Gauss-Jordan algorithm and its applications

By Jose Divasón and Jesús Aransay*

March 19, 2025

Abstract

In this contribution, we present a formalization of the well-known Gauss-Jordan algorithm. It states that any matrix over a field can be transformed by means of elementary row operations to a matrix in reduced row echelon form. The formalization is based on the Rank Nullity Theorem entry of the AFP and on the HOL-Multivariate-Analysis session of Isabelle, where matrices are represented as functions over finite types. We have set up properly the code generator to make this representation executable. In order to improve the performance, a refinement to immutable arrays has been carried out. We have formalized some of the applications of the Gauss-Jordan algorithm. Thanks to this development, the following facts can be computed over matrices whose elements belong to a field:

- Ranks
- Determinants
- Inverses
- Bases and dimensions of the null space, left null space, column space and row space of a matrix
- Solutions of systems of linear equations (considering any case, including systems with one solution, multiple solutions and with no solution)

Code can be exported to both SML and Haskell. In addition, we have introduced some serializations (for instance, from *bit* in Isabelle to booleans in SML and Haskell, and from *rat* in Isabelle to the corresponding one in Haskell), that speed up the performance.

Contents

*This research has been funded by the research grant FPIUR12 of the Universidad de La Rioja.

1 Reduced row echelon form

theory *Rref*
imports
Rank-Nullity-Theorem.Mod-Type
Rank-Nullity-Theorem.Miscellaneous
begin

1.1 Defining the concept of Reduced Row Echelon Form

1.1.1 Previous definitions and properties

This function returns True if each position lesser than k in a column contains a zero.

definition *is-zero-row-upt-k* :: *'rows* => *nat* =>*'a*::{*zero*}^*'columns*::{*mod-type*}^*'rows* => *bool*
where *is-zero-row-upt-k i k A* = (∀ *j*::*'columns*. (*to-nat j*) < *k* ⟶ *A* $ *i* $ *j* = *0*)

definition *is-zero-row* :: *'rows* =>*'a*::{*zero*}^*'columns*::{*mod-type*}^*'rows* => *bool*
where *is-zero-row i A* = *is-zero-row-upt-k i* (*ncols A*) *A*

lemma *is-zero-row-upt-ncols*:
fixes *A*::*'a*::{*zero*}^*'columns*::{*mod-type*}^*'rows*
shows *is-zero-row-upt-k i* (*ncols A*) *A* = (∀ *j*::*'columns*. *A* $ *i* $ *j* = *0*) ⟨*proof*⟩

corollary *is-zero-row-def'*:
fixes *A*::*'a*::{*zero*}^*'columns*::{*mod-type*}^*'rows*
shows *is-zero-row i A* = (∀ *j*::*'columns*. *A* $ *i* $ *j* = *0*) ⟨*proof*⟩

lemma *is-zero-row-eq-row-zero*: *is-zero-row a A* = (*row a A* = *0*)
⟨*proof*⟩

lemma *not-is-zero-row-upt-suc*:
assumes ¬ *is-zero-row-upt-k i* (*Suc k*) *A*
and ∀ *i*. *A* $ *i* $ (*from-nat k*) = *0*
shows ¬ *is-zero-row-upt-k i k A*
⟨*proof*⟩

lemma *is-zero-row-upt-k-suc*:
assumes *is-zero-row-upt-k i k A*
and *A* $ *i* $ (*from-nat k*) = *0*
shows *is-zero-row-upt-k i* (*Suc k*) *A*
⟨*proof*⟩

lemma *is-zero-row-utp-0*:

shows *is-zero-row-upt-k m 0 A* ⟨*proof*⟩

lemma *is-zero-row-utp-0'*:
shows ∀ *m*. *is-zero-row-upt-k m 0 A* ⟨*proof*⟩

lemma *is-zero-row-upt-k-le*:
assumes *is-zero-row-upt-k i* (*Suc k*) *A*
shows *is-zero-row-upt-k i k A*
⟨*proof*⟩

lemma *is-zero-row-imp-is-zero-row-upt*:
assumes *is-zero-row i A*
shows *is-zero-row-upt-k i k A*
⟨*proof*⟩

1.1.2 Definition of reduced row echelon form up to a column

This definition returns True if a matrix is in reduced row echelon form up to the column k (not included), otherwise False.

