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 $\forall n \leq m.\ I(n) \subseteq I(m)$
⟨*proof*⟩

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* (λ*x*. *f x* + *g x*) (*A* ∩ *B*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *sum-distrib-right*:
sum f A ∗ *r* = (∑ *n*∈*A*. *f n* ∗ *r*)
⟨*proof*⟩

end

context *comm-monoid-add*
begin

lemma *sum-two-elements*:
assumes *a* ≠ *b*
shows *sum f* {*a*,*b*} = *f a* + *f b*
⟨*proof*⟩

lemma *sum-singleton*: *sum f* {*x*} = *f x*
⟨*proof*⟩

end

1.2 Subgroups

context *group-add*
begin
definition *subgroup A* ≡ (*0* ∈ *A* ∧ (∀ *a*∈*A*. ∀ *b* ∈ *A*. *a* + *b* ∈ *A*) ∧ (∀ *a*∈*A*. −*a* ∈ *A*))

lemma *subgroup-0*: *subgroup* {*0*}
⟨*proof*⟩

lemma *subgroup-UNIV*: *subgroup* (*UNIV*)
⟨*proof*⟩

lemma *subgroup-inter*:
assumes *subgroup A* **and** *subgroup B*
shows *subgroup* (*A* ∩ *B*)
⟨*proof*⟩

lemma *subgroup-Inter*:
assumes ∀ *I*∈*S*. *subgroup I*

shows *subgroup* $(\bigcap S)$
⟨*proof*⟩

lemma *subgroup-Union*:
fixes *I*::*nat* => *'a set*
defines *S*: $S \equiv \{I\ n | n.\ n \in UNIV\}$
assumes *all-subgroup*: $\forall A \in S.$ *subgroup* A
and *inc*: $\forall n.\ I(n) \subseteq I(n+1)$
shows *subgroup* $(\bigcup S)$
⟨*proof*⟩

end

1.3 Ideals

context *Rings.ring*
begin

lemma *subgroup-left-principal-ideal*: *subgroup* $\{r*a | r.\ r \in UNIV\}$
⟨*proof*⟩

definition *left-ideal* I = (*subgroup* $I \wedge (\forall x \in I.\ \forall r.\ r*x \in I)$)
definition *right-ideal* I = (*subgroup* $I \wedge (\forall x \in I.\ \forall r.\ x*r \in I)$)
definition *ideal* I= (*left-ideal* $I \wedge$ *right-ideal* I)

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

definition *left-principal-ideal* S = $(\exists a.$ *left-ideal-generated* $\{a\} = S)$
definition *right-principal-ideal* S = (*right-ideal* $S \wedge (\exists a.$ *right-ideal-generated* $\{a\}$ $= S))$
definition *principal-ideal* $S = (\exists a.$ *ideal-generated* $\{a\} = S)$

lemma *ideal-inter*:
assumes *ideal* I **and** *ideal* J **shows** *ideal* $(I \cap J)$
⟨*proof*⟩

lemma *ideal-Inter*:
assumes $\forall I \in S.$ *ideal* I
shows *ideal* $(\bigcap S)$
⟨*proof*⟩

lemma *ideal-Union*:
fixes *I*::*nat* => *'a set*

defines S: $S \equiv \{I\ n | n.\ n \in UNIV\}$
assumes *all-ideal*: $\forall A \in S.\ ideal\ A$
and *inc*: $\forall n.\ I(n) \subseteq I(n+1)$
shows *ideal* $(\bigcup S)$
⟨*proof*⟩

lemma *ideal-not-empty*:
assumes *ideal I*
shows $I \neq \{\}$
⟨*proof*⟩

lemma *ideal-0*: *ideal* $\{0\}$
⟨*proof*⟩

lemma *ideal-UNIV*: *ideal UNIV*
⟨*proof*⟩

lemma *ideal-generated-0*: *ideal-generated* $\{0\} = \{0\}$
⟨*proof*⟩

lemma *ideal-generated-subset-generator*:
assumes *ideal-generated* $A = I$
shows $A \subseteq I$
⟨*proof*⟩

lemma *left-ideal-minus*:
assumes *left-ideal I*
and $a \in I$ **and** $b \in I$
shows $a - b \in I$
⟨*proof*⟩

lemma *right-ideal-minus*:
assumes *right-ideal I*
and $a \in I$ **and** $b \in I$
shows $a - b \in I$
⟨*proof*⟩

