and inv-P:
invertible-mat P
    and reduce-a-PA: ?reduce-a = P ∗ ?A by blast
  let ?reduce-b = reduce-row-mod-D-abs ?reduce-a m ys D m
  let ?B' = mat-of-rows n [Matrix.row ?reduce-a i. i ← [0..<m]]
 define reduce-a1 where reduce-a1 = mat-of-rows (dim-col ?reduce-a) [Matrix.row
?reduce-a i. i ← [0..<m]]
 define reduce-a2 where reduce-a2 = mat-of-rows (dim-col ?reduce-a) [Matrix.row
?reduce-a i. i ← [m..<dim-row ?reduce-a]]
```

have *reduce-a-split*: *?reduce-a = reduce-a1* $@_r$ *reduce-a2*
by (*unfold reduce-a1-def reduce-a2-def*, *rule append-rows-split*, *insert mn A*, *auto*)
have *zero-notin-ys*: *0* $\notin$ *set ys*
proof −
have *m*: *m*<*dim-row A* **using** *A n0* **by** *auto*
have *?A* $$ (*m*,*0*) = *u* ∗ *A* $$ (*a*, *0*) + *v* ∗ *A* $$ (*m*, *0*) **using** *m n0 a A* **by** *auto*
also have ... = *0* **using** *pquvd*
by (*smt* (*verit*) *dvd-mult-div-cancel euclid-ext2-def euclid-ext2-works*(*3*) *more-arith-simps*(*11*) *mult.commute mult-minus-left prod.sel*(*1*) *prod.sel*(*2*) *semiring-gcd-class.gcd-dvd1*)
finally show *?thesis* **using** *D0* **unfolding** *ys-def* **by** *auto*
qed
have *reduce-a2*: *reduce-a2* ∈ *carrier-mat n n* **unfolding** *reduce-a2-def* **using** *A* **by** *auto*
have *reduce-a1*: *reduce-a1* ∈ *carrier-mat m n* **unfolding** *reduce-a1-def* **using** *A* **by** *auto*
have *j2*: ∀*j*∈*set ys*. *j* < *n* ∧ *reduce-a2* $$ (*j*, *j*) = *D* ∧ (∀*j*′∈{*0*..<*n*} − {*j*}. *reduce-a2* $$ (*j*, *j*′) = *0*)
proof
fix *j* **assume** *j-in-ys*: *j* ∈ *set ys*
have *a-jm*: *a* ≠ *j*+*m* **using** *a* **by** *auto*
have *m-not-jm*: *m* ≠ *j* + *m* **using** *zero-notin-ys j-in-ys* **by** *fastforce*
have *jm*: *j*+*m* < *dim-row ?A* **using** *A-carrier j-in-ys* **unfolding** *ys-def* **by** *auto*
have *jn*: *j* < *dim-col ?A* **using** *A-carrier j-in-ys* **unfolding** *ys-def* **by** *auto*
have *jm*′: *j*+*m* < *dim-row A* **using** *A-carrier j-in-ys* **unfolding** *ys-def* **by** *auto*
have *jn*′: *j* < *dim-col A* **using** *A-carrier j-in-ys* **unfolding** *ys-def* **by** *auto*
have *reduce-a2* $$ (*j*, *j*′) = *B* $$ (*j*,*j*′) **if** *j*′: *j*′<*n* **for** *j*′
proof −
have *reduce-a2* $$ (*j*, *j*′) = *?reduce-a* $$ (*j*+*m*,*j*′)
by (*rule append-rows-nth2*[*symmetric*, *OF reduce-a1 reduce-a2 reduce-a-split*], *insert j-in-ys mn j*′, *auto simp add*: *ys-def*)
also have ... = *?A* $$ (*j*+*m*, *j*′) **using** *reduce-a-eq jm jn a-jm j*′ *A-carrier* **by** *auto*
also have ... = *A* $$ (*j*+*m*, *j*′) **using** *a-jm m-not-jm jm*′ *jn*′ *j*′ *A-carrier* **by** *auto*
also have ... = *B* $$ (*j*,*j*′)
unfolding *A-def*
by (*meson B append-rows-nth2 assms*(*1*) *j-in-ys j-ys mn nat-SN.compat that*)
finally show *?thesis* **.**
qed
thus *j* < *n* ∧ *reduce-a2* $$ (*j*, *j*) = *D* ∧ (∀*j*′∈{*0*..<*n*} − {*j*}. *reduce-a2* $$ (*j*, *j*′) = *0*)
using *j-ys j-in-ys* **by** *auto*
qed
have *reduce-b-eq*: *?reduce-b* = *Matrix.mat* (*dim-row ?reduce-a*) (*dim-col ?reduce-a*)

(λ(i, k). if i = m ∧ k ∈ set ys then if k = 0 ∧ D dvd ?reduce-a $$ (i, k) then D else ?reduce-a $$ (i, k) gmod D else ?reduce-a $$ (i, k))
by (rule reduce-row-mod-D-abs-case-m''[OF reduce-a-split reduce-a2 reduce-a1 - j2 - mn zero-notin-ys],
insert D0, auto simp add: ys-def)
have ∃ P. P ∈ carrier-mat (m + n) (m + n) ∧ invertible-mat P ∧ ?reduce-b = P ∗ ?reduce-a
by (rule reduce-row-mod-D-abs-invertible-mat-case-m'[OF reduce-a-split reduce-a2 reduce-a1 - j2 - mn zero-notin-ys],
auto simp add: ys-def)
from this obtain Q where Q: Q ∈ carrier-mat (m + n) (m + n) and inv-Q: invertible-mat Q
and reduce-b-Q-reduce: ?reduce-b = Q ∗ ?reduce-a by blast
have reduce-b-eq-reduce: ?reduce-b = (reduce-abs a m D A)
proof (rule eq-matI)
show dr-eq: dim-row ?reduce-b = dim-row (reduce-abs a m D A)
and dc-eq: dim-col ?reduce-b = dim-col (reduce-abs a m D A)
using reduce-preserves-dimensions by auto
fix i ja assume i: i<dim-row (reduce-abs a m D A) and ja: ja< dim-col (reduce-abs a m D A)
have im: i<m+n using A i reduce-preserves-dimensions(3) by auto
have ja-n: ja<n using A ja reduce-preserves-dimensions(4) by auto
show ?reduce-b $$ (i,ja) = (reduce-abs a m D A) $$ (i,ja)
proof (cases (i≠a ∧ i≠m))
case True
have ?reduce-b $$ (i,ja) = ?reduce-a $$ (i,ja) unfolding reduce-b-eq
by (smt (verit) True dr-eq dc-eq i index-mat(1) ja prod.simps(2) reduce-row-mod-D-preserves-dimensions-abs)
also have ... = ?A $$ (i,ja)
by (smt (verit) A True carrier-matD(2) dim-col-mat(1) dim-row-mat(1) i index-mat(1) ja-n
reduce-a-eq reduce-preserves-dimensions(3) split-conv)
also have ... = A $$ (i,ja) using A True im ja-n by auto
also have ... = (reduce-abs a m D A) $$ (i,ja) unfolding reduce-alt-def-not0[OF Aaj pquvd]
using im ja-n A True by auto
finally show ?thesis .
next
case False note a-or-b = False
show ?thesis
proof (cases i=a)
case True note ia = True
hence i-not-b: i≠m using ab by auto
show ?thesis
proof (cases abs((p∗A$$(a,ja) + q∗A$$(m,ja))) > D)
case True note ge-D = True
have ja-in-xs: ja ∈ set xs
unfolding xs-def using True ja-n im a A unfolding set-filter by auto
have 1: ?reduce-b $$ (i,ja) = ?reduce-a $$ (i,ja) unfolding reduce-b-eq

```
            by (smt (verit) ab dc-eq dim-row-mat(1) dr-eq i ia index-mat(1) ja
prod.simps(2)
              reduce-b-eq reduce-row-mod-D-preserves-dimensions-abs(2))
        show ?thesis
        proof (cases ja = 0 ∧ D dvd p∗A$$(a,ja) + q∗A$$(m,ja))
          case True
          have ?reduce-a $$ (i,ja) = D
           unfolding reduce-a-eq using True ab a-or-b i-not-b ja-n im a A ja-in-xs
False by auto
          also have ... = (reduce-abs a m D A) $$ (i,ja)
            unfolding reduce-alt-def-not0[OF Aaj pquvd]
            using True a-or-b i-not-b ja-n im A False ge-D
            by auto
          finally show ?thesis using 1 by simp
        next
          case False
          have ?reduce-a $$ (i,ja) = ?A $$ (i, ja) gmod D
           unfolding reduce-a-eq using True ab a-or-b i-not-b ja-n im a A ja-in-xs
False by auto
          also have ... = (reduce-abs a m D A) $$ (i,ja)
           unfolding reduce-alt-def-not0[OF Aaj pquvd] using True a-or-b i-not-b
ja-n im A False by auto
          finally show ?thesis using 1 by simp
        qed
      next
        case False
        have ja-in-xs: ja ∉ set xs
          unfolding xs-def using False ja-n im a A unfolding set-filter by auto
           have ?reduce-b $$ (i,ja) = ?reduce-a $$ (i,ja) unfolding reduce-b-eq

            by (smt (verit) ab dc-eq dim-row-mat(1) dr-eq i ia index-mat(1) ja
prod.simps(2)
              reduce-b-eq reduce-row-mod-D-preserves-dimensions-abs(2))
        also have ... = ?A $$ (i, ja)
         unfolding reduce-a-eq using False ab a-or-b i-not-b ja-n im a A ja-in-xs
by auto
        also have ... = (reduce-abs a m D A) $$ (i,ja)
          unfolding reduce-alt-def-not0[OF Aaj pquvd] using False a-or-b i-not-b
ja-n im A by auto
        finally show ?thesis .
      qed
    next
      case False note i-not-a = False
      have i-drb: i<dim-row ?reduce-b
        and i-dra: i<dim-row ?reduce-a
        and ja-drb: ja < dim-col ?reduce-b
       and ja-dra: ja < dim-col ?reduce-a using i ja reduce-carrier[OF A] A ja-n
im by auto
```

```
      have ib: i=m using False a-or-b by auto
    show ?thesis
    proof (cases abs((u*A$$(a,ja) + v * A$$(m,ja))) > D)
      case True note ge-D = True
      have ja-in-ys: ja ∈ set ys
        unfolding ys-def using True False ib ja-n im a  A unfolding set-filter
by auto
       have ?reduce-b $$ (i,ja) = (if ja = 0 ∧ D dvd ?reduce-a$$(i,ja) then D
else ?reduce-a $$ (i, ja) gmod D)
        unfolding reduce-b-eq using i-not-a True  ja ja-in-ys
        by (smt (verit) i-dra ja-dra a-or-b index-mat(1) prod.simps(2))
      also have ... = (if ja = 0 ∧ D dvd ?reduce-a$$(i,ja) then D else ?A $$ (i,
ja) gmod D)
       unfolding reduce-a-eq using True ab a-or-b ib False ja-n im a A ja-in-ys
by auto
      also have ... = (reduce-abs a m D A) $$ (i,ja)
      proof (cases ja = 0 ∧ D dvd ?reduce-a$$(i,ja))
        case True
        have ja0: ja=0 using True by auto
        have u * A $$ (a, ja) + v * A $$ (m, ja) = 0
          unfolding euclid-ext2-works[OF pquvd[symmetric]] ja0
        by (smt (verit) euclid-ext2-works[OF pquvd[symmetric]] more-arith-simps(11)
mult.commute mult-minus-left)
        hence abs-0: abs((u*A$$(a,ja) + v * A$$(m,ja))) = 0 by auto
        show ?thesis using abs-0 D0 ge-D by linarith
      next
        case False
        then show ?thesis
          unfolding reduce-alt-def-not0[OF Aaj pquvd] using True ge-D False
a-or-b ib ja-n im A
          using i-not-a by auto
      qed
      finally show ?thesis .
    next
      case False
      have ja-in-ys: ja ∉ set ys
       unfolding ys-def using i-not-a False ib ja-n im a  A unfolding set-filter
by auto
        have ?reduce-b $$ (i,ja) = ?reduce-a $$ (i,ja) unfolding reduce-b-eq

        by (smt (verit) False a-or-b dc-eq dim-row-mat(1) dr-eq i index-mat(1)
ja ja-in-ys
        prod.simps(2) reduce-b-eq reduce-row-mod-D-preserves-dimensions-abs(2))
      also have ... = ?A $$ (i, ja)
       unfolding reduce-a-eq using False ab a-or-b i-not-a ja-n im a A ja-in-ys
by auto
      also have ... = (reduce-abs a m D A) $$ (i,ja)
       unfolding reduce-alt-def-not0[OF Aaj pquvd] using False a-or-b i-not-a
ja-n im A by auto
```

finally show *?thesis* **.**
qed
qed
qed
qed
have *r*: *?reduce-a* = (*P*∗*?BM*) ∗ *A* **using** *A A'-BZ-A BM P reduce-a-PA* **by** *auto*
have *Q* ∗ *P* ∗ *?BM* : *carrier-mat* (*m*+*n*) (*m*+*n*) **using** *P BM Q* **by** *auto*
moreover have *invertible-mat* (*Q* ∗ *P*∗*?BM*)
using *inv-P invertible-bezout BM P invertible-mult-JNF inv-Q Q* **by** (*metis mult-carrier-mat*)
moreover have (*reduce-abs a m D A*) = (*Q* ∗ *P* ∗ *?BM*) ∗ *A* **using** *reduce-a-eq r reduce-b-eq-reduce*
by (*smt* (*verit*) *BM P Q assoc-mult-mat carrier-matD carrier-mat-triv dim-row-mat*(*1*) *index-mult-mat*(*2*,*3*) *reduce-b-Q-reduce*)
ultimately show *?thesis* **by** *auto*
qed

lemma *reduce-not0*:
assumes *A*: *A* ∈ *carrier-mat m n* **and** *a*: *a*<*m* **and** *a-less-b*: *a*<*b* **and** *j*: *0*<*n* **and** *b*: *b*<*m*
and *Aaj*: *A* $$ (*a*,*0*) ≠ *0* **and** *D0*: *D* ≠ *0*
shows *reduce a b D A* $$ (*a*, *0*) ≠ *0* (**is** *?reduce* $$ (*a*,*0*) ≠ -)
and *reduce-abs a b D A* $$ (*a*, *0*) ≠ *0* (**is** *?reduce-abs* $$ (*a*,*0*) ≠ -)
proof −
have *?reduce* $$ (*a*,*0*) = (*let r* = *gcd* (*A* $$ (*a*, *0*)) (*A* $$ (*b*, *0*)) *in if D dvd r then D else r*)
by (*rule reduce-gcd*[*OF A - j Aaj*], *insert a*, *simp*)
also have ... ≠ *0* **unfolding** *Let-def* **using** *D0*
by (*smt* (*verit*) *Aaj gcd-eq-0-iff gmod-0-imp-dvd*)
finally show *reduce a b D A* $$ (*a*, *0*) ≠ *0* **.**
have *?reduce-abs* $$ (*a*,*0*) = (*let r* = *gcd* (*A* $$ (*a*, *0*)) (*A* $$ (*b*, *0*)) *in*
if D < *r then if D dvd r then D else r gmod D else r*)
by (*rule reduce-gcd*[*OF A - j Aaj*], *insert a*, *simp*)
also have ... ≠ *0* **unfolding** *Let-def* **using** *D0*
by (*smt* (*verit*) *Aaj gcd-eq-0-iff gmod-0-imp-dvd*)
finally show *reduce-abs a b D A* $$ (*a*, *0*) ≠ *0* **.**
qed

lemma *reduce-below-not0*:
assumes *A*: *A* ∈ *carrier-mat m n* **and** *a*: *a*<*m* **and** *j*: *0*<*n*
and *Aaj*: *A* $$ (*a*,*0*) ≠ *0*
and *distinct xs* **and** ∀ *x* ∈ *set xs*. *x* < *m* ∧ *a* < *x*
and *D*≠ *0*
shows *reduce-below a xs D A* $$ (*a*, *0*) ≠ *0* (**is** *?R* $$ (*a*,*0*) ≠ -)
using *assms*

proof (*induct a xs D A arbitrary*: *A rule*: *reduce-below.induct*)
case (*1 a D A*)
then show *?case* **by** *auto*
next
case (*2 a x xs D A*)
note *A* = *2.prems*(*1*)
note *a* = *2.prems*(*2*)
note *j* = *2.prems*(*3*)
note *Aaj* = *2.prems*(*4*)
note *d* = *2.prems*(*5*)
note *D0* = *2.prems*(*7*)
note *x-less-xxs* = *2.prems*(*6*)
have *xm*: *x* < *m* **using** *2.prems* **by** *auto*
have *D1*: *D* $\cdot_m$ 1_m *n* ∈ *carrier-mat n n* **by** *simp*
obtain *p q u v d* **where** *pquvd*: (*p*,*q*,*u*,*v*,*d*) = *euclid-ext2* (*A*$$(*a*,*0*)) (*A*$$(*x*,*0*))
by (*metis prod-cases5*)
let *?reduce-ax* = *reduce a x D A*
have *reduce-ax*: *?reduce-ax* ∈ *carrier-mat m n*
by (*metis* (*no-types*, *lifting*) *A carrier-matD carrier-mat-triv reduce-preserves-dimensions*)
have *h*: *reduce-below a xs D* (*reduce a x D A*) $$ (*a*,*0*) ≠ *0*
proof (*rule 2.hyps*)
show *reduce a x D A* $$ (*a*, *0*) ≠ *0*
by (*rule reduce-not0*[*OF A a - j xm Aaj D0*], *insert x-less-xxs*, *simp*)
qed (*insert A a j Aaj d x-less-xxs xm reduce-ax D0*, *auto*)
thus *?case* **by** *auto*
qed

lemma *reduce-below-abs-not0*:
assumes *A*: *A* ∈ *carrier-mat m n* **and** *a*: *a*<*m* **and** *j*: *0*<*n*
and *Aaj*: *A* $$ (*a*,*0*) ≠ *0*
and *distinct xs* **and** ∀ *x* ∈ *set xs*. *x* < *m* ∧ *a* < *x*
and *D*≠ *0*
shows *reduce-below-abs a xs D A* $$ (*a*, *0*) ≠ *0* (**is** *?R* $$ (*a*,*0*) ≠ -)
using *assms*
proof (*induct a xs D A arbitrary*: *A rule*: *reduce-below-abs.induct*)
case (*1 a D A*)
then show *?case* **by** *auto*
next
case (*2 a x xs D A*)
note *A* = *2.prems*(*1*)
note *a* = *2.prems*(*2*)
note *j* = *2.prems*(*3*)
note *Aaj* = *2.prems*(*4*)
note *d* = *2.prems*(*5*)
note *D0* = *2.prems*(*7*)
note *x-less-xxs* = *2.prems*(*6*)
have *xm*: *x* < *m* **using** *2.prems* **by** *auto*

have *D1*: $D \cdot_m 1_m\ n \in$ *carrier-mat n n* **by** *simp*
obtain *p q u v d* **where** *pquvd*: $(p,q,u,v,d) =$ *euclid-ext2* (*A$$*$(a,0)$) (*A$$*$(x,0)$)
by (*metis prod-cases5*)
let *?reduce-ax* = *reduce-abs a x D A*
have *reduce-ax*: *?reduce-ax* $\in$ *carrier-mat m n*
by (*metis* (*no-types*, *lifting*) *A carrier-matD carrier-mat-triv reduce-preserves-dimensions*)
have *h*: *reduce-below-abs a xs D* (*reduce-abs a x D A*) *$$* $(a,0) \neq 0$
proof (*rule 2.hyps*)
show *reduce-abs a x D A $$* $(a, 0) \neq 0$
by (*rule reduce-not0*[*OF A a - j xm Aaj D0*], *insert x-less-xxs*, *simp*)
qed (*insert A a j Aaj d x-less-xxs xm reduce-ax D0*, *auto*)
thus *?case* **by** *auto*
qed

lemma *reduce-below-not0-case-m*:
assumes *A'*: $A' \in$ *carrier-mat m n* **and** *a*: $a<m$ **and** *j*: $0<n$
and *A-def*: $A = A' @_r (D \cdot_m (1_m\ n))$
and *Aaj*: *A $$* $(a,0) \neq 0$
and *mn*: $m \geq n$
and $\forall x \in$ *set xs*. $x < m \wedge a < x$
and $D \neq 0$
shows *reduce-below a* (*xs*@[*m*]) *D A $$* $(a, 0) \neq 0$ (**is** *?R $$* $(a,0) \neq$ -)
using *assms*
proof (*induct a xs D A arbitrary*: *A A' rule*: *reduce-below.induct*)
case (*1 a D A*)
note $A' =$ *1.prems*(*1*)
note $a =$ *1.prems*(*2*)
note $n =$ *1.prems*(*3*)
note *A-def* = *1.prems*(*4*)
note *Aaj* = *1.prems*(*5*)
note *mn* = *1.prems*(*6*)
note *all-less-xxs* = *1.prems*(*7*)
note *D0* = *1.prems*(*8*)
have *A*: $A \in$ *carrier-mat* $(m+n)$ *n* **using** A' *A-def* **by** *auto*
have *reduce-below a* ([] @ [*m*]) *D A $$* $(a, 0) =$ *reduce-below a* [*m*] *D A $$* $(a,$
0) **by** *auto*
also have ... = *reduce a m D A $$* $(a, 0)$ **by** *auto*
also have ... $\neq 0$
by (*rule reduce-not0*[*OF A - a n - Aaj D0*], *insert a n*, *auto*)
finally show *?case* **.**
next
case (*2 a x xs D A*)
note $A' =$ *2.prems*(*1*)
note $a =$ *2.prems*(*2*)
note $n =$ *2.prems*(*3*)
note *A-def* = *2.prems*(*4*)
note *Aaj* = *2.prems*(*5*)

note *mn = 2.prems(6)*
note *x-less-xxs = 2.prems(7)*
note *D0= 2.prems(8)*
have *xm*: $x < m$ **using** *2.prems* **by** *auto*
have *D1*: $D \cdot_m 1_m\ n \in$ *carrier-mat n n* **by** *simp*
have *A*: $A \in$ *carrier-mat* $(m+n)\ n$ **using** *A' A-def* **by** *auto*
obtain *p q u v d* **where** *pquvd*: $(p,q,u,v,d) =$ *euclid-ext2* $(A\$\$(a,0))\ (A\$\$(x,0))$
by (*metis prod-cases5*)
let *?reduce-ax = reduce a x D A*
have *reduce-ax*: *?reduce-ax* $\in$ *carrier-mat* $(m+n)\ n$
by (*metis* (*no-types*, *lifting*) *A carrier-matD carrier-mat-triv reduce-preserves-dimensions*)
have *h*: *reduce-below a* (*xs*@[*m*]) *D* (*reduce a x D A*) $\$\$\ (a,0) \neq 0$
proof (*rule 2.hyps*)
show *reduce a x D A* $\$\$\ (a,\ 0) \neq 0$
by (*rule reduce-not0*[*OF A - - - - - D0*], *insert x-less-xxs j Aaj*, *auto*)
let *?reduce-ax' = mat-of-rows n* (*map* (*Matrix.row ?reduce-ax*) [*0..<m*])
show *?reduce-ax = ?reduce-ax'* $@_r\ D \cdot_m 1_m\ n$ **by** (*rule reduce-append-rows-eq*[*OF A' A-def a xm n Aaj*])
qed (*insert A a j Aaj x-less-xxs xm reduce-ax mn D0*, *auto*)
thus *?case* **by** *auto*
qed

lemma *reduce-below-abs-not0-case-m*:
assumes *A'*: $A' \in$ *carrier-mat m n* **and** *a*: $a<m$ **and** *j*: $0<n$
and *A-def*: $A = A' @_r\ (D \cdot_m (1_m\ n))$
and *Aaj*: $A\ \$\$\ (a,0) \neq 0$
and *mn*: $m \geq n$
and $\forall x \in$ *set xs*. $x < m \wedge a < x$
and $D \neq 0$
shows *reduce-below-abs a* (*xs*@[*m*]) *D A* $\$\$\ (a,\ 0) \neq 0$ (**is** *?R* $\$\$\ (a,0) \neq$ -)
using *assms*
proof (*induct a xs D A arbitrary*: *A A' rule*: *reduce-below-abs.induct*)
case (*1 a D A*)
note *A' = 1.prems(1)*
note *a = 1.prems(2)*
note *n = 1.prems(3)*
note *A-def = 1.prems(4)*
note *Aaj = 1.prems(5)*
note *mn = 1.prems(6)*
note *all-less-xxs = 1.prems(7)*
note *D0 = 1.prems(8)*
have *A*: $A \in$ *carrier-mat* $(m+n)\ n$ **using** *A' A-def* **by** *auto*
have *reduce-below-abs a* ([] @ [*m*]) *D A* $\$\$\ (a,\ 0) =$ *reduce-below-abs a* [*m*] *D A* $\$\$\ (a,\ 0)$ **by** *auto*
also have *... = reduce-abs a m D A* $\$\$\ (a,\ 0)$ **by** *auto*
also have *...* $\neq 0$
by (*rule reduce-not0*[*OF A - a n - Aaj D0*], *insert a n*, *auto*)
finally show *?case* **.**
next

case (*2 a x xs D A*)
note *A′ = 2.prems(1)*
note *a = 2.prems(2)*
note *n = 2.prems(3)*
note *A-def = 2.prems(4)*
note *Aaj = 2.prems(5)*
note *mn = 2.prems(6)*
note *x-less-xxs = 2.prems(7)*
note *D0= 2.prems(8)*
have *xm*: *x < m* **using** *2.prems* **by** *auto*
have *D1*: *D ·$_m$ 1$_m$ n ∈ carrier-mat n n* **by** *simp*
have *A*: *A ∈ carrier-mat (m+n) n* **using** *A′ A-def* **by** *auto*
obtain *p q u v d* **where** *pquvd*: *(p,q,u,v,d) = euclid-ext2 (A$$(a,0)) (A$$(x,0))*
by (*metis prod-cases5*)
let *?reduce-ax = reduce-abs a x D A*
have *reduce-ax*: *?reduce-ax ∈ carrier-mat (m+n) n*
by (*metis* (*no-types*, *lifting*) *A carrier-matD carrier-mat-triv reduce-preserves-dimensions*)
have *h*: *reduce-below-abs a (xs@[m]) D (reduce-abs a x D A) $$ (a,0) ≠ 0*
proof (*rule 2.hyps*)
show *reduce-abs a x D A $$ (a, 0) ≠ 0*
by (*rule reduce-not0*[*OF A - - - - - D0*], *insert x-less-xxs j Aaj*, *auto*)
let *?reduce-ax′ = mat-of-rows n (map (Matrix.row ?reduce-ax) [0..<m])*
show *?reduce-ax = ?reduce-ax′ @$_r$ D ·$_m$ 1$_m$ n* **by** (*rule reduce-append-rows-eq*[*OF A′ A-def a xm n Aaj*])
qed (*insert A a j Aaj x-less-xxs xm reduce-ax mn D0*, *auto*)
thus *?case* **by** *auto*
qed

lemma *reduce-below-invertible-mat*:
assumes *A′*: *A′ ∈ carrier-mat m n* **and** *a*: *a<m* **and** *j*: *0<n*
and *A-def*: *A = A′ @$_r$ (D ·$_m$ (1$_m$ n))*
and *Aaj*: *A $$ (a,0) ≠ 0*
and *distinct xs* **and** *∀ x ∈ set xs. x < m ∧ a < x*
and *m≥n*
and *D>0*
shows (*∃ P. invertible-mat P ∧ P ∈ carrier-mat (m+n) (m+n) ∧ reduce-below a xs D A = P * A*)
using *assms*
proof (*induct a xs D A arbitrary*: *A′ rule*: *reduce-below.induct*)
case (*1 a D A*)
then show *?case*
by (*metis append-rows-def carrier-matD*(*1*) *index-mat-four-block*(*2*) *reduce-below.simps*(*1*) *index-smult-mat*(*2*) *index-zero-mat*(*2*) *invertible-mat-one left-mult-one-mat′ one-carrier-mat*)
next

case (*2 a x xs D A*)
note $A' = 2.prems(1)$
note $a = 2.prems(2)$
note $j = 2.prems(3)$
note *A-def* = *2.prems*(*4*)
note *Aaj* = *2.prems*(*5*)
note $d = 2.prems(6)$
note *x-less-xxs* = *2.prems*(*7*)
note *mn* = *2.prems*(*8*)
note *D-ge0* = *2.prems*(*9*)
have *D0*: *D≠0* **using** *D-ge0* **by** *simp*
have *A*: $A \in carrier\text{-}mat\ (m+n)\ n$ **using** A' *A-def* **by** *auto*
have *xm*: $x < m$ **using** *2.prems* **by** *auto*
have *D1*: $D \cdot_m 1_m\ n \in carrier\text{-}mat\ n\ n$ **by** *simp*
obtain *p q u v d* **where** *pquvd*: $(p,q,u,v,d) = euclid\text{-}ext2\ (A\$\$(a,0))\ (A\$\$(x,0))$
by (*metis prod-cases5*)
let *?reduce-ax* = *reduce a x D A*
have *reduce-ax*: *?reduce-ax* ∈ *carrier-mat* (*m* + *n*) *n*
by (*metis* (*no-types*, *lifting*) *2 add.comm-neutral append-rows-def*
carrier-matD carrier-mat-triv index-mat-four-block(*2*,*3*)
index-one-mat(*2*) *index-smult-mat*(*2*) *index-zero-mat*(*2*,*3*) *reduce-preserves-dimensions*)
have *h*: (∃ *P*. *invertible-mat P* ∧ *P* ∈ *carrier-mat* (*m* + *n*) (*m* + *n*)
∧ *reduce-below a xs D* (*reduce a x D A*) = *P* ∗ *reduce a x D A*)
proof (*rule 2.hyps*[*OF - a j - -*])
let *?A'* = *mat-of-rows n* (*map* (*Matrix.row ?reduce-ax*) [*0*..<*m*])
show *reduce a x D A* = *?A'* @$_r$ *D* ·$_m$ *1*$_m$ *n*
by (*rule reduce-append-rows-eq*[*OF A' A-def a xm j Aaj*])
show *reduce a x D A* $$ (*a*, *0*) ≠ *0*
by (*rule reduce-not0*[*OF A - - j - Aaj*], *insert 2.prems*, *auto*)
qed (*insert mn d x-less-xxs D-ge0*, *auto*)
from *this* **obtain** *P* **where** *inv-P*: *invertible-mat P* **and** *P*: *P* ∈ *carrier-mat* (*m* + *n*) (*m* + *n*)
and *rb-Pr*: *reduce-below a xs D* (*reduce a x D A*) = *P* ∗ *reduce a x D A* **by** *blast*
have ∗: *reduce-below a* (*x* # *xs*) *D A* = *reduce-below a xs D* (*reduce a x D A*) **by** *simp*
have ∃ *Q*. *invertible-mat Q* ∧ *Q* ∈ *carrier-mat* (*m*+*n*) (*m*+*n*) ∧ (*reduce a x D A*) = *Q* ∗ *A*
by (*rule reduce-invertible-mat*[*OF A' a j xm - A-def Aaj*], *insert 2.prems*, *auto*)
from *this* **obtain** *Q* **where** *inv-Q*: *invertible-mat Q* **and** *Q*: *Q* ∈ *carrier-mat* (*m* + *n*) (*m* + *n*)
and *r-QA*: *reduce a x D A* = *Q* ∗ *A* **by** *blast*
have *invertible-mat* (*P*∗*Q*) **using** *inv-P inv-Q P Q invertible-mult-JNF* **by** *blast*
moreover have *P* ∗ *Q* ∈ *carrier-mat* (*m*+*n*) (*m*+*n*) **using** *P Q* **by** *auto*
moreover have *reduce-below a* (*x* # *xs*) *D A* = (*P*∗*Q*) ∗ *A*
by (*smt* (*verit*) *P Q* ∗ *assoc-mult-mat carrier-matD*(*1*) *carrier-mat-triv index-mult-mat*(*2*)
r-QA rb-Pr reduce-preserves-dimensions(*1*))
ultimately show *?case* **by** *blast*
qed

lemma *reduce-below-abs-invertible-mat*:
assumes *A'*: $A' \in$ *carrier-mat m n* **and** *a*: $a<m$ **and** *j*: $0<n$
and *A-def*: $A = A' \mathbin{@_r} (D \cdot_m (1_m\ n))$
and *Aaj*: $A\ \$\$\ (a,0) \neq 0$
and *distinct xs* **and** $\forall x \in$ *set xs*. $x < m \wedge a < x$
and $m \geq n$
and $D>0$
shows ($\exists P$. *invertible-mat P* $\wedge$ $P \in$ *carrier-mat* $(m+n)\ (m+n)$ $\wedge$ *reduce-below-abs a xs D A* $= P * A$)
using *assms*
proof (*induct a xs D A arbitrary*: *A' rule*: *reduce-below-abs.induct*)
case (*1 a D A*)
then show *?case*
by (*metis carrier-append-rows invertible-mat-one left-mult-one-mat one-carrier-mat reduce-below-abs.simps*(*1*) *smult-carrier-mat*)
next
case (*2 a x xs D A*)
note $A' = 2.prems(1)$
note $a = 2.prems(2)$
note $j = 2.prems(3)$
note *A-def* $= 2.prems(4)$
note *Aaj* $= 2.prems(5)$
note $d = 2.prems(6)$
note *x-less-xxs* $= 2.prems(7)$
note $mn = 2.prems(8)$
note *D-ge0* $= 2.prems(9)$
have *D0*: $D \neq 0$ **using** *D-ge0* **by** *simp*
have *A*: $A \in$ *carrier-mat* $(m+n)\ n$ **using** A' *A-def* **by** *auto*
have *xm*: $x < m$ **using** *2.prems* **by** *auto*
have *D1*: $D \cdot_m 1_m\ n \in$ *carrier-mat n n* **by** *simp*
obtain *p q u v d* **where** *pquvd*: $(p,q,u,v,d) =$ *euclid-ext2* $(A\$\$(a,0))\ (A\$\$(x,0))$
by (*metis prod-cases5*)
let *?reduce-ax* $=$ *reduce-abs a x D A*
have *reduce-ax*: *?reduce-ax* $\in$ *carrier-mat* $(m + n)\ n$
by (*metis* (*no-types*, *lifting*) *2 add.comm-neutral append-rows-def carrier-matD carrier-mat-triv index-mat-four-block*(*2*,*3*) *index-one-mat*(*2*) *index-smult-mat*(*2*) *index-zero-mat*(*2*,*3*) *reduce-preserves-dimensions*)
have *h*: ($\exists P$. *invertible-mat P* $\wedge$ $P \in$ *carrier-mat* $(m + n)\ (m + n)$
$\wedge$ *reduce-below-abs a xs D* (*reduce-abs a x D A*) $=$ $P *$ *reduce-abs a x D A*)
proof (*rule 2.hyps*[*OF - a j - -*])
let *?A'* $=$ *mat-of-rows n* (*map* (*Matrix.row ?reduce-ax*) $[0..<m]$)
show *reduce-abs a x D A* $=$ *?A'* $\mathbin{@_r} D \cdot_m 1_m\ n$
by (*rule reduce-append-rows-eq*[*OF A' A-def a xm j Aaj*])
show *reduce-abs a x D A* $\$\$\ (a,\ 0) \neq 0$
by (*rule reduce-not0*[*OF A - - j - Aaj*], *insert 2.prems*, *auto*)
qed (*insert mn d x-less-xxs D-ge0*, *auto*)
from *this* **obtain** *P* **where** *inv-P*: *invertible-mat P* **and** *P*: $P \in$ *carrier-mat* (*m*

+ n) (m + n)
    and *rb-Pr*: *reduce-below-abs a xs D* (*reduce-abs a x D A*) = *P* ∗ *reduce-abs a x D A* **by** *blast*
  have ∗: *reduce-below-abs a* (*x* # *xs*) *D A* = *reduce-below-abs a xs D* (*reduce-abs a x D A*) **by** *simp*
  have ∃ *Q*. *invertible-mat Q* ∧ *Q* ∈ *carrier-mat* (*m*+*n*) (*m*+*n*) ∧ (*reduce-abs a x D A*) = *Q* ∗ *A*
    by (*rule reduce-abs-invertible-mat*[*OF A*′ *a j xm - A-def Aaj*], *insert 2.prems*, *auto*)
  from *this* **obtain** *Q* **where** *inv-Q*: *invertible-mat Q* **and** *Q*: *Q* ∈ *carrier-mat* (*m* + *n*) (*m* + *n*)
    and *r-QA*: *reduce-abs a x D A* = *Q* ∗ *A* **by** *blast*
  have *invertible-mat* (*P*∗*Q*) **using** *inv-P inv-Q P Q invertible-mult-JNF* **by** *blast*
  moreover have *P* ∗ *Q* ∈ *carrier-mat* (*m*+*n*) (*m*+*n*) **using** *P Q* **by** *auto*
  moreover have *reduce-below-abs a* (*x* # *xs*) *D A* = (*P*∗*Q*) ∗ *A*
    by (*smt* (*verit*) *P Q* ∗ *assoc-mult-mat carrier-matD*(*1*) *carrier-mat-triv index-mult-mat*(*2*)
      r-QA rb-Pr reduce-preserves-dimensions(*3*))
  ultimately show *?case* **by** *blast*
qed

lemma *reduce-below-preserves*:
  assumes *A*′: *A*′ ∈ *carrier-mat m n* **and** *a*: *a*<*m* **and** *j*: *j*<*n*
    and *A-def*: *A* = *A*′ @$_r$ (*D* ·$_m$ (*1*$_m$ *n*))
    and *Aaj*: *A* $$ (*a*,*0*) ≠ *0*
    and *mn*: *m*≥*n*
  assumes *i* ∉ *set xs* **and** *distinct xs* **and** ∀ *x* ∈ *set xs*. *x* < *m* ∧ *a* < *x*
    and *i*≠*a* **and** *i*<*m*+*n*
  and *D*>*0*
  shows *reduce-below a xs D A* $$ (*i*,*j*) = *A* $$ (*i*,*j*)
  using *assms*
proof (*induct a xs D A arbitrary*: *A*′ *i rule*: *reduce-below.induct*)
  case (*1 a D A*)
  then show *?case* **by** *auto*
next
  case (*2 a x xs D A*)
  note *A*′ = *2.prems*(*1*)
  note *a* = *2.prems*(*2*)
  note *j* = *2.prems*(*3*)
  note *A-def* = *2.prems*(*4*)
  note *Aaj* = *2.prems*(*5*)
  note *mn* = *2.prems*(*6*)
  note *i-set-xxs* = *2.prems*(*7*)
  note *d* = *2.prems*(*8*)
  note *xxs-less-m* = *2.prems*(*9*)
  note *ia* = *2.prems*(*10*)
  note *imm* = *2.prems*(*11*)

note *D-ge0 = 2.prems(12)*
have *D0*: *D≠0* **using** *D-ge0* **by** *simp*
have *A*: *A ∈ carrier-mat (m+n) n* **using** *A′ mn A-def* **by** *auto*
have *xm*: *x < m* **using** *2.prems* **by** *auto*
have *D1*: *D ·$_m$ 1$_m$ n ∈ carrier-mat n n* **by** (*simp add*: *mn*)
obtain *p q u v d* **where** *pquvd*: *(p,q,u,v,d) = euclid-ext2 (A$$(a,0)) (A$$(x,0))*
by (*metis prod-cases5*)
let *?reduce-ax = (reduce a x D A)*
have *reduce-ax*: *?reduce-ax ∈ carrier-mat (m + n) n*
by (*metis* (*no-types*, *lifting*) *2 add.comm-neutral append-rows-def*
carrier-matD carrier-mat-triv index-mat-four-block(2,3)
index-one-mat(2) index-smult-mat(2) index-zero-mat(2,3) reduce-preserves-dimensions)
have *reduce-below a (x # xs) D A $$ (i, j) = reduce-below a xs D (reduce a x D*
A) $$ (i, j)
by *auto*
also have *... = reduce a x D A $$ (i, j)*
proof (*rule 2.hyps*[*OF - a j - - mn - - - ia imm D-ge0*])
let *?A′ = mat-of-rows n (map (Matrix.row ?reduce-ax) [0..<m])*
show *reduce a x D A = ?A′ @$_r$ D ·$_m$ 1$_m$ n*
by (*rule reduce-append-rows-eq*[*OF A′ A-def a xm - Aaj*], *insert j*, *auto*)
show *i ∉ set xs* **using** *i-set-xxs* **by** *auto*
show *distinct xs* **using** *d* **by** *auto*
show *∀ x∈set xs. x < m ∧ a < x* **using** *xxs-less-m* **by** *auto*
show *reduce a x D A $$ (a, 0) ≠ 0*
by (*rule reduce-not0*[*OF A - - - - Aaj*], *insert 2.prems*, *auto*)
show *?A′ ∈ carrier-mat m n* **by** *auto*
qed
also have *... = A $$ (i,j)* **by** (*rule reduce-preserves*[*OF A j Aaj*], *insert 2.prems*,
auto)
finally show *?case* **.**
qed

lemma *reduce-below-abs-preserves*:
assumes *A′*: *A′ ∈ carrier-mat m n* **and** *a*: *a<m* **and** *j*: *j<n*
and *A-def*: *A = A′ @$_r$ (D ·$_m$ (1$_m$ n))*
and *Aaj*: *A $$ (a,0) ≠ 0*
and *mn*: *m≥n*
assumes *i ∉ set xs* **and** *distinct xs* **and** *∀ x ∈ set xs. x < m ∧ a < x*
and *i≠a* **and** *i<m+n*
and *D>0*
shows *reduce-below-abs a xs D A $$ (i,j) = A $$ (i,j)*
using *assms*
proof (*induct a xs D A arbitrary*: *A′ i rule*: *reduce-below-abs.induct*)
case (*1 a D A*)
then show *?case* **by** *auto*
next

case (*2 a x xs D A*)
note *A′ = 2.prems(1)*
note *a = 2.prems(2)*
note *j = 2.prems(3)*
note *A-def = 2.prems(4)*
note *Aaj = 2.prems(5)*
note *mn = 2.prems(6)*
note *i-set-xxs = 2.prems(7)*
note *d = 2.prems(8)*
note *xxs-less-m = 2.prems(9)*
note *ia = 2.prems(10)*
note *imm = 2.prems(11)*
note *D-ge0 = 2.prems(12)*
have *D0*: *D≠0* **using** *D-ge0* **by** *simp*
have *A*: *A ∈ carrier-mat (m+n) n* **using** *A′ mn A-def* **by** *auto*
have *xm*: *x < m* **using** *2.prems* **by** *auto*
have *D1*: *D ·$_m$ 1$_m$ n ∈ carrier-mat n n* **by** (*simp add*: *mn*)
obtain *p q u v d* **where** *pquvd*: *(p,q,u,v,d) = euclid-ext2 (A$$(a,0)) (A$$(x,0))*
by (*metis prod-cases5*)
let *?reduce-ax = (reduce-abs a x D A)*
have *reduce-ax*: *?reduce-ax ∈ carrier-mat (m + n) n*
by (*metis* (*no-types*, *lifting*) *2 add.comm-neutral append-rows-def*
carrier-matD carrier-mat-triv index-mat-four-block(2,3)
index-one-mat(2) index-smult-mat(2) index-zero-mat(2,3) reduce-preserves-dimensions)
have *reduce-below-abs a (x # xs) D A $$ (i, j) = reduce-below-abs a xs D*
(reduce-abs a x D A) $$ (i, j)
by *auto*
also have *... = reduce-abs a x D A $$ (i, j)*
proof (*rule 2.hyps*[*OF - a j - - mn - - - ia imm D-ge0*])
let *?A′ = mat-of-rows n (map (Matrix.row ?reduce-ax) [0..<m])*
show *reduce-abs a x D A = ?A′ @$_r$ D ·$_m$ 1$_m$ n*
by (*rule reduce-append-rows-eq*[*OF A′ A-def a xm - Aaj*], *insert j*, *auto*)
show *i ∉ set xs* **using** *i-set-xxs* **by** *auto*
show *distinct xs* **using** *d* **by** *auto*
show *∀ x∈set xs. x < m ∧ a < x* **using** *xxs-less-m* **by** *auto*
show *reduce-abs a x D A $$ (a, 0) ≠ 0*
by (*rule reduce-not0*[*OF A - - - - Aaj*], *insert 2.prems*, *auto*)
show *?A′ ∈ carrier-mat m n* **by** *auto*
qed
also have *... = A $$ (i,j)* **by** (*rule reduce-preserves*[*OF A j Aaj*], *insert 2.prems*,
auto)
finally show *?case* **.**
qed

lemma *reduce-below-0*:
assumes *A′*: *A′ ∈ carrier-mat m n* **and** *a*: *a<m* **and** *j*: *0<n*
and *A-def*: *A = A′ @$_r$ (D ·$_m$ (1$_m$ n))*

```
    and Aaj: A $$ (a,0) ≠ 0
    and mn: m≥n
  assumes i ∈ set xs and distinct xs and ∀ x ∈ set xs. x < m ∧ a < x
  and D>0
  shows reduce-below a xs D A $$ (i,0) = 0
  using assms
proof (induct a xs D A arbitrary: A' i rule: reduce-below.induct)
  case (1 a D A)
  then show ?case by auto
next
  case (2 a x xs D A)
  note A' = 2.prems(1)
  note a = 2.prems(2)
  note j = 2.prems(3)
  note A-def = 2.prems(4)
  note Aaj = 2.prems(5)
  note mn = 2.prems(6)
  note i-set-xxs = 2.prems(7)
  note d = 2.prems(8)
  note xxs-less-m = 2.prems(9)
  note D-ge0 = 2.prems(10)
  have D0: D≠0 using D-ge0 by simp
  have A: A ∈ carrier-mat (m+n) n using A' mn A-def by auto
  have xm: x < m  using 2.prems by auto
  have D1: D ·_m 1_m n ∈ carrier-mat n n by (simp add: mn)
  obtain p q u v d where pquvd: (p,q,u,v,d) = euclid-ext2 (A$$(a,0)) (A$$(x,0))
    by (metis prod-cases5)
  let ?reduce-ax = reduce a x D A
  have reduce-ax: ?reduce-ax ∈ carrier-mat (m + n) n
    by (metis (no-types, lifting) 2 add.comm-neutral append-rows-def
        carrier-matD carrier-mat-triv index-mat-four-block(2,3)
      index-one-mat(2) index-smult-mat(2) index-zero-mat(2,3) reduce-preserves-dimensions)
  show ?case
  proof (cases i=x)
    case True
    have reduce-below a (x # xs) D A $$ (i, 0) = reduce-below a xs D (reduce a x
D A) $$ (i, 0)
      by auto
    also have ... = (reduce a x D A) $$ (i, 0)
    proof (rule reduce-below-preserves[OF - a j - - mn ])
      let ?A' = mat-of-rows n (map (Matrix.row ?reduce-ax) [0..<m])
      show reduce a x D A = ?A' @_r D ·_m 1_m n
        by (rule reduce-append-rows-eq[OF A' A-def a xm j Aaj])
      show distinct xs using d by auto
      show ∀ x∈set xs. x < m ∧ a < x using xxs-less-m by auto
      show reduce a x D A $$ (a, 0) ≠ 0
        by (rule reduce-not0[OF A - - j - Aaj], insert 2.prems, auto)
      show ?A' ∈ carrier-mat m n by auto
      show i ∉ set xs using True d by auto
```

```
    show i ≠ a using 2.prems by blast
    show i < m + n
      by (simp add: True trans-less-add1 xm)
  qed (insert D-ge0)
  also have ... = 0 unfolding True by (rule reduce-0[OF A - j - - Aaj], insert
2.prems, auto)
  finally show ?thesis .
 next
  case False note i-not-x = False
  have h: reduce-below a xs D (reduce a x D A) $$ (i, 0) = 0
  proof (rule 2.hyps[OF - a j - - mn])
    let ?A' = mat-of-rows n (map (Matrix.row ?reduce-ax) [0..<m])
    show reduce a x D A = ?A' @r D ·m 1m n
    proof (rule matrix-append-rows-eq-if-preserves[OF reduce-ax D1])
      show ∀ i∈{m..<m + n}. ∀ ja<n. ?reduce-ax $$ (i, ja) = (D ·m 1m n) $$
(i − m, ja)
      proof (rule+)
        fix i ja assume i: i ∈ {m..<m + n} and ja: ja < n
       have ja-dc: ja < dim-col A and i-dr: i < dim-row A using i ja A by auto
        have i-not-a: i ≠ a using i a by auto
        have i-not-x: i ≠ x using i xm by auto
        have ?reduce-ax $$ (i,ja) = A $$ (i,ja)
          unfolding reduce-alt-def-not0[OF Aaj pquvd] using ja-dc i-dr i-not-a
i-not-x by auto
          also have ... = (if i < dim-row A' then A' $$(i,ja) else (D ·m (1m
n))$$(i−m,ja))
          by (unfold A-def, rule append-rows-nth[OF A' D1 - ja], insert A i-dr,
simp)
        also have ... = (D ·m 1m n) $$ (i − m, ja) using i A' by auto
        finally show ?reduce-ax $$ (i,ja) = (D ·m 1m n) $$ (i − m, ja) .
      qed
    qed
    show i ∈ set xs using i-set-xxs i-not-x by auto
    show distinct xs using d by auto
    show ∀ x∈set xs. x < m ∧ a < x using xxs-less-m by auto
    show reduce a x D A $$ (a, 0) ≠ 0
      by (rule reduce-not0[OF A - - j - Aaj], insert 2.prems, auto)
    show ?A' ∈ carrier-mat m n by auto
  qed (insert D-ge0)
  have reduce-below a (x # xs) D A $$ (i, 0) = reduce-below a xs D (reduce a x
D A) $$ (i, 0)
    by auto
  also have ... = 0 using h .
  finally show ?thesis .
 qed
qed

lemma reduce-below-abs-0:
 assumes A': A' ∈ carrier-mat m n and a: a<m and j: 0<n
```

```
    and A-def: A = A' @_r (D ·_m (1_m n))
    and Aaj: A $$ (a,0) ≠ 0
    and mn: m≥n
  assumes i ∈ set xs and distinct xs and ∀ x ∈ set xs. x < m ∧ a < x
  and D>0
  shows reduce-below-abs a xs D A $$ (i,0) = 0
  using assms
proof (induct a xs D A arbitrary: A' i rule: reduce-below-abs.induct)
  case (1 a D A)
  then show ?case by auto
next
  case (2 a x xs D A)
  note A' = 2.prems(1)
  note a = 2.prems(2)
  note j = 2.prems(3)
  note A-def = 2.prems(4)
  note Aaj = 2.prems(5)
  note mn = 2.prems(6)
  note i-set-xxs = 2.prems(7)
  note d = 2.prems(8)
  note xxs-less-m = 2.prems(9)
  note D-ge0 = 2.prems(10)
  have D0: D≠0 using D-ge0 by simp
  have A: A ∈ carrier-mat (m+n) n using A' mn A-def by auto
  have xm: x < m  using 2.prems by auto
  have D1: D ·_m 1_m n ∈ carrier-mat n n by (simp add: mn)
  obtain p q u v d where pquvd: (p,q,u,v,d) = euclid-ext2 (A$$(a,0)) (A$$(x,0))
    by (metis prod-cases5)
  let ?reduce-ax = reduce-abs a x D A
  have reduce-ax: ?reduce-ax ∈ carrier-mat (m + n) n
    by (metis (no-types, lifting) 2 add.comm-neutral append-rows-def
        carrier-matD carrier-mat-triv index-mat-four-block(2,3)
      index-one-mat(2) index-smult-mat(2) index-zero-mat(2,3) reduce-preserves-dimensions)
  show ?case
  proof (cases i=x)
    case True
     have reduce-below-abs a (x # xs) D A $$ (i, 0) = reduce-below-abs a xs D
(reduce-abs a x D A) $$ (i, 0)
      by auto
    also have ... = (reduce-abs a x D A) $$ (i, 0)
    proof (rule reduce-below-abs-preserves[OF - a j - - mn ])
      let ?A' = mat-of-rows n (map (Matrix.row ?reduce-ax) [0..<m])
      show reduce-abs a x D A = ?A' @_r D ·_m 1_m n
        by (rule reduce-append-rows-eq[OF A' A-def a xm j Aaj])
      show distinct xs using d by auto
      show ∀ x∈set xs. x < m ∧ a < x using xxs-less-m by auto
      show reduce-abs a x D A $$ (a, 0) ≠ 0
        by (rule reduce-not0[OF A - - j - Aaj], insert 2.prems, auto)
      show ?A' ∈ carrier-mat m n by auto
```

```
    show i ∉ set xs using True d by auto
    show i ≠ a using 2.prems by blast
    show i < m + n
      by (simp add: True trans-less-add1 xm)
  qed (insert D-ge0)
  also have ... = 0 unfolding True by (rule reduce-0[OF A - j - - Aaj], insert
2.prems, auto)
  finally show ?thesis .
 next
  case False note i-not-x = False
  have h: reduce-below-abs a xs D (reduce-abs a x D A) $$ (i, 0) = 0
  proof (rule 2.hyps[OF - a j - - mn])
    let ?A' = mat-of-rows n (map (Matrix.row ?reduce-ax) [0..<m])
    show reduce-abs a x D A = ?A' @r D ·m 1m n
    proof (rule matrix-append-rows-eq-if-preserves[OF reduce-ax D1])
      show ∀ i∈{m..<m + n}. ∀ ja<n. ?reduce-ax $$ (i, ja) = (D ·m 1m n) $$
(i − m, ja)
      proof (rule+)
        fix i ja assume i: i ∈ {m..<m + n} and ja: ja < n
        have ja-dc: ja < dim-col A and i-dr: i < dim-row A using i ja A by auto
        have i-not-a: i ≠ a using i a by auto
        have i-not-x: i ≠ x using i xm by auto
        have ?reduce-ax $$ (i,ja) = A $$ (i,ja)
          unfolding reduce-alt-def-not0[OF Aaj pquvd] using ja-dc i-dr i-not-a
i-not-x by auto
          also have ... = (if i < dim-row A' then A' $$(i,ja) else (D ·m (1m
n))$$(i−m,ja))
          by (unfold A-def, rule append-rows-nth[OF A' D1 - ja], insert A i-dr,
simp)
        also have ... = (D ·m 1m n) $$ (i − m, ja) using i A' by auto
        finally show ?reduce-ax $$ (i,ja) = (D ·m 1m n) $$ (i − m, ja) .
      qed
    qed
    show i ∈ set xs using i-set-xxs i-not-x by auto
    show distinct xs using d by auto
    show ∀ x∈set xs. x < m ∧ a < x using xxs-less-m by auto
    show reduce-abs a x D A $$ (a, 0) ≠ 0
      by (rule reduce-not0[OF A - - j - Aaj], insert 2.prems, auto)
    show ?A' ∈ carrier-mat m n by auto
  qed (insert D-ge0)
   have reduce-below-abs a (x # xs) D A $$ (i, 0) = reduce-below-abs a xs D
(reduce-abs a x D A) $$ (i, 0)
    by auto
  also have ... = 0 using h .
  finally show ?thesis .
 qed
qed
```

lemma *reduce-below-preserves-case-m*:
assumes *A'*: $A' \in$ *carrier-mat m n* **and** *a*: $a<m$ **and** *j*: $j<n$
and *A-def*: $A = A' \mathbin{@}_r (D \cdot_m (1_m\ n))$
and *Aaj*: $A\ \$\$\ (a,0) \neq 0$
and *mn*: $m \geq n$
assumes $i \notin$ *set xs* **and** *distinct xs* **and** $\forall x \in$ *set xs*. $x < m \wedge a < x$
and $i \neq a$ **and** $i<m+n$ **and** $i \neq m$
and $D>0$
shows *reduce-below a* (*xs* @ [*m*]) *D A* $\$\$\ (i,j) = A\ \$\$\ (i,j)$
using *assms*
proof (*induct a xs D A arbitrary*: *A' i rule*: *reduce-below.induct*)
case (*1 a D A*)
have *reduce-below a* ([] @ [*m*]) *D A* $\$\$\ (i, j) =$ *reduce-below a* [*m*] *D A* $\$\$\ (i, j)$
by *auto*
also have ... = *reduce a m D A* $\$\$\ (i,j)$ **by** *auto*
also have ... = $A\ \$\$\ (i,j)$
by (*rule reduce-preserves*, *insert 1*, *auto*)
finally show *?case* **.**
next
case (*2 a x xs D A*)
note $A' = 2.prems(1)$
note $a = 2.prems(2)$
note $j = 2.prems(3)$
note *A-def* $= 2.prems(4)$
note *Aaj* $= 2.prems(5)$
note *mn* $= 2.prems(6)$
note *i-set-xxs* $= 2.prems(7)$
note $d = 2.prems(8)$
note *xxs-less-m* $= 2.prems(9)$
note *ia* $= 2.prems(10)$
note *imm* $= 2.prems(11)$
note *D-ge0* $= 2.prems(13)$
have *D0*: $D \neq 0$ **using** *D-ge0* **by** *simp*
have *A*: $A \in$ *carrier-mat* $(m+n)\ n$ **using** A' *mn A-def* **by** *auto*
have *xm*: $x < m$ **using** *2.prems* **by** *auto*
have *D1*: $D \cdot_m 1_m\ n \in$ *carrier-mat n n* **by** (*simp add*: *mn*)
obtain *p q u v d* **where** *pquvd*: $(p,q,u,v,d) =$ *euclid-ext2* $(A\$\$(a,0))\ (A\$\$(x,0))$
by (*metis prod-cases5*)
let *?reduce-ax* = (*reduce a x D A*)
have *reduce-ax*: *?reduce-ax* $\in$ *carrier-mat* $(m + n)\ n$
by (*metis* (*no-types*, *lifting*) A' *A-def add.comm-neutral append-rows-def*
carrier-matD carrier-mat-triv index-mat-four-block(*2*,*3*)
index-one-mat(*2*) *index-smult-mat*(*2*) *index-zero-mat*(*2*,*3*) *reduce-preserves-dimensions*)
have *reduce-below a* ((*x* # *xs*) @ [*m*]) *D A* $\$\$\ (i, j)$
$=$ *reduce-below a* (*xs*@[*m*]) *D* (*reduce a x D A*) $\$\$\ (i, j)$
by *auto*
also have ... = *reduce a x D A* $\$\$\ (i, j)$

proof (*rule 2.hyps*[*OF - a j - - mn - - - ia imm - D-ge0*])
let *?A′* = *mat-of-rows n* (*map* (*Matrix.row ?reduce-ax*) [*0..<m*])
show *reduce a x D A* = *?A′* $@_r$ *D* $\cdot_m$ 1_m *n*
by (*rule reduce-append-rows-eq*[*OF A′ A-def a xm - Aaj*], *insert j*, *auto*)
show *i* ∉ *set xs* **using** *i-set-xxs* **by** *auto*
show *distinct xs* **using** *d* **by** *auto*
show ∀ *x*∈*set xs*. *x* < *m* ∧ *a* < *x* **using** *xxs-less-m* **by** *auto*
show *reduce a x D A* $$ (*a*, *0*) ≠ *0*
by (*rule reduce-not0*[*OF A - - - - Aaj*], *insert 2.prems*, *auto*)
show *?A′* ∈ *carrier-mat m n* **by** *auto*
show *i*≠*m* **using** *2.prems* **by** *auto*
qed
also have ... = *A* $$ (*i*,*j*) **by** (*rule reduce-preserves*[*OF A j Aaj*], *insert 2.prems*, *auto*)
finally show *?case* **.**
qed

lemma *reduce-below-abs-preserves-case-m*:
assumes *A′*: *A′* ∈ *carrier-mat m n* **and** *a*: *a*<*m* **and** *j*: *j*<*n*
and *A-def*: *A* = *A′* $@_r$ (*D* $\cdot_m$ (1_m *n*))
and *Aaj*: *A* $$ (*a*,*0*) ≠ *0*
and *mn*: *m*≥*n*
assumes *i* ∉ *set xs* **and** *distinct xs* **and** ∀ *x* ∈ *set xs*. *x* < *m* ∧ *a* < *x*
and *i*≠*a* **and** *i*<*m*+*n* **and** *i*≠ *m*
and *D*>*0*
shows *reduce-below-abs a* (*xs* @ [*m*]) *D A* $$ (*i*,*j*) = *A* $$ (*i*,*j*)
using *assms*
proof (*induct a xs D A arbitrary*: *A′ i rule*: *reduce-below-abs.induct*)
case (*1 a D A*)
have *reduce-below-abs a* ([] @ [*m*]) *D A* $$ (*i*, *j*) = *reduce-below-abs a* [*m*] *D A* $$ (*i*, *j*) **by** *auto*
also have ... = *reduce-abs a m D A* $$ (*i*,*j*) **by** *auto*
also have ... = *A* $$ (*i*,*j*)
by (*rule reduce-preserves*, *insert 1*, *auto*)
finally show *?case* **.**
next
case (*2 a x xs D A*)
note *A′* = *2.prems*(*1*)
note *a* = *2.prems*(*2*)
note *j* = *2.prems*(*3*)
note *A-def* = *2.prems*(*4*)
note *Aaj* = *2.prems*(*5*)
note *mn* = *2.prems*(*6*)
note *i-set-xxs* = *2.prems*(*7*)
note *d* = *2.prems*(*8*)
note *xxs-less-m* = *2.prems*(*9*)
note *ia* = *2.prems*(*10*)
note *imm* = *2.prems*(*11*)