definition *reduced-row-echelon-form-upt-k* :: *'a*::{*zero*, *one*}^'*m*::{*mod-type*}^'*n*::{*finite*, *ord*, *plus*, *one*} => *nat* => *bool*
where *reduced-row-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*. ¬ (*is-zero-row-upt-k i k A*) ⟶ *A* $ *i* $ (*LEAST k*. *A* $ *i* $ *k* ≠ *0*) = *1*) ∧
(∀ *i*. *i*<*i*+*1* ∧ ¬ (*is-zero-row-upt-k i k A*) ∧ ¬ (*is-zero-row-upt-k* (*i*+*1*) *k A*)
⟶ ((*LEAST n*. *A* $ *i* $ *n* ≠ *0*) < (*LEAST n*. *A* $ (*i*+*1*) $ *n* ≠ *0*))) ∧
(∀ *i*. ¬ (*is-zero-row-upt-k i k A*) ⟶ (∀ *j*. *i* ≠ *j* ⟶ *A* $ *j* $ (*LEAST n*. *A* $ *i* $
n ≠ *0*) = *0*))
)

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

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

lemma *rref-upt-condition2*:
assumes *r*: *reduced-row-echelon-form-upt-k A k*
shows (∀ *i*. ¬ (*is-zero-row-upt-k i k A*) ⟶ *A* $ *i* $ (*LEAST k*. *A* $ *i* $ *k* ≠ *0*)
= *1*)
⟨*proof*⟩

lemma *rref-upt-condition3*:
assumes *r*: *reduced-row-echelon-form-upt-k A k*
shows (∀ *i*. *i*<*i*+*1* ∧ ¬ (*is-zero-row-upt-k i k A*) ∧ ¬ (*is-zero-row-upt-k* (*i*+*1*) *k*

$A) \longrightarrow ((LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0) < (LEAST\ n.\ A\ \$\ (i{+}1)\ \$\ n \neq 0)))$
⟨*proof*⟩

lemma *rref-upt-condition4*:
assumes *r*: *reduced-row-echelon-form-upt-k A k*
shows $(\forall i.\ \neg\ (is\text{-}zero\text{-}row\text{-}upt\text{-}k\ i\ k\ A) \longrightarrow (\forall j.\ i \neq j \longrightarrow A\ \$\ j\ \$\ (LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0) = 0))$
⟨*proof*⟩

Explicit lemmas for each condition

lemma *rref-upt-condition1-explicit*:
assumes *reduced-row-echelon-form-upt-k A k*
and *is-zero-row-upt-k i k A*
and $j{>}i$
shows *is-zero-row-upt-k j k A*
⟨*proof*⟩

lemma *rref-upt-condition2-explicit*:
assumes *rref-A*: *reduced-row-echelon-form-upt-k A k*
and ¬ *is-zero-row-upt-k i k A*
shows $A\ \$\ i\ \$\ (LEAST\ k.\ A\ \$\ i\ \$\ k \neq 0) = 1$
⟨*proof*⟩

lemma *rref-upt-condition3-explicit*:
assumes *reduced-row-echelon-form-upt-k A k*
and $i < i + 1$
and ¬ *is-zero-row-upt-k i k A*
and ¬ *is-zero-row-upt-k* $(i + 1)$ *k A*
shows $(LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0) < (LEAST\ n.\ A\ \$\ (i + 1)\ \$\ n \neq 0)$
⟨*proof*⟩

lemma *rref-upt-condition4-explicit*:
assumes *reduced-row-echelon-form-upt-k A k*
and ¬ *is-zero-row-upt-k i k A*
and $i \neq j$
shows $A\ \$\ j\ \$\ (LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0) = 0$
⟨*proof*⟩

Intro lemma and general properties

lemma *reduced-row-echelon-form-upt-k-intro*:
assumes $(\forall i.\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ i\ k\ A \longrightarrow \neg\ (\exists j.\ j{>}i \wedge \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ j\ k\ A))$
and $(\forall i.\ \neg\ (is\text{-}zero\text{-}row\text{-}upt\text{-}k\ i\ k\ A) \longrightarrow A\ \$\ i\ \$\ (LEAST\ k.\ A\ \$\ i\ \$\ k \neq 0) = 1)$
and $(\forall i.\ i{<}i{+}1 \wedge \neg\ (is\text{-}zero\text{-}row\text{-}upt\text{-}k\ i\ k\ A) \wedge \neg\ (is\text{-}zero\text{-}row\text{-}upt\text{-}k\ (i{+}1)\ k\ A) \longrightarrow ((LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0) < (LEAST\ n.\ A\ \$\ (i{+}1)\ \$\ n \neq 0)))$
and $(\forall i.\ \neg\ (is\text{-}zero\text{-}row\text{-}upt\text{-}k\ i\ k\ A) \longrightarrow (\forall j.\ i \neq j \longrightarrow A\ \$\ j\ \$\ (LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0) = 0))$
shows *reduced-row-echelon-form-upt-k A k*