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

lemma *ideal-ideal-generated*: *ideal* (*ideal-generated S*)
⟨*proof*⟩

lemma *sum-left-ideal*:
  assumes *li-X*: *left-ideal X*
  and *U-X*: $U \subseteq X$ **and** *U*: *finite U*
  shows $(\sum i{\in}U.\ f\ i * i) \in X$
  ⟨*proof*⟩

lemma *sum-right-ideal*:
  assumes *li-X*: *right-ideal X*
  and *U-X*: $U \subseteq X$ **and** *U*: *finite U*
  shows $(\sum i{\in}U.\ i * f\ i) \in X$
  ⟨*proof*⟩

lemma *left-ideal-generated-subset*:
  assumes $S \subseteq T$
  shows *left-ideal-generated S* ⊆ *left-ideal-generated T*
  ⟨*proof*⟩

lemma *right-ideal-generated-subset*:
  assumes $S \subseteq T$
  shows *right-ideal-generated S* ⊆ *right-ideal-generated T*
  ⟨*proof*⟩

lemma *ideal-generated-subset*:
  assumes $S \subseteq T$
  shows *ideal-generated S* ⊆ *ideal-generated T*
  ⟨*proof*⟩

lemma *ideal-generated-in*:
  assumes $a \in A$
  shows *a* ∈ *ideal-generated A*
  ⟨*proof*⟩

lemma *ideal-generated-repeated*: *ideal-generated* {*a*,*a*} = *ideal-generated* {*a*}
  ⟨*proof*⟩

end

context *ring-1*
begin

lemma *left-ideal-explicit*:
  left-ideal-generated S = {*y*. ∃*f U*. *finite U* ∧ *U* ⊆ *S* ∧ *sum* (λ*i*. *f i* ∗ *i*) *U* = *y*}
(**is** *?S* = *?B*)
⟨*proof*⟩

lemma *right-ideal-explicit*:
  right-ideal-generated S = {*y*. ∃*f U*. *finite U* ∧ *U* ⊆ *S* ∧ *sum* (λ*i*. *i* ∗ *f i*) *U* =
y} (**is** *?S* = *?B*)

⟨*proof*⟩

end

context *comm-ring*
begin

lemma *left-ideal-eq-right-ideal*: *left-ideal I* = *right-ideal I*
⟨*proof*⟩

corollary *ideal-eq-left-ideal*: *ideal I* = *left-ideal I*
⟨*proof*⟩

lemma *ideal-eq-right-ideal*: *ideal I* = *right-ideal I*
⟨*proof*⟩

lemma *principal-ideal-eq-left*:
principal-ideal S = (∃ *a*. *left-ideal-generated* {*a*} = *S*)
⟨*proof*⟩

end

context *comm-ring-1*
begin

lemma *ideal-generated-eq-left-ideal*: *ideal-generated A* = *left-ideal-generated A*
⟨*proof*⟩

lemma *ideal-generated-eq-right-ideal*: *ideal-generated A* = *right-ideal-generated A*
⟨*proof*⟩

lemma *obtain-sum-ideal-generated*:
assumes *a*: *a* ∈ *ideal-generated A* **and** *A*: *finite A*
obtains *f* **where** *sum* (λ*i*. *f i* ∗ *i*) *A* = *a*
⟨*proof*⟩

lemma *dvd-ideal-generated-singleton*:
assumes *subset*: *ideal-generated* {*a*} ⊆ *ideal-generated* {*b*}
shows *b dvd a*
⟨*proof*⟩

lemma *ideal-generated-singleton*: *ideal-generated* {*a*} = {*k*∗*a*|*k*. *k* ∈ *UNIV*}
⟨*proof*⟩

lemma *dvd-ideal-generated-singleton′*:
assumes *b-dvd-a*: *b dvd a*
shows *ideal-generated* {*a*} ⊆ *ideal-generated* {*b*}

