Echelon Form

By Jose Divasón and Jesús Aransay*

March 19, 2025

Abstract

In this work we present the formalization of an algorithm to compute the Echelon Form of a matrix. We have proved its existence over Bezout domains and we have made it executable over Euclidean domains, such as $\mathbb{Z}$ and $\mathbb{K}[x]$. This allows us to compute determinants, inverses and characteristic polynomials of matrices. The work is based on the *HOL-Multivariate Analysis* library, and on both the Gauss-Jordan and Cayley-Hamilton AFP entries. As a by-product, some algebraic structures have been implemented (principal ideal domains, Bezout domains...). The algorithm has been refined to immutable arrays and code can be generated to functional languages as well.

Contents

*This research has been funded by the research grant FPI-UR-12 of the Universidad de La Rioja and by the project MTM2014-54151-P from Ministerio de Economía y Competitividad (Gobierno de España).

1 Rings

theory *Rings2*
imports
  HOL−Analysis.Analysis
  HOL−Computational-Algebra.Polynomial-Factorial
begin

1.1 Previous lemmas and results

lemma *chain-le*:
  fixes I::*nat* => *'a set*
  assumes *inc*: $\forall n.\ I(n) \subseteq I(n+1)$
  shows $\forall n \le m.\ I(n) \subseteq I(m)$
  using *assms*
proof (*induct m*)
  case *0*
  show *?case* **by** *auto*
next
  case (*Suc m*)
  show *?case* **by** (*metis Suc-eq-plus1 inc lift-Suc-mono-le*)
qed

context *Rings.ring*
begin

lemma *sum-add*:
  assumes *A*: *finite A*

and B: *finite B*
shows *sum f A* + *sum g B* = *sum f* $(A - B)$ + *sum g* $(B - A)$ + *sum* $(\lambda x.\ f\ x + g\ x)$ $(A \cap B)$
proof −
have *1*: *sum f A* = *sum f* $(A - B)$ + *sum f* $(A \cap B)$
by (*metis A Int-Diff-disjoint Un-Diff-Int finite-Diff finite-Int inf-sup-aci*(*1*) *local.sum.union-disjoint*)
have *2*: *sum g B* = *sum g* $(B - A)$ + *sum g* $(A \cap B)$
by (*metis B Int-Diff-disjoint Int-commute Un-Diff-Int finite-Diff finite-Int local.sum.union-disjoint*)
have *3*: *sum f* $(A \cap B)$ + *sum g* $(A \cap B)$ = *sum* $(\lambda x.\ f\ x + g\ x)$ $(A \cap B)$
by (*simp add*: *sum.distrib*)
show *?thesis*
by (*simp add*: *1 2 3 add.assoc add.left-commute*)
qed

This lemma is presented in the library but for additive abelian groups

lemma *sum-negf*:
sum (%x. − ($f\ x$)::$'a$) A = − *sum f A*
proof (*cases finite A*)
case *True* **thus** *?thesis* **by** (*induct set*: *finite*) *auto*
next
case *False* **thus** *?thesis* **by** *simp*
qed

The following lemmas are presented in the library but for other type classes (semiring_0)

lemma *sum-distrib-left*:
shows $r * $ *sum f A* = *sum* (%n. $r * f\ n$) A
proof (*cases finite A*)
case *True*
thus *?thesis*
proof *induct*
case *empty* **thus** *?case* **by** *simp*
next
case (*insert x A*) **thus** *?case* **by** (*simp add*: *distrib-left*)
qed
next
case *False* **thus** *?thesis* **by** *simp*
qed

lemma *sum-distrib-right*:
sum f A $* r = (\sum n \in A.\ f\ n * r)$
proof (*cases finite A*)
case *True*
then show *?thesis*
proof *induct*
case *empty* **thus** *?case* **by** *simp*
next

case (*insert x A*) **thus** *?case* **by** (*simp add*: *distrib-right*)
qed
next
case *False* **thus** *?thesis* **by** *simp*
qed

end

context *comm-monoid-add*
begin

lemma *sum-two-elements*:
assumes $a \neq b$
shows *sum f* $\{a,b\} = f\ a + f\ b$
by (*metis Diff-cancel assms empty-Diff finite.emptyI infinite-remove add-0-right sum.empty sum.insert sum.insert-remove singletonD*)

lemma *sum-singleton*: *sum f* $\{x\} = f\ x$
by *simp*

end

1.2 Subgroups

context *group-add*
begin
definition *subgroup* $A \equiv (0 \in A \wedge (\forall a{\in}A.\ \forall b \in A.\ a + b \in A) \wedge (\forall a{\in}A.\ -a \in A))$

lemma *subgroup-0*: *subgroup* $\{0\}$
unfolding *subgroup-def* **by** *auto*

lemma *subgroup-UNIV*: *subgroup* (*UNIV*)
unfolding *subgroup-def* **by** *auto*

lemma *subgroup-inter*:
assumes *subgroup A* **and** *subgroup B*
shows *subgroup* $(A \cap B)$
using *assms* **unfolding** *subgroup-def* **by** *blast*

lemma *subgroup-Inter*:
assumes $\forall I{\in}S.$ *subgroup I*
shows *subgroup* $(\bigcap S)$
using *assms* **unfolding** *subgroup-def* **by** *auto*

lemma *subgroup-Union*:
fixes I::*nat* => *'a set*
defines S: $S \equiv \{I\ n | n.\ n \in UNIV\}$

```
  assumes all-subgroup: ∀ A∈S. subgroup A
  and inc: ∀ n. I(n) ⊆ I(n+1)
  shows subgroup (⋃ S)
unfolding subgroup-def
proof (safe; (unfold Union-iff)?)
  show ∃ X∈S. 0 ∈ X
  proof (rule bexI[of - I 0])
    show I 0 ∈ S unfolding S by auto
    thus 0 ∈ I 0 using all-subgroup unfolding subgroup-def by auto
  qed
  fix y a ya b assume y: y ∈ S and a: a ∈ y and ya: ya ∈ S and b: b ∈ ya
  obtain n m where In: y=I n and Im: ya = I m using y ya S by auto
  have In-I-max:I n ⊆ I (max n m) using chain-le[OF inc] by auto
  have Im-I-max:I m ⊆ I (max n m) using chain-le[OF inc] by auto
  show ∃ x∈S. a + b ∈ x
  proof (rule bexI[of - I (max n m)])
    show a + b ∈ I (max n m)
      by (metis Im Im-I-max In In-I-max a all-subgroup b in-mono max-def sub-
group-def y ya)
    show I (max n m) ∈ S using S by auto
  qed
  show ∃ x∈S. − a ∈ x
  proof (rule bexI[of - I (max n m)])
    show − a ∈ I (max n m) by (metis In In-I-max a all-subgroup in-mono
subgroup-def y)
    show I (max n m) ∈ S using S by auto
  qed
qed


end
```

1.3 Ideals

```
context Rings.ring
begin

lemma subgroup-left-principal-ideal: subgroup {r∗a|r. r∈UNIV}
proof (unfold subgroup-def, auto)
  show ∃ r. 0 = r ∗ a by (rule exI[of - 0], simp)
  fix r ra show ∃ rb. r ∗ a + ra ∗ a = rb ∗ a
    by (metis add-0-right combine-common-factor)
  show ∃ ra. − (r ∗ a) = ra ∗ a by (metis minus-mult-left)
qed


definition left-ideal I = (subgroup I ∧ (∀ x∈I. ∀ r. r∗x ∈ I))
definition right-ideal I = (subgroup I ∧ (∀ x∈I. ∀ r. x∗r ∈ I))
```

definition *ideal I*= (*left-ideal I* ∧ *right-ideal I*)

definition *left-ideal-generated S* = $\bigcap${*I*. *left-ideal I* ∧ *S* ⊆ *I*}
definition *right-ideal-generated S* = $\bigcap${*I*. *right-ideal I* ∧ *S* ⊆ *I*}
definition *ideal-generated S* = $\bigcap${*I*. *ideal I* ∧ *S* ⊆ *I*}

definition *left-principal-ideal S* = (∃ *a*. *left-ideal-generated* {*a*} = *S*)
definition *right-principal-ideal S* = (*right-ideal S* ∧ (∃ *a*. *right-ideal-generated* {*a*} = *S*))
definition *principal-ideal S* = (∃ *a*. *ideal-generated* {*a*} = *S*)

lemma *ideal-inter*:
  assumes *ideal I* **and** *ideal J* **shows** *ideal* (*I* ∩ *J*)
  using *assms*
  unfolding *ideal-def left-ideal-def right-ideal-def subgroup-def*
  by *auto*

lemma *ideal-Inter*:
  assumes ∀ *I*∈*S*. *ideal I*
  shows *ideal* ($\bigcap$ *S*)
proof (*unfold ideal-def left-ideal-def right-ideal-def*, *auto*)
  show *subgroup* ($\bigcap$ *S*) **and** *subgroup* ($\bigcap$ *S*)
    using *subgroup-Inter assms*
    unfolding *ideal-def left-ideal-def* **by** *auto*
  fix *x r xa* **assume** *X*: ∀ *X*∈*S*. *x* ∈ *X* **and** *xa*: *xa* ∈ *S*
  show *r* ∗ *x* ∈ *xa* **by** (*metis X assms ideal-def left-ideal-def xa*)
next
  fix *x r xa* **assume** *X*: ∀ *X*∈*S*. *x* ∈ *X* **and** *xa*: *xa* ∈ *S*
  show *x* ∗ *r* ∈ *xa* **by** (*metis X assms ideal-def right-ideal-def xa*)
qed

lemma *ideal-Union*:
  fixes *I*::*nat* => *'a set*
  defines *S*: *S*≡{*I n*|*n*. *n*∈*UNIV*}
  assumes *all-ideal*: ∀ *A*∈*S*. *ideal A*
  and *inc*: ∀ *n*. *I*(*n*) ⊆ *I*(*n*+*1*)
  shows *ideal* ($\bigcup$ *S*)
unfolding *ideal-def left-ideal-def right-ideal-def*
proof (*safe*; (*unfold Union-iff*)?)
  fix *y x r*
  assume *y*: *y* ∈ *S* **and** *x*: *x* ∈ *y*
  obtain *n* **where** *n*: *y*=*I n* **using** *y S* **by** *auto*
  show ∃ *xa*∈*S*. *r* ∗ *x* ∈ *xa*
  proof (*rule bexI*[*of - I n*])
    show *r* ∗ *x* ∈ *I n* **by** (*metis n assms*(*2*) *ideal-def left-ideal-def x y*)
    show *I n* ∈ *S* **by** (*metis n y*)
  qed
  show ∃ *xa*∈*S*. *x* ∗ *r* ∈ *xa*

proof (*rule bexI*[*of* - *I n*])
show $x * r \in I\ n$ **by** (*metis n assms*(*2*) *ideal-def right-ideal-def x y*)
show $I\ n \in S$ **by** (*metis n y*)
qed
next
show *subgroup* ($\bigcup S$) **and** *subgroup* ($\bigcup S$)
using *subgroup-Union*
by (*metis* (*mono-tags*) *S all-ideal ideal-def inc right-ideal-def*)+
qed

lemma *ideal-not-empty*:
assumes *ideal I*
shows $I \neq \{\}$
using *assms* **unfolding** *ideal-def left-ideal-def subgroup-def* **by** *auto*

lemma *ideal-0*: *ideal* $\{0\}$
unfolding *ideal-def left-ideal-def right-ideal-def* **using** *subgroup-0* **by** *auto*

lemma *ideal-UNIV*: *ideal UNIV*
unfolding *ideal-def left-ideal-def right-ideal-def* **using** *subgroup-UNIV* **by** *auto*

lemma *ideal-generated-0*: *ideal-generated* $\{0\} = \{0\}$
unfolding *ideal-generated-def* **using** *ideal-0* **by** *auto*

lemma *ideal-generated-subset-generator*:
assumes *ideal-generated* $A = I$
shows $A \subseteq I$
using *assms* **unfolding** *ideal-generated-def* **by** *auto*

lemma *left-ideal-minus*:
assumes *left-ideal I*
and $a \in I$ **and** $b \in I$
shows $a - b \in I$
by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *diff-minus-eq-add left-ideal-def minus-minus subgroup-def*)

lemma *right-ideal-minus*:
assumes *right-ideal I*
and $a \in I$ **and** $b \in I$
shows $a - b \in I$
by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *diff-minus-eq-add minus-minus right-ideal-def subgroup-def*)

lemma *ideal-minus*:
assumes *ideal I*
and $a \in I$ **and** $b \in I$
shows $a - b \in I$

by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *ideal-def right-ideal-minus*)

lemma *ideal-ideal-generated*: *ideal* (*ideal-generated S*)
unfolding *ideal-generated-def*
unfolding *ideal-def left-ideal-def subgroup-def right-ideal-def*
by *blast*

lemma *sum-left-ideal*:
assumes *li-X*: *left-ideal X*
and *U-X*: $U \subseteq X$ **and** *U*: *finite U*
shows $(\sum i \in U.\ f\ i * i) \in X$
using *U U-X*
proof (*induct U*)
case *empty* **show** *?case* **using** *li-X* **by** (*simp add*: *left-ideal-def subgroup-def*)
next
case (*insert x U*)
have *x-in-X*: $x \in X$ **using** *insert.prems* **by** *simp*
have *fx-x-X*: $f\ x * x \in X$ **using** *li-X x-in-X* **unfolding** *left-ideal-def* **by** *simp*
have *sum-in-X*: $(\sum i \in U.\ f\ i * i) \in X$ **using** *insert.prems insert.hyps*(*3*) **by** *simp*
have $(\sum i \in (insert\ x\ U).\ f\ i * i) = f\ x * x + (\sum i \in U.\ f\ i * i)$
by (*simp add*: *insert.hyps*(*1*) *insert.hyps*(*2*))
also have ... $\in X$ **using** *li-X fx-x-X sum-in-X* **unfolding** *left-ideal-def subgroup-def* **by** *auto*
finally show $(\sum i \in (insert\ x\ U).\ f\ i * i) \in X$ **.**
qed

lemma *sum-right-ideal*:
assumes *li-X*: *right-ideal X*
and *U-X*: $U \subseteq X$ **and** *U*: *finite U*
shows $(\sum i \in U.\ i * f\ i) \in X$
using *U U-X*
proof (*induct U*)
case *empty* **show** *?case* **using** *li-X* **by** (*simp add*: *right-ideal-def subgroup-def*)
next
case (*insert x U*)
have *x-in-X*: $x \in X$ **using** *insert.prems* **by** *simp*
have *fx-x-X*: $x * f\ x \in X$ **using** *li-X x-in-X* **unfolding** *right-ideal-def* **by** *simp*
have *sum-in-X*: $(\sum i \in U.\ i * f\ i) \in X$ **using** *insert.prems insert.hyps*(*3*) **by** *simp*
have $(\sum i \in (insert\ x\ U).\ i * f\ i) = x * f\ x + (\sum i \in U.\ i * f\ i)$
by (*simp add*: *insert.hyps*(*1*) *insert.hyps*(*2*))
also have ... $\in X$ **using** *li-X fx-x-X sum-in-X* **unfolding** *right-ideal-def subgroup-def* **by** *auto*
finally show $(\sum i \in (insert\ x\ U).\ i * f\ i) \in X$ **.**
qed

lemma *left-ideal-generated-subset*:
assumes $S \subseteq T$
shows *left-ideal-generated S* $\subseteq$ *left-ideal-generated T*

unfolding *left-ideal-generated-def* **using** *assms* **by** *auto*

lemma *right-ideal-generated-subset*:
assumes $S \subseteq T$
shows *right-ideal-generated* $S \subseteq$ *right-ideal-generated* T
unfolding *right-ideal-generated-def* **using** *assms* **by** *auto*

lemma *ideal-generated-subset*:
assumes $S \subseteq T$
shows *ideal-generated* $S \subseteq$ *ideal-generated* T
unfolding *ideal-generated-def* **using** *assms* **by** *auto*

lemma *ideal-generated-in*:
assumes $a \in A$
shows $a \in$ *ideal-generated* A
unfolding *ideal-generated-def* **using** *assms* **by** *auto*

lemma *ideal-generated-repeated*: *ideal-generated* $\{a,a\}$ = *ideal-generated* $\{a\}$
unfolding *ideal-generated-def* **by** *auto*

end

context *ring-1*
begin

lemma *left-ideal-explicit*:
left-ideal-generated $S = \{y.\ \exists f\ U.\ \textit{finite}\ U \wedge U \subseteq S \wedge \textit{sum}\ (\lambda i.\ f\ i * i)\ U = y\}$
(**is** *?S = ?B*)
proof
have *S-in-B*: $S \subseteq$ *?B*
proof (*auto*)
fix x **assume** x: $x \in S$
show $\exists f\ U.\ \textit{finite}\ U \wedge U \subseteq S \wedge (\sum i \in U.\ f\ i * i) = x$
by (*rule exI*[*of* - $\lambda i.\ 1$], *rule exI*[*of* - $\{x\}$], *simp add*: x)
qed
have *left-ideal-B*: *left-ideal ?B*
proof (*unfold left-ideal-def*, *auto*)
show *subgroup ?B*
proof (*unfold subgroup-def*, *auto*)
show $\exists f\ U.\ \textit{finite}\ U \wedge U \subseteq S \wedge (\sum i \in U.\ f\ i * i) = 0$
by (*rule exI*[*of* - *id*], *rule exI*[*of* - $\{\}$], *auto*)
fix f A **assume** A: *finite* A **and** AS: $A \subseteq S$
show$\exists fa\ Ua.\ \textit{finite}\ Ua \wedge Ua \subseteq S \wedge (\sum i \in Ua.\ fa\ i * i) = - (\sum i \in A.\ f\ i * i)$
by (*rule exI*[*of* - $\lambda i.\ - f\ i$], *rule exI*[*of* - A],
auto simp add: A AS *sum-negf*[*of* $\lambda i.\ f\ i * i\ A$])
fix fa B **assume** B: *finite* B **and** BS: $B \subseteq S$
let $?g = \lambda i.\ \textit{if}\ i \in A - B\ \textit{then}\ f\ i\ \textit{else}\ \textit{if}\ i \in B - A\ \textit{then}\ fa\ i\ \textit{else}\ f\ i + fa\ i$
show $\exists fb\ Ub.\ \textit{finite}\ Ub \wedge Ub \subseteq S \wedge (\sum i \in Ub.\ fb\ i * i)$
$= (\sum i \in A.\ f\ i * i) + (\sum i \in B.\ fa\ i * i)$

proof (*rule exI*[*of* - *?g*], *rule exI*[*of* - $A \cup B$], *simp add*: *A B AS BS*)
let *?g2* = (λi. (*if* $i \in A \wedge i \notin B$ *then* *f i else*
if $i \in B - A$ *then fa i else f i* + *fa i*) $*$ *i*)
have ($\sum i \in A$. *f i* $*$ *i*) + ($\sum i \in B$. *fa i* $*$ *i*)
= ($\sum i \in A - B$. *f i* $*$ *i*) + ($\sum i \in B - A$. *fa i* $*$ *i*) + ($\sum i \in A \cap B$. (*f i* $*$ *i*) + (*fa i* $*$ *i*))
by (*rule sum-add*[*OF A B*])
also have ... = ($\sum i \in A - B$. *f i* $*$ *i*) + ($\sum i \in B - A$. *fa i* $*$ *i*)
+ ($\sum i \in A \cap B$. (*f i* + *fa i*) $*$ *i*)
by (*simp add*: *distrib-right*)
also have ... = *sum ?g2* ($A - B$) + *sum ?g2* ($B - A$) + *sum ?g2* ($A \cap B$) **by** *auto*
also have ... = *sum ?g2* ($A \cup B$) **by** (*rule sum.union-diff2*[*OF A B*, *symmetric*])
finally show *sum ?g2* ($A \cup B$) = ($\sum i \in A$. *f i* $*$ *i*) + ($\sum i \in B$. *fa i* $*$ *i*) **..**
qed
qed
fix *f U r* **assume** *U*: *finite U* **and** *U-in-S*: $U \subseteq S$
show $\exists$*fa Ua*. *finite Ua* $\wedge$ *Ua* $\subseteq$ *S* $\wedge$ ($\sum i \in Ua$. *fa i* $*$ *i*) = *r* $*$ ($\sum i \in U$. *f i* $*$ *i*)
by (*rule exI*[*of* - λi. *r* $*$ *f i*], *rule exI*[*of* - *U*])
(*simp add*: *U U-in-S sum-distrib-left mult-assoc*)
qed
thus *?S* $\subseteq$ *?B* **using** *S-in-B* **unfolding** *left-ideal-generated-def* **by** *auto*
next
show *?B* $\subseteq$ *?S*
proof (*unfold left-ideal-generated-def*, *auto*)
fix *X f U*
assume *li-X*: *left-ideal X* **and** *S-X*: $S \subseteq X$ **and** *U*: *finite U* **and** *U-in-S*: $U \subseteq S$
have *U-in-X*: $U \subseteq X$ **using** *U-in-S S-X* **by** *simp*
show ($\sum i \in U$. *f i* $*$ *i*) $\in$ *X*
by (*rule sum-left-ideal*[*OF li-X U-in-X U*])
qed
qed

lemma *right-ideal-explicit*:
right-ideal-generated S = {*y*. $\exists$*f U*. *finite U* $\wedge$ $U \subseteq S$ $\wedge$ *sum* (λi. *i* $*$ *f i*) *U* = *y*} (**is** *?S* = *?B*)
proof
have *S-in-B*: $S \subseteq$ *?B*
proof (*auto*)
fix *x* **assume** *x*: $x \in S$
show $\exists$*f U*. *finite U* $\wedge$ $U \subseteq S$ $\wedge$ ($\sum i \in U$. *i* $*$ *f i*) = *x*
by (*rule exI*[*of* - λi. *1*], *rule exI*[*of* - {*x*}], *simp add*: *x*)
qed
have *right-ideal-B*: *right-ideal ?B*
proof (*unfold right-ideal-def*, *auto*)
show *subgroup ?B*

proof (*unfold subgroup-def*, *auto*)
show $\exists f\ U.\ finite\ U \wedge U \subseteq S \wedge (\sum i{\in}U.\ i * f\ i) = 0$
by (*rule exI*[*of - id*], *rule exI*[*of -* {}], *auto*)
fix *f A* **assume** *A*: *finite A* **and** *AS*: $A \subseteq S$
show$\exists fa\ Ua.\ finite\ Ua \wedge Ua \subseteq S \wedge (\sum i{\in}Ua.\ \ i * fa\ i) = - (\sum i{\in}A.\ \ i * f\ i)$
by (*rule exI*[*of -* $\lambda i.\ - f\ i$], *rule exI*[*of - A*],
auto simp add: *A AS sum-negf*[*of* $\lambda i.\ i * f\ i\ A$])
fix *fa B* **assume** *B*: *finite B* **and** *BS*: $B \subseteq S$
let $?g{=}\lambda i.\ if\ i \in A - B\ then\ f\ i\ else\ if\ i \in B - A\ then\ fa\ i\ else\ f\ i + fa\ i$
show $\exists fb\ Ub.\ finite\ Ub \wedge Ub \subseteq S \wedge (\sum i{\in}Ub.\ i * fb\ i)$
$= (\sum i{\in}A.\ i * f\ i) + (\sum i{\in}B.\ i * fa\ i)$
proof (*rule exI*[*of - ?g*], *rule exI*[*of -* $A \cup B$], *simp add*: *A B AS BS*)
let $?g2 = (\lambda i.\ i * (if\ i \in A \wedge i \notin B\ then\ f\ i\ else$
$if\ i \in B - A\ then\ fa\ i\ else\ f\ i + fa\ i))$
have $(\sum i{\in}A.\ \ i * f\ i) + (\sum i{\in}B.\ \ i * fa\ i)$
$= (\sum i{\in}A - B.\ \ i * f\ i) + (\sum i{\in}B - A.\ \ i * fa\ i) + (\sum i{\in}A{\cap}B.\ (i * f\ i)$
$+ (i * fa\ i))$
by (*rule sum-add*[*OF A B*])
also have ... $= (\sum i{\in}A - B.\ i * f\ i) + (\sum i{\in}B - A.\ \ i * fa\ i)$
$+ (\sum i{\in}A \cap B.\ i * (f\ i + fa\ i))$
by (*simp add*: *distrib-left*)
also have ... $=\ sum\ ?g2\ (A - B) + sum\ ?g2\ (B - A) + sum\ ?g2\ (A \cap$
$B)$ **by** *auto*
also have ... $= sum\ ?g2\ (A \cup B)$ **by** (*rule sum.union-diff2*[*OF A B*,
symmetric])
finally show $sum\ ?g2\ (A \cup B) = (\sum i{\in}A.\ i * f\ i) + (\sum i{\in}B.\ i * fa\ i)$ **..**
qed
qed
fix *f U r* **assume** *U*: *finite U* **and** *U-in-S*: $U \subseteq S$
show $\exists fa\ Ua.\ finite\ Ua \wedge Ua \subseteq S \wedge (\sum i{\in}Ua.\ i * fa\ i) = (\sum i{\in}U.\ i * f\ i) * r$
by (*rule exI*[*of -* $\lambda i.\ f\ i * r$], *rule exI*[*of - U*])
(*simp add*: *U U-in-S sum-distrib-right mult-assoc*)
qed
thus $?S \subseteq ?B$ **using** *S-in-B* **unfolding** *right-ideal-generated-def* **by** *auto*
next
show $?B \subseteq ?S$
proof (*unfold right-ideal-generated-def*, *auto*)
fix *X f U*
assume *li-X*: *right-ideal X* **and** *S-X*: $S \subseteq X$ **and** *U*: *finite U* **and** *U-in-S*: U
$\subseteq S$
have *U-in-X*: $U \subseteq X$ **using** *U-in-S S-X* **by** *simp*
show $(\sum i{\in}U.\ i * f\ i) \in X$
by (*rule sum-right-ideal*[*OF li-X U-in-X U*])
qed
qed

end

context *comm-ring*

begin

lemma *left-ideal-eq-right-ideal*: *left-ideal* I = *right-ideal* I
  unfolding *left-ideal-def right-ideal-def subgroup-def*
  by *auto* (*metis mult-commute*)+

corollary *ideal-eq-left-ideal*: *ideal* I = *left-ideal* I
  by (*metis ideal-def left-ideal-eq-right-ideal*)

lemma *ideal-eq-right-ideal*: *ideal* I = *right-ideal* I
  by (*metis ideal-def left-ideal-eq-right-ideal*)

lemma *principal-ideal-eq-left*:
  principal-ideal S = ($\exists\, a.$ *left-ideal-generated* $\{a\} = S$)
  unfolding *principal-ideal-def ideal-generated-def left-ideal-generated-def*
  unfolding *ideal-eq-left-ideal* **..**

end

context *comm-ring-1*
begin

lemma *ideal-generated-eq-left-ideal*: *ideal-generated* A = *left-ideal-generated* A
  unfolding *ideal-generated-def ideal-def*
 by (*metis* (*no-types*, *lifting*) *Collect-cong left-ideal-eq-right-ideal left-ideal-generated-def*)

lemma *ideal-generated-eq-right-ideal*: *ideal-generated* A = *right-ideal-generated* A
  unfolding *ideal-generated-def ideal-def*
 by (*metis* (*no-types*, *lifting*) *Collect-cong left-ideal-eq-right-ideal right-ideal-generated-def*)

lemma *obtain-sum-ideal-generated*:
  assumes *a*: $a \in$ *ideal-generated* A **and** *A*: *finite* A
  obtains f **where** *sum* ($\lambda i.\ f\ i * i$) $A = a$
proof −
  obtain $g\ U$ **where** *g*: *sum* ($\lambda i.\ g\ i * i$) $U = a$ **and** *UA*: $U \subseteq A$ **and** *U*: *finite* U
    using *a* **unfolding** *ideal-generated-eq-left-ideal*
    unfolding *left-ideal-explicit* **by** *blast*
  let *?f*=$\lambda i.$ *if* $i \in A - U$ *then 0 else* $g\ i$
  have *A-union*: $A = (A - U) \cup U$ **using** *UA* **by** *auto*
  have *sum* ($\lambda i.$ *?f* $i * i$) A = *sum* ($\lambda i.$ *?f* $i * i$) $((A - U) \cup U)$ **using** *A-union*
by *simp*
  also have ... = *sum* ($\lambda i.$ *?f* $i * i$) $(A - U)$ + *sum* ($\lambda i.$ *?f* $i * i$) U
    by (*rule sum.union-disjoint*[*OF - U*], *auto simp add*: *A U UA*)
  also have ... = *sum* ($\lambda i.$ *?f* $i * i$) U **by** *auto*
  also have ... = a **using** *g* **by** *auto*
  finally have *sum* ($\lambda i.$ *?f* $i * i$) $A = a$ **.**
  with *that* **show** *?thesis* **.**
qed

lemma *dvd-ideal-generated-singleton*:
  assumes *subset*: *ideal-generated* $\{a\} \subseteq$ *ideal-generated* $\{b\}$
  shows *b dvd a*
proof $-$
 have $a \in$ *ideal-generated* $\{a\}$ **by** (*simp add*: *ideal-generated-in*)
 hence *a*: $a \in$ *ideal-generated* $\{b\}$ **by** (*metis subset subsetCE*)
 obtain *f* **where** *sum* $(\lambda i.\ f\ i * i)\ \{b\} = a$ **by** (*rule obtain-sum-ideal-generated*[*OF a*], *simp*)
 hence *fb-b-a*: $f\ b * b = a$ **unfolding** *sum-singleton* **.**
 show *?thesis* **unfolding** *dvd-def* **by** (*rule exI*[*of - f b*], *metis fb-b-a mult-commute*)
qed

lemma *ideal-generated-singleton*: *ideal-generated* $\{a\} = \{k{*}a|k.\ k \in UNIV\}$
proof (*auto simp add*: *ideal-generated-eq-left-ideal left-ideal-explicit*)
  fix *f U*
  assume *U*: *finite U* **and** *U-in-a*: $U \subseteq \{a\}$
  show $\exists k.\ (\sum i{\in}U.\ f\ i * i) = k * a$
  proof (*cases* $U=\{\}$)
    case *True* **show** *?thesis* **by** (*rule exI*[*of - 0*], *simp add*: *True*)
  next
    case *False*
    hence *Ua*: $U = \{a\}$ **using** *U-in-a* **by** *auto*
    show *?thesis* **by** (*rule exI*[*of - f a*]) (*simp add*: *Ua sum-singleton*)
  qed
next
  fix *k*
  show $\exists f\ U.$ *finite* $U \wedge U \subseteq \{a\} \wedge (\sum i{\in}U.\ f\ i * i) = k * a$
    by (*rule exI*[*of -* $\lambda i.\ k$], *rule exI*[*of -* $\{a\}$], *simp*)
qed

lemma *dvd-ideal-generated-singleton′*:
  assumes *b-dvd-a*: *b dvd a*
  shows *ideal-generated* $\{a\} \subseteq$ *ideal-generated* $\{b\}$
  apply (*simp only*: *ideal-generated-singleton*)
  using *assms* **unfolding** *dvd-def*
  apply *auto*
  apply (*simp-all only*: *mult-commute*)
  unfolding *mult-assoc*[*symmetric*]
  apply *blast*
  done

lemma *ideal-generated-subset2*:
  assumes *ac*: *ideal-generated* $\{a\} \subseteq$ *ideal-generated* $\{c\}$
  and *bc*: *ideal-generated* $\{b\} \subseteq$ *ideal-generated* $\{c\}$
  shows *ideal-generated* $\{a,b\} \subseteq$ *ideal-generated* $\{c\}$
proof

fix x
assume x: $x \in$ *ideal-generated* $\{a,\ b\}$
show $x \in$ *ideal-generated* $\{c\}$
proof (*cases* $a=b$)
case *True*
show *?thesis* **using** x *bc* **unfolding** *True ideal-generated-repeated* **by** *fast*
next
case *False*
obtain k **where** k: $a = c * k$
using *dvd-ideal-generated-singleton*[*OF ac*] **unfolding** *dvd-def* **by** *auto*
obtain k' **where** k': $b = c * k'$
using *dvd-ideal-generated-singleton*[*OF bc*] **unfolding** *dvd-def* **by** *auto*
obtain f **where** f: *sum* $(\lambda i.\ f\ i * i)$ $\{a,b\} = x$
by (*rule obtain-sum-ideal-generated*[*OF x*], *simp*)
hence $x = f\ a * a + f\ b * b$ **unfolding** *sum-two-elements*[*OF False*] **by** *simp*
also have ... $= f\ a * (c * k) + f\ b * (c * k')$ **unfolding** k k' **by** *simp*
also have ... $= (f\ a * k) * c + (f\ b * k') * c$
by (*simp only*: *mult-assoc*) (*simp only*: *mult-commute*)
also have ... $= (f\ a * k + f\ b * k') * c$
by (*simp only*: *mult-commute*) (*simp only*: *distrib-left*)
finally have $x = (f\ a * k + f\ b * k') * c$ **.**
thus *?thesis* **unfolding** *ideal-generated-singleton* **by** *auto*
qed
qed

end

lemma *ideal-kZ*: *ideal* $\{k{*}x|x.\ x{\in}(UNIV{::}int\ set)\}$
unfolding *ideal-def left-ideal-def right-ideal-def subgroup-def*
apply *auto*
apply (*metis int-distrib*(*2*))
apply (*metis minus-mult-right*)
apply (*metis int-distrib*(*2*))
apply (*metis minus-mult-right*)
done

1.4 GCD Rings and Bezout Domains

To define GCD rings and Bezout rings, there are at least two options: fix the operation gcd or just assume its existence. We have chosen the second one in order to be able to use subclasses (if we fix a gcd in the bezout ring class, then we couln't prove that principal ideal rings are a subclass of bezout rings).

class *GCD-ring* = *comm-ring-1*
+ **assumes** *exists-gcd*: $\exists\, d.\ d\ dvd\ a \wedge d\ dvd\ b \wedge (\forall\, d'.\ d'\ dvd\ a \wedge d'\ dvd\ b \longrightarrow d'\ dvd\ d)$
begin

In this structure, there is always a gcd for each pair of elements, but maybe not unique. The following definition essentially says if a function satisfies the condition to be a gcd.

definition *is-gcd* :: $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow bool$
  where *is-gcd* $(gcd') = (\forall\, a\ b.\ (gcd'\ a\ b\ dvd\ a)$
  $\wedge\ (gcd'\ a\ b\ dvd\ b)$
  $\wedge\ (\forall\, d'.\ d'\ dvd\ a \wedge d'\ dvd\ b \longrightarrow d'\ dvd\ gcd'\ a\ b))$

lemma *gcd′-dvd1*:
  assumes *is-gcd gcd′* **shows** $gcd'\ a\ b\ dvd\ a$ **using** *assms* **unfolding** *is-gcd-def* **by** *auto*

lemma *gcd′-dvd2*:
  assumes *is-gcd gcd′* **shows** $gcd'\ a\ b\ dvd\ b$
  using *assms* **unfolding** *is-gcd-def* **by** *auto*

lemma *gcd′-greatest*:
  assumes *is-gcd gcd′* **and** $l\ dvd\ a$ **and** $l\ dvd\ b$
  shows $l\ dvd\ gcd'\ a\ b$
  using *assms* **unfolding** *is-gcd-def* **by** *auto*

lemma *gcd′-zero* [*simp*]:
  assumes *is-gcd gcd′*
  shows $gcd'\ x\ y = 0 \longleftrightarrow x = 0 \wedge y = 0$
  by (*metis dvd-0-left dvd-refl gcd′-dvd1 gcd′-dvd2 gcd′-greatest assms*)+

end

class *GCD-domain* = *GCD-ring* + *idom*

class *bezout-ring* = *comm-ring-1* +
  assumes *exists-bezout*: $\exists\, p\ q\ d.\ (p*a + q*b = d)$
  $\wedge\ (d\ dvd\ a)$
  $\wedge\ (d\ dvd\ b)$
  $\wedge\ (\forall\, d'.\ (d'\ dvd\ a \wedge d'\ dvd\ b) \longrightarrow d'\ dvd\ d)$
begin

subclass *GCD-ring*
proof
  fix $a\ b$
  show $\exists\, d.\ d\ dvd\ a \wedge d\ dvd\ b \wedge (\forall\, d'.\ d'\ dvd\ a \wedge d'\ dvd\ b \longrightarrow d'\ dvd\ d)$
    using *exists-bezout*[*of a b*] **by** *auto*
qed

In this structure, there is always a bezout decomposition for each pair of elements, but it is not unique. The following definition essentially says if a function satisfies the condition to be a bezout decomposition.

definition *is-bezout* :: $('a \Rightarrow 'a \Rightarrow ('a \times 'a \times 'a)) \Rightarrow bool$
  where *is-bezout* $(bezout) = (\forall\, a\ b.\ let\ (p,\ q,\ gcd\text{-}a\text{-}b) = bezout\ a\ b$
  in

$p * a + q * b = gcd\text{-}a\text{-}b$
$\wedge$ ($gcd\text{-}a\text{-}b$ *dvd* a)
$\wedge$ ($gcd\text{-}a\text{-}b$ *dvd* b)
$\wedge$ ($\forall d'$. d' *dvd* $a \wedge d'$ *dvd* $b \longrightarrow d'$ *dvd* $gcd\text{-}a\text{-}b$))

The following definition is similar to the previous one, and checks if the input is a function that given two parameters a b returns 5 elements (p,q,u,v,d) where d is a gcd of a and b, p and q are the bezout coefficients such that $p*a+q*b=d$, $d*u = -b$ and $d*v= a$. The elements u and v are useful for defining the bezout matrix.

definition *is-bezout-ext* :: ($'a \Rightarrow 'a \Rightarrow ('a \times 'a \times 'a \times 'a \times 'a)) \Rightarrow$ *bool*
where *is-bezout-ext* (*bezout*) = ($\forall$ *a b*. *let* (*p*, *q*, *u*, *v*, *gcd-a-b*) = *bezout a b*
in
$p * a + q * b = gcd\text{-}a\text{-}b$
$\wedge$ (*gcd-a-b dvd a*)
$\wedge$ (*gcd-a-b dvd b*)
$\wedge$ ($\forall d'$. d' *dvd* $a \wedge d'$ *dvd* $b \longrightarrow d'$ *dvd gcd-a-b*)
$\wedge$ $gcd\text{-}a\text{-}b * u = -b$
$\wedge$ $gcd\text{-}a\text{-}b * v = a$)

lemma *is-gcd-is-bezout-ext*:
assumes *is-bezout-ext bezout*
shows *is-gcd* (λa *b*. *case bezout a b of* (*x*, *xa*,*u*,*v*, *gcd'*) $\Rightarrow$ *gcd'*)
unfolding *is-gcd-def* **using** *assms* **unfolding** *is-bezout-ext-def Let-def* **by** (*simp add*: *split-beta*)

lemma *is-bezout-ext-is-bezout*:
assumes *is-bezout-ext bezout*
shows *is-bezout* (λa *b*. *case bezout a b of* (*x*,*xa*,*u*,*v*, *gcd'*) $\Rightarrow$ (*x*,*xa*,*gcd'*))
unfolding *is-bezout-def* **using** *assms* **unfolding** *is-bezout-ext-def Let-def* **by** (*simp add*: *split-beta*)

lemma *is-gcd-is-bezout*:
assumes *is-bezout bezout*
shows *is-gcd* (λa *b*. (*case bezout a b of* (-, -, *gcd'*) $\Rightarrow$ (*gcd'*)))
unfolding *is-gcd-def* **using** *assms* **unfolding** *is-bezout-def Let-def* **by** (*simp add*: *split-beta*)

The assumptions of the Bezout rings say that there exists a bezout operation. Now we will show that there also exists an operation satisfying *is-bezout-ext*

lemma *exists-bezout-ext-aux*:
fixes a **and** b
shows $\exists p\ q\ u\ v\ d.\ (p * a + q * b = d)$
$\wedge$ (d *dvd* a)
$\wedge$ (d *dvd* b)
$\wedge$ ($\forall d'$. (d' *dvd* $a \wedge d'$ *dvd* b) $\longrightarrow d'$ *dvd* d) $\wedge d * u = -b \wedge d * v = a$
proof –

obtain *p q d* **where** *prems01*: $(p * a + q * b = d)$
$\wedge$ $(d\ dvd\ a)$
$\wedge$ $(d\ dvd\ b)$
$\wedge$ $(\forall d'.\ (d'\ dvd\ a \wedge d'\ dvd\ b) \longrightarrow d'\ dvd\ d)$
using *exists-bezout* [*of a b*] **by** *fastforce*
hence *db*: *d dvd b* **and** *da*: *d dvd a* **by** *blast+*
obtain *u v* **where** *prems02*: $d * u = -b$ **and** *prems03*: $d * v = a$ **using** *db* **and** *da*
by (*metis local.dvdE local.minus-mult-right*)
show *?thesis* **using** *exI* [*of - (p,q,u,v,d)*] *prems01 prems02 prems03*
by *metis*
qed

lemma *exists-bezout-ext*: $\exists$ *bezout-ext. is-bezout-ext bezout-ext*
proof −
define *bezout-ext* **where** *bezout-ext a b* = (*SOME* (*p,q,u,v,d*). $p * a + q * b = d$ $\wedge$ $(d\ dvd\ a) \wedge (d\ dvd\ b) \wedge (\forall d'.\ d'\ dvd\ a \wedge d'\ dvd\ b \longrightarrow d'\ dvd\ d) \wedge d * u = -b \wedge d * v = a$)
for *a b*
show *?thesis*
proof (*rule exI* [*of - bezout-ext*], *unfold is-bezout-ext-def*, *rule+*)
fix *a b*
obtain *p q u v d* **where** *foo*: $p * a + q * b = d \wedge$
$d\ dvd\ a \wedge$
$d\ dvd\ b \wedge$
$(\forall d'.\ d'\ dvd\ a \wedge d'\ dvd\ b \longrightarrow d'\ dvd\ d) \wedge$
$d * u = -\ b \wedge d * v = a$ **using** *exists-bezout-ext-aux* [*of a b*] **by** *fastforce*
show *let* (*p, q, u, v, gcd-a-b*) = *bezout-ext a b*
in $p * a + q * b =$ *gcd-a-b* $\wedge$
gcd-a-b dvd a $\wedge$
gcd-a-b dvd b $\wedge$
($\forall d'.\ d'\ dvd\ a \wedge d'\ dvd\ b \longrightarrow d'\ dvd$ *gcd-a-b*) $\wedge$
gcd-a-b $* u = -\ b \wedge$ *gcd-a-b* $* v = a$
by (*unfold bezout-ext-def Let-def*, *rule someI* [*of - (p,q,u,v,d)*], *clarify*, *rule foo*)
qed
qed

end

class *bezout-domain* = *bezout-ring* + *idom*

subclass (**in** *bezout-domain*) *GCD-domain*
proof
qed

class *bezout-ring-div* = *bezout-ring* + *euclidean-semiring*
class *bezout-domain-div* = *bezout-domain* + *euclidean-semiring*

subclass (**in** *bezout-ring-div*) *bezout-domain-div*
proof qed

1.5 Principal Ideal Domains

class *pir* = *comm-ring-1* + **assumes** *all-ideal-is-principal*: *ideal I* $\Longrightarrow$ *principal-ideal I*
class *pid* = *idom* + *pir*

Thanks to the following proof, we will show that there exist bezout and gcd operations in principal ideal rings for each pair of elements.