```
  note D-ge0 = 2.prems(13)
  have D0: D≠0 using D-ge0 by simp
  have A: A ∈ carrier-mat (m+n) n using A' mn A-def by auto
  have xm: x < m  using 2.prems by auto
  have D1: D ·_m 1_m n ∈ carrier-mat n n by (simp add: mn)
  obtain p q u v d where pquvd: (p,q,u,v,d) = euclid-ext2 (A$$(a,0)) (A$$(x,0))
    by (metis prod-cases5)
  let ?reduce-ax = (reduce-abs a x D A)
  have reduce-ax: ?reduce-ax ∈ carrier-mat (m + n) n
    by (metis (no-types, lifting) A' A-def add.comm-neutral append-rows-def
        carrier-matD carrier-mat-triv index-mat-four-block(2,3)
      index-one-mat(2) index-smult-mat(2) index-zero-mat(2,3) reduce-preserves-dimensions)
  have reduce-below-abs a ((x # xs) @ [m]) D A $$ (i, j)
     = reduce-below-abs a (xs@[m]) D (reduce-abs a x D A) $$ (i, j)
    by auto
  also have ... = reduce-abs a x D A $$ (i, j)
  proof (rule 2.hyps[OF - a j - - mn - - - ia imm - D-ge0])
    let ?A' = mat-of-rows n (map (Matrix.row ?reduce-ax) [0..<m])
    show reduce-abs a x D A = ?A' @_r D ·_m 1_m n
      by (rule reduce-append-rows-eq[OF A' A-def a xm - Aaj], insert j, auto)
    show i ∉ set xs using i-set-xxs by auto
    show distinct xs using d by auto
    show ∀ x∈set xs. x < m ∧ a < x using xxs-less-m by auto
    show reduce-abs a x D A $$ (a, 0) ≠ 0
      by (rule reduce-not0[OF A - - - - Aaj], insert 2.prems, auto)
    show ?A' ∈ carrier-mat m n by auto
    show i≠m using 2.prems by auto
  qed
  also have ... = A $$ (i,j) by (rule reduce-preserves[OF A j Aaj], insert 2.prems,
auto)
  finally show ?case .
qed



lemma reduce-below-0-case-m1:
  assumes A': A' ∈ carrier-mat m n and a: a<m and j: 0<n
    and A-def: A = A' @_r (D ·_m (1_m n))
    and Aaj: A $$ (a,0) ≠ 0
    and mn: m≥n
  assumes distinct xs and ∀ x ∈ set xs. x < m ∧ a < x
    and m≠a
  and D>0
  shows reduce-below a (xs @ [m]) D A $$ (m,0) = 0
  using assms
proof (induct a xs D A arbitrary: A' rule: reduce-below.induct)
  case (1 a D A)
  have A: A ∈ carrier-mat (m+n) n using 1 by auto
  have  reduce-below a ([] @ [m]) D A $$ (m, 0) =  reduce-below a [m] D A $$ (m,
```

0) **by** *auto*
also have ... = *reduce a m D A* $$ (*m*,*0*) **by** *auto*
also have ... = *0* **by** (*rule reduce-0*[*OF A*], *insert 1.prems*, *auto*)
finally show *?case* **.**
next
case (*2 a x xs D A*)
note *A'* = *2.prems*(*1*)
note *a* = *2.prems*(*2*)
note *j* = *2.prems*(*3*)
note *A-def* = *2.prems*(*4*)
note *Aaj* = *2.prems*(*5*)
note *mn* = *2.prems*(*6*)
note *d* = *2.prems*(*7*)
note *xxs-less-m* = *2.prems*(*8*)
note *ma* = *2.prems*(*9*)
note *D-ge0* = *2.prems*(*10*)
have *D0*: *D*≠*0* **using** *D-ge0* **by** *simp*
have *A*: *A* ∈ *carrier-mat* (*m*+*n*) *n* **using** *A'* *mn A-def* **by** *auto*
have *xm*: *x* < *m* **using** *2.prems* **by** *auto*
have *D1*: *D* ·$_m$ *1*$_m$ *n* ∈ *carrier-mat n n* **by** (*simp add*: *mn*)
obtain *p q u v d* **where** *pquvd*: (*p*,*q*,*u*,*v*,*d*) = *euclid-ext2* (*A*$$(*a*,*0*)) (*A*$$(*x*,*0*))
by (*metis prod-cases5*)
let *?reduce-ax* = (*reduce a x D A*)
have *reduce-ax*: *?reduce-ax* ∈ *carrier-mat* (*m* + *n*) *n*
by (*metis* (*no-types*, *lifting*) *2 add.comm-neutral append-rows-def*
carrier-matD carrier-mat-triv index-mat-four-block(*2*,*3*)
index-one-mat(*2*) *index-smult-mat*(*2*) *index-zero-mat*(*2*,*3*) *reduce-preserves-dimensions*)
have *reduce-below a* ((*x* # *xs*) @ [*m*]) *D A* $$ (*m*, *0*) = *reduce-below a* (*xs*@[*m*])
D (*reduce a x D A*) $$ (*m*, *0*)
by *auto*
also have ... = *0*
proof (*rule 2.hyps*[*OF*])
let *?A'* = *mat-of-rows n* (*map* (*Matrix.row ?reduce-ax*) [*0*..<*m*])
show *reduce a x D A* = *?A'* @$_r$ *D* ·$_m$ *1*$_m$ *n*
by (*rule reduce-append-rows-eq*[*OF A' A-def a xm j Aaj*])
show *distinct xs* **using** *d* **by** *auto*
show ∀ *x*∈*set xs*. *x* < *m* ∧ *a* < *x* **using** *xxs-less-m* **by** *auto*
show *reduce a x D A* $$ (*a*, *0*) ≠ *0*
by (*rule reduce-not0*[*OF A - - j - Aaj*], *insert 2.prems*, *auto*)
show *?A'* ∈ *carrier-mat m n* **by** *auto*
qed (*insert 2.prems*, *auto*)
finally show *?case* **.**
qed

lemma *reduce-below-abs-0-case-m1*:
assumes *A'*: *A'* ∈ *carrier-mat m n* **and** *a*: *a*<*m* **and** *j*: *0*<*n*
and *A-def*: *A* = *A'* @$_r$ (*D* ·$_m$ (*1*$_m$ *n*))
and *Aaj*: *A* $$ (*a*,*0*) ≠ *0*
and *mn*: *m*≥*n*

assumes *distinct xs* **and** $\forall x \in set\ xs.\ x < m \wedge a < x$
and $m \neq a$
and $D > 0$
shows *reduce-below-abs a (xs @ [m]) D A $$ (m,0) = 0*
using *assms*
proof (*induct a xs D A arbitrary*: *A' rule*: *reduce-below-abs.induct*)
case (*1 a D A*)
have *A*: $A \in carrier\text{-}mat\ (m+n)\ n$ **using** *1* **by** *auto*
have *reduce-below-abs a ([] @ [m]) D A $$ (m, 0) = reduce-below-abs a [m] D A $$ (m, 0)* **by** *auto*
also have ... = *reduce-abs a m D A $$ (m,0)* **by** *auto*
also have ... = *0* **by** (*rule reduce-0*[*OF A*], *insert 1.prems*, *auto*)
finally show *?case* **.**
next
case (*2 a x xs D A*)
note $A' = 2.prems(1)$
note $a = 2.prems(2)$
note $j = 2.prems(3)$
note *A-def* = *2.prems*(*4*)
note *Aaj* = *2.prems*(*5*)
note *mn* = *2.prems*(*6*)
note *d* = *2.prems*(*7*)
note *xxs-less-m* = *2.prems*(*8*)
note *ma* = *2.prems*(*9*)
note *D-ge0* = *2.prems*(*10*)
have *D0*: $D \neq 0$ **using** *D-ge0* **by** *simp*
have *A*: $A \in carrier\text{-}mat\ (m+n)\ n$ **using** A' *mn A-def* **by** *auto*
have *xm*: $x < m$ **using** *2.prems* **by** *auto*
have *D1*: $D \cdot_m 1_m\ n \in carrier\text{-}mat\ n\ n$ **by** (*simp add*: *mn*)
obtain *p q u v d* **where** *pquvd*: $(p,q,u,v,d) = euclid\text{-}ext2\ (A\$\$(a,0))\ (A\$\$(x,0))$
by (*metis prod-cases5*)
let *?reduce-ax* = (*reduce-abs a x D A*)
have *reduce-ax*: *?reduce-ax* $\in carrier\text{-}mat\ (m + n)\ n$
by (*metis* (*no-types*, *lifting*) *2 add.comm-neutral append-rows-def carrier-matD carrier-mat-triv index-mat-four-block*(*2*,*3*) *index-one-mat*(*2*) *index-smult-mat*(*2*) *index-zero-mat*(*2*,*3*) *reduce-preserves-dimensions*)
have *reduce-below-abs a ((x # xs) @ [m]) D A $$ (m, 0) = reduce-below-abs a (xs@[m]) D (reduce-abs a x D A) $$ (m, 0)*
by *auto*
also have ... = *0*
proof (*rule 2.hyps*[*OF*])
let $?A' = mat\text{-}of\text{-}rows\ n\ (map\ (Matrix.row\ ?reduce\text{-}ax)\ [0..<m])$
show *reduce-abs a x D A* = $?A' \ @_r\ D \cdot_m 1_m\ n$
by (*rule reduce-append-rows-eq*[*OF A' A-def a xm j Aaj*])
show *distinct xs* **using** *d* **by** *auto*
show $\forall x \in set\ xs.\ x < m \wedge a < x$ **using** *xxs-less-m* **by** *auto*
show *reduce-abs a x D A $$ (a, 0)* $\neq$ *0*
by (*rule reduce-not0*[*OF A - - j - Aaj*], *insert 2.prems*, *auto*)
show $?A' \in carrier\text{-}mat\ m\ n$ **by** *auto*

```
  qed (insert 2.prems, auto)
  finally show ?case .
qed



lemma reduce-below-preserves-case-m2:
  assumes A': A' ∈ carrier-mat m n and a: a<m and j: 0<n
    and A-def: A = A' @_r (D ·_m (1_m n))
    and Aaj: A $$ (a,0) ≠ 0
    and mn: m≥n
  assumes i ∈ set xs and distinct xs and ∀ x ∈ set xs. x < m ∧ a < x
    and i≠a and i<m+n
  and D>0
  shows reduce-below a (xs @ [m]) D A $$ (i,0) = reduce-below a xs D A $$ (i,0)
  using assms
proof (induct a xs D A arbitrary: A' i rule: reduce-below.induct)
  case (1 a D A)
  then show ?case by auto
next
  case (2 a x xs D A)
  note A' = 2.prems(1)
  note a = 2.prems(2)
  note j = 2.prems(3)
  note A-def = 2.prems(4)
  note Aaj = 2.prems(5)
  note mn = 2.prems(6)
  note i-set-xxs = 2.prems(7)
  note d = 2.prems(8)
  note xxs-less-m = 2.prems(9)
  note ia = 2.prems(10)
  note imm = 2.prems(11)
  note D-ge0 = 2.prems(12)
  have D0: D≠0 using D-ge0 by simp
  have A: A ∈ carrier-mat (m+n) n using A' mn A-def by auto
  have xm: x < m  using 2.prems by auto
  have D1: D ·_m 1_m n ∈ carrier-mat n n by (simp add: mn)
  obtain p q u v d where pquvd: (p,q,u,v,d) = euclid-ext2 (A$$(a,0)) (A$$(x,0))
    by (metis prod-cases5)
  let ?reduce-ax = (reduce a x D A)
  have reduce-ax: ?reduce-ax ∈ carrier-mat (m + n) n
    by (metis (no-types, lifting) A-def A' add.comm-neutral append-rows-def
        carrier-matD carrier-mat-triv index-mat-four-block(2,3)
      index-one-mat(2) index-smult-mat(2) index-zero-mat(2,3) reduce-preserves-dimensions)
  show ?case
  proof (cases i=x)
    case True
    have reduce-below a ((x # xs) @ [m]) D A $$ (i, 0)
      = reduce-below a (xs @ [m]) D (reduce a x D A) $$ (i, 0)
```

```
      by auto
    also have ... = (reduce a x D A) $$ (i, 0)
    proof (rule reduce-below-preserves-case-m[OF - a j - - mn - - - - - - D-ge0])
      let ?A' = mat-of-rows n (map (Matrix.row ?reduce-ax) [0..<m])
      show reduce a x D A = ?A' @_r D ·_m 1_m n
      proof (rule matrix-append-rows-eq-if-preserves[OF reduce-ax D1])
        show ∀ i∈{m..<m + n}. ∀ ja<n. ?reduce-ax $$ (i, ja) = (D ·_m 1_m n) $$
(i − m, ja)
        proof (rule+)
          fix i ja assume i: i ∈ {m..<m + n} and ja: ja < n
         have ja-dc: ja < dim-col A and i-dr: i < dim-row A using i ja A by auto
          have i-not-a: i ≠ a using i a by auto
          have i-not-x: i ≠ x using i xm by auto
          have ?reduce-ax $$ (i,ja) = A $$ (i,ja)
            unfolding reduce-alt-def-not0[OF Aaj pquvd] using ja-dc i-dr i-not-a
i-not-x by auto
           also have ... = (if i < dim-row A' then A' $$(i,ja) else (D ·_m (1_m
n))$$(i−m,ja))
            by (unfold A-def, rule append-rows-nth[OF A' D1 - ja], insert A i-dr,
simp)
          also have ... = (D ·_m 1_m n) $$ (i − m, ja) using i A' by auto
          finally show ?reduce-ax $$ (i,ja) = (D ·_m 1_m n) $$ (i − m, ja) .
        qed
      qed
      show distinct xs using d by auto
      show ∀ x∈set xs. x < m ∧ a < x using xxs-less-m by auto
      show reduce a x D A $$ (a, 0) ≠ 0
        by (rule reduce-not0[OF A - - - - Aaj], insert 2.prems, auto)
      show ?A' ∈ carrier-mat m n by auto
      show i ∉ set xs using True d by auto
      show i ≠ a using 2.prems by blast
      show i < m + n
        by (simp add: True trans-less-add1 xm)
      show i ≠ m  by (simp add: True less-not-refl3 xm)
    qed
    also have ... = 0 unfolding True by (rule reduce-0[OF A - - - - Aaj], insert
2.prems, auto)
    also have ... = reduce-below a (x # xs) D A $$ (i, 0)
      unfolding True by (rule reduce-below-0[symmetric], insert 2.prems, auto)
    finally show ?thesis .
  next
    case False
    have reduce-below a ((x # xs) @ [m]) D A $$ (i, 0)
      = reduce-below a (xs@[m]) D (reduce a x D A) $$ (i, 0)
      by auto
    also have ... = reduce-below a xs D (reduce a x D A) $$ (i, 0)
    proof (rule 2.hyps[OF - a j - - mn - - - ia imm D-ge0])
      let ?A' = mat-of-rows n (map (Matrix.row ?reduce-ax) [0..<m])
      show reduce a x D A = ?A' @_r D ·_m 1_m n
```

```
      by (rule reduce-append-rows-eq[OF A' A-def a xm j Aaj])
    show i ∈ set xs using i-set-xxs False by auto
    show distinct xs using d by auto
    show ∀ x∈set xs. x < m ∧ a < x using xxs-less-m by auto
    show reduce a x D A $$ (a, 0) ≠ 0
      by (rule reduce-not0[OF A - - j - Aaj], insert 2.prems, auto)
    show ?A' ∈ carrier-mat m n by auto
  qed
  also have ... = reduce-below a (x # xs) D A $$ (i, 0) by auto
  finally show ?thesis .
 qed
qed


lemma reduce-below-abs-preserves-case-m2:
 assumes A': A' ∈ carrier-mat m n and a: a<m and j: 0<n
  and A-def: A = A' @_r (D ·_m (1_m n))
  and Aaj: A $$ (a,0) ≠ 0
  and mn: m≥n
 assumes i ∈ set xs and distinct xs and ∀ x ∈ set xs. x < m ∧ a < x
  and i≠a and i<m+n
 and D>0
 shows reduce-below-abs a (xs @ [m]) D A $$ (i,0) = reduce-below-abs a xs D A
$$ (i,0)
 using assms
proof (induct a xs D A arbitrary: A' i rule: reduce-below-abs.induct)
 case (1 a D A)
 then show ?case by auto
next
 case (2 a x xs D A)
 note A' = 2.prems(1)
 note a = 2.prems(2)
 note j = 2.prems(3)
 note A-def = 2.prems(4)
 note Aaj = 2.prems(5)
 note mn = 2.prems(6)
 note i-set-xxs = 2.prems(7)
 note d = 2.prems(8)
 note xxs-less-m = 2.prems(9)
 note ia = 2.prems(10)
 note imm = 2.prems(11)
 note D-ge0 = 2.prems(12)
 have D0: D≠0 using D-ge0 by simp
 have A: A ∈ carrier-mat (m+n) n using A' mn A-def by auto
 have xm: x < m  using 2.prems by auto
 have D1: D ·_m 1_m n ∈ carrier-mat n n by (simp add: mn)
 obtain p q u v d where pquvd: (p,q,u,v,d) = euclid-ext2 (A$$(a,0)) (A$$(x,0))
  by (metis prod-cases5)
 let ?reduce-ax = (reduce-abs a x D A)
```

have *reduce-ax*: *?reduce-ax* ∈ *carrier-mat* (*m* + *n*) *n*
by (*metis* (*no-types*, *lifting*) *A-def A′ add.comm-neutral append-rows-def carrier-matD carrier-mat-triv index-mat-four-block*(*2*,*3*) *index-one-mat*(*2*) *index-smult-mat*(*2*) *index-zero-mat*(*2*,*3*) *reduce-preserves-dimensions*)
show *?case*
proof (*cases i*=*x*)
case *True*
have *reduce-below-abs a* ((*x* # *xs*) @ [*m*]) *D A* $$ (*i*, *0*)
= *reduce-below-abs a* (*xs* @ [*m*]) *D* (*reduce-abs a x D A*) $$ (*i*, *0*)
by *auto*
also have ... = (*reduce-abs a x D A*) $$ (*i*, *0*)
proof (*rule reduce-below-abs-preserves-case-m*[*OF - a j - - mn - - - - - - D-ge0*])
let *?A′* = *mat-of-rows n* (*map* (*Matrix.row ?reduce-ax*) [*0*..<*m*])
show *reduce-abs a x D A* = *?A′* @$_r$ *D* ·$_m$ *1*$_m$ *n*
proof (*rule matrix-append-rows-eq-if-preserves*[*OF reduce-ax D1*])
show ∀ *i*∈{*m*..<*m* + *n*}. ∀ *ja*<*n*. *?reduce-ax* $$ (*i*, *ja*) = (*D* ·$_m$ *1*$_m$ *n*) $$ (*i* − *m*, *ja*)
proof (*rule*+)
fix *i ja* **assume** *i*: *i* ∈ {*m*..<*m* + *n*} **and** *ja*: *ja* < *n*
have *ja-dc*: *ja* < *dim-col A* **and** *i-dr*: *i* < *dim-row A* **using** *i ja A* **by** *auto*
have *i-not-a*: *i* ≠ *a* **using** *i a* **by** *auto*
have *i-not-x*: *i* ≠ *x* **using** *i xm* **by** *auto*
have *?reduce-ax* $$ (*i*,*ja*) = *A* $$ (*i*,*ja*)
unfolding *reduce-alt-def-not0*[*OF Aaj pquvd*] **using** *ja-dc i-dr i-not-a i-not-x* **by** *auto*
also have ... = (*if i* < *dim-row A′ then A′* $$(*i*,*ja*) *else* (*D* ·$_m$ (*1*$_m$ *n*))$$(*i*−*m*,*ja*))
by (*unfold A-def*, *rule append-rows-nth*[*OF A′ D1 - ja*], *insert A i-dr*, *simp*)
also have ... = (*D* ·$_m$ *1*$_m$ *n*) $$ (*i* − *m*, *ja*) **using** *i A′* **by** *auto*
finally show *?reduce-ax* $$ (*i*,*ja*) = (*D* ·$_m$ *1*$_m$ *n*) $$ (*i* − *m*, *ja*) **.**
qed
qed
show *distinct xs* **using** *d* **by** *auto*
show ∀ *x*∈*set xs*. *x* < *m* ∧ *a* < *x* **using** *xxs-less-m* **by** *auto*
show *reduce-abs a x D A* $$ (*a*, *0*) ≠ *0*
by (*rule reduce-not0*[*OF A - - - - Aaj*], *insert 2.prems*, *auto*)
show *?A′* ∈ *carrier-mat m n* **by** *auto*
show *i* ∉ *set xs* **using** *True d* **by** *auto*
show *i* ≠ *a* **using** *2.prems* **by** *blast*
show *i* < *m* + *n*
by (*simp add*: *True trans-less-add1 xm*)
show *i* ≠ *m* **by** (*simp add*: *True less-not-refl3 xm*)
qed
also have ... = *0* **unfolding** *True* **by** (*rule reduce-0*[*OF A - - - - Aaj*], *insert 2.prems*, *auto*)
also have ... = *reduce-below-abs a* (*x* # *xs*) *D A* $$ (*i*, *0*)
unfolding *True* **by** (*rule reduce-below-abs-0*[*symmetric*], *insert 2.prems*, *auto*)
finally show *?thesis* **.**

```
  next
    case False
    have reduce-below-abs a ((x # xs) @ [m]) D A $$ (i, 0)
      = reduce-below-abs a (xs@[m]) D (reduce-abs a x D A) $$ (i, 0)
      by auto
    also have ... = reduce-below-abs a xs D (reduce-abs a x D A) $$ (i, 0)
    proof (rule 2.hyps[OF - a j - - mn - - - ia imm D-ge0])
      let ?A' = mat-of-rows n (map (Matrix.row ?reduce-ax) [0..<m])
      show reduce-abs a x D A = ?A' @_r D ·_m 1_m n
        by (rule reduce-append-rows-eq[OF A' A-def a xm j Aaj])
      show i ∈ set xs using i-set-xxs False by auto
      show distinct xs using d by auto
      show ∀ x∈set xs. x < m ∧ a < x using xxs-less-m by auto
      show reduce-abs a x D A $$ (a, 0) ≠ 0
        by (rule reduce-not0[OF A - - j - Aaj], insert 2.prems, auto)
      show ?A' ∈ carrier-mat m n by auto
    qed
    also have ... = reduce-below-abs a (x # xs) D A $$ (i, 0) by auto
    finally show ?thesis .
  qed
qed


lemma reduce-below-0-case-m:
  assumes A': A' ∈ carrier-mat m n and a: a<m and j: 0<n
    and A-def: A = A' @_r (D ·_m (1_m n))
    and Aaj: A $$ (a,0) ≠ 0
    and mn: m≥n
  assumes i ∈ set (xs @ [m]) and distinct xs and ∀ x ∈ set xs. x < m ∧ a < x
  and D>0
  shows reduce-below a (xs @ [m]) D A $$ (i,0) = 0
proof (cases i=m)
  case True
  show ?thesis by (unfold True, rule reduce-below-0-case-m1, insert assms, auto)
next
  case False
  have reduce-below a (xs @ [m]) D A $$ (i,0) = reduce-below a (xs) D A $$ (i,0)
    by (rule reduce-below-preserves-case-m2[OF A' a j A-def Aaj mn], insert assms
False, auto)
  also have ... = 0 by (rule reduce-below-0, insert assms False, auto)
  finally show ?thesis .
qed


lemma reduce-below-abs-0-case-m:
  assumes A': A' ∈ carrier-mat m n and a: a<m and j: 0<n
    and A-def: A = A' @_r (D ·_m (1_m n))
    and Aaj: A $$ (a,0) ≠ 0
    and mn: m≥n
```

assumes $i \in$ *set* (*xs* @ [*m*]) **and** *distinct xs* **and** $\forall x \in$ *set xs*. $x < m \wedge a < x$
and *D*>*0*
shows *reduce-below-abs a* (*xs* @ [*m*]) *D A* $$ (*i*,*0*) = *0*
proof (*cases i*=*m*)
case *True*
show *?thesis* **by** (*unfold True*, *rule reduce-below-abs-0-case-m1*, *insert assms*, *auto*)
next
case *False*
have *reduce-below-abs a* (*xs* @ [*m*]) *D A* $$ (*i*,*0*) = *reduce-below-abs a* (*xs*) *D A* $$ (*i*,*0*)
by (*rule reduce-below-abs-preserves-case-m2*[*OF A' a j A-def Aaj mn*], *insert assms False*, *auto*)
also have ... = *0* **by** (*rule reduce-below-abs-0*, *insert assms False*, *auto*)
finally show *?thesis* **.**
qed

lemma *reduce-below-0-case-m-complete*:
assumes *A'*: *A'* ∈ *carrier-mat m n* **and** *a*: *0*<*m* **and** *j*: *0*<*n*
and *A-def*: *A* = *A'* @$_r$ (*D* ·$_m$ (*1*$_m$ *n*))
and *Aaj*: *A* $$ (*0*,*0*) ≠ *0*
and *mn*: *m*≥*n*
assumes *i-mn*: *i* < *m*+*n* **and** *d-xs*: *distinct xs* **and** *xs*: ∀ *x* ∈ *set xs*. *x* < *m* ∧ *0* < *x*
and *ia*: *i*≠*0*
and *xs-def*: *xs* = *filter* (λ*i*. *A* $$ (*i*,*0*) ≠ *0*) [*1*..<*dim-row A*]
and *D*: *D*>*0*
shows *reduce-below 0* (*xs* @ [*m*]) *D A* $$ (*i*,*0*) = *0*
proof (*cases i* ∈ *set* (*xs* @ [*m*]))
case *True*
show *?thesis* **by** (*rule reduce-below-0-case-m*[*OF A' a j A-def Aaj mn True d-xs xs D*])
next
case *False*
have *A*: *A* ∈ *carrier-mat* (*m*+*n*) *n* **using** *A' A-def* **by** *simp*
have *reduce-below 0* (*xs* @ [*m*]) *D A* $$ (*i*,*0*) = *A* $$ (*i*,*0*)
by (*rule reduce-below-preserves-case-m*[*OF A' a j A-def Aaj mn - - - - - - D*], *insert i-mn d-xs xs ia False*, *auto*)
also have ... = *0* **using** *False ia i-mn A* **unfolding** *xs-def* **by** *auto*
finally show *?thesis* **.**
qed

lemma *reduce-below-abs-0-case-m-complete*:
assumes *A'*: *A'* ∈ *carrier-mat m n* **and** *a*: *0*<*m* **and** *j*: *0*<*n*
and *A-def*: *A* = *A'* @$_r$ (*D* ·$_m$ (*1*$_m$ *n*))
and *Aaj*: *A* $$ (*0*,*0*) ≠ *0*

```
    and mn: m≥n
  assumes i-mn: i < m+n and d-xs: distinct xs and xs: ∀ x ∈ set xs. x < m ∧ 0
< x
    and ia: i≠0
  and xs-def: xs = filter (λi. A $$ (i,0) ≠ 0) [1..<dim-row A]
  and D: D>0
  shows reduce-below-abs 0 (xs @ [m]) D A $$ (i,0) = 0
proof (cases i ∈ set (xs @ [m]))
  case True
  show ?thesis by (rule reduce-below-abs-0-case-m[OF A' a j A-def Aaj mn True
d-xs xs D])
next
  case False
  have A: A ∈ carrier-mat (m+n) n using A' A-def by simp
  have reduce-below-abs 0 (xs @ [m]) D A $$ (i,0) = A $$ (i,0)
    by (rule reduce-below-abs-preserves-case-m[OF A' a j A-def Aaj mn - - - - - -
D],
       insert i-mn d-xs xs ia False, auto)
  also have ... = 0  using False ia i-mn A unfolding xs-def by auto
  finally show ?thesis .
qed



lemma reduce-below-invertible-mat-case-m:
  assumes A': A' ∈ carrier-mat m n and a: a<m and n0: 0<n
    and A-def: A = A' @_r (D ·_m (1_m n))
    and Aaj: A $$ (a,0) ≠ 0
    and mn: m≥n and distinct xs and ∀ x ∈ set xs. x < m ∧ a < x
    and D0: D>0
  shows (∃ P. invertible-mat P ∧ P ∈ carrier-mat (m+n) (m+n) ∧ reduce-below
a (xs@[m]) D A = P * A)
  using assms
proof (induct a xs D A arbitrary: A' rule: reduce-below.induct)
  case (1 a D A)
  obtain p q u v d where pquvd: (p,q,u,v,d) = euclid-ext2 (A$$(a,0)) (A$$(m,0))
    by (metis prod-cases5)
  have D: D ·_m (1_m n) : carrier-mat n n by auto
  note A' = 1.prems(1)
  note a = 1.prems(2)
  note j = 1.prems(3)
  note A-def = 1.prems(4)
  note Aaj = 1.prems(5)
  note mn = 1.prems(6)
  note D0 = 1.prems(9)
  have Am0-D: A $$ (m, 0) = D
  proof −
    have A $$ (m, 0) = (D ·_m (1_m n)) $$ (m−m,0)
      unfolding 1(4)
```

```
    by (meson 1(1) 1(3) D append-rows-nth3 less-add-same-cancel1 order.refl)
   also have ... = D by (simp add: n0)
   finally show ?thesis .
  qed
  have reduce-below a ([]@[m]) D A = reduce a m D A by auto
  let ?A = Matrix.mat (dim-row A) (dim-col A)
      (λ(i, k). if i = a then p * A $$ (a, k) + q * A $$ (m, k) else
        if i = m then u * A $$ (a, k) + v * A $$ (m, k) else A $$ (i, k))
  let ?xs = [1..<n]
  let ?ys = [1..<n]
  have ∃ P. invertible-mat P ∧ P ∈ carrier-mat (m + n) (m + n) ∧ reduce a m
D A = P * A
   by (rule reduce-invertible-mat-case-m[OF A' D a - A-def - Aaj mn n0 pquvd, of
?xs - - ?ys],
       insert a D0 Am0-D, auto)
  then show ?case by auto
next
  case (2 a x xs D A)
  note A' = 2.prems(1)
  note a = 2.prems(2)
  note n0 = 2.prems(3)
  note A-def = 2.prems(4)
  note Aaj = 2.prems(5)
  note mn = 2.prems(6)
  note d = 2.prems(7)
  note xxs-less-m = 2.prems(8)
  note D0 = 2.prems(9)
  have A: A ∈ carrier-mat (m+n) n using A' mn A-def by auto
  have xm: x < m  using 2.prems by auto
  have D1: D ·_m 1_m n ∈ carrier-mat n n by (simp add: mn)
  have Am0-D: A $$ (m, 0) = D
  proof −
   have A $$ (m, 0) = (D ·_m (1_m n)) $$ (m−m,0)
     unfolding 2(5)
    by (meson 2(2) 2(4) D1 append-rows-nth3 less-add-same-cancel1 verit-comp-simplify(2))
   also have ... = D by (simp add: n0)
   finally show ?thesis .
  qed
  obtain p q u v d where pquvd: (p,q,u,v,d) = euclid-ext2 (A$$(a,0)) (A$$(x,0))
   by (metis prod-cases5)
  let ?reduce-ax = reduce a x D A
  have reduce-ax: ?reduce-ax ∈ carrier-mat (m + n) n
   by (metis (no-types, lifting) 2 add.comm-neutral append-rows-def
       carrier-matD carrier-mat-triv index-mat-four-block(2,3)
     index-one-mat(2) index-smult-mat(2) index-zero-mat(2,3) reduce-preserves-dimensions)
  have h: (∃ P. invertible-mat P ∧ P ∈ carrier-mat (m + n) (m + n)
   ∧ reduce-below a (xs@[m]) D (reduce a x D A) = P * reduce a x D A)
  proof (rule 2.hyps[OF - a n0 - - ])
   let ?A' = mat-of-rows n (map (Matrix.row ?reduce-ax) [0..<m])
```

show *reduce a x D A = ?A′ @$_r$ D ·$_m$ 1$_m$ n*
by (*rule reduce-append-rows-eq*[*OF A′ A-def a xm n0 Aaj*])
show *reduce a x D A $$ (a, 0) ≠ 0*
by (*rule reduce-not0*[*OF A - - n0 - Aaj*], *insert 2.prems*, *auto*)
qed (*insert d xxs-less-m mn n0 D0*, *auto*)
from *this* **obtain** *P* **where** *inv-P*: *invertible-mat P* **and** *P*: *P ∈ carrier-mat* (*m + n*) (*m + n*)
and *rb-Pr*: *reduce-below a* (*xs@*[*m*]) *D* (*reduce a x D A*) = *P ∗ reduce a x D A*
by *blast*
have ∗: *reduce-below a* ((*x # xs*)@[*m*]) *D A* = *reduce-below a* (*xs@*[*m*]) *D* (*reduce a x D A*) **by** *simp*
have ∃ *Q*. *invertible-mat Q ∧ Q ∈ carrier-mat* (*m+n*) (*m+n*) ∧ (*reduce a x D A*) = *Q ∗ A*
by (*rule reduce-invertible-mat*[*OF A′ a n0 xm - A-def Aaj - mn D0*], *insert xxs-less-m*, *auto*)
from *this* **obtain** *Q* **where** *inv-Q*: *invertible-mat Q* **and** *Q*: *Q ∈ carrier-mat* (*m + n*) (*m + n*)
and *r-QA*: *reduce a x D A* = *Q ∗ A* **by** *blast*
have *invertible-mat* (*P∗Q*) **using** *inv-P inv-Q P Q invertible-mult-JNF* **by** *blast*
moreover have *P ∗ Q ∈ carrier-mat* (*m+n*) (*m+n*) **using** *P Q* **by** *auto*
moreover have *reduce-below a* ((*x # xs*)@[*m*]) *D A* = (*P∗Q*) ∗ *A*
by (*smt* (*verit*) *P Q ∗ assoc-mult-mat carrier-matD*(*1*) *carrier-mat-triv index-mult-mat*(*2*)
r-QA rb-Pr reduce-preserves-dimensions(*1*))
ultimately show *?case* **by** *blast*
qed

lemma *reduce-below-abs-invertible-mat-case-m*:
assumes *A′*: *A′ ∈ carrier-mat m n* **and** *a*: *a<m* **and** *n0*: *0<n*
and *A-def*: *A* = *A′ @$_r$* (*D ·$_m$* (*1$_m$ n*))
and *Aaj*: *A $$* (*a*,*0*) ≠ *0*
and *mn*: *m≥n* **and** *distinct xs* **and** ∀ *x ∈ set xs*. *x < m ∧ a < x*
and *D0*: *D>0*
shows (∃ *P*. *invertible-mat P ∧ P ∈ carrier-mat* (*m+n*) (*m+n*) ∧ *reduce-below-abs a* (*xs@*[*m*]) *D A* = *P ∗ A*)
using *assms*
proof (*induct a xs D A arbitrary*: *A′ rule*: *reduce-below-abs.induct*)
case (*1 a D A*)
obtain *p q u v d* **where** *pquvd*: (*p*,*q*,*u*,*v*,*d*) = *euclid-ext2* (*A$$*(*a*,*0*)) (*A$$*(*m*,*0*))
by (*metis prod-cases5*)
have *D*: *D ·$_m$* (*1$_m$ n*) : *carrier-mat n n* **by** *auto*
note *A′* = *1.prems*(*1*)
note *a* = *1.prems*(*2*)
note *j* = *1.prems*(*3*)
note *A-def* = *1.prems*(*4*)

```
  note Aaj = 1.prems(5)
  note mn = 1.prems(6)
  note D0 = 1.prems(9)
  have Am0-D: A $$ (m, 0) = D
  proof −
    have A $$ (m, 0) = (D ·_m (1_m n)) $$ (m−m,0)
      unfolding 1(4)
      by (meson 1(1) 1(3) D append-rows-nth3 le-refl less-add-same-cancel1)
    also have ... = D by (simp add: n0)
    finally show ?thesis .
  qed
  have reduce-below-abs a ([]@[m]) D A = reduce-abs a m D A by auto
  let ?A = Matrix.mat (dim-row A) (dim-col A)
      (λ(i, k). if i = a then p * A $$ (a, k) + q * A $$ (m, k) else
        if i = m then u * A $$ (a, k) + v * A $$ (m, k) else A $$ (i, k))
  let ?xs = filter (λi. D < |?A $$ (a, i)|) [0..<n]
  let ?ys = filter (λi. D < |?A $$ (m, i)|) [0..<n]
  have ∃ P. invertible-mat P ∧ P ∈ carrier-mat (m + n) (m + n) ∧ reduce-abs a
m D A = P * A
    by (rule reduce-abs-invertible-mat-case-m[OF A' D a - A-def - Aaj mn n0 pquvd,
of ?xs - - ?ys],
        insert a D0 Am0-D, auto)
  then show ?case by auto
next
  case (2 a x xs D A)
  note A' = 2.prems(1)
  note a = 2.prems(2)
  note n0 = 2.prems(3)
  note A-def = 2.prems(4)
  note Aaj = 2.prems(5)
  note mn = 2.prems(6)
  note d = 2.prems(7)
  note xxs-less-m = 2.prems(8)
  note D0 = 2.prems(9)
  have A: A ∈ carrier-mat (m+n) n using A' mn A-def by auto
  have xm: x < m  using 2.prems by auto
  have D1: D ·_m 1_m n ∈ carrier-mat n n by (simp add: mn)
  have Am0-D: A $$ (m, 0) = D
  proof −
    have A $$ (m, 0) = (D ·_m (1_m n)) $$ (m−m,0)
      unfolding 2(5)
      by (meson 2(2) 2(4) D1 append-rows-nth3 less-add-same-cancel1 order-refl)
    also have ... = D by (simp add: n0)
    finally show ?thesis .
  qed
  obtain p q u v d where pquvd: (p,q,u,v,d) = euclid-ext2 (A$$(a,0)) (A$$(x,0))
    by (metis prod-cases5)
  let ?reduce-ax = reduce-abs a x D A
  have reduce-ax: ?reduce-ax ∈ carrier-mat (m + n) n
```

by (*metis* (*no-types*, *lifting*) *2 add.comm-neutral append-rows-def*
carrier-matD carrier-mat-triv index-mat-four-block(*2*,*3*)
index-one-mat(*2*) *index-smult-mat*(*2*) *index-zero-mat*(*2*,*3*) *reduce-preserves-dimensions*)
have *h*: (∃ *P*. *invertible-mat P* ∧ *P* ∈ *carrier-mat* (*m* + *n*) (*m* + *n*)
∧ *reduce-below-abs a* (*xs*@[*m*]) *D* (*reduce-abs a x D A*) = *P* ∗ *reduce-abs a x D A*)
proof (*rule 2.hyps*[*OF - a n0 - -*])
let *?A′* = *mat-of-rows n* (*map* (*Matrix.row ?reduce-ax*) [*0*..<*m*])
show *reduce-abs a x D A* = *?A′* @$_r$ *D* ·$_m$ *1*$_m$ *n*
by (*rule reduce-append-rows-eq*[*OF A′ A-def a xm n0 Aaj*])
show *reduce-abs a x D A* $$ (*a*, *0*) ≠ *0*
by (*rule reduce-not0*[*OF A - - n0 - Aaj*], *insert 2.prems*, *auto*)
qed (*insert d xxs-less-m mn n0 D0*, *auto*)
from *this* **obtain** *P* **where** *inv-P*: *invertible-mat P* **and** *P*: *P* ∈ *carrier-mat* (*m* + *n*) (*m* + *n*)
and *rb-Pr*: *reduce-below-abs a* (*xs*@[*m*]) *D* (*reduce-abs a x D A*) = *P* ∗ *reduce-abs a x D A* **by** *blast*
have ∗: *reduce-below-abs a* ((*x* # *xs*)@[*m*]) *D A* = *reduce-below-abs a* (*xs*@[*m*]) *D* (*reduce-abs a x D A*) **by** *simp*
have ∃ *Q*. *invertible-mat Q* ∧ *Q* ∈ *carrier-mat* (*m*+*n*) (*m*+*n*) ∧ (*reduce-abs a x D A*) = *Q* ∗ *A*
by (*rule reduce-abs-invertible-mat*[*OF A′ a n0 xm - A-def Aaj - mn D0*], *insert xxs-less-m*, *auto*)
from *this* **obtain** *Q* **where** *inv-Q*: *invertible-mat Q* **and** *Q*: *Q* ∈ *carrier-mat* (*m* + *n*) (*m* + *n*)
and *r-QA*: *reduce-abs a x D A* = *Q* ∗ *A* **by** *blast*
have *invertible-mat* (*P*∗*Q*) **using** *inv-P inv-Q P Q invertible-mult-JNF* **by** *blast*
moreover have *P* ∗ *Q* ∈ *carrier-mat* (*m*+*n*) (*m*+*n*) **using** *P Q* **by** *auto*
moreover have *reduce-below-abs a* ((*x* # *xs*)@[*m*]) *D A* = (*P*∗*Q*) ∗ *A*
by (*smt* (*verit*) *P Q* ∗ *assoc-mult-mat carrier-matD*(*1*) *carrier-mat-triv index-mult-mat*(*2*)
r-QA rb-Pr reduce-preserves-dimensions(*3*))
ultimately show *?case* **by** *blast*
qed

end

hide-const (**open**) *C*

This lemma will be very important, since it will allow us to prove that the output matrix is in echelon form.

lemma *echelon-form-four-block-mat*:
assumes *A*: *A* ∈ *carrier-mat 1 1*
and *B*: *B* ∈ *carrier-mat 1* (*n*−*1*)
and *D*: *D* ∈ *carrier-mat* (*m*−*1*) (*n*−*1*)
and *H-def*: *H* = *four-block-mat A B* (*0*$_m$ (*m*−*1*) *1*) *D*
and *A00*: *A* $$ (*0*,*0*) ≠ *0*
and *e-D*: *echelon-form-JNF D*
and *m*: *m*>*0* **and** *n*: *n*>*0*

shows *echelon-form-JNF H*
proof (*rule echelon-form-JNF-intro*)
have *H*: *H* ∈ *carrier-mat m n*
by (*metis H-def Num.numeral-nat*(*7*) *A D m n carrier-matD carrier-mat-triv index-mat-four-block*(*2*,*3*) *linordered-semidom-class.add-diff-inverse not-less-eq*)
have *Hij-Dij*: *H* $$ (*i*+*1*,*j*+*1*) = *D* $$ (*i*,*j*) **if** *i*: *i*<*m*−*1* **and** *j*: *j*<*n*−*1* **for** *i j*
proof −
have *H* $$ (*i*+*1*,*j*+*1*) = (*if* (*i*+*1*) < *dim-row A then if* (*j*+*1*) < *dim-col A then A* $$ ((*i*+*1*), (*j*+*1*))
else B $$ ((*i*+*1*), (*j*+*1*) − *dim-col A*) *else if* (*j*+*1*) < *dim-col A then*
(*0*$_m$ (*m*−*1*) *1*) $$ ((*i*+*1*) − *dim-row A*, (*j*+*1*)) *else D* $$ ((*i*+*1*) − *dim-row A*, (*j*+*1*) − *dim-col A*))
unfolding *H-def* **by** (*rule index-mat-four-block*, *insert A D i j*, *auto*)
also have ... = *D* $$ ((*i*+*1*) − *dim-row A*, (*j*+*1*) − *dim-col A*) **using** *A D i j B m n* **by** *auto*
also have ... = *D* $$ (*i*,*j*) **using** *A* **by** *auto*
finally show *?thesis* **.**
qed
have *Hij-Dij'*: *H* $$ (*i*,*j*) = *D* $$ (*i*−*1*,*j*−*1*)
if *i*: *i*<*m* **and** *j*: *j*<*n* **and** *i0*: *i*>*0* **and** *j0*: *j*>*0* **for** *i j*
by (*metis* (*no-types*, *lifting*) *H H-def Num.numeral-nat*(*7*) *A carrier-matD index-mat-four-block less-Suc0 less-not-refl3 i j i0 j0*)
have *Hi0*: *H*$$(*i*,*0*) = *0* **if** *i*: *i*∈{*1*..<*m*} **for** *i*
proof −
have *H* $$ (*i*,*0*) = (*if i* < *dim-row A then if 0* < *dim-col A then A* $$ (*i*, *0*)
else B $$ (*i*, *0* − *dim-col A*) *else if 0* < *dim-col A then*
(*0*$_m$ (*m*−*1*) *1*) $$ (*i* − *dim-row A*, *0*) *else D* $$ (*i* − *dim-row A*, *0* − *dim-col A*))
unfolding *H-def* **by** (*rule index-mat-four-block*, *insert A D i*, *auto*)
also have ... = (*0*$_m$ (*m*−*1*) *1*) $$ (*i* − *dim-row A*, *0*) **using** *A D i m n* **by** *auto*
also have ... = *0* **using** *i A n* **by** *auto*
finally show *?thesis* **.**
qed
have *A00-H00*: *A* $$ (*0*,*0*) = *H* $$ (*0*,*0*) **unfolding** *H-def* **using** *A* **by** *auto*
have *is-zero-row-JNF j H* **if** *zero-iH*: *is-zero-row-JNF i H* **and** *ij*: *i* < *j* **and** *j*: *j* < *dim-row H*
for *i j*
proof −
have ¬ *is-zero-row-JNF 0 H* **unfolding** *is-zero-row-JNF-def* **using** *m n H A00 A00-H00* **by** *auto*
hence *i-not0*: *i*≠*0* **using** *zero-iH* **by** *meson*
have *is-zero-row-JNF* (*i*−*1*) *D* **using** *zero-iH i-not0 Hij-Dij m n D H* **unfolding** *is-zero-row-JNF-def*
by (*auto*, *smt* (*verit*) *Suc-leI carrier-matD*(*1*) *le-add-diff-inverse2 Hij-Dij One-nat-def Suc-pred carrier-matD*(*1*) *j le-add-diff-inverse2*
less-diff-conv less-imp-add-positive plus-1-eq-Suc that(*2*) *trans-less-add1*)
hence *is-zero-row-JNF* (*j*−*1*) *D* **using** *ij e-D D j m i-not0* **unfolding** *echelon-form-JNF-def*

by (*auto*, *smt* (*verit*) *H Nat.lessE Suc-pred carrier-matD*(*1*) *diff-Suc-1 diff-Suc-less order.strict-trans*)
thus *?thesis*
by (*smt* (*verit*) *A H H-def Hi0 D atLeastLessThan-iff carrier-matD index-mat-four-block*(*1*)
is-zero-row-JNF-def le-add1 less-one linordered-semidom-class.add-diff-inverse not-less-eq
plus-1-eq-Suc ij j zero-order(*3*))
qed
thus $\forall$ *i*<*dim-row H*. *is-zero-row-JNF i H* $\longrightarrow$ $\neg$ ($\exists$ *j*<*dim-row H*. *i* < *j* $\wedge$ $\neg$ *is-zero-row-JNF j H*)
by *blast*
have (*LEAST n*. *H* $$ (*i*, *n*) $\neq$ *0*) < (*LEAST n*. *H* $$ (*j*, *n*) $\neq$ *0*)
if *ij*: *i* < *j* **and** *j*: *j* < *dim-row H* **and** *not-zero-iH*: $\neg$ *is-zero-row-JNF i H*
and *not-zero-jH*: $\neg$ *is-zero-row-JNF j H* **for** *i j*
proof (*cases i* = *0*)
case *True*
have (*LEAST n*. *H* $$ (*i*, *n*) $\neq$ *0*) = *0* **unfolding** *True* **using** *A00-H00 A00* **by** *auto*
then show *?thesis*
by (*metis* (*mono-tags*) *H Hi0 LeastI True atLeastLessThan-iff carrier-matD*(*1*)
is-zero-row-JNF-def leI less-one not-gr0 ij j not-zero-jH)
next
case *False* **note** *i-not0* = *False*
let *?least-H* = (*LEAST n*. *H* $$ (*i*, *n*) $\neq$ *0*)
let *?least-Hj* = (*LEAST n*. *H* $$ (*j*, *n*) $\neq$ *0*)

have *least-not0*: (*LEAST n*. *H* $$ (*i*, *n*) $\neq$ *0*) $\neq$ *0*
proof $-$
have ‹*dim-row H* = *m*›
using *H* **by** *auto*
with ‹*i* < *j*› ‹*j* < *dim-row H*› **have** ‹*i* < *m*›
by *simp*
then have ‹*H* $$ (*i*, *0*) = *0*›
using *i-not0* **by** (*auto simp add*: *Suc-le-eq intro*: *Hi0*)
moreover from *is-zero-row-JNF-def* [*of i H*] *not-zero-iH*
obtain *n* **where** ‹*H* $$ (*i*, *n*) $\neq$ *0*›
by *blast*
ultimately show *?thesis*
by (*metis* (*mono-tags*, *lifting*) *LeastI*)
qed
have *least-not0j*: (*LEAST n*. *H* $$ (*j*, *n*) $\neq$ *0*) $\neq$ *0*
proof $-$
have $\exists$ *n*. *H* $$ (*j*, *0*) = *0* $\wedge$ *H* $$ (*j*, *n*) $\neq$ *0*
by (*metis* (*no-types*) *H Hi0 LeastI-ex Num.numeral-nat*(*7*) *atLeastLessThan-iff carrier-matD*(*1*)
is-zero-row-JNF-def linorder-neqE-nat not-gr0 not-less-Least not-less-eq order-trans-rules(*19*)

ij j not-zero-jH wellorder-Least-lemma(2))
then show *?thesis*
by (*metis* (*mono-tags*, *lifting*) *LeastI-ex*)
qed
have *least-n*: *?least-H<n*
by (*smt* (*verit*) *H carrier-matD(2) dual-order.strict-trans is-zero-row-JNF-def*

not-less-Least not-less-iff-gr-or-eq not-zero-iH)
have *Hil*: *H $$ (i,?least-H) ≠ 0* **and** *ln'*:(∀ *n'*. (*H $$ (i, n') ≠ 0*) ⟶ *?least-H ≤ n'*)
by (*metis* (*mono-tags*, *lifting*) *is-zero-row-JNF-def that(3) wellorder-Least-lemma*)+
have *Hil-Dil*: *H $$ (i,?least-H) = D $$ (i−1,?least-H − 1)*
proof −
have *H $$ (i,?least-H) = (if i < dim-row A then if ?least-H < dim-col A then A $$ (i, ?least-H)*
else B $$ (i, ?least-H − dim-col A) else if ?least-H < dim-col A then
(0$_m$ (m−1) 1) $$ (i − dim-row A, ?least-H) else D $$ (i − dim-row A, ?least-H − dim-col A))
unfolding *H-def*
by (*rule index-mat-four-block*, *insert False j ij H A D n least-n*, *auto simp add*: *H-def*)
also have *... = D $$ (i − 1, ?least-H − 1)*
using *False j ij H A D n least-n B Hi0 Hil* **by** *auto*
finally show *?thesis* **.**
qed
have *not-zero-iD*: ¬ *is-zero-row-JNF (i−1) D*
by (*metis* (*no-types*, *lifting*) *Hil Hil-Dil D carrier-matD(2) is-zero-row-JNF-def le-add1*
le-add-diff-inverse2 least-n least-not0 less-diff-conv less-one
linordered-semidom-class.add-diff-inverse)
have *not-zero-jD*: ¬ *is-zero-row-JNF (j−1) D*
by (*smt* (*verit*) *H Hij-Dij' One-nat-def Suc-pred D m carrier-matD diff-Suc-1 ij is-zero-row-JNF-def j*
least-not0j less-Suc0 less-Suc-eq-0-disj less-one neq0-conv not-less-Least not-less-eq
plus-1-eq-Suc not-zero-jH zero-order(3))
have *?least-H − 1 = (LEAST n. D $$ (i−1, n) ≠ 0 ∧ n<dim-col D)*
proof (*rule Least-equality*[*symmetric*], *rule*)
show *D $$ (i − 1, ?least-H − 1) ≠ 0* **using** *Hil Hil-Dil* **by** *auto*
show (*LEAST n. H $$ (i, n) ≠ 0*) − *1 < dim-col D* **using** *least-n least-not0 H D n* **by** *auto*
fix *n'* **assume** *D $$ (i − 1, n') ≠ 0 ∧ n' < dim-col D*
hence *Di1n'1*: *D $$ (i − 1, n') ≠ 0* **and** *n'*: *n' < dim-col D* **by** *auto*
have (*LEAST n. H $$ (i, n) ≠ 0*) ≤ *n' + 1*
proof (*rule Least-le*)
have *H $$ (i, n'+1) = D $$ (i −1, (n'+1)−1)*
by (*rule Hij-Dij'*, *insert i-not0 False H A ij j n' D*, *auto*)
thus *Hin'*: *H $$ (i, n'+1) ≠ 0* **using** *False Di1n'1 Hij-Dij'* **by** *auto*
qed

thus $(LEAST\ n.\ H\ \$\$\ (i,\ n) \neq 0) - 1 \leq n'$ **using** *least-not0* **by** *auto*
qed
also have $... = (LEAST\ n.\ D\ \$\$\ (i-1,\ n) \neq 0)$
proof (*rule Least-equality*)
have $D\ \$\$\ (i - 1,\ LEAST\ n.\ D\ \$\$\ (i - 1,\ n) \neq 0) \neq 0$
by (*metis* (*mono-tags*, *lifting*) *Hil Hil-Dil LeastI-ex*)
moreover have *leastD*: $(LEAST\ n.\ D\ \$\$\ (i - 1,\ n) \neq 0) < dim\text{-}col\ D$
by (*smt* (*verit*) *dual-order.strict-trans is-zero-row-JNF-def linorder-neqE-nat not-less-Least not-zero-iD*)
ultimately show $D\ \$\$\ (i - 1,\ LEAST\ n.\ D\ \$\$\ (i - 1,\ n) \neq 0) \neq 0 \wedge (LEAST\ n.\ D\ \$\$\ (i - 1,\ n) \neq 0) < dim\text{-}col\ D$ **by** *simp*
fix y **assume** $D\ \$\$\ (i - 1,\ y) \neq 0 \wedge y < dim\text{-}col\ D$
thus $(LEAST\ n.\ D\ \$\$\ (i - 1,\ n) \neq 0) \leq y$ **by** (*meson wellorder-Least-lemma*(2))
qed
finally have *leastHi-eq*: $?least\text{-}H - 1 = (LEAST\ n.\ D\ \$\$\ (i-1,\ n) \neq 0)$ **.**
have *least-nj*: $?least\text{-}Hj<n$
by (*smt* (*verit*) *H carrier-matD*(2) *dual-order.strict-trans is-zero-row-JNF-def not-less-Least not-less-iff-gr-or-eq not-zero-jH*)
have *Hjl*: $H\ \$\$\ (j,?least\text{-}Hj) \neq 0$ **and** *ln'*:$(\forall n'.\ (H\ \$\$\ (j,\ n') \neq 0) \longrightarrow ?least\text{-}Hj \leq n')$
by (*metis* (*mono-tags*, *lifting*) *is-zero-row-JNF-def not-zero-jH wellorder-Least-lemma*)+
have *Hjl-Djl*: $H\ \$\$\ (j,?least\text{-}Hj) = D\ \$\$\ (j-1,?least\text{-}Hj - 1)$
proof −
have $H\ \$\$\ (j,?least\text{-}Hj) = (if\ j < dim\text{-}row\ A\ then\ if\ ?least\text{-}Hj < dim\text{-}col\ A\ then\ A\ \$\$\ (j,\ ?least\text{-}Hj)\ else\ B\ \$\$\ (j,\ ?least\text{-}Hj - dim\text{-}col\ A)\ else\ if\ ?least\text{-}Hj < dim\text{-}col\ A\ then\ (0_m\ (m-1)\ 1)\ \$\$\ (j - dim\text{-}row\ A,\ ?least\text{-}Hj)\ else\ D\ \$\$\ (j - dim\text{-}row\ A,\ ?least\text{-}Hj - dim\text{-}col\ A))$
unfolding *H-def*
by (*rule index-mat-four-block*, *insert False j ij H A D n least-nj*, *auto simp add*: *H-def*)
also have $... = D\ \$\$\ (j - 1,\ ?least\text{-}Hj - 1)$
using *False j ij H A D n least-n B Hi0 Hjl* **by** *auto*
finally show *?thesis* **.**
qed
have $(LEAST\ n.\ H\ \$\$\ (j,\ n) \neq 0) - 1 = (LEAST\ n.\ D\ \$\$\ (j-1,\ n) \neq 0 \wedge n<dim\text{-}col\ D)$
proof (*rule Least-equality*[*symmetric*], *rule*)
show $D\ \$\$\ (j - 1,\ ?least\text{-}Hj - 1) \neq 0$ **using** *Hil Hil-Dil*
by (*smt* (*verit*) *H Hij-Dij' LeastI-ex carrier-matD is-zero-row-JNF-def j least-not0j linorder-neqE-nat not-gr0 not-less-Least order.strict-trans ij not-zero-jH*)
show $(LEAST\ n.\ H\ \$\$\ (j,\ n) \neq 0) - 1 < dim\text{-}col\ D$ **using** *least-nj least-not0j H D n* **by** *auto*
fix n' **assume** $D\ \$\$\ (j - 1,\ n') \neq 0 \wedge n' < dim\text{-}col\ D$
hence *Di1n'1*: $D\ \$\$\ (j - 1,\ n') \neq 0$ **and** n': $n' < dim\text{-}col\ D$ **by** *auto*
have $(LEAST\ n.\ H\ \$\$\ (j,\ n) \neq 0) \leq n' + 1$
proof (*rule Least-le*)

have $H\ \$\$\ (j,\ n'+1) = D\ \$\$\ (j-1,\ (n'+1)-1)$
by (*rule Hij-Dij'*, *insert i-not0 False H A ij j n' D*, *auto*)
thus *Hin'*: $H\ \$\$\ (j,\ n'+1) \neq 0$ **using** *False Di1n'1 Hij-Dij'* **by** *auto*
qed
thus $(LEAST\ n.\ H\ \$\$\ (j,\ n) \neq 0) - 1 \leq n'$ **using** *least-not0* **by** *auto*
qed
also have $... = (LEAST\ n.\ D\ \$\$\ (j-1,\ n) \neq 0)$
proof (*rule Least-equality*)
have $D\ \$\$\ (j-1,\ LEAST\ n.\ D\ \$\$\ (j-1,\ n) \neq 0) \neq 0$
by (*metis* (*mono-tags*, *lifting*) *Hjl Hjl-Djl LeastI-ex*)
moreover have *leastD*: $(LEAST\ n.\ D\ \$\$\ (j-1,\ n) \neq 0) < dim\text{-}col\ D$
by (*smt* (*verit*) *dual-order.strict-trans is-zero-row-JNF-def linorder-neqE-nat not-less-Least not-zero-jD*)
ultimately show $D\ \$\$\ (j-1,\ LEAST\ n.\ D\ \$\$\ (j-1,\ n) \neq 0) \neq 0 \wedge (LEAST\ n.\ D\ \$\$\ (j-1,\ n) \neq 0) < dim\text{-}col\ D$ **by** *simp*
fix y **assume** $D\ \$\$\ (j-1,\ y) \neq 0 \wedge y < dim\text{-}col\ D$
thus $(LEAST\ n.\ D\ \$\$\ (j-1,\ n) \neq 0) \leq y$ **by** (*meson wellorder-Least-lemma*(*2*))
qed
finally have *leastHj-eq*: $(LEAST\ n.\ H\ \$\$\ (j,\ n) \neq 0) - 1 = (LEAST\ n.\ D\ \$\$\ (j-1,\ n) \neq 0)$ **.**
have *ij'*: $i-1 < j-1$ **using** *ij False* **by** *auto*
have $j-1 < dim\text{-}row\ D$ **using** *D H ij j* **by** *auto*
hence $(LEAST\ n.\ D\ \$\$\ (i-1,\ n) \neq 0) < (LEAST\ n.\ D\ \$\$\ (j-1,\ n) \neq 0)$
using *e-D echelon-form-JNF-def ij' not-zero-jD order.strict-trans* **by** *blast*
thus *?thesis* **using** *leastHj-eq leastHi-eq* **by** *auto*
qed
thus $\forall\, i\ j.\ i < j \wedge j < dim\text{-}row\ H \wedge \neg\ is\text{-}zero\text{-}row\text{-}JNF\ i\ H \wedge \neg\ is\text{-}zero\text{-}row\text{-}JNF\ j\ H \longrightarrow (LEAST\ n.\ H\ \$\$\ (i,\ n) \neq 0) < (LEAST\ n.\ H\ \$\$\ (j,\ n) \neq 0)$ **by** *blast*
qed

context *mod-operation*
begin

lemma *reduce-below*:
assumes $A \in carrier\text{-}mat\ m\ n$
shows $reduce\text{-}below\ a\ xs\ D\ A \in carrier\text{-}mat\ m\ n$
using *assms*
by (*induct a xs D A rule*: *reduce-below.induct*, *auto simp add*: *Let-def euclid-ext2-def*)

lemma *reduce-below-preserves-dimensions*:
shows [*simp*]: $dim\text{-}row\ (reduce\text{-}below\ a\ xs\ D\ A) = dim\text{-}row\ A$
and [*simp*]: $dim\text{-}col\ (reduce\text{-}below\ a\ xs\ D\ A) = dim\text{-}col\ A$
using *reduce-below*[*of A dim-row A dim-col A*] **by** *auto*

lemma *reduce-below-abs*:

assumes *A* ∈ *carrier-mat m n*
shows *reduce-below-abs a xs D A* ∈ *carrier-mat m n*
using *assms*
by (*induct a xs D A rule*: *reduce-below-abs.induct*, *auto simp add*: *Let-def euclid-ext2-def*)

lemma *reduce-below-abs-preserves-dimensions*:
shows [*simp*]: *dim-row* (*reduce-below-abs a xs D A*) = *dim-row A*
and [*simp*]: *dim-col* (*reduce-below-abs a xs D A*) = *dim-col A*
using *reduce-below-abs*[*of A dim-row A dim-col A*] **by** *auto*

lemma *FindPreHNF-1xn*:
assumes *A*: *A* ∈ *carrier-mat m n* **and** *m*<2 ∨ *n* = 0
shows *FindPreHNF abs-flag D A* ∈ *carrier-mat m n* **using** *assms* **by** *auto*

lemma *FindPreHNF-mx1*:
assumes *A*: *A* ∈ *carrier-mat m n* **and** *m*≥2 **and** *n* ≠ 0 *n*<2
shows *FindPreHNF abs-flag D A* ∈ *carrier-mat m n*
proof (*cases abs-flag*)
case *True*
let *?nz* = (*filter* (λ*i*. *A* $$ (*i*, 0) ≠ 0) [1..<*m*])
have *FindPreHNF abs-flag D A* = (*let non-zero-positions* = *filter* (λ*i*. *A* $$ (*i*, 0) ≠ 0) [*Suc* 0..<*m*]
in reduce-below-abs 0 non-zero-positions D (*if A* $$ (0, 0) ≠ 0 *then A else*
let i = *non-zero-positions* ! 0 *in swaprows 0 i A*))
using *assms True* **by** *auto*
also have ... = *reduce-below-abs 0 ?nz D* (*if A* $$ (0, 0) ≠ 0 *then A*
else let i = *?nz* ! 0 *in swaprows 0 i A*) **unfolding** *Let-def* **by** *auto*
also have ... ∈ *carrier-mat m n* **using** *A* **by** *auto*
finally show *?thesis* .
next
case *False*
let *?nz* = (*filter* (λ*i*. *A* $$ (*i*, 0) ≠ 0) [1..<*m*])
have *FindPreHNF abs-flag D A* = (*let non-zero-positions* = *filter* (λ*i*. *A* $$ (*i*, 0) ≠ 0) [*Suc* 0..<*m*]
in reduce-below 0 non-zero-positions D (*if A* $$ (0, 0) ≠ 0 *then A else*
let i = *non-zero-positions* ! 0 *in swaprows 0 i A*))
using *assms False* **by** *auto*
also have ... = *reduce-below 0 ?nz D* (*if A* $$ (0, 0) ≠ 0 *then A*
else let i = *?nz* ! 0 *in swaprows 0 i A*) **unfolding** *Let-def* **by** *auto*
also have ... ∈ *carrier-mat m n* **using** *A* **by** *auto*
finally show *?thesis* .
qed

lemma *FindPreHNF-mxn2*:
assumes *A*: *A* ∈ *carrier-mat m n* **and** *m*: *m*≥2 **and** *n*: *n*≥2
shows *FindPreHNF abs-flag D A* ∈ *carrier-mat m n*

using *assms*
proof (*induct abs-flag D A arbitrary*: *m n rule*: *FindPreHNF.induct*)
case (*1 abs-flag D A*)
note *A* = *1.prems*(*1*)
note *m* = *1.prems*(*2*)
note *n* = *1.prems*(*3*)
define *non-zero-positions* **where** *non-zero-positions* = *filter* (λ*i*. *A* $$ (*i*,*0*) ≠ *0*) [*1*..<*dim-row A*]
define *A'* **where** *A'* = (*if A* $$ (*0*, *0*) ≠ *0 then A else let i* = *non-zero-positions* ! *0 in swaprows 0 i A*)
define *Reduce* **where** [*simp*]: *Reduce* = (*if abs-flag then reduce-below-abs else reduce-below*)
obtain *A'-UL A'-UR A'-DL A'-DR* **where** *A'-split*: (*A'-UL*, *A'-UR*, *A'-DL*, *A'-DR*)
= *split-block* (*Reduce 0 non-zero-positions D* (*make-first-column-positive A'*)) *1 1*
by (*metis prod-cases4*)
define *sub-PreHNF* **where** *sub-PreHNF* = *FindPreHNF abs-flag D A'-DR*
have *A'*: *A'* ∈ *carrier-mat m n* **unfolding** *A'-def* **using** *A* **by** *auto*
have *A'-DR*: *A'-DR* ∈ *carrier-mat* (*m* −*1*) (*n*−*1*)
by (*cases abs-flag*; *rule split-block*(*4*)[*OF A'-split*[*symmetric*]], *insert Reduce-def A A' m n*, *auto*)
have *sub-PreHNF*: *sub-PreHNF* ∈ *carrier-mat* (*m* − *1*) (*n*−*1*)
proof (*cases m*−*1*<*2*)
case *True*
show *?thesis* **using** *A'-DR True* **unfolding** *sub-PreHNF-def* **by** *auto*
next
case *False* **note** *m'* = *False*
show *?thesis*
proof (*cases n*−*1*<*2*)
case *True*
show *?thesis*
unfolding *sub-PreHNF-def* **by** (*rule FindPreHNF-mx1*[*OF A'-DR - - True*], *insert n m'*, *auto*)
next
case *False*
show *?thesis*
by (*unfold sub-PreHNF-def*, *rule 1.hyps*
[*of m n*, *OF - - - non-zero-positions-def A'-def Reduce-def - A'-split - - - A'-DR*],
insert A False n m' Reduce-def, *auto*)
qed
qed
have *A'-UL*: *A'-UL* ∈ *carrier-mat 1 1*
by (*cases abs-flag*; *rule split-block*(*1*)[*OF A'-split*[*symmetric*], *of m*−*1 n*−*1*], *insert n m A'*, *auto*)
have *A'-UR*: *A'-UR* ∈ *carrier-mat 1* (*n*−*1*)
by (*cases abs-flag*; *rule split-block*(*2*)[*OF A'-split*[*symmetric*], *of m*−*1*], *insert n m A'*, *auto*)

have *A′-DL*: *A′-DL* ∈ *carrier-mat* (*m* − *1*) *1*
by (*cases abs-flag*; *rule split-block*(*3*)[*OF A′-split*[*symmetric*], *of* - *n*−*1*], *insert n m A′*, *auto*)
have ∗: (*dim-col A* = *0*) = *False* **using** *1*(*2*−) **by** *auto*
have *FindPreHNF-as-fbm*: *FindPreHNF abs-flag D A* = *four-block-mat A′-UL A′-UR A′-DL sub-PreHNF*
unfolding *FindPreHNF.simps*[*of abs-flag D A*] **using** *A′-split m n A*
unfolding *Let-def sub-PreHNF-def A′-def non-zero-positions-def* ∗
apply (*cases abs-flag*)
by (*smt* (*verit*) *Reduce-def carrier-matD*(*1*) *carrier-matD*(*2*) *linorder-not-less prod.simps*(*2*))+
also have ... ∈ *carrier-mat m n*
by (*smt* (*verit*) *m A′-UL One-nat-def add.commute carrier-matD carrier-mat-triv index-mat-four-block*(*2*,*3*)
le-add-diff-inverse2 le-eq-less-or-eq lessI n nat-SN.compat numerals(*2*) *sub-PreHNF*)
finally show *?case* .
qed

lemma *FindPreHNF*:
assumes *A*: *A* ∈ *carrier-mat m n*
shows *FindPreHNF abs-flag D A* ∈ *carrier-mat m n*
using *assms FindPreHNF-mxn2*[*OF A*] *FindPreHNF-mx1*[*OF A*] *FindPreHNF-1xn*[*OF A*]
using *linorder-not-less* **by** *blast*
end

lemma *make-first-column-positive-append-id*:
assumes *A′*: *A′* ∈ *carrier-mat m n*
and *A-def*: *A* = *A′* @$_r$ (*D* ·$_m$ (*1*$_m$ *n*))
and *D0*: *D*>*0*
and *n0*: *0*<*n*
shows *make-first-column-positive A*
= *mat-of-rows n* (*map* (*Matrix.row* (*make-first-column-positive A*)) [*0*..<*m*]) @$_r$ (*D* ·$_m$ (*1*$_m$ *n*))
proof (*rule matrix-append-rows-eq-if-preserves*)
have *A*: *A* ∈ *carrier-mat* (*m*+*n*) *n* **using** *A′ A-def* **by** *auto*
thus *make-first-column-positive A* ∈ *carrier-mat* (*m* + *n*) *n* **by** *auto*
have *make-first-column-positive A* $$ (*i*, *j*) = (*D* ·$_m$ *1*$_m$ *n*) $$ (*i* − *m*, *j*)
if *j*: *j*<*n* **and** *i*: *i* ∈ {*m*..<*m* + *n*} **for** *i j*
proof −
have *i-mn*: *i*<*m*+*n* **using** *i* **by** *auto*
have *A* $$ (*i*,*0*) = (*D* ·$_m$ *1*$_m$ *n*) $$ (*i* − *m*, *0*) **unfolding** *A-def*
by (*smt* (*verit*) *A append-rows-def assms*(*1*) *assms*(*2*) *atLeastLessThan-iff carrier-matD*
index-mat-four-block less-irrefl-nat nat-SN.compat j i n0)
also have ... ≥ *0* **using** *D0 mult-not-zero that*(*2*) **by** *auto*
finally have *Ai0*: *A*$$(*i*,*0*)≥*0* .