⟨*proof*⟩

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*⟩

end

lemma *ideal-kZ*: *ideal* $\{k*x|x.\ x\in(UNIV::int\ set)\}$
⟨*proof*⟩

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** $gcd'\ a\ b\ dvd\ a$ ⟨*proof*⟩

lemma *gcd'-dvd2*:
assumes *is-gcd gcd'* **shows** $gcd'\ a\ b\ dvd\ b$
⟨*proof*⟩

lemma *gcd'-greatest*:
assumes *is-gcd gcd'* **and** $l\ dvd\ a$ **and** $l\ dvd\ b$
shows $l\ dvd\ gcd'\ a\ b$
⟨*proof*⟩

lemma *gcd'-zero* [*simp*]:
assumes *is-gcd gcd'*

shows *gcd′ x y = 0 ⟷ x = 0 ∧ y = 0*
⟨*proof*⟩

end

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

class *bezout-ring = comm-ring-1 +*
    assumes *exists-bezout*: *∃ p q d. (p∗a + q∗b = d)*
        ∧ (d dvd a)
        ∧ (d dvd b)
        ∧ (∀ d′. (d′ dvd a ∧ d′ dvd b) ⟶ d′ dvd d)
begin

subclass *GCD-ring*
⟨*proof*⟩

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 ⇒′a ⇒ (′a × ′a × ′a)) ⇒ bool*
  where *is-bezout (bezout) = (∀ a b. let (p, q, 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))

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 ⇒′a ⇒ (′a × ′a × ′a × ′a × ′a)) ⇒ bool*
  where *is-bezout-ext (bezout) = (∀ 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)

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′) ⇒ gcd′)*
  ⟨*proof*⟩

lemma *is-bezout-ext-is-bezout*:
assumes *is-bezout-ext bezout*
shows *is-bezout* ($\lambda a\ b.$ *case bezout a b of* $(x,xa,u,v,\ gcd') \Rightarrow (x,xa,gcd')$)
⟨*proof*⟩

lemma *is-gcd-is-bezout*:
assumes *is-bezout bezout*
shows *is-gcd* ($\lambda a\ b.$ (*case bezout a b of* (-, -, gcd') $\Rightarrow$ (gcd')))
⟨*proof*⟩

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*⟩

lemma *exists-bezout-ext*: $\exists$ *bezout-ext*. *is-bezout-ext bezout-ext*
⟨*proof*⟩

end

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

subclass (**in** *bezout-domain*) *GCD-domain*
⟨*proof*⟩

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*⟩

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*⟩

subclass (**in** *pid*) *bezout-domain*

⟨*proof*⟩

context *pir*
begin

lemma *ascending-chain-condition*:
  fixes *I*::*nat*=>′*a set*
  assumes *all-ideal*: $\forall n.\ ideal\ (I(n))$
  and *inc*: $\forall n.\ I(n) \subseteq I(n+1)$
  shows $\exists n.\ I(n)=I(n+1)$
⟨*proof*⟩

lemma *ascending-chain-condition2*:
  $\nexists I::(nat \Rightarrow 'a\ set).\ (\forall n.\ ideal\ (I\ n) \wedge I\ n \subset I\ (n + 1))$
⟨*proof*⟩

end

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

subclass (**in** *pir-div*) *pid-div*
⟨*proof*⟩

subclass (**in** *pir-div*) *bezout-ring-div*
⟨*proof*⟩

subclass (**in** *pid-div*) *bezout-domain-div*
⟨*proof*⟩

1.6 Euclidean Domains

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

subclass (**in** *euclidean-ring*) *pid*
⟨*proof*⟩

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*⟩

lemma *is-bezout-euclid-ext*: *is-bezout* (λ*a b*. (*fst* (*bezout-coefficients a b*), *snd* (*bezout-coefficients a b*), *gcd a b*))
⟨*proof*⟩

end

subclass (**in** *euclidean-ring*) *pid-div* ⟨*proof*⟩

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* ⟨*proof*⟩

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* = (λ*i*. *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*
⟨*proof*⟩