subclass (**in** *pir*) *bezout-ring*
proof
  fix *a b*
  define *S* **where** *S* = *ideal-generated* {*a*,*b*}
  have *ideal-S*: *ideal S* **using** *ideal-ideal-generated* **unfolding** *S-def* **by** *simp*
   obtain *d* **where** *d*: *ideal-generated* {*d*} = *S* **using** *all-ideal-is-principal*[*OF ideal-S*]
    unfolding *principal-ideal-def* **by** *blast*
  have *ideal-d*: *ideal* (*ideal-generated* {*d*}) **using** *ideal-ideal-generated* **by** *simp*
  have *a-subset-d*: *ideal-generated* {*a*} ⊆ *ideal-generated* {*d*}
    by (*metis S-def d insertI1 ideal-generated-subset singletonD subsetI*)
  have *b-subset-d*: *ideal-generated* {*b*} ⊆ *ideal-generated* {*d*}
    by (*metis S-def d insert-iff ideal-generated-subset subsetI*)
  have *d-in-S*: *d* ∈ *S* **by** (*metis d insert-subset ideal-generated-subset-generator*)
  obtain *f U* **where** *U*: *U* ⊆ {*a*,*b*} **and** *f*: *sum* (λ*i*. *f i* ∗ *i*) *U* = *d*
   using *left-ideal-explicit*[*of* {*a*,*b*}] *d-in-S* **unfolding** *S-def ideal-generated-eq-left-ideal*
    by *auto*
  define *g* **where** *g i* = (*if i* ∈ *U then f i else 0*) **for** *i*
  show ∃ *p q d*. *p* ∗ *a* + *q* ∗ *b* = *d* ∧ *d dvd a* ∧ *d dvd b* ∧ (∀ *d'*. *d' dvd a* ∧ *d' dvd b* ⟶ *d' dvd d*)
  proof (*cases a* = *b*)
    case *True*
    show *?thesis*
    proof (*rule exI*[*of - g a*], *rule exI*[*of - 0*], *rule exI*[*of - d*], *auto*)
      show *ga-a-d*: *g a* ∗ *a* = *d*
        unfolding *g-def*
      proof *auto*
        assume *a* ∈ *U*
        hence *Ua*: *U* = {*a*} **using** *U True* **by** *auto*
        show *f a* ∗ *a* = *d* **using** *f* **unfolding** *Ua*
          unfolding *sum-singleton* **.**
      next
        assume *a* ∉ *U*
        hence *U-empty*: *U* = {} **using** *U True* **by** *auto*
        show *0* = *d* **using** *f* **unfolding** *U-empty* **by** *auto*
      qed
      show *d dvd a* **by** (*rule dvd-ideal-generated-singleton*[*OF a-subset-d*])
      show *d dvd b* **by** (*rule dvd-ideal-generated-singleton*[*OF b-subset-d*])

```
      fix d′ assume d′-dvd-a: d′ dvd a and d′-dvd-b: d′ dvd b
      show d′ dvd d by (metis ga-a-d d′-dvd-a dvd-mult2 mult-commute)
    qed
  next
    case False
    show ?thesis
    proof (rule exI[of - g a], rule exI[of - g b], rule exI[of - d], auto)
      show g a ∗ a + g b ∗ b = d
      proof (auto simp add: g-def)
        assume a: a ∈ U and b: b ∈ U
        hence U-ab: U = {a,b} using U by auto
        show f a ∗ a + f b ∗ b = d  using f unfolding U-ab sum-two-elements[OF
False] .
      next
        assume a: a ∈ U and b: b ∉ U
        hence U-a: U = {a} using U by auto
        show f a ∗ a = d using f unfolding U-a sum-singleton .
      next
        assume a: a ∉ U and b: b ∈ U
        hence U-b: U = {b} using U by auto
        show f b ∗ b = d using f unfolding U-b sum-singleton .
      next
        assume a: a ∉ U and b: b ∉ U
        hence U = {} using U by auto
        thus 0 = d using f by auto
      qed
      show d dvd a by (rule dvd-ideal-generated-singleton[OF a-subset-d])
      show d dvd b by (rule dvd-ideal-generated-singleton[OF b-subset-d])
      fix d′ assume d′a: d′ dvd a and d′b: d′ dvd b
      have ad′: ideal-generated {a} ⊆ ideal-generated {d′}
        by (rule dvd-ideal-generated-singleton′[OF d′a])
      have bd′: ideal-generated {b} ⊆ ideal-generated {d′}
        by (rule dvd-ideal-generated-singleton′[OF d′b])
      have abd′: ideal-generated {a,b} ⊆ ideal-generated {d′}
        by (rule ideal-generated-subset2[OF ad′ bd′])
      hence dd′: ideal-generated {d} ⊆ ideal-generated {d′}
        by (simp add: S-def d)
      show d′ dvd d by (rule dvd-ideal-generated-singleton[OF dd′])
    qed
  qed
qed

subclass (in pid) bezout-domain
proof
qed

context pir
begin
```

```
lemma ascending-chain-condition:
  fixes I::nat=>'a set
  assumes all-ideal: ∀ n. ideal (I(n))
  and inc: ∀ n. I(n) ⊆ I(n+1)
  shows ∃ n. I(n)=I(n+1)
proof −
  let ?I = ⋃{I(n)|n. n∈(UNIV::nat set)}
  have ideal ?I using ideal-Union[of I] all-ideal inc by fast
  from this obtain a where a: ideal-generated {a} = ?I
    using all-ideal-is-principal
    unfolding principal-ideal-def by fastforce
  have a ∈ ?I using a ideal-generated-subset-generator[of {a} ?I] by simp
  from this obtain k where a-Ik: a ∈ I(k) using Union-iff[of a {I n |n. n ∈
UNIV}] by auto
  show ?thesis
  proof (rule exI[of - k], rule)
    show I k ⊆ I (k + 1) using inc by simp
    show I (k + 1) ⊆ I k
    proof (auto)
      fix x assume x: x ∈ I (Suc k)
      have ideal-generated {a} = I k
      proof
        have ideal-Ik: ideal (I (k)) using all-ideal by simp
        show I k ⊆ ideal-generated {a} using a by auto
        show ideal-generated {a} ⊆ I k
          by (metis (lifting) a-Ik all-ideal ideal-generated-def
            le-Inf-iff mem-Collect-eq singleton-iff subsetI)
      qed
      thus x ∈ I k using x unfolding a by auto
    qed
  qed
qed


lemma ascending-chain-condition2:
  ∄ I::(nat ⇒ 'a set). (∀ n. ideal (I n) ∧ I n ⊂ I (n + 1))
proof (rule ccontr, auto)
  fix I assume a: ∀ n. ideal (I n) ∧ I n ⊂ I (Suc n)
  hence ∀ n. ideal (I n) ∀ n. I n ⊆ I (Suc n) by auto
  hence ∃ n. I(n)=I(n+1) using ascending-chain-condition by auto
  thus False using a by auto
qed

end

class pir-div = pir + euclidean-semiring
class pid-div = pid + euclidean-semiring

subclass (in pir-div) pid-div
```

proof qed

subclass (**in** *pir-div*) *bezout-ring-div*
proof qed

subclass (**in** *pid-div*) *bezout-domain-div*
proof qed

1.6 Euclidean Domains

We make use of the euclidean ring (domain) class developed by Manuel Eberl.

subclass (**in** *euclidean-ring*) *pid*
proof
  fix *I* **assume** *I*: *ideal I*
  show *principal-ideal I*
  proof (*cases I*={*0*})
    case *True* **show** *?thesis* **unfolding** *principal-ideal-def True*
      using *ideal-generated-0 ideal-0* **by** *auto*
  next
    case *False*
   have *fI-not-empty*: (*euclidean-size*‘ (*I*−{*0*}))≠{} **using** *False ideal-not-empty*[*OF I*] **by** *auto*
    from *this* **obtain** *d* **where** *fd*: *euclidean-size d*
      = *Least* (λ*i*. *i* ∈ (*euclidean-size*‘(*I*−{*0*}))) **and** *d*: *d*∈(*I*−{*0*})
      by (*metis* (*lifting*, *mono-tags*) *LeastI-ex imageE ex-in-conv*)
    have *d-not-0*: *d*≠*0* **using** *d* **by** *simp*
    have *fd-le*: ∀ *x*∈*I*−{*0*}. *euclidean-size d* ≤ *euclidean-size x*
      by (*metis* (*mono-tags*) *Least-le fd image-eqI*)
    show *principal-ideal I*
    proof (*unfold principal-ideal-def*, *rule exI*[*of - d*], *auto*)
      fix *x* **assume** *x*:*x* ∈ *ideal-generated* {*d*} **show** *x* ∈ *I*
        using *x* **unfolding** *ideal-generated-def*
        by (*auto*, *metis Diff-iff I d*)
    next
      fix *a* **assume** *a*: *a* ∈ *I*
      obtain *q r* **where** *a* = *q* ∗ *d* + *r*
        and *fr-fd*: *euclidean-size r* < *euclidean-size d*
        using *div-mult-mod-eq* [*of a d*, *symmetric*] *d-not-0 mod-size-less*
        by *blast*
      show *a* ∈ *ideal-generated* {*d*}
      proof (*cases r*=*0*)
        case *True* **hence** *a* = *q* ∗ *d* **using** ‹*a* = *q* ∗ *d* + *r*›
          by *auto*
        then show *?thesis* **unfolding** *ideal-generated-def*
          unfolding *ideal-def right-ideal-def*
          by (*simp add*: *ac-simps*)
      next
        case *False*

hence *r-noteq-0*: $r \neq 0$ **by** *simp*
have $r = a - d * q$ **using** ‹$a = q * d + r$›
by (*simp add*: *algebra-simps*)
also have ... $\in I$
proof (*rule left-ideal-minus*)
show *left-ideal I* **using** *I* **unfolding** *ideal-def* **by** *simp*
show $a \in I$ **using** *a* **.**
show $d * q \in I$ **using** *d I* **unfolding** *ideal-def right-ideal-def* **by** *simp*
qed
finally have $r \in I - \{0\}$ **using** *r-noteq-0* **by** *auto*
hence *euclidean-size d* $\leq$ *euclidean-size r* **using** *fd-le* **by** *auto*
thus *?thesis* **using** *fr-fd* **by** *auto*
qed
qed
qed
qed

context *euclidean-ring-gcd*
begin

This is similar to the *euclid-ext* operation, but involving two more parameters to satisfy that *is-bezout-ext euclid-ext2*

definition *euclid-ext2* :: $'a \Rightarrow 'a \Rightarrow 'a \times 'a \times 'a \times 'a \times 'a$
where *euclid-ext2 a b* =
(*fst* (*bezout-coefficients a b*), *snd* (*bezout-coefficients a b*), $-$ *b div gcd a b*, *a div gcd a b*, *gcd a b*)

lemma *is-bezout-ext-euclid-ext2*: *is-bezout-ext* (*euclid-ext2*)
proof (*unfold is-bezout-ext-def Let-def*, *clarify*, *intro conjI*)
fix *a b p q u v d*
assume *e*: *euclid-ext2 a b* = (*p*, *q*, *u*, *v*, *d*)
then have *bezout-coefficients a b* = (*p*, *q*) **and** *gcd a b* = *d*
by (*auto simp add*: *euclid-ext2-def*)
then show $p * a + q * b = d$
by (*simp add*: *bezout-coefficients*)
from ‹*gcd a b* = *d*› **show** *d dvd a* **and** *d dvd b*
by *auto*
from ‹*gcd a b* = *d*› **show** $\forall d'.\ d'$ *dvd a* $\wedge$ d' *dvd b* $\longrightarrow$ d' *dvd d*
by *auto*
have *a div d* = *v* **and** $-b$ *div d* = *u*
using *e* **by** (*auto simp add*: *euclid-ext2-def*)
then show $d * v = a$ **and** $d * u = -b$
using ‹*d dvd a*› **and** ‹*d dvd b*› **by** *auto*
qed

lemma *is-bezout-euclid-ext*: *is-bezout* ($\lambda a\ b.$ (*fst* (*bezout-coefficients a b*), *snd* (*bezout-coefficients a b*), *gcd a b*))
by (*auto simp add*: *is-bezout-def bezout-coefficients*)

end

subclass (**in** *euclidean-ring*) *pid-div* **..**

1.7 More gcd structures

The following classes represent structures where there exists a gcd for each pair of elements and the operation is fixed.

class *pir-gcd* = *pir* + *semiring-gcd*
class *pid-gcd* = *pid* + *pir-gcd*

subclass (**in** *euclidean-ring-gcd*) *pid-gcd* **..**

1.8 Field

Proving that any field is a euclidean domain. There are alternatives to do this, see https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2014-October/msg00034.html

class *field-euclidean* = *field* + *euclidean-ring* +
  assumes *euclidean-size* = (λi. *if* $i = 0$ *then* 0 *else* 1::*nat*)
  and *normalisation-factor* = *id*

end

theory *Cayley-Hamilton-Compatible*
  imports
    Rings2
    Cayley-Hamilton.Cayley-Hamilton
    Gauss-Jordan.Determinants2
begin

1.9 Compatibility layer btw *Cayley-Hamilton.Square-Matrix* and *Gauss-Jordan.Determinants2*

hide-const (**open**) *Square-Matrix.det*
hide-const (**open**) *Square-Matrix.row*
hide-const (**open**) *Square-Matrix.col*
hide-const (**open**) *Square-Matrix.transpose*
hide-const (**open**) *Square-Matrix.cofactor*
hide-const (**open**) *Square-Matrix.adjugate*

hide-fact (**open**) *det-upperdiagonal*
hide-fact (**open**) *row-def*
hide-fact (**open**) *col-def*
hide-fact (**open**) *transpose-def*

lemma *det-sq-matrix-eq*: *Square-Matrix.det* (*from-vec A*) = *det A*
  unfolding *Square-Matrix.det.rep-eq Determinants.det-def from-vec.rep-eq* ..

lemma *to-vec-matrix-scalar-mult*: *to-vec* ($x *_S A$) = *x *k to-vec A*
  by *transfer* (*simp add*: *matrix-scalar-mult-def*)

lemma *to-vec-matrix-matrix-mult*: *to-vec* (*A * B*) = *to-vec A ** to-vec B*
  by *transfer* (*simp add*: *matrix-matrix-mult-def*)

lemma *to-vec-diag*: *to-vec* (*diag x*) = *mat x*
  by *transfer* (*simp add*: *mat-def*)

lemma *to-vec-one*: *to-vec 1* = *mat 1*
  by *transfer* (*simp add*: *mat-def*)

lemma *to-vec-eq-iff*: *to-vec M* = *to-vec N* ⟷ *M* = *N*
  by *transfer* (*auto simp*: *vec-eq-iff*)

1.10 Some preliminary lemmas and results

lemma *invertible-iff-is-unit*:
  fixes *A*::'*a*::{*comm-ring-1*}^'*n*^'*n*
  shows *invertible A* ⟷ (*det A*) *dvd 1*
proof
  assume *inv-A*: *invertible A*
  obtain *B* **where** *AB-mat*: *A ** B* = *mat 1* **using** *inv-A* **unfolding** *invertible-def*
by *auto*
  have *1* = *det* (*mat 1*::'*a*^'*n*^'*n*) **unfolding** *det-I* ..
  also have ... = *det* (*A ** B*) **unfolding** *AB-mat* ..
  also have ... = *det A * det B* **unfolding** *det-mul* ..
  finally have *1* = *det A * det B* **by** *simp*
  thus (*det A*) *dvd 1* **unfolding** *dvd-def* **by** *auto*
next
  assume *det-unit*: (*det A*) *dvd 1*
  from *this* **obtain** *a* **where** *a*: (*det A*) * *a* = *1* **unfolding** *dvd-def* **by** *auto*
  let *?A* = *to-vec* (*Square-Matrix.adjugate* (*from-vec A*))
  show *invertible A*
  proof (*unfold invertible-def*, *rule exI*[*of - a *k ?A*])
    have *from-vec A* * ($a *_S$ *Square-Matrix.adjugate* (*from-vec A*)) = *1*
      ($a *_S$ *Square-Matrix.adjugate* (*from-vec A*)) * *from-vec A* = *1*
      using *a* **unfolding** *smult-mult2*[*symmetric*] *mult-adjugate-det*[*of from-vec A*]
smult-diag det-sq-matrix-eq
        smult-mult1[*symmetric*] *adjugate-mult-det*[*of from-vec A*]
      by (*simp-all add*: *ac-simps diag-1*)
    then show *A *** (*a *k ?A*) = *mat 1* ∧ *a *k ?A ** A* = *mat 1*
    unfolding *to-vec-eq-iff*[*symmetric*] *to-vec-matrix-matrix-mult to-vec-matrix-scalar-mult*
      to-vec-from-vec to-vec-one **by** *simp*
  qed

qed

definition *minorM M i j* = (χ *k l. if k* = *i* $\wedge$ *l* = *j then 1 else if k* = *i* $\vee$ *l* = *j then 0 else M* \$ *k* \$ *l*)

lemma *minorM-eq*: *minorM M i j* = *to-vec* (*minor* (*from-vec M*) *i j*)
  unfolding *minorM-def* **by** *transfer standard*

definition *cofactor* **where** *cofactor A i j* = *det* (*minorM A i j*)

definition *cofactorM* **where** *cofactorM A* = (χ *i j. cofactor A i j*)

lemma *cofactorM-eq*: *cofactorM* = *to-vec* $\circ$ *Square-Matrix.cofactor* $\circ$ *from-vec*
  unfolding *cofactorM-def cofactor-def*[*abs-def*] *det-sq-matrix-eq*[*symmetric*] *minorM-eq fun-eq-iff*
  apply *rule*
  apply *transfer′*
  apply (*simp add*: *fun-eq-iff vec-eq-iff*)
  apply *transfer*
  apply *simp*
  done

definition *mat2matofpoly* **where** *mat2matofpoly A* = (χ *i j.* [: *A* \$ *i* \$ *j* :])

definition *charpoly* **where** *charpoly-def*: *charpoly A* = *det* (*mat* (*monom 1* (*Suc 0*)) − *mat2matofpoly A*)

lemma *charpoly-eq*: *charpoly A* = *Cayley-Hamilton.charpoly* (*from-vec A*)
  unfolding *charpoly-def Cayley-Hamilton.charpoly-def det-sq-matrix-eq*[*symmetric*] *X-def C-def*
  apply (*intro arg-cong*[**where** *f*=*Square-Matrix.det*])
  apply *transfer′*
  apply (*simp add*: *fun-eq-iff mat-def mat2matofpoly-def C-def monom-Suc*)
  done

definition *adjugate* **where** *adjugate A* = *transpose* (*cofactorM A*)

lemma *adjugate-eq*: *adjugate* = *to-vec* $\circ$ *Square-Matrix.adjugate* $\circ$ *from-vec*
  apply (*simp add*: *adjugate-def Square-Matrix.adjugate-def fun-eq-iff*)
  apply *rule*
  apply *transfer′*
  apply (*simp add*: *transpose-def cofactorM-eq to-vec.rep-eq*
    Square-Matrix.cofactor.rep-eq)
  done

end

2 Code Cayley Hamilton

theory *Code-Cayley-Hamilton*
  imports
  HOL−Computational-Algebra.Polynomial
  Cayley-Hamilton-Compatible
  Gauss-Jordan.Code-Matrix
begin

2.1 Code equations for the definitions presented in the Cayley-Hamilton development

definition *scalar-matrix-mult-row c A i* = (χ *j*. *c* ∗ (*A* $ *i* $ *j*))

lemma *scalar-matrix-mult-row-code* [*code abstract*]:
  vec-nth (*scalar-matrix-mult-row c A i*) =(% *j*. *c* ∗ (*A* $ *i* $ *j*))
  by(*simp add*: *scalar-matrix-mult-row-def fun-eq-iff*)

lemma *scalar-matrix-mult-code* [*code abstract*]: *vec-nth* (*c* ∗*k A*) = *scalar-matrix-mult-row c A*
  unfolding *matrix-scalar-mult-def scalar-matrix-mult-row-def*[*abs-def*]
  using *vec-lambda-beta* **by** *auto*

definition *minorM-row A i j k*= *vec-lambda* (%*l*. *if k* = *i* ∧ *l* = *j then 1 else*
  if k = *i* ∨ *l* = *j then 0 else A*$*k*$*l*)

lemma *minorM-row-code* [*code abstract*]:
  vec-nth (*minorM-row A i j k*) =(%*l*. *if k* = *i* ∧ *l* = *j then 1 else*
  if k = *i* ∨ *l* = *j then 0 else A*$*k*$*l*)
  by(*simp add*: *minorM-row-def fun-eq-iff*)

lemma *minorM-code* [*code abstract*]: *vec-nth* (*minorM A i j*) = *minorM-row A i j*
  unfolding *minorM-def* **by** *transfer* (*auto simp*: *vec-eq-iff fun-eq-iff minorM-row-def*)

definition *cofactorM-row A i* = *vec-lambda* (λ*j*. *cofactorM A* $ *i* $ *j*)

lemma *cofactorM-row-code* [*code abstract*]: *vec-nth* (*cofactorM-row A i*) = *cofactor A i*
  by (*simp add*: *fun-eq-iff cofactorM-row-def cofactor-def cofactorM-def*)

lemma *cofactorM-code* [*code abstract*]: *vec-nth* (*cofactorM A*) = *cofactorM-row A*
  by (*simp add*: *fun-eq-iff cofactorM-row-def vec-eq-iff*)

lemmas *cofactor-def*[*code-unfold*]

definition *mat2matofpoly-row*
  where *mat2matofpoly-row A i* = *vec-lambda* (λ*j*. [: *A* $ *i* $ *j* :])

lemma *mat2matofpoly-row-code* [*code abstract*]:
vec-nth (*mat2matofpoly-row A i*) = (%*j*. [: *A* $ *i* $ *j* :])
unfolding *mat2matofpoly-row-def* **by** *auto*

lemma [*code abstract*]: *vec-nth* (*mat2matofpoly k*) = *mat2matofpoly-row k*
unfolding *mat2matofpoly-def* **unfolding** *mat2matofpoly-row-def*[*abs-def*] **by** *auto*

primrec *matpow* :: 'a::*semiring-1*^'n^'n ⇒ *nat* ⇒ 'a^'n^'n **where**
matpow-0: *matpow A 0* = *mat 1* |
matpow-Suc: *matpow A* (*Suc n*) = *A* ** (*matpow A n*)

definition *evalmat* :: 'a::*comm-ring-1 poly* ⇒ 'a^'n^'n ⇒ 'a^'n^'n **where**
evalmat P A = (∑ *i* ∈ { *n*::*nat* . *n* ≤ (*degree P*) } . (*coeff P i*) *k (*matpow A i*))

lemma *evalmat-unfold*:
evalmat P A = (∑ *i* = *0*..*degree P*. *coeff P i* *k *matpow A i*)
by (*simp add*: *evalmat-def atMost-def* [*symmetric*] *atMost-atLeast0*)

lemma *evalmat-code*[*code*]:
evalmat P A = (∑ *i*←[*0*..*int* (*degree P*)]. *coeff P* (*nat i*) *k *matpow A* (*nat i*))
(**is** - = *?rhs*)
proof −
let *?t* = λ*n*. *coeff P n* *k *matpow A n*
have (∑ *i* = *0*..*degree P*. *coeff P i* *k *matpow A i*) = (∑ *i*∈{*0*..*int* (*degree P*)}. *coeff P* (*nat i*) *k *matpow A* (*nat i*))
by (*rule sum.reindex-cong* [*of nat*])
(*auto simp add*: *eq-nat-nat-iff image-iff intro*: *inj-onI*, *presburger*)

also have ... = *?rhs*
by (*simp add*: *sum-set-upto-conv-sum-list-int* [*symmetric*])
finally show *?thesis*
by (*simp add*: *evalmat-unfold*)
qed

definition *coeffM-zero* :: 'a *poly*^'n^'n ⇒ 'a::*zero*^'n^'n **where**
coeffM-zero A = (χ *i j*. (*coeff* (*A* $ *i* $ *j*) *0*))

definition *coeffM-zero-row A i* = (χ *j*. (*coeff* (*A* $ *i* $ *j*) *0*))

definition *coeffM* :: 'a *poly*^'n^'n ⇒ *nat* ⇒ 'a::*zero*^'n^'n **where**
coeffM A n = (χ *i j*. *coeff* (*A* $ *i* $ *j*) *n*)

lemma *coeffM-zero-row-code* [*code abstract*]:
vec-nth (*coeffM-zero-row A i*) = (% *j*. (*coeff* (*A* $ *i* $ *j*) *0*))
by(*simp add*: *coeffM-zero-row-def fun-eq-iff*)

lemma *coeffM-zero-code* [*code abstract*]: *vec-nth* (*coeffM-zero A*) = *coeffM-zero-row A*

unfolding *coeffM-zero-def coeffM-zero-row-def*[*abs-def*]
using *vec-lambda-beta* **by** *auto*

definition
coeffM-row A n i = (χ *j. coeff* (*A* \$ *i* \$ *j*) *n*)

lemma *coeffM-row-code* [*code abstract*]:
vec-nth (*coeffM-row A n i*) = (% *j. coeff* (*A* \$ *i* \$ *j*) *n*)
by(*simp add*: *coeffM-row-def coeffM-def fun-eq-iff*)

lemma *coeffM-code* [*code abstract*]: *vec-nth* (*coeffM A n*) = *coeffM-row A n*
unfolding *coeffM-def coeffM-row-def*[*abs-def*]
using *vec-lambda-beta* **by** *auto*

end

3 Echelon Form

theory *Echelon-Form*
imports
Rings2
Gauss-Jordan.Determinants2
Cayley-Hamilton-Compatible
begin

3.1 Definition of Echelon Form

Echelon form up to column k (NOT INCLUDED).

definition
echelon-form-upt-k :: $'a$::{*bezout-ring*}$\hat{}$$'cols$::{*mod-type*}$\hat{}$$'rows$::{*finite, ord*} $\Rightarrow$ *nat* $\Rightarrow$ *bool*
where
echelon-form-upt-k A k = (
($\forall$ *i. is-zero-row-upt-k i k A*
$\longrightarrow$ $\neg$ ($\exists$ *j. j*>*i* $\wedge$ $\neg$ *is-zero-row-upt-k j k A*))
$\wedge$
($\forall$ *i j. i*<*j* $\wedge$ $\neg$ (*is-zero-row-upt-k i k A*) $\wedge$ $\neg$ (*is-zero-row-upt-k j k A*)
$\longrightarrow$ ((*LEAST n. A* \$ *i* \$ *n* $\neq$ *0*) < (*LEAST n. A* \$ *j* \$ *n* $\neq$ *0*))))

definition *echelon-form A* = *echelon-form-upt-k A* (*ncols A*)

Some properties of matrices in echelon form.

lemma *echelon-form-upt-k-intro*:
assumes ($\forall$ *i. is-zero-row-upt-k i k A* $\longrightarrow$ $\neg$ ($\exists$ *j. j*>*i* $\wedge$ $\neg$ *is-zero-row-upt-k j k A*))
and ($\forall$ *i j. i*<*j* $\wedge$ $\neg$ (*is-zero-row-upt-k i k A*) $\wedge$ $\neg$ (*is-zero-row-upt-k j k A*)
$\longrightarrow$ ((*LEAST n. A* \$ *i* \$ *n* $\neq$ *0*) < (*LEAST n. A* \$ *j* \$ *n* $\neq$ *0*)))

shows *echelon-form-upt-k A k* **using** *assms* **unfolding** *echelon-form-upt-k-def*
by *fast*

lemma *echelon-form-upt-k-condition1*:
assumes *echelon-form-upt-k A k is-zero-row-upt-k i k A*
shows ¬ (∃*j*. *j*>*i* ∧ ¬ *is-zero-row-upt-k j k A*)
using *assms* **unfolding** *echelon-form-upt-k-def* **by** *auto*

lemma *echelon-form-upt-k-condition1'*:
assumes *echelon-form-upt-k A k is-zero-row-upt-k i k A* **and** *i*<*j*
shows *is-zero-row-upt-k j k A*
using *assms* **unfolding** *echelon-form-upt-k-def* **by** *auto*

lemma *echelon-form-upt-k-condition2*:
assumes *echelon-form-upt-k A k i*<*j*
and ¬ (*is-zero-row-upt-k i k A*) ¬ (*is-zero-row-upt-k j k A*)
shows (*LEAST n*. *A* \$ *i* \$ *n* ≠ *0*) < (*LEAST n*. *A* \$ *j* \$ *n* ≠ *0*)
using *assms* **unfolding** *echelon-form-upt-k-def* **by** *auto*

lemma *echelon-form-upt-k-if-equal*:
assumes *e*: *echelon-form-upt-k A k*
and *eq*: ∀ *a*. ∀ *b*<*from-nat k*. *A* \$ *a* \$ *b* = *B* \$ *a* \$ *b*
and *k*: *k* < *ncols A*
shows *echelon-form-upt-k B k*
unfolding *echelon-form-upt-k-def*
proof (*auto*)
fix *i j* **assume** *zero-iB*: *is-zero-row-upt-k i k B* **and** *ij*: *i* < *j*
have *zero-iA*: *is-zero-row-upt-k i k A*
proof (*unfold is-zero-row-upt-k-def*, *clarify*)
fix *ja*::'*b* **assume** *ja-k*: *to-nat ja* < *k*
have *ja-k2*: *ja* < *from-nat k*
by (*metis* (*full-types*) *add-to-nat-def k from-nat-mono*
ja-k monoid-add-class.add.right-neutral ncols-def to-nat-0)
have *A* \$ *i* \$ *ja* = *B* \$ *i* \$ *ja* **using** *eq ja-k2* **by** *auto*
also have ... = *0* **using** *zero-iB ja-k* **unfolding** *is-zero-row-upt-k-def* **by** *simp*

finally show *A* \$ *i* \$ *ja* = *0* **.**
qed
hence *zero-jA*: *is-zero-row-upt-k j k A* **by** (*metis e echelon-form-upt-k-condition1*
ij)
show *is-zero-row-upt-k j k B*
proof (*unfold is-zero-row-upt-k-def*, *clarify*)
fix *ja*::'*b* **assume** *ja-k*: *to-nat ja* < *k*
have *ja-k2*: *ja* < *from-nat k*
by (*metis* (*full-types*) *add-to-nat-def k from-nat-mono*
ja-k monoid-add-class.add.right-neutral ncols-def to-nat-0)
have *B* \$ *j* \$ *ja* = *A* \$ *j* \$ *ja* **using** *eq ja-k2* **by** *auto*
also have ... = *0* **using** *zero-jA ja-k* **unfolding** *is-zero-row-upt-k-def* **by** *simp*

```
    finally show B $ j $ ja = 0 .
  qed
next
  fix i j
  assume ij: i < j
    and not-zero-iB: ¬ is-zero-row-upt-k i k B
    and not-zero-jB: ¬ is-zero-row-upt-k j k B
  obtain a where Bia: B $ i $ a ≠ 0 and ak: a<from-nat k
    using not-zero-iB k unfolding is-zero-row-upt-k-def ncols-def
      by (metis add-to-nat-def from-nat-mono monoid-add-class.add.right-neutral
to-nat-0)
  have Aia: A $ i $ a ≠ 0 by (metis ak Bia eq)
  obtain b where Bjb: B $ j $ b ≠ 0 and bk: b<from-nat k
    using not-zero-jB k unfolding is-zero-row-upt-k-def ncols-def
      by (metis add-to-nat-def from-nat-mono monoid-add-class.add.right-neutral
to-nat-0)
  have Ajb: A $ j $ b ≠ 0 by (metis bk Bjb eq)
  have not-zero-iA: ¬ is-zero-row-upt-k i k A
    by (metis (full-types) Aia ak is-zero-row-upt-k-def to-nat-le)
  have not-zero-jA: ¬ is-zero-row-upt-k j k A
    by (metis (full-types) Ajb bk is-zero-row-upt-k-def to-nat-le)
  have (LEAST n. A $ i $ n ≠ 0) = (LEAST n. B $ i $ n ≠ 0)
  proof (rule Least-equality)
    have (LEAST n. B $ i $ n ≠ 0) ≤ a by (rule Least-le, simp add: Bia)
    hence least-bi-less: (LEAST n. B $ i $ n ≠ 0) < from-nat k using ak by simp
    thus A $ i $ (LEAST n. B $ i $ n ≠ 0) ≠ 0
      by (metis (mono-tags, lifting) LeastI eq is-zero-row-upt-k-def not-zero-iB)
    fix y assume A $ i $ y ≠ 0
    thus (LEAST n. B $ i $ n ≠ 0) ≤ y
     by (metis (mono-tags, lifting) Least-le dual-order.strict-trans2 eq least-bi-less
linear)
  qed
  moreover have (LEAST n. A $ j $ n ≠ 0) = (LEAST n. B $ j $ n ≠ 0)
  proof (rule Least-equality)
    have (LEAST n. B $ j $ n ≠ 0) ≤ b by (rule Least-le, simp add: Bjb)
    hence least-bi-less: (LEAST n. B $ j $ n ≠ 0) < from-nat k using bk by simp
    thus A $ j $ (LEAST n. B $ j $ n ≠ 0) ≠ 0
      by (metis (mono-tags, lifting) LeastI eq is-zero-row-upt-k-def not-zero-jB)
    fix y assume A $ j $ y ≠ 0
    thus (LEAST n. B $ j $ n ≠ 0) ≤ y
     by (metis (mono-tags, lifting) Least-le dual-order.strict-trans2 eq least-bi-less
linear)
  qed
  moreover have (LEAST n. A $ i $ n ≠ 0) < (LEAST n. A $ j $ n ≠ 0)
    by (rule echelon-form-upt-k-condition2[OF e ij not-zero-iA not-zero-jA])
  ultimately show (LEAST n. B $ i $ n ≠ 0) < (LEAST n. B $ j $ n ≠ 0) by
auto
qed
```

lemma *echelon-form-upt-k-0*: *echelon-form-upt-k A 0*
unfolding *echelon-form-upt-k-def is-zero-row-upt-k-def* **by** *auto*

lemma *echelon-form-condition1*:
assumes *r*: *echelon-form A*
shows ($\forall$ *i*. *is-zero-row i A* $\longrightarrow$ $\neg$ ($\exists$ *j*. *j*>*i* $\wedge$ $\neg$ *is-zero-row j A*))
using *r* **unfolding** *echelon-form-def*
by (*metis echelon-form-upt-k-condition1' is-zero-row-def*)

lemma *echelon-form-condition2*:
assumes *r*: *echelon-form A*
shows ($\forall$ *i*. *i*<*j* $\wedge$ $\neg$ (*is-zero-row i A*) $\wedge$ $\neg$ (*is-zero-row j A*)
$\longrightarrow$ ((*LEAST n*. *A* \$ *i* \$ *n* $\neq$ *0*) < (*LEAST n*. *A* \$ *j* \$ *n* $\neq$ *0*)))
using *r* **unfolding** *echelon-form-def*
by (*metis echelon-form-upt-k-condition2 is-zero-row-def*)

lemma *echelon-form-condition2-explicit*:
assumes *rref-A*: *echelon-form A*
and *i-le*: *i* < *j*
and $\neg$ *is-zero-row i A* **and** $\neg$ *is-zero-row j A*
shows (*LEAST n*. *A* \$ *i* \$ *n* $\neq$ *0*) < (*LEAST n*. *A* \$ *j* \$ *n* $\neq$ *0*)
using *echelon-form-condition2 assms* **by** *blast*

lemma *echelon-form-intro*:
assumes *1*: ($\forall$ *i*. *is-zero-row i A* $\longrightarrow$ $\neg$ ($\exists$ *j*. *j*>*i* $\wedge$ $\neg$ *is-zero-row j A*))
and *2*: ($\forall$ *i j*. *i*<*j* $\wedge$ $\neg$ (*is-zero-row i A*) $\wedge$ $\neg$ (*is-zero-row j A*)
$\longrightarrow$ ((*LEAST n*. *A* \$ *i* \$ *n* $\neq$ *0*) < (*LEAST n*. *A* \$ *j* \$ *n* $\neq$ *0*)))
shows *echelon-form A*
proof (*unfold echelon-form-def*, *rule echelon-form-upt-k-intro*, *auto*)
fix *i j* **assume** *is-zero-row-upt-k i* (*ncols A*) *A* **and** *i* < *j*
thus *is-zero-row-upt-k j* (*ncols A*) *A*
using *1 is-zero-row-imp-is-zero-row-upt* **by** (*metis is-zero-row-def*)
next
fix *i j*
assume *i* < *j* **and** $\neg$ *is-zero-row-upt-k i* (*ncols A*) *A* **and** $\neg$ *is-zero-row-upt-k j* (*ncols A*) *A*
thus (*LEAST n*. *A* \$ *i* \$ *n* $\neq$ *0*) < (*LEAST n*. *A* \$ *j* \$ *n* $\neq$ *0*)
using *2* **by** (*metis is-zero-row-imp-is-zero-row-upt*)
qed

lemma *echelon-form-implies-echelon-form-upt*:
fixes *A*::*'a*::{*bezout-ring*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *rref*: *echelon-form A*
shows *echelon-form-upt-k A k*
proof (*rule echelon-form-upt-k-intro*)
show $\forall$ *i*. *is-zero-row-upt-k i k A* $\longrightarrow$ $\neg$ ($\exists$ *j*>*i*. $\neg$ *is-zero-row-upt-k j k A*)
proof (*auto*, *rule ccontr*)
fix *i j* **assume** *zero-i-k*: *is-zero-row-upt-k i k A* **and** *i-less-j*: *i* < *j*

```
      and not-zero-j-k:¬ is-zero-row-upt-k j k A
    have not-zero-j: ¬ is-zero-row j A
      using is-zero-row-imp-is-zero-row-upt not-zero-j-k by blast
    hence not-zero-i: ¬ is-zero-row i A
      using echelon-form-condition1[OF rref] i-less-j by blast
    have Least-less: (LEAST n. A $ i $ n ≠ 0) < (LEAST n. A $ j $ n ≠ 0)
     by (rule echelon-form-condition2-explicit[OF rref i-less-j not-zero-i not-zero-j])
    moreover have (LEAST n. A $ j $ n ≠ 0) < (LEAST n. A $ i $ n ≠ 0)
    proof (rule LeastI2-ex)
      show ∃ a. A $ i $ a ≠ 0
        using not-zero-i unfolding is-zero-row-def is-zero-row-upt-k-def by fast
      fix x assume Aix-not-0: A $ i $ x ≠ 0
      have k-less-x: k ≤ to-nat x
        using zero-i-k Aix-not-0 unfolding is-zero-row-upt-k-def by force
      hence k-less-ncols: k < ncols A
        unfolding ncols-def using to-nat-less-card[of x] by simp
      obtain s where Ajs-not-zero: A $ j $ s ≠ 0 and s-less-k: to-nat s < k
        using not-zero-j-k unfolding is-zero-row-upt-k-def by blast
      have (LEAST n. A $ j $ n ≠ 0) ≤ s using Ajs-not-zero Least-le by fast
      also have ... = from-nat (to-nat s) unfolding from-nat-to-nat-id ..
      also have ... < from-nat k
        by (rule from-nat-mono[OF s-less-k k-less-ncols[unfolded ncols-def]])
      also have ... ≤ x using k-less-x leD not-le-imp-less to-nat-le by fast
      finally show (LEAST n. A $ j $ n ≠ 0) < x .
    qed
    ultimately show False by fastforce
  qed
  show ∀ i j. i < j ∧ ¬ is-zero-row-upt-k i k A ∧ ¬ is-zero-row-upt-k j k A
    ⟶ (LEAST n. A $ i $ n ≠ 0) < (LEAST n. A $ j $ n ≠ 0)
      using echelon-form-condition2[OF rref] is-zero-row-imp-is-zero-row-upt by
blast
qed

lemma upper-triangular-upt-k-def':
  assumes ∀ i j. to-nat j ≤ k ∧ A $ i $ j ≠ 0 ⟶ j≥i
  shows upper-triangular-upt-k A k
  using assms
  unfolding upper-triangular-upt-k-def
  by (metis leD linear)


lemma echelon-form-imp-upper-triagular-upt:
  fixes A::'a::{bezout-ring}^'n::{mod-type}^'n::{mod-type}
  assumes echelon-form A
  shows upper-triangular-upt-k A k
proof (induct k)
  case 0
  show ?case unfolding upper-triangular-upt-k-def by simp
next
```

```
case (Suc k)
show ?case
  unfolding upper-triangular-upt-k-def
proof (clarify)
  fix i j::'n assume j-less-i: j < i and j-less-suc-k: to-nat j < Suc k
  show A $ i $ j = 0
  proof (cases to-nat j < k)
    case True
    thus ?thesis
      using Suc.hyps
      unfolding upper-triangular-upt-k-def using j-less-i True by auto
  next
    case False
    hence j-eq-k: to-nat j = k using j-less-suc-k by simp
    hence j-eq-k2: from-nat k = j by (metis from-nat-to-nat-id)
    have rref-suc: echelon-form-upt-k A (Suc k)
      by (metis assms echelon-form-implies-echelon-form-upt)
    have zero-j-k: is-zero-row-upt-k j k A
      unfolding is-zero-row-upt-k-def
      by (metis (opaque-lifting, mono-tags) Suc.hyps leD le-less-linear
        j-eq-k to-nat-mono' upper-triangular-upt-k-def)
    hence zero-i-k: is-zero-row-upt-k i k A
     by (metis (poly-guards-query) assms echelon-form-implies-echelon-form-upt
        echelon-form-upt-k-condition1' j-less-i)
    show ?thesis
    proof (cases A $ j $ j = 0)
      case True
      have is-zero-row-upt-k j (Suc k) A
        by (rule is-zero-row-upt-k-suc[OF zero-j-k], simp add: True j-eq-k2)
      hence is-zero-row-upt-k i (Suc k) A
        by (metis echelon-form-upt-k-condition1' j-less-i rref-suc)
      thus ?thesis by (metis is-zero-row-upt-k-def j-eq-k lessI)
    next
      case False note Ajj-not-zero=False
      hence not-zero-j:¬ is-zero-row-upt-k j (Suc k) A
        by (metis is-zero-row-upt-k-def j-eq-k lessI)
      show ?thesis
      proof (rule ccontr)
        assume Aij-not-zero: A $ i $ j ≠ 0
        hence not-zero-i: ¬ is-zero-row-upt-k i (Suc k) A
          by (metis is-zero-row-upt-k-def j-eq-k lessI)
        have Least-eq: (LEAST n. A $ i $ n ≠ 0) = from-nat k
        proof (rule Least-equality)
          show A $ i $ from-nat k ≠ 0 using Aij-not-zero j-eq-k2 by simp
          show ⋀y. A $ i $ y ≠ 0 ⟹ from-nat k ≤ y
               by (metis (full-types) is-zero-row-upt-k-def not-le-imp-less to-nat-le
zero-i-k)
        qed
        moreover have Least-eq2: (LEAST n. A $ j $ n ≠ 0) = from-nat k
```

proof (*rule Least-equality*)
show *A* $ *j* $ *from-nat k* ≠ *0* **using** *Ajj-not-zero j-eq-k2* **by** *simp*
show ⋀*y*. *A* $ *j* $ *y* ≠ *0* ⟹ *from-nat k* ≤ *y*
by (*metis* (*full-types*) *is-zero-row-upt-k-def not-le-imp-less to-nat-le zero-j-k*)
qed
ultimately show *False*
using *echelon-form-upt-k-condition2*[*OF rref-suc j-less-i not-zero-j not-zero-i*]
by *simp*
qed
qed
qed
qed
qed

A matrix in echelon form is upper triangular.

lemma *echelon-form-imp-upper-triagular*:
fixes *A*::'*a*::{*bezout-ring*}^'*n*::{*mod-type*}^'*n*::{*mod-type*}
assumes *echelon-form A*
shows *upper-triangular A*
using *echelon-form-imp-upper-triagular-upt*[*OF assms*]
by (*metis upper-triangular-upt-imp-upper-triangular*)

lemma *echelon-form-upt-k-interchange*:
fixes *A*::'*a*::{*bezout-ring*}^'*c*::{*mod-type*}^'*b*::{*mod-type*}
assumes *e*: *echelon-form-upt-k A k*
and *zero-ikA*: *is-zero-row-upt-k* (*from-nat i*) *k A*
and *Amk-not-0*: *A* $ *m* $ *from-nat k* ≠ *0*
and *i-le-m*: (*from-nat i*)≤*m*
and *k*: *k*<*ncols A*
shows *echelon-form-upt-k* (*interchange-rows A* (*from-nat i*)
(*LEAST n*. *A* $ *n* $ *from-nat k* ≠ *0* ∧ (*from-nat i*) ≤ *n*)) *k*
proof (*rule echelon-form-upt-k-if-equal*[*OF e - k*], *auto*)
fix *a* **and** *b*::'*c*
assume *b*: *b* < *from-nat k*
let *?least* = (*LEAST n*. *A* $ *n* $ *from-nat k* ≠ *0* ∧ (*from-nat i*) ≤ *n*)
let *?interchange* = (*interchange-rows A* (*from-nat i*) *?least*)
have (*from-nat i*)≤*?least* **by** (*metis* (*mono-tags*, *lifting*) *Amk-not-0 LeastI-ex i-le-m*)
hence *zero-leastkA*: *is-zero-row-upt-k ?least k A*
using *echelon-form-upt-k-condition1*[*OF e zero-ikA*]
by (*metis* (*poly-guards-query*) *dual-order.strict-iff-order zero-ikA*)
show *A* $ *a* $ *b* = *?interchange* $ *a* $ *b*
proof (*cases a*=*from-nat i*)
case *True*
hence *?interchange* $ *a* $ *b* = *A* $ *?least* $ *b* **unfolding** *interchange-rows-def* **by** *auto*

```
    also have ... = 0 using zero-leastkA unfolding is-zero-row-upt-k-def
      by (metis (mono-tags) b to-nat-le)
    finally have ?interchange $ a $ b = 0 .
    moreover have A $ a $ b = 0
      by (metis True b is-zero-row-upt-k-def to-nat-le zero-ikA)
    ultimately show ?thesis by simp
  next
    case False note a-not-i=False
    show ?thesis
    proof (cases a=?least)
      case True
        hence ?interchange $ a $ b = A $ (from-nat i) $ b unfolding interchange-rows-def by auto
      also have ... = 0 using zero-ikA  unfolding is-zero-row-upt-k-def
        by (metis (poly-guards-query) b to-nat-le)
      finally have ?interchange $ a $ b = 0 .
     moreover have A $ a $ b = 0 by (metis True b is-zero-row-upt-k-def to-nat-le zero-leastkA)
      ultimately show ?thesis by simp
    next
      case False
      thus ?thesis using a-not-i unfolding interchange-rows-def by auto
    qed
  qed
qed
```

There are similar theorems to the following ones in the Gauss-Jordan developments, but for matrices in reduced row echelon form. It is possible to prove that reduced row echelon form implies echelon form. Then the theorems in the Gauss-Jordan development could be obtained with ease.

```
lemma greatest-less-zero-row:
  fixes A::'a::{bezout-ring}^'cols::{mod-type}^'rows::{finite, wellorder}
  assumes r: echelon-form-upt-k A k
  and zero-i: is-zero-row-upt-k i k A
  and not-all-zero: ¬ (∀ a. is-zero-row-upt-k a k A)
  shows (GREATEST m. ¬ is-zero-row-upt-k m k A) < i
proof (rule ccontr)
  assume not-less-i: ¬ (GREATEST m. ¬ is-zero-row-upt-k m k A) < i
  have i-less-greatest: i < (GREATEST m. ¬ is-zero-row-upt-k m k A)
    by (metis not-less-i neq-iff GreatestI not-all-zero zero-i)
  have is-zero-row-upt-k (GREATEST m. ¬ is-zero-row-upt-k m k A) k A
    using r zero-i i-less-greatest unfolding echelon-form-upt-k-def by blast
  thus False using GreatestI-ex not-all-zero by fast
qed

lemma greatest-ge-nonzero-row':
  fixes A::'a::{bezout-ring}^'cols::{mod-type}^'rows::{mod-type}
  assumes r: echelon-form-upt-k A k
  and i: i ≤ (GREATEST m. ¬ is-zero-row-upt-k m k A)
```

and *not-all-zero*: $\neg$ ($\forall$ *a*. *is-zero-row-upt-k a k A*)
shows $\neg$ *is-zero-row-upt-k i k A*
using *greatest-less-zero-row*[*OF r*] *i not-all-zero* **by** *fastforce*

lemma *rref-imp-ef*:
fixes *A*::*'a*::{*bezout-ring*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *rref*: *reduced-row-echelon-form A*
shows *echelon-form A*
proof (*rule echelon-form-intro*)
show $\forall$ *i*. *is-zero-row i A* $\longrightarrow$ $\neg$ ($\exists$ *j*>*i*. $\neg$ *is-zero-row j A*)
by (*simp add*: *rref rref-condition1*)
show $\forall$ *i j*. *i* < *j* $\wedge$ $\neg$ *is-zero-row i A* $\wedge$ $\neg$ *is-zero-row j A*
$\longrightarrow$ (*LEAST n*. *A* \$ *i* \$ *n* $\neq$ *0*) < (*LEAST n*. *A* \$ *j* \$ *n* $\neq$ *0*)
by (*simp add*: *rref-condition3-equiv rref*)
qed

3.2 Computing the echelon form of a matrix

3.2.1 Demonstration over principal ideal rings

Important remark:

We want to prove that there exist the echelon form of any matrix whose elements belong to a bezout domain. In addition, we want to compute the echelon form, so we will need computable gcd and bezout operations which is possible over euclidean domains. Our approach consists of demonstrating the correctness over bezout domains and executing over euclidean domains.