```
    have make-first-column-positive A $$ (i, j) = A$$(i,j)
      using make-first-column-positive-works[OF A i-mn n0] j Ai0 by auto
    also have ... = (D ·m 1m n) $$ (i − m, j) unfolding A-def
     by (smt (verit) A append-rows-def A' A-def atLeastLessThan-iff carrier-matD

         index-mat-four-block less-irrefl-nat nat-SN.compat i j)
    finally show ?thesis .
  qed
  thus ∀ i∈{m..<m + n}. ∀ j<n. make-first-column-positive A $$ (i, j) = (D ·m
1m n) $$ (i − m, j)
    by simp
qed (auto)


lemma A'-swaprows-invertible-mat:
  fixes A::int mat
  assumes A: A∈carrier-mat m n
  assumes A'-def: A' = (if A $$ (0, 0) ≠ 0 then A else let i = non-zero-positions
! 0 in swaprows 0 i A)
  and nz-def: non-zero-positions = filter (λi. A $$ (i,0) ≠ 0) [1..<dim-row A]
  and nz-empty: A$$(0,0) =0 ⟹ non-zero-positions ≠ []
  and m0: 0<m
shows ∃ P. P ∈ carrier-mat m m ∧ invertible-mat P ∧ A' = P ∗ A
proof (cases A$$(0,0) ≠ 0)
  case True
  then show ?thesis
    by (metis A A'-def invertible-mat-one left-mult-one-mat one-carrier-mat)
next
  case False
  have nz-empty: non-zero-positions ≠ [] using nz-empty False by simp
  let ?i = non-zero-positions ! 0
  let ?M = (swaprows-mat m 0 ?i) :: int mat
  have i-set-nz: ?i ∈ set (non-zero-positions) using nz-empty by auto
  have im: ?i < m using A nz-def i-set-nz by auto
  have i-not0: ?i ≠ 0 using A nz-def i-set-nz by auto
  have A' = swaprows 0 ?i A using False A'-def by simp
  also have ... = ?M ∗ A
    by (rule swaprows-mat[OF A], insert nz-def nz-empty False A m0 im, auto)
  finally have 1: A' = ?M ∗ A .
  have 2: ?M ∈ carrier-mat m m by auto
  have Determinant.det ?M = − 1
    by (rule det-swaprows-mat[OF m0 im i-not0[symmetric]])
  hence 3: invertible-mat ?M using invertible-iff-is-unit-JNF[OF 2] by auto
  show ?thesis using 1 2 3 by blast
qed

lemma swaprows-append-id:
 assumes A': A' ∈ carrier-mat m n
    and A-def: A = A' @r (D ·m (1m n))
```

and *i*:*i*<*m*
shows *swaprows 0 i A*
= *mat-of-rows n* (*map* (*Matrix.row* (*swaprows 0 i A*)) [*0*..<*m*]) @$_r$ (*D* ·$_m$ (*1*$_m$ *n*))
proof (*rule matrix-append-rows-eq-if-preserves*)
have *A*: *A* ∈ *carrier-mat* (*m*+*n*) *n* **using** *A*′ *A-def* **by** *auto*
show *swap*: *swaprows 0 i A* ∈ *carrier-mat* (*m* + *n*) *n* **by** (*simp add*: *A*)
have *swaprows 0 i A* $$ (*ia*, *j*) = (*D* ·$_m$ *1*$_m$ *n*) $$ (*ia* − *m*, *j*)
if *ia*: *ia* ∈ {*m*..<*m* + *n*} **and** *j*: *j*<*n* **for** *ia j*
proof −
have *swaprows 0 i A* $$ (*ia*, *j*) = *A* $$ (*ia*,*j*) **using** *i ia j A* **by** *auto*
also have ... = (*D* ·$_m$ *1*$_m$ *n*) $$ (*ia* − *m*, *j*)
by (*smt* (*verit*) *A append-rows-def A*′ *A-def atLeastLessThan-iff carrier-matD*
index-mat-four-block less-irrefl-nat nat-SN.compat ia j)
finally show *swaprows 0 i A* $$ (*ia*, *j*) = (*D* ·$_m$ *1*$_m$ *n*) $$ (*ia* − *m*, *j*) .
qed
thus ∀ *ia*∈{*m*..<*m* + *n*}. ∀ *j*<*n*. *swaprows 0 i A* $$ (*ia*, *j*) = (*D* ·$_m$ *1*$_m$ *n*) $$ (*ia* − *m*, *j*) **by** *simp*
qed (*simp*)

lemma *non-zero-positions-xs-m*:
fixes *A*::′*a*::*comm-ring-1 mat*
assumes *A-def*: *A* = *A*′ @$_r$ *D* ·$_m$ *1*$_m$ *n*
and *A*′: *A*′ ∈ *carrier-mat m n*
and *nz-def*: *non-zero-positions* = *filter* (λ*i*. *A* $$ (*i*,*0*) ≠ *0*) [*1*..<*dim-row A*]
and *m0*: *0*<*m* **and** *n0*: *0*<*n*
and *D0*: *D* ≠ *0*
shows ∃ *xs*. *non-zero-positions* = *xs* @ [*m*] ∧ *distinct xs* ∧ (∀ *x*∈*set xs*. *x* < *m* ∧ *0* < *x*)
proof −
have *A*: *A* ∈ *carrier-mat* (*m*+*n*) *n* **using** *A*′ *A-def* **by** *auto*
let *?xs* = *filter* (λ*i*. *A* $$ (*i*,*0*) ≠ *0*) [*1*..<*m*]
have *l-rw*: [*1*..<*dim-row A*] = [*1*..<*m*+*1*]@[*m*+*1*..<*dim-row A*] **using** *A m0 n0*
by (*auto*, *metis Suc-leI less-add-same-cancel1 upt-add-eq-append upt-conv-Cons*)
have *f0*: *filter* (λ*i*. *A* $$ (*i*,*0*) ≠ *0*) ([*m*+*1*..<*dim-row A*]) = []
proof (*rule filter-False*)
have *A* $$ (*i*,*0*) = *0* **if** *i*: *i*∈*set* [*m* + *1*..<*dim-row A*] **for** *i*
proof −
have *A* $$ (*i*,*0*) = (*D* ·$_m$ *1*$_m$ *n*) $$ (*i*−*m*,*0*)
by (*rule append-rows-nth3*[*OF A*′ - *A-def* - - *n0*], *insert i A*, *auto*)
also have ... = *0* **using** *i A* **by** *auto*
finally show *?thesis* .
qed
thus ∀ *x*∈*set* [*m* + *1*..<*dim-row A*]. ¬ *A* $$ (*x*, *0*) ≠ *0* **by** *blast*
qed
have *fm*: *filter* (λ*i*. *A* $$ (*i*,*0*) ≠ *0*) [*m*] = [*m*]

proof −
have A $$ $(m,\ 0) = (D \cdot_m 1_m\ n)$ $$ $(m-m,0)$
by (*rule append-rows-nth3*[*OF A' - A-def - - n0*], *insert n0*, *auto*)
also have ... = D **using** *m0 n0* **by** *auto*
finally show *?thesis* **using** *D0* **by** *auto*
qed
have *non-zero-positions* = *filter* ($\lambda i.$ A $$ $(i,0) \neq 0$) ([*1..<m+1*]@[*m+1..<dim-row A*])
using *nz-def l-rw* **by** *auto*
also have ... = *filter* ($\lambda i.$ A $$ $(i,0) \neq 0$) [*1..<m+1*] @ *filter* ($\lambda i.$ A $$ $(i,0) \neq 0$) ([*m+1..<dim-row A*])
by *auto*
also have ... = *filter* ($\lambda i.$ A $$ $(i,0) \neq 0$) [*1..<m+1*] **using** *f0* **by** *auto*
also have ... = *filter* ($\lambda i.$ A $$ $(i,0) \neq 0$) ([*1..<m*]@[*m*]) **using** *m0* **by** *auto*
also have ... = *filter* ($\lambda i.$ A $$ $(i,0) \neq 0$) [*1..<m*] @ [*m*] **using** *fm* **by** *auto*
finally have *non-zero-positions* = *?xs* @ [*m*] .
moreover have *distinct ?xs* **by** *auto*
moreover have ($\forall x \in set$ *?xs*. $x < m \wedge 0 < x$) **by** *auto*
ultimately show *?thesis* **by** *blast*
qed

lemma *non-zero-positions-xs-m'*:
fixes *A*::*'a*::*comm-ring-1 mat*
assumes *A-def*: $A = A' \ @_r\ D \cdot_m 1_m\ n$
and *A'*: $A' \in$ *carrier-mat m n*
and *nz-def*: *non-zero-positions* = *filter* ($\lambda i.$ A $$ $(i,0) \neq 0$) [*1..<dim-row A*]
and *m0*: $0<m$ **and** *n0*: $0<n$
and *D0*: $D \neq 0$
shows *non-zero-positions* = (*filter* ($\lambda i.$ A $$ $(i,0) \neq 0$) [*1..<m*]) @ [*m*]
$\wedge$ *distinct* (*filter* ($\lambda i.$ A $$ $(i,0) \neq 0$) [*1..<m*])
$\wedge$ ($\forall x \in set$ (*filter* ($\lambda i.$ A $$ $(i,0) \neq 0$) [*1..<m*]). $x < m \wedge 0 < x$)
proof −
have *A*: $A \in$ *carrier-mat* $(m+n)$ n **using** *A' A-def* **by** *auto*
let *?xs* = *filter* ($\lambda i.$ A $$ $(i,0) \neq 0$) [*1..<m*]
have *l-rw*: [*1..<dim-row A*] = [*1..<m+1*]@[*m+1..<dim-row A*] **using** *A m0 n0*
by (*auto*, *metis Suc-leI less-add-same-cancel1 upt-add-eq-append upt-conv-Cons*)
have *f0*: *filter* ($\lambda i.$ A $$ $(i,0) \neq 0$) ([*m+1..<dim-row A*]) = []
proof (*rule filter-False*)
have A $$ $(i,0) = 0$ **if** *i*: $i \in set$ [m + *1..<dim-row A*] **for** i
proof −
have A $$ $(i,0) = (D \cdot_m 1_m\ n)$ $$ $(i-m,0)$
by (*rule append-rows-nth3*[*OF A' - A-def - - n0*], *insert i A*, *auto*)
also have ... = *0* **using** *i A* **by** *auto*
finally show *?thesis* .
qed
thus $\forall x \in set$ [m + *1..<dim-row A*]. $\neg$ A $$ $(x,\ 0) \neq 0$ **by** *blast*

```
  qed
  have fm: filter (λi. A $$ (i,0) ≠ 0) [m] = [m]
  proof −
    have A $$ (m, 0) = (D ·_m 1_m n) $$ (m−m,0)
      by (rule append-rows-nth3[OF A' - A-def - - n0], insert n0, auto)
    also have ... = D using m0 n0 by auto
    finally show ?thesis using D0 by auto
  qed
  have non-zero-positions = filter (λi. A $$ (i,0) ≠ 0) ([1..<m+1]@[m+1..<dim-row
A])
    using nz-def l-rw by auto
  also have ... = filter (λi. A $$ (i,0) ≠ 0) [1..<m+1] @ filter (λi. A $$ (i,0) ≠
0) ([m+1..<dim-row A])
    by auto
  also have ... = filter (λi. A $$ (i,0) ≠ 0) [1..<m+1] using f0 by auto
  also have ... = filter (λi. A $$ (i,0) ≠ 0) ([1..<m]@[m]) using m0 by auto
  also have ... = filter (λi. A $$ (i,0) ≠ 0) [1..<m] @ [m] using fm by auto
  finally have non-zero-positions = ?xs @ [m] .
  moreover have distinct ?xs by auto
  moreover have (∀ x∈set ?xs. x < m ∧ 0 < x) by auto
  ultimately show ?thesis by blast
qed

lemma A-A'D-eq-first-n-rows:
 assumes A-def: A = A' @_r D ·_m 1_m n
  and A': A' ∈ carrier-mat m n
  and mn: m≥n
shows (mat-of-rows n (map (Matrix.row A') [0..<n]))
  = (mat-of-rows n (map (Matrix.row A) [0..<n])) (is ?lhs = ?rhs)
proof (rule eq-matI)
  show dr: dim-row ?lhs = dim-row ?rhs and dc: dim-col ?lhs = dim-col ?rhs by
auto
  have D: D ·_m 1_m n : carrier-mat n n by simp
  fix i j assume i: i<dim-row ?rhs and j: j<dim-col ?rhs
 have ?lhs $$ (i,j) = A' $$ (i,j) using i j dr dc A' mn by (simp add: mat-of-rows-def)
  also have ... = A $$ (i,j) using append-rows-nth[OF A' D] i j dr dc A' mn A-def
by auto
  also have ... = ?rhs $$ (i,j) using i j dr dc A' A-def mn
    by (metis D calculation carrier-matD(1) diff-zero gr-implies-not0 length-map
length-upt
        linordered-semidom-class.add-diff-inverse mat-of-rows-carrier(2,3)
        mat-of-rows-index nat-SN.compat nth-map-upt row-append-rows1)
  finally show ?lhs $$ (i,j) = ?rhs $$ (i,j) .
qed

lemma non-zero-positions-xs-m-invertible:
  assumes A-def: A = A' @_r D ·_m 1_m n
  and A': A' ∈ carrier-mat m n
  and nz-def: non-zero-positions = filter (λi. A $$ (i,0) ≠ 0) [1..<dim-row A]
```

```
  and m0: 0<m and n0: 0<n
  and D0: D ≠ 0
  and inv-A'': invertible-mat (map-mat rat-of-int (mat-of-rows n (map (Matrix.row
A') [0..<n])))
  and A'00: A' $$ (0,0) = 0
  and mn: m≥n
shows length non-zero-positions > 1
proof −
  have A: A ∈ carrier-mat (m+n) n using A' A-def by auto
  have D: D ·m 1m n : carrier-mat n n by auto
  let ?RAT = map-mat rat-of-int
  let ?A'' = (mat-of-rows n (map (Matrix.row A') [0..<n]))
  have A'': ?A'' ∈ carrier-mat n n by auto
  have RAT-A'': ?RAT ?A'' ∈ carrier-mat n n by auto
  let ?ys = filter (λi. A $$ (i,0) ≠ 0) [1..<m]
  let ?xs = filter (λi. A $$ (i,0) ≠ 0) [1..<n]
  have xs-not-empty:?xs ≠ []
  proof (rule ccontr)
    assume ¬ ?xs ≠ [] hence xs0: ?xs = [] by simp
    have A00: A $$ (0,0) = 0
    proof −
      have A $$ (0,0) = A'$$(0,0) unfolding A-def using append-rows-nth[OF
A' D] m0 n0 A' by auto
      thus ?thesis using A'00 by simp
    qed
    hence (∀ i∈set [1..<n]. A $$ (i,0) = 0)
      by (metis (mono-tags, lifting) empty-filter-conv xs0)
    hence ∗: (∀ i<n. A $$ (i,0) = 0) using A00 n0 using linorder-not-less by
force
    have col ?A'' 0 = 0v n
    proof (rule eq-vecI)
      show dim-vec (col ?A'' 0) = dim-vec (0vn) using A' by auto
      fix i assume i: i < dim-vec (0v n)
      have col ?A'' 0 $v i = ?A'' $$ (i,0) by (rule index-col, insert i A' n0, auto)
      also have ... = A $$ (i,0)
        unfolding A-def using i A append-rows-nth[OF A' D - n0] A' mn
      by (metis A'' n0 carrier-matD(1) index-zero-vec(2) le-add2 map-first-rows-index
          mat-of-rows-carrier(2) mat-of-rows-index nat-SN.compat)
      also have ... = 0 using ∗ i by auto
      finally show col ?A'' 0 $v i = 0v n $v i using i by auto
    qed
    hence col (?RAT ?A'') 0 = 0v n by auto
    hence ¬ invertible-mat (?RAT ?A'')
      using invertible-mat-first-column-not0[OF RAT-A'' - n0] by auto
    thus False using inv-A'' by contradiction
  qed
  have l-rw: [1..<dim-row A] = [1..<m+1]@[m+1..<dim-row A] using A m0 n0
   by (auto, metis Suc-leI less-add-same-cancel1 upt-add-eq-append upt-conv-Cons)
  have f0: filter (λi. A $$ (i,0) ≠ 0) ([m+1..<dim-row A]) = []
```

```
  proof (rule filter-False)
    have A $$ (i,0) = 0 if i: i∈set [m + 1..<dim-row A] for i
    proof −
      have A $$ (i,0) = (D ·_m 1_m n) $$ (i−m,0)
        by (rule append-rows-nth3[OF A' - A-def - - n0], insert i A, auto)
      also have ... = 0 using i A by auto
      finally show ?thesis .
    qed
    thus ∀ x∈set [m + 1..<dim-row A]. ¬ A $$ (x, 0) ≠ 0 by blast
  qed
  have fm: filter (λi. A $$ (i,0) ≠ 0) [m] = [m]
  proof −
    have A $$ (m, 0) = (D ·_m 1_m n) $$ (m−m,0)
      by (rule append-rows-nth3[OF A' - A-def - - n0], insert n0, auto)
    also have ... = D using m0 n0 by auto
    finally show ?thesis using D0 by auto
  qed
 have non-zero-positions = filter (λi. A $$ (i,0) ≠ 0) ([1..<m+1]@[m+1..<dim-row
A])
    using nz-def l-rw by auto
  also have ... = filter (λi. A $$ (i,0) ≠ 0) [1..<m+1] @ filter (λi. A $$ (i,0) ≠
0) ([m+1..<dim-row A])
    by auto
  also have ... = filter (λi. A $$ (i,0) ≠ 0) [1..<m+1] using f0 by auto
  also have ... = filter (λi. A $$ (i,0) ≠ 0) ([1..<m]@[m]) using m0 by auto
  also have ... = filter (λi. A $$ (i,0) ≠ 0) [1..<m] @ [m] using fm by auto
  finally have nz: non-zero-positions = ?ys @ [m] .
  moreover have ys-not-empty: ?ys ≠ [] using xs-not-empty mn
   by (metis (no-types, lifting) atLeastLessThan-iff empty-filter-conv nat-SN.compat
set-upt)
  show ?thesis unfolding nz using ys-not-empty by auto
qed



corollary non-zero-positions-length-xs:
  assumes A-def: A = A' @_r D ·_m 1_m n
  and A': A' ∈ carrier-mat m n
  and nz-def: non-zero-positions = filter (λi. A $$ (i,0) ≠ 0) [1..<dim-row A]
  and m0: 0<m and n0: 0<n
  and D0: D ≠ 0
 and inv-A'': invertible-mat (map-mat rat-of-int (mat-of-rows n (map (Matrix.row
A') [0..<n])))
  and A'00: A' $$ (0,0) = 0
  and mn: m≥n
  and nz-xs-m: non-zero-positions = xs @ [m]
shows length xs > 0
proof −
  have length non-zero-positions > 1
```

by (*rule non-zero-positions-xs-m-invertible*[*OF A-def A′ nz-def m0 n0 D0 inv-A″ A′00 mn*])
thus *?thesis* **using** *nz-xs-m* **by** *auto*
qed

lemma *make-first-column-positive-nz-conv*:
assumes *i*<*dim-row A* **and** *j*<*dim-col A*
shows (*make-first-column-positive A* $$ (*i*, *j*) ≠ *0*) = (*A* $$ (*i*, *j*) ≠ *0*)
using *assms* **unfolding** *make-first-column-positive.simps* **by** *auto*

lemma *make-first-column-positive-00*:
assumes *A-def*: *A* = *A″* @$_r$ *D* ·$_m$ *1*$_m$ *n*
and *A″*: *A″* : *carrier-mat m n*
assumes *nz-def*: *non-zero-positions* = *filter* (λ*i*. *A* $$ (*i*,*0*) ≠ *0*) [*1*..<*dim-row A*]
and *A′-def*: *A′* = (*if A* $$ (*0*, *0*) ≠ *0 then A else let i* = *non-zero-positions* ! *0 in swaprows 0 i A*)
and *m0*: *0*<*m* **and** *n0*: *0*<*n* **and** *D0*: *D* ≠ *0* **and** *mn*: *m*≥*n*
shows *make-first-column-positive A′* $$ (*0*, *0*) ≠ *0*
proof −
have *A*: *A* ∈ *carrier-mat* (*m*+*n*) *n* **using** *A-def A″* **by** *auto*
hence *A′*: *A′* ∈ *carrier-mat* (*m*+*n*) *n* **unfolding** *A′-def* **by** *auto*
have (*make-first-column-positive A′* $$ (*0*, *0*) ≠ *0*) = (*A′* $$ (*0*, *0*) ≠ *0*)
by (*rule make-first-column-positive-nz-conv*, *insert m0 n0 A′*, *auto*)
moreover have *A′* $$ (*0*, *0*) ≠ *0*
proof (*cases A* $$ (*0*, *0*) ≠ *0*)
case *True*
then show *?thesis* **unfolding** *A′-def* **by** *auto*
next
case *False*
have *A* $$ (*0*, *0*) = *A″* $$ (*0*, *0*)
by (*smt* (*verit*) *add-gr-0 append-rows-def A-def A″ carrier-matD index-mat-four-block*(*1*) *mn n0 nat-SN.compat*)
hence *A″00*: *A″*$$(*0*,*0*) = *0* **using** *False* **by** *auto*
let *?i* = *non-zero-positions* ! *0*
obtain *xs* **where** *non-zero-positions-xs-m*: *non-zero-positions* = *xs* @ [*m*] **and** *d-xs*: *distinct xs*
and *all-less-m*: ∀ *x*∈*set xs*. *x* < *m* ∧ *0* < *x*
using *non-zero-positions-xs-m*[*OF A-def A″ nz-def m0 n0*] **using** *D0* **by** *fast*

have *Ai0*: *A* $$ (*?i*,*0*) ≠ *0*
by (*smt* (*verit*, *ccfv-threshold*) *add-gr-0 length-append less-numeral-extra*(*1*) *list.size*(*4*) *local.non-zero-positions-xs-m mem-Collect-eq nth-mem nz-def plus-1-eq-Suc set-filter*)
have *A′* $$ (*0*, *0*) = *swaprows 0 ?i A* $$ (*0*,*0*) **using** *False A′-def* **by** *auto*

```
    also have ... ≠ 0 using A Ai0 n0 by auto
    finally show ?thesis .
  qed
  ultimately show ?thesis by blast
qed


context proper-mod-operation
begin
lemma reduce-below-0-case-m-make-first-column-positive:
  assumes A': A' ∈ carrier-mat m n and m0: 0<m and n0: 0<n
    and A-def: A = A' @_r (D ·_m (1_m n))
    and mn: m≥n
  assumes i-mn: i < m+n and d-xs: distinct xs and xs: ∀ x ∈ set xs. x < m ∧ 0
< x
    and ia: i≠0
    and A''-def: A'' = (if A $$ (0, 0) ≠ 0 then A else let i = non-zero-positions !
0 in swaprows 0 i A)
    and D0: D>0
    and nz-def: non-zero-positions = filter (λi. A $$ (i,0) ≠ 0) [1..<dim-row A]
  shows reduce-below 0 non-zero-positions D (make-first-column-positive A'') $$
(i,0) = 0
proof −
  have A: A ∈ carrier-mat (m+n) n using A' A-def by auto
  define xs where xs = filter (λi. A $$ (i,0) ≠ 0) [1..<m]
  have nz-xs-m: non-zero-positions = xs @ [m] and d-xs: distinct xs
    and all-less-m: ∀ x∈set xs. x < m ∧ 0 < x
     using non-zero-positions-xs-m'[OF A-def A' nz-def m0 n0] using D0 A un-
folding nz-def xs-def by auto
  have A'': A'' ∈ carrier-mat (m+n) n using A' A-def A''-def by auto
  have D-not0: D≠0 using D0 by auto
  have Ai0: A $$ (i, 0) = 0 if im: i>m and imn: i<m+n for i
  proof−
    have D: (D ·_m (1_m n)) ∈ carrier-mat n n by simp
    have A $$ (i, 0) = (D ·_m (1_m n)) $$ (i−m, 0)
      unfolding A-def using append-rows-nth[OF A' D imn n0] im A' by auto
    also have ... = 0 using im imn n0 by auto
    finally show ?thesis .
  qed
   let ?M' = mat-of-rows n (map (Matrix.row (make-first-column-positive A''))
[0..<m])
  have M': ?M' ∈ carrier-mat m n using A'' by auto
  have mk0: make-first-column-positive A'' $$ (0, 0) ≠ 0
   by (rule make-first-column-positive-00[OF A-def A' nz-def A''-def m0 n0 D-not0
mn])
  have M-M'D: make-first-column-positive A'' = ?M' @_r D ·_m 1_m n if xs-empty:
xs ≠ []
  proof (cases A$$(0,0) ≠ 0)
    case True
```

```
    then have ∗: make-first-column-positive A'' = make-first-column-positive A
      unfolding A''-def by auto
    show ?thesis
      by (unfold ∗, rule make-first-column-positive-append-id[OF A' A-def D0 n0])
  next
    case False
    then have ∗: make-first-column-positive A''
                = make-first-column-positive (swaprows 0 (non-zero-positions ! 0)
A)
      unfolding A''-def by auto
    show ?thesis
    proof (unfold ∗, rule make-first-column-positive-append-id)
      let ?S = mat-of-rows n (map (Matrix.row (swaprows 0 (non-zero-positions !
0) A)) [0..<m])
      show swaprows 0 (non-zero-positions ! 0) A =  ?S @r (D ·m (1m n))
      proof (rule swaprows-append-id[OF A' A-def])
        have A'00: A' $$ (0, 0) = 0
         by (metis (no-types, lifting) A False add-pos-pos append-rows-def A' A-def
             carrier-matD index-mat-four-block m0 n0)
        have length-xs: length xs > 0 using xs-empty by auto
        have non-zero-positions ! 0 = xs ! 0 unfolding nz-xs-m
          by (meson length-xs nth-append)
        thus non-zero-positions ! 0 < m using all-less-m length-xs by simp
      qed
    qed (insert n0 D0, auto)
  qed
  show ?thesis
  proof (cases xs = [])
    case True note xs-empty = True
    have reduce-below 0 non-zero-positions D (make-first-column-positive A'')
      = reduce 0 m D (make-first-column-positive A'')
      unfolding nz-xs-m True by auto

    also have ... $$ (i, 0) = 0
    proof (cases i=m)
      case True
      from D0 have D ≥ 1 D ≥ 0 by auto
      then show ?thesis using D0 True
        by (metis A add-sign-intros(2) A''-def carrier-matD(1) carrier-matD(2)
carrier-matI
          index-mat-swaprows(2) index-mat-swaprows(3) less-add-same-cancel1 m0

            make-first-column-positive-preserves-dimensions mk0 n0 neq0-conv re-
duce-0)
    next
      case False note i-not-m = False
      have nz-m: non-zero-positions ! 0 = m unfolding nz-xs-m True by auto
      let ?M = make-first-column-positive A''
      have M: ?M ∈ carrier-mat (m+n) n using A'' by auto
```

show *?thesis*
proof (*cases A$$(0,0) = 0*)
case *True*
have *reduce 0 m D ?M $$ (i, 0) = ?M $$ (i,0)*
by (*rule reduce-preserves*[*OF M n0 mk0 False ia i-mn*])
also have *Mi0*: *... = abs (A'' $$ (i,0))*
by (*smt* (*verit*) *M carrier-matD*(*1*) *carrier-matD*(*2*) *i-mn index-mat*(*1*) *make-first-column-positive.simps*
make-first-column-positive-preserves-dimensions n0 prod.simps(*2*))
also have *Mi02*: *... = abs (A $$ (i,0))* **unfolding** *A''-def nz-m*
using *True A False i-mn ia n0* **by** *auto*
also have *... = 0*
proof −
have *filter* (*λn. A $$ (n, 0) ≠ 0*) [*1..<m*] = []
using *xs-empty xs-def* **by** *presburger*
then have *∀ n. A $$ (n, 0) = 0 ∨ n ∉ set* [*1..<m*] **using** *filter-empty-conv* **by** *fast*
then show *?thesis*
by (*metis* (*no-types*) *Ai0 False arith-simps*(*43*) *assms*(*9*) *atLeastLessThan-iff i-mn*
le-eq-less-or-eq less-one linorder-neqE-nat set-upt)
qed
finally show *?thesis* **.**
next
case *False* **hence** *A00*: *A $$ (0,0) ≠ 0* **by** *simp*
have *reduce 0 m D ?M $$ (i, 0) = ?M $$ (i,0)*
by (*rule reduce-preserves*[*OF M n0 mk0 i-not-m ia i-mn*])
also have *Mi0*: *... = abs (A'' $$ (i,0))*
by (*smt* (*verit*) *M carrier-matD*(*1*) *carrier-matD*(*2*) *i-mn index-mat*(*1*) *make-first-column-positive.simps*
make-first-column-positive-preserves-dimensions n0 prod.simps(*2*))
also have *Mi02*: *... = abs* (*swaprows 0 m A $$ (i,0)*) **unfolding** *A''-def nz-m*
using *A00 A i-not-m i-mn ia n0* **by** *auto*
also have *... = abs (A $$ (i,0))* **using** *False ia A00 Mi0 A''-def calculation Mi02* **by** *presburger*
also have *... = 0*
proof −
have *filter* (*λn. A $$ (n, 0) ≠ 0*) [*1..<m*] = []
using *True xs-def* **by** *presburger*
then have *∀ n. A $$ (n, 0) = 0 ∨ n ∉ set* [*1..<m*] **using** *filter-empty-conv* **by** *fast*
then show *?thesis*
by (*metis* (*no-types*) *Ai0 i-not-m arith-simps*(*43*) *ia atLeastLessThan-iff i-mn*
le-eq-less-or-eq less-one linorder-neqE-nat set-upt)
qed
finally show *?thesis* **.**
qed

```
    qed
    finally show ?thesis .
  next
    case False note xs-not-empty = False
    note M-M'D = M-M'D[OF xs-not-empty]
    show ?thesis
    proof (cases i ∈ set (xs @ [m]))
      case True
      show ?thesis
        by (unfold nz-xs-m, rule reduce-below-0-case-m[OF M' m0 n0 M-M'D mk0
mn True d-xs all-less-m D0])
    next
      case False note i-notin-xs-m = False
      have 1: reduce-below 0 (xs @ [m]) D (make-first-column-positive A'') $$ (i,0)

        = (make-first-column-positive A'') $$ (i,0)
        by (rule reduce-below-preserves-case-m[OF M' m0 n0 M-M'D mk0 mn - d-xs
all-less-m ia i-mn - D0],
            insert False, auto)
      have ((make-first-column-positive A'') $$ (i,0) ≠ 0) = (A'' $$ (i,0) ≠ 0)
        by (rule make-first-column-positive-nz-conv, insert A'' i-mn n0, auto)
      hence 2: ((make-first-column-positive A'') $$ (i,0) = 0) = (A'' $$ (i,0) =
0) by auto
      have 3: (A'' $$ (i,0) = 0)
      proof (cases A$$(0,0) ≠ 0)
        case True
        then have A'' $$ (i, 0) = A $$ (i, 0) unfolding A''-def by auto
         also have ... = 0 using False ia i-mn A nz-xs-m Ai0 unfolding nz-def
xs-def by auto
        finally show ?thesis by auto
      next
        case False hence A00: A $$ (0,0) = 0 by simp
        let ?i = non-zero-positions ! 0
        have i-noti: i≠?i
          using i-notin-xs-m unfolding nz-xs-m
        by (metis Nil-is-append-conv length-greater-0-conv list.distinct(2) nth-mem)
      have A''$$(i,0) = (swaprows 0 ?i A) $$ (i,0) using False unfolding A''-def
by auto
       also have ... = A $$ (i,0) using i-notin-xs-m ia i-mn A i-noti n0 unfolding
xs-def by fastforce
       also have ... = 0 using i-notin-xs-m ia i-mn A i-noti n0 unfolding xs-def
            by (smt (verit) nz-def atLeastLessThan-iff carrier-matD(1) less-one
linorder-not-less
            mem-Collect-eq nz-xs-m set-filter set-upt xs-def)
        finally show ?thesis .
      qed
      show ?thesis using 1 2 3 nz-xs-m by argo
    qed
  qed
```

qed

lemma *reduce-below-abs-0-case-m-make-first-column-positive*:
assumes *A'*: $A' \in$ *carrier-mat m n* **and** *m0*: $0<m$ **and** *n0*: $0<n$
and *A-def*: $A = A' \mathbin{@_r} (D \cdot_m (1_m\ n))$
and *mn*: $m \geq n$
assumes *i-mn*: $i < m+n$ **and** *d-xs*: *distinct xs* **and** *xs*: $\forall x \in$ *set xs*. $x < m \wedge 0 < x$
and *ia*: $i \neq 0$
and *A''-def*: $A'' =$ (*if A $$ (0, 0)* $\neq$ *0 then A else let i = non-zero-positions* ! *0 in swaprows 0 i A*)
and *D0*: $D>0$
and *nz-def*: *non-zero-positions* = *filter* $(\lambda i.$ *A $$ (i,0)* $\neq$ *0)* [*1..<dim-row A*]
shows *reduce-below-abs 0 non-zero-positions D* (*make-first-column-positive A''*) *$$ (i,0) = 0*
proof −
have *A*: $A \in$ *carrier-mat* $(m+n)$ *n* **using** *A' A-def* **by** *auto*
define *xs* **where** *xs* = *filter* $(\lambda i.$ *A $$ (i,0)* $\neq$ *0)* [*1..<m*]
have *nz-xs-m*: *non-zero-positions* = *xs* @ [*m*] **and** *d-xs*: *distinct xs*
and *all-less-m*: $\forall x \in$ *set xs*. $x < m \wedge 0 < x$
using *non-zero-positions-xs-m'*[*OF A-def A' nz-def m0 n0*] **using** *D0 A* **unfolding** *nz-def xs-def* **by** *auto*
have *A''*: $A'' \in$ *carrier-mat* $(m+n)$ *n* **using** *A' A-def A''-def* **by** *auto*
have *D-not0*: $D \neq 0$ **using** *D0* **by** *auto*
have *Ai0*: *A $$ (i, 0) = 0* **if** *im*: $i>m$ **and** *imn*: $i<m+n$ **for** *i*
proof−
have *D*: $(D \cdot_m (1_m\ n)) \in$ *carrier-mat n n* **by** *simp*
have *A $$ (i, 0) = (D* $\cdot_m$ $(1_m\ n))$ *$$ (i−m, 0)*
unfolding *A-def* **using** *append-rows-nth*[*OF A' D imn n0*] *im A'* **by** *auto*
also have *... = 0* **using** *im imn n0* **by** *auto*
finally show *?thesis* **.**
qed
let *?M'* = *mat-of-rows n* (*map* (*Matrix.row* (*make-first-column-positive A''*)) [*0..<m*])
have *M'*: *?M'* $\in$ *carrier-mat m n* **using** *A''* **by** *auto*
have *mk0*: *make-first-column-positive A'' $$ (0, 0)* $\neq$ *0*
by (*rule make-first-column-positive-00*[*OF A-def A' nz-def A''-def m0 n0 D-not0 mn*])
have *M-M'D*: *make-first-column-positive A''* = *?M'* $\mathbin{@_r}$ *D* $\cdot_m$ 1_m *n* **if** *xs-empty*: *xs* $\neq$ []
proof (*cases A$$(0,0)* $\neq$ *0*)
case *True*
then have $*$: *make-first-column-positive A''* = *make-first-column-positive A*
unfolding *A''-def* **by** *auto*
show *?thesis*
by (*unfold* $*$, *rule make-first-column-positive-append-id*[*OF A' A-def D0 n0*])
next
case *False*

```
    then have ∗: make-first-column-positive A''
                = make-first-column-positive (swaprows 0 (non-zero-positions ! 0)
A)
      unfolding A''-def by auto
    show ?thesis
    proof (unfold ∗, rule make-first-column-positive-append-id)
      let ?S = mat-of-rows n (map (Matrix.row (swaprows 0 (non-zero-positions !
0) A)) [0..<m])
      show swaprows 0 (non-zero-positions ! 0) A =  ?S @r (D ·m (1m n))
      proof (rule swaprows-append-id[OF A' A-def])
        have A'00: A' $$ (0, 0) = 0
         by (metis (no-types, lifting) A False add-pos-pos append-rows-def A' A-def
             carrier-matD index-mat-four-block m0 n0)
        have length-xs: length xs > 0 using xs-empty by auto
        have non-zero-positions ! 0 = xs ! 0 unfolding nz-xs-m
          by (meson length-xs nth-append)
        thus non-zero-positions ! 0 < m using all-less-m length-xs by simp
      qed
    qed (insert n0 D0, auto)
  qed
  show ?thesis
  proof (cases xs = [])
    case True note xs-empty = True
    have reduce-below-abs 0 non-zero-positions D (make-first-column-positive A'')
      = reduce-abs 0 m D (make-first-column-positive A'')
      unfolding nz-xs-m True by auto

    also have ... $$ (i, 0) = 0
    proof (cases i=m)
      case True
      from D0 have D ≥ 1 D ≥ 0 by auto
      then show ?thesis using D0 True
        by (metis A add-sign-intros(2) A''-def carrier-matD(1) carrier-matD(2)
carrier-matI
          index-mat-swaprows(2) index-mat-swaprows(3) less-add-same-cancel1 m0

            make-first-column-positive-preserves-dimensions mk0 n0 neq0-conv re-
duce-0)
    next
      case False note i-not-m = False
      have nz-m: non-zero-positions ! 0 = m unfolding nz-xs-m True by auto
      let ?M = make-first-column-positive A''
      have M: ?M ∈ carrier-mat (m+n) n using A'' by auto
      show ?thesis
      proof (cases A$$(0,0) = 0)
        case True
        have reduce-abs 0 m D ?M $$ (i, 0) = ?M $$ (i,0)
          by (rule reduce-preserves[OF M n0 mk0 False ia i-mn])
        also have Mi0: ... = abs (A'' $$ (i,0))
```

```
          by (smt (verit) M carrier-matD(1) carrier-matD(2) i-mn index-mat(1)
make-first-column-positive.simps
              make-first-column-positive-preserves-dimensions n0 prod.simps(2))
        also have Mi02: ... = abs (A $$ (i,0))  unfolding A''-def nz-m
          using True A False i-mn ia n0 by auto
        also have ... = 0
        proof −
          have filter (λn. A $$ (n, 0) ≠ 0) [1..<m] = []
            using xs-empty xs-def by presburger
         then have ∀ n. A $$ (n, 0) = 0 ∨ n ∉ set [1..<m] using filter-empty-conv
by fast
          then show ?thesis
               by (metis (no-types) Ai0 False arith-simps(43) assms(9) atLeast-
LessThan-iff i-mn
                 le-eq-less-or-eq less-one linorder-neqE-nat set-upt)
        qed
        finally show ?thesis .
      next
        case False hence A00: A $$ (0,0) ≠ 0 by simp
        have reduce-abs 0 m D ?M $$ (i, 0) = ?M $$ (i,0)
          by (rule reduce-preserves[OF M n0 mk0 i-not-m ia i-mn])
        also have Mi0: ... = abs (A'' $$ (i,0))
          by (smt (verit) M carrier-matD(1) carrier-matD(2) i-mn index-mat(1)
make-first-column-positive.simps
              make-first-column-positive-preserves-dimensions n0 prod.simps(2))
         also have Mi02: ... = abs (swaprows 0 m A $$ (i,0))  unfolding A''-def
nz-m
          using A00 A i-not-m i-mn ia n0 by auto
        also have ... = abs (A $$ (i,0)) using False ia A00 Mi0 A''-def calculation
Mi02 by presburger
        also have ... = 0
        proof −
          have filter (λn. A $$ (n, 0) ≠ 0) [1..<m] = []
            using True xs-def by presburger
         then have ∀ n. A $$ (n, 0) = 0 ∨ n ∉ set [1..<m] using filter-empty-conv
by fast
          then show ?thesis
            by (metis (no-types) Ai0 i-not-m arith-simps(43) ia atLeastLessThan-iff
i-mn
                le-eq-less-or-eq less-one linorder-neqE-nat set-upt)
        qed
        finally show ?thesis .
      qed
    qed
    finally show ?thesis .
  next
    case False note xs-not-empty = False
    note M-M'D = M-M'D[OF xs-not-empty]
    show ?thesis
```

```
    proof (cases i ∈ set (xs @ [m]))
      case True
      show ?thesis
        by (unfold nz-xs-m, rule reduce-below-abs-0-case-m[OF M' m0 n0 M-M'D
mk0 mn True d-xs all-less-m D0])
    next
      case False note i-notin-xs-m = False
      have 1: reduce-below-abs 0 (xs @ [m]) D (make-first-column-positive A'') $$
(i,0)
        = (make-first-column-positive A'') $$ (i,0)
        by (rule reduce-below-abs-preserves-case-m[OF M' m0 n0 M-M'D mk0 mn
- d-xs all-less-m ia i-mn - D0],
            insert False, auto)
      have ((make-first-column-positive A'') $$ (i,0) ≠ 0) = (A'' $$ (i,0) ≠ 0)
        by (rule make-first-column-positive-nz-conv, insert A'' i-mn n0, auto)
       hence 2: ((make-first-column-positive A'') $$ (i,0) = 0) = (A'' $$ (i,0) =
0) by auto
      have 3: (A'' $$ (i,0) = 0)
      proof (cases A$$(0,0) ≠ 0)
        case True
        then have A'' $$ (i, 0) = A $$ (i, 0) unfolding A''-def by auto
         also have ... = 0 using False ia i-mn A nz-xs-m Ai0 unfolding nz-def
xs-def by auto
        finally show ?thesis by auto
      next
        case False hence A00: A $$ (0,0) = 0 by simp
        let ?i = non-zero-positions ! 0
        have i-noti: i≠?i
          using i-notin-xs-m unfolding nz-xs-m
        by (metis Nil-is-append-conv length-greater-0-conv list.distinct(2) nth-mem)
       have A''$$(i,0) = (swaprows 0 ?i A) $$ (i,0) using False unfolding A''-def
by auto
       also have ... = A $$ (i,0) using i-notin-xs-m ia i-mn A i-noti n0 unfolding
xs-def by fastforce
        also have ... = 0 using i-notin-xs-m ia i-mn A i-noti n0 unfolding xs-def
             by (smt (verit) nz-def atLeastLessThan-iff carrier-matD(1) less-one
linorder-not-less
             mem-Collect-eq nz-xs-m set-filter set-upt xs-def)
        finally show ?thesis .
      qed
      show ?thesis using 1 2 3 nz-xs-m by argo
    qed
  qed
qed


lemma FindPreHNF-invertible-mat-2xn:
  assumes A: A ∈ carrier-mat m n and m<2
  shows ∃ P. P ∈ carrier-mat m m ∧ invertible-mat P ∧ FindPreHNF abs-flag D
```

$A = P * A$
using *assms*
by (*auto*, *metis invertible-mat-one left-mult-one-mat one-carrier-mat*)

lemma *FindPreHNF-invertible-mat-mx2*:
assumes *A-def*: $A = A'' \,@_r\, D \cdot_m 1_m\, n$
and *A''*: $A'' \in$ *carrier-mat m n* **and** *n2*: $n<2$ **and** *n0*: $0<n$ **and** *D-g0*: $D>0$ **and** *mn*: $m \geq n$
shows $\exists P.\ P \in$ *carrier-mat* $(m+n)\ (m+n) \wedge$ *invertible-mat P* $\wedge$ *FindPreHNF abs-flag D A* $= P * A$
proof −
have *A*: $A \in$ *carrier-mat* $(m+n)\ n$ **using** *A-def A''* **by** *auto*
have *m0*: $m>0$ **using** *mn n2 n0* **by** *auto*
have *D0*: $D \neq 0$ **using** *D-g0* **by** *auto*
show *?thesis*
proof (*cases* $m+n<2$)
case *True*
show *?thesis* **by** (*rule FindPreHNF-invertible-mat-2xn*[*OF A True*])
next
case *False* **note** *mn-le-2 = False*
have *dr-A*: *dim-row A* ≥ 2 **using** *False n2 A* **by** *auto*
have *dc-A*: *dim-col A* < 2 **using** *n2 A* **by** *auto*
let *?non-zero-positions = filter* $(\lambda i.\ A\ \$\$\ (i,\ 0) \neq 0)$ [*Suc 0*..<*dim-row A*]
let *?A'* = (*if A* $\$\$\ (0,\ 0) \neq 0$ *then A else let i = ?non-zero-positions* ! *0 in swaprows 0 i A*)
define *xs* **where** *xs = filter* $(\lambda i.\ A\ \$\$\ (i,0) \neq 0)$ [*1*..<*m*]
let *?Reduce* = (*if abs-flag then reduce-below-abs else reduce-below*)
have *nz-xs-m*: *?non-zero-positions = xs* @ [*m*] **and** *d-xs*: *distinct xs*
and *all-less-m*: $\forall x \in$ *set xs*. $x < m \wedge 0 < x$
using *non-zero-positions-xs-m'*[*OF A-def A'' - m0 n0 D0*] **using** *D0 A* **unfolding** *xs-def* **by** *auto*
have $*$: *FindPreHNF abs-flag D A = (if abs-flag then reduce-below-abs 0 ?non-zero-positions D ?A'*
else reduce-below 0 ?non-zero-positions D ?A')
using *dr-A dc-A* **by** (*auto simp add*: *Let-def*)
have *l*: *length ?non-zero-positions > 1* **if** *xs*≠[] **using** *that* **unfolding** *nz-xs-m* **by** *auto*
have *inv*: $\exists P.$ *invertible-mat P* $\wedge$ $P \in$ *carrier-mat* $(m + n)\ (m + n)$
$\wedge$ *reduce-below 0 ?non-zero-positions D ?A'* $= P * ?A'$
proof (*cases A* $\$\$\ (0,0) \neq 0$)
case *True*
show *?thesis*
by (*unfold nz-xs-m*, *rule reduce-below-invertible-mat-case-m*
[*OF A'' m0 n0 - - mn d-xs all-less-m*], *insert A-def True D-g0*, *auto*)
next
case *False* **hence** *A00*: *A* $\$\$\ (0,0) = 0$ **by** *auto*
let *?S = swaprows 0* (*?non-zero-positions* ! *0*) *A*
have *rw*: (*if A* $\$\$\ (0,\ 0) \neq 0$ *then A else let i = ?non-zero-positions* ! *0 in*

swaprows 0 i A)
        = *?S* **using** *False* **by** *auto*
      show *?thesis*
      proof (*cases xs* = [])
        case *True*
        have *nz-m*: *?non-zero-positions* = [*m*] **using** *True nz-xs-m* **by** *simp*
        obtain *p q u v d* **where** *pquvd*: (*p*,*q*,*u*,*v*,*d*) = *euclid-ext2* (*swaprows 0 m A* $$ (*0*, *0*)) (*swaprows 0 m A* $$ (*m*, *0*))
          by (*metis prod-cases5*)
        have *Am0*: *A* $$ (*m*,*0*) = *D*
        proof −
          have *A* $$ (*m*,*0*) = (*D* ·$_m$ *1*$_m$ *n*) $$ (*m*−*m*, *0*)
            unfolding *A-def*
            by (*meson append-rows-nth3 assms*(*2*) *assms*(*4*) *less-add-same-cancel1 one-carrier-mat order-refl smult-carrier-mat*)
          also have ... = *D* **by** (*simp add*: *n0*)
          finally show *?thesis* **.**
        qed
        have *Sm0*: (*swaprows 0 m A*) $$ (*m*,*0*) = *0* **using** *A False n0* **by** *auto*
        have *S00*: (*swaprows 0 m A*) $$ (*0*,*0*) = *D* **using** *A Am0 n0* **by** *auto*
        have *pquvd2*: (*p*,*q*,*u*,*v*,*d*) = *euclid-ext2* (*A* $$ (*m*, *0*)) (*A* $$ (*0*, *0*))
          using *pquvd Sm0 S00 Am0 A00* **by** *auto*
      have *reduce-below 0 ?non-zero-positions D ?A'* = *reduce 0 m D ?A'* **unfolding** *nz-m* **by** *auto*
        also have ... = *reduce 0 m D* (*swaprows 0 m A*) **using** *True False rw nz-m* **by** *auto*
        have ∃ *P*. *invertible-mat P* ∧ *P* ∈ *carrier-mat* (*m* + *n*) (*m* + *n*) ∧
          reduce 0 m D (*swaprows 0 m A*) = *P* ∗ (*swaprows 0 m A*)
        proof (*rule reduce-invertible-mat-case-m*[*OF - - m0 - - - - mn n0*])
          show *swaprows 0 m A* $$ (*0*, *0*) ≠ *0* **using** *S00 D0* **by** *auto*
          define *S'* **where** *S'* = *mat-of-rows n* (*map* (*Matrix.row ?S*) [*0*..<*m*])
          define *S''* **where** *S''* = *mat-of-rows n* (*map* (*Matrix.row ?S*) [*m*..<*m*+*n*])
          define *A2* **where** *A2* = *Matrix.mat* (*dim-row* (*swaprows 0 m A*)) (*dim-col* (*swaprows 0 m A*))
            (λ(*i*, *k*). *if i* = *0 then p* ∗ *A* $$ (*m*, *k*) + *q* ∗ *A* $$ (*0*, *k*)
            else if i = *m then u* ∗ *A* $$ (*m*, *k*) + *v* ∗ *A* $$ (*0*, *k*) *else A* $$ (*i*, *k*))
          show *S-S'-S''*: *swaprows 0 m A* = *S'* @$_r$ *S''* **unfolding** *S'-def S''-def*
            by (*metis A append-rows-split carrier-matD index-mat-swaprows*(*2*,*3*) *le-add1 nth-Cons-0 nz-m*)
          show *S'*: *S'* ∈ *carrier-mat m n* **unfolding** *S'-def* **by** *fastforce*
          show *S''*: *S''* ∈ *carrier-mat n n* **unfolding** *S''-def* **by** *fastforce*
          show *0* ≠ *m* **using** *m0* **by** *simp*
          show (*p*,*q*,*u*,*v*,*d*) = *euclid-ext2* (*swaprows 0 m A* $$ (*0*, *0*)) (*swaprows 0 m A* $$ (*m*, *0*))
            using *pquvd* **by** *simp*
          show *A2* = *Matrix.mat* (*dim-row* (*swaprows 0 m A*)) (*dim-col* (*swaprows 0 m A*))
          (λ(*i*, *k*). *if i* = *0 then p* ∗ *swaprows 0 m A* $$ (*0*, *k*) + *q* ∗ *swaprows 0 m A* $$ (*m*, *k*)

else if $i = m$ *then* $u * swaprows\ 0\ m\ A\ \$\$\ (0,\ k) + v * swaprows\ 0\ m\ A\ \$\$\ (m,\ k)$ *else* $swaprows\ 0\ m\ A\ \$\$\ (i,\ k)$)
(**is** - = *?rhs*) **using** *A A2-def* **by** *auto*
define xs' **where** $xs' = [1..<n]$
define ys' **where** $ys' = [1..<n]$
show $xs' = [1..<n]$ **unfolding** *xs'-def* **by** *auto*
show $ys' = [1..<n]$ **unfolding** *ys'-def* **by** *auto*
have *S''D*: $(S''\ \$\$\ (j,\ j) = D) \wedge (\forall j' \in \{0..<n\} - \{j\}.\ S''\ \$\$\ (j,\ j') = 0)$
if *jn*: $j<n$ **and** *j0*: $j>0$ **for** j
proof −
have $S''\ \$\$\ (j,\ i) = (D \cdot_m 1_m\ n)\ \$\$\ (j,i)$ **if** *i-n*: $i<n$ **for** i
proof −
have $S''\ \$\$\ (j,\ i) = swaprows\ 0\ m\ A\ \$\$\ (j+m,i)$
by (*metis S' S'' S-S'-S'' append-rows-nth2 mn nat-SN.compat i-n jn*)
also have $... = A\ \$\$\ (j+m,i)$ **using** *A jn j0 i-n* **by** *auto*
also have $... = (D \cdot_m 1_m\ n)\ \$\$\ (j,i)$
by (*smt* (*verit*) *A Groups.add-ac*(2) *add-mono-thms-linordered-field*(1) *append-rows-def A-def A'' i-n*
carrier-matD index-mat-four-block(1,2) *add-diff-cancel-right' not-add-less2 jn trans-less-add1*)
finally show *?thesis* **.**
qed
thus *?thesis* **using** *jn j0* **by** *auto*
qed
have $0 \notin set\ xs'$
proof −
have $A2\ \$\$\ (0,0) = p * A\ \$\$\ (m,\ 0) + q * A\ \$\$\ (0,\ 0)$
using *A A2-def n0* **by** *auto*
also have $... = gcd\ (A\ \$\$\ (m,\ 0))\ (A\ \$\$\ (0,\ 0))$
by (*metis euclid-ext2-works*(1) *euclid-ext2-works*(2) *pquvd2*)
also have $... = D$ **using** *Am0 A00 D-g0* **by** *auto*
finally have $A2\ \$\$\ (0,0) = D$ **.**
thus *?thesis* **unfolding** *xs'-def* **using** *D-g0* **by** *auto*
qed
thus $\forall j \in set\ xs'.\ j<n \wedge (S''\ \$\$\ (j,\ j) = D) \wedge (\forall j' \in \{0..<n\} - \{j\}.\ S''\ \$\$\ (j,\ j') = 0)$
using *S''D xs'-def* **by** *auto*
have $0 \notin set\ ys'$
proof −
have $A2\ \$\$\ (m,0) = u * A\ \$\$\ (m,\ 0) + v * A\ \$\$\ (0,\ 0)$
using *A A2-def n0 m0* **by** *auto*
also have $... = -\ A\ \$\$\ (0,\ 0)\ div\ gcd\ (A\ \$\$\ (m,\ 0))\ (A\ \$\$\ (0,\ 0)) * A\ \$\$\ (m,\ 0)$
$+\ A\ \$\$\ (m,\ 0)\ div\ gcd\ (A\ \$\$\ (m,\ 0))\ (A\ \$\$\ (0,\ 0)) * A\ \$\$\ (0,\ 0)$
by (*simp add*: *euclid-ext2-works*[*OF pquvd2*[*symmetric*]])
also have $... = 0$ **using** *A00 Am0* **by** *auto*
finally have $A2\ \$\$\ (m,0) = 0$ **.**
thus *?thesis* **unfolding** *ys'-def* **using** *D-g0* **by** *auto*
qed

```
        thus ∀j∈set ys'. j<n ∧ (S'' $$ (j, j) = D) ∧ (∀j'∈{0..<n}−{j}. S'' $$
(j, j') = 0)
          using S''D ys'-def by auto
        show swaprows 0 m A $$ (m, 0) ∈ {0, D} using Sm0 by blast
        thus swaprows 0 m A $$ (m, 0) = 0 ⟶ swaprows 0 m A $$ (0, 0) = D
          using S00 by linarith
      qed (insert D-g0)
      then show ?thesis by (simp add: False nz-m)
    next
      case False note xs-not-empty = False
      show ?thesis
    proof (unfold nz-xs-m, rule reduce-below-invertible-mat-case-m[OF - m0 n0 -
- mn d-xs all-less-m D-g0])
      let ?S' = mat-of-rows n (map (Matrix.row ?S) [0..<m])
      show ?S' ∈ carrier-mat m n by auto
      have l: length ?non-zero-positions > 1 using l False by blast
      hence nz0-less-m: ?non-zero-positions ! 0 < m
     by (metis One-nat-def add.commute add.left-neutral all-less-m append-Cons-nth-left

          length-append less-add-same-cancel1 list.size(3,4) nth-mem nz-xs-m)
      have ?S = ?S' @_r D ·_m 1_m n by (rule swaprows-append-id[OF A'' A-def
nz0-less-m])
      thus (if A $$ (0, 0) ≠ 0 then A else let i = (xs @ [m]) ! 0 in swaprows 0 i
A)= ?S' @_r D ·_m 1_m n
        using rw nz-xs-m by argo
      have A $$ (filter (λi. A $$ (i, 0) ≠ 0) [Suc 0..<dim-row A] ! 0, 0) ≠ 0
          by (metis (mono-tags, lifting) Cons-eq-filterD l length-nth-simps(1)
length-nth-simps(3) list.exhaust not-one-less-zero)
      then have ?S $$ (0, 0) ≠ 0
          by (metis A add-sign-intros(2) carrier-matD(1) carrier-matD(2) in-
dex-mat-swaprows(1) m0 n0)
      thus (if A $$ (0, 0) ≠ 0 then A else let i = (xs @ [m]) ! 0 in swaprows 0 i
A) $$ (0, 0) ≠ 0
        using rw nz-xs-m by algebra
    qed
  qed
 qed
  have inv2: ∃P. invertible-mat P ∧ P ∈ carrier-mat (m + n) (m + n)
    ∧ reduce-below-abs 0 ?non-zero-positions D ?A' = P * ?A'
  proof (cases A $$ (0,0) ≠0)
    case True
    show ?thesis
      by (unfold nz-xs-m, rule reduce-below-abs-invertible-mat-case-m
          [OF A'' m0 n0 - - mn d-xs all-less-m], insert A-def True D-g0, auto)
  next
    case False hence A00: A $$ (0,0) = 0 by auto
    let ?S = swaprows 0 (?non-zero-positions ! 0) A
     have rw: (if A $$ (0, 0) ≠ 0 then A else let i = ?non-zero-positions ! 0 in
swaprows 0 i A)
```