lemma *to-vec-matrix-scalar-mult*: *to-vec* ($x *_S A$) = *x* ∗*k to-vec A*
⟨*proof*⟩

lemma *to-vec-matrix-matrix-mult*: *to-vec* (*A* ∗ *B*) = *to-vec A* ∗∗ *to-vec B*
⟨*proof*⟩

lemma *to-vec-diag*: *to-vec* (*diag x*) = *mat x*
⟨*proof*⟩

lemma *to-vec-one*: *to-vec 1* = *mat 1*
⟨*proof*⟩

lemma *to-vec-eq-iff*: *to-vec M* = *to-vec N* ⟷ *M* = *N*
⟨*proof*⟩

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*⟩

definition *minorM M i j* = (χ *k l*. *if k* = *i* ∧ *l* = *j then 1 else if k* = *i* ∨ *l* = *j then 0 else M* $ *k* $ *l*)

lemma *minorM-eq*: *minorM M i j* = *to-vec* (*minor* (*from-vec M*) *i j*)
⟨*proof*⟩

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* ∘ *Square-Matrix.cofactor* ∘ *from-vec*
⟨*proof*⟩

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*)
⟨*proof*⟩

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

lemma *adjugate-eq*: *adjugate* = *to-vec* ∘ *Square-Matrix.adjugate* ∘ *from-vec*
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *scalar-matrix-mult-code* [*code abstract*]: *vec-nth* (*c* ∗*k A*) = *scalar-matrix-mult-row c A*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *minorM-code* [*code abstract*]: *vec-nth* (*minorM A i j*) = *minorM-row A i j*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *cofactorM-code* [*code abstract*]: *vec-nth* (*cofactorM A*) = *cofactorM-row A*
⟨*proof*⟩

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* :])
⟨*proof*⟩

lemma [*code abstract*]: *vec-nth* (*mat2matofpoly k*) = *mat2matofpoly-row k*
⟨*proof*⟩

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 = ($\sum$ *i* ∈ { *n*::*nat* . *n* ≤ (*degree P*) } . (*coeff P i*) ∗*k* (*matpow A i*))

lemma *evalmat-unfold*:
evalmat P A = ($\sum$ *i* = *0*..*degree P*. *coeff P i* ∗*k matpow A i*)
⟨*proof*⟩

lemma *evalmat-code*[*code*]:
evalmat P A = ($\sum$ *i*←[*0*..*int* (*degree P*)]. *coeff P* (*nat i*) ∗*k matpow A* (*nat i*)) (**is** - = *?rhs*)
⟨*proof*⟩

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*))
⟨*proof*⟩

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

A
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *coeffM-code* [*code abstract*]: *vec-nth* (*coeffM A n*) = *coeffM-row A n*
⟨*proof*⟩

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*}^*'cols*::{*mod-type*}^*'rows*::{*finite*, *ord*} ⇒ *nat* ⇒ *bool*
where
echelon-form-upt-k A k = (
(∀ *i*. *is-zero-row-upt-k i k A*
⟶ ¬ (∃ *j*. *j*>*i* ∧ ¬ *is-zero-row-upt-k j k A*))
∧
(∀ *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*))))

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 (∀ *i*. *is-zero-row-upt-k i k A* ⟶ ¬ (∃ *j*. *j*>*i* ∧ ¬ *is-zero-row-upt-k j k A*))
and (∀ *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*)))
shows *echelon-form-upt-k A k* ⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *echelon-form-upt-k-0*: *echelon-form-upt-k A 0*
⟨*proof*⟩

lemma *echelon-form-condition1*:
assumes *r*: *echelon-form A*
shows (∀ *i*. *is-zero-row i A* ⟶ ¬ (∃ *j*. *j*>*i* ∧ ¬ *is-zero-row j A*))
⟨*proof*⟩

lemma *echelon-form-condition2*:
assumes *r*: *echelon-form A*
shows (∀ *i*. *i*<*j* ∧ ¬ (*is-zero-row i A*) ∧ ¬ (*is-zero-row j A*)
⟶ ((*LEAST n*. *A* $ *i* $ *n* ≠ *0*) < (*LEAST n*. *A* $ *j* $ *n* ≠ *0*)))
⟨*proof*⟩