To do that, we have studied several options:

1. We could define a gcd in bezout rings (*bezout-ring-gcd*) as follows: *gcd-bezout-ring a b* = (*SOME d*. *d dvd a* $\wedge$ *d dvd b* $\wedge$ ($\forall$ *d'*. *d' dvd a* $\wedge$ *d' dvd b* $\longrightarrow$ *d' dvd d*))

   And then define an algorithm that computes the Echelon Form using such a definition to the gcd. This would allow us to prove the correctness over bezout rings, but we would not be able to execute over euclidean rings because it is not possible to demonstrate a (code) lemma stating that (*gcd-bezout-ring a b*) = *gcd-eucl a b* (the gcd is not unique over bezout rings and GCD rings).

2. Create a *bezout-ring-norm* class and define a gcd normalized over bezout rings: *definition gcd-bezout-ring-norm a b* = *gcd-bezout-ring a b div normalisation-factor* (*gcd-bezout-ring a b*)

   Then, one could demonstrate a (code) lemma stating that: (*gcd-bezout-ring-norm a b*) = *gcd-eucl a b* This allows us to execute the gcd function, but with bezout it is not possible.

3. The third option (and the chosen one) consists of defining the algorithm over bezout domains and parametrizing the algorithm by a *bezout* operation which must satisfy suitable properties (i.e *is-bezout-ext bezout*). Then we can prove the correctness over bezout domains and we will execute over euclidean domains, since we can prove that the operation *euclid-ext2* is an executable operation which satisfies *is-bezout-ext euclid-ext2*.

3.2.2 Definition of the algorithm

context *bezout-ring*
begin

definition
bezout-matrix :: $'a^{\frown}'cols^{\frown}'rows \Rightarrow 'rows \Rightarrow 'rows \Rightarrow 'cols$
$\Rightarrow ('a \Rightarrow 'a \Rightarrow ('a \times 'a \times 'a \times 'a \times 'a)) \Rightarrow 'a^{\frown}'rows^{\frown}'rows$
where
bezout-matrix A a b j bezout = (χ *x y*.
(*let*
(*p*, *q*, *u*, *v*, *d*) = *bezout* (*A* \$ *a* \$ *j*) (*A* \$ *b* \$ *j*)
in
if x = *a* $\wedge$ *y* = *a then p else*
if x = *a* $\wedge$ *y* = *b then q else*
if x = *b* $\wedge$ *y* = *a then u else*
if x = *b* $\wedge$ *y* = *b then v else*
if x = *y then 1 else 0*))

end

primrec
bezout-iterate :: $'a::\{bezout\text{-}ring\}^{\frown}'cols^{\frown}'rows::\{mod\text{-}type\}$
$\Rightarrow nat \Rightarrow 'rows::\{mod\text{-}type\}$
$\Rightarrow 'cols \Rightarrow ('a \Rightarrow 'a \Rightarrow ('a \times 'a \times 'a \times 'a \times 'a)) \Rightarrow$
$'a^{\frown}'cols^{\frown}'rows::\{mod\text{-}type\}$
where *bezout-iterate A 0 i j bezout* = *A*
| *bezout-iterate A* (*Suc n*) *i j bezout* =
(*if* (*Suc n*) $\leq$ *to-nat i then A else*
bezout-iterate (*bezout-matrix A i* (*from-nat* (*Suc n*)) *j bezout* ** *A*) *n i j bezout*)

If every element in column k over index i are equal to zero, the same input is returned. If every element over i is equal to zero, except the pivot, the algorithm does nothing, but pivot i is increased in a unit. Finally, if there is a position n whose coefficient is different from zero, its row is interchanged with row i and the bezout coefficients are used to produce a zero in its position.

definition

echelon-form-of-column-k bezout A′ k =
  (*let* (*A*, *i*) = *A′*
   in if (∀ *m*≥*from-nat i*. *A* $ *m* $ *from-nat k* = *0*) ∨ (*i* = *nrows A*) *then* (*A*, *i*) *else*
     if (∀ *m*>*from-nat i*. *A* $ *m* $ *from-nat k* = *0*) *then* (*A*, *i* + *1*) *else*
       let n = (*LEAST n*. *A* $ *n* $ *from-nat k* ≠ *0* ∧ *from-nat i* ≤ *n*);
         interchange-A = *interchange-rows A* (*from-nat i*) *n*
       in
       (*bezout-iterate* (*interchange-A*) (*nrows A* − *1*) (*from-nat i*) (*from-nat k*) *bezout*, *i* + *1*))

definition *echelon-form-of-upt-k A k bezout* = (*fst* (*foldl* (*echelon-form-of-column-k bezout*) (*A*,*0*) [*0*..<*Suc k*]))
definition *echelon-form-of A bezout* = *echelon-form-of-upt-k A* (*ncols A* − *1*) *bezout*

3.2.3 The executable definition:

context *euclidean-space*
begin

definition [*code-unfold*]: *echelon-form-of-euclidean A* = *echelon-form-of A euclid-ext2*

end

3.2.4 Properties of the bezout matrix

lemma *bezout-matrix-works1*:
  assumes *ib*: *is-bezout-ext bezout*
  and *a-not-b*: *a* ≠ *b*
  shows (*bezout-matrix A a b j bezout* ∗∗ *A*) $ *a* $ *j* = *snd* (*snd* (*snd* (*snd* (*bezout* (*A* $ *a* $ *j*) (*A* $ *b* $ *j*)))))
proof (*unfold matrix-matrix-mult-def bezout-matrix-def Let-def*, *simp*)
  let *?a* = (*A* $ *a* $ *j*)
  let *?b* = (*A* $ *b* $ *j*)
  let *?z* = *bezout* (*A* $ *a* $ *j*) (*A* $ *b* $ *j*)
  obtain *p q u v d* **where** *bz*: (*p*, *q*, *u*, *v*, *d*) = *?z* **by** (*cases ?z*, *auto*)
  from *ib* **have** *foo*: (⋀*a b*. *let* (*p*, *q*, *u*, *v*, *gcd-a-b*) = *bezout a b*
         in p ∗ *a* + *q* ∗ *b* = *gcd-a-b* ∧
           gcd-a-b dvd a ∧
          gcd-a-b dvd b ∧ (∀ *d′*. *d′ dvd a* ∧ *d′ dvd b* ⟶ *d′ dvd gcd-a-b*) ∧ *gcd-a-b* ∗ *u* = − *b* ∧ *gcd-a-b* ∗ *v* = *a*)
        using *is-bezout-ext-def* [*of bezout*] **by** *simp*
  have *foo*: *p* ∗ *?a* + *q* ∗ *?b* = *d* ∧ *d dvd ?a* ∧
         d dvd ?b ∧ (∀ *d′*. *d′ dvd ?a* ∧ *d′ dvd ?b* ⟶ *d′ dvd d*) ∧ *d* ∗ *u* = − *?b* ∧ *d* ∗ *v* = *?a*
         using *ib* **using** *is-bezout-ext-def* **using** *bz* [*symmetric*]
         using *foo* [*of ?a ?b*] **by** *fastforce*
  have *pa-bq-d*: *p* ∗ *?a* + *?b* ∗ *q* = *d* **using** *foo* **by** (*auto simp add*: *mult.commute*)
  define *f* **where** *f k* = (*if k* = *a then p*

else if $a = a \wedge k = b$ *then* q
else if $a = b \wedge k = a$ *then* u
else if $a = b \wedge k = b$ *then* v
else if $a = k$ *then 1 else 0*) $*$ A \$ k \$ j **for** k
have *UNIV-rw*: *UNIV* = *insert b* (*insert a* (*UNIV* − {*a*} − {*b*})) **by** *auto*
have *sum-rw*: *sum f* (*insert a* (*UNIV* − {*a*} − {*b*})) = *f a* + *sum f* (*UNIV* − {*a*} − {*b*})
by (*rule sum.insert*, *auto*)
have *sum0*: *sum f* (*UNIV* − {*a*} − {*b*}) = *0* **by** (*rule sum.neutral*, *simp add*: *f-def*)
have ($\sum$ *k*∈*UNIV*.
(*case bezout* (*A* \$ *a* \$ *j*) (*A* \$ *b* \$ *j*) *of*
(*p*, *q*, *u*, *v*, *d*) ⇒
if k = *a then p*
else if a = *a* ∧ *k* = *b then q*
else if a = *b* ∧ *k* = *a then u else if a* = *b* ∧ *k* = *b then v else if a* = *k then 1 else 0*) ∗
A \$ *k* \$ *j*) = ($\sum$ *k*∈*UNIV*.
(*if k* = *a then p*
else if a = *a* ∧ *k* = *b then q*
else if a = *b* ∧ *k* = *a then u else if a* = *b* ∧ *k* = *b then v else if a* = *k then 1 else 0*) ∗
A \$ *k* \$ *j*) **unfolding** *bz* [*symmetric*] **by** *auto*
also have ... = *sum f UNIV* **unfolding** *f-def* **..**
also have *sum f UNIV* = *sum f* (*insert b* (*insert a* (*UNIV* − {*a*} − {*b*}))) **using** *UNIV-rw* **by** *simp*
also have ... = *f b* + *sum f* (*insert a* (*UNIV* − {*a*} − {*b*}))
by (*rule sum.insert*, *auto*, *metis a-not-b*)
also have ... = *f b* + *f a* **unfolding** *sum-rw sum0* **by** *simp*
also have ... = *d*
unfolding *f-def* **using** *a-not-b bz* [*symmetric*] **by** (*auto*, *metis add.commute mult.commute pa-bq-d*)
also have ... = *snd* (*snd* (*snd* (*snd* (*bezout* (*A* \$ *a* \$ *j*) (*A* \$ *b* \$ *j*)))))
using *bz* **by** (*metis snd-conv*)
finally show ($\sum$ *k*∈*UNIV*.
(*case bezout* (*A* \$ *a* \$ *j*) (*A* \$ *b* \$ *j*) *of*
(*p*, *q*, *u*, *v*, *d*) ⇒
if k = *a then p*
else if a = *a* ∧ *k* = *b then q*
else if a = *b* ∧ *k* = *a then u else if a* = *b* ∧ *k* = *b then v else if a* = *k then 1 else 0*) ∗
A \$ *k* \$ *j*) =
snd (*snd* (*snd* (*snd* (*bezout* (*A* \$ *a* \$ *j*) (*A* \$ *b* \$ *j*))))) **unfolding** *f-def* **by** *simp*
qed

lemma *bezout-matrix-not-zero*:
assumes *ib*: *is-bezout-ext bezout*
and *a-not-b*: *a* ≠ *b*
and *Aaj*: *A* \$ *a* \$ *j* ≠ *0*

shows (*bezout-matrix A a b j bezout* ∗∗ *A*) $ *a* $ *j* ≠ *0*
proof −
have (*bezout-matrix A a b j bezout* ∗∗ *A*) $ *a* $ *j* = *snd* (*snd* (*snd*(*snd* (*bezout* (*A* $ *a* $ *j*) (*A* $ *b* $ *j*)))))
using *bezout-matrix-works1*[*OF ib a-not-b*] **.**
also have ... = (λ*a b*. (*case bezout a b of* (-, -,- ,-,*gcd′*) ⇒ (*gcd′*))) (*A* $ *a* $ *j*) (*A* $ *b* $ *j*)
by (*simp add*: *split-beta*)
also have ... ≠ *0* **using** *gcd′-zero*[*OF is-gcd-is-bezout-ext*[*OF ib*]] *Aaj* **by** *simp*
finally show *?thesis* **.**
qed

lemma *ua-vb-0*:
fixes *a*::′*a*::*bezout-domain*
assumes *ib*: *is-bezout-ext bezout* **and** *nz*: *snd* (*snd* (*snd* (*snd* (*bezout a b*)))) ≠ *0*
shows *fst* (*snd* (*snd* (*bezout a b*))) ∗ *a* + *fst* (*snd* (*snd* (*snd* (*bezout a b*)))) ∗ *b* = *0*
proof−
obtain *p q u v d* **where** *bz*: (*p*, *q*, *u*, *v*, *d*) = *bezout a b* **by** (*cases bezout a b*, *auto*)
from *ib* **have** *foo*: (⋀*a b*. *let* (*p*, *q*, *u*, *v*, *gcd-a-b*) = *bezout a b*
in p ∗ *a* + *q* ∗ *b* = *gcd-a-b* ∧
gcd-a-b dvd a ∧
gcd-a-b dvd b ∧ (∀ *d′*. *d′ dvd a* ∧ *d′ dvd b* ⟶ *d′ dvd gcd-a-b*) ∧ *gcd-a-b* ∗ *u* = − *b* ∧ *gcd-a-b* ∗ *v* = *a*)
using *is-bezout-ext-def* [*of bezout*] **by** *simp*
have *p* ∗ *a* + *q* ∗ *b* = *d* ∧ *d dvd a* ∧
d dvd b ∧ (∀ *d′*. *d′ dvd a* ∧ *d′ dvd b* ⟶ *d′ dvd d*) ∧ *d* ∗ *u* = − *b* ∧ *d* ∗ *v* = *a*
using *foo* [*of a b*] **using** *bz* **by** *fastforce*
hence *dub*: *d* ∗ *u* = − *b* **and** *dva*: *d* ∗ *v* = *a* **by** (*simp-all*)
hence *d* ∗ *u* ∗ *a* + *d* ∗ *v* ∗ *b* = *0*
using *eq-neg-iff-add-eq-0 mult.commute mult-minus-left* **by** *auto*
hence *u* ∗ *a* + *v* ∗ *b* = *0*
by (*metis* (*no-types*, *lifting*) *dub dva minus-minus mult-minus-left*
neg-eq-iff-add-eq-0 semiring-normalization-rules(*18*) *semiring-normalization-rules*(*7*))
thus *?thesis* **using** *bz* [*symmetric*]
by *simp*
qed

lemma *bezout-matrix-works2*:
fixes *A*::′*a*::*bezout-domain*^′*cols*^′*rows*
assumes *ib*: *is-bezout-ext bezout*
and *a-not-b*: *a* ≠ *b*
and *not-0*: *A* $ *a* $ *j* ≠ *0* ∨ *A* $ *b* $ *j* ≠ *0*
shows (*bezout-matrix A a b j bezout* ∗∗ *A*) $ *b* $ *j* = *0*
proof (*unfold matrix-matrix-mult-def bezout-matrix-def Let-def*, *auto*)
let *?a* = (*A* $ *a* $ *j*)
let *?b* = (*A* $ *b* $ *j*)

let *?z* = *bezout* (*A* $ *a* $ *j*) (*A* $ *b* $ *j*)
from *ib* **have** *foo*: ($\bigwedge$*a b*. *let* (*p*, *q*, *u*, *v*, *gcd-a-b*) = *bezout a b*
in p ∗ *a* + *q* ∗ *b* = *gcd-a-b* ∧
gcd-a-b dvd a ∧
gcd-a-b dvd b ∧ (∀ *d'*. *d' dvd a* ∧ *d' dvd b* ⟶ *d' dvd gcd-a-b*) ∧ *gcd-a-b* ∗ *u* = − *b* ∧ *gcd-a-b* ∗ *v* = *a*)
using *is-bezout-ext-def* [*of bezout*] **by** *simp*
obtain *p q u v d* **where** *bz*: (*p*, *q*, *u*, *v*, *d*) = *?z* **by** (*cases ?z*, *auto*)
hence *pib*: *p* ∗ *?a* + *q* ∗ *?b* = *d* ∧ *d dvd ?a* ∧
d dvd ?b ∧ (∀ *d'*. *d' dvd ?a* ∧ *d' dvd ?b* ⟶ *d' dvd d*) ∧ *d* ∗ *u* = − *?b* ∧ *d* ∗ *v* = *?a*
using *foo* [*of ?a ?b*] **by** *fastforce*
hence *pa-bq-d*: *p* ∗ *?a* + *?b* ∗ *q* = *d* **by** (*simp add*: *mult.commute*)
have *d-dvd-a*: *d dvd ?a* **using** *pib* **by** *auto*
have *d-dvd-b*: *d dvd* −*?b* **using** *pib* **by** *auto*
have *pa-bq-d*: *p* ∗ *?a* + *?b* ∗ *q* = *d* **using** *pa-bq-d* **by** (*simp add*: *mult.commute*)
define *f* **where** *f k* = (*if b* = *a* ∧ *k* = *a then p*
else if b = *a* ∧ *k* = *b then q*
else if b = *b* ∧ *k* = *a then u*
else if b = *b* ∧ *k* = *b then v else if b* = *k then 1 else 0*) ∗
A $ *k* $ *j* **for** *k*
have *UNIV-rw*: *UNIV* = *insert b* (*insert a* (*UNIV* − {*a*} − {*b*})) **by** *auto*
have *sum-rw*: *sum f* (*insert a* (*UNIV* − {*a*} − {*b*})) = *f a* + *sum f* (*UNIV* − {*a*} − {*b*})
by (*rule sum.insert*, *auto*)
have *sum0*: *sum f* (*UNIV* − {*a*} − {*b*}) = *0* **by** (*rule sum.neutral*, *simp add*: *f-def*)
have ($\sum$*k*∈*UNIV*.
(*case bezout* (*A* $ *a* $ *j*) (*A* $ *b* $ *j*) *of*
(*p*, *q*, *u*, *v*, *d*) ⇒
if b = *a* ∧ *k* = *a then p*
else if b = *a* ∧ *k* = *b then q*
else if b = *b* ∧ *k* = *a then u else if b* = *b* ∧ *k* = *b then v else if b* = *k then 1 else 0*) ∗
A $ *k* $ *j*) = *sum f UNIV* **unfolding** *f-def bz* [*symmetric*] **by** *simp*
also have *sum f UNIV* = *sum f* (*insert b* (*insert a* (*UNIV* − {*a*} − {*b*}))) **using** *UNIV-rw* **by** *simp*
also have ... = *f b* + *sum f* (*insert a* (*UNIV* − {*a*} − {*b*}))
by (*rule sum.insert*, *auto*, *metis a-not-b*)
also have ... = *f b* + *f a* **unfolding** *sum-rw sum0* **by** *simp*
also have ... = *v* ∗ *?b* + *u* ∗ *?a* **unfolding** *f-def* **using** *a-not-b* **by** *auto*
also have ... = *u* ∗ *?a* + *v* ∗ *?b* **by** *auto*
also have ... = *0*
using *ua-vb-0* [*OF ib*] *bz*
by (*metis fst-conv minus-minus minus-zero mult-eq-0-iff pib snd-conv*)
finally show ($\sum$*k*∈*UNIV*.
(*case bezout* (*A* $ *a* $ *j*) (*A* $ *b* $ *j*) *of*
(*p*, *q*, *u*, *v*, *d*) ⇒
if b = *a* ∧ *k* = *a then p*

else if b = a ∧ k = b then q
else if b = b ∧ k = a then u else if b = b ∧ k = b then v else if b = k then 1 else 0) ∗
A $ k $ j) =
0 **.**
qed

lemma *bezout-matrix-preserves-previous-columns*:
assumes *ib*: *is-bezout-ext bezout*
and *i-not-j*: *i ≠ j*
and *Aik*: *A $ i $ k ≠ 0*
and *b-k*: *b<k*
and *i*: *is-zero-row-upt-k i* (*to-nat k*) *A* **and** *j*: *is-zero-row-upt-k j* (*to-nat k*) *A*
shows (*bezout-matrix A i j k bezout* ∗∗ *A*) *$ a $ b* = *A $ a $ b*
unfolding *matrix-matrix-mult-def* **unfolding** *bezout-matrix-def Let-def*
proof (*auto*)
let *?B* = *bezout-matrix A i j k bezout*
let *?i* = (*A $ i $ k*)
let *?j* = (*A $ j $ k*)
let *?z* = *bezout* (*A $ i $ k*) (*A $ j $ k*)
from *ib* **have** *foo*: (⋀*a b. let* (*p, q, u, v, gcd-a-b*) = *bezout a b*
in p ∗ a + q ∗ b = gcd-a-b ∧
gcd-a-b dvd a ∧
gcd-a-b dvd b ∧ (∀ *d′. d′ dvd a ∧ d′ dvd b ⟶ d′ dvd gcd-a-b*) ∧ *gcd-a-b ∗ u = − b ∧ gcd-a-b ∗ v = a*)
using *is-bezout-ext-def* [*of bezout*] **by** *simp*
obtain *p q u v d* **where** *bz*: (*p, q, u, v, d*) = *?z* **by** (*cases ?z, auto*)
have *Aib*: *A $ i $ b = 0* **by** (*metis b-k i is-zero-row-upt-k-def to-nat-mono*)
have *Ajb*: *A $ j $ b = 0* **by** (*metis b-k j is-zero-row-upt-k-def to-nat-mono*)
define *f* **where** *f ka* = (*if a = i ∧ ka = i then p*
else if a = i ∧ ka = j then q
else if a = j ∧ ka = i then u
else if a = j ∧ ka = j then v else if a = ka then 1 else 0) ∗ *A $ ka $ b* **for** *ka*
show (∑ *ka*∈*UNIV*.
(*case bezout* (*A $ i $ k*) (*A $ j $ k*) *of*
(*p, q, u, v, d*) ⇒
if a = i ∧ ka = i then p
else if a = i ∧ ka = j then q
else if a = j ∧ ka = i then u else if a = j ∧ ka = j then v else if a = ka then 1 else 0) ∗
A $ ka $ b) =
A $ a $ b
proof (*cases a=i*)
case *True*
have (∑ *ka*∈*UNIV*.
(*case bezout* (*A $ i $ k*) (*A $ j $ k*) *of*
(*p, q, u, v, d*) ⇒
if a = i ∧ ka = i then p

else if $a = i \wedge ka = j$ *then* q
else if $a = j \wedge ka = i$ *then* u *else if* $a = j \wedge ka = j$ *then* v *else if* $a = ka$ *then 1 else 0*) $*$
A \$ *ka* \$ *b*) = *sum f UNIV* **unfolding** *f-def bz* [*symmetric*] **by** *simp*
also have *sum f UNIV* = *0* **by** (*rule sum.neutral*, *auto simp add*: *Aib Ajb f-def True i-not-j*)
also have ... = *A* \$ *a* \$ *b* **unfolding** *True* **using** *Aib* **by** *simp*
finally show *?thesis* **.**
next
case *False* **note** *a-not-i=False*
show *?thesis*
proof (*cases a=j*)
case *True*
have ($\sum$ *ka*$\in$*UNIV*.
(*case bezout* (*A* \$ *i* \$ *k*) (*A* \$ *j* \$ *k*) *of*
(*p*, *q*, *u*, *v*, *d*) $\Rightarrow$
if $a = i \wedge ka = i$ *then* p
else if $a = i \wedge ka = j$ *then* q
else if $a = j \wedge ka = i$ *then* u *else if* $a = j \wedge ka = j$ *then* v *else if* $a = ka$ *then 1 else 0*) $*$
A \$ *ka* \$ *b*) = *sum f UNIV* **unfolding** *f-def bz* [*symmetric*] **by** *simp*
also have *sum f UNIV* = *0* **by** (*rule sum.neutral*, *auto simp add*: *Aib Ajb f-def True i-not-j*)
also have ... = *A* \$ *a* \$ *b* **unfolding** *True* **using** *Ajb* **by** *simp*
finally show *?thesis* **.**
next
case *False*
have *UNIV-rw*: *UNIV* = *insert j* (*insert i* (*UNIV* − {*i*} − {*j*})) **by** *auto*
have *UNIV-rw2*: *UNIV* − {*i*} − {*j*} = *insert a* (*UNIV* − {*i*} − {*j*}−{*a*})
using *False a-not-i* **by** *auto*
have *sum0*: *sum f* (*UNIV* − {*i*} − {*j*} − {*a*}) = *0*
by (*rule sum.neutral*, *simp add*: *f-def*)
have ($\sum$ *ka*$\in$*UNIV*.
(*case bezout* (*A* \$ *i* \$ *k*) (*A* \$ *j* \$ *k*) *of*
(*p*, *q*, *u*, *v*, *d*) $\Rightarrow$
if $a = i \wedge ka = i$ *then* p
else if $a = i \wedge ka = j$ *then* q
else if $a = j \wedge ka = i$ *then* u *else if* $a = j \wedge ka = j$ *then* v *else if* $a = ka$ *then 1 else 0*) $*$
A \$ *ka* \$ *b*) = *sum f UNIV* **unfolding** *f-def bz* [*symmetric*] **by** *simp*
also have *sum f UNIV* = *sum f* (*insert j* (*insert i* (*UNIV* − {*i*} − {*j*})))
using *UNIV-rw* **by** *simp*
also have ... = *f j* + *sum f* (*insert i* (*UNIV* − {*i*} − {*j*}))
by (*rule sum.insert*, *auto*, *metis i-not-j*)
also have ... = *sum f* (*insert i* (*UNIV* − {*i*} − {*j*}))
unfolding *f-def* **using** *False a-not-i* **by** *auto*
also have ... = *f i* + *sum f* (*UNIV* − {*i*} − {*j*}) **by** (*rule sum.insert*, *auto*)
also have ... = *sum f* (*UNIV* − {*i*} − {*j*}) **unfolding** *f-def* **using** *False a-not-i* **by** *auto*

also have ... = *sum f* (*insert a* (*UNIV* − {*i*} − {*j*} − {*a*})) **using** *UNIV-rw2* **by** *simp*
also have ... = *f a* + *sum f* (*UNIV* − {*i*} − {*j*} − {*a*}) **by** (*rule sum.insert*, *auto*)
also have ... = *f a* **unfolding** *sum0* **by** *simp*
also have ... = *A* $ *a* $ *b* **unfolding** *f-def* **using** *False a-not-i* **by** *auto*
finally show *?thesis* **.**
qed
qed
qed

lemma *det-bezout-matrix*:
fixes *A*::′*a*::{*bezout-domain*}^′*cols*^′*rows*::{*finite*,*wellorder*}
assumes *ib*: *is-bezout-ext bezout*
and *a-less-b*: *a* < *b*
and *aj*: *A* $ *a* $ *j* ≠ *0*
shows *det* (*bezout-matrix A a b j bezout*) = *1*
proof −
let *?B* = *bezout-matrix A a b j bezout*
let *?a* = (*A* $ *a* $ *j*)
let *?b* = (*A* $ *b* $ *j*)
let *?z* = *bezout ?a ?b*
from *ib* **have** *foo*: (⋀*a b*. *let* (*p*, *q*, *u*, *v*, *gcd-a-b*) = *bezout a b*
in p ∗ *a* + *q* ∗ *b* = *gcd-a-b* ∧
gcd-a-b dvd a ∧
gcd-a-b dvd b ∧ (∀ *d*′. *d*′ *dvd a* ∧ *d*′ *dvd b* ⟶ *d*′ *dvd gcd-a-b*) ∧ *gcd-a-b* ∗ *u* = − *b* ∧ *gcd-a-b* ∗ *v* = *a*)
using *is-bezout-ext-def* [*of bezout*] **by** *simp*
obtain *p q u v d* **where** *bz*: (*p*, *q*, *u*, *v*, *d*) = *?z* **by** (*cases ?z*, *auto*)
hence *pib*: *p* ∗ *?a* + *q* ∗ *?b* = *d* ∧ *d dvd ?a* ∧
d dvd ?b ∧ (∀ *d*′. *d*′ *dvd ?a* ∧ *d*′ *dvd ?b* ⟶ *d*′ *dvd d*) ∧ *d* ∗ *u* = − *?b* ∧ *d* ∗ *v* = *?a*
using *foo* [*of ?a ?b*] **by** *fastforce*
hence *pa-bq-d*: *p* ∗ *?a* + *?b* ∗ *q* = *d* **by** (*simp add*: *mult.commute*)
have *a-not-b*: *a* ≠ *b* **using** *a-less-b* **by** *auto*
have *d-dvd-a*: *d dvd ?a* **using** *pib* **by** *auto*
have *UNIV-rw*: *UNIV* = *insert b* (*insert a* (*UNIV* − {*a*} − {*b*})) **by** *auto*
show *?thesis*
proof (*cases p* = *0*)
case *True* **note** *p0*=*True*
have *q-not-0*: *q* ≠ *0*
proof (*rule ccontr*, *simp*)
assume *q*: *q* = *0*
have *d* = *0* **using** *pib*
by (*metis True q add.right-neutral mult.commute mult-zero-right*)
hence *A* $ *a* $ *j* = *0* ∧ *A* $ *b* $ *j* = *0*
by (*metis aj d-dvd-a dvd-0-left-iff*)
thus *False* **using** *aj* **by** *auto*
qed

have *d-not-0*: $d \neq 0$
  by (*metis aj d-dvd-a dvd-0-left-iff*)
have *qb-not-0*: $q * (-?b) \neq 0$
  by (*metis d-not-0 mult-cancel-left1 neg-equal-0-iff-equal*
    no-zero-divisors p0 pa-bq-d q-not-0 right-minus)
have *det* (*interchange-rows ?B a b*) = ($\prod$ *i*∈*UNIV*. (*interchange-rows ?B a b*) \$ *i* \$ *i*)
proof (*rule det-upperdiagonal*)
  fix *i ja*::′*rows* **assume** *ja-i*: *ja*<*i*
  show *interchange-rows* (*bezout-matrix A a b j bezout*) *a b* \$ *i* \$ *ja* = *0*
    unfolding *interchange-rows-def* **using** *a-less-b ja-i p0 a-not-b*
    using *bz* [*symmetric*]
    unfolding *bezout-matrix-def Let-def* **by** *auto*
qed
also have ... = −*1*
proof −
  define *f* **where** *f i* = *interchange-rows* (*bezout-matrix A a b j bezout*) *a b* \$ *i* \$ *i* **for** *i*
  have *prod-rw*: *prod f* (*insert a* (*UNIV* − {*a*} − {*b*}))
    = *f a* ∗ *prod f* (*UNIV* − {*a*} − {*b*})
    by (*rule prod.insert*, *simp-all*)
  have *prod1*: *prod f* (*UNIV* − {*a*} − {*b*}) = *1*
    by (*rule prod.neutral*)
      (*simp add*: *f-def interchange-rows-def bezout-matrix-def Let-def*)
  have *prod f UNIV* = *prod f* (*insert b* (*insert a* (*UNIV* − {*a*} − {*b*})))
    using *UNIV-rw* **by** *simp*
  also have ... = *f b* ∗ *prod f* (*insert a* (*UNIV* − {*a*} − {*b*}))
  proof (*rule prod.insert*, *simp*)
    show *b* ∉ *insert a* (*UNIV* − {*a*} − {*b*}) **using** *a-not-b* **by** *auto*
  qed
  also have ... = *f b* ∗ *f a* **unfolding** *prod-rw prod1* **by** *auto*
  also have ... = *q* ∗ *u*
    using *a-not-b*
    using *bz* [*symmetric*]
    unfolding *f-def interchange-rows-def bezout-matrix-def Let-def* **by** *auto*
  also have ... = −*1*
  proof −
    let *?r* = *q* ∗ *u*
    have *du-b*: *d* ∗ *u* = −*?b* **using** *pib* **by** *auto*
    hence *q* ∗ (−*?b*) = *d* ∗ *?r* **by** (*metis mult.left-commute*)
    also have ... = (*p* ∗ *?a* + *?b* ∗ *q*) ∗ *?r* **unfolding** *pa-bq-d* **by** *auto*
    also have ... = *?b* ∗ *q* ∗ *?r* **using** *True* **by** *auto*
    also have ... = *q* ∗ (−*?b*) ∗ (−*?r*) **by** *auto*
    finally show *?thesis* **using** *qb-not-0*
      unfolding *mult-cancel-left1* **by** (*metis minus-minus*)
  qed
  finally show *?thesis* **unfolding** *f-def* **.**
qed
finally have *det-inter-1*: *det* (*interchange-rows ?B a b*) = − *1* **.**

have *det* (*bezout-matrix A a b j bezout*) = − *1* ∗ *det* (*interchange-rows ?B a b*)
unfolding *det-interchange-rows* **using** *a-not-b* **by** *auto*
thus *?thesis* **unfolding** *det-inter-1* **by** *simp*
next
case *False*
define *mult-b-dp* **where** *mult-b-dp* = *mult-row ?B b* (*d* ∗ *p*)
define *sum-ab* **where** *sum-ab* = *row-add mult-b-dp b a ?b*
have *det* (*sum-ab*) = *prod* (λ*i*. *sum-ab* \$ *i* \$ *i*) *UNIV*
proof (*rule det-upperdiagonal*)
fix *i j*::′*rows*
assume *j-less-i*: *j* < *i*
have *d* ∗ *p* ∗ *u* + *?b* ∗ *p* = *0*
using *pib*
by (*metis eq-neg-iff-add-eq-0 mult-minus-left semiring-normalization-rules*(*16*))
thus *sum-ab* \$ *i* \$ *j* = *0*
unfolding *sum-ab-def mult-b-dp-def* **unfolding** *row-add-def*
unfolding *mult-row-def bezout-matrix-def*
using *a-not-b j-less-i a-less-b*
unfolding *bz* [*symmetric*] **by** *auto*
qed
also have ... = *d* ∗ *p*
proof −
define *f* **where** *f i* = *sum-ab* \$ *i* \$ *i* **for** *i*
have *prod-rw*: *prod f* (*insert a* (*UNIV* − {*a*} − {*b*}))
= *f a* ∗ *prod f* (*UNIV* − {*a*} − {*b*})
by (*rule prod.insert*, *simp-all*)
have *prod1*: *prod f* (*UNIV* − {*a*} − {*b*}) = *1*
by (*rule prod.neutral*) (*simp add*: *f-def sum-ab-def row-add-def*
mult-b-dp-def mult-row-def bezout-matrix-def Let-def)
have *ap-bq-d*: *A* \$ *a* \$ *j* ∗ *p* + *A* \$ *b* \$ *j* ∗ *q* = *d* **by** (*metis mult.commute*
pa-bq-d)
have *prod f UNIV* = *prod f* (*insert b* (*insert a* (*UNIV* − {*a*} − {*b*})))
using *UNIV-rw* **by** *simp*
also have ... = *f b* ∗ *prod f* (*insert a* (*UNIV* − {*a*} − {*b*}))
proof (*rule prod.insert*, *simp*)
show *b* ∉ *insert a* (*UNIV* − {*a*} − {*b*}) **using** *a-not-b* **by** *auto*
qed
also have ... = *f b* ∗ *f a* **unfolding** *prod-rw prod1* **by** *auto*
also have ... = (*d* ∗ *p* ∗ *v* + *?b* ∗ *q*) ∗ *p*
unfolding *f-def sum-ab-def row-add-def mult-b-dp-def mult-row-def be-*
zout-matrix-def
unfolding *bz* [*symmetric*]
using *a-not-b* **by** *auto*
also have ... = *d* ∗ *p*
using *pib ap-bq-d semiring-normalization-rules*(*16*) **by** *auto*
finally show *?thesis* **unfolding** *f-def* **.**
qed
finally have *det* (*sum-ab*) = *d* ∗ *p* **.**
moreover have *det* (*sum-ab*) = *d* ∗ *p* ∗ *det ?B*

```
      unfolding sum-ab-def
      unfolding det-row-add'[OF not-sym[OF a-not-b]]
      unfolding mult-b-dp-def unfolding det-mult-row ..
    ultimately show ?thesis
     by (metis (erased, opaque-lifting) False aj d-dvd-a dvd-0-left-iff mult-cancel-left1
mult-eq-0-iff)
  qed
qed

lemma invertible-bezout-matrix:
  fixes A::'a::{bezout-ring-div}^'cols^'rows::{finite,wellorder}
  assumes ib: is-bezout-ext bezout
  and a-less-b: a < b
  and aj: A $ a $ j ≠ 0
  shows invertible (bezout-matrix A a b j bezout)
  unfolding invertible-iff-is-unit
  unfolding det-bezout-matrix[OF assms]
  by simp

lemma echelon-form-upt-k-bezout-matrix:
  fixes A k and i::'b::mod-type
  assumes e: echelon-form-upt-k A k
  and ib: is-bezout-ext bezout
  and Aik-0: A $ i $ from-nat k ≠ 0
  and zero-i: is-zero-row-upt-k i k A
  and i-less-n: i<n
  and k: k<ncols A
  shows echelon-form-upt-k (bezout-matrix A i n (from-nat k) bezout ** A) k
proof −
  let ?B=(bezout-matrix A i n (from-nat k) bezout ** A)
  have i-not-n: i ≠ n using i-less-n by simp
  have zero-n: is-zero-row-upt-k n k A
    by (metis assms(5) e echelon-form-upt-k-condition1 zero-i)
  have zero-i2: is-zero-row-upt-k i (to-nat (from-nat k::'c)) A
    using zero-i
    by (metis k ncols-def to-nat-from-nat-id)
  have zero-n2: is-zero-row-upt-k n (to-nat (from-nat k::'c)) A
    using zero-n by (metis k ncols-def to-nat-from-nat-id)
  show ?thesis
    unfolding echelon-form-upt-k-def
  proof (auto)
    fix ia j
    assume ia: is-zero-row-upt-k ia k ?B
      and ia-j: ia < j
    have ia-A: is-zero-row-upt-k ia k A
    proof (unfold is-zero-row-upt-k-def, clarify)
      fix j::'c assume j-less-k: to-nat j < k
      have A $ ia $ j = ?B $ ia $ j
      proof (rule bezout-matrix-preserves-previous-columns
```

[*symmetric*, *OF ib i-not-n Aik-0 - zero-i2 zero-n2*])
show *j* < *from-nat k* **using** *j-less-k k*
by (*metis from-nat-mono from-nat-to-nat-id ncols-def*)
qed
also have ... = *0* **by** (*metis ia is-zero-row-upt-k-def j-less-k*)
finally show *A* $ *ia* $ *j* = *0* **.**
qed
show *is-zero-row-upt-k j k ?B*
proof (*unfold is-zero-row-upt-k-def*, *clarify*)
fix *ja*::*'c* **assume** *ja-less-k*: *to-nat ja* < *k*
have *?B* $ *j* $ *ja* = *A* $ *j* $ *ja*
proof (*rule bezout-matrix-preserves-previous-columns*[*OF ib i-not-n Aik-0 - zero-i2 zero-n2*])
show *ja* < *from-nat k* **using** *ja-less-k k*
by (*metis from-nat-mono from-nat-to-nat-id ncols-def*)
qed
also have ... = *0*
by (*metis e echelon-form-upt-k-condition1 ia-A ia-j is-zero-row-upt-k-def ja-less-k*)
finally show *?B* $ *j* $ *ja* = *0* **.**
qed
next
fix *ia j*
assume *ia-j*: *ia* < *j*
and *not-zero-ia-B*: ¬ *is-zero-row-upt-k ia k ?B*
and *not-zero-j-B*: ¬ *is-zero-row-upt-k j k ?B*
obtain *na* **where** *na*: *to-nat na* < *k* **and** *Biana*: *?B* $ *ia* $ *na* ≠ *0*
using *not-zero-ia-B* **unfolding** *is-zero-row-upt-k-def* **by** *auto*
obtain *na2* **where** *na2*: *to-nat na2* < *k* **and** *Bjna2*: *?B* $ *j* $ *na2* ≠ *0*
using *not-zero-j-B* **unfolding** *is-zero-row-upt-k-def* **by** *auto*
have *na-less-fn*: *na* < *from-nat k* **by** (*metis from-nat-mono from-nat-to-nat-id k na ncols-def*)
have *A* $ *ia* $ *na* = *?B* $ *ia* $ *na*
by (*rule bezout-matrix-preserves-previous-columns*
[*symmetric*, *OF ib i-not-n Aik-0 na-less-fn zero-i2 zero-n2*])
also have ... ≠ *0* **using** *Biana* **by** *simp*
finally have *A*: *A* $ *ia* $ *na* ≠ *0* **.**
have *na-less-fn2*: *na2* < *from-nat k* **by** (*metis from-nat-mono from-nat-to-nat-id k na2 ncols-def*)
have *A* $ *j* $ *na2* = *?B* $ *j* $ *na2*
by (*rule bezout-matrix-preserves-previous-columns*
[*symmetric*, *OF ib i-not-n Aik-0 na-less-fn2 zero-i2 zero-n2*])
also have ... ≠ *0* **using** *Bjna2* **by** *simp*
finally have *A2*: *A* $ *j* $ *na2* ≠ *0* **.**
have *not-zero-ia-A*: ¬ *is-zero-row-upt-k ia k A*
unfolding *is-zero-row-upt-k-def* **using** *na A* **by** *auto*
have *not-zero-j-A*: ¬ *is-zero-row-upt-k j k A*
unfolding *is-zero-row-upt-k-def* **using** *na2 A2* **by** *auto*
obtain *na* **where** *A*: *A* $ *ia* $ *na* ≠ *0* **and** *na-less-k*: *to-nat na*<*k*

```
    using not-zero-ia-A unfolding is-zero-row-upt-k-def by auto
  have na-less-fn: na<from-nat k using na-less-k
   by (metis from-nat-mono from-nat-to-nat-id k ncols-def)
  obtain na2 where A2: A $ j $ na2 ≠ 0 and na2-less-k: to-nat na2<k
    using not-zero-j-A unfolding is-zero-row-upt-k-def by auto
  have na-less-fn2: na2<from-nat k using na2-less-k
   by (metis from-nat-mono from-nat-to-nat-id k ncols-def)
  have least-eq: (LEAST na. ?B $ ia $ na ≠ 0) = (LEAST na. A $ ia $ na ≠ 0)
  proof (rule Least-equality)
    have ?B $ ia $ (LEAST na. A $ ia $ na ≠ 0) = A $ ia $ (LEAST na. A $
ia $ na ≠ 0)
     proof (rule bezout-matrix-preserves-previous-columns[OF ib i-not-n Aik-0 -
zero-i2 zero-n2])
      show (LEAST na. A $ ia $ na ≠ 0) < from-nat k using Least-le A na-less-fn
by fastforce
    qed
    also have ... ≠ 0 by (metis (mono-tags) A LeastI)
    finally show ?B $ ia $ (LEAST na. A $ ia $ na ≠ 0) ≠ 0 .
    fix y
    assume B-ia-y: ?B $ ia $ y ≠ 0
    show (LEAST na. A $ ia $ na ≠ 0) ≤ y
    proof (cases y<from-nat k)
      case True
      show ?thesis
      proof (rule Least-le)
        have A $ ia $ y = ?B $ ia $ y
          by (rule bezout-matrix-preserves-previous-columns[symmetric,
            OF ib i-not-n Aik-0 True zero-i2 zero-n2])
        also have ... ≠ 0 using B-ia-y .
        finally show A $ ia $ y ≠ 0 .
      qed
    next
      case False
      show ?thesis using False
        by (metis (mono-tags) A Least-le dual-order.strict-iff-order
          le-less-trans na-less-fn not-le)
    qed
  qed
  have least-eq2: (LEAST na. ?B $ j $ na ≠ 0) = (LEAST na. A $ j $ na ≠ 0)
  proof (rule Least-equality)
    have ?B $ j $ (LEAST na. A $ j $ na ≠ 0) = A $ j $ (LEAST na. A $ j $
na ≠ 0)
     proof (rule bezout-matrix-preserves-previous-columns[OF ib i-not-n Aik-0 -
zero-i2 zero-n2])
      show (LEAST na. A $ j $ na ≠ 0) < from-nat k using Least-le A2 na-less-fn2
by fastforce
    qed
    also have ... ≠ 0 by (metis (mono-tags) A2 LeastI)
    finally show ?B $ j $ (LEAST na. A $ j $ na ≠ 0) ≠ 0 .
```

```
      fix y
      assume B-ia-y: ?B $ j $ y ≠ 0
      show (LEAST na. A $ j $ na ≠ 0) ≤ y
      proof (cases y<from-nat k)
        case True
        show ?thesis
        proof (rule Least-le)
          have A $ j $ y = ?B $ j $ y
            by (rule bezout-matrix-preserves-previous-columns[symmetric,
              OF ib i-not-n Aik-0 True zero-i2 zero-n2])
          also have ... ≠ 0 using B-ia-y .
          finally show A $ j $ y ≠ 0 .
        qed
      next
        case False
        show ?thesis using False
          by (metis (mono-tags) A2 Least-le dual-order.strict-iff-order
            le-less-trans na-less-fn2 not-le)
      qed
    qed
  show (LEAST na. ?B $ ia $ na ≠ 0) < (LEAST na. ?B $ j $ na ≠ 0) unfolding
least-eq least-eq2
      by (rule echelon-form-upt-k-condition2[OF e ia-j not-zero-ia-A not-zero-j-A])
 qed
qed

lemma bezout-matrix-preserves-rest:
  assumes ib: is-bezout-ext bezout
  and a-not-n: a≠n
  and i-not-n: i≠n
  and a-not-i: a≠i
  and Aik-0: A $ i $ k ≠ 0
  and zero-ikA: is-zero-row-upt-k i (to-nat k) A
  shows (bezout-matrix A i n k bezout ** A) $ a $ b = A $ a $ b
  unfolding matrix-matrix-mult-def unfolding bezout-matrix-def Let-def
proof (auto simp add: a-not-n i-not-n a-not-i)
  have UNIV-rw: UNIV = insert a (UNIV − {a}) by auto
  let ?f=(λk. (if a = k then 1 else 0) ∗ A $ k $ b)
  have sum0: sum ?f (UNIV − {a}) = 0 by (rule sum.neutral, auto)
  have sum ?f UNIV = sum ?f (insert a (UNIV − {a})) using UNIV-rw by simp
  also have ... = ?f a + sum ?f (UNIV − {a}) by (rule sum.insert, simp-all)
  also have ... = ?f a using sum0 by auto
  also have ... = A $ a $ b by simp
  finally show sum ?f UNIV = A $ a $ b .
qed
```

Code equations to execute the bezout matrix

```
definition bezout-matrix-row A a b j bezout x
  = (let (p, q, u, v, d) = bezout (A $ a $ j) (A $ b $ j)
```

in
vec-lambda (λy. *if* $x = a \wedge y = a$ *then* p *else*
if $x = a \wedge y = b$ *then* q *else*
if $x = b \wedge y = a$ *then* u *else*
if $x = b \wedge y = b$ *then* v *else*
if $x = y$ *then* *1* *else* *0*))

lemma *bezout-matrix-row-code* [*code abstract*]:
vec-nth (*bezout-matrix-row* A a b j *bezout* x) =
(*let* (p, q, u, v, d) = *bezout* (A \$ a \$ j) (A \$ b \$ j)
in
(λy. *if* $x = a \wedge y = a$ *then* p *else*
if $x = a \wedge y = b$ *then* q *else*
if $x = b \wedge y = a$ *then* u *else*
if $x = b \wedge y = b$ *then* v *else*
if $x = y$ *then* *1* *else* *0*)) **unfolding** *bezout-matrix-row-def*
by (*cases bezout* (A \$ a \$ j) (A \$ b \$ j)) *auto*

lemma [*code abstract*]: *vec-nth* (*bezout-matrix* A a b j *bezout*) = *bezout-matrix-row* A a b j *bezout*
unfolding *bezout-matrix-def* **unfolding** *bezout-matrix-row-def*[*abs-def*] *Let-def*
by (*cases bezout* (A \$ a \$ j) (A \$ b \$ j)) *auto*