```
      = ?S using False by auto
    show ?thesis
    proof (cases xs = [])
      case True
      have nz-m: ?non-zero-positions = [m] using True nz-xs-m by simp
      obtain p q u v d where pquvd: (p,q,u,v,d) = euclid-ext2 (swaprows 0 m A
$$ (0, 0)) (swaprows 0 m A $$ (m, 0))
        by (metis prod-cases5)
      have Am0: A $$ (m,0) = D
      proof −
        have A $$ (m,0) = (D ·_m 1_m n) $$ (m−m, 0)
          unfolding A-def
          by (meson append-rows-nth3 assms(2) assms(4) less-add-same-cancel1
one-carrier-mat order.refl smult-carrier-mat)
        also have ... = D by (simp add: n0)
        finally show ?thesis .
      qed
      have Sm0: (swaprows 0 m A) $$ (m,0) = 0 using A False n0 by auto
      have S00: (swaprows 0 m A) $$ (0,0) = D using A Am0 n0 by auto
      have pquvd2: (p,q,u,v,d) = euclid-ext2 (A $$ (m, 0)) (A $$ (0, 0))
        using pquvd Sm0 S00 Am0 A00 by auto
    have reduce-below 0 ?non-zero-positions D ?A' = reduce 0 m D ?A' unfolding
nz-m by auto
    also have ... = reduce 0 m D (swaprows 0 m A) using True False rw nz-m
by auto
      have ∃P. invertible-mat P ∧ P ∈ carrier-mat (m + n) (m + n) ∧
        reduce-abs 0 m D (swaprows 0 m A) = P * (swaprows 0 m A)
      proof (rule reduce-abs-invertible-mat-case-m[OF - - m0 - - - - mn n0])
        show swaprows 0 m A $$ (0, 0) ≠ 0 using S00 D0 by auto
        define S' where S' = mat-of-rows n (map (Matrix.row ?S) [0..<m])
       define S'' where S'' = mat-of-rows n (map (Matrix.row ?S) [m..<m+n])
       define A2 where A2 = Matrix.mat (dim-row (swaprows 0 m A)) (dim-col
(swaprows 0 m A))
          (λ(i, k). if i = 0 then p * A $$ (m, k) + q * A $$ (0, k)
          else if i = m then u * A $$ (m, k) + v * A $$ (0, k) else A $$ (i, k))
        show S-S'-S'': swaprows 0 m A = S' @_r S'' unfolding S'-def S''-def
          by (metis A append-rows-split carrier-matD index-mat-swaprows(2,3)
le-add1 nth-Cons-0 nz-m)
        show S': S' ∈ carrier-mat m n unfolding S'-def by fastforce
        show S'': S'' ∈ carrier-mat n n unfolding S''-def by fastforce
        show 0 ≠ m using m0 by simp
        show (p,q,u,v,d) = euclid-ext2 (swaprows 0 m A $$ (0, 0)) (swaprows 0
m A $$ (m, 0))
          using pquvd by simp
        show A2 = Matrix.mat (dim-row (swaprows 0 m A)) (dim-col (swaprows
0 m A))
       (λ(i, k). if i = 0 then p * swaprows 0 m A $$ (0, k) + q * swaprows 0 m
A $$ (m, k)
       else if i = m then u * swaprows 0 m A $$ (0, k) + v * swaprows 0 m A $$
```

```
(m, k) else swaprows 0 m A $$ (i, k))
          (is - = ?rhs) using A A2-def by auto
        define xs' where xs' = filter (λi. abs (A2 $$ (0,i)) > D) [0..<n]
        define ys' where  ys' = filter (λi. abs (A2 $$ (m,i)) > D) [0..<n]
        show xs' = filter (λi. abs (A2 $$ (0,i)) > D) [0..<n] unfolding xs'-def
by auto
        show ys' = filter (λi. abs (A2 $$ (m,i)) > D) [0..<n] unfolding ys'-def
by auto
        have S''D: (S'' $$ (j, j) = D) ∧ (∀j'∈{0..<n}−{j}. S'' $$ (j, j') = 0)
          if jn: j<n and j0: j>0 for j
        proof −
          have S'' $$ (j, i) = (D ·m 1m n) $$ (j,i) if i-n: i<n for i
          proof −
            have S'' $$ (j, i) = swaprows 0 m A $$ (j+m,i)
             by (metis S' S'' S-S'-S'' append-rows-nth2 mn nat-SN.compat i-n jn)
            also have ... = A $$ (j+m,i) using A jn j0 i-n by auto
            also have ... = (D ·m 1m n) $$ (j,i)
             by (smt (verit) A Groups.add-ac(2) add-mono-thms-linordered-field(1)
append-rows-def A-def A'' i-n
                    carrier-matD index-mat-four-block(1,2) add-diff-cancel-right'
not-add-less2 jn trans-less-add1)
            finally show ?thesis .
          qed
          thus ?thesis using jn j0 by auto
        qed
        have 0 ∉ set xs'
        proof −
          have A2 $$ (0,0) = p * A $$ (m, 0) + q * A $$ (0, 0)
            using A A2-def n0 by auto
          also have ... = gcd (A $$ (m, 0)) (A $$ (0, 0))
            by (metis euclid-ext2-works(1) euclid-ext2-works(2) pquvd2)
          also have ... = D using Am0 A00 D-g0 by auto
          finally have A2 $$ (0,0) = D .
          thus ?thesis unfolding xs'-def using D-g0 by auto
        qed
         thus ∀j∈set xs'. j<n ∧ (S'' $$ (j, j) = D) ∧ (∀j'∈{0..<n}−{j}. S'' $$
(j, j') = 0)
          using S''D xs'-def by auto
        have 0 ∉ set ys'
        proof −
          have A2 $$ (m,0) = u * A $$ (m, 0) + v * A $$ (0, 0)
            using A A2-def n0 m0 by auto
          also have ... = − A $$ (0, 0) div gcd (A $$ (m, 0)) (A $$ (0, 0)) * A
$$ (m, 0)
            + A $$ (m, 0) div gcd (A $$ (m, 0)) (A $$ (0, 0)) * A $$ (0, 0)
            by (simp add: euclid-ext2-works[OF pquvd2[symmetric]])
          also have ... = 0 using A00 Am0 by auto
          finally have A2 $$ (m,0) = 0 .
          thus ?thesis unfolding ys'-def using D-g0 by auto
```

```
        qed
        thus ∀j∈set ys'. j<n ∧ (S'' $$ (j, j) = D) ∧ (∀j'∈{0..<n}−{j}. S'' $$
(j, j') = 0)
          using S''D ys'-def by auto
      qed (insert D-g0)
      then show ?thesis by (simp add: False nz-m)
    next
      case False note xs-not-empty = False
      show ?thesis
    proof (unfold nz-xs-m, rule reduce-below-abs-invertible-mat-case-m[OF - m0
n0 - - mn d-xs all-less-m D-g0])
      let ?S' = mat-of-rows n (map (Matrix.row ?S) [0..<m])
      show ?S' ∈ carrier-mat m n by auto
      have l: length ?non-zero-positions > 1 using l False by blast
      hence nz0-less-m: ?non-zero-positions ! 0 < m
    by (metis One-nat-def add.commute add.left-neutral all-less-m append-Cons-nth-left

          length-append less-add-same-cancel1 list.size(3,4) nth-mem nz-xs-m)
      have ?S = ?S' @r D ·m 1m n by (rule swaprows-append-id[OF A'' A-def
nz0-less-m])
      thus (if A $$ (0, 0) ≠ 0 then A else let i = (xs @ [m]) ! 0 in swaprows 0 i
A)= ?S' @r D ·m 1m n
        using rw nz-xs-m by argo
      have ?S $$ (0, 0) ≠ 0
          by (smt (verit) A l add-pos-pos carrier-matD index-mat-swaprows(1)
le-eq-less-or-eq length-greater-0-conv
            less-one linorder-not-less list.size(3) m0 mem-Collect-eq n0 nth-mem
set-filter)
      thus (if A $$ (0, 0) ≠ 0 then A else let i = (xs @ [m]) ! 0 in swaprows 0 i
A) $$ (0, 0) ≠ 0
        using rw nz-xs-m by algebra
    qed
  qed
 qed
 show ?thesis
 proof (cases abs-flag)
  case False
   from inv obtain P where inv-P: invertible-mat P and P: P ∈ carrier-mat
(m + n) (m + n)
    and r-PA': reduce-below 0 ?non-zero-positions D ?A' = P * ?A' by blast
  have Find-rw: FindPreHNF abs-flag D A = reduce-below 0 ?non-zero-positions
D ?A'
    using n0 A dr-A dc-A False * by (auto simp add: Let-def)
  have ∃P. P ∈ carrier-mat (m + n) (m + n) ∧ invertible-mat P ∧ ?A' = P *
A
   by (rule A'-swaprows-invertible-mat[OF A], insert non-zero-positions-xs-m n0
m0 l nz-xs-m, auto)
  from this obtain Q where Q: Q ∈ carrier-mat (m + n) (m + n)
    and inv-Q: invertible-mat Q and A'-QA: ?A' = Q * A by blast
```

have *reduce-below 0 ?non-zero-positions D ?A′ = (P * Q) * A* **using** *Q A′-QA P r-PA′ A* **by** *auto*
moreover have *invertible-mat (P*Q)* **using** *P Q inv-P inv-Q invertible-mult-JNF* **by** *blast*
moreover have *(P*Q) ∈ carrier-mat (m + n) (m + n)* **using** *P Q* **by** *auto*
ultimately show *?thesis* **using** *Find-rw* **by** *metis*
next
case *True*
from *inv2* **obtain** *P* **where** *inv-P*: *invertible-mat P* **and** *P*: *P ∈ carrier-mat (m + n) (m + n)*
and *r-PA′*: *reduce-below-abs 0 ?non-zero-positions D ?A′ = P * ?A′* **by** *blast*
have *Find-rw*: *FindPreHNF abs-flag D A = reduce-below-abs 0 ?non-zero-positions D ?A′*
using *n0 A dr-A dc-A True* * **by** (*auto simp add*: *Let-def*)
have *∃ P. P ∈ carrier-mat (m + n) (m + n) ∧ invertible-mat P ∧ ?A′ = P * A*
by (*rule A′-swaprows-invertible-mat*[*OF A*], *insert non-zero-positions-xs-m n0 m0 l nz-xs-m*, *auto*)
from *this* **obtain** *Q* **where** *Q*: *Q ∈ carrier-mat (m + n) (m + n)*
and *inv-Q*: *invertible-mat Q* **and** *A′-QA*: *?A′ = Q * A* **by** *blast*
have *reduce-below-abs 0 ?non-zero-positions D ?A′ = (P * Q) * A* **using** *Q A′-QA P r-PA′ A* **by** *auto*
moreover have *invertible-mat (P*Q)* **using** *P Q inv-P inv-Q invertible-mult-JNF* **by** *blast*
moreover have *(P*Q) ∈ carrier-mat (m + n) (m + n)* **using** *P Q* **by** *auto*
ultimately show *?thesis* **using** *Find-rw* **by** *metis*
qed
qed
qed

corollary *FindPreHNF-echelon-form-mx0*:
assumes *A ∈ carrier-mat m 0*
shows *echelon-form-JNF (FindPreHNF abs-flag D A)*
by (*rule echelon-form-mx0*, *rule FindPreHNF*[*OF assms*])

lemma *FindPreHNF-echelon-form-mx1*:
assumes *A-def*: *A = A″ @$_r$ D ·$_m$ 1$_m$ n*
and *A″*: *A″ ∈ carrier-mat m n* **and** *n2*: *n<2* **and** *D-g0*: *D>0* **and** *mn*: *m≥n*
shows *echelon-form-JNF (FindPreHNF abs-flag D A)*
proof (*cases n=0*)
case *True*
have *A*: *A ∈ carrier-mat m 0* **using** *A-def A″ True*
by (*metis add.comm-neutral append-rows-def carrier-matD carrier-matI index-mat-four-block*(*2*,*3*)
index-one-mat(*2*) *index-smult-mat*(*2*) *index-zero-mat*(*2*,*3*))
show *?thesis* **unfolding** *True* **by** (*rule FindPreHNF-echelon-form-mx0*, *insert A*, *auto*)

```
next
  case False hence n0: 0<n by auto
  have A: A ∈ carrier-mat (m+n) n using A-def A'' by auto
  have m0: m>0 using mn n2 n0 by auto
  have D0: D≠0 using D-g0 by auto
  show ?thesis
  proof (cases m+n<2)
    case True
   show ?thesis by (rule echelon-form-JNF-1xn[OF - True], rule FindPreHNF[OF
A])
  next
    case False note mn-le-2 = False
    have dr-A: dim-row A ≥2 using False n2 A by auto
    have dc-A: dim-col A < 2 using n2 A by auto
    let ?non-zero-positions = filter (λi. A $$ (i, 0) ≠ 0) [Suc 0..<dim-row A]
     let ?A' = (if A $$ (0, 0) ≠ 0 then A else let i = ?non-zero-positions ! 0 in
swaprows 0 i A)
    define xs where xs = filter (λi. A $$ (i,0) ≠ 0) [1..<m]
    let ?Reduce = (if abs-flag then reduce-below-abs else reduce-below)
    have nz-xs-m: ?non-zero-positions = xs @ [m] and d-xs: distinct xs
      and all-less-m: ∀ x∈set xs. x < m ∧ 0 < x
      using non-zero-positions-xs-m'[OF A-def A'' - m0 n0 D0] using D0 A un-
folding xs-def by auto
     have ∗: FindPreHNF abs-flag D A = (if abs-flag then reduce-below-abs 0
?non-zero-positions D ?A'
       else reduce-below 0 ?non-zero-positions D ?A')
      using dr-A dc-A by (auto simp add: Let-def)
    have l: length ?non-zero-positions > 1 if xs≠[] using that unfolding nz-xs-m
by auto
    have e: echelon-form-JNF (reduce-below 0 ?non-zero-positions D ?A')
    proof (cases A $$ (0,0) ≠0)
      case True note A00 = True
     have 1: reduce-below 0 ?non-zero-positions D ?A' = reduce-below 0 ?non-zero-positions
D A
       using True by auto
      have echelon-form-JNF (reduce-below 0 ?non-zero-positions D A)
      proof (rule echelon-form-JNF-mx1[OF - n2])
       show reduce-below 0 ?non-zero-positions D A ∈ carrier-mat (m+n) n using
A by auto
        show ∀ i∈{1..<m + n}. reduce-below 0 ?non-zero-positions D A $$ (i, 0)
= 0
       proof
         fix i assume i: i ∈ {1..<m + n}
         show reduce-below 0 ?non-zero-positions D A $$ (i, 0) =0
         proof (cases i∈set ?non-zero-positions)
           case True
           show ?thesis unfolding nz-xs-m
               by (rule reduce-below-0-case-m[OF A'' m0 n0 A-def A00 mn - d-xs
all-less-m D-g0],
```

```
            insert nz-xs-m True, auto)
        next
          case False note i-notin-set = False
           have reduce-below 0 ?non-zero-positions D A $$ (i, 0) = A $$ (i, 0)
unfolding nz-xs-m
            by (rule reduce-below-preserves-case-m[OF A'' m0 n0 A-def A00 mn -
d-xs all-less-m - - - D-g0],
                insert i nz-xs-m i-notin-set, auto)
          also have ... = 0 using i-notin-set i A unfolding set-filter by auto
          finally show ?thesis .
        qed
       qed
      qed
      thus ?thesis using 1 by argo
    next
      case False hence A00: A $$ (0,0) = 0 by simp
      let ?i = ((xs @ [m]) ! 0)
      let ?S = swaprows 0 ?i A
      let ?S' = mat-of-rows n (map (Matrix.row (swaprows 0 ?i A)) [0..<m])
       have rw: (if A$$(0, 0) ≠ 0 then A else let i = ?non-zero-positions!0 in
swaprows 0 i A) = ?S
        using A00 nz-xs-m by auto
      have S: ?S ∈ carrier-mat (m+n) n using A by auto
      have A00-eq-A'00: A $$ (0, 0) = A'' $$ (0, 0)
     by (metis A'' A-def add-gr-0 append-rows-def n0 carrier-matD index-mat-four-block(1)
m0)
      show ?thesis
      proof (cases xs=[])
        case True
        have nz-m: ?non-zero-positions = [m] using True nz-xs-m by simp
        obtain p q u v d where pquvd: (p,q,u,v,d) = euclid-ext2 (swaprows 0 m A
$$ (0, 0)) (swaprows 0 m A $$ (m, 0))
          by (metis prod-cases5)
        have Am0: A $$ (m,0) = D
        proof −
          have A $$ (m,0) = (D ·_m 1_m n) $$ (m−m, 0)
          by (smt (verit) A append-rows-def A-def A'' n0 carrier-matD diff-self-eq-0
index-mat-four-block
            less-add-same-cancel1 less-diff-conv ordered-cancel-comm-monoid-diff-class.diff-add
              nat-less-le)
          also have ... = D by (simp add: n0)
          finally show ?thesis .
        qed
        have Sm0: (swaprows 0 m A) $$ (m,0) = 0 using A False n0 by auto
        have S00: (swaprows 0 m A) $$ (0,0) = D using A Am0 n0 by auto
        have pquvd2: (p,q,u,v,d) = euclid-ext2 (A $$ (m, 0)) (A $$ (0, 0))
          using pquvd Sm0 S00 Am0 A00 by auto
      have reduce-below 0 ?non-zero-positions D ?A' = reduce 0 m D ?A' unfolding
nz-m by auto
```

also have ... = *reduce 0 m D* (*swaprows 0 m A*) **using** *True False rw nz-m* **by** *auto*
finally have ∗: *reduce-below 0 ?non-zero-positions D ?A′ = reduce 0 m D* (*swaprows 0 m A*) **.**
have *echelon-form-JNF* (*reduce 0 m D* (*swaprows 0 m A*))
proof (*rule echelon-form-JNF-mx1*[*OF - n2*])
show *reduce 0 m D* (*swaprows 0 m A*) ∈ *carrier-mat* (*m*+*n*) *n*
using *A n2 reduce-carrier* **by** (*auto simp add*: *Let-def*)
show ∀ *i*∈{*1..*<*m*+*n*}. *reduce 0 m D* (*swaprows 0 m A*) $$ (*i*, *0*) = *0*
proof
fix *i* **assume** *i*: *i* ∈ {*1..*<*m*+*n*}
show *reduce 0 m D* (*swaprows 0 m A*) $$ (*i*, *0*) = *0*
proof (*cases i*=*m*)
case *True*
show *?thesis*
proof (*unfold True*, *rule reduce-0*[*OF - - n0*])
show *swaprows 0 m A* ∈ *carrier-mat* (*m*+*n*) *n* **using** *A* **by** *auto*
qed (*insert m0 n0 S00 D-g0*, *auto*)
next
case *False*
have *reduce 0 m D* (*swaprows 0 m A*) $$ (*i*, *0*) = (*swaprows 0 m A*) $$ (*i*, *0*)
proof (*rule reduce-preserves*[*OF - n0*])
show *swaprows 0 m A* ∈ *carrier-mat* (*m*+*n*) *n* **using** *A* **by** *auto*
qed (*insert m0 n0 S00 D-g0 False i*, *auto*)
also have ... = *A* $$ (*i*, *0*) **using** *i False A n0* **by** *auto*
also have ... = *0*
proof (*rule ccontr*)
assume *A* $$ (*i*, *0*) ≠ *0* **hence** *i* ∈ *set ?non-zero-positions* **using** *i A* **by** *auto*
hence *i*=*m* **using** *nz-xs-m True* **by** *auto*
thus *False* **using** *False* **by** *contradiction*
qed
finally show *?thesis* **.**
qed
qed
qed
then show *?thesis* **using** ∗ **by** *presburger*
next
case *False*
have *l*: *length ?non-zero-positions* > *1* **using** *False nz-xs-m* **by** *auto*
hence *l-xs*: *length xs* > *0* **using** *nz-xs-m* **by** *auto*
hence *xs-m-less-m*: (*xs*@[*m*]) ! *0* < *m* **by** (*simp add*: *all-less-m nth-append*)
have *A* $$ ((*xs* @ [*m*]) ! *0*, *0*) ≠ *0*
by (*metis* (*mono-tags*, *lifting*) *Cons-eq-filterD List.min-list.cases append-is-Nil-conv nth-Cons-0 nz-xs-m*)
then have *S00*: *?S* $$ (*0*,*0*) ≠ *0*
using *A n0* **by** *auto*
have *S′*: *?S′* ∈ *carrier-mat m n* **using** *A* **by** *auto*

```
      have S-S'D: ?S = ?S' @_r D ·_m 1_m n by (rule swaprows-append-id[OF A''
A-def xs-m-less-m])
    have 2: reduce-below 0 ?non-zero-positions D ?A' = reduce-below 0 ?non-zero-positions
D ?S
       using A00 nz-xs-m by algebra
     have echelon-form-JNF (reduce-below 0 ?non-zero-positions D ?S)
     proof (rule echelon-form-JNF-mx1[OF - n2])
      show reduce-below 0 ?non-zero-positions D ?S ∈ carrier-mat (m+n) n using
A by auto
       show ∀ i∈{1..<m + n}. reduce-below 0 ?non-zero-positions D ?S $$ (i, 0)
= 0
       proof
         fix i assume i: i ∈ {1..<m + n}
         show reduce-below 0 ?non-zero-positions D ?S $$ (i, 0) =0
         proof (cases i∈set ?non-zero-positions)
           case True
           show ?thesis unfolding nz-xs-m
               by (rule reduce-below-0-case-m[OF S' m0 n0 S-S'D S00 mn - d-xs
all-less-m D-g0],
                 insert True nz-xs-m, auto)
         next
           case False note i-notin-set = False
            have reduce-below 0 ?non-zero-positions D ?S $$ (i, 0) = ?S $$ (i, 0)
unfolding nz-xs-m
              by (rule reduce-below-preserves-case-m[OF S' m0 n0 S-S'D S00 mn -
d-xs all-less-m - - - D-g0],
                 insert i nz-xs-m i-notin-set, auto)
           also have ... = 0 using i-notin-set i A S00 n0 unfolding set-filter by
auto
           finally show ?thesis .
         qed
       qed
     qed
     thus ?thesis using 2 by argo
   qed
  qed
   have e2: echelon-form-JNF (reduce-below-abs 0 ?non-zero-positions D ?A')
   proof (cases A $$ (0,0) ≠0)
     case True note A00 = True
      have 1: reduce-below-abs 0 ?non-zero-positions D ?A' = reduce-below-abs 0
?non-zero-positions D A
       using True by auto
     have echelon-form-JNF (reduce-below-abs 0 ?non-zero-positions D A)
     proof (rule echelon-form-JNF-mx1[OF - n2])
       show reduce-below-abs 0 ?non-zero-positions D A ∈ carrier-mat (m+n) n
using A by auto
       show ∀ i∈{1..<m + n}. reduce-below-abs 0 ?non-zero-positions D A $$ (i,
0) = 0
       proof
```

```
        fix i assume i: i ∈ {1..<m + n}
        show reduce-below-abs 0 ?non-zero-positions D A $$ (i, 0) =0
        proof (cases i∈set ?non-zero-positions)
          case True
          show ?thesis unfolding nz-xs-m
            by (rule reduce-below-abs-0-case-m[OF A'' m0 n0 A-def A00 mn - d-xs
all-less-m D-g0],
                insert nz-xs-m True, auto)
        next
          case False note i-notin-set = False
          have reduce-below-abs 0 ?non-zero-positions D A $$ (i, 0) = A $$ (i, 0)
unfolding nz-xs-m
            by (rule reduce-below-abs-preserves-case-m[OF A'' m0 n0 A-def A00
mn - d-xs all-less-m - - - D-g0],
                insert i nz-xs-m i-notin-set, auto)
          also have ... = 0 using i-notin-set i A unfolding set-filter by auto
          finally show ?thesis .
        qed
      qed
    qed
    thus ?thesis using 1 by argo
  next
    case False hence A00: A $$ (0,0) = 0 by simp
    let ?i = ((xs @ [m]) ! 0)
    let ?S = swaprows 0 ?i A
    let ?S' = mat-of-rows n (map (Matrix.row (swaprows 0 ?i A)) [0..<m])
      have rw: (if A$$(0, 0) ≠ 0 then A else let i = ?non-zero-positions!0 in
swaprows 0 i A) = ?S
      using A00 nz-xs-m by auto
    have S: ?S ∈ carrier-mat (m+n) n using A by auto
    have A00-eq-A'00: A $$ (0, 0) = A'' $$ (0, 0)
    by (metis A'' A-def add-gr-0 append-rows-def n0 carrier-matD index-mat-four-block(1)
m0)
    show ?thesis
    proof (cases xs=[])
      case True
      have nz-m: ?non-zero-positions = [m] using True nz-xs-m by simp
      obtain p q u v d where pquvd: (p,q,u,v,d) = euclid-ext2 (swaprows 0 m A
$$ (0, 0)) (swaprows 0 m A $$ (m, 0))
        by (metis prod-cases5)
      have Am0: A $$ (m,0) = D
      proof −
        have A $$ (m,0) = (D ·_m 1_m n) $$ (m−m, 0)
        by (smt (verit) A append-rows-def A-def A'' n0 carrier-matD diff-self-eq-0
index-mat-four-block
          less-add-same-cancel1 less-diff-conv ordered-cancel-comm-monoid-diff-class.diff-add
            nat-less-le)
        also have ... = D by (simp add: n0)
        finally show ?thesis .
```

```
      qed
      have Sm0: (swaprows 0 m A) $$ (m,0) = 0 using A False n0 by auto
      have S00: (swaprows 0 m A) $$ (0,0) = D using A Am0 n0 by auto
      have pquvd2: (p,q,u,v,d) = euclid-ext2 (A $$ (m, 0)) (A $$ (0, 0))
        using pquvd Sm0 S00 Am0 A00 by auto
      have reduce-below-abs 0 ?non-zero-positions D ?A' = reduce-abs 0 m D ?A'
unfolding nz-m by auto
       also have ... = reduce-abs 0 m D (swaprows 0 m A) using True False rw
nz-m by auto
       finally have *: reduce-below-abs 0 ?non-zero-positions D ?A' = reduce-abs
0 m D (swaprows 0 m A) .
      have echelon-form-JNF (reduce-abs 0 m D (swaprows 0 m A))
      proof (rule echelon-form-JNF-mx1[OF - n2])
        show reduce-abs 0 m D (swaprows 0 m A) ∈ carrier-mat (m+n) n
          using A n2 reduce-carrier by (auto simp add: Let-def)
        show ∀ i∈{1..<m+n}. reduce-abs 0 m D (swaprows 0 m A) $$ (i, 0) = 0
        proof
          fix i assume i: i ∈ {1..<m+n}
          show reduce-abs 0 m D (swaprows 0 m A) $$ (i, 0) = 0
          proof (cases i=m)
            case True
            show ?thesis
            proof (unfold True, rule reduce-0[OF - - n0])
              show swaprows 0 m A ∈ carrier-mat (m+n) n using A by auto
            qed (insert m0 n0 S00 D-g0, auto)
          next
            case False
             have reduce-abs 0 m D (swaprows 0 m A) $$ (i, 0) = (swaprows 0 m
A) $$ (i, 0)
            proof (rule reduce-preserves[OF - n0])
              show swaprows 0 m A ∈ carrier-mat (m+n) n using A by auto
            qed (insert m0 n0 S00 D-g0 False i, auto)
            also have ... = A $$ (i, 0) using i False A n0 by auto
            also have ... = 0
            proof (rule ccontr)
              assume A $$ (i, 0) ≠ 0 hence i ∈ set ?non-zero-positions using i
A by auto
              hence i=m using nz-xs-m True by auto
              thus False using False by contradiction
            qed
            finally show ?thesis .
          qed
        qed
      qed
      then show ?thesis using * by presburger
    next
      case False
    have l: length ?non-zero-positions > 1    using False nz-xs-m by auto
    hence l-xs: length xs > 0 using nz-xs-m by auto
```

hence *xs-m-less-m*: (*xs*@[*m*]) ! *0* < *m* **by** (*simp add*: *all-less-m nth-append*)
have *A* $$ ((*xs* @ [*m*]) ! *0*, *0*) ≠ *0*
by (*smt* (*verit*) *append-Cons-nth-left l-xs mem-Collect-eq nth-mem set-filter xs-def*)
then have *S00*: *?S* $$ (*0*,*0*) ≠ *0*
using *A n0* **by** *auto*
have *S'*: *?S'* ∈ *carrier-mat m n* **using** *A* **by** *auto*
have *S-S'D*: *?S* = *?S'* @$_r$ *D* ·$_m$ *1*$_m$ *n* **by** (*rule swaprows-append-id*[*OF A'' A-def xs-m-less-m*])
have *2*: *reduce-below-abs 0 ?non-zero-positions D ?A'* = *reduce-below-abs 0 ?non-zero-positions D ?S*
using *A00 nz-xs-m* **by** *algebra*
have *echelon-form-JNF* (*reduce-below-abs 0 ?non-zero-positions D ?S*)
proof (*rule echelon-form-JNF-mx1*[*OF - n2*])
show *reduce-below-abs 0 ?non-zero-positions D ?S* ∈ *carrier-mat* (*m*+*n*) *n* **using** *A* **by** *auto*
show ∀ *i*∈{*1*..<*m* + *n*}. *reduce-below-abs 0 ?non-zero-positions D ?S* $$ (*i*, *0*) = *0*
proof
fix *i* **assume** *i*: *i* ∈ {*1*..<*m* + *n*}
show *reduce-below-abs 0 ?non-zero-positions D ?S* $$ (*i*, *0*) =*0*
proof (*cases i*∈*set ?non-zero-positions*)
case *True*
show *?thesis* **unfolding** *nz-xs-m*
by (*rule reduce-below-abs-0-case-m*[*OF S' m0 n0 S-S'D S00 mn - d-xs all-less-m D-g0*],
insert True nz-xs-m, *auto*)
next
case *False* **note** *i-notin-set* = *False*
have *reduce-below-abs 0 ?non-zero-positions D ?S* $$ (*i*, *0*) = *?S* $$ (*i*, *0*) **unfolding** *nz-xs-m*
by (*rule reduce-below-abs-preserves-case-m*[*OF S' m0 n0 S-S'D S00 mn - d-xs all-less-m - - - D-g0*],
insert i nz-xs-m i-notin-set, *auto*)
also have ... = *0* **using** *i-notin-set i A S00 n0* **unfolding** *set-filter* **by** *auto*
finally show *?thesis* **.**
qed
qed
qed
thus *?thesis* **using** *2* **by** *argo*
qed
qed
thus *?thesis* **using** ∗ *e* **by** *presburger*
qed
qed

lemma *FindPreHNF-works-n-ge2*:

assumes *A-def*: $A = A'' \,@_r\, D \cdot_m 1_m\, n$
and *A''*: $A'' \in$ *carrier-mat m n* **and** $n \geq 2$ **and** *m-le-n*: $m \geq n$ **and** $D > 0$
shows $\exists P.\ P \in$ *carrier-mat* $(m+n)\ (m+n) \wedge$ *invertible-mat* $P \wedge$ *FindPreHNF abs-flag D A* $= P * A \wedge$ *echelon-form-JNF* (*FindPreHNF abs-flag D A*)
using *assms*
proof (*induct abs-flag D A arbitrary*: *A'' m n rule*: *FindPreHNF.induct*)
case (*1 abs-flag D A*)
note *A-def* = *1.prems*(*1*)
note *A''* = *1.prems*(*2*)
note *n* = *1.prems*(*3*)
note *m-le-n* = *1.prems*(*4*)
note *D0* = *1.prems*(*5*)
let *?RAT* = *map-mat rat-of-int*
have *A*: $A \in$ *carrier-mat* $(m+n)\ n$ **using** *A-def A''* **by** *auto*
have *mn*: $2 \leq m+n$ **using** *n* **by** *auto*
have *m0*: $0 < m$ **using** *n m-le-n* **by** *auto*
have *n0*: $0 < n$ **using** *n* **by** *simp*
have *D-not0*: $D \neq 0$ **using** *D0* **by** *auto*
define *non-zero-positions* **where** *non-zero-positions* = *filter* $(\lambda i.\ A\ \$\$\ (i,0) \neq 0)\ [1..<$*dim-row A*$]$
define *A'* **where** $A' =$ (*if A* \$\$ $(0,\ 0) \neq 0$ *then A else let i* = *non-zero-positions* ! *0 in swaprows 0 i A*)
let *?Reduce* = (*if abs-flag then reduce-below-abs else reduce-below*)
obtain *A'-UL A'-UR A'-DL A'-DR* **where** *A'-split*: (*A'-UL*, *A'-UR*, *A'-DL*, *A'-DR*)
= *split-block* (*?Reduce 0 non-zero-positions D* (*make-first-column-positive A'*)) *1 1*
by (*metis prod-cases4*)
define *sub-PreHNF* **where** *sub-PreHNF* = *FindPreHNF abs-flag D A'-DR*
obtain *xs* **where** *non-zero-positions-xs-m*: *non-zero-positions* = *xs* @ $[m]$ **and** *d-xs*: *distinct xs*
and *all-less-m*: $\forall x \in$*set xs*. $x < m \wedge 0 < x$
using *non-zero-positions-xs-m*[*OF A-def A'' non-zero-positions-def m0 n0*] **using** *D0* **by** *fast*
define *M* **where** *M* = (*make-first-column-positive A'*)
have *A'*: $A' \in$ *carrier-mat* $(m+n)\ n$ **unfolding** *A'-def* **using** *A* **by** *auto*
have *mk-A'-not0*:*make-first-column-positive A'* \$\$ $(0,0) \neq 0$
by (*rule make-first-column-positive-00*[*OF A-def A'' non-zero-positions-def A'-def m0 n0 D-not0 m-le-n*])
have *M*: $M \in$ *carrier-mat* $(m+n)\ n$ **using** *A' M-def* **by** *auto*
let *?M'* = *mat-of-rows n* (*map* (*Matrix.row* (*make-first-column-positive A'*)) $[0..<m]$)
have *M'*: *?M'* $\in$ *carrier-mat m n* **by** *auto*
have *M-M'D*: *make-first-column-positive A'* = *?M'* $@_r\ D \cdot_m 1_m\ n$ **if** *xs-empty*: $xs \neq []$
proof (*cases A*\$\$$(0,0) \neq 0$)
case *True*
then have $*$: *make-first-column-positive A'* = *make-first-column-positive A*
unfolding *A'-def* **by** *auto*

```
    show ?thesis
     by (unfold *, rule make-first-column-positive-append-id[OF A'' A-def D0 n0])
  next
    case False
    then have *: make-first-column-positive A'
                 = make-first-column-positive (swaprows 0 (non-zero-positions ! 0)
A)
      unfolding A'-def by auto
    show ?thesis
    proof (unfold *, rule make-first-column-positive-append-id)
      let ?S = mat-of-rows n (map (Matrix.row (swaprows 0 (non-zero-positions !
0) A)) [0..<m])
      show swaprows 0 (non-zero-positions ! 0) A =  ?S @_r (D ·_m (1_m n))
      proof (rule swaprows-append-id[OF A'' A-def])
        have A''00: A'' $$ (0, 0) = 0
            by (metis (no-types, lifting) A A'' A-def False add-sign-intros(2) ap-
pend-rows-def
             carrier-matD index-mat-four-block m0 n0)
        have length-xs: length xs > 0 using xs-empty by auto
        have non-zero-positions ! 0 = xs ! 0 unfolding non-zero-positions-xs-m
          by (meson length-xs nth-append)
        thus non-zero-positions ! 0 < m using all-less-m length-xs by simp
      qed
    qed (insert n0 D0, auto)
  qed
  have A'-DR: A'-DR ∈ carrier-mat (m + (n−1)) (n−1)
    by (rule split-block(4)[OF A'-split[symmetric]], insert n M M-def, auto)
  have sub-PreHNF: sub-PreHNF ∈ carrier-mat (m + (n −1)) (n−1)
    unfolding sub-PreHNF-def by (rule FindPreHNF[OF A'-DR])
  hence sub-PreHNF': sub-PreHNF ∈ carrier-mat (m+n − 1) (n−1) using n by
auto
  have A'-UL: A'-UL ∈ carrier-mat 1 1
     by (rule split-block(1)[OF A'-split[symmetric], of m+n−1 n−1], insert n A',
auto)
  have A'-UR: A'-UR ∈ carrier-mat 1 (n−1)
    by (rule split-block(2)[OF A'-split[symmetric], of m+n−1], insert n A', auto)
  have A'-DL: A'-DL ∈ carrier-mat (m + (n − 1)) 1
    by (rule split-block(3)[OF A'-split[symmetric], of - n−1], insert n A', auto)

  show ?case
  proof (cases abs-flag)
    case True note abs-flag = True
      hence A'-split: (A'-UL, A'-UR, A'-DL, A'-DR)
   = split-block (reduce-below-abs 0 non-zero-positions D (make-first-column-positive
A')) 1 1   using A'-split by auto
     let ?R = reduce-below-abs 0 non-zero-positions D (make-first-column-positive
A')
    have fbm-R: four-block-mat A'-UL A'-UR A'-DL A'-DR
     = reduce-below-abs 0 non-zero-positions D (make-first-column-positive A')
```

by (*rule split-block*(5)[*symmetric*, *OF A′-split*[*symmetric*], *of m+n−1 n−1*], *insert A′ n*, *auto*)
have *A′-DL0*: *A′-DL* = (*0*$_m$ (*m* + (*n* − *1*)) *1*)
proof (*rule eq-matI*)
show *dim-row A′-DL* = *dim-row* (*0*$_m$ (*m* + (*n* − *1*)) *1*)
and *dim-col A′-DL* = *dim-col* (*0*$_m$ (*m* + (*n* − *1*)) *1*) **using** *A′-DL* **by** *auto*

fix *i j* **assume** *i*: *i* < *dim-row* (*0*$_m$ (*m* + (*n* − *1*)) *1*) **and** *j*: *j* < *dim-col* (*0*$_m$ (*m* + (*n* − *1*)) *1*)
have *j0*: *j*=*0* **using** *j* **by** *auto*
have *0* = *?R* $$ (*i*+*1*,*j*)
proof (*unfold M-def non-zero-positions-xs-m j0*,
rule reduce-below-abs-0-case-m-make-first-column-positive[*symmetric*,
OF A″ m0 n0 A-def m-le-n - d-xs all-less-m - - D0 -])
show *A′* = (*if A* $$ (*0*, *0*) ≠ *0 then A else let i* = (*xs* @ [*m*]) ! *0 in swaprows 0 i A*)
using *A′-def non-zero-positions-def non-zero-positions-xs-m* **by** *presburger*
show *xs* @ [*m*] = *filter* (λ*i*. *A* $$ (*i*, *0*) ≠ *0*) [*1*..<*dim-row A*]
using *A′-def non-zero-positions-def non-zero-positions-xs-m* **by** *presburger*
qed (*insert i n0*, *auto*)
also have ... = *four-block-mat A′-UL A′-UR A′-DL A′-DR* $$ (*i*+*1*,*j*) **unfolding** *fbm-R* **..**
also have ... = (*if i*+*1* < *dim-row A′-UL then if j* < *dim-col A′-UL*
then A′-UL $$ (*i*+*1*, *j*) *else A′-UR* $$ (*i*+*1*, *j* − *dim-col A′-UL*)
else if j < *dim-col A′-UL then A′-DL* $$ (*i*+*1* − *dim-row A′-UL*, *j*)
else A′-DR $$ (*i*+*1* − *dim-row A′-UL*, *j* − *dim-col A′-UL*))
by (*rule index-mat-four-block*, *insert A′-UL A′-DR i j*, *auto*)
also have ... = *A′-DL* $$ (*i*, *j*) **using** *A′-UL A′-UR i j* **by** *auto*
finally show *A′-DL* $$ (*i*, *j*) = *0*$_m$ (*m* + (*n* − *1*)) *1* $$ (*i*, *j*) **using** *i j* **by** *auto*
qed

let *?A′-DR-m* = *mat-of-rows* (*n*−*1*) [*Matrix.row A′-DR i*. *i* ← [*0*..<*m*]]
have *A′-DR-m*: *?A′-DR-m* ∈ *carrier-mat m* (*n*−*1*) **by** *auto*
have *A′DR-A′DR-m-D*: *A′-DR* = *?A′-DR-m* @$_r$ *D* ·$_m$ *1*$_m$ (*n* − *1*)
proof (*rule eq-matI*)
show *dr*: *dim-row A′-DR* = *dim-row* (*?A′-DR-m* @$_r$ *D* ·$_m$ *1*$_m$ (*n* − *1*))
by (*metis A′-DR A′-DR-m append-rows-def carrier-matD*(*1*) *index-mat-four-block*(*2*)
index-one-mat(*2*) *index-smult-mat*(*2*) *index-zero-mat*(*2*))
show *dc*: *dim-col A′-DR* = *dim-col* (*?A′-DR-m* @$_r$ *D* ·$_m$ *1*$_m$ (*n* − *1*))
by (*metis A′-DR A′-DR-m add.comm-neutral append-rows-def*
carrier-matD(*2*) *index-mat-four-block*(*3*) *index-zero-mat*(*3*))
fix *i j* **assume** *i*: *i* < *dim-row*(*?A′-DR-m* @$_r$ *D* ·$_m$ *1*$_m$ (*n* − *1*))
and *j*: *j*<*dim-col* (*?A′-DR-m* @$_r$ *D* ·$_m$ *1*$_m$ (*n* − *1*))
have *jn1*: *j*<*n*−*1* **using** *dc j A′-DR* **by** *auto*
show *A′-DR* $$ (*i*,*j*) = (*?A′-DR-m* @$_r$ *D* ·$_m$ *1*$_m$ (*n* − *1*)) $$ (*i*,*j*)
proof (*cases i*<*m*)
case *True*

```
    have A'-DR $$ (i,j) = ?A'-DR-m $$ (i,j)
      by (metis A'-DR A'-DR-m True dc carrier-matD(1) carrier-matD(2) j
le-add1
        map-first-rows-index mat-of-rows-carrier(2) mat-of-rows-index)
    also have ... = (?A'-DR-m @r D ·m 1m (n − 1)) $$ (i,j)
      by (metis (mono-tags, lifting) A'-DR A'-DR-m True append-rows-def
        carrier-matD dc i index-mat-four-block j)
    finally show ?thesis .
  next
    case False note i-ge-m = False
  let ?reduce-below = reduce-below-abs 0 non-zero-positions D (make-first-column-positive
A')
    have 1: (?A'-DR-m @r D ·m 1m (n − 1)) $$ (i,j) = (D ·m 1m (n − 1)) $$
(i−m,j)
      by (smt (verit) A'-DR A'-DR-m False append-rows-nth carrier-matD car-
rier-mat-triv dc dr i
        index-one-mat(2) index-one-mat(3) index-smult-mat(2,3) j)
    have ?reduce-below = four-block-mat A'-UL A'-UR A'-DL A'-DR using fbm-R
..
    also have ... $$ (i+1,j+1) = (if i+1 < dim-row A'-UL then if j+1 < dim-col
A'-UL
        then A'-UL $$ (i+1, j+1) else A'-UR $$ (i+1, j+1 − dim-col A'-UL)
        else if j+1 < dim-col A'-UL then A'-DL $$ (i+1 − dim-row A'-UL,
j+1)
        else A'-DR $$ (i+1 − dim-row A'-UL, j+1 − dim-col A'-UL))
      by (rule index-mat-four-block, insert i j A'-UL A'-DR dr dc, auto)
    also have ... = A'-DR $$ (i,j) using A'-UL by auto
    finally have 2: ?reduce-below $$ (i+1,j+1) = A'-DR $$ (i,j) .
    show ?thesis
    proof (cases xs = [])
      case True note xs-empty = True
      have i1-m: i + 1 ≠ m
        using False less-add-one by blast
      have j1n: j+1<n
        using jn1 less-diff-conv by blast
      have i1-mn: i+1<m + n
        using i i-ge-m
     by (metis A'-DR carrier-matD(1) dr less-diff-conv sub-PreHNF sub-PreHNF')
      have ?reduce-below = reduce-abs 0 m D M
        unfolding non-zero-positions-xs-m xs-empty M-def by auto
      also have ... $$ (i+1,j+1) = M $$ (i+1, j+1)
     by (rule reduce-preserves[OF M j1n - i1-m - i1-mn], insert M-def mk-A'-not0,
auto)
      also have ... = (D ·m 1m n) $$ ((i+1)−m, j+1)
      proof (cases A $$ (0,0) = 0)
        case True
        let ?S = (swaprows 0 m A)
        have S: ?S ∈ carrier-mat (m+n) n using A by auto
        have Si10: ?S $$ (i+1,0) = 0
```

```
      proof −
        have ?S $$ (i+1,0) = A $$ (i+1,0) using i1-m n0 i1-mn S by auto
        also have ... = (D ·m 1m n) $$ (i+1 − m,0)
            by (smt (verit) A-def A'' A i-ge-m append-rows-def carrier-matD
diff-add-inverse2 i1-mn
            index-mat-four-block less-imp-diff-less n0)
        also have ... = 0 using i-ge-m n0 i1-mn by auto
        finally show ?thesis .
      qed
      have M $$ (i+1, j+1) = (make-first-column-positive ?S) $$ (i+1,j+1)
        by (simp add: A'-def M-def True non-zero-positions-xs-m xs-empty)
       also have ... = (if ?S $$ (i+1,0) < 0 then − ?S $$ (i+1,j+1) else ?S
$$ (i+1,j+1))
        unfolding make-first-column-positive.simps using S i1-mn j1n by auto
      also have ... = ?S $$ (i+1,j+1) using Si10 by auto
      also have ... = A $$ (i+1,j+1) using i1-m n0 i1-mn S jn1 by auto
      also have ... = (D ·m 1m n) $$ (i+1 − m,j+1)
        by (smt (verit) A-def A'' A i-ge-m append-rows-def carrier-matD i1-mn
index-mat-four-block(1,3)
            index-one-mat(2) index-smult-mat(2) index-zero-mat(2) j1n
less-imp-diff-less add-diff-cancel-right')
      finally show ?thesis .
    next
      case False
      have Ai10: A $$ (i+1,0) = 0
      proof −
        have A $$ (i+1,0) = (D ·m 1m n) $$ (i+1 − m,0)
            by (smt (verit) A-def A'' A i-ge-m append-rows-def carrier-matD
diff-add-inverse2 i1-mn
            index-mat-four-block less-imp-diff-less n0)
        also have ... = 0 using i-ge-m n0 i1-mn by auto
        finally show ?thesis .
      qed
      have M $$ (i+1, j+1) = (make-first-column-positive A) $$ (i+1,j+1)
        by (simp add: A'-def M-def False True non-zero-positions-xs-m)
       also have ... = (if A $$ (i+1,0) < 0 then − A $$ (i+1,j+1) else A $$
(i+1,j+1))
        unfolding make-first-column-positive.simps using A i1-mn j1n by auto
      also have ... = A $$ (i+1,j+1) using Ai10 by auto
      also have ... = (D ·m 1m n) $$ (i+1 − m,j+1)
        by (smt (verit) A-def A'' A i-ge-m append-rows-def carrier-matD i1-mn
index-mat-four-block(1,3)
            index-one-mat(2) index-smult-mat(2) index-zero-mat(2) j1n
less-imp-diff-less add-diff-cancel-right')
      finally show ?thesis .
    qed
    also have ... = D ∗ (1m n) $$ ((i+1)−m, j+1)
      by (rule index-smult-mat, insert i jn1 A'-DR False dr, auto)
    also have ... = D ∗(1m (n − 1)) $$ (i−m,j) using dc dr i j A'-DR i-ge-m
```

by (*smt* (*verit*) *Nat.add-diff-assoc2 carrier-matD*(*1*) *index-one-mat*(*1*) *jn1 less-diff-conv*
linorder-not-less add-diff-cancel-right' add-diff-cancel-right' add-diff-cancel-left')
also have ... = (*D* $\cdot_m$ 1_m (*n* − *1*)) $$ (*i*−*m*,*j*)
by (*rule index-smult-mat*[*symmetric*], *insert i jn1 A'-DR False dr*, *auto*)
finally show *?thesis* **using** *1 2* **by** *auto*
next
case *False*
have *?reduce-below* $$ (*i*+*1*, *j*+*1*) = *M* $$ (*i*+*1*, *j*+*1*)
proof (*unfold non-zero-positions-xs-m M-def*,
rule reduce-below-abs-preserves-case-m[*OF M' m0 - M-M'D mk-A'-not0 m-le-n - d-xs all-less-m - - - D0*])
show *j* + *1* < *n* **using** *jn1* **by** *auto*
show *i* + *1* ∉ *set xs* **using** *all-less-m i-ge-m non-zero-positions-xs-m* **by** *auto*
show *i* + *1* ≠ *0* **by** *auto*
show *i* + *1* < *m* + *n* **using** *i-ge-m i dr A'-DR* **by** *auto*
show *i* + *1* ≠ *m* **using** *i-ge-m* **by** *auto*
qed (*insert False*)
also have ... = (*?M'* @$_r$ *D* $\cdot_m$ 1_m *n*) $$ (*i*+*1*, *j*+*1*) **unfolding** *M-def* **using** *False M-M'D* **by** *argo*
also have ... = (*D* $\cdot_m$ 1_m *n*) $$ ((*i*+*1*)−*m*, *j*+*1*)
proof −
have *f1*: *1* + *j* < *n*
by (*metis Groups.add-ac*(*2*) *jn1 less-diff-conv*)
have *f2*: ∀ *n*. ¬ *n* + *i* < *m*
by (*meson i-ge-m linorder-not-less nat-SN.compat not-add-less2*)
have *i* < *m* + (*n* − *1*)
by (*metis* (*no-types*) *A'-DR carrier-matD*(*1*) *dr i*)
then have *1* + *i* < *m* + *n*
using *f1* **by** *linarith*
then show *?thesis*
using *f2 f1* **by** (*metis* (*no-types*) *Groups.add-ac*(*2*) *M' append-rows-def carrier-matD*(*1*)
dim-col-mat(*1*) *index-mat-four-block*(*1*) *index-one-mat*(*2*) *index-smult-mat*(*2*)
index-zero-mat(*2*,*3*) *mat-of-rows-def nat-arith.rule0*)
qed
also have ... = *D* ∗ (1_m *n*) $$ ((*i*+*1*)−*m*, *j*+*1*)
by (*rule index-smult-mat*, *insert i jn1 A'-DR False dr*, *auto*)
also have ... = *D* ∗(1_m (*n* − *1*)) $$ (*i*−*m*,*j*) **using** *dc dr i j A'-DR i-ge-m*
by (*smt* (*verit*) *Nat.add-diff-assoc2 carrier-matD*(*1*) *index-one-mat*(*1*) *jn1 less-diff-conv*
linorder-not-less add-diff-cancel-right' add-diff-cancel-left')
also have ... = (*D* $\cdot_m$ 1_m (*n* − *1*)) $$ (*i*−*m*,*j*)
by (*rule index-smult-mat*[*symmetric*], *insert i jn1 A'-DR False dr*, *auto*)
finally have *3*: *?reduce-below* $$ (*i*+*1*,*j*+*1*) = (*D* $\cdot_m$ 1_m (*n* − *1*)) $$ (*i*−*m*,*j*)
.
show *?thesis* **using** *1 2 3* **by** *presburger*

```
    qed
  qed
qed
  let ?A'-DR-n = mat-of-rows (n − 1) (map (Matrix.row A'-DR) [0..<n − 1])
  have hyp: ∃ P. P∈carrier-mat (m + (n−1)) (m + (n−1)) ∧ invertible-mat P ∧
sub-PreHNF = P ∗ A'-DR
  ∧ echelon-form-JNF sub-PreHNF
  proof (cases 2 ≤ n − 1)
    case True
    show ?thesis
     by (unfold sub-PreHNF-def, rule 1.hyps[OF - - - non-zero-positions-def A'-def
- - - - -])
        (insert A n D0 m-le-n True A'DR-A'DR-m-D A A'-split abs-flag, auto)
  next
    case False
     have ∃ P. P∈carrier-mat (m + (n−1)) (m + (n−1)) ∧ invertible-mat P ∧
sub-PreHNF = P ∗ A'-DR
      by (unfold sub-PreHNF-def, rule FindPreHNF-invertible-mat-mx2
          [OF A'DR-A'DR-m-D A'-DR-m - - D0 -])
          (insert False m-le-n n0 m0 1(4), auto)
    moreover have echelon-form-JNF sub-PreHNF unfolding sub-PreHNF-def
     by (rule FindPreHNF-echelon-form-mx1[OF A'DR-A'DR-m-D A'-DR-m - D0
-],
         insert False n0 m-le-n, auto)
    ultimately show ?thesis by simp
  qed
  from this obtain P where P: P ∈ carrier-mat (m + (n − 1)) (m + (n − 1))
    and inv-P: invertible-mat P and sub-PreHNF-P-A'-DR: sub-PreHNF = P ∗
A'-DR by blast
 define P' where P' = (four-block-mat (1_m 1) (0_m 1 (m+(n−1))) (0_m (m+(n−1))
1) P)
  have P': P' ∈ carrier-mat (m+n) (m+n)
  proof −
    have P' ∈ carrier-mat (1 + (m+(n−1))) (1 + (m+(n−1)))
      unfolding P'-def by (rule four-block-carrier-mat[OF - P], simp)
    thus ?thesis using n by auto
  qed
  have inv-P': invertible-mat P'
   unfolding P'-def by (rule invertible-mat-four-block-mat-lower-right[OF P inv-P])
  have dr-A2: dim-row A ≥ 2 using A m0 n by auto
  have dc-A2: dim-col A ≥ 2 using n A by blast
  have ∗: (dim-col A = 0) = False using dc-A2 by auto
  have FindPreHNF-as-fbm: FindPreHNF abs-flag D A = four-block-mat A'-UL
A'-UR A'-DL sub-PreHNF
    unfolding FindPreHNF.simps[of abs-flag D A] using A'-split mn n A dr-A2
dc-A2 abs-flag
    unfolding Let-def sub-PreHNF-def M-def A'-def non-zero-positions-def ∗
    by (smt (verit) linorder-not-less split-conv)
  also have ... = P' ∗ (reduce-below-abs 0 non-zero-positions D M)
```

```
proof −
  have P' ∗ (reduce-below-abs 0 non-zero-positions D M)
    = four-block-mat (1_m 1) (0_m 1 (m + (n − 1))) (0_m (m + (n − 1)) 1) P
    ∗ four-block-mat A'-UL A'-UR A'-DL A'-DR
    unfolding P'-def fbm-R[unfolded M-def[symmetric], symmetric] ..
  also have ... = four-block-mat
        ((1_m 1) ∗ A'-UL + (0_m 1 (m + (n − 1)) ∗ A'-DL))
        ((1_m 1) ∗ A'-UR + (0_m 1 (m + (n − 1))) ∗ A'-DR)
        ((0_m (m + (n − 1)) 1) ∗ A'-UL + P ∗ A'-DL)
        ((0_m (m + (n − 1)) 1) ∗ A'-UR + P ∗ A'-DR)
    by (rule mult-four-block-mat[OF - - - P A'-UL A'-UR A'-DL A'-DR], auto)
  also have ... = four-block-mat A'-UL A'-UR (P ∗ A'-DL) (P ∗ A'-DR)
    by (rule cong-four-block-mat, insert A'-UL A'-UR A'-DL A'-DR P, auto)
 also have ... = four-block-mat A'-UL A'-UR (0_m (m + (n − 1)) 1) sub-PreHNF
    unfolding A'-DL0 sub-PreHNF-P-A'-DR using P by simp
  also have ... = four-block-mat A'-UL A'-UR A'-DL sub-PreHNF
    unfolding A'-DL0 by simp
  finally show ?thesis ..
 qed
 finally have Find-P'-reduceM: FindPreHNF abs-flag D A =  P' ∗ (reduce-below-abs
0 non-zero-positions D M) .
  have ∃ Q. invertible-mat Q ∧ Q ∈ carrier-mat (m + n) (m + n)
   ∧ reduce-below-abs 0 (xs @ [m]) D M = Q ∗ M
  proof (cases xs = [])
   case True note xs-empty = True
    have rw: reduce-below-abs 0 (xs @ [m]) D M = reduce-abs 0 m D M using
True by auto
   obtain p q u v d where pquvd: (p, q, u, v, d) = euclid-ext2 (M $$ (0, 0)) (M
$$ (m, 0))
     by (simp add: euclid-ext2-def)
   have ∃ P. invertible-mat P ∧ P ∈ carrier-mat (m + n) (m + n) ∧ reduce-abs
0 m D M = P ∗ M
   proof (rule reduce-abs-invertible-mat-case-m[OF - - m0 - - - - m-le-n n0 pquvd])
       show M $$ (0, 0) ≠ 0
         using M-def mk-A'-not0 by blast
       define M' where  M' = mat-of-rows n (map (Matrix.row M) [0..<m])
      define M'' where M'' = mat-of-rows n (map (Matrix.row M) [m..<m+n])
       define A2 where A2 = Matrix.mat (dim-row M) (dim-col M)
       (λ(i, k). if i = 0 then p ∗ M $$ (0, k) + q ∗ M $$ (m, k)
              else if i = m then u ∗ M $$ (0, k) + v ∗ M $$ (m, k)
              else M $$ (i, k))
       show M-M'-M'': M = M' @_r M'' unfolding M'-def M''-def
         by (metis M append-rows-split carrier-matD le-add1)
       show M': M' ∈ carrier-mat m n unfolding M'-def by fastforce
       show M'': M'' ∈ carrier-mat n n unfolding M''-def by fastforce
       show 0 ≠ m using m0 by simp
       show A2 = Matrix.mat (dim-row M) (dim-col M)
       (λ(i, k). if i = 0 then p ∗ M $$ (0, k) + q ∗ M $$ (m, k)
              else if i = m then u ∗ M $$ (0, k) + v ∗ M $$ (m, k)
```