lemma *echelon-form-condition2-explicit*:
assumes *rref-A*: *echelon-form A*
and *i-le*: *i* < *j*
and ¬ *is-zero-row i A* **and** ¬ *is-zero-row j A*
shows (*LEAST n*. *A* $ *i* $ *n* ≠ *0*) < (*LEAST n*. *A* $ *j* $ *n* ≠ *0*)
⟨*proof*⟩

lemma *echelon-form-intro*:
assumes *1*: (∀ *i*. *is-zero-row i A* ⟶ ¬ (∃ *j*. *j*>*i* ∧ ¬ *is-zero-row j A*))
and *2*: (∀ *i j*. *i*<*j* ∧ ¬ (*is-zero-row i A*) ∧ ¬ (*is-zero-row j A*)
⟶ ((*LEAST n*. *A* $ *i* $ *n* ≠ *0*) < (*LEAST n*. *A* $ *j* $ *n* ≠ *0*)))

shows *echelon-form A*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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 $(GREATEST\ m.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A) < i$
⟨*proof*⟩

lemma *greatest-ge-nonzero-row′*:
fixes $A::'a::\{bezout\text{-}ring\}^{\frown}'cols::\{mod\text{-}type\}^{\frown}'rows::\{mod\text{-}type\}$
assumes *r*: *echelon-form-upt-k A k*
and *i*: $i \leq (GREATEST\ m.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A)$
and *not-all-zero*: $\neg\ (\forall\ a.\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ a\ k\ A)$
shows $\neg$ *is-zero-row-upt-k i k A*
⟨*proof*⟩

lemma *rref-imp-ef*:
fixes $A::'a::\{bezout\text{-}ring\}^{\frown}'cols::\{mod\text{-}type\}^{\frown}'rows::\{mod\text{-}type\}$
assumes *rref*: *reduced-row-echelon-form A*
shows *echelon-form A*
⟨*proof*⟩

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: $gcd\text{-}bezout\text{-}ring\ a\ b = (SOME\ d.\ d\ dvd\ a \wedge d\ dvd\ b \wedge (\forall\ d'.\ d'\ dvd\ a \wedge d'\ dvd\ b \longrightarrow d'\ dvd\ d))$

   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 $(gcd\text{-}bezout\text{-}ring\ a\ b) = gcd\text{-}eucl\ a\ b$ (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* $gcd\text{-}bezout\text{-}ring\text{-}norm\ a\ b = gcd\text{-}bezout\text{-}ring\ a\ b$ *div normalisation-factor* $(gcd\text{-}bezout\text{-}ring\ a\ b)$

   Then, one could demonstrate a (code) lemma stating that: $(gcd\text{-}bezout\text{-}ring\text{-}norm\ a\ b) = gcd\text{-}eucl\ a\ b$ 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^'cols^'rows ⇒ 'rows ⇒ 'rows ⇒ 'cols*
⇒ (*'a ⇒ 'a ⇒* (*'a × 'a × 'a × 'a × 'a*)) ⇒ *'a^'rows^'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 ∧ 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))

end

primrec
bezout-iterate :: *'a::{bezout-ring}^'cols^'rows::{mod-type}*
⇒ *nat* ⇒ *'rows::{mod-type}*
⇒ *'cols* ⇒ (*'a ⇒'a ⇒* (*'a × 'a × 'a × 'a × 'a*)) ⇒
'a^'cols^'rows::{mod-type}
where *bezout-iterate A 0 i j bezout* = *A*
| *bezout-iterate A* (*Suc n*) *i j bezout* =
(*if* (*Suc n*) ≤ *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*⟩

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*⟩

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*⟩

lemma *bezout-matrix-works2*:
fixes *A*::*′a*::*bezout-domain*^*′cols*^*′rows*

assumes *ib*: *is-bezout-ext bezout*
and *a-not-b*: $a \neq b$
and *not-0*: $A \ \$ \ a \ \$ \ j \neq 0 \vee A \ \$ \ b \ \$ \ j \neq 0$
shows (*bezout-matrix A a b j bezout* ** *A*) $ *b* $ *j* = *0*
⟨*proof*⟩

lemma *bezout-matrix-preserves-previous-columns*:
assumes *ib*: *is-bezout-ext bezout*
and *i-not-j*: $i \neq j$
and *Aik*: $A \ \$ \ i \ \$ \ k \neq 0$
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*
⟨*proof*⟩