3.2.5 Properties of the bezout iterate function

lemma *bezout-iterate-not-zero*:
assumes *Aik-0*: A \$ i \$ *from-nat* $k \neq 0$
and n: n<*nrows* A
and a: *to-nat* $i \leq n$
and *ib*: *is-bezout-ext bezout*
shows *bezout-iterate* A n i (*from-nat* k) *bezout* \$ i \$ *from-nat* $k \neq 0$
using *Aik-0* n a
proof (*induct* n *arbitrary*: A)
case *0*
have *to-nat* $i = 0$ **by** (*metis 0.prems*(*3*) *le-0-eq*)
hence *i0*: *i=0* **by** (*metis to-nat-eq-0*)
show *?case* **using** *0.prems*(*1*) **unfolding** *i0* **by** *auto*
next
case (*Suc* n)
show *?case*
proof (*cases Suc* n = *to-nat* i)
case *True* **show** *?thesis* **unfolding** *bezout-iterate.simps* **using** *True Suc.prems*(*1*) **by** *simp*
next
case *False*
let *?B*=(*bezout-matrix* A i (*from-nat* (*Suc* n)) (*from-nat* k) *bezout* ** A)
have *i-le-n*: *to-nat* i < *Suc* n **using** *Suc.prems*(*3*) *False* **by** *auto*
have *bezout-iterate* A (*Suc* n) i (*from-nat* k) *bezout* \$ i \$ *from-nat* k
= *bezout-iterate* *?B* n i (*from-nat* k) *bezout* \$ i \$ *from-nat* k

```
      unfolding bezout-iterate.simps using i-le-n by auto
    also have ... ≠ 0
    proof (rule Suc.hyps, rule bezout-matrix-not-zero[OF ib])
     show i ≠ from-nat (Suc n) by (metis False Suc.prems(2) nrows-def to-nat-from-nat-id)
      show A $ i $ from-nat k ≠ 0 by (rule Suc.prems(1))
      show n < nrows ?B by (metis Suc.prems(2) Suc-lessD nrows-def)
      show to-nat i ≤ n using i-le-n by auto
    qed
    finally show ?thesis .
  qed
qed


lemma bezout-iterate-preserves:
  fixes A k and i::'b::mod-type
  assumes e: echelon-form-upt-k A k
  and ib: is-bezout-ext bezout
  and Aik-0: A $ i $ from-nat k ≠ 0
  and n: n<nrows A
  and b < from-nat k
  and i-le-n: to-nat i ≤ n
  and k: k<ncols A
  and zero-upt-k-i: is-zero-row-upt-k i k A
  shows bezout-iterate A n i (from-nat k) bezout $ a $ b = A $ a $ b
  using Aik-0 n i-le-n k zero-upt-k-i  e
proof (induct n arbitrary: A)
  case 0
  show ?case unfolding bezout-iterate.simps ..
next
  case (Suc n)
  show ?case
  proof (cases Suc n = to-nat i)
    case True show ?thesis unfolding bezout-iterate.simps using True  by simp
  next
    case False
    have i-not-fn: i ≠ from-nat (Suc n)
      by (metis False Suc.prems(2) nrows-def to-nat-from-nat-id)
    let ?B=(bezout-matrix A i (from-nat (Suc n)) (from-nat k) bezout ** A)
    have i-le-n: to-nat i < Suc n by (metis False Suc.prems(3) le-imp-less-or-eq)
    have bezout-iterate A (Suc n) i (from-nat k) bezout $ a $ b
      = bezout-iterate ?B n i (from-nat k) bezout $ a $ b
      unfolding bezout-iterate.simps using i-le-n by auto
    also have ... = ?B $ a $ b
    proof (rule Suc.hyps)
      show ?B $ i $ from-nat k ≠ 0
        by (metis False Suc.prems(1) Suc.prems(2) bezout-matrix-not-zero
            ib nrows-def to-nat-from-nat-id)
      show n < nrows ?B by (metis Suc.prems(2) Suc-lessD nrows-def)
```

show *k* < *ncols ?B* **by** (*metis Suc.prems*(*4*) *ncols-def*)
show *to-nat i* ≤ *n* **using** *i-le-n* **by** *auto*
show *is-zero-row-upt-k i k ?B*
proof (*unfold is-zero-row-upt-k-def*, *clarify*)
fix *j*::′*c* **assume** *j-k*: *to-nat j* < *k*
have *j-k2*: *j* < *from-nat k* **by** (*metis from-nat-mono from-nat-to-nat-id j-k k ncols-def*)
have *?B* $ *i* $ *j* = *A* $ *i* $ *j*
proof (*rule bezout-matrix-preserves-previous-columns*[*OF ib i-not-fn Suc.prems*(*1*) *j-k2*],
unfold to-nat-from-nat-id[*OF Suc.prems*(*4*)[*unfolded ncols-def*]])
show *is-zero-row-upt-k i k A* **by** (*rule Suc.prems*(*5*))
show *is-zero-row-upt-k* (*from-nat* (*Suc n*)) *k A*
using *echelon-form-upt-k-condition1* [*OF Suc.prems*(*6*) *Suc.prems*(*5*)]
by (*metis Suc.prems*(*2*) *from-nat-mono from-nat-to-nat-id i-le-n nrows-def*)
qed
also have ... = *0* **by** (*metis Suc.prems*(*5*) *is-zero-row-upt-k-def j-k*)
finally show *?B* $ *i* $ *j* = *0* **.**
qed
show *echelon-form-upt-k ?B k*
proof (*rule echelon-form-upt-k-bezout-matrix*)
show *echelon-form-upt-k A k* **by** (*metis Suc.prems*(*6*))
show *is-bezout-ext bezout* **by** (*rule ib*)
show *A* $ *i* $ *from-nat k* ≠ *0* **by** (*rule Suc.prems*(*1*))
show *is-zero-row-upt-k i k A* **by** (*rule Suc.prems*(*5*))
have (*from-nat* (*to-nat i*)::′*b*)≤*from-nat* (*Suc n*)
by (*rule from-nat-mono*′[*OF Suc.prems*(*3*) *Suc.prems*(*2*)[*unfolded nrows-def*]])
thus *i* < *from-nat* (*Suc n*) **using** *i-not-fn* **by** *auto*
show *k* < *ncols A* **by** (*rule Suc.prems*(*4*))
qed
qed
also have ... = *A* $ *a* $ *b*
proof (*rule bezout-matrix-preserves-previous-columns*[*OF ib*])
show *i* ≠ *from-nat* (*Suc n*) **by** (*metis False Suc.prems*(*2*) *nrows-def to-nat-from-nat-id*)
show *A* $ *i* $ *from-nat k* ≠ *0* **by** (*rule Suc.prems*(*1*))
show *b* < *from-nat k* **by** (*rule assms*(*5*))
show *is-zero-row-upt-k i* (*to-nat* (*from-nat k*::′*c*)) *A*
unfolding *to-nat-from-nat-id*[*OF Suc.prems*(*4*)[*unfolded ncols-def*]] **by** (*rule Suc.prems*(*5*))
show *is-zero-row-upt-k* (*from-nat* (*Suc n*)) (*to-nat* (*from-nat k*::′*c*)) *A*
unfolding *to-nat-from-nat-id*[*OF Suc.prems*(*4*)[*unfolded ncols-def*]]
by (*metis Suc.prems*(*2*) *Suc.prems*(*5*) *Suc.prems*(*6*) *add-to-nat-def echelon-form-upt-k-condition1 from-nat-mono i-le-n monoid-add-class.add.right-neutral*
nrows-def to-nat-0)
qed
finally show *?thesis* **.**
qed
qed

```
lemma bezout-iterate-preserves-below-n:
  assumes e: echelon-form-upt-k A k
  and ib: is-bezout-ext bezout
  and Aik-0: A $ i $ from-nat k ≠ 0
  and n: n<nrows A
  and n-less-a: n < to-nat a
  and k: k<ncols A
  and i-le-n: to-nat i ≤ n
  and zero-upt-k-i: is-zero-row-upt-k i k A
  shows bezout-iterate A n i (from-nat k) bezout $ a $ b = A $ a $ b
  using Aik-0 n i-le-n k zero-upt-k-i e n-less-a
proof (induct n arbitrary: A)
  case 0
  show ?case unfolding bezout-iterate.simps ..
next
  case (Suc n)
  show ?case
  proof (cases Suc n = to-nat i)
    case True show ?thesis unfolding bezout-iterate.simps using True  by simp
  next
    case False
    have i-not-fn: i ≠ from-nat (Suc n)
      by (metis False Suc.prems(2) nrows-def to-nat-from-nat-id)
    let ?B=(bezout-matrix A i (from-nat (Suc n)) (from-nat k) bezout ** A)
    have i-le-n: to-nat i < Suc n by (metis False Suc.prems(3) le-imp-less-or-eq)
    have zero-ikB: is-zero-row-upt-k i k ?B
    proof (unfold is-zero-row-upt-k-def, clarify)
      fix j::'b assume j-k: to-nat j < k
      have j-k2: j < from-nat k by (metis from-nat-mono from-nat-to-nat-id j-k k
ncols-def)
      have ?B $ i $ j = A $ i $ j
     proof (rule bezout-matrix-preserves-previous-columns[OF ib i-not-fn Suc.prems(1)
j-k2],
          unfold to-nat-from-nat-id[OF Suc.prems(4)[unfolded ncols-def]])
        show is-zero-row-upt-k i k A by (rule Suc.prems(5))
        show is-zero-row-upt-k (from-nat (Suc n)) k A
          using echelon-form-upt-k-condition1[OF Suc.prems(6) Suc.prems(5)]
         by (metis Suc.prems(2) from-nat-mono from-nat-to-nat-id i-le-n nrows-def)
      qed
      also have ... = 0 by (metis Suc.prems(5) is-zero-row-upt-k-def j-k)
      finally show ?B $ i $ j = 0 .
    qed
    have bezout-iterate A (Suc n) i (from-nat k) bezout $ a $ b
      = bezout-iterate ?B n i (from-nat k) bezout $ a $ b
      unfolding bezout-iterate.simps using i-le-n by auto
```

```
    also have ... = ?B $ a $ b
    proof (rule Suc.hyps)
     show ?B $ i $ from-nat k ≠ 0 by (metis Suc.prems(1) bezout-matrix-not-zero
i-not-fn ib)
      show n < nrows ?B by (metis Suc.prems(2) Suc-lessD nrows-def)
      show to-nat i ≤ n by (metis i-le-n less-Suc-eq-le)
      show k < ncols ?B by (metis Suc.prems(4) ncols-def)
      show is-zero-row-upt-k i k ?B by (rule zero-ikB)
      show echelon-form-upt-k ?B k
      proof (rule echelon-form-upt-k-bezout-matrix[OF Suc.prems(6) ib
          Suc.prems(1) Suc.prems(5) - Suc.prems(4)])
        show i < from-nat (Suc n)
         by (metis (no-types) Suc.prems(7) add-to-nat-def dual-order.strict-iff-order
from-nat-mono
         i-le-n le-less-trans monoid-add-class.add.right-neutral to-nat-0 to-nat-less-card)
      qed
      show n < to-nat a by (metis Suc.prems(7) Suc-lessD)
    qed
    also have ... = A $ a $ b
    proof (rule bezout-matrix-preserves-rest[OF ib - - - Suc.prems(1)])
      show a ≠ from-nat (Suc n)
        by (metis Suc.prems(7) add-to-nat-def from-nat-mono less-irrefl
          monoid-add-class.add.right-neutral to-nat-0 to-nat-less-card)
      show i ≠ from-nat (Suc n) by (rule i-not-fn)
      show a ≠ i by (metis assms(7) n-less-a not-le)
      show is-zero-row-upt-k i (to-nat (from-nat k::'b)) A
        by (metis Suc.prems(4) Suc.prems(5) ncols-def to-nat-from-nat-id)
    qed
    finally show ?thesis .
  qed
qed

lemma bezout-iterate-zero-column-k:
  fixes A::'a::bezout-domain^'cols::{mod-type}^'rows::{mod-type}
  assumes e: echelon-form-upt-k A k
  and ib: is-bezout-ext bezout
  and Aik-0: A $ i $ from-nat k ≠ 0
  and n: n<nrows A
  and i-le-a: i<a
  and k: k<ncols A
  and a-n: to-nat a≤n
  and zero-upt-k-i: is-zero-row-upt-k i k A
  shows bezout-iterate A n i (from-nat k) bezout $ a $ from-nat k = 0
  using e Aik-0 n k a-n zero-upt-k-i
proof (induct n arbitrary: A)
  case 0
  show ?case unfolding bezout-iterate.simps
    using 0.prems(5) i-le-a to-nat-from-nat to-nat-le to-nat-mono by fastforce
next
```

```
  case (Suc n)
  show ?case
  proof (cases Suc n = to-nat i)
    case True show ?thesis unfolding bezout-iterate.simps using True
      by (metis Suc.prems(5) i-le-a leD to-nat-mono)
  next
    case False
    have i-not-fn: i ≠ from-nat (Suc n)
      by (metis False Suc.prems(3) nrows-def to-nat-from-nat-id)
    let ?B=(bezout-matrix A i (from-nat (Suc n)) (from-nat k) bezout ** A)
    have i-le-n: to-nat i < Suc n by (metis Suc.prems(5) i-le-a le-less-trans not-le
to-nat-mono)
    have zero-ikB: is-zero-row-upt-k i k ?B
    proof (unfold is-zero-row-upt-k-def, clarify)
      fix j::'cols assume j-k: to-nat j < k
      have j-k2: j < from-nat k
        using from-nat-mono[OF j-k Suc.prems(4)[unfolded ncols-def]]
        unfolding from-nat-to-nat-id .
      have ?B $ i $ j = A $ i $ j
     proof (rule bezout-matrix-preserves-previous-columns[OF ib i-not-fn Suc.prems(2)
j-k2],
          unfold to-nat-from-nat-id[OF Suc.prems(4)[unfolded ncols-def]])
        show is-zero-row-upt-k i k A by (rule Suc.prems(6))
        show is-zero-row-upt-k (from-nat (Suc n)) k A
          using echelon-form-upt-k-condition1[OF Suc.prems(1)]
          by (metis (mono-tags) Suc.prems(3) Suc.prems(6) add-to-nat-def
                from-nat-mono i-le-n monoid-add-class.add.right-neutral nrows-def
to-nat-0)
      qed
      also have ... = 0 by (metis Suc.prems(6) is-zero-row-upt-k-def j-k)
      finally show ?B $ i $ j = 0 .
    qed
    have bezout-iterate A (Suc n) i (from-nat k) bezout $ a $ (from-nat k)
      = bezout-iterate ?B n i (from-nat k) bezout $ a $ (from-nat k)
      unfolding bezout-iterate.simps using i-le-n by auto
    also have ... = 0
    proof (cases to-nat a = Suc n)
      case True
      have bezout-iterate ?B n i (from-nat k) bezout $ a $ (from-nat k) = ?B $ a
$ from-nat k
      proof (rule bezout-iterate-preserves-below-n[OF - ib])
        show echelon-form-upt-k ?B k
        by (metis (erased, opaque-lifting) Suc.prems(1) Suc.prems(2) Suc.prems(4)
Suc.prems(6) True
            echelon-form-upt-k-bezout-matrix from-nat-to-nat-id i-le-a ib)
        show ?B $ i $ from-nat k ≠ 0
          by (metis Suc.prems(2) bezout-matrix-not-zero i-not-fn ib)
        show n < nrows ?B by (metis Suc.prems(3) Suc-lessD nrows-def)
        show n < to-nat a by (metis True lessI)
```

show *k* < *ncols ?B* **by** (*metis Suc.prems*(*4*) *ncols-def*)
show *to-nat i* ≤ *n* **by** (*metis i-le-n less-Suc-eq-le*)
show *is-zero-row-upt-k i k ?B* **by** (*rule zero-ikB*)
qed
also have ... = *0*
by (*metis Suc.prems*(*2*) *True bezout-matrix-works2*
i-not-fn ib to-nat-from-nat)
finally show *?thesis* **.**
next
case *False*
show *?thesis*
proof (*rule Suc.hyps*)
show *echelon-form-upt-k ?B k*
proof (*rule echelon-form-upt-k-bezout-matrix*
[*OF Suc.prems*(*1*) *ib Suc.prems*(*2*) *Suc.prems*(*6*) - *Suc.prems*(*4*)])
show *i* < *from-nat* (*Suc n*)
by (*metis* (*mono-tags*) *Suc.prems*(*3*) *from-nat-mono from-nat-to-nat-id*
i-le-n nrows-def)
qed
show *?B* $ *i* $ *from-nat k* ≠ *0* **by** (*metis Suc.prems*(*2*) *bezout-matrix-not-zero*
i-not-fn ib)
show *n* < *nrows ?B* **by** (*metis Suc.prems*(*3*) *Suc-lessD nrows-def*)
show *k* < *ncols ?B* **by** (*metis Suc.prems*(*4*) *ncols-def*)
show *to-nat a* ≤ *n* **by** (*metis False Suc.prems*(*5*) *le-SucE*)
show *is-zero-row-upt-k i k ?B* **by** (*rule zero-ikB*)
qed
qed
finally show *?thesis* **.**
qed
qed

3.2.6 Proving the correctness

lemma *condition1-index-le-zero-row*:
fixes *A k*
defines *i*:*i*≡(*if* ∀ *m. is-zero-row-upt-k m k A then 0*
else to-nat ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*)) + *1*)
assumes *e*: *echelon-form-upt-k A k*
and *is-zero-row-upt-k a* (*Suc k*) *A*
shows *from-nat i*≤*a*
proof (*rule ccontr*)
have *zero-ik*: *is-zero-row-upt-k a k A* **by** (*metis assms*(*3*) *is-zero-row-upt-k-le*)
assume *a*: ¬ *from-nat i* ≤ (*a*::'*a*) **hence** *ai*: *a* < *from-nat i* **by** *simp*
show *False*
proof (*cases* (*from-nat i*::'*a*)=*0*)
case *True* **thus** *?thesis* **using** *ai least-mod-type*[*of a*] **unfolding** *True from-nat-0*
by *auto*
next
case *False*

from *a* **have** *a* ≤ *from-nat i* − *1* **by** (*intro leI*) (*auto dest*: *le-Suc*)
also from *False* **have** *i* ≠ *0* **by** (*intro notI*) (*simp-all add*: *from-nat-0*)
hence *i* = (*i* − *1*) + *1* **by** *simp*
also have *from-nat* ... = *from-nat* (*i* − *1*) + *1* **by** (*rule from-nat-suc*)
finally have *ai2*: *a* ≤ *from-nat* (*i* − *1*) **by** *simp*
have *i* = *to-nat* ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)) + *1* **using** *i* *False*
by (*metis from-nat-0*)
hence *i* − *1* = *to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) **by** *simp*
hence *from-nat* (*i* − *1*) = (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)
using *from-nat-to-nat-id* **by** *auto*
hence ¬ *is-zero-row-upt-k* (*from-nat* (*i* − *1*)) *k A* **using** *False GreatestI-ex i*
by (*metis from-nat-to-nat-id to-nat-0*)
moreover have *is-zero-row-upt-k* (*from-nat* (*i* − *1*)) *k A*
using *echelon-form-upt-k-condition1*[*OF e zero-ik*]
using *ai2 zero-ik* **by** (*cases a* = *from-nat* (*i* − *1*), *auto*)
ultimately show *False* **by** *contradiction*
qed
qed

lemma *condition1-part1*:
fixes *A k*
defines *i*:*i*≡(*if* ∀ *m*. *is-zero-row-upt-k m k A then 0*
else to-nat ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)) + *1*)
assumes *e*: *echelon-form-upt-k A k*
and *a*: *is-zero-row-upt-k a* (*Suc k*) *A*
and *ab*: *a* < *b*
and *all-zero*: ∀ *m*≥*from-nat i*. *A* $ *m* $ *from-nat k* = *0*
shows *is-zero-row-upt-k b* (*Suc k*) *A*
proof (*rule is-zero-row-upt-k-suc*)
have *zero-ik*: *is-zero-row-upt-k a k A* **by** (*metis assms*(*3*) *is-zero-row-upt-k-le*)
show *is-zero-row-upt-k b k A*
using *echelon-form-upt-k-condition1*[*OF e zero-ik*] **using** *ab* **by** *auto*
have *from-nat i*≤*a*
using *condition1-index-le-zero-row*[*OF e a*] *all-zero* **unfolding** *i* **by** *auto*
thus *A* $ *b* $ *from-nat k* = *0* **using** *all-zero ab* **by** *auto*
qed

lemma *condition1-part2*:
fixes *A k*
defines *i*:*i*≡(*if* ∀ *m*. *is-zero-row-upt-k m k A then 0*
else to-nat ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)) + *1*)
assumes *e*: *echelon-form-upt-k A k*
and *a*: *is-zero-row-upt-k a* (*Suc k*) *A*
and *ab*: *a* < *b*

and *i-last*: $i = nrows\ A$
and *all-zero*: $\forall m>$*from-nat* (*nrows A*). $A\ \$\ m\ \$$ *from-nat* $k = 0$
shows *is-zero-row-upt-k b* (*Suc k*) *A*
proof −
have *zero-ik*: *is-zero-row-upt-k a k A* **by** (*metis assms*(*3*) *is-zero-row-upt-k-le*)
have *i-le-a*: *from-nat* $i \le a$ **using** *condition1-index-le-zero-row*[*OF e a*] **unfolding** *i* **.**
have (*from-nat* (*nrows A*)::′*a*) = *0* **unfolding** *nrows-def* **using** *from-nat-CARD* **.**
thus *?thesis* **using** *ab i-last i-le-a*
by (*metis all-zero e echelon-form-upt-k-condition1 is-zero-row-upt-k-suc le-less-trans zero-ik*)
qed

lemma *condition1-part3*:
fixes *A k bezout*
defines *i*:*i*≡(*if* ∀*m*. *is-zero-row-upt-k m k A then 0*
else to-nat ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)) + *1*)
defines *B*: *B* ≡ *fst* ((*echelon-form-of-column-k bezout*) (*A*,*i*) *k*)
assumes *e*: *echelon-form-upt-k A k* **and** *ib*: *is-bezout-ext bezout*
and *a*: *is-zero-row-upt-k a* (*Suc k*) *B*
and $a < b$
and *all-zero*: $\forall m>$*from-nat i*. $A\ \$\ m\ \$$ *from-nat* $k = 0$
and *i-not-last*: $i \neq nrows\ A$
and *i-le-m*: *from-nat* $i \le m$
and *Amk-not-0*: $A\ \$\ m\ \$$ *from-nat* $k \neq 0$
shows *is-zero-row-upt-k b* (*Suc k*) *A*
proof (*rule is-zero-row-upt-k-suc*)
have *AB*: $A = B$ **unfolding** *B echelon-form-of-column-k-def Let-def* **using** *all-zero* **by** *auto*
have *i-le-a*: *from-nat* $i \le a$
using *condition1-index-le-zero-row*[*OF e a*[*unfolded AB*[*symmetric*]]] **unfolding** *i* **.**
show $A\ \$\ b\ \$$ *from-nat* $k = 0$ **by** (*metis i-le-a all-zero assms*(*6*) *le-less-trans*)
show *is-zero-row-upt-k b k A*
by (*metis* (*poly-guards-query*) *AB a assms*(*6*) *e*
echelon-form-upt-k-condition1 is-zero-row-upt-k-le)
qed

lemma *condition1-part4*:
fixes *A k bezout i*
defines *i*:*i*≡(*if* ∀*m*. *is-zero-row-upt-k m k A then 0*
else to-nat ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)) + *1*)
defines *B*: *B*≡ *fst* ((*echelon-form-of-column-k bezout*) (*A*,*i*) *k*)
assumes *e*: *echelon-form-upt-k A k*
assumes *a*: *is-zero-row-upt-k a* (*Suc k*) *A*
and *i-nrows*: $i = nrows\ A$
shows *is-zero-row-upt-k b* (*Suc k*) *A*

proof −
  have *eq-G*: *from-nat* (*i* − *1*) = (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)
    by (*metis One-nat-def Suc-eq-plus1 i-nrows diff-Suc-Suc*
        diff-zero from-nat-to-nat-id i nrows-not-0)
  hence *a-le*: *a*≤*from-nat* (*i* − *1*)
    by (*metis One-nat-def Suc-pred i-nrows lessI not-less not-less-eq nrows-def*
      to-nat-from-nat-id to-nat-less-card to-nat-mono zero-less-card-finite)
  have *not-zero-G*: ¬ *is-zero-row-upt-k* (*from-nat*(*i* − *1*)) *k A*
    unfolding *eq-G*
    by (*metis* (*mono-tags*) *GreatestI-ex i-nrows i nrows-not-0*)
  hence ¬ *is-zero-row-upt-k a k A*
    by (*metis a-le dual-order.strict-iff-order e echelon-form-upt-k-condition1*)
  hence ¬ *is-zero-row-upt-k a* (*Suc k*) *A*
    by (*metis is-zero-row-upt-k-le*)
  thus *?thesis* **using** *a* **by** *contradiction*
qed

lemma *condition1-part5*:
  fixes *A*::′*a*::*bezout-domain*^′*cols*::{*mod-type*}^′*rows*::{*mod-type*}
  and *k bezout*
  defines *i*:*i*≡(*if* ∀ *m*. *is-zero-row-upt-k m k A then 0*
    else to-nat ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)) + *1*)
  defines *B*: *B* ≡ *fst*((*echelon-form-of-column-k bezout*) (*A*,*i*) *k*)
  assumes *ib*: *is-bezout-ext bezout* **and** *e*: *echelon-form-upt-k A k*
  assumes *zero-a-B*: *is-zero-row-upt-k a* (*Suc k*) *B*
  and *ab*: *a* < *b*
  and *im*: *from-nat i* < *m*
  and *Amk-not-0*: *A* $ *m* $ *from-nat k* ≠ *0*
  and *not-last-row*: *i* ≠ *nrows A*
  and *k*: *k*<*ncols A*
  shows *is-zero-row-upt-k b* (*Suc k*) (*bezout-iterate*
  (*interchange-rows A* (*from-nat i*) (*LEAST n*. *A* $ *n* $ *from-nat k* ≠ *0* ∧ (*from-nat*
i) ≤ *n*))
  (*nrows A* − *Suc 0*) (*from-nat i*) (*from-nat k*) *bezout*)
proof (*rule is-zero-row-upt-k-suc*)
  let *?least*=(*LEAST n*. *A* $ *n* $ *from-nat k* ≠ *0* ∧ *from-nat i* ≤ *n*)
  let *?interchange*=(*interchange-rows A* (*from-nat i*) *?least*)
  let *?bezout-iterate*=(*bezout-iterate ?interchange*
    (*nrows A* − *Suc 0*) (*from-nat i*) (*from-nat k*) *bezout*)
  have *B-eq*: *B* = *?bezout-iterate* **unfolding** *B echelon-form-of-column-k-def*
    Let-def fst-conv snd-conv **using** *im Amk-not-0 not-last-row* **by** *auto*
  have *zero-ikA*: *is-zero-row-upt-k* (*from-nat i*) *k A*
  proof (*cases* ∀ *m*. *is-zero-row-upt-k m k A*)
    case *True*
    thus *?thesis* **by** *simp*
  next
    case *False*
     hence *i-eq*: *i*=*to-nat* ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)) + *1* **un-**

folding *i* **by** *auto*
show *?thesis*
proof (*rule row-greater-greatest-is-zero*, *simp add*: *i-eq from-nat-to-nat-greatest*,*rule Suc-le'*)
show (*GREATEST m*. ¬ *is-zero-row-upt-k m k A*) + *1* ≠ *0*
proof −
have $\bigwedge x_1$. ¬ x_1 < *i* ∨ ¬ *to-nat* (*GREATEST R*. ¬ *is-zero-row-upt-k R k A*) < x_1
using *i-eq* **by** *linarith*
thus (*GREATEST m*. ¬ *is-zero-row-upt-k m k A*) + *1* ≠ *0*
by (*metis One-nat-def add-Suc-right neq-iff*
from-nat-to-nat-greatest i-eq monoid-add-class.add.right-neutral
nat.distinct(*1*) *not-last-row nrows-def to-nat-0 to-nat-from-nat-id to-nat-less-card*)
qed
qed
qed
have *zero-interchange*: *is-zero-row-upt-k* (*from-nat i*) *k ?interchange*
proof (*unfold is-zero-row-upt-k-def*, *clarify*)
fix *j*::*'cols* **assume** *j-less-k*: *to-nat j* < *k*
have *i-le-least*: *from-nat i*≤*?least*
by (*metis* (*mono-tags*, *lifting*) *Amk-not-0 LeastI2-wellorder less-imp-le im*)
hence *zero-least-kA*: *is-zero-row-upt-k ?least k A*
using *echelon-form-upt-k-condition1*[*OF e zero-ikA*]
by (*metis* (*poly-guards-query*) *dual-order.strict-iff-order zero-ikA*)
have *?interchange* \$ *from-nat i* \$ *j* = *A* \$ *?least* \$ *j* **by** *simp*
also have ... = *0* **using** *zero-least-kA j-less-k* **unfolding** *is-zero-row-upt-k-def* **by** *simp*
finally show *?interchange* \$ *from-nat i* \$ *j* = *0* **.**
qed
have *zero-a-k*: *is-zero-row-upt-k a k A*
proof (*unfold is-zero-row-upt-k-def*, *clarify*)
fix *j*::*'cols* **assume** *j-less-k*: *to-nat j* < *k*
have *?interchange* \$ *a* \$ *j* = *?bezout-iterate* \$ *a* \$ *j*
proof (*rule bezout-iterate-preserves*[*symmetric*])
show *echelon-form-upt-k ?interchange k*
proof (*rule echelon-form-upt-k-interchange*[*OF e zero-ikA Amk-not-0 - k*])
show *from-nat i* ≤ *m* **using** *im* **by** *auto*
qed
show *is-bezout-ext bezout* **using** *ib* **.**
show *?interchange* \$ (*from-nat i*) \$ *from-nat k* ≠ *0*
by (*metis* (*mono-tags*, *lifting*) *Amk-not-0 LeastI-ex dual-order.strict-iff-order*

im interchange-rows-i)
show *nrows A* − *Suc 0* < *nrows ?interchange* **unfolding** *nrows-def* **by** *simp*
show *j* < *from-nat k* **by** (*metis from-nat-mono from-nat-to-nat-id j-less-k k ncols-def*)
show *to-nat* (*from-nat i*::*'rows*) ≤ *nrows A* − *Suc 0*
by (*simp add*: *nrows-def le-diff-conv2 Suc-le-eq to-nat-less-card*)
show *k* < *ncols ?interchange* **using** *k* **unfolding** *ncols-def* **by** *auto*

show *is-zero-row-upt-k* (*from-nat i*) *k ?interchange* **using** *zero-interchange* **.**
qed
also have ... = *0* **using** *zero-a-B j-less-k* **unfolding** *B-eq is-zero-row-upt-k-def*
by *auto*
finally have ∗: *?interchange* \$ *a* \$ *j* = *0* **.**
show *A* \$ *a* \$ *j* = *0*
proof (*cases a*=*from-nat i*)
case *True*
show *?thesis* **unfolding** *True* **using** *zero-ikA j-less-k* **unfolding** *is-zero-row-upt-k-def*
by *auto*
next
case *False* **note** *a-not-i*=*False*
show *?thesis*
proof (*cases a*=*?least*)
case *True*
show *?thesis*
using *zero-interchange* **unfolding** *True is-zero-row-upt-k-def* **using** *j-less-k*
by *auto*
next
case *False* **note** *a-not-least*=*False*
have *?interchange* \$ *a* \$ *j* = *A* \$ *a* \$ *j* **using** *a-not-least a-not-i*
by (*metis* (*erased*, *lifting*) *interchange-rows-preserves*)
thus *?thesis* **unfolding** ∗ **..**
qed
qed
qed
hence *zero-b-k*: *is-zero-row-upt-k b k A*
by (*metis ab e echelon-form-upt-k-condition1*)
have *i-le-a*: *from-nat i*≤*a*
unfolding *i*
proof (*auto simp add*: *from-nat-to-nat-greatest from-nat-0*)
show *0* ≤ *a* **by** (*metis least-mod-type*)
fix *m* **assume** *m*: ¬ *is-zero-row-upt-k m k A*
have (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) < *a*
by (*metis* (*no-types*, *lifting*) *GreatestI-ex neq-iff*
e echelon-form-upt-k-condition1 m zero-a-k)
thus (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *a* **by** (*metis le-Suc*)
qed
have *i-not-b*: *from-nat i* ≠ *b* **using** *i-le-a ab* **by** *simp*
show *is-zero-row-upt-k b k ?bezout-iterate*
proof (*unfold is-zero-row-upt-k-def*, *clarify*)
fix *j*::*′cols* **assume** *j-less-k*: *to-nat j* < *k*
have *?bezout-iterate* \$ *b* \$ *j* = *?interchange* \$ *b* \$ *j*
proof (*rule bezout-iterate-preserves*)
show *echelon-form-upt-k ?interchange k*
proof (*rule echelon-form-upt-k-interchange*[*OF e zero-ikA Amk-not-0* - *k*])
show *from-nat i* ≤ *m* **using** *im* **by** *auto*
qed
show *is-bezout-ext bezout* **using** *ib* **.**

```
      show ?interchange $ from-nat i $ from-nat k ≠ 0
        by (metis (mono-tags, lifting) Amk-not-0 LeastI-ex
          dual-order.strict-iff-order im interchange-rows-i)
      show nrows A − Suc 0 < nrows ?interchange unfolding nrows-def by simp
       show j < from-nat k by (metis from-nat-mono from-nat-to-nat-id j-less-k k
ncols-def)
      show to-nat (from-nat i::'rows) ≤ nrows A − Suc 0
        by (simp add: nrows-def le-diff-conv2 Suc-le-eq to-nat-less-card)
      show k < ncols ?interchange using k unfolding ncols-def by auto
     show is-zero-row-upt-k (from-nat i) k ?interchange by (rule zero-interchange)
    qed
    also have ... = A $ b $ j
    proof (cases b=?least)
      case True
      have ?interchange $ b $ j = A $ (from-nat i) $ j using True by auto
      also have ... = A $ b $ j
        using zero-b-k zero-ikA j-less-k unfolding is-zero-row-upt-k-def by auto
      finally show ?thesis .
    next
      case False
      show ?thesis using False using interchange-rows-preserves[OF i-not-b]
        by (metis (no-types, lifting))
    qed
     also have ... = 0 using zero-b-k j-less-k unfolding is-zero-row-upt-k-def by
auto
    finally show ?bezout-iterate $ b $ j = 0 .
  qed
  show ?bezout-iterate $ b $ from-nat k = 0
  proof (rule bezout-iterate-zero-column-k[OF - ib])
    show echelon-form-upt-k ?interchange k
    proof (rule echelon-form-upt-k-interchange[OF e zero-ikA Amk-not-0 - k])
      show from-nat i ≤ m using im by auto
    qed
    show ?interchange $ from-nat i $ from-nat k ≠ 0
      by (metis (mono-tags, lifting) Amk-not-0 LeastI-ex
        dual-order.strict-iff-order im interchange-rows-i)
    show nrows A − Suc 0 < nrows ?interchange unfolding nrows-def by simp
    show from-nat i < b by (metis ab i-le-a le-less-trans)
    show k < ncols ?interchange by (metis (full-types, lifting) k ncols-def)
    show to-nat b ≤ nrows A − Suc 0
     by (metis Suc-pred leD not-less-eq-eq nrows-def to-nat-less-card zero-less-card-finite)
    show is-zero-row-upt-k (from-nat i) k ?interchange by (rule zero-interchange)
  qed
qed


lemma condition2-part1:
  fixes A::'a::{bezout-ring}^'cols::{mod-type}^'rows::{mod-type} and k bezout i
  defines i:i≡(if ∀m. is-zero-row-upt-k m k A then 0
```

else to-nat ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)) + *1*)
defines *B*:*B* ≡ *fst* ((*echelon-form-of-column-k bezout*) (*A*,*i*) *k*)
assumes *e*: *echelon-form-upt-k A k*
and *ab*: *a* < *b* **and** *not-zero-aB*: ¬ *is-zero-row-upt-k a* (*Suc k*) *B*
and *not-zero-bB*: ¬ *is-zero-row-upt-k b* (*Suc k*) *B*
and *all-zero*: ∀ *m*≥*from-nat i*. *A* $ *m* $ *from-nat k* = *0*
shows (*LEAST n*. *A* $ *a* $ *n* ≠ *0*) < (*LEAST n*. *A* $ *b* $ *n* ≠ *0*)
proof −
have *B-eq-A*: *B*=*A*
unfolding *B echelon-form-of-column-k-def Let-def fst-conv snd-conv*
using *all-zero* **by** *auto*
show *?thesis*
proof (*cases* ∀ *m*. *is-zero-row-upt-k m k A*)
case *True*
have *i0*: *i* = *0* **unfolding** *i* **using** *True* **by** *simp*
have *is-zero-row-upt-k a k B* **using** *True* **unfolding** *B-eq-A* **by** *auto*
moreover have *B* $ *a* $ *from-nat k* = *0* **using** *all-zero* **unfolding** *i0 from-nat-0*

by (*metis B-eq-A least-mod-type*)
ultimately have *is-zero-row-upt-k a* (*Suc k*) *B* **by** (*rule is-zero-row-upt-k-suc*)
thus *?thesis* **using** *not-zero-aB* **by** *contradiction*
next
case *False* **note** *not-all-zero*=*False*
have *i2*: *i* = *to-nat* ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)) + *1*
unfolding *i* **using** *False* **by** *auto*
have *not-zero-aA*: ¬ *is-zero-row-upt-k a k A*
by (*metis* (*erased*, *lifting*) *B-eq-A GreatestI-ex add-to-nat-def all-zero neq-iff*
e
echelon-form-upt-k-condition1 i2 is-zero-row-upt-k-suc le-Suc
not-all-zero not-zero-aB to-nat-1)
moreover have *not-zero-bA*: ¬ *is-zero-row-upt-k b k A*
by (*metis* (*erased*, *lifting*) *B-eq-A GreatestI-ex add-to-nat-def all-zero neq-iff*
e
echelon-form-upt-k-condition1 i2 is-zero-row-upt-k-suc le-Suc
not-all-zero not-zero-bB to-nat-1)
ultimately show *?thesis* **using** *echelon-form-upt-k-condition2*[*OF e ab*] **by**
simp
qed
qed

lemma *condition2-part2*:
fixes *A*::'*a*::{*bezout-ring*} ^'*cols*::{*mod-type*} ^'*rows*::{*mod-type*} **and** *k bezout i*
defines *i*:*i*≡(*if* ∀ *m*. *is-zero-row-upt-k m k A then 0 else*
to-nat ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)) + *1*)
assumes *e*: *echelon-form-upt-k A k*
and *ab*: *a* < *b*
and *all-zero*: ∀ *m*>*from-nat* (*nrows A*). *A* $ *m* $ *from-nat k* = *0*
and *i-nrows*: *i* = *nrows A*
shows (*LEAST n*. *A* $ *a* $ *n* ≠ *0*) < (*LEAST n*. *A* $ *b* $ *n* ≠ *0*)

proof −
  have *not-all-zero*: ¬ (∀ *m*. *is-zero-row-upt-k m k A*)
    by (*metis i i-nrows nrows-not-0*)
  have (*GREATEST m*. ¬ *is-zero-row-upt-k m k A*) + *1* = *0*
    by (*metis* (*mono-tags*, *lifting*) *add-0-right One-nat-def Suc-le′ add-Suc-right i i-nrows*
      less-not-refl less-trans-Suc nrows-def to-nat-less-card to-nat-mono zero-less-card-finite)
  hence *g-minus-1*: (*GREATEST m*. ¬ *is-zero-row-upt-k m k A*) = − *1* **by** (*simp add*: *a-eq-minus-1*)
  have ¬ *is-zero-row-upt-k a k A*
  proof (*rule greatest-ge-nonzero-row′*[*OF e - not-all-zero*])
    show *a* ≤ (*GREATEST m*. ¬ *is-zero-row-upt-k m k A*)
      by (*simp add*: *Greatest-is-minus-1 g-minus-1*)
  qed
  moreover have ¬ *is-zero-row-upt-k b k A*
  proof (*rule greatest-ge-nonzero-row′*[*OF e - not-all-zero*])
    show *b* ≤ (*GREATEST m*. ¬ *is-zero-row-upt-k m k A*)
      by (*simp add*: *Greatest-is-minus-1 g-minus-1*)
  qed
  ultimately show *?thesis* **using** *echelon-form-upt-k-condition2*[*OF e ab*] **by** *simp*
qed

lemma *condition2-part3*:
  fixes *A*::′*a*::{*bezout-ring*}^′*cols*::{*mod-type*}^′*rows*::{*mod-type*} **and** *k bezout i*
  defines *i*:*i*≡(*if* ∀ *m*. *is-zero-row-upt-k m k A then 0*
    else to-nat ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)) + *1*)
  defines *B*:*B* ≡ *fst* ((*echelon-form-of-column-k bezout*) (*A*,*i*) *k*)
  assumes *e*: *echelon-form-upt-k A k* **and** *k*: *k*<*ncols A*
  and *ab*: *a* < *b* **and** *not-zero-aB*: ¬ *is-zero-row-upt-k a* (*Suc k*) *B*
  and *not-zero-bB*: ¬ *is-zero-row-upt-k b* (*Suc k*) *B*
  and *all-zero*: ∀ *m*>*from-nat i*. *A* $ *m* $ *from-nat k* = *0*
  and *i-ma*: *from-nat i* ≤ *ma* **and** *A-ma-k*: *A* $ *ma* $ *from-nat k* ≠ *0*
  shows (*LEAST n*. *A* $ *a* $ *n* ≠ *0*) < (*LEAST n*. *A* $ *b* $ *n* ≠ *0*)
proof −
  have *B-eq-A*: *B*=*A*
    unfolding *B echelon-form-of-column-k-def Let-def fst-conv snd-conv*
    using *all-zero* **by** *simp*
  have *not-all-zero*: ¬ (∀ *m*. *is-zero-row-upt-k m k A*)
    by (*metis B-eq-A ab all-zero from-nat-0 i is-zero-row-upt-k-suc*
      le-less-trans least-mod-type not-zero-bB)
  have *i2*: *i* = *to-nat* ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)) + *1*
    unfolding *i* **using** *not-all-zero* **by** *auto*
  have *not-zero-aA*: ¬ *is-zero-row-upt-k a k A*
  proof −
    have $\bigwedge x_1\ x_2$. *from-nat* (*to-nat* (x_1::′*rows*) + *1*) ≤ x_2 ∨ ¬ x_1 < x_2
      by (*metis* (*no-types*) *add-to-nat-def le-Suc to-nat-1*)
    moreover
    { **assume** ¬ *is-zero-row-upt-k b k A*
     hence ¬ *is-zero-row-upt-k a k A* **using** *ab e echelon-form-upt-k-condition1* **by**

blast }
ultimately show ¬ *is-zero-row-upt-k a k A*
by (*metis B-eq-A greatest-less-zero-row ab all-zero le-imp-less-or-eq e i2 is-zero-row-upt-k-suc not-all-zero not-zero-aB not-zero-bB*)
qed
show *?thesis*
proof (*cases* ¬ *is-zero-row-upt-k b k A*)
case *True*
thus *?thesis* **using** *not-zero-aA echelon-form-upt-k-condition2*[*OF e ab*] **by** *simp*
next
case *False* **note** *zero-bA=False*
obtain *v* **where** *Aav*: $A \$ a \$ v \neq 0$ **and** *v*: *v<from-nat k*
using *not-zero-aA* **unfolding** *is-zero-row-upt-k-def*
by (*metis from-nat-mono from-nat-to-nat-id k ncols-def*)
have *least-v*: (*LEAST n.* $A \$ a \$ n \neq 0$) $\leq v$ **by** (*rule Least-le*, *simp add*: *Aav*)
have *b-ge-greatest*: *b*>(*GREATEST n.* ¬ *is-zero-row-upt-k n k A*)
using *False* **by** (*simp add*: *greatest-less-zero-row e not-all-zero*)
have *i-eq-b*: *from-nat i* = *b*
proof (*rule ccontr*, *cases from-nat i* < *b*)
case *True*
hence *Abk-0*: $A \$ b \$$ *from-nat k* = *0* **using** *all-zero* **by** *auto*
have *is-zero-row-upt-k b* (*Suc k*) *B*
proof (*rule is-zero-row-upt-k-suc*)
show *is-zero-row-upt-k b k B* **using** *zero-bA* **unfolding** *B-eq-A* **by** *simp*
show $B \$ b \$$ *from-nat k* = *0* **using** *Abk-0* **unfolding** *B-eq-A* **by** *simp*
qed
thus *False* **using** *not-zero-bB* **by** *contradiction*
next
case *False*
assume *i-not-b*: *from-nat i* ≠ *b*
hence *b-less-i*: *from-nat i* > *b* **using** *False* **by** *simp*
thus *False* **using** *b-ge-greatest* **unfolding** *i*
by (*metis* (*no-types*, *lifting*) *False Suc-less add-to-nat-def i2 i-not-b to-nat-1*)
qed
have *Abk-not-0*: $A \$ b \$$ *from-nat k* ≠ *0*
using *False not-zero-bB* **unfolding** *B-eq-A is-zero-row-upt-k-def*
by (*metis B-eq-A False is-zero-row-upt-k-suc not-zero-bB*)
have (*LEAST n.* $A \$ b \$ n \neq 0$) = *from-nat k*
proof (*rule Least-equality*)
show $A \$ b \$$ *from-nat k* ≠ *0* **by** (*rule Abk-not-0*)
show $\bigwedge y.\ A \$ b \$ y \neq 0 \Longrightarrow$ *from-nat k* ≤ *y*
by (*metis False is-zero-row-upt-k-def k ncols-def not-less to-nat-from-nat-id to-nat-mono*)
qed
thus *?thesis* **using** *least-v v* **by** *auto*
qed
qed

lemma *condition2-part4*:
fixes *A*::′*a*::{*bezout-ring*}^′*cols*::{*mod-type*}^′*rows*::{*mod-type*} **and** *k bezout i*
defines *i*:*i*≡(*if* ∀ *m*. *is-zero-row-upt-k m k A then 0*
else to-nat ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)) + *1*)
assumes *e*: *echelon-form-upt-k A k*
and *ab*: *a* < *b*
and *i-nrows*: *i* = *nrows A*
shows (*LEAST n*. *A* \$ *a* \$ *n* ≠ *0*) < (*LEAST n*. *A* \$ *b* \$ *n* ≠ *0*)
proof −
have *not-all-zero*: ¬ (∀ *m*. *is-zero-row-upt-k m k A*) **by** (*metis i-nrows i nrows-not-0*)
then have *i* = *to-nat* ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)) + *1* **by** (*simp add*: *i*)
then have *nrows A* = *to-nat* ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)) + *1* **by** (*simp add*: *i-nrows*)
then have *CARD*(′*rows*) = *mod-type-class*.*to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*
unfolding *nrows-def* **.**
then have (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1* = *0*
using *to-nat-plus-one-less-card* **by** *auto*
hence *g*: (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) = −*1* **by** (*simp add*: *a-eq-minus-1*)
have ¬ *is-zero-row-upt-k a k A*
proof (*rule greatest-ge-nonzero-row*′[*OF e* - *not-all-zero*])
show *a* ≤ (*GREATEST m*. ¬ *is-zero-row-upt-k m k A*) **by** (*simp add*: *Greatest-is-minus-1 g*)
qed
moreover have ¬ *is-zero-row-upt-k b k A*
proof (*rule greatest-ge-nonzero-row*′[*OF e* - *not-all-zero*])
show *b* ≤ (*GREATEST m*. ¬ *is-zero-row-upt-k m k A*) **by** (*simp add*: *Greatest-is-minus-1 g*)
qed
ultimately show *?thesis* **using** *echelon-form-upt-k-condition2*[*OF e ab*] **by** *simp*
qed

lemma *condition2-part5*:
fixes *A*::′*a*::{*bezout-domain*}^′*cols*::{*mod-type*}^′*rows*::{*mod-type*} **and** *k bezout i*
defines *i*:*i*≡(*if* ∀ *m*. *is-zero-row-upt-k m k A then 0*
else to-nat ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)) + *1*)
defines *B*:*B* ≡ *fst* ((*echelon-form-of-column-k bezout*) (*A*,*i*) *k*)
assumes *ib*: *is-bezout-ext bezout* **and** *e*: *echelon-form-upt-k A k* **and** *k*: *k*<*ncols A*
and *ab*: *a* < *b* **and** *not-zero-aB*: ¬ *is-zero-row-upt-k a* (*Suc k*) *B*
and *not-zero-bB*: ¬ *is-zero-row-upt-k b* (*Suc k*) *B*
and *i-m*:*from-nat i* < *m*
and *A-mk*: *A* \$ *m* \$ *from-nat k* ≠ *0*
and *i-not-nrows*: *i* ≠ *nrows A*
shows (*LEAST n*. *B* \$ *a* \$ *n* ≠ *0*) < (*LEAST n*. *B* \$ *b* \$ *n* ≠ *0*)
proof −
have *B-eq*: *B* = *bezout-iterate* (*interchange-rows A* (*from-nat i*)