```
            else M $$ (i, k))
        (is - = ?rhs) using A A2-def by auto
      define xs' where xs' =  filter (λi. abs (A2 $$ (0,i)) > D) [0..<n]
      define ys' where  ys' = filter (λi. abs (A2 $$ (m,i)) > D) [0..<n]
      show xs' = filter (λi. abs (A2 $$ (0,i)) > D) [0..<n] unfolding xs'-def
by auto
      show ys' = filter (λi. abs (A2 $$ (m,i)) > D) [0..<n] unfolding ys'-def
by auto
      have M''D: (M'' $$ (j, j) = D) ∧ (∀j'∈{0..<n}−{j}. M'' $$ (j, j') = 0)
        if jn: j<n and j0: j>0 for j
      proof −
        have Ajm0: A $$ (j+m,0) = 0
        proof −
          have A $$ (j+m,0) = (D ·m 1m n) $$ (j+m−m,0)
            by (smt (verit) 1(2) 1(3) M M' M'' M-M'-M'' add.commute
append-rows-def carrier-matD
              diff-add-inverse2 index-mat-four-block index-one-mat(2) in-
dex-smult-mat(2)
              le-add2 less-diff-conv2 n0 not-add-less2 that(1))
          also have ... = 0 using jn j0 by auto
          finally show ?thesis .
        qed
        have M'' $$ (j, i) = (D ·m 1m n) $$ (j,i) if i-n: i<n for i
        proof (cases A$$(0,0) = 0)
          case True
           have M'' $$ (j, i) = make-first-column-positive (swaprows 0 m A) $$
(j+m,i)
            using M' M'' M-M'-M'' unfolding M-def A-def
            by (metis (no-types, lifting) 1(2) 1(5) A'-def True append.simps(1)
append-rows-nth2 i-n jn
              non-zero-positions-xs-m nat-SN.compat nth-Cons-0 xs-empty)
          also have ... = A $$ (j+m,i) using A jn j0 i-n Ajm0 by auto
          also have ... = (D ·m 1m n) $$ (j,i)
          by (smt (verit) A Groups.add-ac(2) add-mono-thms-linordered-field(1)
append-rows-def A-def A'' i-n
              carrier-matD index-mat-four-block(1,2) add-diff-cancel-right'
not-add-less2 jn trans-less-add1)
          finally show ?thesis .
        next
          case False
           have A' = A unfolding A'-def non-zero-positions-xs-m using False
True by auto
          hence M'' $$ (j, i) = make-first-column-positive A $$ (j+m,i)
            using M' M'' M-M'-M'' unfolding M-def
              by (metis (no-types, opaque-lifting) 1(5) append-rows-nth2 jn
nat-SN.compat that)
          also have ... = A $$ (j+m,i) using A jn j0 i-n Ajm0 by auto
          also have ... = (D ·m 1m n) $$ (j,i)
          by (smt (verit) A Groups.add-ac(2) add-mono-thms-linordered-field(1)
```

append-rows-def A-def A'' i-n
carrier-matD index-mat-four-block(1,2) add-diff-cancel-right' not-add-less2 jn trans-less-add1)
finally show *?thesis* **.**
qed
thus *?thesis* **using** *jn j0* **by** *auto*
qed
have *Am0D*: *A$$(m,0)* = *D*
proof −
have *A$$(m,0)* = (*D* ·$_m$ *1*$_m$ *n*) $$ (*m*−*m*,*0*)
by (*smt* (*verit*) *1(2) 1(3) M M' M'' M-M'-M'' append-rows-def carrier-matD*
diff-less-mono2 diff-self-eq-0 index-mat-four-block index-one-mat(2) index-smult-mat(2) less-add-same-cancel1 n0 semiring-norm(137))
also have ... = *D* **using** *m0 n0* **by** *auto*
finally show *?thesis* **.**
qed
hence *S00D*: (*swaprows 0 m A*) $$ (*0*,*0*) = *D* **using** *n0 m0 A* **by** *auto*
have *Sm00*: (*swaprows 0 m A*) $$ (*m*,*0*) = *A$$(0,0)* **using** *n0 m0 A* **by** *auto*
have *M00D*: *M* $$ (*0*, *0*) = *D* **if** *A00*: *A$$(0,0)* = *0*
proof −
have *M* $$ (*0*,*0*) = (*make-first-column-positive* (*swaprows 0 m A*)) $$ (*0*,*0*)
unfolding *M-def A'-def* **using** *A00*
by (*simp add*: *True non-zero-positions-xs-m*)
also have ... = (*if* (*swaprows 0 m A*) $$ (*0*,*0*) < *0 then* − (*swaprows 0 m A*) $$(*0*,*0*)
else (*swaprows 0 m A*) $$(*0*,*0*))
unfolding *make-first-column-positive.simps* **using** *m0 n0 A* **by** *auto*
also have ... = (*swaprows 0 m A*) $$(*0*,*0*) **using** *S00D D0* **by** *auto*
also have ... = *D* **using** *S00D* **by** *auto*
finally show *?thesis* **.**
qed
have *Mm00*: *M* $$ (*m*, *0*) = *0* **if** *A00*: *A$$(0,0)* = *0*
proof −
have *M* $$ (*m*,*0*) = (*make-first-column-positive* (*swaprows 0 m A*)) $$ (*m*,*0*)
unfolding *M-def A'-def* **using** *A00*
by (*simp add*: *True non-zero-positions-xs-m*)
also have ... = (*if* (*swaprows 0 m A*) $$ (*m*,*0*) < *0 then* − (*swaprows 0 m A*) $$(*m*,*0*)
else (*swaprows 0 m A*) $$(*m*,*0*))
unfolding *make-first-column-positive.simps* **using** *m0 n0 A* **by** *auto*
also have ... = (*swaprows 0 m A*) $$(*m*,*0*) **using** *Sm00 A00 D0* **by** *auto*
also have ... = *0* **using** *Sm00 A00* **by** *auto*
finally show *?thesis* **.**
qed
have *M000*: *M* $$ (*0*, *0*) = *abs* (*A$$(0,0)*) **if** *A00*: *A$$(0,0)* ≠ *0*

proof −
have *M* $$ (*0*,*0*) = (*make-first-column-positive A*) $$ (*0*,*0*)
unfolding *M-def A′-def* **using** *A00*
by (*simp add*: *True non-zero-positions-xs-m*)
also have ... = (*if A* $$ (*0*,*0*) < *0 then* − *A* $$(*0*,*0*)
else A $$(*0*,*0*))
unfolding *make-first-column-positive.simps* **using** *m0 n0 A* **by** *auto*
also have ... = *abs* (*A*$$(*0*,*0*)) **using** *Sm00 A00* **by** *auto*
finally show *?thesis* **.**
qed
have *Mm0D*: *M* $$ (*m*, *0*) = *D* **if** *A00*: *A* $$ (*0*,*0*) ≠ *0*
proof −
have *M* $$ (*m*,*0*) = (*make-first-column-positive A*) $$ (*m*,*0*)
unfolding *M-def A′-def* **using** *A00*
by (*simp add*: *True non-zero-positions-xs-m*)
also have ... = (*if A* $$ (*m*,*0*) < *0 then* − *A* $$(*m*,*0*)
else A $$(*m*,*0*))
unfolding *make-first-column-positive.simps* **using** *m0 n0 A* **by** *auto*
also have ... = *A* $$(*m*,*0*) **using** *S00D D0 Am0D* **by** *auto*
also have ... = *D* **using** *Am0D D0* **by** *auto*
finally show *?thesis* **.**
qed
have *0* ∉ *set xs′*
proof −
have *A2* $$ (*0*,*0*) = *p* ∗ *M* $$ (*0*, *0*) + *q* ∗ *M* $$ (*m*, *0*)
using *A A2-def n0 M* **by** *auto*
also have ... = *gcd* (*M* $$ (*0*, *0*)) (*M* $$ (*m*, *0*))
by (*metis euclid-ext2-works*(*1*,*2*) *pquvd*)
also have *abs* ... ≤ *D* **using** *M00D Mm00 M000 Mm0D* **using** *gcd-0-int D0* **by** *fastforce*
finally have *abs* (*A2* $$ (*0*,*0*)) ≤ *D* **.**
thus *?thesis* **unfolding** *xs′-def* **using** *D0* **by** *auto*
qed
thus ∀ *j*∈*set xs′*. *j*<*n* ∧ (*M′′* $$ (*j*, *j*) = *D*) ∧ (∀ *j′*∈{*0*..<*n*}−{*j*}. *M′′* $$ (*j*, *j′*) = *0*)
using *M′′D xs′-def* **by** *auto*
have *0* ∉ *set ys′*
proof −
have *A2* $$ (*m*,*0*) = *u* ∗ *M* $$ (*0*, *0*) + *v* ∗ *M* $$ (*m*, *0*)
using *A A2-def n0 m0 M* **by** *auto*
also have ... = − *M* $$ (*m*, *0*) *div gcd* (*M* $$ (*0*, *0*)) (*M* $$ (*m*, *0*)) ∗ *M* $$ (*0*, *0*)
+ *M* $$ (*0*, *0*) *div gcd* (*M* $$ (*0*, *0*)) (*M* $$ (*m*, *0*)) ∗ *M* $$ (*m*, *0*)
by (*simp add*: *euclid-ext2-works*[*OF pquvd*[*symmetric*]])
also have ... = *0* **using** *M00D Mm00 M000 Mm0D*
by (*smt* (*verit*) *dvd-div-mult-self euclid-ext2-works*(*3*) *euclid-ext2-works*(*5*) *more-arith-simps*(*11*) *mult.commute mult-minus-left pquvd semiring-gcd-class.gcd-dvd1*)
finally have *A2* $$ (*m*,*0*) = *0* **.**

```
      thus ?thesis unfolding ys'-def using D0 by auto
    qed
    thus ∀j∈set ys'. j<n ∧ (M'' $$ (j, j) = D) ∧ (∀j'∈{0..<n}−{j}. M'' $$
(j, j') = 0)
      using M''D ys'-def by auto
  qed (insert D0)
  then show ?thesis using rw by auto
next
  case False
  show ?thesis
    by (unfold M-def, rule reduce-below-abs-invertible-mat-case-m[OF M' m0 n0
M-M'D[OF False]
      mk-A'-not0 m-le-n d-xs all-less-m D0])
qed

from this obtain Q where inv-Q: invertible-mat Q and Q: Q ∈ carrier-mat (m
+ n) (m + n)
  and reduce-QM: reduce-below-abs 0 (xs @ [m]) D M = Q * M by blast
have ∃R. invertible-mat R
  ∧ R ∈ carrier-mat (dim-row A') (dim-row A') ∧ M = R * A'
  by (unfold M-def, rule make-first-column-positive-invertible)
from this obtain R where inv-R: invertible-mat R
  and R: R ∈ carrier-mat (dim-row A') (dim-row A') and M-RA': M = R * A'
by blast
have ∃P. P ∈ carrier-mat (m + n) (m + n) ∧ invertible-mat P ∧ A' = P * A
  by (rule A'-swaprows-invertible-mat[OF A A'-def non-zero-positions-def],
    insert non-zero-positions-xs-m n m0, auto)
from this obtain S where inv-S: invertible-mat S
 and S: S ∈ carrier-mat (dim-row A) (dim-row A) and A'-SA: A' = S * A
  using A by auto
have (P'*Q*R*S) ∈ carrier-mat (m+n) (m+n) using P' Q R S A' A by auto
moreover have FindPreHNF abs-flag D A = (P'*Q*R*S) * A using Find-P'-reduceM
reduce-QM
  unfolding M-RA' A'-SA M-def
  by (smt (verit) A' A'-SA P' Q R S assoc-mult-mat carrier-matD carrier-mat-triv
index-mult-mat(2,3)
    non-zero-positions-xs-m)
 moreover have invertible-mat (P'*Q*R*S) using inv-P' inv-Q inv-R inv-S
using P' Q R S A' A
  by (metis carrier-matD carrier-mat-triv index-mult-mat(2,3) invertible-mult-JNF)
ultimately have exists-inv: ∃P. P ∈ carrier-mat (m + n) (m + n) ∧ invert-
ible-mat P
  ∧ FindPreHNF abs-flag D A = P * A by blast
moreover have echelon-form-JNF (FindPreHNF abs-flag D A)
proof (rule echelon-form-four-block-mat[OF A'-UL A'-UR sub-PreHNF' ])
  show FindPreHNF abs-flag D A = four-block-mat A'-UL A'-UR (0_m (m + n
− 1) 1) sub-PreHNF
    using A'-DL0 FindPreHNF-as-fbm sub-PreHNF sub-PreHNF' by auto
  have A'-UL $$ (0, 0) = ?R $$ (0,0)
```

by (*metis* (*mono-tags*, *lifting*) *A A′-DR A′-UL Find-P′-reduceM M-def*
‹*FindPreHNF abs-flag D A = P′ * Q * R * S * A*› *add-Suc-right add-sign-intros*(*2*) *carrier-matD fbm-R*
index-mat-four-block(*1*,*3*) *index-mult-mat*(*3*) *m0 n0 plus-1-eq-Suc*
zero-less-one-class.zero-less-one)
also have ... ≠ *0*
proof (*cases xs*=[])
case *True*
have *?R* $$ (*0*,*0*) = *reduce-abs 0 m D M* $$ (*0*,*0*)
unfolding *non-zero-positions-xs-m True M-def* **by** *simp*
also have ... ≠ *0*
by (*metis D-not0 M M-def add-pos-pos less-add-same-cancel1 m0 mk-A′-not0 n0 reduce-not0*)
finally show *?thesis* **.**
next
case *False*
show *?thesis*
by (*unfold non-zero-positions-xs-m*,
rule reduce-below-abs-not0-case-m[*OF M′ m0 n0 M-M′D*[*OF False*] *mk-A′-not0 m-le-n all-less-m D-not0*])
qed
finally show *A′-UL* $$ (*0*, *0*) ≠ *0* **.**
qed (*insert mn n hyp*, *auto*)
ultimately show *?thesis* **by** *blast*
next
case *False*
hence *A′-split*: (*A′-UL*, *A′-UR*, *A′-DL*, *A′-DR*)
= *split-block* (*reduce-below 0 non-zero-positions D* (*make-first-column-positive A′*)) *1 1* **using** *A′-split* **by** *auto*
let *?R* = *reduce-below 0 non-zero-positions D* (*make-first-column-positive A′*)
have *fbm-R*: *four-block-mat A′-UL A′-UR A′-DL A′-DR*
= *reduce-below 0 non-zero-positions D* (*make-first-column-positive A′*)
by (*rule split-block*(*5*)[*symmetric*, *OF A′-split*[*symmetric*], *of m+n−1 n−1*], *insert A′ n*, *auto*)
have *A′-DL0*: *A′-DL* = (*0*$_m$ (*m* + (*n* − *1*)) *1*)
proof (*rule eq-matI*)
show *dim-row A′-DL* = *dim-row* (*0*$_m$ (*m* + (*n* − *1*)) *1*)
and *dim-col A′-DL* = *dim-col* (*0*$_m$ (*m* + (*n* − *1*)) *1*) **using** *A′-DL* **by** *auto*

fix *i j* **assume** *i*: *i* < *dim-row* (*0*$_m$ (*m* + (*n* − *1*)) *1*) **and** *j*: *j* < *dim-col* (*0*$_m$ (*m* + (*n* − *1*)) *1*)
have *j0*: *j*=*0* **using** *j* **by** *auto*
have *0* = *?R* $$ (*i*+*1*,*j*)
proof (*unfold M-def non-zero-positions-xs-m j0*,
rule reduce-below-0-case-m-make-first-column-positive[*symmetric*,
OF A″ m0 n0 A-def m-le-n - d-xs all-less-m - - D0 -])
show *A′* = (*if A* $$ (*0*, *0*) ≠ *0 then A else let i* = (*xs* @ [*m*]) ! *0 in swaprows 0 i A*)
using *A′-def non-zero-positions-def non-zero-positions-xs-m* **by** *presburger*

show *xs* @ [*m*] = *filter* (λ*i*. *A* $$ (*i*, *0*) ≠ *0*) [*1*..<*dim-row A*]
using *A′-def non-zero-positions-def non-zero-positions-xs-m* **by** *presburger*
qed (*insert i n0*, *auto*)
also have ... = *four-block-mat A′-UL A′-UR A′-DL A′-DR* $$ (*i*+*1*,*j*) **unfolding** *fbm-R* **..**
also have ... = (*if i*+*1* < *dim-row A′-UL then if j* < *dim-col A′-UL*
then A′-UL $$ (*i*+*1*, *j*) *else A′-UR* $$ (*i*+*1*, *j* − *dim-col A′-UL*)
else if j < *dim-col A′-UL then A′-DL* $$ (*i*+*1* − *dim-row A′-UL*, *j*)
else A′-DR $$ (*i*+*1* − *dim-row A′-UL*, *j* − *dim-col A′-UL*))
by (*rule index-mat-four-block*, *insert A′-UL A′-DR i j*, *auto*)
also have ... = *A′-DL* $$ (*i*, *j*) **using** *A′-UL A′-UR i j* **by** *auto*
finally show *A′-DL* $$ (*i*, *j*) = *0*$_m$ (*m* + (*n* − *1*)) *1* $$ (*i*, *j*) **using** *i j* **by** *auto*
qed

let *?A′-DR-m* = *mat-of-rows* (*n*−*1*) [*Matrix.row A′-DR i*. *i* ← [*0*..<*m*]]
have *A′-DR-m*: *?A′-DR-m* ∈ *carrier-mat m* (*n*−*1*) **by** *auto*
have *A′DR-A′DR-m-D*: *A′-DR* = *?A′-DR-m* @$_r$ *D* ·$_m$ *1*$_m$ (*n* − *1*)
proof (*rule eq-matI*)
show *dr*: *dim-row A′-DR* = *dim-row* (*?A′-DR-m* @$_r$ *D* ·$_m$ *1*$_m$ (*n* − *1*))
by (*metis A′-DR A′-DR-m append-rows-def carrier-matD*(*1*) *index-mat-four-block*(*2*)
index-one-mat(*2*) *index-smult-mat*(*2*) *index-zero-mat*(*2*))
show *dc*: *dim-col A′-DR* = *dim-col* (*?A′-DR-m* @$_r$ *D* ·$_m$ *1*$_m$ (*n* − *1*))
by (*metis A′-DR A′-DR-m add.comm-neutral append-rows-def*
carrier-matD(*2*) *index-mat-four-block*(*3*) *index-zero-mat*(*3*))
fix *i j* **assume** *i*: *i* < *dim-row*(*?A′-DR-m* @$_r$ *D* ·$_m$ *1*$_m$ (*n* − *1*))
and *j*: *j*<*dim-col* (*?A′-DR-m* @$_r$ *D* ·$_m$ *1*$_m$ (*n* − *1*))
have *jn1*: *j*<*n*−*1* **using** *dc j A′-DR* **by** *auto*
show *A′-DR* $$ (*i*,*j*) = (*?A′-DR-m* @$_r$ *D* ·$_m$ *1*$_m$ (*n* − *1*)) $$ (*i*,*j*)
proof (*cases i*<*m*)
case *True*
have *A′-DR* $$ (*i*,*j*) = *?A′-DR-m* $$ (*i*,*j*)
by (*metis A′-DR A′-DR-m True dc carrier-matD*(*1*) *carrier-matD*(*2*) *j le-add1*
map-first-rows-index mat-of-rows-carrier(*2*) *mat-of-rows-index*)
also have ... = (*?A′-DR-m* @$_r$ *D* ·$_m$ *1*$_m$ (*n* − *1*)) $$ (*i*,*j*)
by (*metis* (*mono-tags*, *lifting*) *A′-DR A′-DR-m True append-rows-def*
carrier-matD dc i index-mat-four-block j)
finally show *?thesis* **.**
next
case *False* **note** *i-ge-m* = *False*
let *?reduce-below* = *reduce-below 0 non-zero-positions D* (*make-first-column-positive A′*)
have *1*: (*?A′-DR-m* @$_r$ *D* ·$_m$ *1*$_m$ (*n* − *1*)) $$ (*i*,*j*) = (*D* ·$_m$ *1*$_m$ (*n* − *1*)) $$ (*i*−*m*,*j*)
by (*smt* (*verit*) *A′-DR A′-DR-m False append-rows-nth carrier-matD carrier-mat-triv dc dr i*
index-one-mat(*2*) *index-one-mat*(*3*) *index-smult-mat*(*2*,*3*) *j*)

have *?reduce-below = four-block-mat A′-UL A′-UR A′-DL A′-DR* **using** *fbm-R*
..
also have ... $$ (*i+1,j+1*) = (*if i+1 < dim-row A′-UL then if j+1 < dim-col A′-UL*
then A′-UL $$ (*i+1, j+1*) *else A′-UR* $$ (*i+1, j+1 − dim-col A′-UL*)
else if j+1 < dim-col A′-UL then A′-DL $$ (*i+1 − dim-row A′-UL, j+1*)
else A′-DR $$ (*i+1 − dim-row A′-UL, j+1 − dim-col A′-UL*))
by (*rule index-mat-four-block, insert i j A′-UL A′-DR dr dc, auto*)
also have ... = *A′-DR* $$ (*i,j*) **using** *A′-UL* **by** *auto*
finally have *2*: *?reduce-below* $$ (*i+1,j+1*) = *A′-DR* $$ (*i,j*) **.**
show *?thesis*
proof (*cases xs = []*)
case *True* **note** *xs-empty = True*
have *i1-m*: *i + 1 ≠ m*
using *False less-add-one* **by** *blast*
have *j1n*: *j+1<n*
using *jn1 less-diff-conv* **by** *blast*
have *i1-mn*: *i+1<m + n*
using *i i-ge-m*
by (*metis A′-DR carrier-matD(1) dr less-diff-conv sub-PreHNF sub-PreHNF′*)
have *?reduce-below = reduce 0 m D M*
unfolding *non-zero-positions-xs-m xs-empty M-def* **by** *auto*
also have ... $$ (*i+1,j+1*) = *M* $$ (*i+1, j+1*)
by (*rule reduce-preserves*[*OF M j1n - i1-m - i1-mn*], *insert M-def mk-A′-not0, auto*)
also have ... = (*D ·m 1m n*) $$ ((*i+1*)−*m, j+1*)
proof (*cases A* $$ (*0,0*) = *0*)
case *True*
let *?S* = (*swaprows 0 m A*)
have *S*: *?S ∈ carrier-mat* (*m+n*) *n* **using** *A* **by** *auto*
have *Si10*: *?S* $$ (*i+1,0*) = *0*
proof −
have *?S* $$ (*i+1,0*) = *A* $$ (*i+1,0*) **using** *i1-m n0 i1-mn S* **by** *auto*
also have ... = (*D ·m 1m n*) $$ (*i+1 − m,0*)
by (*smt* (*verit*) *A-def A′′ A i-ge-m append-rows-def carrier-matD diff-add-inverse2 i1-mn*
index-mat-four-block less-imp-diff-less n0)
also have ... = *0* **using** *i-ge-m n0 i1-mn* **by** *auto*
finally show *?thesis* **.**
qed
have *M* $$ (*i+1, j+1*) = (*make-first-column-positive ?S*) $$ (*i+1,j+1*)
by (*simp add*: *A′-def M-def True non-zero-positions-xs-m xs-empty*)
also have ... = (*if ?S* $$ (*i+1,0*) < *0 then − ?S* $$ (*i+1,j+1*) *else ?S* $$ (*i+1,j+1*))
unfolding *make-first-column-positive.simps* **using** *S i1-mn j1n* **by** *auto*
also have ... = *?S* $$ (*i+1,j+1*) **using** *Si10* **by** *auto*
also have ... = *A* $$ (*i+1,j+1*) **using** *i1-m n0 i1-mn S jn1* **by** *auto*
also have ... = (*D ·m 1m n*) $$ (*i+1 − m,j+1*)

by (*smt* (*verit*) *A-def A'' A i-ge-m append-rows-def carrier-matD i1-mn index-mat-four-block*(*1*,*3*)
index-one-mat(*2*) *index-smult-mat*(*2*) *index-zero-mat*(*2*) *j1n less-imp-diff-less add-diff-cancel-right'*)
finally show *?thesis* **.**
next
case *False*
have *Ai10*: *A* $$ (*i+1*,*0*) = *0*
proof −
have *A* $$ (*i+1*,*0*) = (*D* ·$_m$ *1*$_m$ *n*) $$ (*i+1* − *m*,*0*)
by (*smt* (*verit*) *A-def A'' A i-ge-m append-rows-def carrier-matD diff-add-inverse2 i1-mn*
index-mat-four-block less-imp-diff-less n0)
also have ... = *0* **using** *i-ge-m n0 i1-mn* **by** *auto*
finally show *?thesis* **.**
qed
have *M* $$ (*i+1*, *j+1*) = (*make-first-column-positive A*) $$ (*i+1*,*j+1*)
by (*simp add*: *A'-def M-def False True non-zero-positions-xs-m*)
also have ... = (*if A* $$ (*i+1*,*0*) < *0 then* − *A* $$ (*i+1*,*j+1*) *else A* $$ (*i+1*,*j+1*))
unfolding *make-first-column-positive.simps* **using** *A i1-mn j1n* **by** *auto*
also have ... = *A* $$ (*i+1*,*j+1*) **using** *Ai10* **by** *auto*
also have ... = (*D* ·$_m$ *1*$_m$ *n*) $$ (*i+1* − *m*,*j+1*)
by (*smt* (*verit*) *A-def A'' A i-ge-m append-rows-def carrier-matD i1-mn index-mat-four-block*(*1*,*3*)
index-one-mat(*2*) *index-smult-mat*(*2*) *index-zero-mat*(*2*) *j1n less-imp-diff-less add-diff-cancel-right'*)
finally show *?thesis* **.**
qed
also have ... = *D* ∗ (*1*$_m$ *n*) $$ ((*i+1*)−*m*, *j+1*)
by (*rule index-smult-mat*, *insert i jn1 A'-DR False dr*, *auto*)
also have ... = *D* ∗(*1*$_m$ (*n* − *1*)) $$ (*i*−*m*,*j*) **using** *dc dr i j A'-DR i-ge-m*
by (*smt* (*verit*) *Nat.add-diff-assoc2 carrier-matD*(*1*) *index-one-mat*(*1*) *jn1 less-diff-conv*
linorder-not-less add-diff-cancel-right' add-diff-cancel-right' add-diff-cancel-left')
also have ... = (*D* ·$_m$ *1*$_m$ (*n* − *1*)) $$ (*i*−*m*,*j*)
by (*rule index-smult-mat*[*symmetric*], *insert i jn1 A'-DR False dr*, *auto*)
finally show *?thesis* **using** *1 2* **by** *auto*
next
case *False*
have *?reduce-below* $$ (*i+1*, *j+1*) = *M* $$ (*i+1*, *j+1*)
proof (*unfold non-zero-positions-xs-m M-def*,
rule reduce-below-preserves-case-m[*OF M' m0 - M-M'D mk-A'-not0 m-le-n - d-xs all-less-m - - - D0*])
show *j* + *1* < *n* **using** *jn1* **by** *auto*
show *i* + *1* ∉ *set xs* **using** *all-less-m i-ge-m non-zero-positions-xs-m* **by** *auto*
show *i* + *1* ≠ *0* **by** *auto*
show *i* + *1* < *m* + *n* **using** *i-ge-m i dr A'-DR* **by** *auto*

```
      show  i + 1 ≠ m using i-ge-m by auto
    qed (insert False)
   also have ... = (?M' @r D ·m 1m n) $$ (i+1, j+1) unfolding M-def using
False M-M'D by argo
    also have ... = (D ·m 1m n) $$ ((i+1)−m, j+1)
    proof −
      have f1: 1 + j < n
        by (metis Groups.add-ac(2) jn1 less-diff-conv)
      have f2: ∀ n. ¬ n + i < m
        by (meson i-ge-m linorder-not-less nat-SN.compat not-add-less2)
      have i < m + (n − 1)
        by (metis (no-types) A'-DR carrier-matD(1) dr i)
      then have 1 + i < m + n
        using f1 by linarith
      then show ?thesis
        using f2 f1 by (metis (no-types) Groups.add-ac(2) M' append-rows-def
carrier-matD(1)
        dim-col-mat(1) index-mat-four-block(1) index-one-mat(2) index-smult-mat(2)

          index-zero-mat(2,3) mat-of-rows-def nat-arith.rule0)
    qed
    also have ... = D * (1m n) $$ ((i+1)−m, j+1)
      by (rule index-smult-mat, insert i jn1 A'-DR False dr, auto)
    also have ... = D *(1m (n − 1)) $$ (i−m,j) using dc dr i j A'-DR i-ge-m
      by (smt (verit) Nat.add-diff-assoc2 carrier-matD(1) index-one-mat(1) jn1
less-diff-conv
        linorder-not-less add-diff-cancel-right' add-diff-cancel-left')
    also have ... = (D ·m 1m (n − 1)) $$ (i−m,j)
      by (rule index-smult-mat[symmetric], insert i jn1 A'-DR False dr, auto)
   finally have 3: ?reduce-below $$ (i+1,j+1) = (D ·m 1m (n − 1)) $$ (i−m,j)
.
    show ?thesis using 1 2 3 by presburger
  qed
 qed
qed
 let ?A'-DR-n = mat-of-rows (n − 1) (map (Matrix.row A'-DR) [0..<n − 1])
 have hyp: ∃ P. P∈carrier-mat (m + (n−1)) (m + (n−1)) ∧ invertible-mat P ∧
sub-PreHNF = P * A'-DR
 ∧ echelon-form-JNF sub-PreHNF
 proof (cases 2 ≤ n − 1)
  case True
  show ?thesis
   by (unfold sub-PreHNF-def, rule 1.hyps[OF - - - non-zero-positions-def A'-def
- - - - -])
      (insert A n D0 m-le-n True A'DR-A'DR-m-D A A'-split False, auto)
 next
  case False
   have ∃ P. P∈carrier-mat (m + (n−1)) (m + (n−1)) ∧ invertible-mat P ∧
sub-PreHNF = P * A'-DR
```

```
    by (unfold sub-PreHNF-def, rule FindPreHNF-invertible-mat-mx2
        [OF A'DR-A'DR-m-D A'-DR-m - - D0 -])
        (insert False m-le-n n0 m0 1(4), auto)
   moreover have echelon-form-JNF sub-PreHNF unfolding sub-PreHNF-def
    by (rule FindPreHNF-echelon-form-mx1[OF A'DR-A'DR-m-D A'-DR-m - D0
-],
        insert False n0 m-le-n, auto)
   ultimately show ?thesis by simp
  qed
  from this obtain P where P: P ∈ carrier-mat (m + (n − 1)) (m + (n − 1))
    and inv-P: invertible-mat P and sub-PreHNF-P-A'-DR: sub-PreHNF = P ∗
A'-DR by blast
  define P' where P' = (four-block-mat (1_m 1) (0_m 1 (m+(n−1))) (0_m (m+(n−1))
1) P)
  have P': P' ∈ carrier-mat (m+n) (m+n)
  proof −
   have P' ∈ carrier-mat (1 + (m+(n−1))) (1 + (m+(n−1)))
     unfolding P'-def by (rule four-block-carrier-mat[OF - P], simp)
   thus ?thesis using n by auto
  qed
  have inv-P': invertible-mat P'
   unfolding P'-def by (rule invertible-mat-four-block-mat-lower-right[OF P inv-P])
  have dr-A2: dim-row A ≥ 2 using A m0 n by auto
  have dc-A2: dim-col A ≥ 2 using n A by blast
  have ∗: (dim-col A = 0) = False using dc-A2 by auto
  have FindPreHNF-as-fbm: FindPreHNF abs-flag D A = four-block-mat A'-UL
A'-UR A'-DL sub-PreHNF
    unfolding FindPreHNF.simps[of abs-flag D A] using A'-split mn n A dr-A2
dc-A2 False
    unfolding Let-def sub-PreHNF-def M-def A'-def non-zero-positions-def ∗
    by (smt (verit) linorder-not-less split-conv)
  also have ... = P' ∗ (reduce-below 0 non-zero-positions D M)
  proof −
    have P' ∗ (reduce-below 0 non-zero-positions D M)
      = four-block-mat (1_m 1) (0_m 1 (m + (n − 1))) (0_m (m + (n − 1)) 1) P
      ∗ four-block-mat A'-UL A'-UR A'-DL A'-DR
      unfolding P'-def fbm-R[unfolded M-def[symmetric], symmetric] ..
    also have ... = four-block-mat
          ((1_m 1) ∗ A'-UL + (0_m 1 (m + (n − 1)) ∗ A'-DL))
          ((1_m 1) ∗ A'-UR + (0_m 1 (m + (n − 1))) ∗ A'-DR)
          ((0_m (m + (n − 1)) 1) ∗ A'-UL + P ∗ A'-DL)
          ((0_m (m + (n − 1)) 1) ∗ A'-UR + P ∗ A'-DR)
      by (rule mult-four-block-mat[OF - - - P A'-UL A'-UR A'-DL A'-DR], auto)
    also have ... = four-block-mat A'-UL A'-UR (P ∗ A'-DL) (P ∗ A'-DR)
      by (rule cong-four-block-mat, insert A'-UL A'-UR A'-DL A'-DR P, auto)
   also have ... = four-block-mat A'-UL A'-UR (0_m (m + (n − 1)) 1) sub-PreHNF
      unfolding A'-DL0 sub-PreHNF-P-A'-DR using P by simp
    also have ... = four-block-mat A'-UL A'-UR A'-DL sub-PreHNF
      unfolding A'-DL0 by simp
```

finally show *?thesis* **..**
qed
finally have *Find-P'-reduceM*: *FindPreHNF abs-flag D A = P' ∗ (reduce-below 0 non-zero-positions D M)* **.**
have ∃ *Q*. *invertible-mat Q* ∧ *Q* ∈ *carrier-mat* (*m* + *n*) (*m* + *n*)
∧ *reduce-below 0* (*xs* @ [*m*]) *D M* = *Q* ∗ *M*
proof (*cases xs* = [])
case *True* **note** *xs-empty* = *True*
have *rw*: *reduce-below 0* (*xs* @ [*m*]) *D M* = *reduce 0 m D M* **using** *True* **by** *auto*
obtain *p q u v d* **where** *pquvd*: (*p*, *q*, *u*, *v*, *d*) = *euclid-ext2* (*M* $$ (*0*, *0*)) (*M* $$ (*m*, *0*))
by (*simp add*: *euclid-ext2-def*)
have ∃ *P*. *invertible-mat P* ∧ *P* ∈ *carrier-mat* (*m* + *n*) (*m* + *n*) ∧ *reduce 0 m D M* = *P* ∗ *M*
proof (*rule reduce-invertible-mat-case-m*[*OF - - m0 - - - - m-le-n n0 pquvd*])
show *M* $$ (*0*, *0*) ≠ *0*
using *M-def mk-A'-not0* **by** *blast*
define *M'* **where** *M'* = *mat-of-rows n* (*map* (*Matrix.row M*) [*0*..<*m*])
define *M''* **where** *M''* = *mat-of-rows n* (*map* (*Matrix.row M*) [*m*..<*m*+*n*])
define *A2* **where** *A2* = *Matrix.mat* (*dim-row M*) (*dim-col M*)
(λ(*i*, *k*). *if i* = *0 then p* ∗ *M* $$ (*0*, *k*) + *q* ∗ *M* $$ (*m*, *k*)
else if i = *m then u* ∗ *M* $$ (*0*, *k*) + *v* ∗ *M* $$ (*m*, *k*)
else M $$ (*i*, *k*))
show *M-M'-M''*: *M* = *M'* @$_r$ *M''* **unfolding** *M'-def M''-def*
by (*metis M append-rows-split carrier-matD le-add1*)
show *M'*: *M'* ∈ *carrier-mat m n* **unfolding** *M'-def* **by** *fastforce*
show *M''*: *M''* ∈ *carrier-mat n n* **unfolding** *M''-def* **by** *fastforce*
show *0* ≠ *m* **using** *m0* **by** *simp*
show *A2* = *Matrix.mat* (*dim-row M*) (*dim-col M*)
(λ(*i*, *k*). *if i* = *0 then p* ∗ *M* $$ (*0*, *k*) + *q* ∗ *M* $$ (*m*, *k*)
else if i = *m then u* ∗ *M* $$ (*0*, *k*) + *v* ∗ *M* $$ (*m*, *k*)
else M $$ (*i*, *k*))
(**is** - = *?rhs*) **using** *A A2-def* **by** *auto*
define *xs'* **where** *xs'* = [*1*..<*n*]
define *ys'* **where** *ys'* = [*1*..<*n*]
show *xs'* = [*1*..<*n*] **unfolding** *xs'-def* **by** *auto*
show *ys'* = [*1*..<*n*] **unfolding** *ys'-def* **by** *auto*
have *M''D*: (*M''* $$ (*j*, *j*) = *D*) ∧ (∀ *j'*∈{*0*..<*n*}−{*j*}. *M''* $$ (*j*, *j'*) = *0*)
if *jn*: *j*<*n* **and** *j0*: *j*>*0* **for** *j*
proof −
have *Ajm0*: *A* $$ (*j*+*m*,*0*) = *0*
proof −
have *A* $$ (*j*+*m*,*0*) = (*D* ·$_m$ *1*$_m$ *n*) $$ (*j*+*m*−*m*,*0*)
by (*smt* (*verit*) *1*(*2*) *1*(*3*) *M M' M'' M-M'-M'' add.commute append-rows-def carrier-matD
diff-add-inverse2 index-mat-four-block index-one-mat(*2*) *index-smult-mat*(*2*)
le-add2 less-diff-conv2 n0 not-add-less2 that(*1*))

```
        also have ... = 0 using jn j0 by auto
        finally show ?thesis .
      qed
      have M'' $$ (j, i) = (D ·_m 1_m n) $$ (j,i) if i-n: i<n for i
      proof (cases A$$(0,0) = 0)
        case True
        have M'' $$ (j, i) = make-first-column-positive (swaprows 0 m A) $$
(j+m,i)
          using M' M'' M-M'-M''
            by (simp add: M-def A'-def True append-rows-nth3 i-n jn lo-
cal.non-zero-positions-xs-m xs-empty)
        also have ... = A $$ (j+m,i) using A jn j0 i-n Ajm0 by auto
        also have ... = (D ·_m 1_m n) $$ (j,i)
        by (smt (verit) A Groups.add-ac(2) add-mono-thms-linordered-field(1)
append-rows-def A-def A'' i-n
              carrier-matD index-mat-four-block(1,2) add-diff-cancel-right'
not-add-less2 jn trans-less-add1)
        finally show ?thesis .
      next
        case False
        have A' = A unfolding A'-def non-zero-positions-xs-m using False
True by auto
      hence M'' $$ (j, i) = make-first-column-positive A $$ (j+m,i)
          using M' M'' M-M'-M'' unfolding M-def
            by (metis (no-types, opaque-lifting) 1(5) append-rows-nth2 jn
nat-SN.compat that)
        also have ... = A $$ (j+m,i) using A jn j0 i-n Ajm0 by auto
        also have ... = (D ·_m 1_m n) $$ (j,i)
        by (smt (verit) A Groups.add-ac(2) add-mono-thms-linordered-field(1)
append-rows-def A-def A'' i-n
              carrier-matD index-mat-four-block(1,2) add-diff-cancel-right'
not-add-less2 jn trans-less-add1)
        finally show ?thesis .
      qed
      thus ?thesis using jn j0 by auto
    qed
    have Am0D: A$$(m,0) = D
    proof −
      have A$$(m,0) = (D ·_m 1_m n) $$ (m−m,0)
          by (smt (verit) 1(2) 1(3) M M' M'' M-M'-M'' append-rows-def
carrier-matD
          diff-less-mono2 diff-self-eq-0 index-mat-four-block index-one-mat(2)
          index-smult-mat(2) less-add-same-cancel1 n0 semiring-norm(137))
      also have ... = D using m0 n0 by auto
      finally show ?thesis .
    qed
    hence S00D: (swaprows 0 m A) $$ (0,0) = D using n0 m0 A by auto
    have Sm00: (swaprows 0 m A) $$ (m,0) = A$$(0,0) using n0 m0 A by
auto
```

```
        have M00D: M $$ (0, 0) = D if A00: A$$(0,0) = 0
        proof −
          have M $$ (0,0) = (make-first-column-positive (swaprows 0 m A)) $$
(0,0)
            unfolding M-def A'-def using A00
            by (simp add: True non-zero-positions-xs-m)
          also have ... = (if (swaprows 0 m A) $$ (0,0) < 0 then − (swaprows 0
m A) $$(0,0)
                          else (swaprows 0 m A) $$(0,0))
            unfolding make-first-column-positive.simps using m0 n0 A by auto
          also have ... = (swaprows 0 m A) $$(0,0) using S00D D0 by auto
          also have ... = D using S00D by auto
          finally show ?thesis .
        qed
        have Mm00: M $$ (m, 0) = 0 if A00: A$$(0,0) = 0
        proof −
          have M $$ (m,0) = (make-first-column-positive (swaprows 0 m A)) $$
(m,0)
            unfolding M-def A'-def using A00
            by (simp add: True non-zero-positions-xs-m)
          also have ... = (if (swaprows 0 m A) $$ (m,0) < 0 then − (swaprows 0
m A) $$(m,0)
                          else (swaprows 0 m A) $$(m,0))
            unfolding make-first-column-positive.simps using m0 n0 A by auto
          also have ... = (swaprows 0 m A) $$(m,0) using Sm00 A00 D0 by auto
          also have ... = 0 using Sm00 A00 by auto
          finally show ?thesis .
        qed
        have M000: M $$ (0, 0) = abs (A$$(0,0)) if A00: A$$(0,0) ≠ 0
        proof −
          have M $$ (0,0) = (make-first-column-positive A) $$ (0,0)
            unfolding M-def A'-def using A00
            by (simp add: True non-zero-positions-xs-m)
          also have ... = (if A $$ (0,0) < 0 then − A $$(0,0)
                          else A $$(0,0))
            unfolding make-first-column-positive.simps using m0 n0 A by auto
          also have ... = abs (A$$(0,0)) using Sm00 A00 by auto
          finally show ?thesis .
        qed
        have Mm0D: M $$ (m, 0) = D if A00: A $$ (0,0) ≠ 0
        proof −
          have M $$ (m,0) = (make-first-column-positive A) $$ (m,0)
            unfolding M-def A'-def using A00
            by (simp add: True non-zero-positions-xs-m)
          also have ... = (if A $$ (m,0) < 0 then − A $$(m,0)
                          else A $$(m,0))
            unfolding make-first-column-positive.simps using m0 n0 A by auto
          also have ... = A $$(m,0) using S00D D0 Am0D by auto
          also have ... = D using Am0D D0 by auto
```

```
      finally show ?thesis .
    qed
    have 0 ∉ set xs'
    proof −
      have A2 $$ (0,0) = p * M $$ (0, 0) + q * M $$ (m, 0)
        using A A2-def n0 M by auto
      also have ... = gcd (M $$ (0, 0)) (M $$ (m, 0))
        by (metis euclid-ext2-works(1,2) pquvd)
     also have abs ... ≤ D using M00D Mm00 M000 Mm0D using gcd-0-int
D0 by fastforce
      finally have abs (A2 $$ (0,0)) ≤ D .
      thus ?thesis unfolding xs'-def using D0 by auto
    qed
    thus ∀j∈set xs'. j<n ∧ (M'' $$ (j, j) = D) ∧ (∀j'∈{0..<n}−{j}. M'' $$
(j, j') = 0)
      using M''D xs'-def by auto
    have 0 ∉ set ys'
    proof −
      have A2 $$ (m,0) = u * M $$ (0, 0) + v * M $$ (m, 0)
        using A A2-def n0 m0 M by auto
      also have ... = − M $$ (m, 0) div gcd (M $$ (0, 0)) (M $$ (m, 0)) *
M $$ (0, 0)
          + M $$ (0, 0) div gcd (M $$ (0, 0)) (M $$ (m, 0)) * M $$ (m, 0)
        by (simp add: euclid-ext2-works[OF pquvd[symmetric]])
      also have ... = 0 using M00D Mm00 M000 Mm0D
    by (smt (verit) dvd-div-mult-self euclid-ext2-works(3) euclid-ext2-works(5)
          more-arith-simps(11) mult.commute mult-minus-left pquvd semir-
ing-gcd-class.gcd-dvd1)
      finally have A2 $$ (m,0) = 0 .
      thus ?thesis unfolding ys'-def using D0 by auto
    qed
    thus ∀j∈set ys'. j<n ∧ (M'' $$ (j, j) = D) ∧ (∀j'∈{0..<n}−{j}. M'' $$
(j, j') = 0)
      using M''D ys'-def by auto
    show M $$ (m, 0) ∈ {0,D} using Mm00 Mm0D by blast
     show  M $$ (m, 0) = 0 ⟶ M $$ (0, 0) = D  using Mm00 Mm0D
D-not0 M00D by blast
   qed (insert D0)
   then show ?thesis using rw by auto
 next
   case False
   show ?thesis
     by (unfold M-def, rule reduce-below-invertible-mat-case-m[OF M' m0 n0
M-M'D[OF False]
      mk-A'-not0 m-le-n d-xs all-less-m D0])
 qed

 from this obtain Q where inv-Q: invertible-mat Q and Q: Q ∈ carrier-mat (m
+ n) (m + n)
```

and *reduce-QM*: *reduce-below 0* (*xs* @ [*m*]) *D M* = *Q* ∗ *M* **by** *blast*
have ∃ *R*. *invertible-mat R*
∧ *R* ∈ *carrier-mat* (*dim-row A′*) (*dim-row A′*) ∧ *M* = *R* ∗ *A′*
by (*unfold M-def*, *rule make-first-column-positive-invertible*)
from *this* **obtain** *R* **where** *inv-R*: *invertible-mat R*
and *R*: *R* ∈ *carrier-mat* (*dim-row A′*) (*dim-row A′*) **and** *M-RA′*: *M* = *R* ∗ *A′*
by *blast*
have ∃ *P*. *P* ∈ *carrier-mat* (*m* + *n*) (*m* + *n*) ∧ *invertible-mat P* ∧ *A′* = *P* ∗ *A*
by (*rule A′-swaprows-invertible-mat*[*OF A A′-def non-zero-positions-def*],
insert non-zero-positions-xs-m n m0, *auto*)
from *this* **obtain** *S* **where** *inv-S*: *invertible-mat S*
and *S*: *S* ∈ *carrier-mat* (*dim-row A*) (*dim-row A*) **and** *A′-SA*: *A′* = *S* ∗ *A*
using *A* **by** *auto*
have (*P′*∗*Q*∗*R*∗*S*) ∈ *carrier-mat* (*m*+*n*) (*m*+*n*) **using** *P′ Q R S A′ A* **by** *auto*
moreover have *FindPreHNF abs-flag D A* = (*P′*∗*Q*∗*R*∗*S*) ∗ *A* **using** *Find-P′-reduceM reduce-QM*
unfolding *M-RA′ A′-SA M-def*
by (*smt* (*verit*) *A′ A′-SA P′ Q R S assoc-mult-mat carrier-matD carrier-mat-triv index-mult-mat*(*2*,*3*)
non-zero-positions-xs-m)
moreover have *invertible-mat* (*P′*∗*Q*∗*R*∗*S*) **using** *inv-P′ inv-Q inv-R inv-S* **using** *P′ Q R S A′ A*
by (*metis carrier-matD carrier-mat-triv index-mult-mat*(*2*,*3*) *invertible-mult-JNF*)
ultimately have *exists-inv*: ∃ *P*. *P* ∈ *carrier-mat* (*m* + *n*) (*m* + *n*) ∧ *invertible-mat P*
∧ *FindPreHNF abs-flag D A* = *P* ∗ *A* **by** *blast*
moreover have *echelon-form-JNF* (*FindPreHNF abs-flag D A*)
proof (*rule echelon-form-four-block-mat*[*OF A′-UL A′-UR sub-PreHNF′*])
show *FindPreHNF abs-flag D A* = *four-block-mat A′-UL A′-UR* (*0*$_m$ (*m* + *n* − *1*) *1*) *sub-PreHNF*
using *A′-DL0 FindPreHNF-as-fbm sub-PreHNF sub-PreHNF′* **by** *auto*
have *A′-UL* $$ (*0*, *0*) = *?R* $$ (*0*,*0*)
by (*metis* (*mono-tags*, *lifting*) *A A′-DR A′-UL Find-P′-reduceM M-def*
‹*FindPreHNF abs-flag D A* = *P′* ∗ *Q* ∗ *R* ∗ *S* ∗ *A*› *add-Suc-right add-sign-intros*(*2*) *carrier-matD fbm-R*
index-mat-four-block(*1*,*3*) *index-mult-mat*(*3*) *m0 n0 plus-1-eq-Suc*
zero-less-one-class.zero-less-one)
also have ... ≠ *0*
proof (*cases xs*=[])
case *True*
have *?R* $$ (*0*,*0*) = *reduce 0 m D M* $$ (*0*,*0*)
unfolding *non-zero-positions-xs-m True M-def* **by** *simp*
also have ... ≠ *0*
by (*metis D-not0 M M-def add-pos-pos less-add-same-cancel1 m0 mk-A′-not0 n0 reduce-not0*)
finally show *?thesis* **.**
next
case *False*
show *?thesis*

by (*unfold non-zero-positions-xs-m*,
rule reduce-below-not0-case-m[*OF M' m0 n0 M-M'D*[*OF False*] *mk-A'-not0 m-le-n all-less-m D-not0*])
qed
finally show *A'-UL* $$ (*0*, *0*) ≠ *0* **.**
qed (*insert mn n hyp*, *auto*)
ultimately show *?thesis* **by** *blast*
qed
qed

lemma
assumes *A-def*: $A = A'' \,@_r\, D \cdot_m 1_m\, n$
and *A''*: *A'' ∈ carrier-mat m n* **and** *n≥2* **and** *m-le-n*: *m≥n* **and** *D>0*
shows *FindPreHNF-invertible-mat-n-ge2*: ∃ *P*. *P ∈ carrier-mat* (*m+n*) (*m+n*) ∧ *invertible-mat P* ∧ *FindPreHNF abs-flag D A* = *P* ∗ *A*
and *FindPreHNF-echelon-form-n-ge2*: *echelon-form-JNF* (*FindPreHNF abs-flag D A*)
using *FindPreHNF-works-n-ge2*[*OF assms*] **by** *blast+*

lemma *FindPreHNF-invertible-mat*:
assumes *A-def*: $A = A'' \,@_r\, D \cdot_m 1_m\, n$
and *A''*: *A'' ∈ carrier-mat m n* **and** *n0*: *0<n* **and** *mn*: *m≥n* **and** *D*: *D>0*
shows ∃ *P*. *P ∈ carrier-mat* (*m+n*) (*m+n*) ∧ *invertible-mat P* ∧ *FindPreHNF abs-flag D A* = *P* ∗ *A*
proof −
have *A*: *A ∈ carrier-mat* (*m+n*) *n* **using** *A-def A''* **by** *auto*
show *?thesis*
proof (*cases m+n<2*)
case *True*
show *?thesis* **by** (*rule FindPreHNF-invertible-mat-2xn*[*OF A True*])
next
case *False* **note** *m-ge2* = *False*
show *?thesis*
proof (*cases n<2*)
case *True*
show *?thesis* **by** (*rule FindPreHNF-invertible-mat-mx2*[*OF A-def A'' True n0 D mn*])
next
case *False*
show *?thesis*
by (*rule FindPreHNF-invertible-mat-n-ge2*[*OF A-def A'' - mn D*], *insert False*, *auto*)
qed
qed
qed

lemma *FindPreHNF-echelon-form*:
assumes *A-def*: $A = A'' \,@_r\, D \cdot_m 1_m\, n$

```
    and A'': A'' ∈ carrier-mat m n and mn: m≥n and D: D>0
  shows echelon-form-JNF (FindPreHNF abs-flag D A)
proof −
  have A: A ∈ carrier-mat (m+n) n using A-def A'' by auto
  have FindPreHNF: (FindPreHNF abs-flag D A) ∈ carrier-mat (m+n) n by (rule
FindPreHNF[OF A])
  show ?thesis
  proof (cases m+n<2)
    case True
    show ?thesis by (rule echelon-form-JNF-1xn[OF FindPreHNF True])
  next
    case False note m-ge2 = False
    show ?thesis
    proof (cases n<2)
      case True
      show ?thesis by (rule FindPreHNF-echelon-form-mx1[OF A-def A'' True D
mn])
    next
      case False
      show ?thesis
        by (rule FindPreHNF-echelon-form-n-ge2[OF A-def A'' - mn D], insert
False, auto)
    qed
  qed
qed
end
```

We connect the algorithm developed in the Hermite AFP entry with ours. This would permit to reuse many existing results and prove easily the soundness.

```
thm Hermite.Hermite-reduce-above.simps
thm Hermite.Hermite-of-row-i-def
thm Hermite.Hermite-of-upt-row-i-def
thm Hermite.Hermite-of-def


thm Hermite-reduce-above.simps
thm Hermite-of-row-i-def
thm Hermite-of-list-of-rows.simps
thm mod-operation.Hermite-mod-det-def


thm Hermite.Hermite-reduce-above.simps Hermite-reduce-above.simps

context includes lifting-syntax
begin

definition res-int = (λb n::int. n mod b)
```

lemma *res-function-res-int*:
res-function res-int
using *res-function-euclidean2* **unfolding** *res-int-def* **by** *auto*

lemma *HMA-Hermite-reduce-above*[*transfer-rule*]:
assumes *n*<*CARD*('*m*)
shows ((*Mod-Type-Connect.HMA-M* :: - ⇒ *int* ^ '*n* :: *mod-type* ^ '*m* :: *mod-type* ⇒ -)
===> (*Mod-Type-Connect.HMA-I*) ===> (*Mod-Type-Connect.HMA-I*) ===> (*Mod-Type-Connect.HMA-M*))
(λ*A i j*. *Hermite-reduce-above A n i j*)
(λ*A i j*. *Hermite.Hermite-reduce-above A n i j res-int*)
proof (*intro rel-funI*, *goal-cases*)
case (*1 A A' i i' j j'*)
then show *?case* **using** *assms*
proof (*induct n arbitrary*: *A A'*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc n*)
note *AA'*[*transfer-rule*] = *Suc.prems*(*1*)
note *ii'*[*transfer-rule*] = *Suc.prems*(*2*)
note *jj'*[*transfer-rule*] = *Suc.prems*(*3*)
note *Suc-n-less-m* = *Suc.prems*(*4*)

let *?H-JNF* = *HNF-Mod-Det-Algorithm.Hermite-reduce-above*
let *?H-HMA* = *Hermite.Hermite-reduce-above*
let *?from-nat-rows* = *mod-type-class.from-nat* :: - ⇒ '*m*
have *nn*[*transfer-rule*]: *Mod-Type-Connect.HMA-I n* (*?from-nat-rows n*)
unfolding *Mod-Type-Connect.HMA-I-def*
by (*simp add*: *Suc-lessD Suc-n-less-m mod-type-class.from-nat-to-nat*)

have *Anj*: *A'* $*h* (*?from-nat-rows n*) $*h j'* = *A* $$ (*n*,*j*)
by (*unfold index-hma-def*[*symmetric*], *transfer*, *simp*)
have *Aij*: *A'* $*h i'* $*h j'* = *A* $$ (*i*,*j*) **by** (*unfold index-hma-def*[*symmetric*], *transfer*, *simp*)
let *?s* = (− (*A* $$ (*n*, *j*) *div A* $$ (*i*, *j*)))
let *?s'* = ((*res-int* (*A'* $*h i'* $*h j'*) (*A'* $*h ?from-nat-rows n* $*h j'*)
− *A'* $*h ?from-nat-rows n* $*h j'*) *div A'* $*h i'* $*h j'*)
have *ss'*[*transfer-rule*]: *?s* = *?s'* **unfolding** *res-int-def Anj Aij*
by (*metis* (*no-types*, *opaque-lifting*) *Groups.add-ac*(*2*) *add-diff-cancel-left' div-by-0*
minus-div-mult-eq-mod more-arith-simps(*7*) *nat-arith.rule0 nonzero-mult-div-cancel-right*
uminus-add-conv-diff)
have *H-JNF-eq*: *?H-JNF A* (*Suc n*) *i j* = *?H-JNF* (*addrow* (− (*A* $$ (*n*, *j*) *div A* $$ (*i*, *j*))) *n i A*) *n i j*
by *auto*
have *H-HMA-eq*: *?H-HMA A'* (*Suc n*) *i' j' res-int* = *?H-HMA* (*row-add A'* (*?from-nat-rows n*) *i' ?s'*) *n i' j' res-int*

```
    by (auto simp add: Let-def)
  have Mod-Type-Connect.HMA-M (?H-JNF (addrow ?s n i A) n i j)
    (?H-HMA (row-add A' (?from-nat-rows n) i' ?s') n i' j' res-int)
    by (rule Suc.hyps[OF - ii' jj'], transfer-prover, insert Suc-n-less-m, simp)
  thus ?case using H-JNF-eq H-HMA-eq by auto
 qed
qed


corollary HMA-Hermite-reduce-above':
 assumes n<CARD('m)
 and Mod-Type-Connect.HMA-M A (A':: int ^ 'n :: mod-type ^ 'm :: mod-type)
 and Mod-Type-Connect.HMA-I i i' and Mod-Type-Connect.HMA-I j j'
 shows Mod-Type-Connect.HMA-M (Hermite-reduce-above A n i j) (Hermite.Hermite-reduce-above
A' n i' j' res-int)
 using HMA-Hermite-reduce-above assms unfolding rel-fun-def by metis


lemma HMA-Hermite-of-row-i[transfer-rule]:
 assumes upt-A: upper-triangular' A
 and AA': Mod-Type-Connect.HMA-M A (A':: int ^ 'n :: mod-type ^ 'm :: mod-type)
 and ii': Mod-Type-Connect.HMA-I i i'
 shows Mod-Type-Connect.HMA-M (Hermite-of-row-i A i)
 (Hermite.Hermite-of-row-i ass-function-euclidean res-int A' i')
proof -
 note AA'[transfer-rule]
 note ii'[transfer-rule]
 have i: i<dim-row A
   by (metis (full-types) AA' ii' Mod-Type-Connect.HMA-I-def
       Mod-Type-Connect.dim-row-transfer-rule mod-type-class.to-nat-less-card)
 show ?thesis
 proof (cases is-zero-row i' A')
   case True
   hence is-zero-row-JNF i A by (transfer, simp)
   hence find-fst-non0-in-row i A = None using find-fst-non0-in-row-None[OF -
upt-A i] by auto
    thus ?thesis using True AA' unfolding Hermite.Hermite-of-row-i-def Her-
mite-of-row-i-def by auto
 next
   case False
   have nz-iA: ¬ is-zero-row-JNF i A using False by transfer
   hence find-fst-non0-in-row i A ≠ None using find-fst-non0-in-row-None[OF -
upt-A i] by auto
   from this obtain j where j: find-fst-non0-in-row i A = Some j by blast
   have j-eq: j = (LEAST n. A $$ (i,n) ≠ 0)
     by (rule find-fst-non0-in-row-LEAST[OF - upt-A j i], auto)
   have H-JNF-rw: (Hermite-of-row-i A i) =
   (if A $$ (i, j) < 0 then Hermite-reduce-above (multrow i (− 1) A) i i j
    else Hermite-reduce-above A i i j) unfolding Hermite-of-row-i-def using j by
```

```
auto
  let ?H-HMA = Hermite.Hermite-of-row-i
  let ?j' = (LEAST n. A' $h i' $h n ≠ 0)
  have ii'2: (mod-type-class.to-nat i') = i using ii'
   by (simp add: Mod-Type-Connect.HMA-I-def)
  have jj'[transfer-rule]: Mod-Type-Connect.HMA-I j ?j'
    unfolding j-eq index-hma-def[symmetric] by (rule HMA-LEAST[OF AA' ii'
nz-iA])
  have Aij: A $$ (i, j) = A' $h i' $h (LEAST n. A' $h i' $h n ≠ 0)
    by (subst index-hma-def[symmetric], transfer', simp)
  have H-HMA-rw: ?H-HMA ass-function-euclidean res-int A' i' =
  Hermite.Hermite-reduce-above (mult-row A' i' (|A' $h i' $h ?j'|
    div A' $h i' $h ?j'))
   (mod-type-class.to-nat i') i' ?j' res-int
    unfolding Hermite.Hermite-of-row-i-def Let-def ass-function-euclidean-def
    by (auto simp add: False)
  have im: i < CARD('m) using ii' unfolding Mod-Type-Connect.HMA-I-def
    using mod-type-class.to-nat-less-card by blast
  show ?thesis
  proof (cases A $$ (i, j) < 0)
    case True
    have A'i'j'-le-0: A' $h i' $h ?j' < 0 using Aij True by auto
    hence 1: (|A' $h i' $h ?j'| div A' $h i' $h ?j')
       = −1 using div-pos-neg-trivial by auto
    have [transfer-rule]: Mod-Type-Connect.HMA-M (multrow i (− 1) A)
    (mult-row A' i' (|A' $h i' $h ?j'|
    div A' $h i' $h ?j')) unfolding 1 by transfer-prover
    have H-HMA-rw2: Hermite-of-row-i A i = Hermite-reduce-above (multrow i
(− 1) A) i i j
      using True H-JNF-rw by auto
    have ∗: Mod-Type-Connect.HMA-M (Hermite-reduce-above (multrow i (− 1)
A) i i j)
      (Hermite.Hermite-reduce-above (mult-row A' i' (|A' $h i' $h ?j'|
      div A' $h i' $h ?j'))
      (mod-type-class.to-nat i') i' ?j' res-int)
      unfolding 1 ii'2
      by (rule HMA-Hermite-reduce-above'[OF im - ii' jj'], transfer-prover)
       show ?thesis unfolding H-JNF-rw H-HMA-rw unfolding H-HMA-rw2
using True ∗ by auto
  next
    case False
    have Aij-not0: A $$ (i, j) ≠ 0 using j-eq nz-iA
      by (metis (mono-tags) LeastI is-zero-row-JNF-def)
    have A'i'j'-le-0: A' $h i' $h ?j' > 0 using False Aij-not0 Aij by auto
    hence 1: (|A' $h i' $h ?j'| div A' $h i' $h ?j') = 1 by auto
    have H-HMA-rw2: Hermite-of-row-i A i = Hermite-reduce-above A i i j
      using False H-JNF-rw by auto
    have ∗: ?H-HMA ass-function-euclidean res-int A' i' =
    (Hermite.Hermite-reduce-above A' (mod-type-class.to-nat i') i' ?j' res-int)
```

```
      using H-HMA-rw unfolding 1 unfolding mult-row-1-id by simp
    have Mod-Type-Connect.HMA-M (Hermite-reduce-above A i i j)
      (Hermite.Hermite-reduce-above A' (mod-type-class.to-nat i') i' ?j' res-int)
      unfolding 1 ii'2
      by (rule HMA-Hermite-reduce-above'[OF im AA' ii' jj'])
    then show ?thesis using H-HMA-rw * H-HMA-rw2 by presburger
  qed
 qed
qed


lemma Hermite-of-list-of-rows-append:
Hermite-of-list-of-rows A (xs @ [x]) = Hermite-of-row-i (Hermite-of-list-of-rows A
xs) x
 by (induct xs arbitrary: A, auto)


lemma Hermite-reduce-above[simp]: Hermite-reduce-above A n i j ∈ carrier-mat
(dim-row A) (dim-col A)
proof (induct n arbitrary: A)
 case 0
 then show ?case by auto
next
 case (Suc n)
 let ?A = (addrow (− (A $$ (n, j) div A $$ (i, j))) n i A)
 have Hermite-reduce-above A (Suc n) i j = Hermite-reduce-above ?A n i j
  by (auto simp add: Let-def)
 also have ... ∈ carrier-mat (dim-row ?A) (dim-col ?A) by(rule Suc.hyps)
 finally show ?case by auto
qed


lemma Hermite-of-row-i: Hermite-of-row-i A i ∈ carrier-mat (dim-row A) (dim-col
A)
proof −
 have Hermite-reduce-above (multrow i (− 1) A) i i a
  ∈ carrier-mat (dim-row (multrow i (− 1) A)) (dim-col (multrow i (− 1) A))
  for a by (rule Hermite-reduce-above)
 thus ?thesis
  unfolding Hermite-of-row-i-def using Hermite-reduce-above
  by (cases find-fst-non0-in-row i A, auto)
qed

end
```

We now move more lemmas from HOL Analysis (with mod-type restrictions) to the JNF matrix representation.