⟨*proof*⟩

lemma *rref-suc-imp-rref*:
fixes *A*::*'a*::{*semiring-1*}^*'n*::{*mod-type*}^*'m*::{*mod-type*}
assumes *r*: *reduced-row-echelon-form-upt-k A* (*Suc k*)
and *k-le-card*: *Suc k* < *ncols A*
shows *reduced-row-echelon-form-upt-k A k*
⟨*proof*⟩

lemma *reduced-row-echelon-if-all-zero*:
assumes *all-zero*: ∀ *n*. *is-zero-row-upt-k n k A*
shows *reduced-row-echelon-form-upt-k A k*
⟨*proof*⟩

1.1.3 The definition of reduced row echelon form

Definition of reduced row echelon form, based on *reduced-row-echelon-form-upt-k-def*

definition *reduced-row-echelon-form* :: *'a*::{*zero*, *one*}^*'m*::{*mod-type*}^*'n*::{*finite*, *ord*, *plus*, *one*} => *bool*
where *reduced-row-echelon-form A* = *reduced-row-echelon-form-upt-k A* (*ncols A*)

Equivalence between our definition of reduced row echelon form and the one presented in Steven Roman's book: Advanced Linear Algebra.

lemma *reduced-row-echelon-form-def'*:
reduced-row-echelon-form A =
(
(∀ *i*. *is-zero-row i A* ⟶ ¬ (∃ *j*. *j*>*i* ∧ ¬ *is-zero-row j A*)) ∧
(∀ *i*. ¬ (*is-zero-row i A*) ⟶ *A* $ *i* $ (*LEAST k*. *A* $ *i* $ *k* ≠ *0*) = *1*) ∧
(∀ *i*. *i*<*i*+*1* ∧ ¬ (*is-zero-row i A*) ∧ ¬ (*is-zero-row* (*i*+*1*) *A*) ⟶ ((*LEAST k*. *A* $ *i* $ *k* ≠ *0*) < (*LEAST k*. *A* $ (*i*+*1*) $ *k* ≠ *0*))) ∧
(∀ *i*. ¬ (*is-zero-row i A*) ⟶ (∀ *j*. *i* ≠ *j* ⟶ *A* $ *j* $ (*LEAST k*. *A* $ *i* $ *k* ≠ *0*) = *0*))
) ⟨*proof*⟩

1.2 Properties of the reduced row echelon form of a matrix

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

lemma *rref-condition2*:
assumes *r*: *reduced-row-echelon-form A*
shows (∀ *i*. ¬ (*is-zero-row i A*) ⟶ *A* $ *i* $ (*LEAST k*. *A* $ *i* $ *k* ≠ *0*) = *1*)
⟨*proof*⟩

lemma *rref-condition3*:
assumes *r*: *reduced-row-echelon-form A*

shows $(\forall i.\ i<i+1 \wedge \neg\ (\textit{is-zero-row}\ i\ A) \wedge \neg\ (\textit{is-zero-row}\ (i+1)\ A) \longrightarrow ((\textit{LEAST}\ n.\ A\ \$\ i\ \$\ n \neq 0) < (\textit{LEAST}\ n.\ A\ \$\ (i+1)\ \$\ n \neq 0)))$
⟨*proof*⟩

lemma *rref-condition4*:
assumes *r*: *reduced-row-echelon-form A*
shows $(\forall i.\ \neg\ (\textit{is-zero-row}\ i\ A) \longrightarrow (\forall j.\ i \neq j \longrightarrow A\ \$\ j\ \$\ (\textit{LEAST}\ n.\ A\ \$\ i\ \$\ n \neq 0) = 0))$
⟨*proof*⟩