(*LEAST n. A* \$ *n* \$ *from-nat k* ≠ *0* ∧ *from-nat i* ≤ *n*))
(*nrows A* − *Suc 0*) (*from-nat i*) (*from-nat k*) *bezout*
unfolding *B echelon-form-of-column-k-def Let-def fst-conv snd-conv*
using *i-m A-mk i-not-nrows* **by** *auto*
let *?least*=(*LEAST n. A* \$ *n* \$ *from-nat k* ≠ *0* ∧ *from-nat i* ≤ *n*)
let *?interchange*=*interchange-rows A* (*from-nat i*) *?least*
let *?greatest*=(*GREATEST n.* ¬ *is-zero-row-upt-k n k A*)
have *nrows-less*: *nrows A* − *Suc 0* < *nrows ?interchange* **unfolding** *nrows-def*
by *auto*
have *interchange-ik-not-zero*: *?interchange* \$ *from-nat i* \$ *from-nat k* ≠ *0*
by (*metis* (*mono-tags*, *lifting*) *A-mk LeastI-ex dual-order.strict-iff-order*
i-m interchange-rows-i)
have *k2*: *k* < *ncols ?interchange* **using** *k* **unfolding** *ncols-def* **by** *simp*
have *to-nat-b*: *to-nat b* ≤ *nrows A* − *Suc 0*
by (*metis Suc-pred leD not-less-eq-eq nrows-def to-nat-less-card zero-less-card-finite*)
have *to-nat-from-nat-i*: *to-nat* (*from-nat i*::′*rows*) ≤ *nrows A* − *Suc 0*
using *i-not-nrows* **unfolding** *nrows-def*
by (*metis Suc-pred less-Suc-eq-le to-nat-less-card zero-less-card-finite*)
have *not-all-zero*: ¬ (∀ *m. is-zero-row-upt-k m k A*)
proof (*rule ccontr*)
assume *all-zero*: ¬¬(∀ *m. is-zero-row-upt-k m k A*)
hence *zero-aA*: *is-zero-row-upt-k a k A* **and** *zero-bA*: *is-zero-row-upt-k b k A*
by *auto*
have *echelon-interchange*: *echelon-form-upt-k ?interchange k*
proof (*rule echelon-form-upt-k-interchange*[*OF e - A-mk - k*])
show *is-zero-row-upt-k* (*from-nat i*) *k A* **using** *all-zero* **by** *auto*
show *from-nat i* ≤ *m* **using** *i-m* **by** *auto*
qed
have *zero-i-interchange*: *is-zero-row-upt-k* (*from-nat i*) *k ?interchange*
using *all-zero* **unfolding** *is-zero-row-upt-k-def* **by** *auto*
have *zero-bB*: *is-zero-row-upt-k b k B*
proof (*unfold is-zero-row-upt-k-def*, *clarify*)
fix *j*::′*cols* **assume** *j*: *to-nat j* < *k*
have *B* \$ *b* \$ *j* = *?interchange* \$ *b* \$ *j*
proof (*unfold B-eq*, *rule bezout-iterate-preserves*
[*OF echelon-interchange ib interchange-ik-not-zero nrows-less -*
to-nat-from-nat-i k2 zero-i-interchange])
show *j* < *from-nat k* **using** *j* **by** (*metis from-nat-mono from-nat-to-nat-id*
k ncols-def)
qed
also have ... = *0*
using *all-zero j*
unfolding *is-zero-row-upt-k-def interchange-rows-def* **by** *auto*
finally show *B* \$ *b* \$ *j* = *0* **.**
qed
have *i-not-b*: *from-nat i* ≠ *b*
using *i all-zero ab least-mod-type* **by** (*metis leD from-nat-0*)
have *B* \$ *b* \$ *from-nat k* ≠ *0* **by** (*metis is-zero-row-upt-k-suc not-zero-bB*
zero-bB)

moreover have *B* \$ *b* \$ *from-nat k* = *0*
proof (*unfold B-eq*, *rule bezout-iterate-zero-column-k*
[*OF echelon-interchange ib interchange-ik-not-zero nrows-less*
- *k2 to-nat-b zero-i-interchange*])
show *from-nat i* < *b*
by (*metis all-zero antisym-conv1 from-nat-0 i i-not-b least-mod-type*)
qed
ultimately show *False* **by** *contradiction*
qed
have *i2*: *i* = *to-nat* (*?greatest*) + *1*
unfolding *i* **using** *not-all-zero* **by** *auto*
have *g-rw*: (*from-nat* (*to-nat ?greatest* + *1*))
= *?greatest* + *1* **by** (*metis add-to-nat-def to-nat-1*)
have *zero-least-kA*: *is-zero-row-upt-k ?least k A*
proof (*rule row-greater-greatest-is-zero*)
have *?greatest* < *from-nat i*
by (*metis Suc-eq-plus1 Suc-le′ add-to-nat-def neq-iff from-nat-0 from-nat-mono*

i2 i-not-nrows not-less-eq nrows-def to-nat-1 to-nat-less-card zero-less-Suc)
also have ... ≤ *?least*
by (*metis* (*mono-tags*, *lifting*) *A-mk LeastI-ex dual-order.strict-iff-order i-m*)
finally show *?greatest* < *?least* **.**
qed

have *zero-ik-interchange*: *is-zero-row-upt-k* (*from-nat i*) *k ?interchange*
by (*metis* (*no-types*, *lifting*) *interchange-rows-i is-zero-row-upt-k-def zero-least-kA*)
have *echelon-form-interchange*: *echelon-form-upt-k ?interchange k*
proof (*rule echelon-form-upt-k-interchange*[*OF e - A-mk - k*])
show *is-zero-row-upt-k* (*from-nat i*) *k A*
by (*metis* (*mono-tags*) *greatest-ge-nonzero-row′ Greatest-is-minus-1 Suc-le′*
a-eq-minus-1 e g-rw i2 row-greater-greatest-is-zero zero-least-kA)
show *from-nat i* ≤ *m* **using** *i-m* **by** *simp*
qed
have *b-le-i*: *b* ≤ *from-nat i*
proof (*rule ccontr*)
assume ¬ *b* ≤ *from-nat i*
hence *b-gr-i*: *b* > *from-nat i* **by** *simp*
have *is-zero-row-upt-k b* (*Suc k*) *B*
proof (*rule is-zero-row-upt-k-suc*)
show *B* \$ *b* \$ *from-nat k* = *0*
by (*unfold B-eq*, *rule bezout-iterate-zero-column-k*[*OF echelon-form-interchange*
ib
interchange-ik-not-zero nrows-less b-gr-i k2 to-nat-b zero-ik-interchange])
show *is-zero-row-upt-k b k B*
proof (*unfold is-zero-row-upt-k-def*, *clarify*)
fix *j*::′*cols*
assume *j-k*: *to-nat j* < *k*
have *B* \$ *b* \$ *j* = *?interchange* \$ *b* \$ *j*
proof (*unfold B-eq*, *rule bezout-iterate-preserves*[*OF echelon-form-interchange*

```
ib
        interchange-ik-not-zero nrows-less - to-nat-from-nat-i k2 zero-ik-interchange])
          show j < from-nat k by (metis from-nat-mono from-nat-to-nat-id j-k k
ncols-def)
       qed
       also have ... = 0
       by (metis (erased, lifting) b-gr-i echelon-form-interchange echelon-form-upt-k-condition1

          is-zero-row-upt-k-def j-k zero-ik-interchange)
       finally show B $ b $ j = 0 .
     qed
   qed
   thus False using not-zero-bB by contradiction
 qed
 hence a-less-i: a < from-nat i using ab by simp
 have not-zero-aA: ¬ is-zero-row-upt-k a k A
 proof (rule greatest-ge-nonzero-row'[OF e - not-all-zero])
   show  a ≤ (GREATEST m. ¬ is-zero-row-upt-k m k A)
     using a-less-i unfolding i2 g-rw
     by (metis le-Suc not-le)
 qed
 have least-eq1:(LEAST n. B $ a $ n ≠ 0) = (LEAST n. A $ a $ n ≠ 0)
 proof (rule Least-equality)
   have B $ a $ (LEAST n. A $ a $ n ≠ 0) = ?interchange $ a $ (LEAST n. A
$ a $ n ≠ 0)
    proof (unfold B-eq, rule bezout-iterate-preserves[OF echelon-form-interchange
ib
       interchange-ik-not-zero nrows-less - to-nat-from-nat-i k2 zero-ik-interchange])
     obtain j::'cols where j: to-nat j < k and Aaj: A $ a $ j ≠ 0
       using not-zero-aA unfolding is-zero-row-upt-k-def by auto
     have (LEAST n. A $ a $ n ≠ 0) ≤ j by (rule Least-le, simp add: Aaj)
     also have ... < from-nat k
       by (metis (full-types) from-nat-mono from-nat-to-nat-id j k ncols-def)
     finally show (LEAST n. A $ a $ n ≠ 0) < from-nat k .
   qed
   also have ... = A $ a $ (LEAST n. A $ a $ n ≠ 0)
     by (metis (no-types, lifting) ab b-le-i interchange-rows-preserves
       leD not-zero-aA zero-least-kA)
   also have ... ≠ 0
     by (metis (mono-tags) LeastI is-zero-row-def' is-zero-row-imp-is-zero-row-upt
not-zero-aA)
   finally show B $ a $ (LEAST n. A $ a $ n ≠ 0) ≠ 0 .
   fix y assume Bay:B $ a $ y ≠ 0
   show (LEAST n. A $ a $ n ≠ 0) ≤ y
   proof (cases y<from-nat k)
     case True
     have B $ a $ y = ?interchange $ a $ y
       by (unfold B-eq, rule bezout-iterate-preserves[OF echelon-form-interchange
ib
```

interchange-ik-not-zero nrows-less True to-nat-from-nat-i k2 zero-ik-interchange])
also have ... = A \$ a \$ y
by (*metis* (*no-types*, *lifting*) *ab b-le-i interchange-rows-preserves*
leD not-zero-aA zero-least-kA)
finally have A \$ a \$ $y \neq 0$ **using** *Bay* **by** *simp*
thus *?thesis* **using** *Least-le* **by** *fast*
next
case *False*
obtain *j*::*'cols* **where** *j*: *to-nat* $j < k$ **and** *Aaj*: A \$ a \$ $j \neq 0$
using *not-zero-aA* **unfolding** *is-zero-row-upt-k-def* **by** *auto*
have (*LEAST n*. A \$ a \$ $n \neq 0$) $\leq j$ **by** (*rule Least-le*, *simp add*: *Aaj*)
also have ... < *from-nat* k
by (*metis* (*full-types*) *from-nat-mono from-nat-to-nat-id j k ncols-def*)
also have ...$\leq y$ **using** *False* **by** *auto*
finally show *?thesis* **by** *simp*
qed
qed
show *?thesis*
proof (*cases b*=*from-nat i*)
case *True*
have *zero-bB*: *is-zero-row-upt-k b k B*
proof (*unfold is-zero-row-upt-k-def*, *clarify*)
fix *j*::*'cols* **assume** *jk*:*to-nat* $j < k$
have *jk2*: $j <$ *from-nat* k **by** (*metis from-nat-mono from-nat-to-nat-id jk k*
ncols-def)
have B \$ b \$ j = *?interchange* \$ b \$ j
by (*unfold B-eq*, *rule bezout-iterate-preserves*[*OF echelon-form-interchange*
ib
interchange-ik-not-zero nrows-less jk2 to-nat-from-nat-i k2 zero-ik-interchange])
also have ... = A \$ *?least* \$ j **unfolding** *True* **by** *auto*
also have ... = *0* **using** *zero-least-kA jk* **unfolding** *is-zero-row-upt-k-def* **by**
simp
finally show B \$ b \$ j = *0* **.**
qed
have *least-eq2*: (*LEAST n*. B \$ b \$ $n \neq 0$) = *from-nat* k
proof (*rule Least-equality*)
show B \$ b \$ *from-nat* $k \neq 0$
unfolding *B-eq True*
by (*rule bezout-iterate-not-zero*[*OF interchange-ik-not-zero*
nrows-less to-nat-from-nat-i ib])
show $\bigwedge y$. B \$ b \$ $y \neq 0 \Longrightarrow$ *from-nat* $k \leq y$
by (*metis is-zero-row-upt-k-def le-less-linear to-nat-le zero-bB*)
qed
obtain *j*::*'cols* **where** *j*: *to-nat* $j < k$ **and** *Abj*: A \$ a \$ $j \neq 0$
using *not-zero-aA* **unfolding** *is-zero-row-upt-k-def* **by** *auto*
have (*LEAST n*. A \$ a \$ $n \neq 0$) $\leq j$ **by** (*rule Least-le*, *simp add*: *Abj*)
also have ... < *from-nat* k
by (*metis* (*full-types*) *from-nat-mono from-nat-to-nat-id j k ncols-def*)
finally show *?thesis* **unfolding** *least-eq1 least-eq2* **.**

next
  case *False* **note** *b-not-i=False*
  hence *b-less-i*: $b <$ *from-nat* i **using** *b-le-i* **by** *simp*
  have *not-zero-bA*: $\neg$ *is-zero-row-upt-k b k A*
  proof (*rule greatest-ge-nonzero-row'*[*OF e - not-all-zero*])
    show $b \leq$ (*GREATEST m.* $\neg$ *is-zero-row-upt-k m k A*)
      using *b-less-i* **unfolding** *i2 g-rw*
      by (*metis le-Suc not-le*)
  qed
  have *least-eq2*: (*LEAST n. B* \$ *b* \$ $n \neq 0$) = (*LEAST n. A* \$ *b* \$ $n \neq 0$)
  proof (*rule Least-equality*)
    have *B* \$ *b* \$ (*LEAST n. A* \$ *b* \$ $n \neq 0$) = *?interchange* \$ *b* \$ (*LEAST n.*
A \$ *b* \$ $n \neq 0$)
    proof (*unfold B-eq*, *rule bezout-iterate-preserves*[*OF echelon-form-interchange*
ib
      interchange-ik-not-zero nrows-less - to-nat-from-nat-i k2 zero-ik-interchange])
      obtain *j::'cols* **where** *j*: *to-nat j* $< k$ **and** *Abj*: *A* \$ *b* \$ $j \neq 0$
        using *not-zero-bA* **unfolding** *is-zero-row-upt-k-def* **by** *auto*
      have (*LEAST n. A* \$ *b* \$ $n \neq 0$) $\leq j$ **by** (*rule Least-le*, *simp add*: *Abj*)
      also have ... < *from-nat k*
        by (*metis* (*full-types*) *from-nat-mono from-nat-to-nat-id j k ncols-def*)
      finally show (*LEAST n. A* \$ *b* \$ $n \neq 0$) < *from-nat k* **.**
    qed
    also have ... = *A* \$ *b* \$ (*LEAST n. A* \$ *b* \$ $n \neq 0$)
        by (*metis* (*mono-tags*) *b-not-i interchange-rows-preserves not-zero-bA*
zero-least-kA)
    also have ... $\neq 0$
    by (*metis* (*mono-tags*) *LeastI is-zero-row-def' is-zero-row-imp-is-zero-row-upt*
not-zero-bA)
    finally show *B* \$ *b* \$ (*LEAST n. A* \$ *b* \$ $n \neq 0$) $\neq 0$ **.**
    fix *y* **assume** *Bby*:*B* \$ *b* \$ $y \neq 0$
    show (*LEAST n. A* \$ *b* \$ $n \neq 0$) $\leq y$
    proof (*cases y*<*from-nat k*)
      case *True*
      have *B* \$ *b* \$ *y* = *?interchange* \$ *b* \$ *y*
        by (*unfold B-eq*, *rule bezout-iterate-preserves*[*OF echelon-form-interchange*
ib
      interchange-ik-not-zero nrows-less True to-nat-from-nat-i k2 zero-ik-interchange])
      also have ... = *A* \$ *b* \$ *y*
        by (*metis* (*mono-tags*) *b-not-i interchange-rows-preserves not-zero-bA*
zero-least-kA)
      finally have *A* \$ *b* \$ $y \neq 0$ **using** *Bby* **by** *simp*
      thus *?thesis* **using** *Least-le* **by** *fast*
    next
      case *False*
      obtain *j::'cols* **where** *j*: *to-nat j* $< k$ **and** *Abj*: *A* \$ *b* \$ $j \neq 0$
        using *not-zero-bA* **unfolding** *is-zero-row-upt-k-def* **by** *auto*
      have (*LEAST n. A* \$ *b* \$ $n \neq 0$) $\leq j$ **by** (*rule Least-le*, *simp add*: *Abj*)
      also have ... < *from-nat k*

by (*metis* (*full-types*) *from-nat-mono from-nat-to-nat-id j k ncols-def*)
also have ...$\leq$ *y* **using** *False* **by** *auto*
finally show *?thesis* **by** *simp*
qed
qed
show *?thesis*
unfolding *least-eq1 least-eq2*
by (*rule echelon-form-upt-k-condition2*[*OF e ab not-zero-aA not-zero-bA*])
qed
qed

lemma *echelon-echelon-form-column-k*:
fixes *A*::*'a*::{*bezout-domain*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k bezout*
defines *i*:*i* $\equiv$ (*if* $\forall$ *m. is-zero-row-upt-k m k A then 0*
else to-nat ((*GREATEST n.* $\neg$ *is-zero-row-upt-k n k A*)) + *1*)
defines *B*: *B* $\equiv$ *fst* ((*echelon-form-of-column-k bezout*) (*A*,*i*) *k*)
assumes *ib*: *is-bezout-ext bezout* **and** *e*: *echelon-form-upt-k A k* **and** *k*: *k*<*ncols A*
shows *echelon-form-upt-k B* (*Suc k*)
unfolding *echelon-form-upt-k-def echelon-form-of-column-k-def Let-def*
proof *auto*
fix *a b*
let *?B2*=(*fst* (*if* $\forall$ *m*$\geq$*from-nat i. A* \$ *m* \$ *from-nat k* = *0 then* (*A*, *i*)
else if $\forall$ *m*>*from-nat i. A* \$ *m* \$ *from-nat k* = *0 then* (*A*, *i* + *1*)
else (*bezout-iterate*
(*interchange-rows A* (*from-nat i*) (*LEAST n. A* \$ *n* \$ *from-nat k* $\neq$ *0* $\wedge$ *from-nat i* $\leq$ *n*))
(*nrows A* − *1*) (*from-nat i*) (*from-nat k*) *bezout*, *i* + *1*)))
show *is-zero-row-upt-k a* (*Suc k*) *B* $\Longrightarrow$ *a* < *b* $\Longrightarrow$ *is-zero-row-upt-k b* (*Suc k*) *B*

proof (*unfold B echelon-form-of-column-k-def Let-def fst-conv snd-conv*, *auto*)
assume *1*: *is-zero-row-upt-k a* (*Suc k*) *A* **and** *2*: *a* < *b*
and *3*: $\forall$ *m*$\geq$*from-nat i. A* \$ *m* \$ *from-nat k* = *0*
show *is-zero-row-upt-k b* (*Suc k*) *A* **by** (*rule condition1-part1*[*OF e 1 2 3*[*unfolded i*]])
next
assume *1*: *is-zero-row-upt-k a* (*Suc k*) *A* **and** *2*: *a* < *b*
and *3*: *i* = *nrows A* **and** *4*: $\forall$ *m*>*from-nat* (*nrows A*). *A* \$ *m* \$ *from-nat k* = *0*
show *is-zero-row-upt-k b* (*Suc k*) *A* **by** (*rule condition1-part2*[*OF e 1 2 3*[*unfolded i*] *4*])
next
fix *m*
assume *1*: *is-zero-row-upt-k a* (*Suc k*) *?B2*
and *2*: *a* < *b* **and** *3*: $\forall$ *m*>*from-nat i. A* \$ *m* \$ *from-nat k* = *0*
and *4*: *i* $\neq$ *nrows A* **and** *5*: *from-nat i* $\leq$ *m*
and *6*: *A* \$ *m* \$ *from-nat k* $\neq$ *0*
show *is-zero-row-upt-k b* (*Suc k*) *A*
using *condition1-part3*[*OF e ib - 2 - - - 6*]

```
      using 1 3 4 5 unfolding i echelon-form-of-column-k-def Let-def fst-conv
snd-conv by auto
  next
    fix m::'c assume 1: is-zero-row-upt-k a (Suc k) A and 2: i = nrows A
   show is-zero-row-upt-k b (Suc k) A by (rule condition1-part4[OF e 1 2[unfolded
i]])
  next
    let ?B2=(fst (if ∀m≥from-nat i. A $ m $ from-nat k = 0 then (A, i)
      else if ∀m>from-nat i. A $ m $ from-nat k = 0 then (A, i + 1)
      else (bezout-iterate
      (interchange-rows A (from-nat i)
      (LEAST n. A $ n $ from-nat k ≠ 0 ∧ from-nat i ≤ n))
      (nrows A − 1) (from-nat i) (from-nat k) bezout,
      i + 1)))
    fix m
    assume 1: is-zero-row-upt-k a (Suc k) ?B2
      and 2: a < b
      and 3: from-nat i < m
      and 4: A $ m $ from-nat k ≠ 0
      and 5: i ≠ nrows A
    show is-zero-row-upt-k b (Suc k)
      (bezout-iterate
        (interchange-rows A (from-nat i) (LEAST n. A $ n $ from-nat k ≠ 0 ∧
from-nat i ≤ n))
      (nrows A − Suc 0) (from-nat i) (from-nat k) bezout)
      using condition1-part5[OF ib e - 2 - 4 - k]
     using 1 3 5 unfolding i echelon-form-of-column-k-def Let-def fst-conv snd-conv

      by auto
  qed
next
  fix a b assume ab: a < b and not-zero-aB: ¬ is-zero-row-upt-k a (Suc k) B
    and not-zero-bB: ¬ is-zero-row-upt-k b (Suc k) B
  show (LEAST n. B $ a $ n ≠ 0) < (LEAST n. B $ b $ n ≠ 0)
  proof (unfold B echelon-form-of-column-k-def Let-def fst-conv snd-conv, auto)
    assume all-zero: ∀m≥from-nat i. A $ m $ from-nat k = 0
    show (LEAST n. A $ a $ n ≠ 0) < (LEAST n. A $ b $ n ≠ 0)
      using condition2-part1[OF e ab] not-zero-aB not-zero-bB all-zero
      unfolding B i by simp
  next
     assume 1: ∀m>from-nat (nrows A). A $ m $ from-nat k = 0 and 2: i =
nrows A
    show (LEAST n. A $ a $ n ≠ 0) < (LEAST n. A $ b $ n ≠ 0)
      using condition2-part2[OF e ab 1] 2 unfolding i by simp
  next
    fix ma
    assume 1: ∀m>from-nat i. A $ m $ from-nat k = 0
      and 2: from-nat i ≤ ma and 3: A $ ma $ from-nat k ≠ 0
    show (LEAST n. A $ a $ n ≠ 0) < (LEAST n. A $ b $ n ≠ 0)
```

using *condition2-part3*[*OF e k ab - - - - 3*]
using *1 2 not-zero-aB not-zero-bB* **unfolding** *i B*
by *auto*
next
assume $i = nrows\ A$
thus $(LEAST\ n.\ A\ \$\ a\ \$\ n \neq 0) < (LEAST\ n.\ A\ \$\ b\ \$\ n \neq 0)$
using *condition2-part4*[*OF e ab*] **unfolding** *i* **by** *simp*
next
let *?B2=bezout-iterate* (*interchange-rows A* (*from-nat i*)
(*LEAST n. A \$ n \$ from-nat k* $\neq$ *0* $\wedge$ *from-nat i* $\leq$ *n*))
(*nrows A* $-$ *Suc 0*) (*from-nat i*) (*from-nat k*) *bezout*
fix *m*
assume *1*: *from-nat i* $<$ *m*
and *2*: *A \$ m \$ from-nat k* $\neq$ *0*
and *3*: *i* $\neq$ *nrows A*
have *B-eq*: *B=?B2* **unfolding** *B echelon-form-of-column-k-def Let-def* **using** *1 2 3* **by** *auto*
show (*LEAST n. ?B2 \$ a \$ n* $\neq$ *0*) $<$ (*LEAST n. ?B2 \$ b \$ n* $\neq$ *0*)
using *condition2-part5*[*OF ib e k ab - - - 2*] *1 3 not-zero-aB not-zero-bB*
unfolding *i*[*symmetric*] *B*[*symmetric*] **unfolding** *B-eq* **by** *auto*
qed
qed

lemma *echelon-foldl-condition1*:
assumes *ib*: *is-bezout-ext bezout*
and *A \$ ma \$ from-nat* (*Suc k*) $\neq$ *0*
and *k*: *k<ncols A*
shows $\exists$ *m*. $\neg$ *is-zero-row-upt-k m* (*Suc* (*Suc k*))
(*bezout-iterate* (*interchange-rows A 0* (*LEAST n. A \$ n \$ from-nat* (*Suc k*) $\neq$ *0*))
(*nrows A* $-$ *Suc 0*) *0* (*from-nat* (*Suc k*)) *bezout*)
proof (*rule exI*[*of - 0*], *unfold is-zero-row-upt-k-def*,
auto, *rule exI*[*of - from-nat* (*Suc k*)], *rule conjI*)
show *to-nat* (*from-nat* (*Suc k*)) $<$ *Suc* (*Suc k*)
by (*metis from-nat-mono from-nat-to-nat-id less-irrefl not-less-eq to-nat-less-card*)
show *bezout-iterate* (*interchange-rows A 0* (*LEAST n. A \$ n \$ from-nat* (*Suc k*) $\neq$ *0*))
(*nrows A* $-$ *Suc 0*) *0* (*from-nat* (*Suc k*)) *bezout \$ 0 \$ from-nat* (*Suc k*) $\neq$ *0*
proof (*rule bezout-iterate-not-zero*[*OF - - - ib*])
show *interchange-rows A 0* (*LEAST n. A \$ n \$ from-nat* (*Suc k*) $\neq$ *0*) *\$ 0 \$ from-nat* (*Suc k*) $\neq$ *0*
by (*metis* (*mono-tags*, *lifting*) *LeastI-ex assms*(*2*) *interchange-rows-i*)
show *nrows A* $-$ *Suc 0* $<$ *nrows* (*interchange-rows A 0* (*LEAST n. A \$ n \$ from-nat* (*Suc k*) $\neq$ *0*))
unfolding *nrows-def* **by** *simp*
show *to-nat 0* $\leq$ *nrows A* $-$ *Suc 0* **unfolding** *to-nat-0 nrows-def* **by** *simp*
qed

qed

lemma *echelon-foldl-condition2*:
fixes $A::'a::\{bezout\text{-}ring\}^{\frown}'cols::\{mod\text{-}type\}^{\frown}'rows::\{mod\text{-}type\}$
assumes *n*: ¬ *is-zero-row-upt-k ma k A*
and *all-zero*: $\forall m \geq$ (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*)+*1*. *A* \$ *m* \$ *from-nat k* = *0*
shows (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) = (*GREATEST n.* ¬ *is-zero-row-upt-k n* (*Suc k*) *A*)
proof (*rule Greatest-equality*[*symmetric*])
show ¬ *is-zero-row-upt-k* (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) (*Suc k*) *A*
by (*metis GreatestI-ex n is-zero-row-upt-k-le*)
fix *y* **assume** *y*: ¬ *is-zero-row-upt-k y* (*Suc k*) *A*
show *y* ≤ (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*)
proof (*rule ccontr*)
assume ¬ *y* ≤ (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*)
hence *y2*: *y* > (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) **by** *simp*
hence *is-zero-row-upt-k y k A* **by** (*metis row-greater-greatest-is-zero*)
moreover have *A* \$ *y* \$ *from-nat k* = *0*
by (*metis* (*no-types*, *lifting*) *all-zero le-Suc y2*)
ultimately have *is-zero-row-upt-k y* (*Suc k*) *A* **by** (*rule is-zero-row-upt-k-suc*)
thus *False* **using** *y* **by** *contradiction*
qed
qed

lemma *echelon-foldl-condition3*:
fixes $A::'a::\{bezout\text{-}domain\}^{\frown}'cols::\{mod\text{-}type\}^{\frown}'rows::\{mod\text{-}type\}$
assumes *ib*: *is-bezout-ext bezout*
and *Am0*: *A* \$ *m* \$ *from-nat k* ≠ *0*
and *all-zero*: ∀ *m. is-zero-row-upt-k m k A*
and *e*: *echelon-form-upt-k A k*
and *k*: *k* < *ncols A*
shows *to-nat* (*GREATEST n.* ¬ *is-zero-row-upt-k n* (*Suc k*)
(*bezout-iterate* (*interchange-rows A 0* (*LEAST n. A* \$ *n* \$ *from-nat k* ≠ *0*))
(*nrows A* − (*Suc 0*)) *0* (*from-nat k*) *bezout*)) = *0*
proof (*unfold to-nat-eq-0*, *rule Greatest-equality*)
let *?interchange*=(*interchange-rows A 0* (*LEAST n. A* \$ *n* \$ *from-nat k* ≠ *0*))
let *?b*=(*bezout-iterate ?interchange* (*nrows A* − (*Suc 0*)) *0* (*from-nat k*) *bezout*)
have *b0k*: *?b* \$ *0* \$ *from-nat k* ≠ *0*
proof (*rule bezout-iterate-not-zero*[*OF - - - ib*])
show *interchange-rows A 0* (*LEAST n. A* \$ *n* \$ *from-nat k* ≠ *0*) \$ *0* \$ *from-nat k* ≠ *0*
by (*metis* (*mono-tags*, *lifting*) *LeastI Am0 interchange-rows-i*)
show *nrows A* − (*Suc 0*) < *nrows* (*interchange-rows A 0* (*LEAST n. A* \$ *n* \$ *from-nat k* ≠ *0*))
unfolding *nrows-def* **by** *simp*
show *to-nat 0* ≤ *nrows A* − (*Suc 0*) **unfolding** *to-nat-0 nrows-def* **by** *simp*
qed
have *least-eq*: (*LEAST n. A* \$ *n* \$ *from-nat k* ≠ *0*)

= (*LEAST n. A $ n $ from-nat k* $\neq$ *0* $\wedge$ *0* $\leq$ *n*)
by (*metis least-mod-type*)
have *all-zero-below*: $\forall$ *a>0. ?b $ a $ from-nat k = 0*
proof (*auto*)
fix *a::'rows*
assume *a*: *0 < a*
show *bezout-iterate* (*interchange-rows A 0* (*LEAST n. A $ n $ from-nat k* $\neq$ *0*))
(*nrows A* $-$ *Suc 0*) *0* (*from-nat k*) *bezout $ a $ from-nat k = 0*
proof (*rule bezout-iterate-zero-column-k*[*OF - ib - - a*])
show *echelon-form-upt-k* (*interchange-rows A 0* (*LEAST n. A $ n $ from-nat k* $\neq$ *0*)) *k*
proof(*unfold from-nat-0*[*symmetric*] *least-eq*,
rule echelon-form-upt-k-interchange[*OF e - Am0 - k*])
show *is-zero-row-upt-k* (*from-nat 0*) *k A* **by** (*metis all-zero*)
show *from-nat 0* $\leq$ *m* **unfolding** *from-nat-0* **by** (*metis least-mod-type*)
qed
show *interchange-rows A 0* (*LEAST n. A $ n $ from-nat k* $\neq$ *0*) *$ 0 $ from-nat k* $\neq$ *0*
by (*metis* (*mono-tags*, *lifting*) *Am0 LeastI interchange-rows-i*)
show *nrows A* $-$ *Suc 0 < nrows* (*interchange-rows A 0* (*LEAST n. A $ n $ from-nat k* $\neq$ *0*))
unfolding *nrows-def* **by** *simp*
show *k < ncols* (*interchange-rows A 0* (*LEAST n. A $ n $ from-nat k* $\neq$ *0*))
using *k* **unfolding** *ncols-def* **by** *simp*
show *to-nat a* $\leq$ *nrows A* $-$ *Suc 0*
by (*metis* (*erased*, *opaque-lifting*) *One-nat-def Suc-leI Suc-le-D diff-Suc-eq-diff-pred*

not-le nrows-def to-nat-less-card zero-less-diff)
show *is-zero-row-upt-k 0 k* (*interchange-rows A 0* (*LEAST n. A $ n $ from-nat k* $\neq$ *0*))
by (*metis all-zero interchange-rows-i is-zero-row-upt-k-def*)
qed
qed
show $\neg$ *is-zero-row-upt-k 0* (*Suc k*) *?b*
by (*metis b0k is-zero-row-upt-k-def k lessI ncols-def to-nat-from-nat-id*)
fix *y* **assume** *y*: $\neg$ *is-zero-row-upt-k y* (*Suc k*) *?b*
show *y* $\leq$ *0*
proof (*rule ccontr*)
assume $\neg$ *y* $\leq$ *0* **hence** *y2*: *y>0* **by** *simp*
have *is-zero-row-upt-k y* (*Suc k*) *?b*
proof (*rule is-zero-row-upt-k-suc*)
show *is-zero-row-upt-k y k ?b*
proof (*unfold is-zero-row-upt-k-def*, *clarify*)
fix *j::'cols* **assume** *j*: *to-nat j < k*
have *?b $ y $ j = ?interchange $ y $ j*
proof (*rule bezout-iterate-preserves*[*OF - ib*])
show *echelon-form-upt-k ?interchange k*
proof (*unfold least-eq from-nat-0*[*symmetric*],

```
          rule echelon-form-upt-k-interchange[OF e - Am0 - k])
        show is-zero-row-upt-k (from-nat 0) k A
          by (metis all-zero)
        show from-nat 0 ≤ m
          by (metis from-nat-0 least-mod-type)
      qed
      show ?interchange $ 0 $ from-nat k ≠ 0
       by (metis (mono-tags, lifting) Am0 LeastI interchange-rows-i)
       show nrows A − Suc 0 < nrows ?interchange unfolding nrows-def by
simp
      show j < from-nat k
        by (metis (full-types) j from-nat-mono from-nat-to-nat-id k ncols-def)
      show to-nat 0 ≤ nrows A − Suc 0
        by (metis le0 to-nat-0)
      show k < ncols ?interchange using k unfolding ncols-def by simp
      show is-zero-row-upt-k 0 k ?interchange
        by (metis all-zero interchange-rows-i is-zero-row-upt-k-def)
    qed
    also have ... = 0
      by (metis all-zero dual-order.strict-iff-order interchange-rows-j
        interchange-rows-preserves is-zero-row-upt-k-def j y2)
    finally show ?b $ y $ j = 0 .
  qed
  show ?b $ y $ from-nat k = 0 using all-zero-below using y2 by auto
 qed
 thus False using y by contradiction
qed
qed


lemma echelon-foldl-condition4:
 fixes A::'a::{bezout-ring}^'cols::{mod-type}^'rows::{mod-type}
 assumes all-zero: ∀ m>(GREATEST n. ¬ is-zero-row-upt-k n k A)+1.
 A $ m $ from-nat k = 0
 and greatest-nrows: Suc (to-nat (GREATEST n. ¬ is-zero-row-upt-k n k A)) ≠
nrows A
 and le-mb: (GREATEST n. ¬ is-zero-row-upt-k n k A)+1 ≤ mb
 and A-mb-k: A $ mb $ from-nat k ≠ 0
 shows Suc (to-nat (GREATEST n. ¬ is-zero-row-upt-k n k A)) =
 to-nat (GREATEST n. ¬ is-zero-row-upt-k n (Suc k) A)
proof −
 let ?greatest = (GREATEST n. ¬ is-zero-row-upt-k n k A)
 have mb-eq: mb=(GREATEST n. ¬ is-zero-row-upt-k n k A) + 1
  by (metis all-zero le-mb A-mb-k le-less )
 have (GREATEST n. ¬ is-zero-row-upt-k n k A) + 1
  = (GREATEST n. ¬ is-zero-row-upt-k n (Suc k) A)
 proof (rule Greatest-equality[symmetric])
  show ¬ is-zero-row-upt-k (?greatest + 1) (Suc k) A
   by (metis (no-types, lifting) A-mb-k is-zero-row-upt-k-def less-Suc-eq less-trans
```

```
      mb-eq not-less-eq to-nat-from-nat-id to-nat-less-card)
    fix y
    assume y: ¬ is-zero-row-upt-k y (Suc k) A
    show y ≤ ?greatest + 1
    proof (rule ccontr)
      assume ¬ y ≤ (GREATEST n. ¬ is-zero-row-upt-k n k A) + 1
      hence y-greatest: y > ?greatest + 1 by simp
      have is-zero-row-upt-k y (Suc k) A
      proof (rule is-zero-row-upt-k-suc)
        show is-zero-row-upt-k y k A
        proof (rule row-greater-greatest-is-zero)
          show ?greatest < y
            using y-greatest greatest-nrows unfolding nrows-def
            by (metis Suc-eq-plus1 dual-order.strict-implies-order
              le-Suc' suc-not-zero to-nat-plus-one-less-card')
        qed
        show A $ y $ from-nat k = 0
          using all-zero y-greatest
          unfolding from-nat-to-nat-greatest by auto
      qed
      thus False using y by contradiction
    qed
  qed
  thus ?thesis
   by (metis (mono-tags, lifting) Suc-eq-plus1 Suc-lessI add-to-nat-def greatest-nrows

      nrows-def to-nat-1 to-nat-from-nat-id to-nat-less-card)
qed

lemma echelon-foldl-condition5:
  fixes A::'a::{bezout-ring}^'cols::{mod-type}^'rows::{mod-type}
  assumes mb: ¬ is-zero-row-upt-k mb k A
  and nrows: Suc (to-nat (GREATEST n. ¬ is-zero-row-upt-k n k A)) = nrows A
  shows nrows A = Suc (to-nat (GREATEST n. ¬ is-zero-row-upt-k n (Suc k) A))
 by (metis (no-types, lifting) GreatestI Suc-lessI Suc-less-eq mb nrows from-nat-mono

   from-nat-to-nat-id is-zero-row-upt-k-le not-greater-Greatest nrows-def to-nat-less-card)



lemma echelon-foldl-condition6:
  fixes A::'a::{bezout-ring}^'cols::{mod-type}^'rows::{mod-type}
  assumes ib: is-bezout-ext bezout
  and g-mc: (GREATEST n. ¬ is-zero-row-upt-k n k A) + 1 ≤ mc
  and A-mc-k: A $ mc $ from-nat k ≠ 0
  shows ∃ m. ¬ is-zero-row-upt-k m (Suc k)
  (bezout-iterate (interchange-rows A ((GREATEST n. ¬ is-zero-row-upt-k n k A)
+ 1)
```