```
context
begin
```

private lemma *echelon-form-Hermite-of-row-mod-type*:
fixes *A*::*int mat*
assumes *A* ∈ *carrier-mat CARD*('*m*::*mod-type*) *CARD*('*n*::*mod-type*)
assumes *eA*: *echelon-form-JNF A*
and *i*: *i*<*CARD*('*m*)
shows *echelon-form-JNF* (*Hermite-of-row-i A i*)
proof −
have *uA*: *upper-triangular' A* **by** (*rule echelon-form-JNF-imp-upper-triangular*[*OF eA*])
define *A'* **where** *A'* = (*Mod-Type-Connect.to-hma*$_m$ *A* :: *int* ^'*n* :: *mod-type* ^'*m* :: *mod-type*)
define *i'* **where** *i'* = (*Mod-Type.from-nat i* :: '*m*)
have *AA'*[*transfer-rule*]: *Mod-Type-Connect.HMA-M A A'*
unfolding *Mod-Type-Connect.HMA-M-def* **using** *assms A'-def* **by** *auto*
have *ii'*[*transfer-rule*]: *Mod-Type-Connect.HMA-I i i'*
unfolding *Mod-Type-Connect.HMA-I-def i'-def* **using** *assms*
using *from-nat-not-eq order.strict-trans* **by** *blast*
have *eA'*[*transfer-rule*]: *echelon-form A'* **using** *eA* **by** *transfer*
have [*transfer-rule*]: *Mod-Type-Connect.HMA-M*
(*HNF-Mod-Det-Algorithm.Hermite-of-row-i A i*)
(*Hermite.Hermite-of-row-i ass-function-euclidean res-int A' i'*)
by (*rule HMA-Hermite-of-row-i*[*OF uA AA' ii'*])
have *echelon-form* (*Hermite.Hermite-of-row-i ass-function-euclidean res-int A' i'*)
by (*rule echelon-form-Hermite-of-row*[*OF ass-function-euclidean res-function-res-int eA'*])
thus *?thesis* **by** (*transfer*, *simp*)
qed

private lemma *echelon-form-Hermite-of-row-nontriv-mod-ring*:
fixes *A*::*int mat*
assumes *A* ∈ *carrier-mat CARD*('*m*::*nontriv mod-ring*) *CARD*('*n*::*nontriv mod-ring*)
assumes *eA*: *echelon-form-JNF A*
and *i*<*CARD*('*m*)
shows *echelon-form-JNF* (*Hermite-of-row-i A i*)
using *assms echelon-form-Hermite-of-row-mod-type* **by** (*smt* (*verit*) *CARD-mod-ring*)

lemmas *echelon-form-Hermite-of-row-nontriv-mod-ring-internalized* =
echelon-form-Hermite-of-row-nontriv-mod-ring[*unfolded CARD-mod-ring*,
internalize-sort '*m*::*nontriv*, *internalize-sort* '*b*::*nontriv*]

context
fixes *m*::*nat* **and** *n*::*nat*
assumes *local-typedef1*: ∃(*Rep* :: ('*b* ⇒ *int*)) *Abs*. *type-definition Rep Abs* {*0*..<*m* :: *int*}

assumes *local-typedef2*: $\exists$ (*Rep* :: ('*c* $\Rightarrow$ *int*)) *Abs*. *type-definition Rep Abs* {*0*..<*n* :: *int*}
and *m*: *m*>*1*
and *n*: *n*>*1*
begin

lemma *echelon-form-Hermite-of-row-nontriv-mod-ring-aux*:
fixes *A*::*int mat*
assumes *A* $\in$ *carrier-mat m n*
assumes *eA*: *echelon-form-JNF A*
and *i*<*m*
shows *echelon-form-JNF* (*Hermite-of-row-i A i*)
using *echelon-form-Hermite-of-row-nontriv-mod-ring-internalized*
[*OF type-to-set2*(*1*)[*OF local-typedef1 local-typedef2*]
type-to-set1(*1*)[*OF local-typedef1 local-typedef2*]]
using *assms*
using *type-to-set1*(*2*) *local-typedef1 local-typedef2 n m* **by** *metis*

end

context
begin

private lemma *echelon-form-Hermite-of-row-i-cancelled-first*:
$\exists$ *Rep Abs*. *type-definition Rep Abs* {*0*..<*int n*} $\Longrightarrow$ *1* < *m* $\Longrightarrow$ *1* < *n*
$\Longrightarrow$ *A* $\in$ *carrier-mat m n* $\Longrightarrow$ *echelon-form-JNF A* $\Longrightarrow$ *i* < *m*
$\Longrightarrow$ *echelon-form-JNF* (*HNF-Mod-Det-Algorithm.Hermite-of-row-i A i*)
using *echelon-form-Hermite-of-row-nontriv-mod-ring-aux*[*cancel-type-definition*, *of m n A i*]
by *auto*

private lemma *echelon-form-Hermite-of-row-i-cancelled-both*:
1 < *m* $\Longrightarrow$ *1* < *n* $\Longrightarrow$ *A* $\in$ *carrier-mat m n* $\Longrightarrow$ *echelon-form-JNF A* $\Longrightarrow$ *i* < *m*
$\Longrightarrow$ *echelon-form-JNF* (*HNF-Mod-Det-Algorithm.Hermite-of-row-i A i*)
using *echelon-form-Hermite-of-row-i-cancelled-first*[*cancel-type-definition*, *of n m A i*] **by** *simp*

lemma *echelon-form-JNF-Hermite-of-row-i'*:
fixes *A*::*int mat*
assumes *A* $\in$ *carrier-mat m n*
assumes *eA*: *echelon-form-JNF A*
and *i*<*m*
and *1* < *m* **and** *1* < *n*

shows *echelon-form-JNF* (*Hermite-of-row-i A i*)
using *echelon-form-Hermite-of-row-i-cancelled-both assms* **by** *auto*

corollary *echelon-form-JNF-Hermite-of-row-i*:
fixes *A*::*int mat*
assumes *eA*: *echelon-form-JNF A*
and *i*: *i*<*dim-row A*
shows *echelon-form-JNF* (*Hermite-of-row-i A i*)
proof (*cases dim-row A* < *2*)
case *True*
show *?thesis*
by (*rule echelon-form-JNF-1xn*[*OF Hermite-of-row-i True*])
next
case *False* **note** *m-ge2* = *False*
show *?thesis*
proof (*cases 1*<*dim-col A*)
case *True*
show *?thesis* **by** (*rule echelon-form-JNF-Hermite-of-row-i'*[*OF - eA i - True*], *insert m-ge2*, *auto*)
next
case *False*
hence *dc-01*: *dim-col A* ∈ {*0*,*1*} **by** *auto*
show *?thesis*
proof (*cases dim-col A* = *0*)
case *True*
have *H*: *Hermite-of-row-i A i* ∈ *carrier-mat* (*dim-row A*) (*dim-col A*)
using *Hermite-of-row-i* **by** *blast*
show *?thesis* **by** (*rule echelon-form-mx0*, *insert True H*, *auto*)
next
case *False*
hence *dc-1*: *dim-col A* = *1* **using** *dc-01* **by** *simp*
then show *?thesis*
proof (*cases i=0*)
case *True*
have *eA'*: *echelon-form-JNF* (*multrow 0* (− *1*) *A*)
by (*rule echelon-form-JNF-multrow*[*OF - - eA*], *insert m-ge2*, *auto*)
show *?thesis* **using** *True* **unfolding** *Hermite-of-row-i-def*
by (*cases find-fst-non0-in-row 0 A*, *insert eA eA'*, *auto*)
next
case *False*
have *all-zero*: (∀ *j*∈{*i*..<*dim-col A*}. *A* $$ (*i*, *j*) = *0*) **unfolding** *dc-1* **using** *False* **by** *auto*
hence *find-fst-non0-in-row i A* = *None* **using** *find-fst-non0-in-row-None'*[*OF - i*] **by** *blast*
hence *Hermite-of-row-i A i* = *A* **unfolding** *Hermite-of-row-i-def* **by** *auto*
then show *?thesis* **using** *eA* **by** *auto*
qed
qed

qed
qed

lemma *Hermite-of-list-of-rows*:
(*Hermite-of-list-of-rows A xs*) ∈ *carrier-mat* (*dim-row A*) (*dim-col A*)
proof (*induct xs arbitrary*: *A rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc x xs*)
let *?A* = (*Hermite-of-list-of-rows A xs*)
have *hyp*: (*Hermite-of-list-of-rows A xs*) ∈ *carrier-mat* (*dim-row A*) (*dim-col A*)
by (*rule snoc.hyps*)
have *Hermite-of-list-of-rows A* (*xs* @ [*x*]) = *Hermite-of-row-i ?A x*
using *Hermite-of-list-of-rows-append* **by** *auto*
also have ... ∈ *carrier-mat* (*dim-row ?A*) (*dim-col ?A*) **using** *Hermite-of-row-i*
by *auto*
finally show *?case* **using** *hyp* **by** *auto*
qed

lemma *echelon-form-JNF-Hermite-of-list-of-rows*:
assumes *A*∈*carrier-mat m n*
and ∀ *x*∈*set xs*. *x* < *m*
and *echelon-form-JNF A*
shows *echelon-form-JNF* (*Hermite-of-list-of-rows A xs*)
using *assms*
proof (*induct xs arbitrary*: *A rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc x xs*)
have *hyp*: *echelon-form-JNF* (*Hermite-of-list-of-rows A xs*)
by (*rule snoc.hyps*, *insert snoc.prems*, *auto*)
have *H-Axs*: (*Hermite-of-list-of-rows A xs*) ∈ *carrier-mat* (*dim-row A*) (*dim-col A*)
by (*rule Hermite-of-list-of-rows*)
have (*Hermite-of-list-of-rows A* (*xs* @ [*x*])) = *Hermite-of-row-i* (*Hermite-of-list-of-rows A xs*) *x*
using *Hermite-of-list-of-rows-append* **by** *simp*
also have *echelon-form-JNF* ...
proof (*rule echelon-form-JNF-Hermite-of-row-i*[*OF hyp*])
show *x* < *dim-row* (*Hermite-of-list-of-rows A xs*) **using** *snoc.prems H-Axs* **by** *auto*
qed
finally show *?case* **.**
qed

lemma *HMA-Hermite-of-upt-row-i*[*transfer-rule*]:
assumes *xs* = [*0*..<*i*]
and ∀ *x*∈*set xs*. *x* < *CARD*('*m*)
assumes *Mod-Type-Connect.HMA-M A* (*A'*:: *int* ^'*n* :: *mod-type* ^'*m* :: *mod-type*)

and *echelon-form-JNF A*
shows *Mod-Type-Connect.HMA-M* (*Hermite-of-list-of-rows A xs*)
(*Hermite.Hermite-of-upt-row-i A' i ass-function-euclidean res-int*)
using *assms*
proof (*induct xs arbitrary*: *A A' i rule*: *rev-induct*)
case *Nil*
have *i=0* **using** *Nil* **by** (*metis le-0-eq upt-eq-Nil-conv*)
then show *?case* **using** *Nil* **unfolding** *Hermite-of-upt-row-i-def* **by** *auto*
next
case (*snoc x xs*)
note *xs-x-eq* = *snoc.prems*(*1*)
note *all-xm* = *snoc.prems*(*2*)
note *AA'* = *snoc.prems*(*3*)
note *upt-A* = *snoc.prems*(*4*)
let *?x'* = (*mod-type-class.from-nat x*::'*m*)
have *xm*: *x* < *CARD*('*m*) **using** *all-xm* **by** *auto*
have *xx'*[*transfer-rule*]: *Mod-Type-Connect.HMA-I x ?x'*
unfolding *Mod-Type-Connect.HMA-I-def* **using** *from-nat-not-eq xm* **by** *blast*
have *last-i1*: *last* [*0*..<*i*] = *i*−*1*
by (*metis append-is-Nil-conv last-upt list.simps*(*3*) *neq0-conv xs-x-eq upt.simps*(*1*))
have *last* (*xs* @ [*x*]) = *i*−*1* **using** *xs-x-eq last-i1* **by** *auto*
hence *x-i1*: *x* = *i*−*1* **by** *auto*
have *xs-eq*: *xs* = [*0*..<*x*] **using** *xs-x-eq x-i1*
by (*metis add-diff-inverse-nat append-is-Nil-conv append-same-eq less-one list.simps*(*3*)
plus-1-eq-Suc upt-Suc upt-eq-Nil-conv)
have *list-rw*: [*0*..<*i*] = *0* #[*1*..<*i*]
by (*auto*, *metis append-is-Nil-conv list.distinct*(*2*) *upt-rec xs-x-eq*)
have *1*: *Hermite-of-list-of-rows A* (*xs* @ [*x*]) = *Hermite-of-row-i* (*Hermite-of-list-of-rows A xs*) *x*
unfolding *Hermite-of-list-of-rows-append* **by** *auto*
let *?H-upt-HA* = *Hermite.Hermite-of-upt-row-i*
let *?H-HA* = *Hermite.Hermite-of-row-i ass-function-euclidean res-int*
have (*Hermite-of-upt-row-i A' i ass-function-euclidean res-int*) =
foldl ?H-HA A' (*map mod-type-class.from-nat* [*0*..<*i*])
unfolding *Hermite-of-upt-row-i-def* **by** *auto*
also have ... = *foldl ?H-HA A'* ((*map mod-type-class.from-nat* [*0*..<*i*−*1*])@[*?x'*])
by (*metis list.simps*(*8*) *list.simps*(*9*) *map-append x-i1 xs-eq xs-x-eq*)
also have ... = *foldl ?H-HA* (*?H-upt-HA A'* (*i* − *1*) *ass-function-euclidean res-int*) [*?x'*]
unfolding *foldl-append* **unfolding** *Hermite-of-upt-row-i-def*[*symmetric*] **by** *auto*

also have ... = *?H-HA* (*Hermite-of-upt-row-i A′* (*i* − *1*) *ass-function-euclidean res-int*) *?x′* **by** *auto*
finally have *2*: *?H-upt-HA A′ i ass-function-euclidean res-int* =
?H-HA (*Hermite-of-upt-row-i A′* (*i* − *1*) *ass-function-euclidean res-int*) *?x′* **.**

have *hyp*[*transfer-rule*]: *Mod-Type-Connect.HMA-M* (*Hermite-of-list-of-rows A xs*)
(*Hermite-of-upt-row-i A′* (*i* − *1*) *ass-function-euclidean res-int*)
by (*rule snoc.hyps*[*OF - - AA′ upt-A*], *insert xs-eq x-i1 xm*, *auto*)

have *upt-H-Axs*:*upper-triangular′* (*Hermite-of-list-of-rows A xs*)
proof (*rule echelon-form-JNF-imp-upper-triangular*,
rule echelon-form-JNF-Hermite-of-list-of-rows[*OF - - upt-A*])
show *A*∈*carrier-mat* (*CARD*(*′m*)) (*CARD*(*′n*))
using *Mod-Type-Connect.dim-col-transfer-rule*
Mod-Type-Connect.dim-row-transfer-rule snoc(*4*) **by** *blast*
show ∀ *x*∈*set xs*. *x* < *CARD*(*′m*) **using** *all-xm* **by** *auto*
qed
show *?case* **unfolding** *1 2*
by (*rule HMA-Hermite-of-row-i*[*OF upt-H-Axs hyp xx′*])
qed

lemma *Hermite-Hermite-of-upt-row-i*:
assumes *a*: *ass-function ass*
and *r*: *res-function res*
and *eA*: *echelon-form A*
shows *Hermite* (*range ass*) (λ*c*. *range* (*res c*)) (*Hermite-of-upt-row-i A* (*nrows A*) *ass res*)
proof −
let *?H* = (*Hermite-of-upt-row-i A* (*nrows A*) *ass res*)
show *?thesis*
proof (*rule Hermite-intro*, *auto*)
show *Complete-set-non-associates* (*range ass*)
by (*simp add*: *ass-function-Complete-set-non-associates a*)
show *Complete-set-residues* (λ*c*. *range* (*res c*))
by (*simp add*: *r res-function-Complete-set-residues*)
show *echelon-form ?H*
by (*rule echelon-form-Hermite-of-upt-row-i*[*OF eA a r*])
fix *i*
assume *i*: ¬ *is-zero-row i ?H*
show *?H* $ *i* $ (*LEAST n*. *?H* $ *i* $ *n* ≠ *0*) ∈ *range ass*
proof −
have *non-zero-i-eA*: ¬ *is-zero-row i A*
using *Hermite-of-upt-row-preserves-zero-rows*[*OF - - a r*] *i eA* **by** *blast*
have *least*: (*LEAST n*. *?H* $*h i* $*h n* ≠ *0*) = (*LEAST n*. *A* $*h i* $*h n* ≠ *0*)
by (*rule Hermite-of-upt-row-i-Least*[*OF non-zero-i-eA eA a r*], *simp*)
have *?H* $ *i* $ (*LEAST n*. *A* $ *i* $ *n* ≠ *0*) ∈ *range ass*
by (*rule Hermite-of-upt-row-i-in-range*[*OF non-zero-i-eA eA a r*], *auto*)

thus *?thesis* **unfolding** *least* **by** *auto*
qed
next
fix *i j* **assume** *i*: ¬ *is-zero-row i ?H* **and** *j*: *j* < *i*
show *?H* $ *j* $ (*LEAST n. ?H* $ *i* $ *n* ≠ *0*)
∈ *range* (*res* (*?H* $ *i* $ (*LEAST n. ?H* $ *i* $ *n* ≠ *0*)))
proof −
have *non-zero-i-eA*: ¬ *is-zero-row i A*
using *Hermite-of-upt-row-preserves-zero-rows*[*OF - - a r*] *i eA* **by** *blast*
have *least*: (*LEAST n. ?H* $*h i* $*h n* ≠ *0*) = (*LEAST n. A* $*h i* $*h n* ≠ *0*)
by (*rule Hermite-of-upt-row-i-Least*[*OF non-zero-i-eA eA a r*], *simp*)
have *?H* $ *j* $ (*LEAST n. A* $ *i* $ *n* ≠ *0*) ∈ *range* (*res* (*?H* $ *i* $ (*LEAST n. A* $ *i* $ *n* ≠ *0*)))
by (*rule Hermite-of-upt-row-i-in-range-res*[*OF non-zero-i-eA eA a r - - j*], *auto*)
thus *?thesis* **unfolding** *least* **by** *auto*
qed
qed
qed

lemma *Hermite-of-row-i-0*:
Hermite-of-row-i A 0 = *A* ∨ *Hermite-of-row-i A 0* = *multrow 0* (− *1*) *A*
by (*cases find-fst-non0-in-row 0 A*, *unfold Hermite-of-row-i-def*, *auto*)

lemma *Hermite-JNF-intro*:
assumes
Complete-set-non-associates associates (*Complete-set-residues res*) *echelon-form-JNF A*
(∀ *i*<*dim-row A*. ¬ *is-zero-row-JNF i A* ⟶ *A* $$ (*i*, *LEAST n. A* $$ (*i*, *n*) ≠ *0*) ∈ *associates*)
(∀ *i*<*dim-row A*. ¬ *is-zero-row-JNF i A* ⟶ (∀ *j*. *j*<*i* ⟶ *A* $$ (*j*, (*LEAST n. A* $$ (*i*, *n*) ≠ *0*))
∈ *res* (*A* $$ (*i*,(*LEAST n. A* $$ (*i*,*n*) ≠ *0*)))))
shows *Hermite-JNF associates res A*
using *assms* **unfolding** *Hermite-JNF-def* **by** *auto*

lemma *least-multrow*:
assumes *A* ∈ *carrier-mat m n* **and** *i*<*m* **and** *eA*: *echelon-form-JNF A*
assumes *ia*: *ia* < *dim-row A* **and** *nz-ia-mrA*: ¬ *is-zero-row-JNF ia* (*multrow i* (− *1*) *A*)
shows (*LEAST n. multrow i* (− *1*) *A* $$ (*ia*, *n*) ≠ *0*) = (*LEAST n. A* $$ (*ia*, *n*) ≠ *0*)
proof (*rule Least-equality*)
have *nz-ia-A*: ¬ *is-zero-row-JNF ia A* **using** *nz-ia-mrA ia* **by** *auto*
have *Least-Aian-n*: (*LEAST n. A* $$ (*ia*, *n*) ≠ *0*) < *dim-col A*
by (*smt* (*verit*) *dual-order.strict-trans is-zero-row-JNF-def not-less-Least not-less-iff-gr-or-eq nz-ia-A*)
show *multrow i* (− *1*) *A* $$ (*ia*, *LEAST n. A* $$ (*ia*, *n*) ≠ *0*) ≠ *0*

by (*smt* (*verit*) *LeastI Least-Aian-n class-cring.cring-simprules*(22) *equation-minus-iff ia*
index-mat-multrow(1) *is-zero-row-JNF-def mult-minus1 nz-ia-A*)
show ⋀*y*. *multrow i* (− 1) *A* $$ (*ia*, *y*) ≠ 0 ⟹ (*LEAST n*. *A* $$ (*ia*, *n*) ≠ 0) ≤ *y*
by (*metis* (*mono-tags*, *lifting*) *Least-Aian-n class-cring.cring-simprules*(22) *ia*
index-mat-multrow(1) *leI mult-minus1 order.strict-trans wellorder-Least-lemma*(2))
qed

lemma *Hermite-Hermite-of-row-i*:
assumes *A*: *A* ∈ *carrier-mat 1 n*
shows *Hermite-JNF* (*range ass-function-euclidean*) (λ*c*. *range* (*res-int c*)) (*Hermite-of-row-i A 0*)
proof (*rule Hermite-JNF-intro*)
show *Complete-set-non-associates* (*range ass-function-euclidean*)
using *ass-function-Complete-set-non-associates ass-function-euclidean* **by** *blast*
show *Complete-set-residues* (λ*c*. *range* (*res-int c*))
using *res-function-Complete-set-residues res-function-res-int* **by** *blast*
show *echelon-form-JNF* (*HNF-Mod-Det-Algorithm.Hermite-of-row-i A 0*)
by (*metis* (*full-types*) *assms carrier-matD*(1) *echelon-form-JNF-Hermite-of-row-i*
echelon-form-JNF-def less-one not-less-zero)
let *?H* = *Hermite-of-row-i A 0*
show ∀ *i*<*dim-row ?H*. ¬ *is-zero-row-JNF i ?H*
⟶ *?H* $$ (*i*, *LEAST n*. *?H* $$ (*i*, *n*) ≠ 0) ∈ *range ass-function-euclidean*
proof (*auto*)
fix *i* **assume** *i*: *i*<*dim-row ?H* **and** *nz-iH*: ¬ *is-zero-row-JNF i ?H*
have *nz-iA*: ¬ *is-zero-row-JNF i A*
by (*metis* (*full-types*) *Hermite-of-row-i Hermite-of-row-i-0 carrier-matD*(1)
i is-zero-row-JNF-multrow nz-iH)
have *?H* $$ (*i*, *LEAST n*. *?H* $$ (*i*, *n*) ≠ 0) ≥ 0
proof (*cases find-fst-non0-in-row 0 A*)
case *None*
then show *?thesis* **using** *nz-iH* **unfolding** *Hermite-of-row-i-def*
by (*smt* (*verit*) *HNF-Mod-Det-Algorithm.Hermite-of-row-i-def upper-triangular′-def assms*
carrier-matD(1) *find-fst-non0-in-row-None i less-one not-less-zero option.simps*(4))
next
case (*Some a*)
have *upA*: *upper-triangular′ A* **using** *A* **unfolding** *upper-triangular′-def* **by** *auto*
have *eA*: *echelon-form-JNF A* **by** (*metis A Suc-1 echelon-form-JNF-1xn lessI*)
have *i0*: *i*=*0* **using** *Hermite-of-row-i*[*of A 0*] *A i* **by** *auto*
have *Aia*: *A* $$ (*i*,*a*) ≠ 0 **and** *a0*: 0 ≤ *a* **and** *an*: *a*<*n*
using *i0 Some assms find-fst-non0-in-row* **by** *auto*+
have *l*: (*LEAST n*. *A* $$ (*i*, *n*) ≠ 0) = (*LEAST n*. *multrow 0* (− 1) *A* $$ (*i*, *n*) ≠ 0)
by (*rule least-multrow*[*symmetric*, *OF A - eA* -], *insert nz-iA i A i0*, *auto*)

```
    have a1: a = (LEAST n. A $$ (i, n) ≠ 0)
      by (rule find-fst-non0-in-row-LEAST[OF A upA], insert Some i0, auto)
    hence a2: a = (LEAST n.  multrow 0 (− 1) A $$ (i, n) ≠ 0) unfolding l
by simp
    have m1: multrow 0 (− 1) A $$ (i, LEAST n. multrow 0 (− 1) A $$ (i, n)
≠ 0)
        = (− 1) ∗ A $$ (i, LEAST n. A $$ (i, n) ≠ 0)
       by (metis Hermite-of-row-i-0 a1 a2 an assms carrier-matD(2) i i0 in-
dex-mat-multrow(1,4))
   then show ?thesis using nz-iH Some a1 Aia a2 i0 unfolding Hermite-of-row-i-def
by auto
   qed
   thus ?H $$ (i, LEAST n. ?H $$ (i, n) ≠ 0) ∈ range ass-function-euclidean
     using ass-function-int ass-function-int-UNIV by auto
   qed
   show ∀ i<dim-row ?H. ¬ is-zero-row-JNF i ?H ⟶ (∀ j<i. ?H $$ (j, LEAST
n. ?H $$ (i, n) ≠ 0)
   ∈ range (res-int (?H $$ (i, LEAST n. ?H $$ (i, n) ≠ 0))))
     using Hermite-of-row-i[of A 0] A by auto
 qed

lemma Hermite-of-row-i-0-eq-0:
 assumes A: A∈carrier-mat m n and i: i>0 and eA: echelon-form-JNF A and
im: i<m
   and n0: 0<n
 shows Hermite-of-row-i A 0 $$ (i, 0) = 0
proof −
 have Ai0: A $$ (i, 0) = 0 by (rule echelon-form-JNF-first-column-0[OF eA A
i im n0])
 show ?thesis
 proof (cases find-fst-non0-in-row 0 A)
   case None
   thus ?thesis using Ai0 unfolding Hermite-of-row-i-def by auto
 next
   case (Some a)
   have A $$ (0, a) ≠ 0  and a0: 0 ≤ a and an: a<n
     using find-fst-non0-in-row[OF A Some] A by auto
  then show ?thesis using Some Ai0 A an a0 im unfolding Hermite-of-row-i-def
mat-multrow-def by auto
 qed
qed



lemma Hermite-Hermite-of-row-i-mx1:
 assumes A: A ∈ carrier-mat m 1 and eA: echelon-form-JNF A
 shows Hermite-JNF (range ass-function-euclidean) (λc. range (res-int c)) (Hermite-of-row-i
A 0)
proof (rule Hermite-JNF-intro)
```

```
  show Complete-set-non-associates (range ass-function-euclidean)
    using ass-function-Complete-set-non-associates ass-function-euclidean by blast
  show Complete-set-residues (λc. range (res-int c))
    using res-function-Complete-set-residues res-function-res-int by blast
  have H: Hermite-of-row-i A 0 : carrier-mat m 1 using A Hermite-of-row-i[of
A] by auto
  have upA: upper-triangular' A
    by (simp add: eA echelon-form-JNF-imp-upper-triangular)
  show eH: echelon-form-JNF (Hermite-of-row-i A 0)
  proof (rule echelon-form-JNF-mx1[OF H])
    show ∀ i∈{1..<m}. HNF-Mod-Det-Algorithm.Hermite-of-row-i A 0 $$ (i, 0) =
0
      using Hermite-of-row-i-0-eq-0 assms by auto
  qed (simp)
  let ?H = Hermite-of-row-i A 0
  show ∀ i<dim-row ?H. ¬ is-zero-row-JNF i ?H
      ⟶ ?H $$ (i, LEAST n. ?H $$ (i, n) ≠ 0) ∈ range ass-function-euclidean
  proof (auto)
    fix i assume i: i<dim-row ?H and nz-iH: ¬ is-zero-row-JNF i ?H
    have nz-iA: ¬ is-zero-row-JNF i A
      by (metis (full-types) Hermite-of-row-i Hermite-of-row-i-0 carrier-matD(1)
          i is-zero-row-JNF-multrow nz-iH)
    have ?H $$ (i, LEAST n. ?H $$ (i, n) ≠ 0) ≥ 0
    proof (cases find-fst-non0-in-row 0 A)
      case None
      have is-zero-row-JNF i A
      by (metis H upper-triangular'-def None assms(1) carrier-matD find-fst-non0-in-row-None
            i is-zero-row-JNF-def less-one linorder-neqE-nat not-less0 upA)
        then show ?thesis using nz-iH None unfolding Hermite-of-row-i-def by
auto
     next
      case (Some a)
      have Aia: A $$ (0,a) ≠ 0 and a0: 0 ≤ a and an: a<1
        using find-fst-non0-in-row[OF A Some] A by auto
       have nz-j-mA: is-zero-row-JNF j (multrow 0 (− 1) A) if j0: j>0 and jm:
j<m for j
        unfolding is-zero-row-JNF-def using A j0 jm upA by auto
      show ?thesis
      proof (cases i=0)
        case True
        then show ?thesis
          using nz-iH Some nz-j-mA A H i Aia an unfolding Hermite-of-row-i-def
by auto
      next
        case False
        have nz-iA: is-zero-row-JNF i A
       by (metis False H Hermite-of-row-i-0 carrier-matD(1) i is-zero-row-JNF-multrow
not-gr0 nz-iH nz-j-mA)
        hence is-zero-row-JNF i (multrow 0 (− 1) A) using A H i by auto
```

```
      then show ?thesis using nz-iH Some nz-j-mA False nz-iA
        unfolding Hermite-of-row-i-def by fastforce
    qed
  qed
  thus ?H $$ (i, LEAST n. ?H $$ (i, n) ≠ 0) ∈ range ass-function-euclidean
    using ass-function-int ass-function-int-UNIV by auto
  qed
  show ∀ i<dim-row ?H. ¬ is-zero-row-JNF i ?H ⟶ (∀ j<i. ?H $$ (j, LEAST
n. ?H $$ (i, n) ≠ 0)
  ∈ range (res-int (?H $$ (i, LEAST n. ?H $$ (i, n) ≠ 0))))
  proof auto
    fix i j assume i: i<dim-row ?H and nz-iH: ¬ is-zero-row-JNF i ?H and ji:
j<i
    have i=0
      by (metis H upper-triangular'-def One-nat-def nz-iH eH i carrier-matD(2)
nat-neq-iff
          echelon-form-JNF-imp-upper-triangular is-zero-row-JNF-def less-Suc0
not-less-zero)
    thus ?H $$ (j, LEAST n. ?H $$ (i, n) ≠ 0)
        ∈ range (res-int (?H $$ (i, LEAST n. ?H $$ (i, n) ≠ 0))) using ji by
auto
  qed
qed


lemma Hermite-of-list-of-rows-1xn:
  assumes A: A ∈ carrier-mat 1 n
    and eA: echelon-form-JNF A
    and x: ∀ x ∈ set xs. x < 1 and xs: xs≠[]
  shows Hermite-JNF (range ass-function-euclidean)
  (λc. range (res-int c)) (Hermite-of-list-of-rows A xs)
  using x xs
proof (induct xs rule: rev-induct)
  case Nil
  then show ?case by auto
next
  case (snoc x xs)
  have x0: x=0 using snoc.prems by auto
  show ?case
  proof (cases xs = [])
    case True
    have Hermite-of-list-of-rows A (xs @ [x]) = Hermite-of-row-i A 0
      unfolding Hermite-of-list-of-rows-append x0 using True by auto
    then show ?thesis using Hermite-Hermite-of-row-i[OF A] by auto
  next
    case False
    have x0: x=0 using snoc.prems by auto
    have hyp: Hermite-JNF (range ass-function-euclidean)
      (λc. range (res-int c)) (Hermite-of-list-of-rows A xs)
```

```
      by (rule snoc.hyps, insert snoc.prems False, auto)
  have Hermite-of-list-of-rows A (xs @ [x]) = Hermite-of-row-i (Hermite-of-list-of-rows
A xs) 0
    unfolding Hermite-of-list-of-rows-append hyp x0 ..
   thus ?thesis
   by (metis A Hermite-Hermite-of-row-i Hermite-of-list-of-rows carrier-matD(1))
 qed
qed


lemma Hermite-of-row-i-id-mx1:
 assumes H': Hermite-JNF (range ass-function-euclidean) (λc. range (res-int c))
A
   and x: x<dim-row A and A: A∈carrier-mat m 1
shows Hermite-of-row-i A x = A
proof (cases find-fst-non0-in-row x A)
 case None
 then show ?thesis unfolding Hermite-of-row-i-def by auto
next
 case (Some a)
 have eH: echelon-form-JNF A using H' unfolding Hermite-JNF-def by simp
 have ut-A: upper-triangular' A by (simp add: eH echelon-form-JNF-imp-upper-triangular)
 have a-least: a = (LEAST n. A $$ (x,n) ≠ 0)
   by (rule find-fst-non0-in-row-LEAST[OF - ut-A Some], insert x, auto)
 have Axa: A $$ (x, a) ≠ 0 and xa: x≤a and a: a<dim-col A
   using find-fst-non0-in-row[OF A Some] unfolding a-least by auto
 have nz-xA: ¬ is-zero-row-JNF x A using Axa xa x a unfolding is-zero-row-JNF-def
by blast
 have a0: a = 0 using a A by auto
 have x0: x=0 using echelon-form-JNF-first-column-0[OF eH A] Axa a0 xa by
blast
 have A $$ (x, a) ∈ (range ass-function-euclidean)
   using nz-xA H' x unfolding a-least unfolding Hermite-JNF-def by auto
 hence A $$ (x, a) > 0 using Axa unfolding image-def ass-function-euclidean-def
by auto
 then show ?thesis unfolding Hermite-of-row-i-def using Some x0 by auto
qed

lemma Hermite-of-row-i-id-mx1':
 assumes eA: echelon-form-JNF A
   and x: x<dim-row A and A: A∈carrier-mat m 1
shows Hermite-of-row-i A x = A ∨ Hermite-of-row-i A x = multrow 0 (− 1) A
proof (cases find-fst-non0-in-row x A)
 case None
 then show ?thesis unfolding Hermite-of-row-i-def by auto
next
 case (Some a)
 have ut-A: upper-triangular' A by (simp add: eA echelon-form-JNF-imp-upper-triangular)
 have a-least: a = (LEAST n. A $$ (x,n) ≠ 0)
```

```
    by (rule find-fst-non0-in-row-LEAST[OF - ut-A Some], insert x, auto)
  have Axa: A $$ (x, a) ≠ 0 and xa: x≤a and a: a<dim-col A
    using find-fst-non0-in-row[OF A Some] unfolding a-least by auto
 have nz-xA: ¬ is-zero-row-JNF x A using Axa xa x a unfolding is-zero-row-JNF-def
by blast
  have a0: a = 0 using a A by auto
  have x0: x=0 using echelon-form-JNF-first-column-0[OF eA A] Axa a0 xa by
blast
  show ?thesis by (cases A $$(x,a)>0, unfold Hermite-of-row-i-def, insert Some
x0, auto)
qed


lemma Hermite-of-list-of-rows-mx1:
  assumes A: A ∈ carrier-mat m 1
    and eA: echelon-form-JNF A
    and x: ∀ x ∈ set xs. x < m and xs: xs=[0..<i] and i: i>0
  shows Hermite-JNF (range ass-function-euclidean)
  (λc. range (res-int c)) (Hermite-of-list-of-rows A xs)
  using x xs i
proof (induct xs arbitrary: i rule: rev-induct)
  case Nil
  then show ?case by (metis neq0-conv not-less upt-eq-Nil-conv)
next
  case (snoc x xs)
  note all-n-xs-x = snoc.prems(1)
  note xs-x = snoc.prems(2)
  note i0 = snoc.prems(3)
 have i-list-rw:[0..<i] = [0..<i−1] @ [i−1] using i0 less-imp-Suc-add by fastforce
  hence xs: xs = [0..<i−1] using xs-x by force
  hence x: x=i−1 using i-list-rw xs-x by auto
  have H: Hermite-of-list-of-rows A xs ∈ carrier-mat m 1
    using A Hermite-of-list-of-rows[of A xs] by auto
  show ?case
  proof (cases i−1=0)
    case True
    hence xs-empty: xs = [] using xs by auto
    have ∗: Hermite-of-list-of-rows A (xs @ [x]) = Hermite-of-row-i A 0
      unfolding Hermite-of-list-of-rows-append xs-empty x True by simp
    show ?thesis unfolding ∗ by (rule Hermite-Hermite-of-row-i-mx1[OF A eA])
  next
    case False
    have hyp: Hermite-JNF (range ass-function-euclidean)
      (λc. range (res-int c)) (Hermite-of-list-of-rows A xs)
      by (rule snoc.hyps[OF - xs], insert False all-n-xs-x, auto)
    have Hermite-of-list-of-rows A (xs @ [x])
       = Hermite-of-row-i (Hermite-of-list-of-rows A xs) x
      unfolding Hermite-of-list-of-rows-append ..
    also have ... = (Hermite-of-list-of-rows A xs)
```

```
    by (rule Hermite-of-row-i-id-mx1[OF hyp - H], insert snoc.prems H x, auto)
   finally show ?thesis using hyp by auto
  qed
qed


lemma invertible-Hermite-of-list-of-rows-1xn:
  assumes A ∈ carrier-mat 1 n
  shows ∃ P. P ∈ carrier-mat 1 1 ∧ invertible-mat P ∧ Hermite-of-list-of-rows A
[0..<1] = P * A
proof −
  let ?H = Hermite-of-list-of-rows A [0..<1]
  have ?H = Hermite-of-row-i A 0 by auto
  hence H-or: ?H = A ∨ ?H = multrow 0 (− 1) A
    using Hermite-of-row-i-0 by simp
  show ?thesis
  proof (cases ?H = A)
    case True
    then show ?thesis
      by (metis assms invertible-mat-one left-mult-one-mat one-carrier-mat)
  next
    case False
    hence H-mr: ?H = multrow 0 (− 1) A using H-or by simp
    let ?M = multrow-mat 1 0 (−1)::int mat
    show ?thesis
    proof (rule exI[of - ?M])
      have ?M ∈ carrier-mat 1 1 by auto
      moreover have invertible-mat ?M
      by (metis calculation det-multrow-mat det-one dvd-mult-right invertible-iff-is-unit-JNF
          invertible-mat-one one-carrier-mat square-eq-1-iff zero-less-one-class.zero-less-one)
      moreover have ?H= ?M * A
        by (metis H-mr assms multrow-mat)
      ultimately show ?M ∈ carrier-mat 1 1 ∧ invertible-mat (?M)
 ∧ Hermite-of-list-of-rows A [0..<1] = ?M * A by blast
    qed
  qed
qed


lemma invertible-Hermite-of-list-of-rows-mx1':
  assumes A: A ∈ carrier-mat m 1 and eA: echelon-form-JNF A
    and xs-i: xs = [0..<i] and xs-m: ∀ x∈set xs. x < m and i: i>0
  shows ∃ P. P ∈ carrier-mat m m ∧ invertible-mat P ∧ Hermite-of-list-of-rows
A xs = P * A
  using xs-i xs-m i
proof (induct xs arbitrary: i rule: rev-induct)
  case Nil
```

then show *?case* **by** (*metis diff-zero length-upt list.size*(*3*) *zero-order*(*3*))
next
case (*snoc x xs*)
note *all-n-xs-x* = *snoc.prems*(*2*)
note *xs-x* = *snoc.prems*(*1*)
note *i0* = *snoc.prems*(*3*)
have *i-list-rw*:[*0*..<*i*] = [*0*..<*i*−*1*] @ [*i*−*1*] **using** *i0 less-imp-Suc-add* **by** *fastforce*
hence *xs*: *xs* = [*0*..<*i*−*1*] **using** *xs-x* **by** *force*
hence *x*: *x*=*i*−*1* **using** *i-list-rw xs-x* **by** *auto*
have *H*: *Hermite-of-list-of-rows A xs* ∈ *carrier-mat m 1*
using *A Hermite-of-list-of-rows*[*of A xs*] **by** *auto*
show *?case*
proof (*cases i*−*1*=*0*)
case *True*
hence *xs-empty*: *xs* = [] **using** *xs* **by** *auto*
let *?H* = *Hermite-of-list-of-rows A* (*xs* @ [*x*])
have ∗: *Hermite-of-list-of-rows A* (*xs* @ [*x*]) = *Hermite-of-row-i A 0*
unfolding *Hermite-of-list-of-rows-append xs-empty x True* **by** *simp*
hence *H-or*: *?H* = *A* ∨ *?H* = *multrow 0* (− *1*) *A* **using** *Hermite-of-row-i-0*
by *simp*
thus *?thesis*
proof (*cases ?H*=*A*)
case *True*
then show *?thesis* **unfolding** ∗
by (*metis A invertible-mat-one left-mult-one-mat one-carrier-mat*)
next
case *False*
hence *H-mr*: *?H* = *multrow 0* (− *1*) *A* **using** *H-or* **by** *simp*
let *?M* = *multrow-mat m 0* (−*1*)::*int mat*
show *?thesis*
proof (*rule exI*[*of* - *?M*])
have *?M* ∈ *carrier-mat m m* **by** *auto*
moreover have *invertible-mat ?M*
by (*metis* (*full-types*) *det-multrow-mat dvd-mult-right invertible-iff-is-unit-JNF*
invertible-mat-zero more-arith-simps(*10*) *mult-minus1-right multrow-mat-carrier*
neq0-conv)
moreover have *?H* = *?M* ∗ *A* **unfolding** *H-mr* **using** *A multrow-mat* **by**
blast
ultimately show *?M* ∈ *carrier-mat m m* ∧ *invertible-mat ?M* ∧ *?H* = *?M*
∗ *A* **by** *blast*
qed
qed
next
case *False*
let *?A* = (*Hermite-of-list-of-rows A xs*)
have *A'*: *?A* ∈ *carrier-mat m 1* **using** *A Hermite-of-list-of-rows*[*of A xs*] **by**
simp
have *hyp*: ∃ *P*. *P* ∈ *carrier-mat m m* ∧ *invertible-mat P* ∧ *Hermite-of-list-of-rows*
A xs = *P* ∗ *A*

```
    by (rule snoc.hyps[OF xs], insert False all-n-xs-x, auto)
  have rw: Hermite-of-list-of-rows A (xs @ [x])
    = Hermite-of-row-i (Hermite-of-list-of-rows A xs) x
    unfolding Hermite-of-list-of-rows-append ..
  have ∗: Hermite-of-row-i ?A x = ?A ∨ Hermite-of-row-i ?A x = multrow 0 (−
1) ?A
  proof (rule Hermite-of-row-i-id-mx1'[OF - - A'])
    show echelon-form-JNF ?A
      using A eA echelon-form-JNF-Hermite-of-list-of-rows snoc(3) by auto
    show x < dim-row ?A using A' x i A by (simp add: snoc(3))
  qed
  show ?thesis
  proof (cases Hermite-of-row-i ?A x = ?A)
  case True
    then show ?thesis
      by (simp add: hyp rw)
  next
    case False
    let ?M = multrow-mat m 0 (−1)::int mat
    obtain P where P: P ∈ carrier-mat m m
      and inv-P: invertible-mat P and H-PA: Hermite-of-list-of-rows A xs = P ∗
A
      using hyp by auto
    have M: ?M ∈ carrier-mat m m by auto
    have inv-M: invertible-mat ?M
    by (metis (full-types) det-multrow-mat dvd-mult-right invertible-iff-is-unit-JNF
      invertible-mat-zero more-arith-simps(10) mult-minus1-right multrow-mat-carrier
neq0-conv)
   have H-MA': Hermite-of-row-i ?A x = ?M ∗ ?A using False ∗ H multrow-mat
by metis
  have inv-MP: invertible-mat (?M∗P) using M inv-M P inv-P invertible-mult-JNF
by blast
    moreover have MP: (?M∗P) ∈ carrier-mat m m using M P by fastforce
    moreover have Hermite-of-list-of-rows A (xs @ [x]) = (?M∗P) ∗ A
      by (metis A H-MA' H-PA M P assoc-mult-mat rw)
    ultimately show ?thesis by blast
  qed
 qed
qed


corollary invertible-Hermite-of-list-of-rows-mx1:
 assumes A ∈ carrier-mat m 1 and eA: echelon-form-JNF A
 shows ∃ P. P ∈ carrier-mat m m ∧ invertible-mat P ∧ Hermite-of-list-of-rows
A [0..<m] = P ∗ A
proof (cases m=0)
 case True
 then show ?thesis
  by (auto, metis assms(1) invertible-mat-one left-mult-one-mat one-carrier-mat)
```

```
next
  case False
  then show ?thesis using invertible-Hermite-of-list-of-rows-mx1' assms by simp
qed


lemma Hermite-of-list-of-rows-mx0:
  assumes A: A ∈ carrier-mat m 0
  and xs: xs = [0..<i] and x: ∀ x∈ set xs. x < m
shows Hermite-of-list-of-rows A xs = A
  using xs x
proof (induct xs arbitrary: i rule: rev-induct)
  case Nil
  then show ?case by auto
next
  case (snoc x xs)
  note all-n-xs-x = snoc.prems(2)
  note xs-x = snoc.prems(1)
  have i0: i>0 using neq0-conv snoc(2) by fastforce
 have i-list-rw:[0..<i] = [0..<i−1] @ [i−1] using i0 less-imp-Suc-add by fastforce
  hence xs: xs = [0..<i−1] using xs-x by force
  hence x: x=i−1 using i-list-rw xs-x by auto
  have H: Hermite-of-list-of-rows A xs ∈ carrier-mat m 0
    using A Hermite-of-list-of-rows[of A xs] by auto
  define A' where A' = (Hermite-of-list-of-rows A xs)
  have A'A: A' = A by (unfold A'-def, rule snoc.hyps, insert snoc.prems xs, auto)
  have Hermite-of-list-of-rows A (xs @ [x]) = Hermite-of-row-i A' x
    using Hermite-of-list-of-rows-append A'-def by auto
  also have ... = A
  proof (cases find-fst-non0-in-row x A')
    case None
    then show ?thesis unfolding Hermite-of-row-i-def using A'A by auto
    next
      case (Some a)
    then show ?thesis
    by (metis (full-types) A'A A carrier-matD(2) find-fst-non0-in-row(3) zero-order(3))
  qed
  finally show ?case .
qed
```

Again, we move more lemmas from HOL Analysis (with mod-type restrictions) to the JNF matrix representation.

```
context
begin

private lemma Hermite-Hermite-of-list-of-rows-mod-type:
  fixes A::int mat
  assumes A ∈ carrier-mat CARD('m::mod-type) CARD('n::mod-type)
  assumes eA: echelon-form-JNF A
```

shows *Hermite-JNF* (*range ass-function-euclidean*)
(λ*c*. *range* (*res-int c*)) (*Hermite-of-list-of-rows A* [*0*..<*CARD*('*m*)])
proof −
define *A'* **where** *A'* = (*Mod-Type-Connect.to-hma*$_m$ *A* :: *int* ^'*n* :: *mod-type* ^'*m*
:: *mod-type*)
have *AA'*[*transfer-rule*]: *Mod-Type-Connect.HMA-M A A'*
unfolding *Mod-Type-Connect.HMA-M-def* **using** *assms A'-def* **by** *auto*
have *eA'*[*transfer-rule*]: *echelon-form A'* **using** *eA* **by** *transfer*
have [*transfer-rule*]: *Mod-Type-Connect.HMA-M* (*Hermite-of-list-of-rows A* [*0*..<*CARD*('*m*)])

(*Hermite-of-upt-row-i A'* (*CARD*('*m*)) *ass-function-euclidean res-int*)
by (*rule HMA-Hermite-of-upt-row-i*[*OF - - AA' eA*], *auto*)
have [*transfer-rule*]: (*range ass-function-euclidean*) = (*range ass-function-euclidean*)
..
have [*transfer-rule*]: (λ*c*. *range* (*res-int c*)) = (λ*c*. *range* (*res-int c*)) ..
have *n*: *CARD*('*m*) = *nrows A'* **using** *AA'* **unfolding** *nrows-def* **by** *auto*
have *Hermite* (*range ass-function-euclidean*) (λ*c*. *range* (*res-int c*))
(*Hermite-of-upt-row-i A'* (*CARD*('*m*)) *ass-function-euclidean res-int*)
by (*unfold n*, *rule Hermite-Hermite-of-upt-row-i*[*OF ass-function-euclidean
res-function-res-int eA'*])
thus *?thesis* **by** *transfer*
qed

private lemma *invertible-Hermite-of-list-of-rows-mod-type*:
fixes *A*::*int mat*
assumes *A* ∈ *carrier-mat CARD*('*m*::*mod-type*) *CARD*('*n*::*mod-type*)
assumes *eA*: *echelon-form-JNF A*
shows ∃ *P*. *P* ∈ *carrier-mat CARD*('*m*) *CARD*('*m*) ∧
invertible-mat P ∧ *Hermite-of-list-of-rows A* [*0*..<*CARD*('*m*)] = *P* ∗ *A*
proof −
define *A'* **where** *A'* = (*Mod-Type-Connect.to-hma*$_m$ *A* :: *int* ^'*n* :: *mod-type* ^'*m*
:: *mod-type*)
have *AA'*[*transfer-rule*]: *Mod-Type-Connect.HMA-M A A'*
unfolding *Mod-Type-Connect.HMA-M-def* **using** *assms A'-def* **by** *auto*
have *eA'*[*transfer-rule*]: *echelon-form A'* **using** *eA* **by** *transfer*
have [*transfer-rule*]: *Mod-Type-Connect.HMA-M* (*Hermite-of-list-of-rows A* [*0*..<*CARD*('*m*)])

(*Hermite-of-upt-row-i A'* (*CARD*('*m*)) *ass-function-euclidean res-int*)
by (*rule HMA-Hermite-of-upt-row-i*[*OF - - AA' eA*], *auto*)
have [*transfer-rule*]: (*range ass-function-euclidean*) = (*range ass-function-euclidean*)
..
have [*transfer-rule*]: (λ*c*. *range* (*res-int c*)) = (λ*c*. *range* (*res-int c*)) ..
have *n*: *CARD*('*m*) = *nrows A'* **using** *AA'* **unfolding** *nrows-def* **by** *auto*
have ∃ *P*. *invertible P* ∧ *Hermite-of-upt-row-i A'* (*CARD*('*m*)) *ass-function-euclidean
res-int*
= *P* ∗∗ *A'* **by** (*rule invertible-Hermite-of-upt-row-i*[*OF ass-function-euclidean*])
thus *?thesis* **by** (*transfer*, *auto*)
qed

private lemma *Hermite-Hermite-of-list-of-rows-nontriv-mod-ring*:
fixes *A*::*int mat*
assumes *A* ∈ *carrier-mat CARD*('*m*::*nontriv mod-ring*) *CARD*('*n*::*nontriv mod-ring*)
assumes *eA*: *echelon-form-JNF A*
shows *Hermite-JNF* (*range ass-function-euclidean*)
(λ*c*. *range* (*res-int c*)) (*Hermite-of-list-of-rows A* [*0*..<*CARD*('*m*)])
using *assms Hermite-Hermite-of-list-of-rows-mod-type* **by** (*smt* (*verit*) *CARD-mod-ring*)

private lemma *invertible-Hermite-of-list-of-rows-nontriv-mod-ring*:
fixes *A*::*int mat*
assumes *A* ∈ *carrier-mat CARD*('*m*::*nontriv mod-ring*) *CARD*('*n*::*nontriv mod-ring*)
assumes *eA*: *echelon-form-JNF A*
shows ∃ *P*. *P* ∈ *carrier-mat CARD*('*m*) *CARD*('*m*) ∧
invertible-mat P ∧ *Hermite-of-list-of-rows A* [*0*..<*CARD*('*m*)] = *P* ∗ *A*
using *assms invertible-Hermite-of-list-of-rows-mod-type* **by** (*smt* (*verit*) *CARD-mod-ring*)

lemmas *Hermite-Hermite-of-list-of-rows-nontriv-mod-ring-internalized* =
Hermite-Hermite-of-list-of-rows-nontriv-mod-ring[*unfolded CARD-mod-ring*,
internalize-sort '*m*::*nontriv*, *internalize-sort* '*b*::*nontriv*]

lemmas *invertible-Hermite-of-list-of-rows-nontriv-mod-ring-internalized* =
invertible-Hermite-of-list-of-rows-nontriv-mod-ring[*unfolded CARD-mod-ring*,
internalize-sort '*m*::*nontriv*, *internalize-sort* '*b*::*nontriv*]

context
fixes *m*::*nat* **and** *n*::*nat*
assumes *local-typedef1*: ∃ (*Rep* :: ('*b* ⇒ *int*)) *Abs*. *type-definition Rep Abs* {*0*..<*m*
:: *int*}
assumes *local-typedef2*: ∃ (*Rep* :: ('*c* ⇒ *int*)) *Abs*. *type-definition Rep Abs* {*0*..<*n*
:: *int*}
and *m*: *m*>*1*
and *n*: *n*>*1*
begin

lemma *Hermite-Hermite-of-list-of-rows-nontriv-mod-ring-aux*:
fixes *A*::*int mat*
assumes *A* ∈ *carrier-mat m n*
assumes *eA*: *echelon-form-JNF A*
shows *Hermite-JNF* (*range ass-function-euclidean*)
(λ*c*. *range* (*res-int c*)) (*Hermite-of-list-of-rows A* [*0*..<*m*])
using *Hermite-Hermite-of-list-of-rows-nontriv-mod-ring-internalized*
[*OF type-to-set2*(*1*)[*OF local-typedef1 local-typedef2*]
type-to-set1(*1*)[*OF local-typedef1 local-typedef2*]]

using *assms*
using *type-to-set1*(*2*) *local-typedef1 local-typedef2 n m* **by** *metis*

lemma *invertible-Hermite-of-list-of-rows-nontriv-mod-ring-aux*:
fixes *A*::*int mat*
assumes *A* ∈ *carrier-mat m n*
assumes *eA*: *echelon-form-JNF A*
shows ∃ *P*. *P* ∈ *carrier-mat m m* ∧ *invertible-mat P* ∧ *Hermite-of-list-of-rows A* [*0*..<*m*] = *P* ∗ *A*
using *invertible-Hermite-of-list-of-rows-nontriv-mod-ring-internalized*
[*OF type-to-set2*(*1*)[*OF local-typedef1 local-typedef2*]
type-to-set1(*1*)[*OF local-typedef1 local-typedef2*]]
using *assms*
using *type-to-set1*(*2*) *local-typedef1 local-typedef2 n m* **by** *metis*
end

context
begin

private lemma *invertible-Hermite-of-list-of-rows-cancelled-first*:
∃ *Rep Abs*. *type-definition Rep Abs* {*0*..<*int n*}
⟹ *1* < *m* ⟹ *1* < *n* ⟹ *A* ∈ *carrier-mat m n* ⟹ *echelon-form-JNF A*
⟹ ∃ *P*. *P* ∈ *carrier-mat m m* ∧ *invertible-mat P* ∧ *Hermite-of-list-of-rows A* [*0*..<*m*] = *P* ∗ *A*
using *invertible-Hermite-of-list-of-rows-nontriv-mod-ring-aux*[*cancel-type-definition*, *of m n A*]
by *auto*

private lemma *invertible-Hermite-of-list-of-rows-cancelled-both*:
1 < *m* ⟹ *1* < *n* ⟹ *A* ∈ *carrier-mat m n* ⟹ *echelon-form-JNF A*
⟹ ∃ *P*. *P* ∈ *carrier-mat m m* ∧ *invertible-mat P* ∧ *Hermite-of-list-of-rows A* [*0*..<*m*] = *P* ∗ *A*
using *invertible-Hermite-of-list-of-rows-cancelled-first*[*cancel-type-definition*, *of n m A*] **by** *simp*

private lemma *Hermite-Hermite-of-list-of-rows-cancelled-first*:
∃ *Rep Abs*. *type-definition Rep Abs* {*0*..<*int n*} ⟹
1 < *m* ⟹
1 < *n* ⟹
A ∈ *carrier-mat m n* ⟹

echelon-form-JNF A
⟹ *Hermite-JNF* (*range ass-function-euclidean*) (λ*c*. *range* (*res-int c*)) (*Hermite-of-list-of-rows A* [*0*..<*m*])
using *Hermite-Hermite-of-list-of-rows-nontriv-mod-ring-aux*[*cancel-type-definition*, *of m n A*]
by *auto*

private lemma *Hermite-Hermite-of-list-of-rows-cancelled-both*:
1 < *m* ⟹
1 < *n* ⟹
A ∈ *carrier-mat m n* ⟹
echelon-form-JNF A
⟹ *Hermite-JNF* (*range ass-function-euclidean*) (λ*c*. *range* (*res-int c*)) (*Hermite-of-list-of-rows A* [*0*..<*m*])
using *Hermite-Hermite-of-list-of-rows-cancelled-first*[*cancel-type-definition*, *of n m A*] **by** *simp*

lemma *Hermite-Hermite-of-list-of-rows'*:
fixes *A*::*int mat*
assumes *A* ∈ *carrier-mat m n*
and *echelon-form-JNF A*
and *1* < *m* **and** *1* < *n*
shows *Hermite-JNF* (*range ass-function-euclidean*)
(λ*c*. *range* (*res-int c*)) (*Hermite-of-list-of-rows A* [*0*..<*m*])
using *Hermite-Hermite-of-list-of-rows-cancelled-both assms* **by** *auto*

corollary *Hermite-Hermite-of-list-of-rows*:
fixes *A*::*int mat*
assumes *A*: *A* ∈ *carrier-mat m n*
and *eA*: *echelon-form-JNF A*
shows *Hermite-JNF* (*range ass-function-euclidean*)
(λ*c*. *range* (*res-int c*)) (*Hermite-of-list-of-rows A* [*0*..<*m*])
proof (*cases m=0* ∨ *n=0*)
case *True*
then show *?thesis*
by (*auto*, *metis Hermite-Hermite-of-row-i Hermite-JNF-def A eA carrier-matD*(*1*) *one-carrier-mat zero-order*(*3*))
(*metis Hermite-Hermite-of-row-i Hermite-JNF-def Hermite-of-list-of-rows A carrier-matD*(*2*)
echelon-form-mx0 is-zero-row-JNF-def mat-carrier zero-order(*3*))
next
case *False* **note** *not-m0-or-n0* = *False*
show *?thesis*
proof (*cases m=1* ∨ *n=1*)
case *True*

```
    then show ?thesis
      by (metis False Hermite-of-list-of-rows-1xn Hermite-of-list-of-rows-mx1 A eA

          atLeastLessThan-iff linorder-not-less neq0-conv set-upt upt-eq-Nil-conv)
  next
    case False
    show ?thesis
     by (rule Hermite-Hermite-of-list-of-rows'[OF A eA], insert not-m0-or-n0 False,
auto)
  qed
qed

lemma invertible-Hermite-of-list-of-rows:
  assumes A: A ∈ carrier-mat m n
  and eA: echelon-form-JNF A
shows ∃ P. P ∈ carrier-mat m m ∧ invertible-mat P ∧ Hermite-of-list-of-rows A
[0..<m] = P * A
proof (cases m=0 ∨ n=0)
  case True
  have *: Hermite-of-list-of-rows A [0..<m] = A if n: n=0
    by (rule Hermite-of-list-of-rows-mx0, insert A n, auto)
  show ?thesis using True
   by (auto, metis assms(1) invertible-mat-one left-mult-one-mat one-carrier-mat)
    (metis (full-types) * assms(1) invertible-mat-one left-mult-one-mat one-carrier-mat)
next
  case False note mn = False
  show ?thesis
  proof (cases m=1 ∨ n=1)
    case True
    then show ?thesis
    using A eA invertible-Hermite-of-list-of-rows-1xn invertible-Hermite-of-list-of-rows-mx1
by blast
  next
    case False
    then show ?thesis
      using invertible-Hermite-of-list-of-rows-cancelled-both[OF - - A eA] False mn
by auto
  qed
qed
end
end
end
end
```

Now we have all the required stuff to prove the soundness of the algorithm.