Explicit lemmas for each condition

lemma *rref-condition1-explicit*:
assumes *rref-A*: *reduced-row-echelon-form A*
and *is-zero-row i A*
shows $\forall j>i.$ *is-zero-row j A*
⟨*proof*⟩

lemma *rref-condition2-explicit*:
assumes *rref-A*: *reduced-row-echelon-form A*
and ¬ *is-zero-row i A*
shows $A\ \$\ i\ \$\ (\textit{LEAST}\ k.\ A\ \$\ i\ \$\ k \neq 0) = 1$
⟨*proof*⟩

lemma *rref-condition3-explicit*:
assumes *rref-A*: *reduced-row-echelon-form A*
and *i-le*: $i < i + 1$
and ¬ *is-zero-row i A* **and** ¬ *is-zero-row* $(i + 1)$ *A*
shows $(\textit{LEAST}\ n.\ A\ \$\ i\ \$\ n \neq 0) < (\textit{LEAST}\ n.\ A\ \$\ (i + 1)\ \$\ n \neq 0)$
⟨*proof*⟩

lemma *rref-condition4-explicit*:
assumes *rref-A*: *reduced-row-echelon-form A*
and ¬ *is-zero-row i A*
and $i \neq j$
shows $A\ \$\ j\ \$\ (\textit{LEAST}\ n.\ A\ \$\ i\ \$\ n \neq 0) = 0$
⟨*proof*⟩

Other properties and equivalences

lemma *rref-condition3-equiv1*:
fixes *A*::'*a*::{*one*, *zero*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
assumes *rref*: *reduced-row-echelon-form A*
and *i-less-j*: *i*<*j*
and *j-less-nrows*: *j*<*nrows A*
and *not-zero-i*: ¬ *is-zero-row* (*from-nat i*) *A*
and *not-zero-j*: ¬ *is-zero-row* (*from-nat j*) *A*
shows $(\textit{LEAST}\ n.\ A\ \$\ (\textit{from-nat}\ i)\ \$\ n \neq 0) < (\textit{LEAST}\ n.\ A\ \$\ (\textit{from-nat}\ j)\ \$\ n \neq 0)$
⟨*proof*⟩

corollary *rref-condition3-equiv*:
fixes *A*::*'a*::{*one*, *zero*}^*cols*::{*mod-type*}^*rows*::{*mod-type*}
assumes *rref*: *reduced-row-echelon-form A*
and *i-less-j*: *i*<*j*
and *i*: ¬ *is-zero-row i A*
and *j*: ¬ *is-zero-row j A*
shows (*LEAST n. A $ i $ n* ≠ *0*) < (*LEAST n. A $ j $ n* ≠ *0*)
⟨*proof*⟩

lemma *rref-implies-rref-upt*:
fixes *A*::*'a*::{*one*,*zero*}^*cols*::{*mod-type*}^*rows*::{*mod-type*}
assumes *rref*: *reduced-row-echelon-form A*
shows *reduced-row-echelon-form-upt-k A k*
⟨*proof*⟩

lemma *rref-first-position-zero-imp-column-0*:
fixes *A*::*'a*::{*one*,*zero*}^*cols*::{*mod-type*}^*rows*::{*mod-type*}
assumes *rref*: *reduced-row-echelon-form A*
and *A-00*: *A $ 0 $ 0 = 0*
shows *column 0 A = 0*
⟨*proof*⟩

lemma *rref-first-element*:
fixes *A*::*'a*::{*one*,*zero*}^*cols*::{*mod-type*}^*rows*::{*mod-type*}
assumes *rref*: *reduced-row-echelon-form A*
and *column-not-0*: *column 0 A* ≠ *0*
shows *A $ 0 $ 0 = 1*
⟨*proof*⟩

end

2 Code set

theory *Code-Set*
imports
HOL−Library.Code-Cardinality
begin

The following setup could help to get code generation for List.coset, but it neither works correctly it complains that code equations for remove are missed, even when List.coset should be rewritten to an enum:

declare *minus-coset-filter* [*code del*]
declare *remove-code*(*2*) [*code del*]
declare *insert-code*(*2*) [*code del*]
declare *inter-coset-fold* [*code del*]

declare *compl-coset*[*code del*]
declare *Code-Cardinality.card'-code*(*2*)[*code del*]

code-datatype *set*

The following code equation could be useful to avoid the problem of code generation for List.coset []:

lemma [*code*]: *List.coset* (*l*::'*a*::*enum list*) = *set* (*enum-class.enum*) − *set l*
⟨*proof*⟩

Now the following examples work:

value *UNIV*::*bool set*
value *List.coset* ([]::*bool list*)
value *UNIV*::*Enum.finite-2 set*
value *List.coset* ([]::*Enum.finite-2 list*)
value *List.coset* ([]::*Enum.finite-5 list*)

end

3 Code generation for vectors and matrices

theory *Code-Matrix*
imports
Rank-Nullity-Theorem.Miscellaneous
Code-Set
begin

In this file the code generator is set up properly to allow the execution of matrices represented as funcions over finite types.