(*LEAST n. A $ n $ from-nat k* ≠ *0* ∧ (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *n*))
(*nrows A* − *Suc 0*) ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) *bezout*)
proof −
let *?greatest*=(*GREATEST n.* ¬ *is-zero-row-upt-k n k A*)
let *?interchange*=*interchange-rows A* (*?greatest* + *1*)
(*LEAST n. A $ n $ from-nat k* ≠ *0* ∧ *?greatest* + *1* ≤ *n*)
let *?B*=(*bezout-iterate ?interchange* (*nrows A* − *Suc 0*) (*?greatest* + *1*) (*from-nat k*) *bezout*)
have *?B* $ (*?greatest* + *1*) $ *from-nat k* ≠ *0*
proof (*rule bezout-iterate-not-zero*[*OF - - - ib*])
show *?interchange* $ (*?greatest* + *1*) $ *from-nat k* ≠ *0*
by (*metis* (*mono-tags*, *lifting*) *LeastI-ex g-mc A-mc-k interchange-rows-i*)
show *nrows A* − *Suc 0* < *nrows ?interchange* **unfolding** *nrows-def* **by** *simp*
show *to-nat* (*?greatest* + *1*) ≤ *nrows A* − *Suc 0*
by (*metis Suc-pred less-Suc-eq-le nrows-def to-nat-less-card zero-less-card-finite*)
qed
thus *?thesis*
by (*metis* (*no-types*, *lifting*) *from-nat-mono from-nat-to-nat-id is-zero-row-upt-k-def*

less-irrefl not-less-eq to-nat-less-card)
qed

lemma *echelon-foldl-condition7*:
fixes *A*::′*a*::{*bezout-domain*}^′*cols*::{*mod-type*}^′*rows*::{*mod-type*}
assumes *ib*: *is-bezout-ext bezout*
and *e*: *echelon-form-upt-k A k*
and *k*: *k*<*ncols A*
and *mb*: ¬ *is-zero-row-upt-k mb k A*
and *not-nrows*: *Suc* (*to-nat* (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*)) ≠ *nrows A*
and *g-mc*: (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *mc*
and *A-mc-k*: *A $ mc $ from-nat k* ≠ *0*
shows *Suc* (*to-nat* (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*)) =
to-nat (*GREATEST n.* ¬ *is-zero-row-upt-k n* (*Suc k*) (*bezout-iterate*
(*interchange-rows A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*)
(*LEAST n. A $ n $ from-nat k* ≠ *0* ∧ (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *n*))
(*nrows A* − *Suc 0*) ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) *bezout*))
proof −
let *?greatest*=(*GREATEST n.* ¬ *is-zero-row-upt-k n k A*)
let *?interchange*=*interchange-rows A* (*?greatest* + *1*)
(*LEAST n. A $ n $ from-nat k* ≠ *0* ∧ *?greatest* + *1* ≤ *n*)
let *?B*=(*bezout-iterate ?interchange* (*nrows A* − *Suc 0*) (*?greatest* + *1*) (*from-nat k*) *bezout*)
have *g-rw*: (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*

```
    = from-nat (to-nat ((GREATEST n. ¬ is-zero-row-upt-k n k A) + 1))
    unfolding from-nat-to-nat-id ..
  have B-gk:?B $ (?greatest + 1) $ from-nat k ≠ 0
  proof (rule bezout-iterate-not-zero[OF - - - ib])
      show ?interchange $ ((GREATEST n. ¬ is-zero-row-upt-k n k A) + 1) $
from-nat k ≠ 0
      by (metis (mono-tags, lifting) LeastI-ex g-mc A-mc-k interchange-rows-i)
    show nrows A − Suc 0 < nrows (?interchange) unfolding nrows-def by simp
    show to-nat (?greatest + 1) ≤ nrows A − Suc 0
      by (metis Suc-pred less-Suc-eq-le nrows-def to-nat-less-card
        zero-less-card-finite)
  qed
  have (GREATEST n. ¬ is-zero-row-upt-k n (Suc k) ?B) = ?greatest + 1
  proof (rule Greatest-equality)
    show ¬ is-zero-row-upt-k (?greatest + 1) (Suc k) ?B
      by (metis (no-types, lifting) B-gk from-nat-mono from-nat-to-nat-id
        is-zero-row-upt-k-def less-irrefl not-less-eq to-nat-less-card)
    fix y
    assume y: ¬ is-zero-row-upt-k y (Suc k) ?B
    show y ≤ ?greatest + 1
    proof (rule ccontr)
      assume ¬ y ≤ (GREATEST n. ¬ is-zero-row-upt-k n k A) + 1
      hence y-gr: y > (GREATEST n. ¬ is-zero-row-upt-k n k A) + 1 by simp
      hence y-gr2: y > (GREATEST n. ¬ is-zero-row-upt-k n k A)
        by (metis (erased, lifting) Suc-eq-plus1 leI le-Suc' less-irrefl less-trans
          not-nrows nrows-def suc-not-zero to-nat-plus-one-less-card')
      have echelon-interchange: echelon-form-upt-k ?interchange k
      proof (subst (1 2) from-nat-to-nat-id
          [of (GREATEST n. ¬ is-zero-row-upt-k n k A) + 1, symmetric],
            rule echelon-form-upt-k-interchange[OF e - A-mc-k - k])
        show is-zero-row-upt-k
          (from-nat (to-nat ((GREATEST n. ¬ is-zero-row-upt-k n k A) + 1))) k A
          by (metis Suc-eq-plus1 Suc-le' g-rw not-nrows nrows-def
            row-greater-greatest-is-zero suc-not-zero)
         show from-nat (to-nat ((GREATEST n. ¬ is-zero-row-upt-k n k A) + 1))
≤ mc
          by (metis g-mc g-rw)
      qed
      have i: ?interchange $ (?greatest + 1) $ from-nat k ≠ 0
        by (metis (mono-tags, lifting) A-mc-k LeastI-ex g-mc interchange-rows-i)
      have zero-greatest: is-zero-row-upt-k (?greatest + 1) k A
        by (metis Suc-eq-plus1 Suc-le' not-nrows nrows-def
          row-greater-greatest-is-zero suc-not-zero)
      {
        fix j::'cols assume to-nat j < k
        have ?greatest < ?greatest + 1
          by (metis greatest-less-zero-row e mb zero-greatest)
       also have ... ≤(LEAST n. A $ n $ from-nat k ≠ 0 ∧ (?greatest + 1) ≤ n)
          by (metis (mono-tags, lifting) A-mc-k LeastI-ex g-mc)
```

finally have *least-less*: *?greatest*
< (*LEAST n. A* $ *n* $ *from-nat k* ≠ *0* ∧ (*?greatest* + *1*) ≤ *n*) **.**
have *is-zero-row-upt-k* (*LEAST n. A* $ *n* $ *from-nat k* ≠ *0* ∧ (*?greatest* + *1*) ≤ *n*) *k A*
by (*rule row-greater-greatest-is-zero*[*OF least-less*])
}
hence *zero-g1*: *is-zero-row-upt-k* (*?greatest* + *1*) *k ?interchange*
unfolding *is-zero-row-upt-k-def* **by** *auto*
hence *zero-y*: *is-zero-row-upt-k y k ?interchange*
by (*metis* (*erased*, *lifting*) *echelon-form-upt-k-condition1′ echelon-interchange y-gr*)
have *is-zero-row-upt-k y* (*Suc k*) *?B*
proof (*rule is-zero-row-upt-k-suc*)
show *?B* $ *y* $ *from-nat k* = *0*
proof (*rule bezout-iterate-zero-column-k*[*OF echelon-interchange ib i - y-gr - - zero-g1*])
show *nrows A* − *Suc 0* < *nrows ?interchange* **unfolding** *nrows-def* **by** *simp*
show *k* < *ncols ?interchange* **using** *k* **unfolding** *ncols-def* **by** *simp*
show *to-nat y* ≤ *nrows A* − *Suc 0*
by (*metis One-nat-def Suc-eq-plus1 Suc-leI nrows-def*
le-diff-conv2 to-nat-less-card zero-less-card-finite)
qed
show *is-zero-row-upt-k y k ?B*
proof (*subst is-zero-row-upt-k-def*, *clarify*)
fix *j*::*′cols* **assume** *j*: *to-nat j* < *k*
have *?B* $ *y* $ *j* = *?interchange* $ *y* $ *j*
proof (*rule bezout-iterate-preserves*[*OF echelon-interchange ib i - - - - zero-g1*])
show *nrows A* − *Suc 0* < *nrows ?interchange* **unfolding** *nrows-def* **by** *simp*
show *j* < *from-nat k* **using** *j*
by (*metis* (*poly-guards-query*) *from-nat-mono from-nat-to-nat-id k ncols-def*)
show *to-nat* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) ≤ *nrows A* − *Suc 0*
by (*metis Suc-pred less-Suc-eq-le nrows-def to-nat-less-card zero-less-card-finite*)
show *k* < *ncols ?interchange* **using** *k* **unfolding** *ncols-def* **.**
qed
also have ... = *0* **using** *zero-y* **unfolding** *is-zero-row-upt-k-def* **using** *j* **by** *simp*
finally show *?B* $ *y* $ *j* = *0* **.**
qed
qed
thus *False* **using** *y* **by** *contradiction*
qed
qed
thus *?thesis*
by (*metis* (*erased*, *lifting*) *Suc-eq-plus1 add-to-nat-def not-nrows nrows-def*

suc-not-zero
    to-nat-1 to-nat-from-nat-id to-nat-plus-one-less-card')
qed

lemma
  fixes *A*::*'a*::{*bezout-domain*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
  assumes *k*: *k*<*ncols A* **and** *ib*: *is-bezout-ext bezout*
  shows *echelon-echelon-form-of-upt-k*:
  echelon-form-upt-k (*echelon-form-of-upt-k A k bezout*) (*Suc k*)
  and *foldl* (*echelon-form-of-column-k bezout*) (*A*, *0*) [*0*..<*Suc k*] =
  (*fst* (*foldl* (*echelon-form-of-column-k bezout*) (*A*, *0*) [*0*..<*Suc k*]),
  if ∀ *m*. *is-zero-row-upt-k m* (*Suc k*)
  (*fst* (*foldl* (*echelon-form-of-column-k bezout*) (*A*, *0*) [*0*..<*Suc k*])) *then 0*
  else to-nat (*GREATEST n*. ¬ *is-zero-row-upt-k n* (*Suc k*)
  (*fst* (*foldl* (*echelon-form-of-column-k bezout*) (*A*, *0*) [*0*..<*Suc k*]))) + *1*)
using *k*
proof (*induct k*)
  let *?interchange*=*interchange-rows A 0* (*LEAST n*. *A* $ *n* $ *0* ≠ *0*)
  have *i-rw*: (*if* ∀ *m*. *is-zero-row-upt-k m 0 A then 0*
    else to-nat (*GREATEST n*. ¬ *is-zero-row-upt-k n 0 A*) + *1*) = *0*
    unfolding *is-zero-row-upt-k-def* **by** *auto*
  show *echelon-form-upt-k* (*echelon-form-of-upt-k A 0 bezout*) (*Suc 0*)
    unfolding *echelon-form-of-upt-k-def*
    by (*auto*, *subst i-rw*[*symmetric*], *rule echelon-echelon-form-column-k*[*OF ib ech-elon-form-upt-k-0*],
      simp add: *ncols-def*)
  have *rw-upt*: [*0*..<*Suc 0*] = [*0*] **by** *simp*
  show *foldl* (*echelon-form-of-column-k bezout*) (*A*, *0*) [*0*..<*Suc 0*] =
    (*fst* (*foldl* (*echelon-form-of-column-k bezout*) (*A*, *0*) [*0*..<*Suc 0*]),
    if ∀ *m*. *is-zero-row-upt-k m* (*Suc 0*) (*fst* (*foldl* (*echelon-form-of-column-k bezout*)

    (*A*, *0*) [*0*..<*Suc 0*])) *then 0 else to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n* (*Suc 0*)
    (*fst* (*foldl* (*echelon-form-of-column-k bezout*) (*A*, *0*) [*0*..<*Suc 0*]))) + *1*)
    unfolding *rw-upt*
    unfolding *foldl.simps*
    unfolding *echelon-form-of-column-k-def*
    unfolding *Let-def*
    unfolding *split-beta*
    unfolding *from-nat-0 fst-conv snd-conv*
    unfolding *is-zero-row-upt-k-def*
    apply (*auto simp add*: *least-mod-type to-nat-eq-0*)
    apply (*metis* (*mono-tags*, *lifting*) *GreatestI least-mod-type less-le*)
  proof −
    fix *m mb* **assume***A* $ *m* $ *0* ≠ *0*
      and *all-zero*: ∀ *m*. *bezout-iterate ?interchange* (*nrows A* − *Suc 0*) *0 0 bezout* $ *m* $ *0* = *0*
    have *bezout-iterate ?interchange* (*nrows A* − *Suc 0*) *0 0 bezout* $ *0* $ *0* =

bezout-iterate ?interchange (*nrows A* − *Suc 0*) *0* (*from-nat 0*) *bezout* \$ *0* \$ *from-nat 0*
using *from-nat-0* **by** *metis*
also have ... ≠ *0*
proof (*rule bezout-iterate-not-zero*[*OF* - - - *ib*], *simp-all add*: *nrows-def*)
show *A* \$ (*LEAST n. A* \$ *n* \$ *0* ≠ *0*) \$ *from-nat 0* ≠ *0*
by (*metis* (*mono-tags*) *LeastI* ‹*A* \$ *m* \$ *0* ≠ *0*› *from-nat-0*)
show *to-nat 0* ≤ *CARD*(′*rows*) − *Suc 0* **by** (*metis le0 to-nat-0*)
qed
finally have *bezout-iterate ?interchange* (*nrows A* − *Suc 0*) *0 0 bezout* \$ *0* \$ *0* ≠ *0* **.**
thus *A* \$ *mb* \$ *0* = *0* **using** *all-zero* **by** *auto*
next
fix *m* **assume** *Am0*: *A* \$ *m* \$ *0* ≠ *0*
and *all-zero*: ∀ *m*>*0*. *A* \$ *m* \$ *0* = *0* **thus** (*GREATEST n. A* \$ *n* \$ *0* ≠ *0*) = *0*
by (*metis* (*mono-tags*, *lifting*) *GreatestI neq-iff not-less0 to-nat-0 to-nat-mono*)
next
fix *m ma mb*
assume *A* \$ *m* \$ *0* ≠ *0* **and** *bezout-iterate* (*interchange-rows A 0* (*LEAST n. A* \$ *n* \$ *0* ≠ *0*))
(*nrows A* − *Suc 0*) *0 0 bezout* \$ *ma* \$ *0* ≠ *0*
have *bezout-iterate ?interchange* (*nrows A* − *Suc 0*) *0 0 bezout* \$ *0* \$ *0* =
bezout-iterate ?interchange (*nrows A* − *Suc 0*) *0* (*from-nat 0*) *bezout* \$ *0* \$ *from-nat 0*
using *from-nat-0* **by** *metis*
also have ... ≠ *0*
proof (*rule bezout-iterate-not-zero*[*OF* - - - *ib*], *simp-all add*: *nrows-def*)
show *A* \$ (*LEAST n. A* \$ *n* \$ *0* ≠ *0*) \$ *from-nat 0* ≠ *0*
by (*metis* (*mono-tags*) *LeastI* ‹*A* \$ *m* \$ *0* ≠ *0*› *from-nat-0*)
show *to-nat 0* ≤ *CARD*(′*rows*) − *Suc 0* **by** (*metis le0 to-nat-0*)
qed
finally have *1*: *bezout-iterate ?interchange* (*nrows A* − *Suc 0*) *0 0 bezout* \$ *0* \$ *0* ≠ *0* **.**
have *2*: ∀ *m*>*0*. *bezout-iterate ?interchange* (*nrows A* − *Suc 0*) *0 0 bezout* \$ *m* \$ *0* = *0*
proof (*auto*)
fix *b*::′*rows*
assume *b*: *0*<*b*
have *bezout-iterate ?interchange* (*nrows A* − *Suc 0*) *0 0 bezout* \$ *b* \$ *0*
= *bezout-iterate ?interchange* (*nrows A* − *Suc 0*) *0* (*from-nat 0*) *bezout* \$ *b* \$ *from-nat 0*
using *from-nat-0* **by** *metis*
also have ... = *0*
proof (*rule bezout-iterate-zero-column-k*[*OF* - *ib*])
show *echelon-form-upt-k* (*?interchange*) *0* **by** (*metis echelon-form-upt-k-0*)

show *?interchange* \$ *0* \$ *from-nat 0* ≠ *0*
by (*metis* (*mono-tags*, *lifting*) *LeastI-ex* ‹*A* \$ *m* \$ *0* ≠ *0*› *from-nat-0*

interchange-rows-i)
show *nrows A* − *Suc 0* < *nrows* (*?interchange*) **unfolding** *nrows-def* **by** *simp*
show *0* < *b* **using** *b* **.**
show *0* < *ncols* (*?interchange*) **unfolding** *ncols-def* **by** *auto*
show *to-nat b* ≤ *nrows A* − *Suc 0*
by (*simp add*: *nrows-def le-diff-conv2 Suc-le-eq to-nat-less-card*)
show *is-zero-row-upt-k 0 0* (*?interchange*) **by** (*metis is-zero-row-utp-0*)
qed
finally show *bezout-iterate* (*interchange-rows A 0* (*LEAST n. A* $ *n* $ *0* ≠ *0*))
(*nrows A* − *Suc 0*) *0 0 bezout* $ *b* $ *0* = *0* **.**
qed
show (*GREATEST n. bezout-iterate* (*interchange-rows A 0* (*LEAST n. A* $ *n* $ *0* ≠ *0*))
(*nrows A* − *Suc 0*) *0 0 bezout* $ *n* $ *0* ≠ *0*) = *0*
apply (*rule Greatest-equality*, *simp add*: *1*)
using *2* **by** *force*
qed
next
fix *k*
let *?fold*=(*foldl* (*echelon-form-of-column-k bezout*)(*A*, *0*) [*0*..<*Suc* (*Suc k*)])
let *?fold2*=(*foldl* (*echelon-form-of-column-k bezout*) (*A*, *0*) [*0*..<(*Suc k*)])
assume (*k* < *ncols A* ⟹ *echelon-form-upt-k* (*echelon-form-of-upt-k A k bezout*) (*Suc k*)) **and**
(*k* < *ncols A* ⟹ *foldl* (*echelon-form-of-column-k bezout*) (*A*, *0*) [*0*..<*Suc k*] =
(*fst ?fold2*, *if* ∀ *m. is-zero-row-upt-k m* (*Suc k*) (*fst ?fold2*) *then 0*
else to-nat (*GREATEST n.* ¬ *is-zero-row-upt-k n* (*Suc k*) (*fst ?fold2*)) + *1*))
and *Suc-k*: *Suc k* < *ncols A*
hence *hyp-foldl*: *foldl* (*echelon-form-of-column-k bezout*) (*A*, *0*) [*0*..<*Suc k*] =
(*fst ?fold2*, *if* ∀ *m. is-zero-row-upt-k m* (*Suc k*) (*fst ?fold2*) *then 0*
else to-nat (*GREATEST n.* ¬ *is-zero-row-upt-k n* (*Suc k*) (*fst ?fold2*)) + *1*)
and *hyp-echelon*: *echelon-form-upt-k* (*echelon-form-of-upt-k A k bezout*) (*Suc k*) **by** *auto*
have *rw*: [*0*..<*Suc* (*Suc k*)]= [*0*..<(*Suc k*)] @ [(*Suc k*)] **by** *auto*
have *rw2*: *?fold2* = (*echelon-form-of-upt-k A k bezout*, *if* ∀ *m. is-zero-row-upt-k m* (*Suc k*)
(*echelon-form-of-upt-k A k bezout*) *then 0 else*
to-nat (*GREATEST n.* ¬ *is-zero-row-upt-k n* (*Suc k*) (*echelon-form-of-upt-k A k bezout*)) + *1*)
unfolding *echelon-form-of-upt-k-def* **using** *hyp-foldl* **by** *fast*
show *echelon-form-upt-k* (*echelon-form-of-upt-k A* (*Suc k*) *bezout*) (*Suc* (*Suc k*))
unfolding *echelon-form-of-upt-k-def*
unfolding *rw* **unfolding** *foldl-append* **unfolding** *foldl.simps* **unfolding** *rw2*
proof (*rule echelon-echelon-form-column-k*[*OF ib hyp-echelon*])
show *Suc k* < *ncols* (*echelon-form-of-upt-k A k bezout*) **using** *Suc-k* **unfolding** *ncols-def* **.**
qed
show *foldl* (*echelon-form-of-column-k bezout*) (*A*, *0*) [*0*..<*Suc* (*Suc k*)] =

(*fst ?fold*,
if ∀ *m*. *is-zero-row-upt-k m* (*Suc* (*Suc k*))
(*fst ?fold*) *then 0*
else to-nat
(*GREATEST n*. ¬ *is-zero-row-upt-k n* (*Suc* (*Suc k*))
(*fst ?fold*)) + *1*)
proof (*rule prod-eqI*, *metis fst-conv*)
define *A′* **where** *A′* = *fst ?fold2*
let *?greatest*=(*GREATEST n*. ¬ *is-zero-row-upt-k n* (*Suc k*) *A′*)
have *k*: *k* < *ncols A′* **using** *Suc-k* **unfolding** *ncols-def* **by** *auto*
have *k2*: *Suc k* < *ncols A′* **using** *Suc-k* **unfolding** *ncols-def* **by** *auto*
have *fst-snd-foldl*: *snd ?fold2* = *snd* (*fst ?fold2*,
if ∀ *m*. *is-zero-row-upt-k m* (*Suc k*) (*fst ?fold2*) *then 0*
else to-nat (*GREATEST n*. ¬ *is-zero-row-upt-k n* (*Suc k*) (*fst ?fold2*)) + *1*)
using *hyp-foldl* **by** *simp*
have *ncols-eq*: *ncols A* = *ncols A′* **unfolding** *A′-def ncols-def* **..**
have *rref-A′*: *echelon-form-upt-k A′* (*Suc k*)
using *hyp-echelon* **unfolding** *A′-def echelon-form-of-upt-k-def* **.**
show *snd ?fold* = *snd* (*fst ?fold*, *if* ∀ *m*. *is-zero-row-upt-k m* (*Suc* (*Suc k*)) (*fst ?fold*) *then 0*
else to-nat (*GREATEST n*. ¬ *is-zero-row-upt-k n* (*Suc* (*Suc k*)) (*fst ?fold*)) + *1*)
using [[*unfold-abs-def* = *false*]]
unfolding *fst-conv snd-conv* **unfolding** *rw*
unfolding *foldl-append* **unfolding** *foldl.simps*
unfolding *echelon-form-of-column-k-def Let-def split-beta fst-snd-foldl*
unfolding *A′-def*[*symmetric*]
proof (*auto simp add*: *least-mod-type from-nat-0 from-nat-to-nat-greatest*)
fix *m* **assume** *A′* $ *m* $ *from-nat* (*Suc k*) ≠ *0*
thus ∃ *m*. ¬ *is-zero-row-upt-k m* (*Suc* (*Suc k*)) *A′*
and ∃ *m*. ¬ *is-zero-row-upt-k m* (*Suc* (*Suc k*)) *A′*
and ∃ *m*. ¬ *is-zero-row-upt-k m* (*Suc* (*Suc k*)) *A′*
and ∃ *m*. ¬ *is-zero-row-upt-k m* (*Suc* (*Suc k*)) *A′*
and ∃ *m*. ¬ *is-zero-row-upt-k m* (*Suc* (*Suc k*)) *A′*
and ∃ *m*. ¬ *is-zero-row-upt-k m* (*Suc* (*Suc k*)) *A′*
and ∃ *m*. ¬ *is-zero-row-upt-k m* (*Suc* (*Suc k*)) *A′*
and ∃ *m*. ¬ *is-zero-row-upt-k m* (*Suc* (*Suc k*)) *A′*
unfolding *is-zero-row-upt-k-def*
by (*metis add-to-nat-def from-nat-mono less-irrefl*
monoid-add-class.add.right-neutral not-less-eq to-nat-0 to-nat-less-card)+
next
fix *m*
assume ¬ *is-zero-row-upt-k m* (*Suc k*) *A′*
thus ∃ *m*. ¬ *is-zero-row-upt-k m* (*Suc* (*Suc k*)) *A′*
and ∃ *m*. ¬ *is-zero-row-upt-k m* (*Suc* (*Suc k*)) *A′*
by (*metis is-zero-row-upt-k-le*)+
next
fix *m*
assume ∀ *ma*. *is-zero-row-upt-k ma* (*Suc k*) *A′* **and** ∀ *mb*. *A′* $ *mb* $ *from-nat*

(*Suc k*) = *0*
    thus *is-zero-row-upt-k m* (*Suc* (*Suc k*)) *A′*
      by (*metis is-zero-row-upt-k-suc*)
  next
    fix *ma*
    assume ∀ *m*>*0*. *A′* $ *m* $ *from-nat* (*Suc k*) = *0*
      and ∀ *m*. *is-zero-row-upt-k m* (*Suc k*) *A′*
      and ¬ *is-zero-row-upt-k ma* (*Suc* (*Suc k*)) *A′*
    thus *to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n* (*Suc* (*Suc k*)) *A′*) = *0*
      and *to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n* (*Suc* (*Suc k*)) *A′*) = *0*
      by (*metis* (*erased*, *lifting*) *GreatestI-ex le-less*
        is-zero-row-upt-k-suc least-mod-type to-nat-0)+
  next
    fix *m*
    assume ∀ *m*>*0*. *A′* $ *m* $ *from-nat* (*Suc k*) = *0*
      and ¬ *is-zero-row-upt-k m* (*Suc k*) *A′*
      and ∀ *m*≥*?greatest*+*1*. *A′* $ *m* $ *from-nat* (*Suc k*) = *0*
    thus *?greatest*
      = (*GREATEST n*. ¬ *is-zero-row-upt-k n* (*Suc* (*Suc k*)) *A′*)
      by (*metis* (*mono-tags*, *lifting*) *echelon-form-upt-k-condition1 from-nat-0*
        is-zero-row-upt-k-le is-zero-row-upt-k-suc less-nat-zero-code neq-iff rref-A′
to-nat-le)
  next
    fix *m ma*
    assume ∀ *m*>*?greatest*+*1*.
      A′ $ *m* $ *from-nat* (*Suc k*) = *0*
      and ∀ *m*>*0*. *A′* $ *m* $ *from-nat* (*Suc k*) = *0*
      and*Suc* (*to-nat ?greatest*) ≠ *nrows A′*
      and *?greatest* + *1* ≤ *ma*
      and *A′* $ *ma* $ *from-nat* (*Suc k*) ≠ *0*
    thus *Suc* (*to-nat ?greatest*) = *to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n*
(*Suc* (*Suc k*)) *A′*)
      by (*metis* (*mono-tags*) *Suc-eq-plus1 less-linear*
        leD least-mod-type nrows-def suc-not-zero)
  next
    fix *m ma*
    assume ∀ *m*>*?greatest* + *1*. *A′* $ *m* $ *from-nat* (*Suc k*) = *0*
      and ∀ *m*>*0*. *A′* $ *m* $ *from-nat* (*Suc k*) = *0*
      and ¬ *is-zero-row-upt-k m* (*Suc k*) *A′*
      and *Suc* (*to-nat ?greatest*) = *nrows A′*
      and ¬ *is-zero-row-upt-k ma* (*Suc* (*Suc k*)) *A′*
    thus *nrows A′* = *Suc* (*to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n* (*Suc*
(*Suc k*)) *A′*))
      by (*metis echelon-foldl-condition5*)
  next
    fix *ma* **assume** *1*: *A′* $ *ma* $ *from-nat* (*Suc k*) ≠ *0*
    show ∃ *m*. ¬ *is-zero-row-upt-k m* (*Suc* (*Suc k*))
      (*bezout-iterate* (*interchange-rows A′ 0* (*LEAST n*. *A′* $ *n* $ *from-nat* (*Suc*
k) ≠ *0*))

$(nrows\ A' - Suc\ 0)\ 0\ (from\text{-}nat\ (Suc\ k))\ bezout)$
and $\exists\, m.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ (Suc\ (Suc\ k))$
$(bezout\text{-}iterate\ (interchange\text{-}rows\ A'\ 0\ (LEAST\ n.\ A'\ \$\ n\ \$\ from\text{-}nat\ (Suc\ k) \neq 0))$
$(nrows\ A' - Suc\ 0)\ 0\ (from\text{-}nat\ (Suc\ k))\ bezout)$
by $(rule\ echelon\text{-}foldl\text{-}condition1[OF\ ib\ 1\ k])+$
next
fix $m\ ma\ mb$
assume 1: $\neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ ma\ (Suc\ k)\ A'$
and 2:$\forall\, m \geq ?greatest + 1.\ A'\ \$\ m\ \$\ from\text{-}nat\ (Suc\ k) = 0$
show $?greatest$
$= (GREATEST\ n.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ n\ (Suc\ (Suc\ k))\ A')$
by $(rule\ echelon\text{-}foldl\text{-}condition2[OF\ 1\ 2])$
next
fix m
assume 1: $A'\ \$\ m\ \$\ from\text{-}nat\ (Suc\ k) \neq 0$
and 2: $\forall\, m.\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ (Suc\ k)\ A'$
show $to\text{-}nat\ (GREATEST\ n.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ n\ (Suc\ (Suc\ k))$
$(bezout\text{-}iterate\ (interchange\text{-}rows\ A'\ 0\ (LEAST\ n.\ A'\ \$\ n\ \$\ from\text{-}nat\ (Suc\ k) \neq 0))$
$(nrows\ A' - Suc\ 0)\ 0\ (from\text{-}nat\ (Suc\ k))\ bezout)) = 0$
and $to\text{-}nat\ (GREATEST\ n.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ n\ (Suc\ (Suc\ k))$
$(bezout\text{-}iterate\ (interchange\text{-}rows\ A'\ 0\ (LEAST\ n.\ A'\ \$\ n\ \$\ from\text{-}nat\ (Suc\ k) \neq 0))$
$(nrows\ A' - Suc\ 0)\ 0\ (from\text{-}nat\ (Suc\ k))\ bezout)) = 0$
by $(rule\ echelon\text{-}foldl\text{-}condition3[OF\ ib\ 1\ 2\ rref\text{-}A'],\ metis\ ncols\text{-}def\ Suc\text{-}k)+$
next
fix m **assume** $\forall\, m > ?greatest + 1.$
$A'\ \$\ m\ \$\ from\text{-}nat\ (Suc\ k) = 0$
and $0 < m$
and $A'\ \$\ m\ \$\ from\text{-}nat\ (Suc\ k) \neq 0$
and $Suc\ (to\text{-}nat\ ?greatest) = nrows\ A'$
thus $nrows\ A' = Suc\ (to\text{-}nat\ (GREATEST\ n.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ n\ (Suc\ (Suc\ k))\ A'))$
by $(metis\ (mono\text{-}tags)\ Suc\text{-}eq\text{-}plus1\ Suc\text{-}le'\ from\text{-}nat\text{-}suc$
$from\text{-}nat\text{-}to\text{-}nat\text{-}id\ not\text{-}less\text{-}eq\ nrows\text{-}def\ to\text{-}nat\text{-}less\text{-}card\ to\text{-}nat\text{-}mono)$
next
fix mb
assume 1: $\forall\, m > ?greatest+1.$
$A'\ \$\ m\ \$\ from\text{-}nat\ (Suc\ k) = 0$
and 2: $Suc\ (to\text{-}nat\ ?greatest) \neq nrows\ A'$
and 3: $?greatest+1 \leq mb$
and 4: $A'\ \$\ mb\ \$\ from\text{-}nat\ (Suc\ k) \neq 0$
show $Suc\ (to\text{-}nat\ ?greatest) =$
$to\text{-}nat\ (GREATEST\ n.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ n\ (Suc\ (Suc\ k))\ A')$
by $(rule\ echelon\text{-}foldl\text{-}condition4[OF\ 1\ 2\ 3\ 4])$
next
fix m
assume $?greatest + 1 < m$

and A' \$ m \$ *from-nat* (*Suc k*) $\neq$ *0*
and $\forall m>0$. A' \$ m \$ *from-nat* (*Suc k*) = *0*
thus *nrows* A' = *Suc* (*to-nat* (*GREATEST n.* $\neg$ *is-zero-row-upt-k n* (*Suc* (*Suc k*)) A'))
by (*metis le-less-trans least-mod-type*)
next
fix *m*
assume *?greatest* + *1* < *m*
and A' \$ m \$ *from-nat* (*Suc k*) $\neq$ *0*
and $\forall m>0$. A' \$ m \$ *from-nat* (*Suc k*) = *0*
thus $\exists$ *m.* $\neg$ *is-zero-row-upt-k m* (*Suc* (*Suc k*)) (*bezout-iterate*
(*interchange-rows* A' (*?greatest* + *1*) (*LEAST n.* A' \$ *n* \$ *from-nat* (*Suc* *k*) $\neq$ *0*
$\wedge$ *?greatest* + *1* $\leq$ *n*)) (*nrows* A' $-$ *Suc 0*) (*?greatest* + *1*) (*from-nat* (*Suc* *k*)) *bezout*)
by (*metis le-less-trans least-mod-type*)
next
fix *mb*
assume $\neg$ *is-zero-row-upt-k mb* (*Suc k*) A'
and *Suc* (*to-nat ?greatest*) = *nrows* A'
thus *nrows* A' = *Suc* (*to-nat* (*GREATEST n.* $\neg$ *is-zero-row-upt-k n* (*Suc* (*Suc k*)) A'))
by (*rule echelon-foldl-condition5*)
next
fix *m*
assume (*GREATEST n.* $\neg$ *is-zero-row-upt-k n* (*Suc k*) A') + *1* < *m*
and A' \$ m \$ *from-nat* (*Suc k*) $\neq$ *0*
and $\forall m>0$. A' \$ m \$ *from-nat* (*Suc k*) = *0*
thus *Suc* (*to-nat ?greatest*) =
to-nat (*GREATEST n.* $\neg$ *is-zero-row-upt-k n* (*Suc* (*Suc k*))
(*bezout-iterate* (*interchange-rows* A' (*?greatest* + *1*)
(*LEAST n.* A' \$ *n* \$ *from-nat* (*Suc k*) $\neq$ *0* $\wedge$ *?greatest* + *1* $\leq$ *n*))
(*nrows* A' $-$ *Suc 0*) (*?greatest* + *1*) (*from-nat* (*Suc k*)) *bezout*))
by (*metis le-less-trans least-mod-type*)
next
fix *mc*
assume *?greatest* + *1* $\leq$ *mc*
and A' \$ *mc* \$ *from-nat* (*Suc k*) $\neq$ *0*
thus $\exists$ *m.* $\neg$ *is-zero-row-upt-k m* (*Suc* (*Suc k*))
(*bezout-iterate* (*interchange-rows* A' (*?greatest* + *1*)
(*LEAST n.* A' \$ *n* \$ *from-nat* (*Suc k*) $\neq$ *0* $\wedge$ *?greatest* + *1* $\leq$ *n*))
(*nrows* A' $-$ *Suc 0*) (*?greatest* + *1*) (*from-nat* (*Suc k*)) *bezout*)
using *echelon-foldl-condition6*[*OF ib*] **by** *blast*
next
fix *mb mc*
assume *1*: $\neg$ *is-zero-row-upt-k mb* (*Suc k*) A'
and *2*: *Suc* (*to-nat ?greatest*) $\neq$ *nrows* A'
and *3*: *?greatest* + *1* $\leq$ *mc*
and *4*:A' \$ *mc* \$ *from-nat* (*Suc k*) $\neq$ *0*

show *Suc* (*to-nat ?greatest*) = *to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n* (*Suc* (*Suc k*))
(*bezout-iterate* (*interchange-rows A′* (*?greatest* + *1*)
(*LEAST n*. *A′* $ *n* $ *from-nat* (*Suc k*) ≠ *0* ∧ *?greatest* + *1* ≤ *n*))
(*nrows A′* − *Suc 0*) (*?greatest* + *1*) (*from-nat* (*Suc k*)) *bezout*))
by (*rule echelon-foldl-condition7*[*OF ib rref-A′ k2 1 2 3 4*])
qed
qed
qed

3.2.7 Proving the existence of invertible matrices which do the transformations

lemma *bezout-iterate-invertible*:
fixes *A*::′*a*::{*bezout-domain*}^′*cols*^′*rows*::{*mod-type*}
assumes *ib*: *is-bezout-ext bezout*
assumes *n*<*nrows A*
and *to-nat i*≤*n*
and *A* $ *i* $ *j* ≠ *0*
shows ∃ *P*. *invertible P* ∧ *P*∗∗*A* = *bezout-iterate A n i j bezout*
using *assms*
proof (*induct n arbitrary*: *A*)
case *0*
show *?case*
unfolding *bezout-iterate.simps*
by (*simp add*: *exI*[*of - mat 1*] *matrix-mul-lid invertible-def*)
next
case (*Suc n*)
show *?case*
proof (*cases Suc n* = *to-nat i*)
case *True* **show** *?thesis* **unfolding** *bezout-iterate.simps* **using** *True Suc.prems*(*1*)

by (*simp add*: *exI*[*of - mat 1*] *matrix-mul-lid invertible-def*)
next
case *False*
have *i-le-n*: *to-nat i* < *Suc n* **using** *Suc.prems*(*3*) *False* **by** *auto*
let *?B*=(*bezout-matrix A i* (*from-nat* (*Suc n*)) *j bezout* ∗∗ *A*)
have *b*: *bezout-iterate A* (*Suc n*) *i j bezout* = *bezout-iterate ?B n i j bezout*
unfolding *bezout-iterate.simps* **using** *i-le-n* **by** *auto*
have ∃ *P*. *invertible P* ∧ *P*∗∗*?B* = *bezout-iterate ?B n i j bezout*
proof (*rule Suc.hyps*[*OF ib -*])
show *n* < *nrows ?B* **using** *Suc.prems* (*2*) **unfolding** *nrows-def* **by** *simp*
show *to-nat i* ≤ *n* **using** *i-le-n* **by** *auto*
show *?B* $ *i* $ *j* ≠ *0*
by (*metis False Suc.prems*(*2*) *Suc.prems*(*4*) *bezout-matrix-not-zero*
ib nrows-def to-nat-from-nat-id)
qed
from *this* **obtain** *P* **where** *inv-P*: *invertible P* **and** *P*: *P*∗∗*?B* = *bezout-iterate ?B n i j bezout*

by *blast*
show *?thesis*
proof (*rule exI*[*of - P* ∗∗ *bezout-matrix A i* (*from-nat* (*Suc n*)) *j bezout*],
rule conjI, *rule invertible-mult*)
show *P* ∗∗ *bezout-matrix A i* (*from-nat* (*Suc n*)) *j bezout* ∗∗ *A*
= *bezout-iterate A* (*Suc n*) *i j bezout* **using** *P* **unfolding** *b* **by** (*metis matrix-mul-assoc*)
have *det* (*bezout-matrix A i* (*from-nat* (*Suc n*)) *j bezout*) = *1*
proof (*rule det-bezout-matrix*[*OF ib*])
show *i* < *from-nat* (*Suc n*)
using *i-le-n from-nat-mono*[*of to-nat i Suc n*] *Suc.prems*(*2*)
unfolding *nrows-def* **by** (*metis from-nat-to-nat-id*)
show *A* $ *i* $ *j* ≠ *0* **by** (*rule Suc.prems*(*4*))
qed
thus *invertible* (*bezout-matrix A i* (*mod-type-class.from-nat* (*Suc n*)) *j bezout*)
unfolding *invertible-iff-is-unit* **by** *simp*
show *invertible P* **using** *inv-P* **.**
qed
qed
qed

lemma *echelon-form-of-column-k-invertible*:
fixes *A*::′*a*::{*bezout-domain*}^′*cols*::{*mod-type*}^′*rows*::{*mod-type*}
assumes *ib*: *is-bezout-ext bezout*
shows ∃ *P*. *invertible P* ∧ *P* ∗∗ *A* = *fst* ((*echelon-form-of-column-k bezout*) (*A*,*i*) *k*)
proof −
have ∃ *P*. *invertible P* ∧ *P* ∗∗ *A* = *A*
by (*simp add*: *exI*[*of - mat 1*] *matrix-mul-lid invertible-def*)
thus *?thesis*
proof (*unfold echelon-form-of-column-k-def Let-def*, *auto*)
fix *P m ma*
let *?least* = (*LEAST n*. *A* $ *n* $ *from-nat k* ≠ *0* ∧ *from-nat i* ≤ *n*)
let *?interchange* =(*interchange-rows A* (*from-nat i*) *?least*)
assume *i*: *i* ≠ *nrows A*
and *i2*: *mod-type-class.from-nat i* ≤ *ma*
and *ma*: *A* $ *ma* $ *mod-type-class.from-nat k* ≠ *0*
have ∃ *P*. *invertible P* ∧
P ∗∗ *?interchange* =
bezout-iterate ?interchange (*nrows A* − *Suc 0*) (*from-nat i*) (*from-nat k*) *bezout*
proof (*rule bezout-iterate-invertible*[*OF ib*])
show *nrows A* − *Suc 0* < *nrows ?interchange* **unfolding** *nrows-def* **by** *simp*
show *to-nat* (*from-nat i*::′*rows*) ≤ *nrows A* − *Suc 0*
by (*metis Suc-leI Suc-le-mono Suc-pred nrows-def to-nat-less-card zero-less-card-finite*)
show *?interchange* $ *from-nat i* $ *from-nat k* ≠ *0*
by (*metis* (*mono-tags*, *lifting*) *LeastI-ex i2 ma interchange-rows-i*)
qed
from *this* **obtain** *P* **where** *inv-P*: *invertible P* **and** *P*: *P* ∗∗ *?interchange* =
bezout-iterate ?interchange (*nrows A* − *Suc 0*) (*from-nat i*) (*from-nat k*) *bezout*

by *blast*
show $\exists$ *P*. *invertible P* $\wedge$ *P* ** *A*
= *bezout-iterate ?interchange* (*nrows A* − *Suc 0*) (*from-nat i*) (*from-nat k*) *bezout*
proof (*rule exI*[*of - P* ** *interchange-rows* (*mat 1*) (*from-nat i*) *?least*],
rule conjI, *rule invertible-mult*)
show *P* ** *interchange-rows* (*mat 1*) (*from-nat i*) *?least* ** *A* =
bezout-iterate ?interchange (*nrows A* − *Suc 0*) (*from-nat i*) (*from-nat k*) *bezout*
using *P* **by** (*metis* (*no-types*, *lifting*) *interchange-rows-mat-1 matrix-mul-assoc*)

show *invertible P* **by** (*rule inv-P*)
show *invertible* (*interchange-rows* (*mat 1*) (*from-nat i*) *?least*)
by (*simp add*: *invertible-interchange-rows*)
qed
qed
qed

lemma *echelon-form-of-upt-k-invertible*:
fixes *A*::*'a*::{*bezout-domain*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *ib*: *is-bezout-ext bezout*
shows $\exists$ *P*. *invertible P* $\wedge$ *P***A* = (*echelon-form-of-upt-k A k bezout*)
proof (*induct k*)
case *0*
show *?case*
unfolding *echelon-form-of-upt-k-def*
by (*simp add*: *echelon-form-of-column-k-invertible*[*OF ib*])
next
case (*Suc k*)
have *set-rw*: [*0..<Suc* (*Suc k*)] = [*0..<Suc k*] @ [*Suc k*] **by** *simp*
let *?foldl* = *foldl* (*echelon-form-of-column-k bezout*) (*A*, *0*) [*0..<Suc k*]
obtain *P* **where** *invP*: *invertible P*
and *P*: *P* ** *A* = *fst ?foldl*
using *Suc.hyps* **unfolding** *echelon-form-of-upt-k-def* **by** *auto*
obtain *Q* **where** *invQ*: *invertible Q* **and** *Q*:
Q ** *fst ?foldl* = *fst* ((*echelon-form-of-column-k bezout*) (*fst ?foldl*, *snd ?foldl*) (*Suc k*))
using *echelon-form-of-column-k-invertible* [*OF ib*] **by** *blast*
show *?case*
proof (*rule exI*[*of - Q**P*], *rule conjI*)
show *invertible* (*Q**P*) **by** (*metis invP invQ invertible-mult*)
show *Q* ** *P* ** *A* = *echelon-form-of-upt-k A* (*Suc k*) *bezout*
unfolding *echelon-form-of-upt-k-def*
unfolding *set-rw* **unfolding** *foldl-append* **unfolding** *foldl.simps*
unfolding *matrix-mul-assoc*[*symmetric*]
unfolding *P Q* **by** *auto*
qed
qed

3.2.8 Final results

lemma *echelon-form-echelon-form-of*:
fixes A::$'a$::{*bezout-domain*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
assumes *ib*: *is-bezout-ext bezout*
shows *echelon-form* (*echelon-form-of A bezout*)
proof −
have *n*: *ncols A* − *1* < *ncols A* **unfolding** *ncols-def* **by** *auto*
show *?thesis*
unfolding *echelon-form-def echelon-form-of-def*
using *echelon-echelon-form-of-upt-k*[*OF n ib*]
unfolding *ncols-def* **by** *simp*
qed

lemma *echelon-form-of-invertible*:
fixes A::$'a$::{*bezout-domain*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
assumes *ib*: *is-bezout-ext* (*bezout*)
shows ∃ *P*. *invertible P*
∧ *P* ∗∗ *A* = (*echelon-form-of A bezout*)
∧ *echelon-form* (*echelon-form-of A bezout*)
using *echelon-form-of-upt-k-invertible*[*OF ib*] *echelon-form-echelon-form-of*[*OF ib*]
unfolding *echelon-form-of-def* **by** *fast*

Executable version

corollary *echelon-form-echelon-form-of-euclidean*:
fixes A::$'a$::{*euclidean-ring-gcd*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
shows *echelon-form* (*echelon-form-of-euclidean A*)
using *echelon-form-echelon-form-of is-bezout-ext-euclid-ext2*
unfolding *echelon-form-of-euclidean-def*
by *auto*

corollary *echelon-form-of-euclidean-invertible*:
fixes A::$'a$::{*euclidean-ring-gcd*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
shows ∃ *P*. *invertible P* ∧ *P*∗∗*A* = (*echelon-form-of A euclid-ext2*)
∧ *echelon-form* (*echelon-form-of A euclid-ext2*)
using *echelon-form-of-invertible*[*OF is-bezout-ext-euclid-ext2*] **.**

3.3 More efficient code equations

definition
echelon-form-of-column-k-efficient bezout A′ k =
(*let* (*A*, *i*) = *A′*;
from-nat-k = *from-nat k*;
from-nat-i = *from-nat i*;
all-zero-below-i = (∀ *m*>*from-nat-i*. *A* \$ *m* \$ *from-nat-k* = *0*)
in if (*i* = *nrows A*) ∨ (*A* \$ *from-nat-i* \$ *from-nat-k* = *0*) ∧ *all-zero-below-i* *then* (*A*, *i*)
else if all-zero-below-i then (*A*, *i* + *1*)
else

let n = (*LEAST n. A* \$ *n* \$ *from-nat-k* ≠ *0* ∧ *from-nat-i* ≤ *n*);
interchange-A = *interchange-rows A* (*from-nat-i*) *n*
in
(*bezout-iterate* (*interchange-A*) (*nrows A* − *1*) (*from-nat-i*) (*from-nat-k*) *bezout*, *i* + *1*))

lemma *echelon-form-of-column-k-efficient*[*code*]:
(*echelon-form-of-column-k bezout*) (*A*,*i*) *k*
= (*echelon-form-of-column-k-efficient bezout*) (*A*,*i*) *k*
unfolding *echelon-form-of-column-k-def echelon-form-of-column-k-efficient-def*
unfolding *Let-def* **by** *force*

end

4 Determinant of matrices over principal ideal rings

theory *Echelon-Form-Det*
imports *Echelon-Form*
begin

4.1 Definitions

The following definition can be improved in terms of performance, because it checks if there exists an element different from zero twice.

definition
echelon-form-of-column-k-det :: (′*b* ⇒ ′*b* ⇒ ′*b* × ′*b* × ′*b* × ′*b* × ′*b*)
⇒ ′*b*::{*bezout-domain*}
× ((′*b*, ′*c*::{*mod-type*}) *vec*, ′*d*::{*mod-type*}) *vec*
× *nat*
⇒ *nat* ⇒ ′*b*
× ((′*b*, ′*c*) *vec*, ′*d*) *vec*
× *nat*
where
echelon-form-of-column-k-det bezout A′ *k* =
(*let* (*det-P*, *A*, *i*) = *A*′;
from-nat-i = *from-nat i*;
from-nat-k = *from-nat k*
in
if ((*i* ≠ *nrows A*) ∧
(*A* \$ *from-nat-i* \$ *from-nat-k* = *0*) ∧
(∃ *m*>*from-nat i*. *A* \$ *m* \$ *from-nat k* ≠ *0*))
then (−*1* ∗ *det-P*, (*echelon-form-of-column-k bezout*) (*A*, *i*) *k*)
else (*det-P*, (*echelon-form-of-column-k bezout*) (*A*, *i*) *k*))

definition
  echelon-form-of-upt-k-det bezout A′ k =
    (*let A* = (*snd A′*);
        f = (*foldl* (*echelon-form-of-column-k-det bezout*) (*1*, *A*, *0*) [*0*..<*Suc k*])
     in (*fst f*, *fst* (*snd f*)))

definition
  echelon-form-of-det :: ′*a*::{*bezout-domain*}^′*n*::{*mod-type*}^′*n*::{*mod-type*}
      ⇒ (′*a* ⇒ ′*a* ⇒ ′*a* × ′*a* × ′*a* × ′*a* × ′*a*)
      ⇒ (′*a* × (′*a*::{*bezout-domain*}^′*n*::{*mod-type*}^′*n*::{*mod-type*}))
  where
  echelon-form-of-det A bezout = *echelon-form-of-upt-k-det bezout* (*1*::′*a*,*A*) (*ncols A* − *1*)

4.2 Properties

4.2.1 Bezout Iterate

lemma *det-bezout-iterate*:
  fixes *A*::′*a*::{*bezout-domain*}^′*n*::{*mod-type*}^′*n*::{*mod-type*}
  assumes *ib*: *is-bezout-ext bezout*
  and *Aik*: *A* $ *i* $ *from-nat k* ≠ *0*
  and *n*: *n*<*ncols A*
  shows *det* (*bezout-iterate A n i* (*from-nat k*) *bezout*) = *det A*
  using *Aik n*
proof (*induct n arbitrary*: *A*)
  case *0*
  show *?case* **unfolding** *bezout-iterate.simps* **..**
next
  case (*Suc n*)
  show *?case*
  proof (*cases Suc n* ≤ *to-nat i*)
    case *True* **thus** *?thesis* **unfolding** *bezout-iterate.simps* **by** *simp*
  next
    let *?B* = *bezout-matrix A i* (*from-nat* (*Suc n*)) (*from-nat k*) *bezout*
    let *?A*=(*?B* ∗∗ *A*)
    case *False*
    hence (*bezout-iterate A* (*Suc n*) *i* (*mod-type-class.from-nat k*) *bezout*)
      = *bezout-iterate ?A n i* (*mod-type-class.from-nat k*) *bezout*
      unfolding *bezout-iterate.simps* **by** *auto*
    also have *det* (...) = *det ?A*
    proof (*rule Suc.hyps*, *rule bezout-matrix-not-zero*[*OF ib* - *Suc.prems*(*1*)])
      show *n* < *ncols ?A* **using** *Suc.prems*(*2*) **unfolding** *ncols-def* **by** *simp*
      show *i* ≠ *from-nat* (*Suc n*) **using** *False*
        by (*metis Suc.prems*(*2*) *eq-imp-le ncols-def to-nat-from-nat-id*)
    qed
    also have ... = *det A*
    proof −
      have *det ?B* = *1*
      proof (*rule det-bezout-matrix*[*OF ib* - *Suc.prems*(*1*)])

have *from-nat* (*to-nat* *i*) $<$ (*from-nat* (*Suc* *n*)::$'n$)
proof (*rule* *from-nat-mono*)
show *to-nat* *i* $<$ *Suc* *n* **using** *False* **by** *simp*
show *Suc* *n* $<$ *CARD*($'n$) **using** *Suc.prems*(*2*) **unfolding** *ncols-def* **.**
qed
thus *i* $<$ *mod-type-class.from-nat* (*Suc* *n*) **unfolding** *from-nat-to-nat-id* **.**
qed
thus *?thesis* **unfolding** *det-mul* **by** *auto*
qed
finally show *?thesis* **.**
qed
qed