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*: $A \ \$ \ a \ \$ \ j \neq 0$
shows *det* (*bezout-matrix A a b j bezout*) = *1*
⟨*proof*⟩

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*: $A \ \$ \ a \ \$ \ j \neq 0$
shows *invertible* (*bezout-matrix A a b j bezout*)
⟨*proof*⟩

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*⟩

lemma *bezout-matrix-preserves-rest*:
assumes *ib*: *is-bezout-ext bezout*
and *a-not-n*: $a \neq n$
and *i-not-n*: $i \neq n$
and *a-not-i*: $a \neq i$
and *Aik-0*: $A \ \$ \ i \ \$ \ k \neq 0$
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*
⟨*proof*⟩

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 (λ*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*))

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
(λ*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*)) ⟨*proof*⟩

lemma [*code abstract*]: *vec-nth* (*bezout-matrix A a b j bezout*) = *bezout-matrix-row A a b j bezout*
⟨*proof*⟩

3.2.5 Properties of the bezout iterate function

lemma *bezout-iterate-not-zero*:
assumes *Aik-0*: *A* $ *i* $ *from-nat k* ≠ *0*
and *n*: *n*<*nrows A*
and *a*: *to-nat i* ≤ *n*
and *ib*: *is-bezout-ext bezout*
shows *bezout-iterate A n i* (*from-nat k*) *bezout* $ *i* $ *from-nat k* ≠ *0*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

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*: ∀ *m*>*from-nat* (*nrows A*). *A* $ *m* $ *from-nat k* = *0*
shows *is-zero-row-upt-k b* (*Suc k*) *A*
⟨*proof*⟩

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*: ∀ *m*>*from-nat i*. *A* $ *m* $ *from-nat k* = *0*
and *i-not-last*: *i* ≠ *nrows A*
and *i-le-m*: *from-nat i* ≤ *m*
and *Amk-not-0*: *A* $ *m* $ *from-nat k* ≠ *0*
shows *is-zero-row-upt-k b* (*Suc k*) *A*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

lemma *echelon-echelon-form-column-k*:
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* **and** *k*: *k*<*ncols*
A
shows *echelon-form-upt-k B* (*Suc k*)
⟨*proof*⟩

lemma *echelon-foldl-condition1*:
assumes *ib*: *is-bezout-ext bezout*
and *A* $ *ma* $ *from-nat* (*Suc k*) ≠ *0*
and *k*: *k*<*ncols A*
shows ∃ *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-nat* (*Suc k*)) *bezout*)
⟨*proof*⟩

lemma *echelon-foldl-condition2*:
  fixes *A*::′*a*::{*bezout-ring*}^′*cols*::{*mod-type*}^′*rows*::{*mod-type*}
  assumes *n*: ¬ *is-zero-row-upt-k ma k A*
  and *all-zero*: ∀ *m*≥ (*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*⟩

lemma *echelon-foldl-condition3*:
  fixes *A*::′*a*::{*bezout-domain*}^′*cols*::{*mod-type*}^′*rows*::{*mod-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*⟩

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*⟩

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*))
  ⟨*proof*⟩

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*⟩

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*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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 ∃*P*. *invertible P* ∧ *P*∗∗*A* = (*echelon-form-of-upt-k A k bezout*)
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
 ⟨*proof*⟩

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*
⟨*proof*⟩

4.2.2 Echelon Form of column k

lemma *det-echelon-form-of-column-k-det*:
fixes *A*::′*a*::{*bezout-domain*}^′*n*::{*mod-type*}^′*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*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*⟩