```
context proper-mod-operation
begin
```

lemma *Hermite-mod-det-mx0*:
assumes $A \in$ *carrier-mat m 0*
shows *Hermite-mod-det abs-flag A* $= A$
unfolding *Hermite-mod-det-def Let-def* **using** *assms* **by** *auto*

lemma *Hermite-JNF-mx0*:
assumes *A*: $A \in$ *carrier-mat m 0*
shows *Hermite-JNF* (*range ass-function-euclidean*) (λ*c. range* (*res-int c*)) *A*
unfolding *Hermite-JNF-def* **using** *A echelon-form-mx0* **unfolding** *is-zero-row-JNF-def*

using *ass-function-Complete-set-non-associates*[*OF ass-function-euclidean*]
using *res-function-Complete-set-residues*[*OF res-function-res-int*] **by** *auto*

lemma *Hermite-mod-det-soundness-mx0*:
assumes *A*: $A \in$ *carrier-mat m n*
and *n0*: n=*0*
shows *Hermite-JNF* (*range ass-function-euclidean*) (λ*c. range* (*res-int c*)) (*Hermite-mod-det abs-flag A*)
and ($\exists$ *P*. *invertible-mat P* $\wedge$ $P \in$ *carrier-mat m m* $\wedge$ (*Hermite-mod-det abs-flag A*) $= P *$ *A*)
proof −
have *A*: $A \in$ *carrier-mat m 0* **using** *A n0* **by** *blast*
then show *Hermite-JNF* (*range ass-function-euclidean*) (λ*c. range* (*res-int c*)) (*Hermite-mod-det abs-flag A*)
using *Hermite-JNF-mx0*[*OF A*] *Hermite-mod-det-mx0*[*OF A*] **by** *auto*
show ($\exists$ *P*. *invertible-mat P* $\wedge$ $P \in$ *carrier-mat m m* $\wedge$ (*Hermite-mod-det abs-flag A*) $= P *$ *A*)
by (*metis A Hermite-mod-det-mx0 invertible-mat-one left-mult-one-mat one-carrier-mat*)
qed

lemma *Hermite-mod-det-soundness-mxn*:
assumes *mn*: $m = n$
and *A*: $A \in$ *carrier-mat m n*
and *n0*: 0<n
and *inv-RAT-A*: *invertible-mat* (*map-mat rat-of-int A*)
shows *Hermite-JNF* (*range ass-function-euclidean*) (λ*c. range* (*res-int c*)) (*Hermite-mod-det abs-flag A*)
and ($\exists$ *P*. *invertible-mat P* $\wedge$ $P \in$ *carrier-mat m m* $\wedge$ (*Hermite-mod-det abs-flag A*) $= P *$ *A*)
proof −
define *D A′ E H H′* **where** *D-def*: $D = |$*Determinant.det A*$|$
and *A′-def*: $A' = A$ @$_r$ D ·$_m$ 1_m n **and** *E-def*: $E =$ *FindPreHNF abs-flag D A′*
and *H-def*: $H =$ *Hermite-of-list-of-rows E* [*0*..<*m*+*n*]
and *H′-def*: $H' =$ *mat-of-rows n* (*map* (*Matrix.row H*) [*0*..<*m*])
have *A′*: $A' \in$ *carrier-mat* (m+n) n **using** *A A A′-def* **by** *auto*
let *?RAT* = *of-int-hom.mat-hom* :: *int mat* ⇒ *rat mat*

```
  have RAT-A: ?RAT A ∈ carrier-mat n n
    using A map-carrier-mat mat-of-rows-carrier(1) mn by auto
  have det-RAT-fs-init: det (?RAT A) ≠ 0
    using inv-RAT-A unfolding invertible-iff-is-unit-JNF[OF RAT-A] by auto
  moreover have mat-of-rows n (map (Matrix.row A') [0..<n]) = A
  proof
    let ?A' = mat-of-rows n (map (Matrix.row A') [0..<n])
    show dr: dim-row ?A' = dim-row A and dc: dim-col ?A' = dim-col A using
A mn by auto
    fix i j assume i: i < dim-row A and j: j < dim-col A
    have D: D ·_m 1_m n ∈ carrier-mat n n using mn by auto
    have ?A' $$ (i,j) =  (map (Matrix.row A') [0..<n]) ! i $v j
      by (rule mat-of-rows-index, insert i j dr dc A, auto)
    also have ... = A' $$ (i,j) using A' mn i j A by auto
   also have ... = A $$ (i,j) unfolding A'-def using i append-rows-nth[OF A D]
mn j A by auto
    finally show ?A' $$ (i, j) = A $$ (i, j) .
  qed
  ultimately have inv-RAT-A'n:
   invertible-mat (map-mat rat-of-int (mat-of-rows n (map (Matrix.row A') [0..<n])))

    using inv-RAT-A by auto
  have eE: echelon-form-JNF E
    by (unfold E-def, rule FindPreHNF-echelon-form[OF A'-def A - -],
        insert mn D-def det-RAT-fs-init, auto)
  have E: E ∈ carrier-mat (m+n) n unfolding E-def by (rule FindPreHNF[OF
A'])
  have ∃ P. P ∈ carrier-mat (m + n) (m + n) ∧ invertible-mat P ∧ E = P * A'
  by (unfold E-def, rule FindPreHNF-invertible-mat[OF A'-def A n0 - -],
      insert mn D-def det-RAT-fs-init, auto)
  from this obtain P where P: P ∈ carrier-mat (m + n) (m + n)
    and inv-P: invertible-mat P and E-PA': E = P * A'
    by blast
  have ∃ Q. Q ∈ carrier-mat (m+n) (m+n) ∧ invertible-mat Q ∧ H = Q * E
    by (unfold H-def, rule invertible-Hermite-of-list-of-rows[OF E eE])
  from this obtain Q where Q: Q ∈ carrier-mat (m+n) (m+n)
    and inv-Q: invertible-mat Q and H-QE: H = Q * E by blast
  let ?ass =(range ass-function-euclidean)
  let ?res = (λc. range (res-int c))
 have Hermite-H: Hermite-JNF (range ass-function-euclidean) (λc. range (res-int
c)) H
    by (unfold H-def, rule Hermite-Hermite-of-list-of-rows[OF E eE])
  hence eH: echelon-form-JNF H unfolding Hermite-JNF-def by auto
  have H': H' ∈ carrier-mat m n using H'-def by auto
  have H-H'0: H = H' @_r 0_m m n
  proof (unfold H'-def, rule upper-triangular-append-zero)
   show upper-triangular' H using eH by (rule echelon-form-JNF-imp-upper-triangular)
    show H ∈ carrier-mat (m + m) n
      unfolding H-def using Hermite-of-list-of-rows[of E] E mn by auto
```

qed (*insert mn*, *simp*)
obtain P' **where** *PP'*: *inverts-mat P P'*
and *P'P*: *inverts-mat P' P* **and** *P'*: $P' \in$ *carrier-mat* $(m+n)$ $(m+n)$
using *P inv-P obtain-inverse-matrix* **by** *blast*
obtain Q' **where** *QQ'*: *inverts-mat Q Q'*
and *Q'Q*: *inverts-mat Q' Q* **and** *Q'*: $Q' \in$ *carrier-mat* $(m+n)$ $(m+n)$
using *Q inv-Q obtain-inverse-matrix* **by** *blast*
have *P'Q'*: $(P'*Q') \in$ *carrier-mat* $(m + m)$ $(m + m)$ **using** *P' Q' mn* **by** *simp*
have *A'-P'Q'H*: $A' = P' * Q' * H$
proof −
have *QP*: $Q * P \in$ *carrier-mat* $(m + m)$ $(m + m)$ **using** *Q P mn* **by** *auto*
have $H = Q * (P * A')$ **using** *H-QE E-PA'* **by** *auto*
also have $... = (Q * P) * A'$ **using** *A' P Q* **by** *auto*
also have $(P' * Q') * ... = ((P' * Q') * (Q * P)) * A'$ **using** *A' P'Q' QP mn*
by *auto*
also have $... = \ (P' * (Q' * Q) * P) * A'$
by (*smt* (*verit*) *P P' P'Q' Q Q' assms*(*1*) *assoc-mult-mat*)
also have $... = (P'*P) * A'$
by (*metis P' Q' Q'Q carrier-matD*(*1*) *inverts-mat-def right-mult-one-mat*)
also have $... = A'$
by (*metis A' P' P'P carrier-matD*(*1*) *inverts-mat-def left-mult-one-mat*)
finally show $A' = P' * Q' * H$ **..**
qed
have *inv-P'Q'*: *invertible-mat* $(P' * Q')$
by (*metis P' P'P PP' Q' Q'Q QQ' carrier-matD*(*1*) *carrier-matD*(*2*) *invertible-mat-def*
invertible-mult-JNF square-mat.simps)
interpret *vec-module TYPE*(*int*) **.**
interpret *B*: *cof-vec-space n TYPE*(*rat*) **.**
interpret *A*: *LLL-with-assms n m* (*Matrix.rows A*) *4/3*
proof
show *length* (*rows A*) $= m$ **using** *A* **unfolding** *Matrix.rows-def* **by** *simp*
have *s*: *set* (*map of-int-hom.vec-hom* (*rows A*)) $\subseteq$ *carrier-vec n*
using *A* **unfolding** *Matrix.rows-def* **by** *auto*
have *rw*: (*map of-int-hom.vec-hom* (*rows A*)) = (*rows* (*?RAT A*))
by (*metis A s carrier-matD*(*2*) *mat-of-rows-map mat-of-rows-rows rows-mat-of-rows set-rows-carrier subsetI*)
have *B.lin-indpt* (*set* (*map of-int-hom.vec-hom* (*rows A*)))
unfolding *rw* **by** (*rule B.det-not-0-imp-lin-indpt-rows*[*OF RAT-A det-RAT-fs-init*])
moreover have *distinct* (*map of-int-hom.vec-hom* (*rows A*)::*rat Matrix.vec list*)
proof (*rule ccontr*)
assume ¬ *distinct* (*map of-int-hom.vec-hom* (*rows A*)::*rat Matrix.vec list*)
from *this* **obtain** *i j* **where** *row* (*?RAT A*) *i* = *row* (*?RAT A*) *j* **and** $i \neq j$
and $i < n$ **and** $j < n$
unfolding *rw*
by (*metis Determinant.det-transpose RAT-A add-0 cols-transpose det-RAT-fs-init*
not-add-less2 transpose-carrier-mat vec-space.det-rank-iff vec-space.non-distinct-low-rank)
thus *False* **using** *Determinant.det-identical-rows*[*OF RAT-A*] **using** *det-RAT-fs-init*

RAT-A **by** *auto*
qed
ultimately show *B.lin-indpt-list* (*map of-int-hom.vec-hom* (*rows A*))
using *s* **unfolding** *B.lin-indpt-list-def* **by** *auto*
qed (*simp*)
have *A-eq*: *mat-of-rows n* (*Matrix.rows A*) = *A* **using** *A mat-of-rows-rows* **by** *blast*
have *D-A*: *D* = |*det* (*mat-of-rows n* (*rows A*))| **using** *D-def A-eq* **by** *auto*
have *Hermite-H'*: *Hermite-JNF ?ass ?res H'*
by (*rule A.Hermite-append-det-id*(*1*)[*OF - mn - H' H-H'0 P'Q' inv-P'Q' A'-P'Q'H Hermite-H*],
insert D-def A'-def mn A inv-RAT-A D-A A-eq, *auto*)
have *dc*: *dim-row A* = *m* **and** *dr*: *dim-col A* = *n* **using** *A* **by** *auto*
have *Hermite-mod-det-H'*: *Hermite-mod-det abs-flag A* = *H'*
unfolding *Hermite-mod-det-def Let-def H'-def H-def E-def A'-def D-def dc dr det-int* **by** *blast*
show *Hermite-JNF ?ass ?res* (*Hermite-mod-det abs-flag A*) **using** *Hermite-mod-det-H' Hermite-H'* **by** *simp*
have ∃ *R*. *invertible-mat R* ∧ *R* ∈ *carrier-mat m m* ∧ *A* = *R* ∗ *H'*
by (*subst A-eq*[*symmetric*],
rule A.Hermite-append-det-id(*2*)[*OF - mn - H' H-H'0 P'Q' inv-P'Q' A'-P'Q'H Hermite-H*],
insert D-def A'-def mn A inv-RAT-A D-A A-eq, *auto*)
from *this* **obtain** *R* **where** *inv-R*: *invertible-mat R*
and *R*: *R* ∈ *carrier-mat m m* **and** *A-RH'*: *A* = *R* ∗ *H'*
by *blast*
obtain *R'* **where** *inverts-R*: *inverts-mat R R'* **and** *R'*: *R'* ∈ *carrier-mat m m*
by (*meson R inv-R obtain-inverse-matrix*)
have *inv-R'*: *invertible-mat R'* **using** *inverts-R* **unfolding** *invertible-mat-def inverts-mat-def*
using *R R' mat-mult-left-right-inverse* **by** *auto*
moreover have *H'* = *R'* ∗ *A*
proof −
have *R'* ∗ *A* = *R'* ∗ (*R* ∗ *H'*) **using** *A-RH'* **by** *auto*
also have ... = (*R'*∗*R*) ∗ *H'* **using** *H' R R'* **by** *auto*
also have ... = *H'*
by (*metis H' R R' mat-mult-left-right-inverse carrier-matD*(*1*)
inverts-R inverts-mat-def left-mult-one-mat)
finally show *?thesis* **..**
qed
ultimately show ∃ *S*. *invertible-mat S* ∧ *S* ∈ *carrier-mat m m* ∧ *Hermite-mod-det abs-flag A* = *S* ∗ *A*
using *R' Hermite-mod-det-H'* **by** *blast*
qed

lemma *Hermite-mod-det-soundness*:
assumes *mn*: *m* = *n*
and *A-def*: *A* ∈ *carrier-mat m n*

and *i*: *invertible-mat* (*map-mat rat-of-int A*)
shows *Hermite-JNF* (*range ass-function-euclidean*) (λ*c*. *range* (*res-int c*)) (*Hermite-mod-det abs-flag A*)
and (∃ *P*. *invertible-mat P* ∧ *P* ∈ *carrier-mat m m* ∧ (*Hermite-mod-det abs-flag A*) = *P* ∗ *A*)
using *A-def Hermite-mod-det-soundness-mx0*(*1*) *Hermite-mod-det-soundness-mxn*(*1*) *mn i*
by *blast* (*insert Hermite-mod-det-soundness-mx0*(*2*) *Hermite-mod-det-soundness-mxn*(*2*) *assms*, *blast*)

We can even move the whole echelon form algorithm *echelon-form-of* from HOL Analysis to JNF and then we can combine it with *Hermite-of-list-of-rows* to have another HNF algorithm which is not efficient, but valid for arbitrary matrices.

lemma *reduce-D0*:
reduce a b 0 A = (*let Aaj* = *A*$$(*a*,*0*); *Abj* = *A* $$ (*b*,*0*)
in
if Aaj = *0 then A else*
case euclid-ext2 Aaj Abj of (*p*,*q*,*u*,*v*,*d*) ⇒
Matrix.mat (*dim-row A*) (*dim-col A*)
(λ(*i*,*k*). *if i* = *a then* (*p*∗*A*$$(*a*,*k*) + *q*∗*A*$$(*b*,*k*))
else if i = *b then u* ∗ *A*$$(*a*,*k*) + *v* ∗ *A*$$(*b*,*k*)
else A$$(*i*,*k*)
)
) (**is** *?lhs* = *?rhs*)
proof
obtain *p q u v d* **where** *pquvd*: (*p*,*q*,*u*,*v*,*d*) = *euclid-ext2* (*A* $$ (*a*, *0*)) (*A* $$ (*b*, *0*))
by (*simp add*: *euclid-ext2-def*)
have ∗: *Matrix.mat* (*dim-row A*) (*dim-col A*)
(λ(*i*, *k*).
if i = *a then let r* = *p* ∗ *A* $$ (*a*, *k*) + *q* ∗ *A* $$ (*b*, *k*) *in if 0* < |*r*| *then*
if k = *0* ∧ *0 dvd r then 0 else r mod 0 else r*
else if i = *b then let r* = *u* ∗ *A* $$ (*a*, *k*) + *v* ∗ *A* $$ (*b*, *k*) *in*
if 0 < |*r*| *then r mod 0 else r else A* $$ (*i*, *k*))
= *Matrix.mat* (*dim-row A*) (*dim-col A*)
(λ(*i*,*k*). *if i* = *a then* (*p*∗*A*$$(*a*,*k*) + *q*∗*A*$$(*b*,*k*))
else if i = *b then u* ∗ *A*$$(*a*,*k*) + *v* ∗ *A*$$(*b*,*k*)
else A$$(*i*,*k*)
)
by (*rule eq-matI*, *auto simp add*: *Let-def*)
show *dim-row ?lhs* = *dim-row ?rhs*
unfolding *reduce.simps Let-def* **by** (*smt* (*verit*) *dim-row-mat*(*1*) *pquvd prod.simps*(*2*))
show *dim-col ?lhs* = *dim-col ?rhs*
unfolding *reduce.simps Let-def* **by** (*smt* (*verit*) *dim-col-mat*(*1*) *pquvd prod.simps*(*2*))
fix *i j* **assume** *i*: *i*<*dim-row ?rhs* **and** *j*: *j*<*dim-col ?rhs*
show *?lhs* $$ (*i*,*j*) = *?rhs* $$ (*i*,*j*)
by (*cases A* $$ (*a*, *0*) = *0*, *insert* ∗ *pquvd i j*, *auto simp add*: *case-prod-beta Let-def*)

qed

lemma *bezout-matrix-JNF-mult-eq'*:
assumes *A'*: $A' \in$ *carrier-mat m n* **and** *a*: $a<m$ **and** *b*: $b<m$ **and** *ab*: $a \neq b$
and *A-def*: $A = A' \mathbin{@_r} B$ **and** *B*: $B \in$ *carrier-mat t n*
assumes *pquvd*: $(p,q,u,v,d) =$ *euclid-ext2* (*A$$(a,j)*) (*A$$(b,j)*)
shows *Matrix.mat* (*dim-row A*) (*dim-col A*)
(λ(*i*,*k*). *if i = a then* (*p*∗*A$$(a,k)* + *q*∗*A$$(b,k)*)
else if i = b then u ∗ *A$$(a,k)* + *v* ∗ *A$$(b,k)*
else A$$(i,k)
) = (*bezout-matrix-JNF A a b j euclid-ext2*) ∗ *A* (**is** *?A = ?BM* ∗ *A*)
proof (*rule eq-matI*)
have *A*: $A \in$ *carrier-mat* (*m*+*t*) *n* **using** *A-def A' B* **by** *simp*
hence *A-carrier*: *?A* ∈ *carrier-mat* (*m*+*t*) *n* **by** *auto*
show *dr*: *dim-row ?A = dim-row* (*?BM* ∗ *A*) **and** *dc*: *dim-col ?A = dim-col* (*?BM* ∗ *A*)
unfolding *bezout-matrix-JNF-def* **by** *auto*
fix *i ja* **assume** *i*: *i* < *dim-row* (*?BM* ∗ *A*) **and** *ja*: *ja* < *dim-col* (*?BM* ∗ *A*)
let *?f* = λ*ia*. (*bezout-matrix-JNF A a b j euclid-ext2*) $$ (*i*,*ia*) ∗ *A* $$ (*ia*,*ja*)
have *dv*: *dim-vec* (*col A ja*) = *m*+*t* **using** *A* **by** *auto*
have *i-dr*: *i*<*dim-row A* **using** *i A* **unfolding** *bezout-matrix-JNF-def* **by** *auto*
have *a-dr*: *a*<*dim-row A* **using** *A a ja* **by** *auto*
have *b-dr*: *b*<*dim-row A* **using** *A b ja* **by** *auto*
show *?A* $$ (*i*,*ja*) = (*?BM* ∗ *A*) $$ (*i*,*ja*)
proof −
have (*?BM* ∗ *A*) $$ (*i*,*ja*) = *Matrix.row ?BM i* · *col A ja*
by (*rule index-mult-mat*, *insert i ja*, *auto*)
also have ... = (∑ *ia* = *0*..<*dim-vec* (*col A ja*).
Matrix.row (*bezout-matrix-JNF A a b j euclid-ext2*) *i* $*v ia* ∗ *col A ja* $*v ia*)
by (*simp add*: *scalar-prod-def*)
also have ... = (∑ *ia* = *0*..<*m*+*t*. *?f ia*)
by (*rule sum.cong*, *insert A i dr dc*, *auto*) (*smt* (*verit*) *bezout-matrix-JNF-def carrier-matD*(*1*)
dim-col-mat(*1*) *index-col index-mult-mat*(*3*) *index-row*(*1*) *ja*)
also have ... = (∑ *ia* ∈ ({*a*,*b*} ∪ ({*0*..<*m*+*t*} − {*a*,*b*})). *?f ia*)
by (*rule sum.cong*, *insert a a-dr b A ja*, *auto*)
also have ... = *sum ?f* {*a*,*b*} + *sum ?f* ({*0*..<*m*+*t*} − {*a*,*b*})
by (*rule sum.union-disjoint*, *auto*)
finally have *BM-A-ija-eq*: (*?BM* ∗ *A*) $$ (*i*,*ja*) = *sum ?f* {*a*,*b*} + *sum ?f* ({*0*..<*m*+*t*} − {*a*,*b*}) **by** *auto*
show *?thesis*
proof (*cases i = a*)
case *True*
have *sum0*: *sum ?f* ({*0*..<*m*+*t*} − {*a*,*b*}) = *0*
proof (*rule sum.neutral*, *rule*)
fix *x* **assume** *x*: *x* ∈ {*0*..<*m* + *t*} − {*a*, *b*}

hence *xm*: $x < m+t$ **by** *auto*
have *x-not-i*: $x \neq i$ **using** *True x* **by** *blast*
have *x-dr*: $x <$ *dim-row A* **using** *x A* **by** *auto*
have *bezout-matrix-JNF A a b j euclid-ext2* $$ $(i, x) = 0$
unfolding *bezout-matrix-JNF-def*
unfolding *index-mat(1)*[*OF i-dr x-dr*] **using** *x-not-i x* **by** *auto*
thus *bezout-matrix-JNF A a b j euclid-ext2* $$ $(i, x) * A$ $$ $(x, ja) = 0$ **by** *auto*
qed
have *fa*: *bezout-matrix-JNF A a b j euclid-ext2* $$ $(i, a) = p$
unfolding *bezout-matrix-JNF-def index-mat(1)*[*OF i-dr a-dr*] **using** *True pquvd*
by (*auto*, *metis split-conv*)
have *fb*: *bezout-matrix-JNF A a b j euclid-ext2* $$ $(i, b) = q$
unfolding *bezout-matrix-JNF-def index-mat(1)*[*OF i-dr b-dr*] **using** *True pquvd ab*
by (*auto*, *metis split-conv*)
have *sum ?f* $\{a,b\}$ + *sum ?f* $(\{0..<m+t\} - \{a,b\})$ = *?f a* + *?f b* **using** *sum0* **by** (*simp add*: *ab*)
also have ... = $p * A$ $$ $(a, ja) + q * A$ $$ (b, ja) **unfolding** *fa fb* **by** *simp*
also have ... = *?A* $$ (i,ja) **using** *A True dr i ja* **by** *auto*
finally show *?thesis* **using** *BM-A-ija-eq* **by** *simp*
next
case *False* **note** *i-not-a* = *False*
show *?thesis*
proof (*cases i=b*)
case *True*
have *sum0*: *sum ?f* $(\{0..<m+t\} - \{a,b\}) = 0$
proof (*rule sum.neutral*, *rule*)
fix *x* **assume** *x*: $x \in \{0..<m + t\} - \{a, b\}$
hence *xm*: $x < m+t$ **by** *auto*
have *x-not-i*: $x \neq i$ **using** *True x* **by** *blast*
have *x-dr*: $x <$ *dim-row A* **using** *x A* **by** *auto*
have *bezout-matrix-JNF A a b j euclid-ext2* $$ $(i, x) = 0$
unfolding *bezout-matrix-JNF-def*
unfolding *index-mat(1)*[*OF i-dr x-dr*] **using** *x-not-i x* **by** *auto*
thus *bezout-matrix-JNF A a b j euclid-ext2* $$ $(i, x) * A$ $$ $(x, ja) = 0$ **by** *auto*
qed
have *fa*: *bezout-matrix-JNF A a b j euclid-ext2* $$ $(i, a) = u$
unfolding *bezout-matrix-JNF-def index-mat(1)*[*OF i-dr a-dr*] **using** *True i-not-a pquvd*
by (*auto*, *metis split-conv*)
have *fb*: *bezout-matrix-JNF A a b j euclid-ext2* $$ $(i, b) = v$
unfolding *bezout-matrix-JNF-def index-mat(1)*[*OF i-dr b-dr*] **using** *True i-not-a pquvd ab*
by (*auto*, *metis split-conv*)
have *sum ?f* $\{a,b\}$ + *sum ?f* $(\{0..<m+t\} - \{a,b\})$ = *?f a* + *?f b* **using** *sum0* **by** (*simp add*: *ab*)

```
      also have ... = u * A $$ (a, ja) + v * A $$ (b, ja) unfolding fa fb by simp
       also have ... = ?A $$ (i,ja) using A True i-not-a dr i ja by auto
       finally show ?thesis using BM-A-ija-eq by simp
     next
       case False note i-not-b = False
       have sum0: sum ?f ({0..<m+t} - {a,b} - {i}) = 0
       proof (rule sum.neutral, rule)
         fix x assume x: x ∈ {0..<m + t} - {a, b} - {i}
         hence xm: x < m+t by auto
         have x-not-i: x ≠ i using x by blast
         have x-dr: x < dim-row A using x A by auto
         have bezout-matrix-JNF A a b j euclid-ext2 $$ (i, x) = 0
           unfolding bezout-matrix-JNF-def
           unfolding index-mat(1)[OF i-dr x-dr] using x-not-i x by auto
          thus bezout-matrix-JNF A a b j euclid-ext2 $$ (i, x) * A $$ (x, ja) = 0
by auto
       qed
       have fa: bezout-matrix-JNF A a b j euclid-ext2 $$ (i, a) = 0
        unfolding bezout-matrix-JNF-def index-mat(1)[OF i-dr a-dr] using False
i-not-a pquvd
         by auto
       have fb: bezout-matrix-JNF A a b j euclid-ext2 $$ (i, b) = 0
        unfolding bezout-matrix-JNF-def index-mat(1)[OF i-dr b-dr] using False
i-not-a pquvd
         by auto
      have sum ?f ({0..<m+t} - {a,b}) = sum ?f (insert i ({0..<m+t} - {a,b}
- {i}))
         by (rule sum.cong, insert i-dr A i-not-a i-not-b, auto)
          also have ... = ?f i + sum ?f ({0..<m+t} - {a,b} - {i}) by (rule
sum.insert, auto)
       also have ... = ?f i using sum0 by simp
       also have ... = ?A $$ (i,ja)
           unfolding bezout-matrix-JNF-def using i-not-a i-not-b  A dr i ja by
fastforce
       finally show ?thesis unfolding BM-A-ija-eq by (simp add: ab fa fb)
     qed
   qed
  qed
qed



lemma bezout-matrix-JNF-mult-eq2:
  assumes A: A ∈ carrier-mat m n and a: a<m  and b: b<m and ab: a ≠ b
  assumes pquvd: (p,q,u,v,d) = euclid-ext2 (A$$(a,j)) (A$$(b,j))
  shows Matrix.mat (dim-row A) (dim-col A)
        (λ(i,k). if i = a then (p*A$$(a,k) + q*A$$(b,k))
                 else if i = b then u * A$$(a,k) + v * A$$(b,k)
                 else A$$(i,k)
```

```
        ) = (bezout-matrix-JNF A a b j euclid-ext2) * A (is ?A = ?BM * A)
proof (rule bezout-matrix-JNF-mult-eq'[OF A a b ab - - pquvd])
  show A = A @_r (0_m 0 n) by (rule eq-matI, unfold append-rows-def, auto)
  show (0_m 0 n) ∈ carrier-mat 0 n by auto
qed


lemma reduce-invertible-mat-D0-BM:
  assumes A: A ∈ carrier-mat m n
  and a: a < m
  and b: b < m
  and ab: a ≠ b
  and Aa0: A$$(a,0) ≠ 0
  shows reduce a b 0 A = (bezout-matrix-JNF A a b 0 euclid-ext2) * A
proof −
 obtain p q u v d where pquvd: (p,q,u,v,d) = euclid-ext2 (A$$(a,0)) (A$$(b,0))
    by (simp add: euclid-ext2-def)
  let ?BM = bezout-matrix-JNF A a b 0 euclid-ext2
  let ?A = Matrix.mat (dim-row A) (dim-col A)
        (λ(i,k). if i = a then  (p*A$$(a,k) + q*A$$(b,k))
                 else if i = b then  u * A$$(a,k) + v * A$$(b,k)
                 else A$$(i,k))
  have A'-BZ-A: ?A = ?BM * A
    by (rule bezout-matrix-JNF-mult-eq2[OF A - - ab pquvd], insert a b, auto)
  moreover have ?A = reduce a b 0 A using pquvd Aa0 unfolding reduce-D0
Let-def
    by (metis (no-types, lifting) split-conv)
  ultimately show ?thesis by simp
qed


lemma reduce-invertible-mat-D0:
  assumes A: A ∈ carrier-mat m n
  and a: a < m
  and b: b < m
  and n0: 0<n
  and ab: a ≠ b
  and a-less-b: a<b
  shows ∃ P. invertible-mat P ∧ P ∈ carrier-mat m m ∧ reduce a b 0 A = P * A
proof (cases A$$(a,0) = 0)
  case True
  then show ?thesis
     by (smt (verit) A invertible-mat-one left-mult-one-mat one-carrier-mat re-
duce.simps)
next
  case False
  obtain p q u v d where pquvd: (p,q,u,v,d) = euclid-ext2 (A$$(a,0)) (A$$(b,0))
    by (simp add: euclid-ext2-def)
  let ?BM = bezout-matrix-JNF A a b 0 euclid-ext2
```

have *reduce a b 0 A = ?BM ∗ A* **by** (*rule reduce-invertible-mat-D0-BM*[*OF A a b ab False*])
moreover have *invertible-bezout*: *invertible-mat ?BM*
by (*rule invertible-bezout-matrix-JNF*[*OF A is-bezout-ext-euclid-ext2 a-less-b - n0 False*],
insert a-less-b b, *auto*)
moreover have *BM*: *?BM ∈ carrier-mat m m* **unfolding** *bezout-matrix-JNF-def* **using** *A* **by** *auto*
ultimately show *?thesis* **by** *blast*
qed

lemma *reduce-below-invertible-mat-D0*:
assumes *A'*: *A ∈ carrier-mat m n* **and** *a*: *a<m* **and** *j*: *0<n*
and *distinct xs* **and** ∀ *x ∈ set xs*. *x < m ∧ a < x*
and *D=0*
shows (∃ *P*. *invertible-mat P ∧ P ∈ carrier-mat m m ∧ reduce-below a xs D A = P ∗ A*)
using *assms*
proof (*induct a xs D A arbitrary*: *A rule*: *reduce-below.induct*)
case (*1 a D A*)
then show *?case*
by (*auto*, *metis invertible-mat-one left-mult-one-mat one-carrier-mat*)
next
case (*2 a x xs D A*)
note *A = 2.prems(1)*
note *a = 2.prems(2)*
note *j = 2.prems(3)*
note *d = 2.prems(4)*
note *x-xs = 2.prems(5)*
note *D0 = 2.prems(6)*
have *xm*: *x < m* **using** *2.prems* **by** *auto*
obtain *p q u v d* **where** *pquvd*: *(p,q,u,v,d) = euclid-ext2 (A$$(a,0)) (A$$(x,0))*
by (*metis prod-cases5*)
let *?reduce-ax = reduce a x D A*
have *reduce-ax*: *?reduce-ax ∈ carrier-mat m n*
by (*metis* (*no-types*, *lifting*) *2 add.comm-neutral append-rows-def*
carrier-matD carrier-mat-triv index-mat-four-block(*2*,*3*)
index-one-mat(*2*) *index-smult-mat*(*2*) *index-zero-mat*(*2*,*3*) *reduce-preserves-dimensions*)
have *h*: (∃ *P*. *invertible-mat P ∧ P ∈ carrier-mat m m*
∧ reduce-below a xs D (*reduce a x D A*) *= P ∗ reduce a x D A*)
by (*rule 2.hyps*[*OF - a j - - *],*insert d x-xs D0 reduce-ax*, *auto*)
from *this* **obtain** *P* **where** *inv-P*: *invertible-mat P* **and** *P*: *P ∈ carrier-mat m m*
and *rb-Pr*: *reduce-below a xs D* (*reduce a x D A*) *= P ∗ reduce a x D A* **by** *blast*
have ∗: *reduce-below a* (*x # xs*) *D A = reduce-below a xs D* (*reduce a x D A*) **by** *simp*
have ∃ *Q*. *invertible-mat Q ∧ Q ∈ carrier-mat m m ∧* (*reduce a x D A*) *= Q ∗ A*
by (*unfold D0*, *rule reduce-invertible-mat-D0*[*OF A a xm j*], *insert 2.prems*, *auto*)

from *this* **obtain** *Q* **where** *inv-Q*: *invertible-mat Q* **and** *Q*: $Q \in$ *carrier-mat m m*
and *r-QA*: *reduce a x D A* $= Q * A$ **by** *blast*
have *invertible-mat* $(P*Q)$ **using** *inv-P inv-Q P Q invertible-mult-JNF* **by** *blast*
moreover have $P * Q \in$ *carrier-mat m m* **using** *P Q* **by** *auto*
moreover have *reduce-below a* $(x \# xs)$ *D A* $= (P*Q) * A$
by (*smt* (*verit*) *P Q* $*$ *assoc-mult-mat carrier-matD*(*1*) *carrier-mat-triv index-mult-mat*(*2*)
r-QA rb-Pr reduce-preserves-dimensions(*1*))
ultimately show *?case* **by** *blast*
qed

lemma *reduce-not0'*:
assumes *A*: $A \in$ *carrier-mat m n* **and** *a*: $a<m$ **and** *a-less-b*: $a<b$ **and** *j*: $0<n$ **and** *b*: $b<m$
and *Aaj*: *A* $\$\$ (a,0) \neq 0$
shows *reduce a b 0 A* $\$\$ (a,\ 0) \neq 0$ (**is** *?reduce-ab* $\$\$ (a,0) \neq$ -)
proof −
have *?reduce-ab* $\$\$ (a,0) =$ (*let r* $=$ *gcd* (*A* $\$\$ (a,\ 0)$) (*A* $\$\$ (b,\ 0)$) *in if 0 dvd r then 0 else r*)
by (*rule reduce-gcd*[*OF A - j Aaj*], *insert a*, *simp*)
also have ... $\neq 0$ **unfolding** *Let-def*
by (*simp add*: *assms*(*6*))
finally show *?thesis* **.**
qed

lemma *reduce-below-preserves-D0*:
assumes *A'*: $A \in$ *carrier-mat m n* **and** *a*: $a<m$ **and** *j*: $j<n$
and *Aaj*: *A* $\$\$ (a,0) \neq 0$
assumes $i \notin$ *set xs* **and** *distinct xs* **and** $\forall x \in$ *set xs*. $x < m \wedge a < x$
and $i \neq a$ **and** $i<m$
and $D=0$
shows *reduce-below a xs D A* $\$\$ (i,j) =$ *A* $\$\$ (i,j)$
using *assms*
proof (*induct a xs D A arbitrary*: *A i rule*: *reduce-below.induct*)
case (*1 a D A*)
then show *?case* **by** *auto*
next
case (*2 a x xs D A*)
note $A =$ *2.prems*(*1*)
note $a =$ *2.prems*(*2*)
note $j =$ *2.prems*(*3*)
note *Aaj* $=$ *2.prems*(*4*)
note *i-set-xxs* $=$ *2.prems*(*5*)
note $d =$ *2.prems*(*6*)
note *xxs-less-m* $=$ *2.prems*(*7*)

note *ia* = *2.prems*(*8*)
note *im* = *2.prems*(*9*)
note *D0* = *2.prems*(*10*)
have *xm*: *x* < *m* **using** *2.prems* **by** *auto*
obtain *p q u v d* **where** *pquvd*: (*p*,*q*,*u*,*v*,*d*) = *euclid-ext2* (*A*$$(*a*,*0*)) (*A*$$(*x*,*0*))
by (*metis prod-cases5*)
let *?reduce-ax* = (*reduce a x D A*)
have *reduce-ax*: *?reduce-ax* ∈ *carrier-mat m n*
by (*metis* (*no-types*, *lifting*) *2 add.comm-neutral append-rows-def*
carrier-matD carrier-mat-triv index-mat-four-block(*2*,*3*)
index-one-mat(*2*) *index-smult-mat*(*2*) *index-zero-mat*(*2*,*3*) *reduce-preserves-dimensions*)
have *reduce-below a* (*x* # *xs*) *D A* $$ (*i*, *j*) = *reduce-below a xs D* (*reduce a x D A*) $$ (*i*, *j*)
by *auto*
also have ... = *reduce a x D A* $$ (*i*, *j*)
proof (*rule 2.hyps*[*OF - a j - -*])
show *i* ∉ *set xs* **using** *i-set-xxs* **by** *auto*
show *distinct xs* **using** *d* **by** *auto*
show ∀ *x*∈*set xs*. *x* < *m* ∧ *a* < *x* **using** *xxs-less-m* **by** *auto*
show *reduce a x D A* $$ (*a*, *0*) ≠ *0*
by (*unfold D0*, *rule reduce-not0'*[*OF A - - - - Aaj*], *insert 2.prems*, *auto*)
show *reduce a x D A* ∈ *carrier-mat m n* **using** *reduce-ax* **by** *linarith*
qed (*insert 2.prems*, *auto*)
also have ... = *A* $$ (*i*,*j*) **by** (*rule reduce-preserves*[*OF A j Aaj*], *insert 2.prems*, *auto*)
finally show *?case* **.**
qed

lemma *reduce-below-0-D0*:
assumes *A*: *A* ∈ *carrier-mat m n* **and** *a*: *a*<*m* **and** *j*: *0*<*n*
and *Aaj*: *A* $$ (*a*,*0*) ≠ *0*
assumes *i* ∈ *set xs* **and** *distinct xs* **and** ∀ *x* ∈ *set xs*. *x* < *m* ∧ *a* < *x*
and *D*=*0*
shows *reduce-below a xs D A* $$ (*i*,*0*) = *0*
using *assms*
proof (*induct a xs D A arbitrary*: *A i rule*: *reduce-below.induct*)
case (*1 a D A*)
then show *?case* **by** *auto*
next
case (*2 a x xs D A*)
note *A* = *2.prems*(*1*)
note *a* = *2.prems*(*2*)
note *j* = *2.prems*(*3*)
note *Aaj* = *2.prems*(*4*)
note *i-set-xxs* = *2.prems*(*5*)
note *d* = *2.prems*(*6*)
note *xxs-less-m* = *2.prems*(*7*)

```
  note D0 = 2.prems(8)
  have xm: x < m  using 2.prems by auto
  obtain p q u v d where pquvd: (p,q,u,v,d) = euclid-ext2 (A$$(a,0)) (A$$(x,0))
    by (metis prod-cases5)
  let ?reduce-ax = reduce a x D A
  have reduce-ax: ?reduce-ax ∈ carrier-mat m n
    by (metis (no-types, lifting) 2 add.comm-neutral append-rows-def
        carrier-matD carrier-mat-triv index-mat-four-block(2,3)
      index-one-mat(2) index-smult-mat(2) index-zero-mat(2,3) reduce-preserves-dimensions)
  show ?case
  proof (cases i=x)
    case True
    have reduce-below a (x # xs) D A $$ (i, 0) = reduce-below a xs D (reduce a x
D A) $$ (i, 0)
      by auto
    also have ... = (reduce a x D A) $$ (i, 0)
    proof (rule reduce-below-preserves-D0[OF - a j - - ])
      show reduce a x D A ∈ carrier-mat m n using reduce-ax by linarith
      show distinct xs using d by auto
      show ∀ x∈set xs. x < m ∧ a < x using xxs-less-m by auto
      show reduce a x D A $$ (a, 0) ≠ 0
        by (unfold D0, rule reduce-not0'[OF A - - j - Aaj], insert 2.prems, auto)
      show i ∉ set xs using True d by auto
      show i ≠ a using 2.prems by blast
      show i < m by (simp add: True trans-less-add1 xm)
    qed (insert D0)
    also have ... = 0 unfolding True by (rule reduce-0[OF A - j - - Aaj], insert
2.prems, auto)
    finally show ?thesis .
  next
    case False note i-not-x = False
    have h: reduce-below a xs D (reduce a x D A) $$ (i, 0) = 0
    proof (rule 2.hyps[OF - a j - - ])
      show reduce a x D A ∈ carrier-mat m n using reduce-ax by linarith
      show i ∈ set xs using i-set-xxs i-not-x by auto
      show distinct xs using d by auto
      show ∀ x∈set xs. x < m ∧ a < x using xxs-less-m by auto
      show reduce a x D A $$ (a, 0) ≠ 0
        by (unfold D0, rule reduce-not0'[OF A - - j - Aaj], insert 2.prems, auto)
    qed (insert D0)
    have reduce-below a (x # xs) D A $$ (i, 0) = reduce-below a xs D (reduce a x
D A) $$ (i, 0)
      by auto
    also have ... = 0 using h .
    finally show ?thesis .
  qed
qed

end
```

Definition of the echelon form algorithm in JNF

primrec *bezout-iterate-JNF*
where *bezout-iterate-JNF A 0 i j bezout = A*
| *bezout-iterate-JNF A (Suc n) i j bezout =*
(if (Suc n) ≤ i then A else
bezout-iterate-JNF (bezout-matrix-JNF A i ((Suc n)) j bezout ∗ A) n i j bezout)

definition
echelon-form-of-column-k-JNF bezout A′ k =
(let (A, i) = A′
in if (i = dim-row A) ∨ (∀ m ∈ {i..<dim-row A}. A $$ (m, k) = 0) then (A, i) else
if (∀ m∈{i+1..<dim-row A}. A $$ (m,k) = 0) then (A, i + 1) else
let n = (LEAST n. A $$ (n,k) ≠ 0 ∧ i ≤ n);
interchange-A = swaprows i n A
in
(bezout-iterate-JNF (interchange-A) (dim-row A − 1) i k bezout, i + 1))

definition *echelon-form-of-upt-k-JNF A k bezout = (fst (foldl (echelon-form-of-column-k-JNF bezout) (A,0) [0..<Suc k]))*
definition *echelon-form-of-JNF A bezout = echelon-form-of-upt-k-JNF A (dim-col A − 1) bezout*

context includes *lifting-syntax*
begin

lemma *HMA-bezout-iterate*[*transfer-rule*]:
assumes *n<CARD('m)*
shows *((Mod-Type-Connect.HMA-M :: - ⇒ int ^ 'n :: mod-type ^ 'm :: mod-type ⇒ -)*
===> (Mod-Type-Connect.HMA-I) ===> (Mod-Type-Connect.HMA-I) ===> (=) ===> (Mod-Type-Connect.HMA-M))
(λA i j bezout. bezout-iterate-JNF A n i j bezout)
(λA i j bezout. bezout-iterate A n i j bezout)

proof (*intro rel-funI, goal-cases*)
case (*1 A A′ i i′ j j′ bezout bezout′*)
then show *?case* **using** *assms*
proof (*induct n arbitrary*: *A A′*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc n*)
note *AA′*[*transfer-rule*] = *Suc.prems*(*1*)
note *ii′*[*transfer-rule*] = *Suc.prems*(*2*)
note *jj′*[*transfer-rule*] = *Suc.prems*(*3*)

note *bb'*[*transfer-rule*] = *Suc.prems*(*4*)
note *Suc-n-less-m* = *Suc.prems*(*5*)
let *?BI-JNF* = *bezout-iterate-JNF*
let *?BI-HMA* = *bezout-iterate*
let *?from-nat-rows* = *mod-type-class.from-nat* :: - ⇒ *'m*
have *Sucn*[*transfer-rule*]: *Mod-Type-Connect.HMA-I* (*Suc n*) (*?from-nat-rows* (*Suc n*))
unfolding *Mod-Type-Connect.HMA-I-def*
by (*simp add*: *Suc-lessD Suc-n-less-m mod-type-class.from-nat-to-nat*)
have *n*: *n* < *CARD*('*m*) **using** *Suc-n-less-m* **by** *simp*
have [*transfer-rule*]:
Mod-Type-Connect.HMA-M (*?BI-JNF* (*bezout-matrix-JNF A i* (*Suc n*) *j bezout* ∗ *A*) *n i j bezout*)
(*?BI-HMA* (*bezout-matrix A' i'* (*?from-nat-rows* (*Suc n*)) *j' bezout'* ∗∗ *A'*) *n i' j' bezout'*)
by (*rule Suc.hyps*[*OF - ii' jj' bb' n*], *transfer-prover*)
moreover have *Suc n* ≤ *i* ⟹ *Suc n* ≤ *mod-type-class.to-nat i'*
and *Suc n* > *i* ⟹ *Suc n* > *mod-type-class.to-nat i'*
by (*metis 1*(*2*) *Mod-Type-Connect.HMA-I-def*)+
ultimately show *?case* **using** *AA'* **by** *auto*
qed
qed

corollary *HMA-bezout-iterate'*[*transfer-rule*]:
fixes *A'*::*int* ^ '*n* :: *mod-type* ^ '*m* :: *mod-type*
assumes *n*: *n*<*CARD*('*m*)
and *Mod-Type-Connect.HMA-M A A'*
and *Mod-Type-Connect.HMA-I i i'* **and** *Mod-Type-Connect.HMA-I j j'*
shows *Mod-Type-Connect.HMA-M* (*bezout-iterate-JNF A n i j bezout*) (*bezout-iterate A' n i' j' bezout*)
using *assms HMA-bezout-iterate*[*OF n*] **unfolding** *rel-fun-def* **by** *force*

lemma *snd-echelon-form-of-column-k-JNF-le-dim-row*:
assumes *i*<*dim-row A*
shows *snd* (*echelon-form-of-column-k-JNF bezout* (*A*,*i*) *k*) ≤ *dim-row A*
using *assms* **unfolding** *echelon-form-of-column-k-JNF-def* **by** *auto*

lemma *HMA-echelon-form-of-column-k*[*transfer-rule*]:
assumes *k*: *k*<*CARD*('*n*)
shows ((=) ===> *rel-prod* (*Mod-Type-Connect.HMA-M* :: - ⇒ *int* ^ '*n* :: *mod-type* ^ '*m* :: *mod-type* ⇒ -) (λ*a b*. *a*=*b* ∧ *a*≤*CARD*('*m*))
===> (*rel-prod* (*Mod-Type-Connect.HMA-M*) (λ*a b*. *a*=*b* ∧ *a*≤*CARD*('*m*))))
(λ*bezout A*. *echelon-form-of-column-k-JNF bezout A k*)
(λ*bezout A*. *echelon-form-of-column-k bezout A k*)

proof (*intro rel-funI*, *goal-cases*)
case (*1 bezout bezout' xa ya*)
obtain *A i* **where** *xa*: *xa* = (*A*,*i*) **using** *surjective-pairing* **by** *blast*
obtain *A' i'* **where** *ya*: *ya* = (*A'*,*i'*) **using** *surjective-pairing* **by** *blast*
have *ii'*[*transfer-rule*]: *i*=*i'* **using** *1*(*2*) *xa ya* **by** *auto*
have *i-le-m*: *i*≤*CARD*('*m*) **using** *1*(*2*) *xa ya* **by** *auto*
have *AA'*[*transfer-rule*]: *Mod-Type-Connect.HMA-M A A'* **using** *1*(*2*) *xa ya* **by** *auto*
have *bb'*[*transfer-rule*]: *bezout*=*bezout'* **using** *1* **by** *auto*
let *?from-nat-rows* = *mod-type-class.from-nat* :: - ⇒ '*m*
let *?from-nat-cols* = *mod-type-class.from-nat* :: - ⇒ '*n*
have *kk'*[*transfer-rule*]: *Mod-Type-Connect.HMA-I k* (*?from-nat-cols k*)
by (*simp add*: *Mod-Type-Connect.HMA-I-def assms mod-type-class.to-nat-from-nat-id*)
have *c1-eq*: (*i* = *dim-row A*) = (*i* = *nrows A'*)
by (*metis AA' Mod-Type-Connect.dim-row-transfer-rule nrows-def*)
have *c2-eq*: (∀ *m* ∈ {*i*..<*dim-row A*}. *A* $$ (*m*, *k*) = *0*)
= (∀ *m*≥*?from-nat-rows i*. *A'* $ *m* $ *?from-nat-cols k* = *0*) (**is** *?lhs* = *?rhs*) **if** *i-not*: *i*≠*dim-row A*
proof
assume *lhs*: *?lhs*
show *?rhs*
proof (*rule*+)
fix *m*
assume *im*: *?from-nat-rows i* ≤ *m*
have *im'*: *i*<*CARD*('*m*) **using** *i-le-m i-not*
by (*simp add*: *c1-eq dual-order.order-iff-strict nrows-def*)
let *?m'* = *mod-type-class.to-nat m*
have *mm'*[*transfer-rule*]: *Mod-Type-Connect.HMA-I ?m' m*
by (*simp add*: *Mod-Type-Connect.HMA-I-def*)
from *im* **have** *mod-type-class.to-nat* (*?from-nat-rows i*) ≤ *?m'*
by (*simp add*: *to-nat-mono'*)
hence *?m'* >= *i* **using** *im im'* **by** (*simp add*: *mod-type-class.to-nat-from-nat-id*)
hence *?m'* ∈ {*i*..<*dim-row A*}
using *AA' Mod-Type-Connect.dim-row-transfer-rule mod-type-class.to-nat-less-card* **by** *fastforce*
hence *A* $$ (*?m'*, *k*) = *0* **using** *lhs* **by** *auto*
moreover have *A* $$ (*?m'*, *k*) = *A'* $*h m* $*h ?from-nat-cols k* **unfolding** *index-hma-def*[*symmetric*] **by** *transfer-prover*
ultimately show *A'* $*h m* $*h ?from-nat-cols k* = *0* **by** *simp*
qed
next
assume *rhs*: *?rhs*
show *?lhs*
proof (*rule*)
fix *m* **assume** *m*: *m* ∈ {*i*..<*dim-row A*}
let *?m* = *?from-nat-rows m*
have *mm'*[*transfer-rule*]: *Mod-Type-Connect.HMA-I m ?m*
by (*metis AA' Mod-Type-Connect.HMA-I-def Mod-Type-Connect.dim-row-transfer-rule*