4.2.2 Echelon Form of column k

lemma *det-echelon-form-of-column-k-det*:
fixes *A*::$'a$::{*bezout-domain*}$^{\frown}'n$::{*mod-type*}$^{\frown}'n$::{*mod-type*}
assumes *ib*: *is-bezout-ext bezout*
and *det*: *det-P* $*$ *det B* $=$ *det A*
shows *fst* ((*echelon-form-of-column-k-det bezout*) (*det-P*,*A*,*i*) *k*) $*$ *det B*
$=$ *det* (*fst* (*snd* ((*echelon-form-of-column-k-det bezout*) (*det-P*,*A*,*i*) *k*)))
proof $-$
let *?interchange*=(*interchange-rows A* (*from-nat i*)
(*LEAST n. A* \$ *n* \$ *from-nat k* $\neq$ *0* $\wedge$ *from-nat i* $\leq$ *n*))
let *?B*=(*bezout-iterate ?interchange* (*nrows A* $-$ *Suc 0*) (*from-nat i*) (*from-nat k*) *bezout*)
show *?thesis*
proof (*unfold echelon-form-of-column-k-det-def Let-def echelon-form-of-column-k-def*,

auto simp add: *assms*)
fix *m*
assume *i*: *from-nat i* $<$ *m*
and *Amk*: *A* \$ *m* \$ *from-nat k* $\neq$ *0*
and *i-not-nrows*: *i* $\neq$ *nrows A*
and *Aik*: *A* \$ *from-nat i* \$ *from-nat k* $=$ *0*
have *det ?B* $=$ *det ?interchange*
proof (*rule det-bezout-iterate*[*OF ib*])
show *?interchange* \$ *from-nat i* \$ *from-nat k* $\neq$ *0*
by (*metis* (*mono-tags*, *lifting*) *Amk LeastI-ex*
dual-order.strict-iff-order i interchange-rows-i)
show *nrows A* $-$ *Suc 0* $<$ *ncols ?interchange* **unfolding** *nrows-def ncols-def*
by *simp*
qed
also have ... $= -$ *det A*
proof (*rule det-interchange-different-rows*, *rule ccontr*, *simp*)
assume *i-least*: *from-nat i* $=$ (*LEAST n. A* \$ *n* \$ *from-nat k* $\neq$ *0* $\wedge$ *from-nat i* $\leq$ *n*)
have *A* \$ *from-nat i* \$ *from-nat k* $\neq$ *0*
by (*metis* (*poly-guards-query*, *lifting*) *Amk LeastI-ex*

linear i i-least leD)
thus *False* **using** *Aik* **by** *contradiction*
qed
finally show − *det A* = *det ?B* **by** *simp*
next
assume *i*: *i* ≠ *nrows A*
and *Aik*: *A* $ *from-nat i* $ *from-nat k* ≠ *0*
have *det ?B* = *det ?interchange*
proof (*rule det-bezout-iterate*[*OF ib*])
show *?interchange* $ *from-nat i* $ *from-nat k* ≠ *0*
by (*metis* (*mono-tags*, *lifting*) *Aik LeastI order-refl interchange-rows-i*)
show *nrows A* − *Suc 0* < *ncols ?interchange* **unfolding** *nrows-def ncols-def* **by** *simp*
qed
also have ... = *det A*
by (*rule det-interchange-same-rows*, *rule Least-equality*[*symmetric*], *auto simp add*: *Aik*)
finally show *det A* = *det ?B* **..**
qed
qed

lemma *snd-echelon-form-of-column-k-det-eq*:
shows *snd* ((*echelon-form-of-column-k-det bezout*) (*n*, *A*, *i*) *k*)
= (*echelon-form-of-column-k bezout*) (*A*,*i*) *k*
unfolding *echelon-form-of-column-k-det-def echelon-form-of-column-k-def Let-def snd-conv fst-conv*
by *auto*

4.2.3 Echelon form up to column k

lemma *snd-foldl-ef-det-eq*: *snd* (*foldl* (*echelon-form-of-column-k-det bezout*) (*n*, *A*, *0*) [*0*..<*k*])
= *foldl* (*echelon-form-of-column-k bezout*) (*A*, *0*) [*0*..<*k*]
proof (*induct k*)
case *0*
show *?case*
by (*simp add*: *echelon-form-of-column-k-det-def Let-def*)
next
case (*Suc k*)
have *Suc-rw*: [*0*..<(*Suc k*)] = [*0*..<*k*] @ [*k*] **by** *simp*
show *?case*
unfolding *Suc-rw foldl-append List.foldl.simps fst-conv snd-conv*
using *Suc.hyps*[*unfolded echelon-form-of-upt-k-det-def Let-def snd-conv*, *simplified*]
by (*metis prod.collapse snd-echelon-form-of-column-k-det-eq*)
qed

lemma *snd-echelon-form-of-upt-k-det-eq*:
shows *snd* ((*echelon-form-of-upt-k-det bezout*) (*n*, *A*) *k*) = *echelon-form-of-upt-k A k bezout*
unfolding *echelon-form-of-upt-k-det-def echelon-form-of-upt-k-def Let-def fst-conv snd-conv*
unfolding *snd-foldl-ef-det-eq* **by** *auto*

lemma *det-echelon-form-of-upt-k-det*:
fixes *A*::′*a*::{*bezout-domain*}^′*n*::{*mod-type*}^′*n*::{*mod-type*}
assumes *ib*: *is-bezout-ext bezout*
shows *fst* ((*echelon-form-of-upt-k-det bezout*) (*1*::′*a*,*A*) *k*) ∗ *det A*
= *det* (*snd* ((*echelon-form-of-upt-k-det bezout*) (*1*::′*a*,*A*) *k*))
proof (*induct k*)
case *0*
show *?case*
unfolding *echelon-form-of-upt-k-det-def Let-def*
by (*auto*, *rule det-echelon-form-of-column-k-det*[*OF ib*], *simp*)
next
case (*Suc k*)
let *?f* = *foldl* (*echelon-form-of-column-k-det bezout*) (*1*,*A*,*0*) [*0*..<*Suc k*]
have *Suc-rw*: [*0*..<*Suc* (*Suc k*)] = [*0*..<(*Suc k*)] @ [*Suc k*] **by** *simp*
have *fold-expand*: *?f* = (*fst ?f*, *fst* (*snd ?f*), *snd* (*snd ?f*)) **by** *simp*
show *?case* **unfolding** *echelon-form-of-upt-k-det-def Let-def*
unfolding *Suc-rw foldl-append List.foldl.simps fst-conv snd-conv*
apply (*subst* (*1 2*) *fold-expand*)
apply (*rule det-echelon-form-of-column-k-det*)
apply (*rule ib*)
apply (*subst* (*1*) *snd-foldl-ef-det-eq*)
by (*metis Suc.hyps echelon-form-of-upt-k-det-def fst-conv snd-conv snd-foldl-ef-det-eq*)
qed

4.2.4 Echelon form

lemma *det-echelon-form-of-det*:
fixes *A*::′*a*::{*bezout-domain*}^′*n*::{*mod-type*}^′*n*::{*mod-type*}
assumes *ib*: *is-bezout-ext bezout*
shows (*fst* (*echelon-form-of-det A bezout*)) ∗ *det A* = *det* (*snd* (*echelon-form-of-det A bezout*))
using *det-echelon-form-of-upt-k-det ib* **unfolding** *echelon-form-of-det-def* **by** *simp*

4.2.5 Proving that the first component is a unit

lemma *echelon-form-of-column-k-det-unit*:
fixes *A*::′*a*::{*bezout-domain-div*}^′*n*::{*mod-type*}^′*n*::{*mod-type*}
assumes *det*: *is-unit* (*det-P*)
shows *is-unit* (*fst* ((*echelon-form-of-column-k-det bezout*) (*det-P*,*A*,*i*) *k*))
unfolding *echelon-form-of-column-k-det-def Let-def fst-conv snd-conv* **using** *det* **by** *auto*

lemma *echelon-form-of-upt-k-det-unit*:
  fixes *A*::*'a*::{*bezout-domain-div*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
  shows *is-unit* (*fst* ((*echelon-form-of-upt-k-det bezout*) (*1*::*'a*,*A*) *k*))
proof (*induct k*)
  case *0*
  show *?case* **unfolding** *echelon-form-of-upt-k-det-def Let-def fst-conv*
    using *echelon-form-of-column-k-det-unit*[*of 1*] **by** *auto*
next
  case (*Suc k*)
  let *?f* = *foldl* (*echelon-form-of-column-k-det bezout*) (*1*,*A*,*0*) [*0*..<*Suc k*]
  have *Suc-rw*: [*0*..<*Suc* (*Suc k*)] = [*0*..<(*Suc k*)] @ [*Suc k*] **by** *simp*
  have *fold-expand*: *?f* = (*fst ?f*, *fst* (*snd ?f*), *snd* (*snd ?f*))
    by *simp*
  show *?case*
    unfolding *echelon-form-of-upt-k-det-def Let-def*
    unfolding *Suc-rw foldl-append List.foldl.simps fst-conv snd-conv*
    by (*subst fold-expand*, *rule echelon-form-of-column-k-det-unit*
   [*OF Suc.hyps*[*unfolded echelon-form-of-upt-k-det-def Let-def fst-conv snd-conv*]])
qed

lemma *echelon-form-of-unit*:
  fixes *A*::*'a*::{*bezout-domain-div*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
  shows *is-unit* (*fst* (*echelon-form-of-det A k*))
  unfolding *echelon-form-of-det-def*
  by (*rule echelon-form-of-upt-k-det-unit*)

4.2.6 Final lemmas

corollary *det-echelon-form-of-det'*:
  fixes *A*::*'a*::{*bezout-domain-div*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
  assumes *ib*: *is-bezout-ext bezout*
  shows *det A* = *1 div* (*fst* (*echelon-form-of-det A bezout*))
  ∗ *det* (*snd* (*echelon-form-of-det A bezout*))
proof −
 have (*fst* (*echelon-form-of-det A bezout*)) ∗ *det A* = *det* (*snd* (*echelon-form-of-det A bezout*))
    by (*rule det-echelon-form-of-det*[*OF ib*])
  thus *det A* = *1 div* (*fst* (*echelon-form-of-det A bezout*))
    ∗ *det* (*snd* (*echelon-form-of-det A bezout*))
    by (*auto simp add*: *echelon-form-of-unit dest*: *sym*)
qed

lemma *ef-echelon-form-of-det*:
  fixes *A*::*'a*::{*bezout-domain*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
  assumes *ib*: *is-bezout-ext bezout*
  shows *echelon-form* (*snd* (*echelon-form-of-det A bezout*))
  unfolding *echelon-form-of-det-def*
  unfolding *snd-echelon-form-of-upt-k-det-eq*

unfolding *echelon-form-of-def*[*symmetric*]
by (*rule echelon-form-echelon-form-of*[*OF ib*])

lemma *det-echelon-form*:
fixes $A::'a::\{bezout\text{-}domain\}^{\frown}'n::\{mod\text{-}type\}^{\frown}'n::\{mod\text{-}type\}$
assumes *ef*: *echelon-form A*
shows $det\ A = prod\ (\lambda i.\ A\ \$\ i\ \$\ i)\ (UNIV:: 'n\ set)$
using *det-upperdiagonal echelon-form-imp-upper-triagular*[*OF ef*]
unfolding *upper-triangular-def* **by** *blast*

corollary *det-echelon-form-of-det-prod*:
fixes $A::'a::\{bezout\text{-}domain\text{-}div\}^{\frown}'n::\{mod\text{-}type\}^{\frown}'n::\{mod\text{-}type\}$
assumes *ib*: *is-bezout-ext bezout*
shows $det\ A = 1\ div\ (fst\ (echelon\text{-}form\text{-}of\text{-}det\ A\ bezout))$
$*\ prod\ (\lambda i.\ snd\ (echelon\text{-}form\text{-}of\text{-}det\ A\ bezout)\ \$\ i\ \$\ i)\ (UNIV:: 'n\ set)$
using *det-echelon-form-of-det'*[*OF ib*]
unfolding *det-echelon-form*[*OF ef-echelon-form-of-det*[*OF ib*]] **by** *auto*

corollary *det-echelon-form-of-euclidean*[*code*]:
fixes $A::'a::\{euclidean\text{-}ring\text{-}gcd\}^{\frown}'n::\{mod\text{-}type\}^{\frown}'n::\{mod\text{-}type\}$
shows $det\ A = 1\ div\ (fst\ (echelon\text{-}form\text{-}of\text{-}det\ A\ euclid\text{-}ext2))$
$*\ prod\ (\lambda i.\ snd\ (echelon\text{-}form\text{-}of\text{-}det\ A\ euclid\text{-}ext2)\ \$\ i\ \$\ i)\ (UNIV:: 'n\ set)$
by (*rule det-echelon-form-of-det-prod*[*OF is-bezout-ext-euclid-ext2*])

end

5 Inverse matrix over principal ideal rings

theory *Echelon-Form-Inverse*
imports
Echelon-Form-Det
Gauss-Jordan.Inverse
begin

5.1 Computing the inverse of matrix over rings

lemma *scalar-mult-mat*:
fixes $x :: 'a::comm\text{-}semiring\text{-}0$
shows $x *k\ mat\ y = mat\ (x * y)$
by (*simp add*: *matrix-scalar-mult-def mat-def vec-eq-iff*)

lemma *matrix-mul-mat*:
fixes $A :: 'a::comm\text{-}semiring\text{-}1 \ ^{\frown}\ 'm\ ^{\frown}\ 'n$
shows $A ** mat\ x = x *k\ A$
by (*simp add*: *matrix-matrix-mult-def mat-def if-distrib sum.If-cases matrix-scalar-mult-def vec-eq-iff ac-simps*)

lemma *mult-adjugate-det*: $A ** adjugate\ A = mat\ (det\ A)$
using *mult-adjugate-det*[*of from-vec A*]

unfolding *det-sq-matrix-eq adjugate-eq to-vec-eq-iff* [*symmetric*] *to-vec-matrix-matrix-mult to-vec-from-vec*
by (*simp add*: *to-vec-diag*)

lemma *invertible-imp-matrix-inv*:
assumes *i*: *invertible* (*A* :: ('*a* :: {*comm-ring-1*,*euclidean-semiring*}) ^ '*b* ^ '*b*)
shows *matrix-inv A* = (*1 div* (*det A*)) ∗*k adjugate A*
proof −
let *?A* = *adjugate A*
have *A* ∗∗ *?A* = *det A* ∗*k mat 1*
unfolding *mult-adjugate-det* **by** (*simp add*: *scalar-mult-mat*)
hence *matrix-inv A* ∗∗ (*A* ∗∗ *?A*) = *matrix-inv A* ∗∗ (*det A* ∗*k mat 1*)
by *auto*
hence *?A* = *det A* ∗*k matrix-inv A*
unfolding *matrix-mul-assoc matrix-inv-left*[*OF i*] *matrix-mul-lid scalar-mult-mat matrix-mul-mat*
by *simp*
with *i* **show** *?thesis*
by (*metis* (*no-types*, *lifting*) *dvd-mult-div-cancel invertible-iff-is-unit matrix-mul-assoc matrix-mul-mat matrix-mul-rid scalar-mult-mat mult.commute*)
qed

lemma *inverse-matrix-code-rings*[*code-unfold*]:
fixes *A*::'*a*::{*euclidean-ring*}^'*n*::{*mod-type*}^'*n*::{*mod-type*}
shows *inverse-matrix A* = (*let d*=*det A in if is-unit d then Some* ((*1 div d*) ∗*k adjugate A*) *else None*)
using *invertible-imp-matrix-inv*[*of A*]
unfolding *inverse-matrix-def invertible-iff-is-unit* **by** *auto*

end

6 Examples of execution over matrices represented as functions

theory *Examples-Echelon-Form-Abstract*
imports
Code-Cayley-Hamilton
Gauss-Jordan.Examples-Gauss-Jordan-Abstract
Echelon-Form-Inverse
HOL−Computational-Algebra.Field-as-Ring
begin

The definitions introduced in this file will be also used in the computations presented in file `Examples_Echelon_Form_IArrays.thy`. Some of these definitions are not even used in this file since they are quite time consuming.

definition *test-real-6x4* :: *real*^*6*^*4*
where *test-real-6x4* = *list-of-list-to-matrix*

[[0,0,0,0,0,0],
 [0,1,0,0,0,0],
 [0,0,0,0,0,0],
 [0,0,0,0,8,2]]

value *matrix-to-list-of-list* (*minorM test-real-6x4 0 0*)

value *cofactor* (*mat 1*::*rat*^*3*^*3*) *0 0*

value *vec-to-list* (*cofactorM-row* (*mat 1*::*int*^*3*^*3*) *1*)

value *matrix-to-list-of-list* (*cofactorM* (*mat 1*::*int*^*3*^*3*))

definition *test-rat-3x3* :: *rat*^*3*^*3*
 where *test-rat-3x3* = *list-of-list-to-matrix* [[3,5,1],[2,1,3],[1,2,1]]

value *matrix-to-list-of-list* (*matpow test-rat-3x3 5*)

definition *test-int-3x3* :: *int*^*3*^*3*
 where *test-int-3x3* = *list-of-list-to-matrix* [[3,2,8], [0,3,9], [8,7,9]]

value *det test-int-3x3*

definition *test-real-3x3* :: *real*^*3*^*3*
 where *test-real-3x3* = *list-of-list-to-matrix* [[3,5,1],[2,1,3],[1,2,1]]

value *charpoly test-real-3x3*

We check that the Cayley-Hamilton theorem holds for this particular case:

value *matrix-to-list-of-list* (*evalmat* (*charpoly test-real-3x3*) *test-real-3x3*)

definition *test-int-3x3-02* :: *int*^*3*^*3*
 where *test-int-3x3-02* = *list-of-list-to-matrix* [[3,5,1],[2,1,3],[1,2,1]]

value *matrix-to-list-of-list* (*adjugate test-int-3x3-02*)

The following integer matrix is not invertible, so the result is *None*

value *inverse-matrix test-int-3x3-02*

definition *test-int-3x3-03* :: *int*^*3*^*3*
 where *test-int-3x3-03* = *list-of-list-to-matrix* [[1,−2,4],[1,−1,1],[0,1,−2]]

value *matrix-to-list-of-list* (*the* (*inverse-matrix test-int-3x3-03*))

We check that the previous inverse has been correctly computed:

value *test-int-3x3-03* ** (*the* (*inverse-matrix test-int-3x3-03*)) = (*mat 1*::*int*^*3*^*3*)

definition *test-int-8x8* :: *int*^*8*^*8*
 where *test-int-8x8* = *list-of-list-to-matrix*

```
[[ 3, 2, 3, 6, 2, 8, 5, 6],
 [ 0, 5, 5, 2, 3, 9, 4, 7],
 [ 8, 7, 9, 1, 4,−2, 2, 0],
 [ 0, 1, 5, 6, 5, 1, 1, 4],
 [ 0, 3, 4, 5, 2,−4, 2, 1],
 [ 6, 8, 6, 2, 2,−3, 3, 5],
 [−2, 4,−2, 6, 7, 8, 0, 3],
 [ 7, 1, 3, 0,−9,−3, 4,−5]]
```

SLOW; several minutes.

The following definitions will be used in file `Examples_Echelon_Form_IArrays.thy`. Using the abstract version of matrices would produce lengthy computations.

definition *test-int-6x6* :: *int^6^6*
  where *test-int-6x6* = *list-of-list-to-matrix*

```
[[ 3, 2, 3, 6, 2, 8],
 [ 0, 5, 5, 2, 3, 9],
 [ 8, 7, 9, 1, 4,−2],
 [ 0, 1, 5, 6, 5, 1],
 [ 0, 3, 4, 5, 2,−4],
 [ 6, 8, 6, 2, 2,−3]]
```

definition *test-real-6x6* :: *real^6^6*
  where *test-real-6x6* = *list-of-list-to-matrix*

```
[[ 3, 2, 3, 6, 2, 8],
 [ 0, 5, 5, 2, 3, 9],
 [ 8, 7, 9, 1, 4,−2],
 [ 0, 1, 5, 6, 5, 1],
 [ 0, 3, 4, 5, 2,−4],
 [ 6, 8, 6, 2, 2,−3]]
```

definition *test-int-20x20* :: *int^20^20*
  where *test-int-20x20* = *list-of-list-to-matrix*

```
[[3,2,3,6,2,8,5,9,8,7,5,4,7,8,9,8,7,4,5,2],
 [0,5,5,2,3,9,1,2,4,6,1,2,3,6,5,4,5,8,7,1],
 [8,7,9,1,4,−2,8,7,1,4,1,4,5,8,7,4,1,0,0,2],
 [0,1,5,6,5,1,3,5,4,9,3,2,1,4,5,6,9,8,7,4],
 [0,3,4,5,2,−4,0,2,1,0,0,0,1,2,4,5,1,1,2,0],
 [6,8,6,2,2,−3,2,4,7,9,1,2,3,6,5,4,1,2,8,7],
 [3,8,3,6,2,8,8,9,6,7,8,9,7,8,9,5,4,1,2,3,0],
 [0,8,5,2,8,9,1,2,4,6,4,6,5,8,7,9,8,7,4,5],
 [8,8,8,1,4,−2,8,7,1,4,5,5,5,6,4,5,1,2,3,6],
 [0,8,5,6,5,1,3,5,4,9::int,1,2,3,5,4,7,8,9,6,4],
 [3,2,3,6,2,8,5,9,8,7,5,4,7,3,9,8,7,4,5,2],
 [0,5,5,2,3,9,1,2,4,3,1,2,3,6,5,4,5,8,7,1],
 [1,7,9,1,4,−2,8,7,1,4,1,4,5,8,7,4,1,0,0,2],
 [1,1,5,6,5,1,3,5,4,9,3,4,5,6,9,8,7,4,5,4],
 [3,3,4,5,2,−4,0,2,1,0,0,3,1,2,4,5,1,1,2,0],
 [4,8,6,5,2,−3,2,4,2,9,1,2,3,2,5,4,1,2,8,7],
```

```
    [5,8,3,6,2,2,9,9,6,7,2,7,7,2,9,5,4,1,2,3,0],
    [2,8,5,2,8,9,5,2,4,6,4,6,5,2,7,1,8,7,4,5],
    [2,1,8,1,4,−2,8,3,1,4,5,5,5,6,4,5,1,2,3,6],
    [0,2,5,6,5,1,3,5,4,9::int,1,2,3,5,4,7,8,9,6,4]]
```

```
definition test-int-20x20-2 :: int^20^20
where test-int-20x20-2 = list-of-list-to-matrix
    [[58,18,18,41,68,62,6,21,19,78,34,22,108,63,71,38,43,52,37,24],
    [18,51,29,91,76,98,56,37,47,61,88,99,88,78,210,57,27,87,72,79],
    [49,19,81,107,43,34,69,28,101,39,21,910,27,53,15,38,5,34,47,23],
    [97,102,68,27,56,56,102,210,68,56,24,33,88,110,71,23,35,36,72,1],
    [63,11,39,16,32,81,16,98,94,26,53,23,11,51,98,51,81,57,610,85],
    [46,61,68,710,11,105,3,5,61,210,67,34,108,10,44,71,36,66,38,42],
    [39,75,106,42,36,92,110,42,89,105,11,108,22,61,65,101,410,1,1,31],
    [106,94,24,63,16,75,47,82,62,210,52,57,810,41,55,93,73,58,41,82],
    [55,49,102,9,8,41,12,110,109,310,95,51,103,71,92,85,910,410,17,21],
    [31,2,77,93,8,98,510,94,56,5,12,91,69,31,62,4,11,5,92,65],
    [22,29,103,34,64,11,9,610,1,19,35,24,21,49,31,43,81,102,14,11],
    [75,81,5,109,61,110,19,46,55,23,31,1,98,28,56,2,83,81,91,41],
    [4,510,58,41,38,106,99,103,31,84,110,63,17,105,210,61,95,103,63,51],
    [38,32,510,62,410,14,86,310,59,69,107,13,29,610,38,103,43,98,98,1],
    [101,11,3,101,99,810,10,3,510,8,35,62,45,49,34,86,63,66,71,9],
    [16,5,77,110,109,13,63,54,310,102,92,103,310,26,15,22,66,106,210,91],
    [13,810,66,51,91,84,19,25,110,41,51,87,27,79,18,69,99,95,11,46],
    [410,910,62,89,43,23,108,52,33,67,31,105,26,106,108,85,87,68,56,23],
    [310,68,21,91,107,85,94,28,101,34,109,27,63,84,25,106,65,81,7,310],
    [42,63,27,24,1010,11,107,69,910,810,31,15,97,3,56,77,51,108,31,26::int]]
end
```

7 Echelon Form refined to immutable arrays

```
theory Echelon-Form-IArrays
imports
  Echelon-Form
  Gauss-Jordan.Gauss-Jordan-IArrays
begin
```

7.1 The algorithm over immutable arrays

```
definition
  bezout-matrix-iarrays A a b j bezout =
    tabulate2 (nrows-iarray A) (nrows-iarray A)
      (let (p, q, u, v, d) = bezout (A !! a !! j) (A !! b !! j)
        in (%x y. if x = a ∧ y = a then p else
              if x = a ∧ y = b then q else
              if x = b ∧ y = a then u else
              if x = b ∧ y = b then v else
```

if x = y then 1 else 0))

primrec
bezout-iterate-iarrays :: *'a*::{*bezout-ring*} *iarray iarray* ⇒ *nat* ⇒ *nat* ⇒ *nat*
⇒ (*'a* ⇒ *'a* ⇒ (*'a* × *'a* × *'a* × *'a* × *'a*))
⇒ *'a iarray iarray*
where *bezout-iterate-iarrays A 0 i j bezout = A*
| *bezout-iterate-iarrays A* (*Suc n*) *i j bezout =*
(*if* (*Suc n*) ≤ *i*
then A
else bezout-iterate-iarrays (*bezout-matrix-iarrays A i* (*Suc n*) *j bezout* ***i*
A) *n i j bezout*)

definition
echelon-form-of-column-k-iarrays A' k =
(*let* (*A*, *i*, *bezout*) = *A'*;
nrows-A = *nrows-iarray A*;
column-Ak = *column-iarray k A*;
all-zero-below-i = *vector-all-zero-from-index* (*i+1*, *column-Ak*)
in if i = *nrows-A* ∨ (*A* !! *i* !! *k* = *0*) ∧ *all-zero-below-i*
then (*A*, *i*, *bezout*) *else*
if all-zero-below-i
then (*A*, *i* + *1*, *bezout*) *else*
let n = *least-non-zero-position-of-vector-from-index column-Ak i*;
interchange-A = *interchange-rows-iarray A i n*
in
(*bezout-iterate-iarrays interchange-A* (*nrows-A* − *1*) *i k bezout*, *i* + *1*,
bezout))

definition *echelon-form-of-upt-k-iarrays A k bezout*
= *fst* (*foldl echelon-form-of-column-k-iarrays* (*A*,*0*,*bezout*) [*0*..<*Suc k*])

definition *echelon-form-of-iarrays A bezout*
= *echelon-form-of-upt-k-iarrays A* (*ncols-iarray A* − *1*) *bezout*

7.2 Properties

7.2.1 Bezout Matrix for immutable arrays

lemma *matrix-to-iarray-bezout-matrix*:
shows *matrix-to-iarray* (*bezout-matrix A a b j bezout*)
= *bezout-matrix-iarrays* (*matrix-to-iarray A*) (*to-nat a*) (*to-nat b*) (*to-nat j*) *bezout*
(**is** *?lhs* = *?rhs*)
proof −
have *n*: *nrows-iarray* (*IArray* (*map* (*vec-to-iarray* ∘ ($) *A* ∘ *from-nat*) [*0*..<*CARD*(*'b*)]))

= *CARD*(*'b*) **unfolding** *nrows-iarray-def vec-to-iarray-def o-def* **by** *auto*
have *rw1*:(*map* (λ*x*. *IArray.of-fun*
(λ*i*. *A* $ *from-nat x* $ *from-nat i*) *CARD*(*'c*)) [*0*..<*CARD*(*'b*)] ! *to-nat a* !!
to-nat j) = *A* $ *a* $ *j*

by (*metis* (*erased*, *lifting*) *from-nat-to-nat-id length-upt minus-nat.diff-0 nth-map*

nth-upt of-fun-nth plus-nat.add-0 to-nat-less-card)
have *rw2*: (*map* (λ*x*. *IArray.of-fun*
(λ*i*. *A* \$ *from-nat x* \$ *from-nat i*) *CARD*(*'c*)) [*0*..<*CARD*(*'b*)] ! *to-nat b* !! *to-nat j*) = (*A* \$ *b* \$ *j*)
by (*metis* (*erased*, *lifting*) *from-nat-to-nat-id length-upt minus-nat.diff-0 nth-map*

nth-upt of-fun-nth plus-nat.add-0 to-nat-less-card)
have *rw3*: *IArray* (*map* (λ*x*. *IArray.of-fun*
(λ*i*. *A* \$ *from-nat x* \$ *from-nat i*) *CARD*(*'c*)) [*0*..<*CARD*(*'b*)]) !! *to-nat a* !! *to-nat j* = *A* \$ *a* \$ *j*
by (*metis IArray.sub-def list-of.simps rw1*)
have *rw4*: *IArray* (*map* (λ*x*. *IArray.of-fun*
(λ*i*. *A* \$ *from-nat x* \$ *from-nat i*) *CARD*(*'c*)) [*0*..<*CARD*(*'b*)]) !! *to-nat b* !! *to-nat j* = *A* \$ *b* \$ *j*
by (*metis IArray.sub-def list-of.simps rw2*)
show *?thesis*
unfolding *matrix-to-iarray-def bezout-matrix-iarrays-def tabulate2-def*
apply *auto* **unfolding** *n* **apply** (*rule map-ext*, *auto simp add*: *bezout-matrix-def Let-def*)
unfolding *o-def vec-to-iarray-def Let-def*
unfolding *IArray.sub-def*[*symmetric*] *rw1 rw2 rw3 rw4*
unfolding *IArray.of-fun-def iarray.inject*
apply (*rule map-ext*) **unfolding** *vec-lambda-beta*
proof
fix *x xa*
assume *x*: *x* < *CARD*(*'b*)
assume *xa* ∈ *set* [*0*..<*CARD*(*'b*)]
hence *xa*: *xa* < *CARD*(*'b*) **using** *atLeast-upt* **by** *blast*
have *rw5*: (*from-nat x* = *a*) = (*x* = *to-nat a*)
using *x from-nat-not-eq from-nat-to-nat-id* **by** *blast*
have *rw6*: (*from-nat x* = *b*) = (*x* = *to-nat b*)
by (*metis x from-nat-to-nat-id to-nat-from-nat-id*)
have *rw7*: (*from-nat xa* = *b*) = (*xa* = *to-nat b*)
by (*metis xa from-nat-to-nat-id to-nat-from-nat-id*)
have *rw8*: ((*from-nat x*::*'b*) = (*from-nat xa*::*'b*)) = (*x* = *xa*)
by (*metis from-nat-not-eq x xa*)
have *rw9*: (*from-nat xa* = *a*) = (*xa* = *to-nat a*)
by (*metis from-nat-to-nat-id to-nat-from-nat-id xa*)
have *cond01*: (*from-nat x* = *a* ∧ *from-nat xa* = *a*) == (*x* = *to-nat a* ∧ *xa* = *to-nat a*)
using *rw5 rw9* **by** *simp*
have *cond02*: (*from-nat x* = *a* ∧ *from-nat xa* = *b*) == (*x* = *to-nat a* ∧ *xa* = *to-nat b*)
using *rw5 rw7* **by** *simp*
have *cond03*: (*from-nat x* = *b* ∧ *from-nat xa* = *a*) == (*x* = *to-nat b* ∧ *xa* = *to-nat a*)
using *rw6 rw9* **by** *simp*

have *cond04*: (*from-nat x* = *b* ∧ *from-nat xa* = *b*) == (*x* = *to-nat b* ∧ *xa* = *to-nat b*)
using *rw6 rw7* **by** *simp*
have *cond05*: ((*from-nat x*::′*b*) = (*from-nat xa*::′*b*)) == (*x* = *xa*)
using *rw8* **by** *simp*
show (*case bezout* (*A* $ *a* $ *j*) (*A* $ *b* $ *j*) *of*
(*p*, *q*, *u*, *v*, *d*) ⇒
if from-nat x = *a* ∧ *from-nat xa* = *a then p*
else if from-nat x = *a* ∧ *from-nat xa* = *b then q*
else if from-nat x = *b* ∧ *from-nat xa* = *a then u*
else if from-nat x = *b* ∧ *from-nat xa* = *b then v*
else if (*from-nat x*::′*b*) = *from-nat xa then 1 else 0*) =
(*case bezout* (*A* $ *a* $ *j*) (*A* $ *b* $ *j*) *of*
(*p*, *q*, *u*, *v*, *d*) ⇒
λ*x y*. *if x* = *to-nat a* ∧ *y* = *to-nat a then p*
else if x = *to-nat a* ∧ *y* = *to-nat b then q*
else if x = *to-nat b* ∧ *y* = *to-nat a then u*
else if x = *to-nat b* ∧ *y* = *to-nat b then v*
else if x = *y then 1 else 0*)
x xa
proof (*cases bezout* (*A* $ *a* $ *j*) (*A* $ *b* $ *j*))
fix *p q u v d*
assume *b*: *bezout* (*A* $ *a* $ *j*) (*A* $ *b* $ *j*) = (*p*, *q*, *u*, *v*, *d*)
show *?thesis*
unfolding *b*
apply *clarify*
unfolding *cond01*
unfolding *cond02*
unfolding *cond03*
unfolding *cond04*
unfolding *cond05* **by** (*rule refl*)
qed
qed
qed

7.2.2 Bezout Iterate for immutable arrays

lemma *matrix-to-iarray-bezout-iterate*:
assumes *n*: *n*<*nrows A*
shows *matrix-to-iarray* (*bezout-iterate A n i j bezout*)
= *bezout-iterate-iarrays* (*matrix-to-iarray A*) *n* (*to-nat i*) (*to-nat j*) *bezout*
using *n*
proof (*induct n arbitrary*: *A*)
case *0*
thus *?case* **unfolding** *bezout-iterate-iarrays.simps bezout-iterate.simps* **by** *simp*
next
case (*Suc n*)
show *?case*
proof (*cases Suc n* ≤ *to-nat i*)

case *True*
show *?thesis*
unfolding *bezout-iterate.simps bezout-iterate-iarrays.simps*
using *True* **by** *auto*
next
case *False*
let *?B*=(*bezout-matrix-iarrays* (*matrix-to-iarray A*) (*to-nat i*) (*Suc n*) (*to-nat j*) *bezout*
∗∗*i matrix-to-iarray A*)
let *?B2*=*matrix-to-iarray* (*bezout-matrix A i* (*from-nat* (*Suc n*)) *j bezout* ∗∗ *A*)
have *matrix-to-iarray* (*bezout-iterate A* (*Suc n*) *i j bezout*)
= *matrix-to-iarray* (*bezout-iterate* (*bezout-matrix A i* (*from-nat* (*Suc n*)) *j bezout* ∗∗ *A*) *n i j bezout*)
unfolding *bezout-iterate.simps* **using** *False* **by** *auto*
also have ... = *bezout-iterate-iarrays ?B2 n* (*to-nat i*) (*to-nat j*) *bezout*
proof (*rule Suc.hyps*)
show *n* < *nrows* (*bezout-matrix A i* (*from-nat* (*Suc n*)) *j bezout* ∗∗ *A*)
using *Suc.prems* **unfolding** *nrows-def* **by** *simp*
qed
also have ... = *bezout-iterate-iarrays ?B n* (*to-nat i*) (*to-nat j*) *bezout*
unfolding *matrix-to-iarray-matrix-matrix-mult*
unfolding *matrix-to-iarray-bezout-matrix*[*of A i from-nat* (*Suc n*) *j bezout*]
unfolding *to-nat-from-nat-id*[*OF Suc.prems*[*unfolded nrows-def*]] **..**
also have ... = *bezout-iterate-iarrays* (*matrix-to-iarray A*) (*Suc n*) (*to-nat i*) (*to-nat j*) *bezout*
unfolding *bezout-iterate-iarrays.simps* **using** *False* **by** *auto*
finally show *?thesis* **.**
qed
qed

lemma *matrix-vector-all-zero-from-index2*:
fixes *A*::*'a*::{*zero*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}
shows (∀ *m*>*i*. *A* $ *m* $ *k* = *0*) = *vector-all-zero-from-index* ((*to-nat i*)+*1*, *vec-to-iarray* (*column k A*))
proof (*cases to-nat i* = *nrows A* − *1*)
case *True*
have (∀ *m*>*i*. *A* $ *m* $ *k* = *0*) = *True*
by (*metis One-nat-def Suc-pred True not-less-eq nrows-def to-nat-0 to-nat-less-card to-nat-mono*)
also have ... = *vector-all-zero-from-index* ((*to-nat i*)+*1*, *vec-to-iarray* (*column k A*))
unfolding *vector-all-zero-from-index-def Let-def*
unfolding *vec-to-iarray-def column-def*
by (*auto*, *metis True nrows-def One-nat-def Suc-pred not-le zero-less-card-finite*)
finally show *?thesis* **.**
next
case *False*
have *i-le*: *i*<*i*+*1*

by (*metis False Suc-le′ add-diff-cancel-right′ nrows-def suc-not-zero*)
hence ($\forall$ *m*>*i*. *A* \$ *m* \$ *k* = *0*) = ($\forall$ *m*≥*i*+*1*. *A* \$ *m* \$ *k* = *0*) **using** *i-le le-Suc* **by** *auto*
also have ... = *vector-all-zero-from-index* ((*to-nat i*)+*1*, *vec-to-iarray* (*column k A*))
unfolding *matrix-vector-all-zero-from-index*
by (*metis* (*mono-tags*, *opaque-lifting*) *from-nat-suc from-nat-to-nat-id i-le not-less0*

to-nat-0 to-nat-from-nat-id to-nat-mono to-nat-plus-one-less-card)
finally show *?thesis* **.**
qed

7.2.3 Echelon form of column k for immutable arrays

lemma *matrix-to-iarray-echelon-form-of-column-k*:
fixes *A*::′*a*::{*bezout-ring*}^′*cols*::{*mod-type*}^′*rows*::{*mod-type*}
assumes *k*: *k*<*ncols A*
and *i*: *i*≤*nrows A*
shows *matrix-to-iarray* (*fst* ((*echelon-form-of-column-k bezout*) (*A*,*i*) *k*))
= *fst* (*echelon-form-of-column-k-iarrays* (*matrix-to-iarray A*, *i*, *bezout*) *k*)
proof (*cases i*<*nrows A*)
case *False*
have *i-eq*: *i*=*nrows A* **by** (*metis False le-imp-less-or-eq i*)
show *matrix-to-iarray* (*fst* ((*echelon-form-of-column-k bezout*) (*A*,*i*) *k*))
= *fst* (*echelon-form-of-column-k-iarrays* (*matrix-to-iarray A*, *i*, *bezout*) *k*)
unfolding *echelon-form-of-column-k-efficient echelon-form-of-column-k-def Let-def*
unfolding *echelon-form-of-column-k-iarrays-def Let-def snd-conv fst-conv*
unfolding *matrix-to-iarray-nrows*
unfolding *i-eq matrix-to-iarray-nrows* **by** *auto*
next
case *True*
let *?interchange*=(*interchange-rows A* (*from-nat i*)
(*LEAST n*. *A* \$ *n* \$ *from-nat k* ≠ *0* ∧ *from-nat i* ≤ *n*))
have *all-zero*: ($\forall$ *m*≥*mod-type-class.from-nat i*. *A* \$ *m* \$ *mod-type-class.from-nat k* = *0*)
= *vector-all-zero-from-index* (*i*, *column-iarray k* (*matrix-to-iarray A*))
unfolding *matrix-vector-all-zero-from-index*
unfolding *to-nat-from-nat-id*[*OF True*[*unfolded nrows-def*]]
unfolding *vec-to-iarray-column*′[*OF k*] **..**
have *all-zero2*: ($\forall$ *m*>*from-nat i*. *A* \$ *m* \$ *mod-type-class.from-nat k* = *0*)
= (*vector-all-zero-from-index* (*i* + *1*, *column-iarray k* (*matrix-to-iarray A*)))
unfolding *matrix-vector-all-zero-from-index2*
unfolding *to-nat-from-nat-id*[*OF True*[*unfolded nrows-def*]]
unfolding *vec-to-iarray-column*′[*OF k*] **..**
have *n*: (*nrows-iarray* (*matrix-to-iarray A*) − *Suc 0*) < *nrows ?interchange*
unfolding *matrix-to-iarray-nrows*[*symmetric*]
unfolding *nrows-def* **by** *simp*
show *?thesis*
using *True*

```
  unfolding echelon-form-of-column-k-efficient echelon-form-of-column-k-def Let-def
split-beta
   unfolding echelon-form-of-column-k-iarrays-def Let-def snd-conv fst-conv
   unfolding matrix-to-iarray-nrows
   unfolding all-zero all-zero2 apply auto
   unfolding matrix-to-iarray-bezout-iterate[OF n]
   unfolding matrix-to-iarray-interchange-rows
    using vec-to-iarray-least-non-zero-position-of-vector-from-index'[of from-nat i
from-nat k A]
   unfolding to-nat-from-nat-id[OF True[unfolded nrows-def]]
   unfolding to-nat-from-nat-id[OF k[unfolded ncols-def]]
   unfolding vec-to-iarray-column'[OF k]
   by (auto, metis Suc-eq-plus1 all-zero all-zero2 less-le)
qed

lemma snd-matrix-to-iarray-echelon-form-of-column-k:
 fixes A::'a::{bezout-ring}^'cols::{mod-type}^'rows::{mod-type}
 assumes k: k<ncols A
 and i: i≤nrows A
 shows snd ((echelon-form-of-column-k bezout) (A,i) k)
 = fst (snd (echelon-form-of-column-k-iarrays (matrix-to-iarray A, i, bezout) k))
proof (cases i<nrows A)
 case False
 have i-eq: i=nrows A by (metis False le-imp-less-or-eq i)
 show snd ((echelon-form-of-column-k bezout) (A,i) k)
  = fst (snd (echelon-form-of-column-k-iarrays (matrix-to-iarray A, i, bezout) k))
  unfolding echelon-form-of-column-k-efficient echelon-form-of-column-k-def Let-def
   unfolding echelon-form-of-column-k-iarrays-def Let-def snd-conv fst-conv
   unfolding i-eq matrix-to-iarray-nrows by auto
next
 case True
 let ?interchange=(interchange-rows A (from-nat i)
   (LEAST n. A $ n $ from-nat k ≠ 0 ∧ from-nat i ≤ n))
 have all-zero: (∀ m≥mod-type-class.from-nat i. A $ m $ mod-type-class.from-nat
k = 0)
   = vector-all-zero-from-index (i, column-iarray k (matrix-to-iarray A))
   unfolding matrix-vector-all-zero-from-index
   unfolding to-nat-from-nat-id[OF True[unfolded nrows-def]]
   unfolding vec-to-iarray-column'[OF k] ..
 have all-zero2: (∀ m>from-nat i. A $ m $ mod-type-class.from-nat k = 0)
   = (vector-all-zero-from-index (i + 1, column-iarray k (matrix-to-iarray A)))
   unfolding matrix-vector-all-zero-from-index2
   unfolding to-nat-from-nat-id[OF True[unfolded nrows-def]]
   unfolding vec-to-iarray-column'[OF k] ..
   have Aik: A $ from-nat i $ from-nat k = matrix-to-iarray A !! i !! k
    by (metis True k matrix-to-iarray-nth ncols-def nrows-def to-nat-from-nat-id)
 show ?thesis
   using True Aik
   unfolding echelon-form-of-column-k-efficient
```

unfolding *echelon-form-of-column-k-efficient-def Let-def split-beta*
unfolding *echelon-form-of-column-k-iarrays-def Let-def snd-conv fst-conv*
unfolding *all-zero all-zero2*
unfolding *matrix-to-iarray-nrows* **by** *auto*
qed

corollary *fst-snd-matrix-to-iarray-echelon-form-of-column-k*:
fixes $A::'a::\{bezout\text{-}ring\}^{\frown}'cols::\{mod\text{-}type\}^{\frown}'rows::\{mod\text{-}type\}$
assumes *k*: *k<ncols A*
and *i*: *i≤nrows A*
shows *snd* ((*echelon-form-of-column-k bezout*) (*A*,*i*) *k*)
= *fst* (*snd* (*echelon-form-of-column-k-iarrays* (*matrix-to-iarray A*, *i*, *bezout*) *k*))
using *snd-matrix-to-iarray-echelon-form-of-column-k*[*OF assms*] **by** *simp*