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*))
⟨*proof*⟩

4.2.6 Final lemmas

corollary *det-echelon-form-of-det'*:
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-form-of-det A bezout*))
∗ *det* (*snd* (*echelon-form-of-det A bezout*))
⟨*proof*⟩

lemma *ef-echelon-form-of-det*:
fixes $A::'a::\{bezout\text{-}domain\}^{\frown}'n::\{mod\text{-}type\}^{\frown}'n::\{mod\text{-}type\}$
assumes *ib*: *is-bezout-ext bezout*
shows *echelon-form* (*snd* (*echelon-form-of-det A bezout*))
⟨*proof*⟩

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* (λ*i*. *A* $ *i* $ *i*) (*UNIV*:: '*n set*)
⟨*proof*⟩

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-form-of-det A bezout*))
∗ *prod* (λ*i*. *snd* (*echelon-form-of-det A bezout*) $ *i* $ *i*) (*UNIV*:: '*n set*)
⟨*proof*⟩

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-form-of-det A euclid-ext2*))
∗ *prod* (λ*i*. *snd* (*echelon-form-of-det A euclid-ext2*) $ *i* $ *i*) (*UNIV*:: '*n set*)
⟨*proof*⟩

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-semiring-0*

shows *x* ∗*k mat y* = *mat* (*x* ∗ *y*)
⟨*proof*⟩

lemma *matrix-mul-mat*:
fixes *A* :: *'a*::*comm-semiring-1* ^ *'m* ^ *'n*
shows *A* ∗∗ *mat x* = *x* ∗*k A*
⟨*proof*⟩

lemma *mult-adjugate-det*: *A* ∗∗ *adjugate A* = *mat* (*det A*)
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

corollary *fst-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*⟩

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*⟩

lemma *foldl-echelon-form-column-k-eq*:
fixes *A*::'*a*::{*bezout-ring*}^'*cols*::{*mod-type*}^'*rows*::{*mod-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*
⟨*proof*⟩

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*
⟨*proof*⟩

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-ring*} × *'a iarray iarray* × *nat* × (*'a* ⇒ *'a* ⇒ *'a* × *'a* × *'a* × *'a* × *'a*)
  ⇒ *nat*
  ⇒ *'a* × *'a iarray iarray* × *nat* × (*'a* ⇒ *'a* ⇒ *'a* × *'a* × *'a* × *'a* × *'a*)
 where
 echelon-form-of-column-k-det-iarrays A' k =
  (*let* (*det-P*, *A*, *i*, *bezout*) = *A'*
   in if ((*i* ≠ *nrows-iarray A*) ∧ (*A* !! *i* !! *k* = *0*)
     ∧ (¬ *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-ring*} *iarray iarray*
  ⇒ (*'a* ⇒ *'a* ⇒ *'a* × *'a* × *'a* × *'a* × *'a*)
  ⇒ (*'a* × (*'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* (λ*i*. (*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*)))
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*)))
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*]))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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* (λ*i*. (*snd A*′) !! *i* !! *i*) [*0*..<*nrows-iarray* (*matrix-to-iarray A*)]))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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

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*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *matrix-to-iarray-cofactor[code-unfold]*:
(cofactor A i j) = cofactor-iarrays (matrix-to-iarray A) (to-nat i) (to-nat j)
⟨*proof*⟩

lemma *matrix-to-iarray-cofactorM[code-unfold]*:
matrix-to-iarray (cofactorM A) = cofactorM-iarrays (matrix-to-iarray A)
⟨*proof*⟩

lemma *matrix-to-iarray-adjugate[code-unfold]*:
matrix-to-iarray (adjugate A) = adjugate-iarrays (matrix-to-iarray A)
⟨*proof*⟩

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 $A::'a::\{euclidean\text{-}ring\text{-}gcd\}^{\frown}'n::\{mod\text{-}type\}^{\frown}'n::\{mod\text{-}type\}$

shows *matrix-to-iarray-option* (*inverse-matrix A*) = *inverse-matrix-ring-iarray* (*matrix-to-iarray A*)
⟨*proof*⟩

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* ⟨*proof*⟩

lemma *gcd-integer-code* [*code*]:
gcd-integer l k = |*if l* = (*0*::*integer*) *then k else gcd-integer l* (|*k*| *mod* |*l*|)|
⟨*proof*⟩

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*))
  ⟨*proof*⟩

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*^*20*^*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