```
        atLeastLessThan-iff m mod-type-class.from-nat-to-nat)
    have m-ge-i: ?m≥?from-nat-rows i
      using AA' Mod-Type-Connect.dim-row-transfer-rule from-nat-mono' m by
fastforce
    hence A' $h ?m $h ?from-nat-cols k = 0 using rhs by auto
    moreover have A $$ (m, k) = A' $h ?m $h ?from-nat-cols k
      unfolding index-hma-def[symmetric] by transfer-prover
    ultimately show A $$ (m, k) = 0 by simp
   qed
  qed
  show ?case
  proof (cases (i = dim-row A) ∨ (∀ m ∈ {i..<dim-row A}. A $$ (m, k) = 0))
   case True
   hence ∗: (∀ m≥?from-nat-rows i. A' $ m $ ?from-nat-cols k = 0) ∨ (i = nrows
A')
    using c1-eq c2-eq by auto
   have echelon-form-of-column-k-JNF bezout xa k = (A,i)
    unfolding echelon-form-of-column-k-JNF-def using True xa by auto
   moreover have echelon-form-of-column-k bezout ya k = (A',i')
    unfolding echelon-form-of-column-k-def Let-def using ∗ ya ii' by simp
   ultimately show ?thesis unfolding xa ya rel-prod.simps using AA' ii' bb'
i-le-m by blast
  next
   case False note not-c1 = False
   hence im': i<CARD('m)
    by (metis c1-eq dual-order.order-iff-strict i-le-m nrows-def)
   have ∗: (∀ m∈{i+1..<dim-row A}. A $$ (m,k) = 0)
    = (∀ m>?from-nat-rows i. A' $ m $ ?from-nat-cols k = 0) (is ?lhs = ?rhs)
   proof
    assume lhs: ?lhs
    show ?rhs
    proof (rule+)
     fix m
     assume im: ?from-nat-rows i < m
     let ?m' = mod-type-class.to-nat m
     have mm'[transfer-rule]: Mod-Type-Connect.HMA-I ?m' m
      by (simp add: Mod-Type-Connect.HMA-I-def)
     from im have mod-type-class.to-nat (?from-nat-rows i) < ?m'
      by (simp add: to-nat-mono)
    hence ?m' > i using im im' by (simp add: mod-type-class.to-nat-from-nat-id)
     hence ?m' ∈ {i+1..<dim-row A}
     using AA' Mod-Type-Connect.dim-row-transfer-rule mod-type-class.to-nat-less-card
by fastforce
     hence A $$ (?m', k) = 0 using lhs by auto
     moreover have A $$ (?m', k) = A' $h m $h ?from-nat-cols k unfolding
index-hma-def[symmetric] by transfer-prover
     ultimately show A' $h m $h ?from-nat-cols k = 0 by simp
    qed
   next
```

assume *rhs*: *?rhs*
show *?lhs*
proof (*rule*)
fix *m* **assume** *m*: *m* ∈ {*i+1*..<*dim-row A*}
let *?m* = *?from-nat-rows m*
have *mm'*[*transfer-rule*]: *Mod-Type-Connect.HMA-I m ?m*
by (*metis AA' Mod-Type-Connect.HMA-I-def Mod-Type-Connect.dim-row-transfer-rule atLeastLessThan-iff m mod-type-class.from-nat-to-nat*)
have *m-ge-i*: *?m*>*?from-nat-rows i*
by (*metis Mod-Type-Connect.HMA-I-def One-nat-def add-Suc-right atLeastLessThan-iff from-nat-mono le-simps*(*3*) *m mm' mod-type-class.to-nat-less-card nat-arith.rule0*)
hence *A' $h ?m $h ?from-nat-cols k = 0* **using** *rhs* **by** *auto*
moreover have *A $$ (m, k) = A' $h ?m $h ?from-nat-cols k*
unfolding *index-hma-def*[*symmetric*] **by** *transfer-prover*
ultimately show *A $$ (m, k) = 0* **by** *simp*
qed
qed
show *?thesis*
proof (*cases* (∀ *m*∈{*i+1*..<*dim-row A*}. *A $$ (m,k) = 0*))
case *True*
have *echelon-form-of-column-k-JNF bezout xa k* = (*A*,*i+1*)
unfolding *echelon-form-of-column-k-JNF-def* **using** *True xa not-c1* **by** *auto*
moreover have *echelon-form-of-column-k bezout ya k* = (*A'*,*i'+1*)
unfolding *echelon-form-of-column-k-def Let-def* **using** *ya ii'* ∗ *True c1-eq c2-eq not-c1* **by** *auto*
ultimately show *?thesis* **unfolding** *xa ya rel-prod.simps* **using** *AA' ii' bb' i-le-m im'*
by *auto*
next
case *False*
hence ∗: ¬ (∀ *m*>*?from-nat-rows i*. *A' $ m $?from-nat-cols k = 0*) **using** ∗ **by** *auto*
have ∗∗: ¬ ((∀ *m*≥*?from-nat-rows i*. *A' $h m $h ?from-nat-cols k = 0*) ∨ *i* = *nrows A'*)
using *c1-eq c2-eq not-c1* **by** *auto*
define *n* **where** *n*=(*LEAST n*. *A $$ (n,k)* ≠ *0* ∧ *i* ≤ *n*)
define *n'* **where** *n'*=(*LEAST n*. *A' $ n $?from-nat-cols k* ≠ *0* ∧ *?from-nat-rows i* ≤ *n*)
let *?interchange-A* = *swaprows i n A*
let *?interchange-A'* = *interchange-rows A' (?from-nat-rows i') n'*
have *nn'*[*transfer-rule*]: *Mod-Type-Connect.HMA-I n n'*
proof −
let *?n'* = *mod-type-class.to-nat n'*
have *exist*: ∃ *n*. *A' $ n $?from-nat-cols k* ≠ *0* ∧ *?from-nat-rows i* ≤ *n*
using ∗ **by** *auto*
from *this* **obtain** *a* **where** *c*: *A' $ a $?from-nat-cols k* ≠ *0* ∧ *?from-nat-rows i* ≤ *a* **by** *blast*
have *n* = *?n'*

proof (*unfold n-def*, *rule Least-equality*)
have *n′n′*[*transfer-rule*]: *Mod-Type-Connect.HMA-I ?n′ n′*
by (*simp add*: *Mod-Type-Connect.HMA-I-def*)
have *e*: (*A′ $ n′ $?from-nat-cols k ≠ 0 ∧ ?from-nat-rows i ≤ n′*)
by (*metis* (*mono-tags*, *lifting*) *LeastI c2-eq n′-def not-c1*)
hence *i ≤ mod-type-class.to-nat n′*
using *im′ mod-type-class.from-nat-to-nat to-nat-mono′* **by** *fastforce*
moreover have *A′ $ n′ $?from-nat-cols k = A $$* (*?n′*, *k*)
unfolding *index-hma-def*[*symmetric*] **by** (*transfer′*, *auto*)
ultimately show *A $$* (*?n′*, *k*) *≠ 0 ∧ i ≤ ?n′*
using *e* **by** *auto*
show *⋀y. A $$* (*y*, *k*) *≠ 0 ∧ i ≤ y ⟹ mod-type-class.to-nat n′ ≤ y*
unfolding *n′-def*
by (*smt* (*verit*, *del-insts*) *AA′ Mod-Type-Connect.HMA-M-def Mod-Type-Connect.from-hma$_m$-def assms from-nat-mono*
from-nat-mono′ index-mat(*1*) *less-trans mod-type-class.from-nat-to-nat-id mod-type-class.to-nat-less-card*
not-le prod.simps(*2*) *wellorder-Least-lemma*(*2*))
qed
thus *?thesis* **unfolding** *Mod-Type-Connect.HMA-I-def* **by** *auto*
qed
have *dr1*[*transfer-rule*]: (*nrows A′ − 1*) = (*dim-row A − 1*) **unfolding** *nrows-def*
using *AA′ Mod-Type-Connect.dim-row-transfer-rule* **by** *force*
have *ii′2*[*transfer-rule*]: *Mod-Type-Connect.HMA-I i* (*?from-nat-rows i′*)
by (*metis ∗∗ Mod-Type-Connect.HMA-I-def i-le-m ii′ le-neq-implies-less*
mod-type-class.to-nat-from-nat-id nrows-def)
have *ii′3*[*transfer-rule*]: *Mod-Type-Connect.HMA-I i′* (*?from-nat-rows i′*)
using *ii′ ii′2* **by** *blast*
let *?BI-JNF* = (*bezout-iterate-JNF* (*?interchange-A*) (*dim-row A − 1*) *i k bezout*)
let *?BI-HA* = (*bezout-iterate* (*?interchange-A′*) (*nrows A′ − 1*) (*?from-nat-rows i*) (*?from-nat-cols k*) *bezout*)
have *e-rw*: *echelon-form-of-column-k-JNF bezout xa k* = (*?BI-JNF*,*i+1*)
unfolding *echelon-form-of-column-k-JNF-def n-def* **using** *False xa not-c1* **by** *auto*
have *e-rw2*: *echelon-form-of-column-k bezout ya k* = (*?BI-HA*,*i+1*)
unfolding *echelon-form-of-column-k-def Let-def n′-def* **using** *∗ ya ∗∗ ii′* **by** *auto*
have *s*[*transfer-rule*]: *Mod-Type-Connect.HMA-M* (*swaprows i′ n A*) (*interchange-rows A′* (*?from-nat-rows i′*) *n′*)
by *transfer-prover*
have *n-CARD*: (*nrows A′ − 1*) *< CARD*(*′m*) **unfolding** *nrows-def* **by** *auto*
note *a*[*transfer-rule*] = *HMA-bezout-iterate*[*OF n-CARD*]
have *BI*[*transfer-rule*]:*Mod-Type-Connect.HMA-M ?BI-JNF ?BI-HA* **unfolding** *ii′ dr1*
by (*rule HMA-bezout-iterate′*[*OF - s ii′3 kk′*], *insert n-CARD*, *transfer′*, *simp*)
thus *?thesis* **using** *e-rw e-rw2 bb′*

by (*metis* (*mono-tags*, *lifting*) *AA' False Mod-Type-Connect.dim-row-transfer-rule*

atLeastLessThan-iff dual-order.trans order-less-imp-le rel-prod-inject)
qed
qed
qed

corollary *HMA-echelon-form-of-column-k'*[*transfer-rule*]:
assumes *k*: *k*<*CARD*('*n*) **and** *i*≤*CARD*('*m*)
and (*Mod-Type-Connect.HMA-M* :: - ⇒ *int* ^ '*n* :: *mod-type* ^ '*m* :: *mod-type* ⇒ -) *A A'*
shows (*rel-prod* (*Mod-Type-Connect.HMA-M*) (λ*a b*. *a*=*b* ∧ *a*≤*CARD*('*m*)))
(*echelon-form-of-column-k-JNF bezout* (*A*,*i*) *k*)
(*echelon-form-of-column-k bezout* (*A'*,*i*) *k*)
using *assms HMA-echelon-form-of-column-k*[*OF k*] **unfolding** *rel-fun-def* **by** *force*

lemma *HMA-foldl-echelon-form-of-column-k*:
assumes *k*: *k*≤*CARD*('*n*)
shows ((*Mod-Type-Connect.HMA-M* :: - ⇒ *int* ^ '*n* :: *mod-type* ^ '*m* :: *mod-type* ⇒ -) ===> (=)
===> (*rel-prod* (*Mod-Type-Connect.HMA-M*) (λ*a b*. *a*=*b* ∧ *a*≤*CARD*('*m*))))
(λ*A bezout*. (*foldl* (*echelon-form-of-column-k-JNF bezout*) (*A*,*0*) [*0*..<*k*]))
(λ*A bezout*. (*foldl* (*echelon-form-of-column-k bezout*) (*A*,*0*) [*0*..<*k*]))
proof (*intro rel-funI*, *goal-cases*)
case (*1 A A' bezout bezout'*)
then show *?case* **using** *assms*
proof (*induct k arbitrary*: *A A'*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc k*)
note *AA'*[*transfer-rule*] = *Suc.prems*(*1*)
note *bb'*[*transfer-rule*] = *Suc.prems*(*2*)
note *Suc-k-less-m* = *Suc.prems*(*3*)
let *?foldl-JNF* = *foldl* (*echelon-form-of-column-k-JNF bezout*) (*A*,*0*)
let *?foldl-HA* = *foldl* (*echelon-form-of-column-k bezout'*) (*A'*,*0*)
have *set-rw*: [*0*..<*Suc k*] = [*0*..<*k*] @ [*k*] **by** *auto*
have *f-JNF*: *?foldl-JNF* [*0*..<*Suc k*] = *echelon-form-of-column-k-JNF bezout* (*?foldl-JNF* [*0*..<*k*]) *k*
by *auto*
have *f-HA*: *?foldl-HA* [*0*..<*Suc k*] = *echelon-form-of-column-k bezout'* (*?foldl-HA* [*0*..<*k*]) *k*
by *auto*
have *hyp*[*transfer-rule*]: *rel-prod Mod-Type-Connect.HMA-M* (λ*a b*. *a*=*b* ∧ *a*≤*CARD*('*m*)) (*?foldl-JNF* [*0*..<*k*]) (*?foldl-HA* [*0*..<*k*])
by (*rule Suc.hyps*[*OF AA'*], *insert Suc.prems*, *auto*)
show *?case* **unfolding** *f-JNF* **unfolding** *f-HA bb'* **using** *HMA-echelon-form-of-column-k'*
by (*smt* (*verit*) *1*(*2*) *Suc-k-less-m Suc-le-lessD hyp rel-prod.cases*)

qed
qed

lemma *HMA-echelon-form-of-upt-k*[*transfer-rule*]:
assumes *k*: *k*<*CARD*('*n*)
shows ((*Mod-Type-Connect.HMA-M* :: - ⇒ *int* ^ '*n* :: *mod-type* ^ '*m* :: *mod-type*
⇒ -) ===> (=)
===> (*Mod-Type-Connect.HMA-M*))
(λ*A bezout*. *echelon-form-of-upt-k-JNF A k bezout*)
(λ*A bezout*. *echelon-form-of-upt-k A k bezout*)

proof (*intro rel-funI*, *goal-cases*)
case (*1 A A' bezout bezout'*)
have *k'*: *Suc k* ≤ *CARD*('*n*) **using** *k* **by** *auto*
have *rel-foldl*: (*rel-prod* (*Mod-Type-Connect.HMA-M*) (λ*a b*. *a*=*b* ∧ *a*≤*CARD*('*m*)))

(*foldl* (*echelon-form-of-column-k-JNF bezout*) (*A*,*0*) [*0*..<*Suc k*])
(*foldl* (*echelon-form-of-column-k bezout*) (*A'*,*0*) [*0*..<*Suc k*])
using *HMA-foldl-echelon-form-of-column-k*[*OF k'*] **by** (*smt* (*verit*) *1*(*1*) *rel-fun-def*)
then show *?case* **using** *assms* **unfolding** *echelon-form-of-upt-k-JNF-def echelon-form-of-upt-k-def*
by (*metis* (*no-types*, *lifting*) *1*(*2*) *prod.collapse rel-prod-inject*)
qed

lemma *HMA-echelon-form-of*[*transfer-rule*]:
shows ((*Mod-Type-Connect.HMA-M* :: - ⇒ *int* ^ '*n* :: *mod-type* ^ '*m* :: *mod-type*
⇒ -) ===> (=)
===> (*Mod-Type-Connect.HMA-M*))
(λ*A bezout*. *echelon-form-of-JNF A bezout*)
(λ*A bezout*. *echelon-form-of A bezout*)

proof (*intro rel-funI*, *goal-cases*)
case (*1 A A' bezout bezout'*)
note *AA'*[*transfer-rule*] = *1*(*1*)
note *bb'*[*transfer-rule*] = *1*(*2*)
have ∗: (*dim-col A* − *1*) < *CARD*('*n*) **using** *1*
using *Mod-Type-Connect.dim-col-transfer-rule* **by** *force*
note ∗∗[*transfer-rule*] = *HMA-echelon-form-of-upt-k*[*OF* ∗]
have [*transfer-rule*]: (*ncols A'* − *1*) = (*dim-col A* − *1*)
by (*metis 1*(*1*) *Mod-Type-Connect.dim-col-transfer-rule ncols-def*)
have [*transfer-rule*]: (*dim-col A* − *1*) = (*dim-col A* − *1*) **..**
show *?case* **unfolding** *echelon-form-of-def echelon-form-of-JNF-def bb'*
by (*metis* (*mono-tags*) ∗∗ *1*(*1*) ‹*ncols A'* − *1* = *dim-col A* − *1*› *rel-fun-def*)
qed
end

context
begin

private lemma *echelon-form-of-euclidean-invertible-mod-type*:
fixes *A*::*int mat*
assumes $A \in$ *carrier-mat CARD*('*m*::*mod-type*) *CARD*('*n*::*mod-type*)
shows $\exists P.$ *invertible-mat* $P \wedge P \in$ *carrier-mat* (*CARD*('*m*::*mod-type*)) (*CARD*('*m*::*mod-type*))

$\wedge$ $P * A =$ *echelon-form-of-JNF A euclid-ext2*
$\wedge$ *echelon-form-JNF* (*echelon-form-of-JNF A euclid-ext2*)
proof −
define A' **where** $A' = (Mod\text{-}Type\text{-}Connect.to\text{-}hma_m\ A :: int\ \widehat{}'n :: mod\text{-}type\ \widehat{}'m :: mod\text{-}type)$
have *AA'*[*transfer-rule*]: *Mod-Type-Connect.HMA-M A A'*
unfolding *Mod-Type-Connect.HMA-M-def* **using** *assms A'-def* **by** *auto*
have [*transfer-rule*]: *Mod-Type-Connect.HMA-M*
(*echelon-form-of-JNF A euclid-ext2*) (*echelon-form-of A' euclid-ext2*)
by *transfer-prover*
have $\exists P.$ *invertible* $P \wedge P{*}{*}A' =$ (*echelon-form-of A' euclid-ext2*)
$\wedge$ *echelon-form* (*echelon-form-of A' euclid-ext2*)
by (*rule echelon-form-of-euclidean-invertible*)
thus *?thesis* **by** (*transfer*, *auto*)
qed

private lemma *echelon-form-of-euclidean-invertible-nontriv-mod-ring*:
fixes *A*::*int mat*
assumes *A* ∈ *carrier-mat CARD*('*m*::*nontriv mod-ring*) *CARD*('*n*::*nontriv mod-ring*)
shows ∃ *P*. *invertible-mat P* ∧ *P* ∈ *carrier-mat* (*CARD*('*m*)) (*CARD*('*m*))
∧ *P* ∗ *A* = *echelon-form-of-JNF A euclid-ext2*
∧ *echelon-form-JNF* (*echelon-form-of-JNF A euclid-ext2*)
using *assms echelon-form-of-euclidean-invertible-mod-type* **by** (*smt* (*verit*) *CARD-mod-ring*)

lemmas *echelon-form-of-euclidean-invertible-nontriv-mod-ring-internalized* =
echelon-form-of-euclidean-invertible-nontriv-mod-ring[*unfolded CARD-mod-ring*,
internalize-sort '*m*::*nontriv*, *internalize-sort* '*b*::*nontriv*]

context
fixes *m*::*nat* **and** *n*::*nat*
assumes *local-typedef1*: ∃ (*Rep* :: ('*b* ⇒ *int*)) *Abs*. *type-definition Rep Abs* {*0*..<*m* :: *int*}
assumes *local-typedef2*: ∃ (*Rep* :: ('*c* ⇒ *int*)) *Abs*. *type-definition Rep Abs* {*0*..<*n* :: *int*}
and *m*: *m>1*
and *n*: *n>1*
begin

lemma *echelon-form-of-euclidean-invertible-nontriv-mod-ring-aux*:
fixes *A*::*int mat*
assumes $A \in$ *carrier-mat m n*
shows $\exists P.$ *invertible-mat* $P \wedge P \in$ *carrier-mat m m*
$\wedge$ $P *$ *A* = *echelon-form-of-JNF A euclid-ext2*
$\wedge$ *echelon-form-JNF* (*echelon-form-of-JNF A euclid-ext2*)
using *echelon-form-of-euclidean-invertible-nontriv-mod-ring-internalized*
[*OF type-to-set2*(*1*)[*OF local-typedef1 local-typedef2*]
type-to-set1(*1*)[*OF local-typedef1 local-typedef2*]]
using *assms*
using *type-to-set1*(*2*) *local-typedef1 local-typedef2 n m* **by** *metis*

end

context
begin

private lemma *echelon-form-of-euclidean-invertible-cancelled-first*:
$\exists$ *Rep Abs. type-definition Rep Abs* {*0*..<*int n*} $\Longrightarrow$ *1* < *m* $\Longrightarrow$ *1* < *n* $\Longrightarrow$
A $\in$ *carrier-mat m n* $\Longrightarrow$ $\exists P.$ *invertible-mat* $P \wedge P \in$ *carrier-mat m m*
$\wedge$ $P *$ (*A*::*int mat*) = *echelon-form-of-JNF A euclid-ext2* $\wedge$ *echelon-form-JNF* (*echelon-form-of-JNF A euclid-ext2*)
using *echelon-form-of-euclidean-invertible-nontriv-mod-ring-aux*[*cancel-type-definition*, *of m n A*]
by *force*

private lemma *echelon-form-of-euclidean-invertible-cancelled-both*:
1 < *m* $\Longrightarrow$ *1* < *n* $\Longrightarrow$ *A* $\in$ *carrier-mat m n* $\Longrightarrow$ $\exists P.$ *invertible-mat* $P \wedge P \in$ *carrier-mat m m*
$\wedge$ $P *$ (*A*::*int mat*) = *echelon-form-of-JNF A euclid-ext2* $\wedge$ *echelon-form-JNF* (*echelon-form-of-JNF A euclid-ext2*)
using *echelon-form-of-euclidean-invertible-cancelled-first*[*cancel-type-definition*, *of n m A*]
by *force*

lemma *echelon-form-of-euclidean-invertible′*:
fixes *A*::*int mat*
assumes *A* $\in$ *carrier-mat m n*
and *1* < *m* **and** *1* < *n*
shows $\exists P.$ *invertible-mat* $P \wedge$
$P \in$ *carrier-mat m m* $\wedge$ $P *$ *A* = *echelon-form-of-JNF A euclid-ext2*
$\wedge$ *echelon-form-JNF* (*echelon-form-of-JNF A euclid-ext2*)

using *echelon-form-of-euclidean-invertible-cancelled-both assms* **by** *auto*
end
end

context *mod-operation*
begin

definition *FindPreHNF-rectangular A*
= (*let m = dim-row A*; *n = dim-col A in*
if m < 2 ∨ n = 0 then A else — No operations are carried out if m = 1
if n = 1 then
let non-zero-positions = filter (λ*i. A* $$ (*i*,*0*) ≠ *0*) [*1*..<*dim-row A*] *in*
if non-zero-positions = [] *then A*
else let A′ = (*if A*$$(*0*,*0*) ≠ *0 then A else let i = non-zero-positions* ! *0 in*
swaprows 0 i A)
in reduce-below-impl 0 non-zero-positions 0 A′
else (*echelon-form-of-JNF A euclid-ext2*))

This is the (non-efficient) HNF algorithm obtained from the echelon form and Hermite normal form AFP entries

definition *HNF-algorithm-from-HA A*
= *Hermite-of-list-of-rows* (*FindPreHNF-rectangular A*) [*0*..<(*dim-row A*)]

Now we can combine *FindPreHNF-rectangular*, *FindPreHNF* and *Hermite-of-list-of-rows* to get an algorithm to compute the HNF of any matrix (if it is square and invertible, then the HNF is computed reducing entries modulo D)

definition *HNF-algorithm abs-flag A* =
(*let m = dim-row A*; *n = dim-col A in*
if m ≠ n then Hermite-of-list-of-rows (*FindPreHNF-rectangular A*) [*0*..<*m*]
else
let D = abs (*det-int A*) *in*
if D = 0 then Hermite-of-list-of-rows (*FindPreHNF-rectangular A*) [*0*..<*m*]
else
let A′ = *A* @$_r$ *D* ·$_m$ *1*$_m$ *n*;
E = FindPreHNF abs-flag D A′;
H = Hermite-of-list-of-rows E [*0*..<*m*+*n*]
in mat-of-rows n (*map* (*Matrix.row H*) [*0*..<*m*]))

end

declare *mod-operation.FindPreHNF-rectangular-def*[*code*]
declare *mod-operation.HNF-algorithm-from-HA-def*[*code*]
declare *mod-operation.HNF-algorithm-def*[*code*]

context *proper-mod-operation*
begin

lemma *FindPreHNF-rectangular-soundness*:

fixes *A*::*int mat*
assumes *A*: *A* ∈ *carrier-mat m n*
shows ∃ *P*. *invertible-mat P* ∧ *P* ∈ *carrier-mat m m* ∧ *P* ∗ *A* = *FindPreHNF-rectangular A*
∧ *echelon-form-JNF* (*FindPreHNF-rectangular A*)
proof (*cases m* < *2* ∨ *n* = *0*)
case *True*
then show *?thesis*
by (*smt* (*verit*) *A FindPreHNF-rectangular-def carrier-matD echelon-form-JNF-1xn echelon-form-mx0*
invertible-mat-one left-mult-one-mat one-carrier-mat)
next
case *False*
have *m1*: *m*>*1* **using** *False* **by** *auto*
have *n0*: *n*>*0* **using** *False* **by** *auto*
show *?thesis*
proof (*cases n*=*1*)
case *True* **note** *n1* = *True*
let *?nz* = *filter* (λ*i*. *A* $$ (*i*,*0*) ≠ *0*) [*1*..<*dim-row A*]
let *?A′* = (*if A*$$(*0*,*0*) ≠ *0 then A else let i* = *?nz* ! *0 in swaprows 0 i A*)
have *A′*: *?A′* ∈ *carrier-mat m n* **using** *A* **by** *auto*
have *A′00*: *?A′* $$ (*0*,*0*) ≠ *0* **if** *?nz* ≠ []
by (*smt* (*verit*) *True assms carrier-matD index-mat-swaprows*(*1*) *length-greater-0-conv m1*
mem-Collect-eq nat-SN.gt-trans nth-mem set-filter that zero-less-one-class.zero-less-one)
have *e-r*: *echelon-form-JNF* (*reduce-below 0 ?nz 0 ?A′*) **if** *nz-not-empty*: *?nz* ≠ []
proof (*rule echelon-form-JNF-mx1*)
show (*reduce-below 0 ?nz 0 ?A′*) ∈ *carrier-mat m n* **using** *A reduce-below* **by** *auto*
have (*reduce-below 0 ?nz 0 ?A′*) $$ (*i*,*0*) = *0* **if** *i*: *i* ∈ {*1*..<*m*} **for** *i*
proof (*cases i* ∈ *set ?nz*)
case *True*
show *?thesis*
by (*rule reduce-below-0-D0*[*OF A′ - - A′00 True*], *insert m1 n0 True A nz-not-empty*, *auto*)
next
case *False*
have (*reduce-below 0 ?nz 0 ?A′*) $$ (*i*,*0*) = *?A′* $$ (*i*,*0*)
by (*rule reduce-below-preserves-D0*[*OF A′ - - A′00 False*], *insert m1 n0 True A i nz-not-empty*, *auto*)
also have ... = *0* **using** *False n1 assms that* **by** *auto*
finally show *?thesis* **.**
qed
thus ∀ *i* ∈ {*1*..<*m*}. (*reduce-below 0 ?nz 0 ?A′*) $$ (*i*,*0*) = *0*
by *simp*
qed (*insert True*, *simp*)
have ∃ *P*. *invertible-mat P* ∧ *P* ∈ *carrier-mat m m* ∧ *reduce-below 0 ?nz 0 ?A′* = *P* ∗ *?A′*

by (*rule reduce-below-invertible-mat-D0*[*OF A'*], *insert m1 n0 True A*, *auto*)
moreover have ∃ *P*. *invertible-mat P* ∧ *P* ∈ *carrier-mat m m* ∧ *?A'* = *P* ∗ *A*
if *?nz* ≠ []
using *A A'-swaprows-invertible-mat m1 that* **by** *blast*
ultimately have *e-inv*: ∃ *P*. *invertible-mat P* ∧ *P* ∈ *carrier-mat m m* ∧ *reduce-below 0 ?nz 0 ?A'* = *P* ∗ *A*
if *?nz* ≠ []
by (*smt* (*verit*) *that A assoc-mult-mat invertible-mult-JNF mult-carrier-mat*)
have *e-r1*: *echelon-form-JNF A* **if** *nz-empty*: *?nz* = []
proof (*rule echelon-form-JNF-mx1*[*OF A*])
show ∀ *i*∈{*1..*<*m*}. *A* $$ (*i*, *0*) = *0* **using** *nz-empty*
by (*metis* (*mono-tags*, *lifting*) *A carrier-matD*(*1*) *empty-filter-conv set-upt*)
qed (*insert n1*, *simp*)
have *e-inv1*: ∃ *P*. *invertible-mat P* ∧ *P* ∈ *carrier-mat m m* ∧ *A* = *P* ∗ *A*
by (*metis A invertible-mat-one left-mult-one-mat one-carrier-mat*)
have *FindPreHNF-rectangular A* = (*if ?nz* = [] *then A else reduce-below-impl 0 ?nz 0 ?A'*)
unfolding *FindPreHNF-rectangular-def Let-def* **using** *m1 n1 A True* **by** *auto*
also have *reduce-below-impl 0 ?nz 0 ?A'* = *reduce-below 0 ?nz 0 ?A'*
by (*rule reduce-below-impl*[*OF - - - - A'*], *insert m1 n0 A*, *auto*)
finally show *?thesis* **using** *e-inv e-r e-r1 e-inv1* **by** *metis*
next
case *False*
have *f-rw*: *FindPreHNF-rectangular A* = *echelon-form-of-JNF A euclid-ext2*
unfolding *FindPreHNF-rectangular-def Let-def* **using** *m1 n0 A False* **by** *auto*
show *?thesis* **unfolding** *f-rw*
by (*rule echelon-form-of-euclidean-invertible'*[*OF A*], *insert False n0 m1*, *auto*)
qed
qed

lemma *HNF-algorithm-from-HA-soundness*:
assumes *A*: *A* ∈ *carrier-mat m n*
shows *Hermite-JNF* (*range ass-function-euclidean*) (λ*c*. *range* (*res-int c*)) (*HNF-algorithm-from-HA A*)
∧ (∃ *P*. *P* ∈ *carrier-mat m m* ∧ *invertible-mat P* ∧ (*HNF-algorithm-from-HA A*) = *P* ∗ *A*)
proof −
have *m*: *dim-row A* = *m* **using** *A* **by** *auto*
have (∃ *P*. *P* ∈ *carrier-mat m m* ∧ *invertible-mat P* ∧ (*HNF-algorithm-from-HA A*) = *P* ∗ (*FindPreHNF-rectangular A*))
unfolding *HNF-algorithm-from-HA-def m*
proof (*rule invertible-Hermite-of-list-of-rows*)
show *FindPreHNF-rectangular A* ∈ *carrier-mat m n*
by (*smt* (*verit*) *A FindPreHNF-rectangular-soundness mult-carrier-mat*)
show *echelon-form-JNF* (*FindPreHNF-rectangular A*)
using *FindPreHNF-rectangular-soundness* **by** *blast*
qed
moreover have (∃ *P*. *P* ∈ *carrier-mat m m* ∧ *invertible-mat P* ∧ (*FindPreHNF-rectangular A*) = *P* ∗ *A*)

by (*metis A FindPreHNF-rectangular-soundness*)
ultimately have (∃ *P*. *P* ∈ *carrier-mat m m* ∧ *invertible-mat P* ∧ (*HNF-algorithm-from-HA A*) = *P* ∗ *A*)
by (*smt* (*verit*) *assms assoc-mult-mat invertible-mult-JNF mult-carrier-mat*)
moreover have *Hermite-JNF* (*range ass-function-euclidean*) (λ*c*. *range* (*res-int c*)) (*HNF-algorithm-from-HA A*)
by (*metis A FindPreHNF-rectangular-soundness HNF-algorithm-from-HA-def m*
Hermite-Hermite-of-list-of-rows mult-carrier-mat)
ultimately show *?thesis* **by** *simp*
qed

Soundness theorem for any matrix

lemma *HNF-algorithm-soundness*:
assumes *A*: *A* ∈ *carrier-mat m n*
shows *Hermite-JNF* (*range ass-function-euclidean*) (λ*c*. *range* (*res-int c*)) (*HNF-algorithm abs-flag A*)
∧ (∃ *P*. *P* ∈ *carrier-mat m m* ∧ *invertible-mat P* ∧ (*HNF-algorithm abs-flag A*) = *P* ∗ *A*)
proof (*cases m*≠*n* ∨ *Determinant.det A* = *0*)
case *True*
have *H-rw*: *HNF-algorithm abs-flag A* = *Hermite-of-list-of-rows* (*FindPreHNF-rectangular A*) [*0*..<*m*]
using *True A* **unfolding** *HNF-algorithm-def Let-def* **by** *auto*
have (∃ *P*. *P* ∈ *carrier-mat m m* ∧ *invertible-mat P* ∧ (*HNF-algorithm abs-flag A*) = *P* ∗ (*FindPreHNF-rectangular A*))
unfolding *H-rw*
proof (*rule invertible-Hermite-of-list-of-rows*)
show *FindPreHNF-rectangular A* ∈ *carrier-mat m n*
by (*smt* (*verit*) *A FindPreHNF-rectangular-soundness mult-carrier-mat*)
show *echelon-form-JNF* (*FindPreHNF-rectangular A*)
using *FindPreHNF-rectangular-soundness* **by** *blast*
qed
moreover have (∃ *P*. *P* ∈ *carrier-mat m m* ∧ *invertible-mat P* ∧ (*FindPreHNF-rectangular A*) = *P* ∗ *A*)
by (*metis A FindPreHNF-rectangular-soundness*)
ultimately have (∃ *P*. *P* ∈ *carrier-mat m m* ∧ *invertible-mat P* ∧ (*HNF-algorithm abs-flag A*) = *P* ∗ *A*)
by (*smt* (*verit*) *assms assoc-mult-mat invertible-mult-JNF mult-carrier-mat*)
moreover have *Hermite-JNF* (*range ass-function-euclidean*) (λ*c*. *range* (*res-int c*)) (*HNF-algorithm abs-flag A*)
by (*metis A FindPreHNF-rectangular-soundness H-rw Hermite-Hermite-of-list-of-rows mult-carrier-mat*)
ultimately show *?thesis* **by** *simp*
next
case *False*
hence *mn*: *m*=*n* **and** *det-A-not0*:(*Determinant.det A*) ≠ *0* **by** *auto*
have *inv-RAT-A*: *invertible-mat* (*map-mat rat-of-int A*)
proof −

have *det* (*map-mat rat-of-int A*) ≠ *0* **using** *det-A-not0* **by** *auto*
thus *?thesis*
by (*metis False assms dvd-field-iff invertible-iff-is-unit-JNF map-carrier-mat*)
qed
have *HNF-algorithm abs-flag A = Hermite-mod-det abs-flag A*
unfolding *HNF-algorithm-def Hermite-mod-det-def Let-def* **using** *False A* **by** *simp*
then show *?thesis* **using** *Hermite-mod-det-soundness*[*OF mn A inv-RAT-A*] **by** *auto*
qed
end

New predicate of soundness of a HNF algorithm, without providing explicitly the transformation matrix.

definition *is-sound-HNF′ algorithm associates res*
= (∀ *A*. *let H = algorithm A*; *m = dim-row A*; *n = dim-col A in Hermite-JNF associates res H*
∧ *H* ∈ *carrier-mat m n* ∧ (∃ *P*. *P* ∈ *carrier-mat m m* ∧ *invertible-mat P* ∧ *A = P* ∗ *H*))

lemma *is-sound-HNF-conv*:
assumes *s*: *is-sound-HNF′ algorithm associates res*
shows *is-sound-HNF* (λ*A*. *let H = algorithm A in* (*SOME P*. *P* ∈ *carrier-mat* (*dim-row A*) (*dim-row A*)
∧ *invertible-mat P* ∧ *A = P* ∗ *H*, *H*)) *associates res*
proof (*unfold is-sound-HNF-def Let-def prod.case*, *rule allI*)
fix *A*::′*a mat*
define *m* **where** *m = dim-row A*
obtain *P* **where** *P*: *P* ∈ *carrier-mat m m* ∧ *invertible-mat P* ∧ *A = P* ∗ (*algorithm A*)
using *s* **unfolding** *is-sound-HNF′-def Let-def m-def* **by** *auto*
let *?some-P* = (*SOME P*. *P* ∈ *carrier-mat m m* ∧ *invertible-mat P* ∧ *A = P* ∗ *algorithm A*)
have *some-P*: *?some-P* ∈ *carrier-mat m m* ∧ *invertible-mat ?some-P* ∧ *A = ?some-P* ∗ *algorithm A*
by (*metis* (*mono-tags*, *lifting*) *P someI-ex*)
moreover have *algorithm A* ∈ *carrier-mat* (*dim-row A*) (*dim-col A*)
and *Hermite-JNF associates res* (*algorithm A*) **using** *s* **unfolding** *is-sound-HNF′-def Let-def* **by** *auto*
ultimately show *?some-P* ∈ *carrier-mat m m* ∧ *algorithm A* ∈ *carrier-mat m* (*dim-col A*)
∧ *invertible-mat ?some-P* ∧ *A = ?some-P* ∗ *algorithm A* ∧ *Hermite-JNF associates res* (*algorithm A*)
unfolding *is-sound-HNF-def Let-def m-def* **by** (*auto split*: *prod.split*)
qed

context *proper-mod-operation*
begin
corollary *is-sound-HNF′-HNF-algorithm*:

is-sound-HNF' (*HNF-algorithm abs-flag*) (*range ass-function-euclidean*) (λ*c*. *range* (*res-int c*))
proof −
have *Hermite-JNF* (*range ass-function-euclidean*) (λ*c*. *range* (*res-int c*)) (*HNF-algorithm abs-flag A*) **for** *A*
using *HNF-algorithm-soundness* **by** *blast*
moreover have *HNF-algorithm abs-flag A* ∈ *carrier-mat* (*dim-row A*) (*dim-col A*) **for** *A*
by (*metis HNF-algorithm-soundness carrier-matI mult-carrier-mat*)
moreover have ∃ *P*. *P* ∈ *carrier-mat* (*dim-row A*) (*dim-row A*) ∧ *invertible-mat P* ∧ *A* = *P* ∗ *HNF-algorithm abs-flag A* **for** *A*
proof −
have ∃ *P*. *P* ∈ *carrier-mat* (*dim-row A*) (*dim-row A*) ∧ *invertible-mat P* ∧ *HNF-algorithm abs-flag A* = *P* ∗ *A*
using *HNF-algorithm-soundness* **by** *blast*
from *this* **obtain** *P* **where** *P*: *P* ∈ *carrier-mat* (*dim-row A*) (*dim-row A*) **and** *inv-P*: *invertible-mat P*
and *H-PA*: *HNF-algorithm abs-flag A* = *P* ∗ *A* **by** *blast*
obtain *P'* **where** *PP'*: *inverts-mat P P'* **and** *P'P*: *inverts-mat P' P*
using *inv-P* **unfolding** *invertible-mat-def* **by** *auto*
have *P'*: *P'* ∈ *carrier-mat* (*dim-row A*) (*dim-row A*)
by (*metis P PP' P'P carrier-matD carrier-mat-triv index-mult-mat*(*3*) *index-one-mat*(*3*) *inverts-mat-def*)
moreover have *inv-P'*: *invertible-mat P'*
by (*metis P' P'P PP' carrier-matD*(*1*) *carrier-matD*(*2*) *invertible-mat-def square-mat.simps*)
moreover have *A* = *P'* ∗ *HNF-algorithm abs-flag A*
by (*smt* (*verit*) *H-PA P P'P assoc-mult-mat calculation*(*1*) *carrier-matD*(*1*) *carrier-matI inverts-mat-def left-mult-one-mat'*)
ultimately show *?thesis* **by** *auto*
qed
ultimately show *?thesis*
unfolding *is-sound-HNF'-def Let-def* **by** *auto*
qed

corollary *is-sound-HNF'-HNF-algorithm-from-HA*:
is-sound-HNF' (*HNF-algorithm-from-HA*) (*range ass-function-euclidean*) (λ*c*. *range* (*res-int c*))
proof −
have *Hermite-JNF* (*range ass-function-euclidean*) (λ*c*. *range* (*res-int c*)) (*HNF-algorithm-from-HA A*) **for** *A*
using *HNF-algorithm-from-HA-soundness* **by** *blast*
moreover have *HNF-algorithm-from-HA A* ∈ *carrier-mat* (*dim-row A*) (*dim-col A*) **for** *A*
by (*metis HNF-algorithm-from-HA-soundness carrier-matI mult-carrier-mat*)
moreover have ∃ *P*. *P* ∈ *carrier-mat* (*dim-row A*) (*dim-row A*) ∧ *invertible-mat P* ∧ *A* = *P* ∗ *HNF-algorithm-from-HA A* **for** *A*
proof −

have $\exists P.\ P \in$ *carrier-mat* (*dim-row A*) (*dim-row A*) $\wedge$ *invertible-mat P* $\wedge$ *HNF-algorithm-from-HA A* $= P *\ A$
using *HNF-algorithm-from-HA-soundness* **by** *blast*
from *this* **obtain** *P* **where** *P*: $P \in$ *carrier-mat* (*dim-row A*) (*dim-row A*) **and** *inv-P*: *invertible-mat P*
and *H-PA*: *HNF-algorithm-from-HA A* $= P *\ A$ **by** *blast*
obtain P' **where** *PP'*: *inverts-mat P P'* **and** *P'P*: *inverts-mat P' P*
using *inv-P* **unfolding** *invertible-mat-def* **by** *auto*
have *P'*: $P' \in$ *carrier-mat* (*dim-row A*) (*dim-row A*)
by (*metis P PP' P'P carrier-matD carrier-mat-triv index-mult-mat*(3) *index-one-mat*(3) *inverts-mat-def*)
moreover have *inv-P'*: *invertible-mat P'*
by (*metis P' P'P PP' carrier-matD*(1) *carrier-matD*(2) *invertible-mat-def square-mat.simps*)
moreover have *A* $= P' *$ *HNF-algorithm-from-HA A*
by (*smt* (*verit*) *H-PA P P'P assoc-mult-mat calculation*(1) *carrier-matD*(1) *carrier-matI inverts-mat-def left-mult-one-mat'*)
ultimately show *?thesis* **by** *auto*
qed
ultimately show *?thesis*
unfolding *is-sound-HNF'-def Let-def* **by** *auto*
qed
end

Some work to make the algorithm executable

definition *find-non0'* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *'a*::*comm-ring-1 mat* $\Rightarrow$ *nat option* **where**
find-non0' i k A = (*let is* = [*i* ..< *dim-row A*];
Ais = *filter* ($\lambda j.$ *A* $$ (*j*, *k*) $\neq$ *0*) *is*
in case Ais of [] $\Rightarrow$ *None* | - $\Rightarrow$ *Some* (*Ais*!*0*))

lemma *find-non0'*:
assumes *A*: *A* $\in$ *carrier-mat m n*
and *res*: *find-non0' i k A* = *Some j*
shows *A* $$ (*j*,*k*) $\neq$ *0 i* $\leq$ *j j* < *dim-row A*
proof −
let *?xs* = *filter* ($\lambda j.$ *A* $$ (*j*,*k*) $\neq$ *0*) [*i* ..< *dim-row A*]
from *res*[*unfolded find-non0'-def Let-def*]
have *xs*: *?xs* $\neq$ [] **by** (*cases ?xs*, *auto*)
have *j-in-xs*: *j* $\in$ *set ?xs* **using** *res* **unfolding** *find-non0'-def Let-def*
by (*metis* (*no-types*, *lifting*) *length-greater-0-conv list.case*(2) *list.exhaust nth-mem option.simps*(1) *xs*)
show *A* $$ (*j*,*k*) $\neq$ *0 i* $\leq$ *j j* < *dim-row A* **using** *j-in-xs* **by** *auto*+
qed

lemma *find-non0'-w-zero-before*:
assumes *A*: *A* $\in$ *carrier-mat m n*
and *res*: *find-non0' i k A* = *Some j*
shows $\forall j' \in \{i..<j\}.$ *A* $$ (*j'*,*k*) = *0*

proof −
let *?xs* = *filter* (λ*j*. *A* $$ (*j*, *k*) ≠ *0*) [*i* ..< *dim-row A*]
from *res*[*unfolded find-non0'-def Let-def*]
have *xs*: *?xs* ≠ [] **by** (*cases ?xs*, *auto*)
have *j-in-xs*: *j* ∈ *set ?xs* **using** *res* **unfolding** *find-non0'-def Let-def*
by (*metis* (*no-types*, *lifting*) *length-greater-0-conv list.case*(*2*) *list.exhaust nth-mem option.simps*(*1*) *xs*)
have *j-xs0*: *j* = *?xs* ! *0*
by (*smt* (*verit*) *res*[*unfolded find-non0'-def Let-def*] *list.case*(*2*) *list.exhaust option.inject xs*)
show ∀*j'*∈{*i*..<*j*}. *A* $$ (*j'*,*k*) = *0*
proof (*rule*+, *rule ccontr*)
fix *j'* **assume** *j'*: *j'* : {*i*..<*j*} **and** *Alj'*: *A* $$ (*j'*,*k*) ≠ *0*
have *j'j*: *j'*<*j* **using** *j'* **by** *auto*
have *j'-in-xs*: *j'* ∈ *set ?xs*
by (*metis* (*mono-tags*, *lifting*) *A Alj' Set.member-filter atLeastLessThan-iff filter-set*
find-non0'(*3*) *j' nat-SN.gt-trans res set-upt*)
have *l-rw*: [*i*..<*dim-row A*] = [*i* ..<*j*] @[*j*..<*dim-row A*]
using *assms*(*1*) *assms*(*2*) *find-non0'*(*3*) *j' upt-append*
by (*metis atLeastLessThan-iff le-trans linorder-not-le*)
have *xs-rw*: *?xs* = *filter* (λ*j*. *A* $$ (*j*,*k*) ≠ *0*) ([*i* ..<*j*] @[*j*..<*dim-row A*])
using *l-rw* **by** *auto*
hence *filter* (λ*j*. *A* $$ (*j*,*k*) ≠ *0*) [*i* ..<*j*] = [] **using** *j-xs0*
by (*metis* (*no-types*, *lifting*) *Set.member-filter atLeastLessThan-iff filter-append filter-set*
length-greater-0-conv nth-append nth-mem order-less-irrefl set-upt)
thus *False* **using** *j-xs0 j' j-xs0*
by (*metis Set.member-filter filter-empty-conv filter-set j'-in-xs set-upt*)
qed
qed

lemma *find-non0'-LEAST*:
assumes *A*: *A* ∈ *carrier-mat m n*
and *res*: *find-non0' i k A* = *Some j*
shows *j* = (*LEAST n*. *A* $$ (*n*,*k*) ≠ *0* ∧ *i*≤*n*)
proof (*rule Least-equality*[*symmetric*])
show *A* $$ (*j*, *k*) ≠ *0* ∧ *i* ≤ *j*
using *A res find-non0'*[*OF A*] **by** *auto*
show ⋀*y*. *A* $$ (*y*, *k*) ≠ *0* ∧ *i* ≤ *y* ⟹ *j* ≤ *y*
by (*meson A res atLeastLessThan-iff find-non0'-w-zero-before linorder-not-le*)
qed

lemma *echelon-form-of-column-k-JNF-code*[*code*]:
echelon-form-of-column-k-JNF bezout (*A*,*i*) *k* =
(*if* (*i* = *dim-row A*) ∨ (∀ *m* ∈ {*i*..<*dim-row A*}. *A* $$ (*m*, *k*) = *0*) *then* (*A*, *i*) *else*
if (∀ *m*∈{*i*+*1*..<*dim-row A*}. *A* $$ (*m*,*k*) = *0*) *then* (*A*, *i* + *1*) *else*

```
        let n = the (find-non0' i k A);
            interchange-A = swaprows i n A
        in
        (bezout-iterate-JNF (interchange-A) (dim-row A − 1) i k bezout, i + 1))
proof (cases ¬ ((i = dim-row A) ∨ (∀ m ∈ {i..<dim-row A}. A $$ (m, k) = 0))
        ∧ ¬ (∀ m∈{i+1..<dim-row A}. A $$ (m,k) = 0))
  case True
  let ?n = the (find-non0' i k A)
  let ?interchange-A = swaprows i ?n A
  have f-rw: (the (find-non0' i k A)) = (LEAST n. A $$ (n, k) ≠ 0 ∧ i ≤ n)
  proof (rule find-non0'-LEAST)
    have find-non0' i k A ≠ None using True unfolding find-non0'-def Let-def
      by (auto split: list.split)
      (metis (mono-tags, lifting) atLeastLessThan-iff atLeastLessThan-upt empty-filter-conv)
    thus find-non0' i k A = Some (the (find-non0' i k A)) by auto
  qed (auto)
  show ?thesis unfolding echelon-form-of-column-k-JNF-def Let-def f-rw using
True by auto
next
  case False
  then show ?thesis unfolding echelon-form-of-column-k-JNF-def by auto
qed
```

8.3 Instantiation of the HNF-algorithm with modulo-operation

We currently use a Boolean flag to indicate whether standard-mod or symmetric modulo should be used.

```
lemma sym-mod: proper-mod-operation sym-mod sym-div
  by (unfold-locales, auto simp: sym-mod-sym-div)

lemma standard-mod: proper-mod-operation (mod) (div)
  by (unfold-locales, auto, intro HOL.nitpick-unfold(7))

definition HNF-algorithm :: bool ⇒ int mat ⇒ int mat where
  HNF-algorithm use-sym-mod = (if use-sym-mod
   then mod-operation.HNF-algorithm sym-mod False else mod-operation.HNF-algorithm
(mod) True)

definition HNF-algorithm-from-HA :: bool ⇒ int mat ⇒ int mat where
  HNF-algorithm-from-HA use-sym-mod = (if use-sym-mod
   then mod-operation.HNF-algorithm-from-HA sym-mod else mod-operation.HNF-algorithm-from-HA
(mod))


corollary is-sound-HNF'-HNF-algorithm:
    is-sound-HNF' (HNF-algorithm use-sym-mod) (range ass-function-euclidean)
(λc. range (res-int c))
  using proper-mod-operation.is-sound-HNF'-HNF-algorithm[OF sym-mod]
    proper-mod-operation.is-sound-HNF'-HNF-algorithm[OF standard-mod]
```

unfolding *HNF-algorithm-def* **by** (*cases use-sym-mod*, *auto*)

corollary *is-sound-HNF'-HNF-algorithm-from-HA*:
is-sound-HNF' (*HNF-algorithm-from-HA use-sym-mod*) (*range ass-function-euclidean*)
(λ*c*. *range* (*res-int c*))
using *proper-mod-operation.is-sound-HNF'-HNF-algorithm-from-HA*[*OF sym-mod*]
proper-mod-operation.is-sound-HNF'-HNF-algorithm-from-HA[*OF standard-mod*]
unfolding *HNF-algorithm-from-HA-def* **by** (*cases use-sym-mod*, *auto*)

value [*code*]*let A* = *mat-of-rows-list 4* (
[[*0*,*3*,*1*,*4*],
[*7*,*1*,*0*,*0*],
[*8*,*0*,*19*,*16*],
[*2*,*0*,*0*,*3*::*int*],
[*9*,−*3*,*2*,*5*],
[*6*,*3*,*2*,*4*]]) *in*
show (*HNF-algorithm True A*)

value [*code*]*let A* = *mat-of-rows-list 6* (
[[*0*,*3*,*1*,*4*,*8*,*7*],
[*7*,*1*,*0*,*0*,*4*,*1*],
[*8*,*0*,*19*,*16*,*33*,*5*],
[*2*,*0*,*0*,*3*::*int*,−*5*,*8*]]) *in*
show (*HNF-algorithm False A*)

value [*code*]*let A* = *mat-of-rows-list 6* (
[[*0*,*3*,*1*,*4*,*8*,*7*],
[*7*,*1*,*0*,*0*,*4*,*1*],
[*8*,*0*,*19*,*16*,*33*,*5*],
[*0*,*3*,*1*,*4*,*8*,*7*],
[*2*,*0*,*0*,*3*::*int*,−*5*,*8*],
[*2*,*4*,*6*,*8*,*10*,*12*]]) *in*
show (*Determinant.det A*, *HNF-algorithm True A*)

value [*code*]*let A* = *mat-of-rows-list 6* (
[[*0*,*3*,*1*,*4*,*8*,*7*],
[*7*,*1*,*0*,*0*,*4*,*1*],
[*8*,*0*,*19*,*16*,*33*,*5*],
[*5*,*6*,*1*,*2*,*8*,*7*],
[*2*,*0*,*0*,*3*::*int*,−*5*,*8*],
[*2*,*4*,*6*,*8*,*10*,*12*]]) *in*
show (*Determinant.det A*, *HNF-algorithm True A*)

end

9 LLL certification via Hermite normal forms

In this file, we define the new certified approach and prove its soundness.

theory *LLL-Certification-via-HNF*
imports
LLL-Basis-Reduction.LLL-Certification
Jordan-Normal-Form.DL-Rank
HNF-Mod-Det-Soundness
begin

context *LLL-with-assms*
begin

lemma *m-le-n*: $m \leq n$
proof −
have *gs.lin-indpt* (*set* (*RAT fs-init*))
using *cof-vec-space.lin-indpt-list-def lin-dep* **by** *blast*
moreover have *gs.dim* = *n*
by (*simp add*: *gs.dim-is-n*)
moreover have *card* (*set* (*RAT fs-init*)) = *m*
using *LLL-invD*(*2*) *LLL-inv-initial-state cof-vec-space.lin-indpt-list-def distinct-card lin-dep*
by *blast*
ultimately show *?thesis* **using** *gs.li-le-dim*
by (*metis cof-vec-space.lin-indpt-list-def gs.fin-dim lin-dep*)
qed

end

This lemma is a generalization of the theorem named *HNF-A-eq-HNF-PA*, using the new uniqueness statement of the HNF. We provide two versions, one assuming the existence and the other one obtained from a sound algorithm.

lemma *HNF-A-eq-HNF-PA'-exist*:
fixes *A*::*int mat*
assumes *A*: *A* ∈ *carrier-mat n n* **and** *inv-A*: *invertible-mat* (*map-mat rat-of-int A*)
and *inv-P*: *invertible-mat P* **and** *P*: *P* ∈ *carrier-mat n n*
and *HNF-H1*: *Hermite-JNF associates res H1*
and *H1*: *H1* ∈ *carrier-mat n n*
and *HNF-H2*: *Hermite-JNF associates res H2*
and *H2*: *H2* ∈ *carrier-mat n n*
and *sound-HNF1*: ∃ *P1*. *P1* ∈ *carrier-mat n n* ∧ *invertible-mat P1* ∧ (*P* ∗ *A*) = *P1* ∗ *H1*

and *sound-HNF2*: $\exists P2.\ P2 \in carrier\text{-}mat\ n\ n \wedge invertible\text{-}mat\ P2 \wedge A = P2 * H2$
shows $H1 = H2$
proof −
obtain *inv-P* **where** *P-inv-P*: *inverts-mat P inv-P* **and** *inv-P-P*: *inverts-mat inv-P P*
and *inv-P*: $inv\text{-}P \in carrier\text{-}mat\ n\ n$
using *P inv-P obtain-inverse-matrix* **by** *blast*
obtain *P1* **where** *P1*: $P1 \in carrier\text{-}mat\ n\ n$ **and** *inv-P1*: *invertible-mat P1* **and** *P1-H1*: $P1 * A = P1 * H1$
using *sound-HNF1* **by** *auto*
obtain *P2* **where** *P2*: $P2 \in carrier\text{-}mat\ n\ n$ **and** *inv-P2*: *invertible-mat P2* **and** *P2-H2*: $A = P2 * H2$
using *sound-HNF2* **by** *auto*
have *invertible-inv-P*: *invertible-mat inv-P*
using *P-inv-P inv-P inv-P-P invertible-mat-def square-mat.simps* **by** *blast*
have *P-A-P1-H1*: $P * A = P1 * H1$ **using** *P1-H1 P2-H2* **unfolding** *is-sound-HNF-def Let-def*
by (*metis* (*mono-tags*, *lifting*) *case-prod-conv*)
hence $A = inv\text{-}P * (P1 * H1)$
by (*smt A P inv-P-P inv-P assoc-mult-mat carrier-matD*(*1*) *inverts-mat-def left-mult-one-mat*)
hence *A-inv-P-P1-H1*: $A = (inv\text{-}P * P1) * H1$ **using** *P P1-H1 assoc-mult-mat inv-P H1 P1* **by** *auto*
have *invertible-inv-P-P1*: *invertible-mat* $(inv\text{-}P * P1)$
by (*rule invertible-mult-JNF*[*OF inv-P P1 invertible-inv-P inv-P1*])
show *?thesis*
proof (*rule HNF-unique-generalized-JNF*[*OF A - H1 P2 H2 A-inv-P-P1-H1 P2-H2*
inv-A invertible-inv-P-P1 inv-P2 HNF-H1 HNF-H2])
show $inv\text{-}P * P1 \in carrier\text{-}mat\ n\ n$
by (*metis carrier-matD*(*1*) *carrier-matI index-mult-mat*(*2*) *inv-P invertible-inv-P-P1 invertible-mat-def square-mat.simps*)
qed
qed

corollary *HNF-A-eq-HNF-PA'*:
fixes *A*::*int mat*
assumes *A*: $A \in carrier\text{-}mat\ n\ n$ **and** *inv-A*: *invertible-mat* (*map-mat rat-of-int A*)
and *inv-P*: *invertible-mat P* **and** *P*: $P \in carrier\text{-}mat\ n\ n$
and *sound-HNF*: *is-sound-HNF HNF associates res*
and *P1-H1*: $(P1,H1) = HNF\ (P*A)$
and *P2-H2*: $(P2,H2) = HNF\ A$
shows $H1 = H2$
proof −
have *H1*: $H1 \in carrier\text{-}mat\ n\ n$
by (*smt P1-H1 A P carrier-matD index-mult-mat is-sound-HNF-def prod.sel*(*2*)

sound-HNF split-beta)
have *H2*: *H2* ∈ *carrier-mat n n*
by (*smt P2-H2 A carrier-matD index-mult-mat is-sound-HNF-def prod.sel*(*2*) *sound-HNF split-beta*)
have *HNF-H1*: *Hermite-JNF associates res H1*
by (*smt P1-H1 is-sound-HNF-def prod.sel*(*2*) *sound-HNF split-beta*)
have *HNF-H2*: *Hermite-JNF associates res H2*
by (*smt P2-H2 is-sound-HNF-def prod.sel*(*2*) *sound-HNF split-beta*)
have *sound-HNF1*: ∃ *P1*. *P1* ∈ *carrier-mat n n* ∧ *invertible-mat P1* ∧ (*P* ∗ *A*) = *P1* ∗ *H1*
using *sound-HNF P1-H1* **unfolding** *is-sound-HNF-def Let-def*
by (*metis* (*mono-tags*, *lifting*) *P carrier-matD*(*1*) *index-mult-mat*(*2*) *old.prod.simps*(*2*))
have *sound-HNF2*: ∃ *P2*. *P2* ∈ *carrier-mat n n* ∧ *invertible-mat P2* ∧ *A* = *P2* ∗ *H2*
using *sound-HNF P1-H1* **unfolding** *is-sound-HNF-def Let-def*
by (*metis* (*mono-tags*, *lifting*) *A P2-H2 carrier-matD*(*1*) *old.prod.simps*(*2*))
show *?thesis*
by (*rule HNF-A-eq-HNF-PA′-exist*[*OF A inv-A inv-P P HNF-H1 H1 HNF-H2 H2 sound-HNF1 sound-HNF2*])
qed

context *LLL-with-assms*
begin

lemma *certification-via-eq-HNF2-exist*:
assumes *HNF-H1*: *Hermite-JNF associates res H1*
and *H1*: *H1* ∈ *carrier-mat n n*
and *HNF-H2*: *Hermite-JNF associates res H2*
and *H2*: *H2* ∈ *carrier-mat n n*
and *sound-HNF1*: ∃ *P1*. *P1* ∈ *carrier-mat n n* ∧ *invertible-mat P1* ∧ (*mat-of-rows n fs-init*) = *P1* ∗ *H1*
and *sound-HNF2*: ∃ *P2*. *P2* ∈ *carrier-mat n n* ∧ *invertible-mat P2* ∧ (*mat-of-rows n gs*) = *P2* ∗ *H2*
and *gs*: *set gs* ⊆ *carrier-vec n*
and *l*: *lattice-of fs-init* = *lattice-of gs*
and *mn*: *m* = *n* **and** *len-gs*: *length gs* = *n*
shows *H1* = *H2*
proof −
have ∃ *P* ∈ *carrier-mat n n*. *invertible-mat P* ∧ *mat-of-rows n fs-init* = *P* ∗ *mat-of-rows n gs*
by (*rule eq-lattice-imp-mat-mult-invertible-rows*[*OF fs-init gs lin-dep len*[*unfolded mn*] *len-gs l*])
from *this* **obtain** *P* **where** *P*: *P* ∈ *carrier-mat n n* **and** *inv-P*: *invertible-mat P*
and *fs-P-gs*: *mat-of-rows n fs-init* = *P* ∗ *mat-of-rows n gs* **by** *auto*
obtain *P1* **where** *P1*: *P1* ∈ *carrier-mat n n* **and** *inv-P1*: *invertible-mat P1* **and** *P1-H1*: (*mat-of-rows n fs-init*) = *P1* ∗ *H1*
using *sound-HNF1* **by** *auto*

obtain *P2* **where** *P2*: *P2* ∈ *carrier-mat n n* **and** *inv-P2*: *invertible-mat P2* **and** *P2-H2*: (*mat-of-rows n gs*) = *P2* ∗ *H2*
using *sound-HNF2* **by** *auto*
have *P1-H1-2*: *P* ∗ *mat-of-rows n gs* = *P1* ∗ *H1*
using *P1-H1 fs-P-gs* **by** *auto*
have *gs-carrier*: *mat-of-rows n gs* ∈ *carrier-mat n n* **by** (*simp add*: *len-gs carrier-matI*)
show *?thesis*
proof (*rule HNF-A-eq-HNF-PA'-exist*[*OF gs-carrier - inv-P P HNF-H1 H1 HNF-H2 H2 - sound-HNF2*])
from *inv-P* **obtain** *P'* **where** *PP'*: *inverts-mat P P'* **and** *P'P*: *inverts-mat P' P*
using *invertible-mat-def* **by** *blast*
let *?RAT* = *of-int-hom.mat-hom* :: *int mat* ⇒ *rat mat*
have *det-RAT-fs-init*: *det* (*?RAT* (*mat-of-rows n fs-init*)) ≠ *0*
proof (*rule gs.lin-indpt-rows-imp-det-not-0*)
show *?RAT* (*mat-of-rows n fs-init*) ∈ *carrier-mat n n*
using *len map-carrier-mat mat-of-rows-carrier*(*1*) *mn* **by** *blast*
have *rw*: *Matrix.rows* (*?RAT* (*mat-of-rows n fs-init*)) = *RAT fs-init*
by (*metis cof-vec-space.lin-indpt-list-def fs-init lin-dep mat-of-rows-map rows-mat-of-rows*)
thus *gs.lin-indpt* (*set* (*Matrix.rows* (*?RAT* (*mat-of-rows n fs-init*))))
by (*insert lin-dep*, *simp add*: *cof-vec-space.lin-indpt-list-def*)
show *distinct* (*Matrix.rows* (*?RAT* (*mat-of-rows n fs-init*)))
using *rw cof-vec-space.lin-indpt-list-def lin-dep* **by** *auto*
qed
hence *d*: *det* (*?RAT* (*mat-of-rows n fs-init*)) *dvd 1* **using** *dvd-field-iff* **by** *blast*
hence *inv-RAT-fs-init*: *invertible-mat* (*?RAT* (*mat-of-rows n fs-init*))
using *invertible-iff-is-unit-JNF* **by** (*metis mn len map-carrier-mat mat-of-rows-carrier*(*1*))
have *invertible-mat* (*?RAT P*)
by (*metis P dvd-field-iff inv-P invertible-iff-is-unit-JNF map-carrier-mat not-is-unit-0 of-int-hom.hom-0 of-int-hom.hom-det*)
have *det* (*?RAT* (*mat-of-rows n fs-init*)) = *det* (*?RAT P*) ∗ *det* (*?RAT* (*mat-of-rows n gs*))
by (*metis Determinant.det-mult P fs-P-gs gs-carrier of-int-hom.hom-det of-int-hom.hom-mult*)
hence *det* (*?RAT* (*mat-of-rows n gs*)) ≠ *0* **using** *d* **by** *auto*
thus *invertible-mat* (*?RAT* (*mat-of-rows n gs*))
by (*meson dvd-field-iff gs-carrier invertible-iff-is-unit-JNF map-carrier-mat*)
show ∃ *P1*. *P1* ∈ *carrier-mat n n* ∧ *invertible-mat P1* ∧ *P* ∗ *mat-of-rows n gs* = *P1* ∗ *H1*
using *P1 P1-H1-2 inv-P1* **by** *blast*
qed
qed

lemma *certification-via-eq-HNF2*:
assumes *sound-HNF*: *is-sound-HNF HNF associates res*
and *P1-H1*: (*P1*,*H1*) = *HNF* (*mat-of-rows n fs-init*)
and *P2-H2*: (*P2*,*H2*) = *HNF* (*mat-of-rows n gs*)

```
    and gs: set gs ⊆ carrier-vec n
    and l: lattice-of fs-init = lattice-of gs
    and mn: m = n and len-gs: length gs = n
  shows H1 = H2
proof −
  have ∃ P ∈ carrier-mat n n. invertible-mat P ∧ mat-of-rows n fs-init = P ∗
mat-of-rows n gs
   by (rule eq-lattice-imp-mat-mult-invertible-rows[OF fs-init gs lin-dep len[unfolded
mn] len-gs l])
  from this obtain P where P: P ∈ carrier-mat n n and inv-P: invertible-mat P
    and fs-P-gs: mat-of-rows n fs-init = P ∗ mat-of-rows n gs by auto
  have P1-H1-2: (P1,H1) = HNF (P ∗ mat-of-rows n gs) using fs-P-gs P1-H1
by auto
  have gs-carrier: mat-of-rows n gs ∈ carrier-mat n n by (simp add: len-gs car-
rier-matI)
  show ?thesis
  proof (rule HNF-A-eq-HNF-PA'[OF gs-carrier - inv-P P sound-HNF P1-H1-2
P2-H2])
    from inv-P obtain P' where PP': inverts-mat P P' and P'P: inverts-mat P'
P
      using invertible-mat-def by blast
    let ?RAT = of-int-hom.mat-hom :: int mat ⇒ rat mat
    have det-RAT-fs-init: det (?RAT (mat-of-rows n fs-init)) ≠ 0
    proof (rule gs.lin-indpt-rows-imp-det-not-0)
      show ?RAT (mat-of-rows n fs-init) ∈ carrier-mat n n
        using len map-carrier-mat mat-of-rows-carrier(1) mn by blast
      have rw: Matrix.rows (?RAT (mat-of-rows n fs-init)) = RAT fs-init
          by (metis cof-vec-space.lin-indpt-list-def fs-init lin-dep mat-of-rows-map
rows-mat-of-rows)
      thus gs.lin-indpt (set (Matrix.rows (?RAT (mat-of-rows n fs-init))))
        by (insert lin-dep, simp add: cof-vec-space.lin-indpt-list-def)
      show distinct (Matrix.rows (?RAT (mat-of-rows n fs-init)))
        using rw cof-vec-space.lin-indpt-list-def lin-dep by auto
    qed
    hence d: det (?RAT (mat-of-rows n fs-init)) dvd 1 using dvd-field-iff by blast
    hence inv-RAT-fs-init: invertible-mat (?RAT (mat-of-rows n fs-init))
     using invertible-iff-is-unit-JNF by (metis mn len map-carrier-mat mat-of-rows-carrier(1))
    have invertible-mat (?RAT P)
      by (metis P dvd-field-iff inv-P invertible-iff-is-unit-JNF map-carrier-mat
          not-is-unit-0 of-int-hom.hom-0 of-int-hom.hom-det)
      have det (?RAT (mat-of-rows n fs-init)) = det (?RAT P) ∗ det (?RAT
(mat-of-rows n gs))
        by (metis Determinant.det-mult P fs-P-gs gs-carrier of-int-hom.hom-det
of-int-hom.hom-mult)
    hence det (?RAT (mat-of-rows n gs)) ≠ 0 using d by auto
    thus invertible-mat (?RAT (mat-of-rows n gs))
      by (meson dvd-field-iff gs-carrier invertible-iff-is-unit-JNF map-carrier-mat)
  qed
qed
```

corollary *lattice-of-eq-via-HNF*:
assumes *sound-HNF*: *is-sound-HNF HNF associates res*
and *P1-H1*: (*P1*,*H1*) = *HNF* (*mat-of-rows n fs-init*)
and *P2-H2*: (*P2*,*H2*) = *HNF* (*mat-of-rows n gs*)
and *gs*: *set gs* ⊆ *carrier-vec n*
and *mn*: *m* = *n* **and** *len-gs*: *length gs* = *n*
shows (*H1* = *H2*) ⟷ (*lattice-of fs-init* = *lattice-of gs*)
using *certification-via-eq-HNF certification-via-eq-HNF2 assms* **by** *metis*
end

context
begin

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

lemma *lattice-of-eq-via-HNF-paper*:
fixes *F G* :: *int mat* **and** *HNF* :: *int mat* ⇒ *int mat*
assumes *sound-HNF'*: *is-sound-HNF' HNF* $\mathcal{A}$ $\mathcal{R}$
and *inv-F-Q*: *invertible-mat* (*map-mat rat-of-int F*)
and *FG*: {*F*,*G*} ⊆ *carrier-mat n n*
shows (*HNF F* = *HNF G*) ⟷ (*lattice-of* (*rows F*) = *lattice-of* (*rows G*))
proof −
define *HNF'*
where *HNF'* = (λ*A*. *let H* = *HNF A*
in (*SOME P*. *P* ∈ *carrier-mat* (*dim-row A*) (*dim-row A*) ∧ *invertible-mat P* ∧
A = *P* ∗ *H*, *H*))
have *sound-HNF'*: *is-sound-HNF HNF'* $\mathcal{A}$ $\mathcal{R}$ **by** (*unfold HNF'-def*, *rule is-sound-HNF-conv*[*OF sound-HNF*'])
have *F-eq*: *F* = *mat-of-rows n* (*rows F*) **and** *G-eq*: *G* = *mat-of-rows n* (*rows G*)
using *FG* **by** *auto*
interpret *L*: *LLL-with-assms n n* (*rows F*) *4*/*3*
proof
interpret *gs*: *cof-vec-space n TYPE*(*rat*) **.**
thm *gs.upper-triangular-imp-lin-indpt-rows*
let *?RAT* =*map-mat rat-of-int*
have *m-rw*: (*map* (*map-vec rat-of-int*) (*rows F*)) = *rows* (*?RAT F*)
unfolding *Matrix.rows-def* **by** *auto*
show *gs.lin-indpt-list* (*map* (*map-vec rat-of-int*) (*rows F*))
proof −
have *det-RAT-F*: *det* (*?RAT F*) ≠ *0*
by (*metis inv-F-Q carrier-mat-triv invertible-iff-is-unit-JNF*
invertible-mat-def not-is-unit-0 square-mat.simps)
have *d-RAT-F*: *distinct* (*rows* (*?RAT F*))
proof (*rule ccontr*)
assume ¬ *distinct* (*rows* (*?RAT F*))

```
      from this obtain i j
        where ij: row (?RAT F) i = row (?RAT F) j
          and i: i<dim-row (?RAT F) and j: j<dim-row (?RAT F)
          and i-not-j: i≠j
        unfolding Matrix.rows-def distinct-conv-nth by auto
      have det (?RAT F) = 0 using ij i j i-not-j
     by (metis Determinant.det-def Determinant.det-identical-rows carrier-mat-triv)
      thus False using inv-F-Q
        by (metis carrier-mat-triv invertible-iff-is-unit-JNF invertible-mat-def
            not-is-unit-0 square-mat.simps)
    qed
    moreover have ¬ gs.lin-dep (set (rows (?RAT F)))
      using gs.det-not-0-imp-lin-indpt-rows[OF - det-RAT-F] using FG by auto
    ultimately show ?thesis
      unfolding gs.lin-indpt-list-def m-rw using FG unfolding Matrix.rows-def
by auto
  qed
 qed (insert FG F-eq, auto)
 show ?thesis
 proof (rule L.lattice-of-eq-via-HNF[OF sound-HNF'])
  show (fst (HNF' F), HNF F) = HNF' (mat-of-rows n (rows F))
    unfolding HNF'-def Let-def using F-eq by auto
  show (fst (HNF' G), HNF G) = HNF' (mat-of-rows n (rows G))
    unfolding HNF'-def Let-def using G-eq by auto
  show length (rows G) = n  using FG by auto
  show set (rows G) ⊆ carrier-vec n using FG
    by (metis G-eq mat-of-rows-carrier(3) rows-carrier)
 qed (simp)
qed
end
```

We define a new const similar to *external-lll-solver*, but now it only returns the reduced matrix.

```
consts external-lll-solver' :: integer × integer ⇒ integer list list ⇒ integer list list
```

```
hide-type (open) Finite-Cartesian-Product.vec
```

The following definition is an adaptation of *reduce-basis-external*

```
definition reduce-basis-external' :: (int mat ⇒ int mat) ⇒ rat ⇒ int vec list ⇒
int vec list where
 reduce-basis-external' HNF 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;
  gsi = external-lll-solver' (map-prod integer-of-int integer-of-int (quotient-of α))
fsi;
   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
      let Fs = mat-of-rows n fs;
          Gs = mat-of-rows n gs;
          H1 = HNF Fs;
          H2 = HNF Gs in
          if (H1 = H2) then rb gs
           else Code.abort (STR ''the reduced matrix does not span the same lat-
tice↩f,g,P1,P2,H1,H2 are as follows↩''
           + String.implode (show Fs) + STR ''↩|↩''
           + String.implode (show Gs) + STR ''↩|↩''
           + String.implode (show H1) + STR ''↩|↩''
           + String.implode (show H2) + STR ''↩|↩''
           ) (λ -. rb fs))
   )

locale certification = LLL-with-assms +
  fixes HNF::int mat ⇒ int mat and associates res
  assumes sound-HNF': is-sound-HNF' HNF associates res
begin

lemma reduce-basis-external': assumes res: reduce-basis-external' HNF α fs-init
= fs
  shows reduced fs m LLL-invariant True m fs
proof (atomize(full), goal-cases)
  case 1
  show ?case
  proof (cases LLL-Impl.reduce-basis α fs-init = fs)
    case True
    from reduce-basis[OF this] show ?thesis by simp
  next
    case False note a = False
    show ?thesis
    proof (cases fs-init)
      case Nil
      with res have fs = [] unfolding reduce-basis-external'-def by auto
      with False Nil have False by (simp add: LLL-Impl.reduce-basis-def)
      thus ?thesis ..
    next
      case (Cons f rest)
      from Cons fs-init len have dim-fs-n: dim-vec f = n by auto
    let ?ext = external-lll-solver' (map-prod integer-of-int integer-of-int (quotient-of
α))
       (map (map integer-of-int ∘ list-of-vec) fs-init)
        note res = res[unfolded reduce-basis-external'-def Cons Let-def list.case
Code.abort-def dim-fs-n,
```

```
        folded Cons]
      define gs where gs = map (vec-of-list o map int-of-integer) ?ext
      define Fs where Fs = mat-of-rows n fs-init
      define Gs where Gs = mat-of-rows n gs
      define H1 where H1 = HNF Fs
      define H2 where H2 = HNF Gs
      note res = res[unfolded ext option.simps split len dim-fs-n, folded gs-def]
     from res False have not: (¬ (length gs = m ∧ (∀ gi∈set gs. dim-vec gi = n)))
= False
       by (auto split: if-splits)
      note res = res[unfolded this if-False]
      from not have gs: set gs ⊆ carrier-vec n
        and len-gs: length gs = m by auto
      show ?thesis
      proof (cases H1 = H2)
        case True
        hence H1-eq-H2: H1 = H2 by auto
       let ?HNF = (λA. let H = HNF A in (SOME P. P ∈ carrier-mat (dim-row
A) (dim-row A) ∧ invertible-mat P ∧ A = P * H, H))
        obtain P1 where P1-H1: (P1,H1) = ?HNF Fs by (metis H1-def)
        obtain P2 where P2-H2: (P2,H2) = ?HNF Gs by (metis H2-def)
        have sound-HNF: is-sound-HNF ?HNF associates res
          by (rule is-sound-HNF-conv[OF sound-HNF'])
        have laticce-gs-fs-init: lattice-of gs = lattice-of fs-init
          and gs-assms: LLL-with-assms n m gs α
         by (rule certification-via-eq-HNF[OF sound-HNF P1-H1[unfolded Fs-def]
               P2-H2[unfolded Gs-def] H1-eq-H2 gs len-gs])+
        from res a True
        have gs-fs: LLL-Impl.reduce-basis α gs = fs by (auto split: prod.split)
        have lattice-gs-fs: lattice-of gs = lattice-of fs
          and gram-schmidt-fs.reduced n (map of-int-hom.vec-hom fs) α m
          and gs.lin-indpt-list (map of-int-hom.vec-hom fs)
          and length fs = length gs
         using LLL-with-assms.reduce-basis gs-fs gs-assms laticce-gs-fs-init gs-assms

         using LLL-with-assms-def len-gs unfolding LLL.L-def by fast+
        from this show ?thesis
          using laticce-gs-fs-init gs-assms LLL-with-assms-def lattice-gs-fs
          unfolding LLL-invariant-def L-def by auto
      next
        case False
        then show ?thesis
          using a Fs-def Gs-def res H1-def H2-def by auto
      qed
    qed
  qed
qed
end
```

context *LLL-with-assms*
begin

We interpret the certification context using our formalized *HNF-algorithm*

interpretation *efficient-cert*: *certification n m fs-init α HNF-algorithm use-sym-mod range ass-function-euclidean λc. range* (*res-int c*)
by (*unfold-locales*, *rule is-sound-HNF'-HNF-algorithm*)

thm *efficient-cert.reduce-basis-external'*

Same, but applying the naive HNF algorithm, moved to JNF library from the echelon form and Hermite normal form AFP entries

interpretation *cert*: *certification n m fs-init α HNF-algorithm-from-HA use-sym-mod range ass-function-euclidean λc. range* (*res-int c*)
by (*unfold-locales*, *rule is-sound-HNF'-HNF-algorithm-from-HA*)
thm *cert.reduce-basis-external'*

lemma *RBE-HNF-algorithm-efficient*:
assumes *reduce-basis-external'* (*HNF-algorithm use-sym-mod*) *α fs-init* = *fs*
shows *gram-schmidt-fs.reduced n* (*map of-int-hom.vec-hom fs*) *α m*
and *LLL-invariant True m fs* **using** *efficient-cert.reduce-basis-external' assms*
by *blast+*

lemma *RBE-HNF-algorithm-naive*:
assumes *reduce-basis-external'* (*HNF-algorithm-from-HA use-sym-mod*) *α fs-init* = *fs*
shows *gram-schmidt-fs.reduced n* (*map of-int-hom.vec-hom fs*) *α m*
and *LLL-invariant True m fs* **using** *cert.reduce-basis-external' assms* **by** *blast+*

end

lemma *external-lll-solver'-code*[*code*]:
external-lll-solver' = *Code.abort* (*STR ''require proper implementation of external-lll-solver''*) (*λ -. external-lll-solver'*)
by *simp*
end