7.2.4 Echelon form up to column k for immutable arrays

lemma *snd-snd-foldl-echelon-form-of-column-k-iarrays*:
snd (*snd* (*foldl echelon-form-of-column-k-iarrays* (*matrix-to-iarray A*, *0*, *bezout*) [*0*..<*k*]))
= *bezout*
proof (*induct k*)
case *0* **thus** *?case* **by** *auto*
next
case (*Suc k*)
show *?case*
using *Suc.hyps*
unfolding *echelon-form-of-column-k-iarrays-def*
unfolding *Let-def* **unfolding** *split-beta* **by** *auto*
qed

lemma *foldl-echelon-form-column-k-eq*:
fixes $A::'a::\{bezout\text{-}ring\}^{\frown}'cols::\{mod\text{-}type\}^{\frown}'rows::\{mod\text{-}type\}$
assumes *k*: *k<ncols A*
shows *matrix-to-iarray-echelon-form-of-upt-k*[*code-unfold*]:
matrix-to-iarray (*echelon-form-of-upt-k A k bezout*)
= *echelon-form-of-upt-k-iarrays* (*matrix-to-iarray A*) *k bezout*
and *fst-foldl-ef-k-eq*: *fst* (*snd* (*foldl echelon-form-of-column-k-iarrays*
(*matrix-to-iarray A*,*0*,*bezout*) [*0*..<*Suc k*]))
= *snd* (*foldl* (*echelon-form-of-column-k bezout*) (*A*,*0*) [*0*..<*Suc k*])
and *fst-foldl-ef-k-less*:
snd (*foldl* (*echelon-form-of-column-k bezout*) (*A*,*0*) [*0*..<*Suc k*]) ≤ *nrows A*
using *assms*
proof (*induct k*)
show *matrix-to-iarray* (*echelon-form-of-upt-k A 0 bezout*)
= *echelon-form-of-upt-k-iarrays* (*matrix-to-iarray A*) *0 bezout*
unfolding *echelon-form-of-upt-k-def echelon-form-of-upt-k-iarrays-def*
by (*simp*, *metis le0 matrix-to-iarray-echelon-form-of-column-k ncols-not-0 neq0-conv*)
show *fst* (*snd* (*foldl echelon-form-of-column-k-iarrays* (*matrix-to-iarray A*, *0*, *bezout*) [*0*..<*Suc 0*]))

= *snd* (*foldl* (*echelon-form-of-column-k bezout*) (*A*, *0*) [*0*..<*Suc 0*])
by (*simp*, *metis le0 ncols-not-0 not-gr0 snd-matrix-to-iarray-echelon-form-of-column-k*)
show *snd* (*foldl* (*echelon-form-of-column-k bezout*) (*A*, *0*) [*0*..<*Suc 0*]) ≤ *nrows A*
apply *simp*
unfolding *echelon-form-of-column-k-def Let-def snd-conv fst-conv*
unfolding *nrows-def* **by** *auto*
next
fix *k*
assume (*k* < *ncols A* ⟹ *matrix-to-iarray* (*echelon-form-of-upt-k A k bezout*)
= *echelon-form-of-upt-k-iarrays* (*matrix-to-iarray A*) *k bezout*)
and (*k* < *ncols A* ⟹
fst (*snd* (*foldl echelon-form-of-column-k-iarrays* (*matrix-to-iarray A*, *0*, *bezout*) [*0*..<*Suc k*])) =
snd (*foldl* (*echelon-form-of-column-k bezout*) (*A*, *0*) [*0*..<*Suc k*]))
and *hyp3*: (*k* < *ncols A* ⟹ *snd* (*foldl* (*echelon-form-of-column-k bezout*) (*A*, *0*) [*0*..<*Suc k*]) ≤ *nrows A*)
and *Suc-k-less-card*: *Suc k* < *ncols A*
hence *hyp1*: *matrix-to-iarray* (*echelon-form-of-upt-k A k bezout*)
= *echelon-form-of-upt-k-iarrays* (*matrix-to-iarray A*) *k bezout*
and *hyp2*: *fst* (*snd* (*foldl echelon-form-of-column-k-iarrays*
(*matrix-to-iarray A*, *0*, *bezout*) [*0*..<*Suc k*]))
= *snd* (*foldl* (*echelon-form-of-column-k bezout*) (*A*, *0*) [*0*..<*Suc k*])
and *hyp3*: *snd* (*foldl* (*echelon-form-of-column-k bezout*) (*A*, *0*) [*0*..<*Suc k*]) ≤ *nrows A*
by *auto*
hence *hyp1-unfolded*: *matrix-to-iarray* (*fst* (*foldl* (*echelon-form-of-column-k bezout*) (*A*,*0*) [*0*..<*Suc k*]))
= *fst* (*foldl echelon-form-of-column-k-iarrays* (*matrix-to-iarray A*,*0*,*bezout*) [*0*..<*Suc k*])
using *hyp1* **unfolding** *echelon-form-of-upt-k-def echelon-form-of-upt-k-iarrays-def* **by** *simp*
have *upt-rw*: [*0*..<*Suc* (*Suc k*)] = [*0*..<*Suc k*] @ [(*Suc k*)] **by** *auto*
let *?f* = *foldl echelon-form-of-column-k-iarrays* (*matrix-to-iarray A*, *0*, *bezout*) [*0*..<*Suc k*]
let *?f2* = *foldl* (*echelon-form-of-column-k bezout*) (*A*,*0*) [*0*..<(*Suc k*)]
have *fold-rw*: *?f* = (*fst ?f*, *fst* (*snd ?f*), *snd* (*snd ?f*)) **by** *simp*
have *fold-rw'*: *?f2* = (*fst ?f2*, *snd ?f2*) **by** *simp*
have *rw*: *snd* (*foldl* (*echelon-form-of-column-k bezout*) (*A*, *0*) [*0*..<*Suc k*])
= *fst* (*snd* (*foldl echelon-form-of-column-k-iarrays* (*matrix-to-iarray A*, *0*, *bezout*) [*0*..<*Suc k*]))
using *hyp2* **by** *auto*
show *fst* (*snd* (*foldl echelon-form-of-column-k-iarrays* (*matrix-to-iarray A*, *0*, *bezout*)
[*0*..<*Suc* (*Suc k*)])) = *snd* (*foldl* (*echelon-form-of-column-k bezout*) (*A*, *0*) [*0*..<*Suc* (*Suc k*)])
unfolding *upt-rw foldl-append* **unfolding** *List.foldl.simps* **apply** (*subst fold-rw*)

apply (*subst fold-rw'*) **unfolding** *hyp2* **unfolding** *hyp1-unfolded*[*symmetric*]

unfolding *rw*
unfolding *snd-snd-foldl-echelon-form-of-column-k-iarrays*
proof (*rule fst-snd-matrix-to-iarray-echelon-form-of-column-k* [*symmetric*])
show *Suc k* < *ncols* (*fst ?f2*) **using** *Suc-k-less-card* **unfolding** *ncols-def* **.**
show *fst* (*snd* (*foldl echelon-form-of-column-k-iarrays* (*matrix-to-iarray A*, *0*, *bezout*) [*0*..<*Suc k*]))
≤ *nrows* (*fst* (*foldl* (*echelon-form-of-column-k bezout*) (*A*, *0*) [*0*..<*Suc k*]))
by (*metis hyp2 hyp3 nrows-def*)
qed
show *matrix-to-iarray* (*echelon-form-of-upt-k A* (*Suc k*) *bezout*)
= *echelon-form-of-upt-k-iarrays* (*matrix-to-iarray A*) (*Suc k*) *bezout*
unfolding *echelon-form-of-upt-k-def echelon-form-of-upt-k-iarrays-def*
upt-rw foldl-append List.foldl.simps **apply** (*subst fold-rw*) **apply** (*subst fold-rw′*)
unfolding *hyp2 hyp1-unfolded*[*symmetric*]
unfolding *rw*
unfolding *snd-snd-foldl-echelon-form-of-column-k-iarrays*
proof (*rule matrix-to-iarray-echelon-form-of-column-k*)
show *Suc k* < *ncols* (*fst ?f2*) **using** *Suc-k-less-card* **unfolding** *ncols-def* **.**
show *fst* (*snd* (*foldl echelon-form-of-column-k-iarrays* (*matrix-to-iarray A*, *0*, *bezout*) [*0*..<*Suc k*]))
≤ *nrows* (*fst* (*foldl* (*echelon-form-of-column-k bezout*) (*A*, *0*) [*0*..<*Suc k*]))
by (*metis hyp2 hyp3 nrows-def*)
qed
show *snd* (*foldl* (*echelon-form-of-column-k bezout*) (*A*, *0*) [*0*..<*Suc* (*Suc k*)]) ≤ *nrows A*
using [[*unfold-abs-def* = *false*]]
unfolding *upt-rw foldl-append* **unfolding** *List.foldl.simps* **apply** (*subst fold-rw′*)
unfolding *echelon-form-of-column-k-def Let-def*
using *hyp3 le-antisym not-less-eq-eq* **unfolding** *nrows-def* **by** *fastforce*
qed

7.2.5 Echelon form up to column k for immutable arrays

lemma *matrix-to-iarray-echelon-form-of*[*code-unfold*]:
matrix-to-iarray (*echelon-form-of A bezout*)
= *echelon-form-of-iarrays* (*matrix-to-iarray A*) *bezout*
unfolding *echelon-form-of-def echelon-form-of-iarrays-def*
by (*metis* (*poly-guards-query*) *One-nat-def diff-less lessI matrix-to-iarray-echelon-form-of-upt-k*
ncols-def ncols-eq-card-columns zero-less-card-finite)

end

8 Determinant of matrices computed using immutable arrays

theory *Echelon-Form-Det-IArrays*

imports
Echelon-Form-Det
Echelon-Form-IArrays
begin

8.1 Definitions

definition *echelon-form-of-column-k-det-iarrays* ::
$'a::\{bezout\text{-}ring\} \times 'a\ iarray\ iarray \times nat \times ('a \Rightarrow 'a \Rightarrow 'a \times 'a \times 'a \times 'a \times 'a)$
$\Rightarrow nat$
$\Rightarrow 'a \times 'a\ iarray\ iarray \times nat \times ('a \Rightarrow 'a \Rightarrow 'a \times 'a \times 'a \times 'a \times 'a)$
where
echelon-form-of-column-k-det-iarrays A' k =
(*let* (*det-P*, A, i, *bezout*) = A'
in if (($i \neq$ *nrows-iarray* A) $\wedge$ (A !! i !! k = *0*)
$\wedge$ ($\neg$ *vector-all-zero-from-index* (i + *1*, (*column-iarray* k A))))
then (−*1* ∗ *det-P*, *echelon-form-of-column-k-iarrays* (A, i, *bezout*) k)
else (*det-P*,*echelon-form-of-column-k-iarrays* (A,i,*bezout*) k))

definition *echelon-form-of-upt-k-det-iarrays* A' k *bezout* =
(*let* A = *snd* A';
f = *foldl echelon-form-of-column-k-det-iarrays* (*1*, A, *0*, *bezout*) [*0*..<*Suc* k]
in (*fst* f, *fst* (*snd* f)))

definition *echelon-form-of-det-iarrays* ::
$'a::\{bezout\text{-}ring\}$ *iarray iarray*
$\Rightarrow ('a \Rightarrow 'a \Rightarrow 'a \times 'a \times 'a \times 'a \times 'a)$
$\Rightarrow ('a \times ('a\ iarray\ iarray))$
where
echelon-form-of-det-iarrays A *bezout* =
echelon-form-of-upt-k-det-iarrays (*1*::$'a$, A) (*ncols-iarray* A − *1*) *bezout*

definition *det-iarrays-rings* A =
(*let* A' = *echelon-form-of-det-iarrays* A *euclid-ext2*
in 1 div (*fst* A') ∗ *prod-list* (*map* (λi. (*snd* A') !! i !! i) [*0*..<*nrows-iarray* A]))

8.2 Properties

8.2.1 Echelon Form of column k

lemma *vector-all-zero-from-index3*:
fixes $A::'a::\{bezout\text{-}ring\}^{\frown}'cols::\{mod\text{-}type\}^{\frown}'rows::\{mod\text{-}type\}$
shows ($\exists m>i$. A \$ m \$ $k \neq$ *0*)
= ($\neg$ *vector-all-zero-from-index* (*to-nat* i + *1*, *vec-to-iarray* (*column* k A)))
using *matrix-vector-all-zero-from-index2*
proof −
have ($\forall m>i$. A \$ m \$ k = *0*) = (*vector-all-zero-from-index* (*to-nat* i + *1*, *vec-to-iarray* (*column* k A)))

using *matrix-vector-all-zero-from-index2*[*of i A k*] **by** *auto*
hence (¬ (∀ *m*>*i*. *A* $ *m* $ *k* = *0*))
= (¬(*vector-all-zero-from-index* (*to-nat i* + *1*, *vec-to-iarray* (*column k A*))))
by *auto*
thus *?thesis* **by** *auto*
qed

lemma *fst-matrix-to-iarray-echelon-form-of-column-k-det*:
assumes *k*: *k*<*ncols A* **and** *i*: *i*≤*nrows A*
shows *fst* ((*echelon-form-of-column-k-det bezout*) (*det-P*, *A*, *i*) *k*)
= *fst* (*echelon-form-of-column-k-det-iarrays* (*det-P*, *matrix-to-iarray A*, *i*, *bezout*) *k*)
proof (*cases i*<*nrows A*)
case *True*
have *ex-rw*: (∃ *m*>*from-nat i*. *A* $ *m* $ *from-nat k* ≠ *0*)
= (¬ *vector-all-zero-from-index* (*i* + *1*, *column-iarray k* (*matrix-to-iarray A*)))
using *vector-all-zero-from-index3*[*of from-nat i A from-nat k*]
unfolding *vec-to-iarray-column*
unfolding *to-nat-from-nat-id*[*OF k*[*unfolded ncols-def*]]
unfolding *to-nat-from-nat-id*[*OF True*[*unfolded nrows-def*]] **.**
have *Aik*: *matrix-to-iarray A* !! *i* !! *k* = *A* $ (*from-nat i*) $ (*from-nat k*)
by (*metis True k matrix-to-iarray-nth ncols-def nrows-def to-nat-from-nat-id*)
show *?thesis*
unfolding *echelon-form-of-column-k-det-iarrays-def echelon-form-of-column-k-det-def*
unfolding *Let-def*
unfolding *split-beta*
unfolding *fst-conv snd-conv*
unfolding *matrix-to-iarray-nrows*
unfolding *ex-rw Aik* **by** *auto*
next
case *False*
hence *i2*: *i*=*nrows A* **using** *i* **by** *simp*
thus *?thesis*
unfolding *echelon-form-of-column-k-det-iarrays-def echelon-form-of-column-k-det-def*
unfolding *Let-def fst-conv snd-conv*
unfolding *matrix-to-iarray-nrows*
unfolding *i2* **unfolding** *matrix-to-iarray-nrows* **by** *auto*
qed

lemma *snd-echelon-form-of-column-k-det*:
shows (*snd* (*echelon-form-of-column-k-det-iarrays* (*det-P*, *A*, *i*, *bezout*) *k*))
= *echelon-form-of-column-k-iarrays* (*A*,*i*,*bezout*) *k*
unfolding *echelon-form-of-column-k-det-iarrays-def Let-def* **by** *auto*

lemma *fst-snd-echelon-form-of-column-k-le-nrows*:
assumes *i*≤*nrows A*
shows *snd* ((*echelon-form-of-column-k bezout*) (*A*, *i*) *k*) ≤ *nrows A*
using *assms*

unfolding *echelon-form-of-column-k-def Let-def fst-conv snd-conv*
unfolding *nrows-def* **by** *auto*

lemma *fst-snd-snd-echelon-form-of-column-k-det-le-nrows*:
assumes *i*≤*nrows A*
shows *snd* (*snd* ((*echelon-form-of-column-k-det bezout*) (*n*, *A*, *i*) *k*)) ≤ *nrows A*
unfolding *echelon-form-of-column-k-det-def Let-def fst-conv snd-conv*
by (*simp add*: *assms fst-snd-echelon-form-of-column-k-le-nrows*)

8.2.2 Echelon Form up to column k

lemma *snd-snd-snd-foldl-echelon-form-of-column-k-det-iarrays*:
snd (*snd* (*snd* (*foldl echelon-form-of-column-k-det-iarrays* (*n*, *A*, *0*, *bezout*) [*0*..<*k*])))
= *bezout*
proof (*induct k*)
case *0*
show *?case* **by** *auto*
next
case (*Suc k*)
show *?case*
apply *auto*
apply (*simp only*: *echelon-form-of-column-k-det-iarrays-def Let-def*)
apply (*auto simp add*: *split-beta echelon-form-of-column-k-iarrays-def Let-def Suc.hyps*)
done
qed

lemma *matrix-to-iarray-echelon-form-of-column-k-det*:
assumes *k*<*ncols A* **and** *i*≤*nrows A*
shows *matrix-to-iarray* (*fst* (*snd* ((*echelon-form-of-column-k-det bezout*) (*n*, *A*, *i*) *k*)))
= (*fst* (*snd* (*echelon-form-of-column-k-det-iarrays* (*n*, *matrix-to-iarray A*, *i*, *bezout*) *k*)))
unfolding *snd-echelon-form-of-column-k-det*
unfolding *echelon-form-of-column-k-det-def Let-def fst-conv snd-conv*
using *assms matrix-to-iarray-echelon-form-of-column-k* **by** *auto*

lemma *fst-snd-snd-echelon-form-of-column-k-det*:
assumes *k* < *ncols A*
and *i* ≤ *nrows A*
shows *snd* (*snd* ((*echelon-form-of-column-k-det bezout*) (*n*,*A*,*i*) *k*))
= *fst* (*snd* (*snd* (*echelon-form-of-column-k-det-iarrays* (*n*,*matrix-to-iarray A*, *i*, *bezout*) *k*)))
unfolding *snd-echelon-form-of-column-k-det-eq*
unfolding *snd-echelon-form-of-column-k-det*
by (*rule fst-snd-matrix-to-iarray-echelon-form-of-column-k*[*OF assms*])

lemma
fixes *A::'a::{bezout-domain}^'cols::{mod-type}^'rows::{mod-type}*
assumes *k<ncols A*
shows *matrix-to-iarray-fst-echelon-form-of-upt-k-det*:
fst ((echelon-form-of-upt-k-det bezout) (1::'a,A) k)
= fst (echelon-form-of-upt-k-det-iarrays (1::'a,matrix-to-iarray A) k bezout)
and *matrix-to-iarray-snd-echelon-form-of-upt-k-det*:
matrix-to-iarray ((snd ((echelon-form-of-upt-k-det bezout) (1::'a,A) k)))
= (snd (echelon-form-of-upt-k-det-iarrays (1::'a, matrix-to-iarray A) k bezout))
and *snd (snd (foldl (echelon-form-of-column-k-det bezout) (1::'a,A,0) [0..<Suc k])) ≤ nrows A*
and *fst (snd (snd (foldl echelon-form-of-column-k-det-iarrays*
(1::'a,matrix-to-iarray A,0,bezout) [0..<Suc k]))) = snd (snd
(foldl (echelon-form-of-column-k-det bezout) (1::'a,A,0) [0..<Suc k]))
using *assms*
proof *(induct k)*
show *fst ((echelon-form-of-upt-k-det bezout) (1, A) 0)*
= fst (echelon-form-of-upt-k-det-iarrays (1, matrix-to-iarray A) 0 bezout)
unfolding *echelon-form-of-upt-k-det-def echelon-form-of-upt-k-det-iarrays-def Let-def*
by *(auto, metis fst-matrix-to-iarray-echelon-form-of-column-k-det le0 ncols-not-0 neq0-conv)*
show *matrix-to-iarray (snd ((echelon-form-of-upt-k-det bezout) (1, A) 0)) =*
snd (echelon-form-of-upt-k-det-iarrays (1, matrix-to-iarray A) 0 bezout)
unfolding *echelon-form-of-upt-k-det-def echelon-form-of-upt-k-det-iarrays-def Let-def*
by *(auto, metis le0 matrix-to-iarray-echelon-form-of-column-k ncols-not-0 neq0-conv*

snd-echelon-form-of-column-k-det snd-echelon-form-of-column-k-det-eq)
show *snd (snd (foldl (echelon-form-of-column-k-det bezout)(1, A, 0) [0..<Suc 0])) ≤ nrows A*
by *(simp add: fst-snd-snd-echelon-form-of-column-k-det-le-nrows)*
show *fst (snd (snd (foldl echelon-form-of-column-k-det-iarrays (1, matrix-to-iarray A, 0, bezout) [0..<Suc 0]))) =*
snd (snd (foldl (echelon-form-of-column-k-det bezout) (1, A, 0) [0..<Suc 0]))
by *(auto, metis fst-snd-matrix-to-iarray-echelon-form-of-column-k le0 ncols-not-0 neq0-conv*
snd-echelon-form-of-column-k-det snd-echelon-form-of-column-k-det-eq)
next
fix *k*
assume *(k < ncols A ⟹ fst ((echelon-form-of-upt-k-det bezout) (1::'a, A) k)*
= fst (echelon-form-of-upt-k-det-iarrays (1::'a, matrix-to-iarray A) k bezout))
and *(k < ncols A ⟹*
matrix-to-iarray (snd ((echelon-form-of-upt-k-det bezout) (1::'a, A) k))
= snd (echelon-form-of-upt-k-det-iarrays (1::'a, matrix-to-iarray A) k bezout))
and *(k < ncols A ⟹*
snd (snd (foldl (echelon-form-of-column-k-det bezout) (1::'a, A, 0) [0..<Suc k])) ≤ nrows A)

and (k < *ncols A* ⟹
fst (*snd* (*snd* (*foldl echelon-form-of-column-k-det-iarrays* (*1*::′*a*, *matrix-to-iarray A*, *0*, *bezout*) [*0*..<*Suc k*]))) =
snd (*snd* (*foldl* (*echelon-form-of-column-k-det bezout*) (*1*::′*a*, *A*, *0*) [*0*..<*Suc k*])))
and *S*: *Suc k* < *ncols A*
hence *hyp1*: *fst* ((*echelon-form-of-upt-k-det bezout*) (*1*::′*a*, *A*) *k*)
= *fst* (*echelon-form-of-upt-k-det-iarrays* (*1*::′*a*, *matrix-to-iarray A*) *k bezout*)
and *hyp2*: *matrix-to-iarray* (*snd* ((*echelon-form-of-upt-k-det bezout*) (*1*::′*a*, *A*) *k*))
= *snd* (*echelon-form-of-upt-k-det-iarrays* (*1*::′*a*, *matrix-to-iarray A*) *k bezout*)
and *hyp3*: *snd* (*snd* (*foldl* (*echelon-form-of-column-k-det bezout*) (*1*::′*a*, *A*, *0*) [*0*..<*Suc k*]))
≤ *nrows A*
and *hyp4*: *fst* (*snd* (*snd* (*foldl echelon-form-of-column-k-det-iarrays*
(*1*::′*a*, *matrix-to-iarray A*, *0*, *bezout*) [*0*..<*Suc k*])))
= *snd* (*snd* (*foldl* (*echelon-form-of-column-k-det bezout*) (*1*::′*a*, *A*, *0*) [*0*..<*Suc k*]))
by *auto*
have *list-rw*: [*0*..<*Suc* (*Suc k*)] = [*0*..<(*Suc k*)] @ [*Suc k*] **by** *simp*
let *?f* = *foldl* (*echelon-form-of-column-k-det bezout*) (*1*, *A*, *0*) [*0*..<*Suc k*]
have *f-rw*: *?f*= (*fst ?f*, *fst* (*snd ?f*), *snd* (*snd ?f*)) **by** *simp*
let *?g*=(*foldl echelon-form-of-column-k-det-iarrays* (*1*, *matrix-to-iarray A*, *0*, *bezout*) [*0*..<*Suc k*])
have *g-rw*: *?g* = (*fst ?g*, *fst* (*snd ?g*), *fst* (*snd* (*snd ?g*)), *snd* (*snd* (*snd ?g*))) **by** *simp*
have *rw1*: *fst ?g* = *fst ?f*
using *hyp1*[*unfolded echelon-form-of-upt-k-det-def echelon-form-of-upt-k-det-iarrays-def Let-def*
fst-conv snd-conv] **..**
have *rw2*: *fst* (*snd ?g*) = *matrix-to-iarray* (*fst* (*snd ?f*))
using *hyp2*[*unfolded echelon-form-of-upt-k-det-def*
echelon-form-of-upt-k-det-iarrays-def Let-def snd-conv] **..**
have *rw3*: *fst* (*snd* (*snd ?g*)) = *snd* (*snd ?f*)
using *hyp4* **.**

show *fst* ((*echelon-form-of-upt-k-det bezout*) (*1*, *A*) (*Suc k*))
= *fst* (*echelon-form-of-upt-k-det-iarrays* (*1*, *matrix-to-iarray A*) (*Suc k*) *bezout*)
unfolding *echelon-form-of-upt-k-det-iarrays-def echelon-form-of-upt-k-det-def Let-def fst-conv snd-conv*
unfolding *list-rw foldl-append*
unfolding *List.foldl.simps*
apply (*subst f-rw*)
apply (*subst g-rw*)
unfolding *rw1*[*symmetric*] *rw2 rw3*
unfolding *snd-snd-snd-foldl-echelon-form-of-column-k-det-iarrays*
proof (*rule fst-matrix-to-iarray-echelon-form-of-column-k-det*)
show *Suc k* < *ncols* (*fst* (*snd ?f*)) **using** *S* **unfolding** *ncols-def* **.**
show *snd* (*snd* (*foldl* (*echelon-form-of-column-k-det bezout*) (*1*, *A*, *0*) [*0*..<*Suc*

```
k]))
   ≤ nrows (fst (snd (foldl (echelon-form-of-column-k-det bezout) (1, A, 0) [0..<Suc
k])))
    by (metis hyp3 nrows-def)
  qed
  show matrix-to-iarray (snd ((echelon-form-of-upt-k-det bezout) (1, A) (Suc k)))
=
    snd (echelon-form-of-upt-k-det-iarrays (1, matrix-to-iarray A) (Suc k) bezout)
     unfolding echelon-form-of-upt-k-det-iarrays-def echelon-form-of-upt-k-det-def
Let-def fst-conv snd-conv
    unfolding list-rw foldl-append
    unfolding List.foldl.simps
    apply (subst f-rw)
    apply (subst g-rw)
   unfolding rw1[symmetric] rw2 rw3 unfolding snd-snd-snd-foldl-echelon-form-of-column-k-det-iarrays
  proof (rule matrix-to-iarray-echelon-form-of-column-k-det)
    show Suc k < ncols (fst (snd ?f)) using S unfolding ncols-def .
   show snd (snd (foldl (echelon-form-of-column-k-det bezout) (1, A, 0) [0..<Suc
k]))
   ≤ nrows (fst (snd (foldl (echelon-form-of-column-k-det bezout) (1, A, 0) [0..<Suc
k])))
    by (metis hyp3 nrows-def)
  qed
  show snd (snd (foldl (echelon-form-of-column-k-det bezout) (1, A, 0) [0..<Suc
(Suc k)])) ≤ nrows A
    unfolding list-rw foldl-append List.foldl.simps
    apply (subst f-rw)
    using fst-snd-snd-echelon-form-of-column-k-det-le-nrows
    by (metis hyp3 nrows-def)
  show fst (snd (snd (foldl echelon-form-of-column-k-det-iarrays
    (1, matrix-to-iarray A, 0, bezout) [0..<Suc (Suc k)])))
     = snd (snd (foldl (echelon-form-of-column-k-det bezout) (1, A, 0) [0..<Suc
(Suc k)]))
     unfolding echelon-form-of-upt-k-det-iarrays-def echelon-form-of-upt-k-det-def
Let-def fst-conv snd-conv
    unfolding list-rw foldl-append
    unfolding List.foldl.simps
    apply (subst f-rw)
    apply (subst g-rw)
    unfolding rw1[symmetric] rw2 rw3
     unfolding snd-snd-snd-foldl-echelon-form-of-column-k-det-iarrays
  proof (rule fst-snd-snd-echelon-form-of-column-k-det[symmetric])
    show Suc k < ncols (fst (snd ?f)) using S unfolding ncols-def .
   show snd (snd (foldl (echelon-form-of-column-k-det bezout) (1, A, 0) [0..<Suc
k]))
   ≤ nrows (fst (snd (foldl (echelon-form-of-column-k-det bezout) (1, A, 0) [0..<Suc
k])))
    by (metis hyp3 nrows-def)
  qed
```

qed

8.2.3 Echelon Form

lemma *matrix-to-iarray-echelon-form-of-det*[*code-unfold*]:
matrix-to-iarray (*snd* (*echelon-form-of-det A bezout*))
= *snd* (*echelon-form-of-det-iarrays* (*matrix-to-iarray A*) *bezout*)
unfolding *echelon-form-of-det-def echelon-form-of-det-iarrays-def*
unfolding *matrix-to-iarray-ncols*[*symmetric*]
by (*rule matrix-to-iarray-snd-echelon-form-of-upt-k-det*, *simp add*: *ncols-def*)

lemma *fst-echelon-form-of-det*[*code-unfold*]:
(*fst* (*echelon-form-of-det A bezout*))
= *fst* (*echelon-form-of-det-iarrays* (*matrix-to-iarray A*) *bezout*)
unfolding *echelon-form-of-det-def echelon-form-of-det-iarrays-def*
unfolding *matrix-to-iarray-ncols*[*symmetric*]
by (*rule matrix-to-iarray-fst-echelon-form-of-upt-k-det*, *simp add*: *ncols-def*)

8.2.4 Computing the determinant

lemma *det-echelon-form-of-euclidean-iarrays*[*code*]:
fixes *A*::*'a*::{*euclidean-ring-gcd*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *det A* = (*let A'* = *echelon-form-of-det-iarrays* (*matrix-to-iarray A*) *euclid-ext2*
in 1 div (*fst A'*)
∗ *prod-list* (*map* (*λi.* (*snd A'*) !! *i* !! *i*) [*0*..<*nrows-iarray* (*matrix-to-iarray A*)]))
proof −
let *?f*=(*λi. snd* (*echelon-form-of-det-iarrays* (*matrix-to-iarray A*) *euclid-ext2*) !! *i* !! *i*)
have *prod-list* (*map ?f* [*0*..<*nrows-iarray* (*matrix-to-iarray A*)])
= *prod ?f* (*set* [*0*..<*nrows-iarray* (*matrix-to-iarray A*)])
by (*metis* (*mono-tags*, *lifting*) *distinct-upt prod.distinct-set-conv-list*)
also have ... = *prod* (*λi. snd* (*echelon-form-of-det A euclid-ext2*) \$ *i* \$ *i*) (*UNIV*::*'n set*)
proof (*rule prod.reindex-cong*[*of to-nat*::(*'n*=>*nat*)])
show *inj* (*to-nat*::(*'n*=>*nat*)) **by** (*metis strict-mono-imp-inj-on strict-mono-to-nat*)
show *set* [*0*..<*nrows-iarray* (*matrix-to-iarray A*)] = *range* (*to-nat*::*'n*=>*nat*)
unfolding *nrows-eq-card-rows* **using** *bij-to-nat*[**where** *?'a*=*'n*]
unfolding *bij-betw-def*
unfolding *atLeast0LessThan atLeast-upt* **by** *auto*
fix *x*
show *snd* (*echelon-form-of-det-iarrays* (*matrix-to-iarray A*) *euclid-ext2*) !! *to-nat x* !! *to-nat x*
= *snd* (*echelon-form-of-det A euclid-ext2*) \$ *x* \$ *x*
unfolding *matrix-to-iarray-echelon-form-of-det*[*symmetric*]
unfolding *matrix-to-iarray-nth* **..**
qed
finally have ∗:*prod-list* (*map* (*λi. snd* (*echelon-form-of-det-iarrays*
(*matrix-to-iarray A*) *euclid-ext2*) !! *i* !! *i*) [*0*..<*nrows-iarray* (*matrix-to-iarray A*)]) =

($\prod i{\in}UNIV.$ *snd* (*echelon-form-of-det A euclid-ext2*) $ *i* $ *i*) **.**
have *det A* = *1 div* (*fst* (*echelon-form-of-det A euclid-ext2*))
∗ *prod* (λ*i*. *snd* (*echelon-form-of-det A euclid-ext2*) $ *i* $ *i*) (*UNIV*:: *'n set*)
unfolding *det-echelon-form-of-euclidean* **..**
also have ... = (*let A'* = *echelon-form-of-det-iarrays* (*matrix-to-iarray A*) *euclid-ext2*
in 1 div (*fst A'*)
∗ *prod-list* (*map* (λ*i*. (*snd A'*) !! *i* !! *i*) [*0*..<*nrows-iarray* (*matrix-to-iarray A*)]))
unfolding *Let-def* **unfolding** ∗ *fst-echelon-form-of-det* **..**
finally show *?thesis* **.**
qed

corollary *matrix-to-iarray-det-euclidean-ring*:
fixes *A*::*'a*::{*euclidean-ring-gcd*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *det A* = *det-iarrays-rings* (*matrix-to-iarray A*)
unfolding *det-echelon-form-of-euclidean-iarrays det-iarrays-rings-def* **..**

8.2.5 Computing the characteristic polynomial of a matrix

definition *mat2matofpoly-iarrays A*
= *tabulate2* (*nrows-iarray A*) (*ncols-iarray A*) (λ*i j*. [:*A* !! *i* !! *j*:])

lemma *matrix-to-iarray-mat2matofpoly*[*code-unfold*]:
matrix-to-iarray (*mat2matofpoly A*) = *mat2matofpoly-iarrays* (*matrix-to-iarray A*)
unfolding *mat2matofpoly-def mat2matofpoly-iarrays-def tabulate2-def*
proof (*rule matrix-to-iarray-eq-of-fun*, *auto*)
show *nrows-iarray* (*matrix-to-iarray A*) = *length* (*IArray.list-of* (*matrix-to-iarray* (χ *i j*. [:*A* $ *i* $ *j*:])))
unfolding *nrows-iarray-def matrix-to-iarray-def* **by** *simp*
fix *i*
show *vec-to-iarray* (χ *j*. [:*A* $ *i* $ *j*:]) =
IArray (*map* (λ*j*. [:*IArray.list-of* (*IArray.list-of* (*matrix-to-iarray A*) ! *mod-type-class.to-nat i*) ! *j*:])
[*0*..<*ncols-iarray* (*matrix-to-iarray A*)])
unfolding *vec-to-iarray-def*
unfolding *matrix-to-iarray-ncols*[*symmetric*] **unfolding** *ncols-def*
by (*auto*, *metis IArray.sub-def vec-matrix vec-to-iarray-nth*)
qed

The following two lemmas must be added to the file *Matrix-To-IArray* of the AFP Gauss-Jordan development.

lemma *vec-to-iarray-minus*[*code-unfold*]: *vec-to-iarray* (*a* − *b*)
= (*vec-to-iarray a*) − (*vec-to-iarray b*)
unfolding *vec-to-iarray-def*
unfolding *minus-iarray-def* **by** *auto*

lemma *matrix-to-iarray-minus*[*code-unfold*]: *matrix-to-iarray* (*A* − *B*)

= (*matrix-to-iarray A*) − (*matrix-to-iarray B*)
unfolding *matrix-to-iarray-def o-def*
by (*simp add*: *minus-iarray-def Let-def vec-to-iarray-minus*)

definition *charpoly-iarrays A*
= *det-iarrays-rings* (*mat-iarray* (*monom 1* (*Suc 0*)) (*nrows-iarray A*) − *mat2matofpoly-iarrays A*)

lemma *matrix-to-iarray-charpoly*[*code*]: *charpoly A* = *charpoly-iarrays* (*matrix-to-iarray A*)
unfolding *charpoly-def charpoly-iarrays-def*
unfolding *matrix-to-iarray-mat2matofpoly*[*symmetric*]
unfolding *matrix-to-iarray-nrows*[*symmetric*] *nrows-def*
unfolding *matrix-to-iarray-mat*[*symmetric*]
unfolding *matrix-to-iarray-minus*[*symmetric*]
unfolding *det-iarrays-rings-def*
unfolding *det-echelon-form-of-euclidean-iarrays* **..**

end

9 Code Cayley Hamilton

theory *Code-Cayley-Hamilton-IArrays*
imports
Cayley-Hamilton.Cayley-Hamilton
Echelon-Form-Det-IArrays
begin

9.1 Implementations over immutable arrays of some definitions presented in the Cayley-Hamilton development

definition *scalar-matrix-mult-iarrays* :: (*'a*::*ab-semigroup-mult*) ⇒ (*'a iarray iarray*) ⇒ (*'a iarray iarray*)
(**infixl** ‹∗*ssi*› *70*) **where** *c* ∗*ssi A* = *tabulate2* (*nrows-iarray A*) (*ncols-iarray A*) (% *i j*. *c* ∗ (*A* !! *i* !! *j*))

definition *minorM-iarrays A i j* = *tabulate2* (*nrows-iarray A*) (*ncols-iarray A*)
(%*k l*. *if k* = *i* ∧ *l* = *j then 1 else if k* = *i* ∨ *l* = *j then 0 else A* !! *k* !! *l*)
definition *cofactor-iarrays A i j* = *det-iarrays-rings* (*minorM-iarrays A i j*)
definition *cofactorM-iarrays A* = *tabulate2* (*nrows-iarray A*) (*nrows-iarray A*)
(%*i j*. *cofactor-iarrays A i j*)
definition *adjugate-iarrays A* = *transpose-iarray* (*cofactorM-iarrays A*)

lemma *matrix-to-iarray-scalar-matrix-mult*[*code-unfold*]:
matrix-to-iarray (*k* ∗*k A*) = *k* ∗*ssi* (*matrix-to-iarray A*)
proof −
have *n-rw*: *nrows-iarray* (*IArray* (*map* (λ*x*. *vec-to-iarray* (*A* $ *from-nat x*)) [*0*..<*CARD*(*'c*)]))
= *CARD*(*'c*) **unfolding** *nrows-iarray-def* **by** *auto*

have *n-rw2*: *nrows-iarray* (*IArray* (*map* (*λx*. *IArray* (*map* (*λi*. *A* $ *from-nat* *x* $ *from-nat* *i*)
[*0*..<*CARD*('*b*)])) [*0*..<*CARD*('*c*)])) = *CARD*('*c*) **unfolding** *nrows-iarray-def* **by** *auto*
have *n-rw3*: *ncols-iarray* (*IArray* (*map* (*λx*. *IArray* (*map* (*λi*. *A* $ *from-nat* *x* $ *from-nat* *i*)
[*0*..<*CARD*('*b*)])) [*0*..<*CARD*('*c*)])) = *CARD*('*b*)
unfolding *ncols-iarray-def* **by** *auto*
show *?thesis*
unfolding *matrix-to-iarray-def o-def matrix-scalar-mult-def scalar-matrix-mult-iarrays-def tabulate2-def vec-to-iarray-def*
by (*simp add*: *n-rw n-rw2 n-rw3*)
qed

lemma *matrix-to-iarray-minorM*[*code-unfold*]:
matrix-to-iarray (*minorM* *A* *i* *j*) = *minorM-iarrays* (*matrix-to-iarray* *A*) (*to-nat* *i*) (*to-nat* *j*)
proof −
have *n-rw*: *nrows-iarray* (*IArray* (*map* (*λx*. *IArray* (*map* (*λi*. *A* $ *from-nat* *x* $ *from-nat* *i*)
[*0*..<*CARD*('*b*)])) [*0*..<*CARD*('*c*)])) = *CARD*('*c*)
unfolding *nrows-iarray-def* **by** *auto*
have *n-rw2*: *ncols-iarray* (*IArray* (*map* (*λx*. *IArray* (*map* (*λi*. *A* $ *from-nat* *x* $ *from-nat* *i*)
[*0*..<*CARD*('*b*)])) [*0*..<*CARD*('*c*)])) = *CARD*('*b*)
unfolding *ncols-iarray-def* **by** *simp*
show *?thesis*
unfolding *matrix-to-iarray-def o-def*
minorM-def minorM-iarrays-def
tabulate2-def vec-to-iarray-def
by (*auto simp add*: *n-rw n-rw2 to-nat-from-nat-id*)
qed

lemma *matrix-to-iarray-cofactor*[*code-unfold*]:
(*cofactor* *A* *i* *j*) = *cofactor-iarrays* (*matrix-to-iarray* *A*) (*to-nat* *i*) (*to-nat* *j*)
unfolding *o-def cofactor-iarrays-def cofactor-def cofactorM-def*
unfolding *matrix-to-iarray-minorM*[*symmetric*]
unfolding *matrix-to-iarray-det-euclidean-ring*[*symmetric*] **by** *simp*

lemma *matrix-to-iarray-cofactorM*[*code-unfold*]:
matrix-to-iarray (*cofactorM* *A*) = *cofactorM-iarrays* (*matrix-to-iarray* *A*)
proof −
have *n-rw*: *nrows-iarray* (*IArray* (*map* (*λx*. *IArray* (*map* (*λi*. *A* $ *from-nat* *x* $ *from-nat* *i*)
[*0*..<*CARD*('*b*)])) [*0*..<*CARD*('*b*)])) = *CARD*('*b*)
unfolding *nrows-iarray-def* **by** *simp*
show *?thesis*
unfolding *cofactorM-iarrays-def tabulate2-def cofactorM-def*
by (*auto simp add*: *n-rw matrix-to-iarray-cofactor*

matrix-to-iarray-def o-def vec-to-iarray-def to-nat-from-nat-id)
qed

lemma *matrix-to-iarray-adjugate*[*code-unfold*]:
matrix-to-iarray (*adjugate A*) = *adjugate-iarrays* (*matrix-to-iarray A*)
unfolding *adjugate-def adjugate-iarrays-def*
unfolding *matrix-to-iarray-cofactorM*[*symmetric*]
unfolding *matrix-to-iarray-transpose*[*symmetric*] **..**

end

10 Inverse matrices over principal ideal rings using immutable arrays

theory *Echelon-Form-Inverse-IArrays*
imports
Echelon-Form-Inverse
Code-Cayley-Hamilton-IArrays
Gauss-Jordan.Inverse-IArrays
begin

10.1 Computing the inverse of matrices over rings using immutable arrays

definition *inverse-matrix-ring-iarray A* = (*let d*=*det-iarrays-rings A in*
if is-unit d then Some(*1 div d* ∗*ssi adjugate-iarrays A*) *else None*)

lemma *matrix-to-iarray-inverse*:
fixes A::′*a*::{*euclidean-ring-gcd*}^′*n*::{*mod-type*}^′*n*::{*mod-type*}
shows *matrix-to-iarray-option* (*inverse-matrix A*) = *inverse-matrix-ring-iarray* (*matrix-to-iarray A*)
unfolding *inverse-matrix-ring-iarray-def inverse-matrix-code-rings matrix-to-iarray-option-def*
unfolding *matrix-to-iarray-det-euclidean-ring matrix-to-iarray-adjugate*
by (*simp add*: *matrix-to-iarray-adjugate matrix-to-iarray-scalar-matrix-mult Let-def*)

end

11 Examples of computations using immutable arrays

theory *Examples-Echelon-Form-IArrays*
imports
Echelon-Form-Inverse-IArrays
HOL−*Library.Code-Target-Numeral*
Gauss-Jordan.Examples-Gauss-Jordan-Abstract
Examples-Echelon-Form-Abstract
begin

The file `Examples_Echelon_Form_Abstract.thy` is only imported to include the definitions of matrices that we use in the following examples. Otherwise, it could be removed.

11.1 Computing echelon forms, determinants, characteristic polynomials and so on using immutable arrays

11.1.1 Serializing gcd

First of all, we serialize the gcd to the ones of PolyML and MLton as we did in the Gauss-Jordan development.

context
includes *integer.lifting*
begin

lift-definition *gcd-integer* :: *integer => integer => integer*
 is *gcd* :: *int => int => int* **.**

lemma *gcd-integer-code* [*code*]:
gcd-integer l k = |*if l* = (*0*::*integer*) *then k else gcd-integer l* (|*k*| *mod* |*l*|)|
 by *transfer* (*simp add*: *gcd-code-int* [*symmetric*] *ac-simps*)

end

code-printing
 constant *abs* :: *integer* => - ⇀ (*SML*) *IntInf.abs*
 | **constant** *gcd-integer* :: *integer* => - => - ⇀ (*SML*) (*PolyML.IntInf.gcd* ((-),(-)))

lemma *gcd-code* [*code*]:
 gcd a b = *int-of-integer* (*gcd-integer* (*of-int a*) (*of-int b*))
 by (*metis gcd-integer.abs-eq int-of-integer-integer-of-int integer-of-int-eq-of-int*)

code-printing
 constant *abs* :: *real* => *real* ⇀
 (*SML*) *Real.abs*

declare [[*code drop*: *abs* :: *real* ⇒ *real*]]

code-printing
constant *divmod-integer* :: *integer* => - => - ⇀ (*SML*) (*IntInf.divMod* ((-),(-)))

11.1.2 Examples

value *det test-int-3x3*

value *det test-int-3x3-03*

value *det test-int-6x6*

value *det test-int-8x8*

value *det test-int-20x20*

value *charpoly test-real-3x3*

value *charpoly test-real-6x6*

value *inverse-matrix test-int-3x3-02*

value *matrix-to-iarray* (*echelon-form-of test-int-3x3 euclid-ext2*)

value *matrix-to-iarray* (*echelon-form-of test-int-8x8 euclid-ext2*)

The following computations are much faster when code is exported.

The following matrix will have an integer inverse since its determinant is equal to one

value *det test-int-3x3-03*

value *the* (*matrix-to-iarray-option* (*inverse-matrix test-int-3x3-03*))

We check that the previous inverse has been correctly computed:

value *matrix-matrix-mult-iarray*
  (*matrix-to-iarray test-int-3x3-03*)
  (*the* (*matrix-to-iarray-option* (*inverse-matrix test-int-3x3-03*)))

value *matrix-matrix-mult-iarray*
  (*the* (*matrix-to-iarray-option* (*inverse-matrix test-int-3x3-03*)))
  (*matrix-to-iarray test-int-3x3-03*)

The following matrices have determinant different from zero, and thus do not have an integer inverse

value *det test-int-6x6*

value *matrix-to-iarray-option* (*inverse-matrix test-int-6x6*)

value *det test-int-20x20*

value *matrix-to-iarray-option* (*inverse-matrix test-int-20x20*)

The inverse in dimension 20 has (trivial) inverse.

value *the* (*matrix-to-iarray-option* (*inverse-matrix* (*mat 1*::*int^20^20*)))

value *the* (*matrix-to-iarray-option* (*inverse-matrix* (*mat 1*::*int^20^20*))) = *matrix-to-iarray* (*mat 1*::*int^20^20*)

definition *print-echelon-int* $(A::int\hat{}20\hat{}20)$ = *echelon-form-of-iarrays* (*matrix-to-iarray* *A*) *euclid-ext2*

Performance is better when code is exported. In addition, it depends on the growth of the integer coefficients of the matrices. For instance, *test-int-20x20* is a matrix of integer numbers between -10 and 10. The computation of its echelon form (by means of *print-echelon-int*) needs about 2 seconds. However, the matrix *test-int-20x20-2* has elements between 0 and 1010. The computation of its echelon form (by means of *print-echelon-int* too) needs about 0.310 seconds. These benchmarks have been carried out in a laptop with an i5-3360M processor with 4 GB of RAM.

export-code *charpoly det echelon-form-of test-int-8x8 test-int-20x20 test-int-20x20-2 print-echelon-int*
in *SML* **module-name** *Echelon*

end