lemmas *vec.vec-nth-inverse*[*code abstype*]

lemma [*code abstract*]: *vec-nth 0* = (%*x*. *0*) ⟨*proof*⟩
lemma [*code abstract*]: *vec-nth 1* = (%*x*. *1*) ⟨*proof*⟩
lemma [*code abstract*]: *vec-nth* (*a* + *b*) = (%*i*. *a*$*i* + *b*$*i*) ⟨*proof*⟩
lemma [*code abstract*]: *vec-nth* (*a* − *b*) = (%*i*. *a*$*i* − *b*$*i*) ⟨*proof*⟩
lemma [*code abstract*]: *vec-nth* (*vec n*) = (λ*i*. *n*) ⟨*proof*⟩
lemma [*code abstract*]: *vec-nth* (*a* ∗ *b*) = (%*i*. *a*$*i* ∗ *b*$*i*) ⟨*proof*⟩
lemma [*code abstract*]: *vec-nth* (*c* ∗*s x*) = (λ*i*. *c* ∗ (*x*$*i*)) ⟨*proof*⟩
lemma [*code abstract*]: *vec-nth* (*a* − *b*) = (%*i*. *a*$*i* − *b*$*i*) ⟨*proof*⟩

definition *mat-mult-row*
where *mat-mult-row m m' f* = *vec-lambda* (%*c*. *sum* (%*k*. ((*m*$*f*)$*k*) ∗ ((*m'*$*k*)$*c*)) (*UNIV* :: '*n*::*finite set*))

lemma *mat-mult-row-code* [*code abstract*]:
vec-nth (*mat-mult-row m m' f*) = (%*c*. *sum* (%*k*. ((*m*$*f*)$*k*) ∗ ((*m'*$*k*)$*c*)) (*UNIV* :: '*n*::*finite set*))
⟨*proof*⟩

lemma *mat-mult* [*code abstract*]: *vec-nth* (*m* ** *m'*) = *mat-mult-row m m'*
⟨*proof*⟩

lemma *matrix-vector-mult-code* [*code abstract*]:
vec-nth (*A* *v *x*) = (%*i*. (∑ *j*∈*UNIV*. *A* $ *i* $ *j* * *x* $ *j*)) ⟨*proof*⟩

lemma *vector-matrix-mult-code* [*code abstract*]:
vec-nth (*x* v* *A*) = (%*j*. (∑ *i*∈*UNIV*. *x* $ *i* * *A* $ *i* $ *j*)) ⟨*proof*⟩

definition *mat-row*
where *mat-row k i* = *vec-lambda* (%*j*. *if i* = *j then k else 0*)

lemma *mat-row-code* [*code abstract*]:
vec-nth (*mat-row k i*) = (%*j*. *if i* = *j then k else 0*) ⟨*proof*⟩

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

definition *transpose-row*
where *transpose-row A i* = *vec-lambda* (%*j*. *A* $ *j* $ *i*)

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

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

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

definition *rowvector-row v i* = *vec-lambda* (%*j*. (*v*$*j*))

lemma *rowvector-row-code* [*code abstract*]:
vec-nth (*rowvector-row v i*) = (%*j*. (*v*$*j*)) ⟨*proof*⟩

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

definition *columnvector-row v i* = *vec-lambda* (%*j*. (*v*$*i*))

lemma *columnvector-row-code* [*code abstract*]:
vec-nth (*columnvector-row v i*) = (%*j*. (*v*$*i*)) ⟨*proof*⟩

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

end

4 Elementary Operations over matrices

theory *Elementary-Operations*
imports
Rank-Nullity-Theorem.Fundamental-Subspaces
Code-Matrix
begin

4.1 Some previous results:

lemma *mat-1-fun*: *mat 1 $ a $ b* = (λ*i j. if i=j then 1 else 0*) *a b* ⟨*proof*⟩

lemma *mat1-sum-eq*:
shows (∑ *k*∈*UNIV*. *mat* (*1*::*'a*::{*semiring-1*}) $ *s* $ *k* ∗ *mat 1* $ *k* $ *t*) = *mat 1* $ *s* $ *t*
⟨*proof*⟩

lemma *invertible-mat-n*:
fixes *n*::*'a*::{*field*}
assumes *n*: *n* ≠ *0*
shows *invertible* ((*mat n*)::*'a*^*'n*^*'n*)
⟨*proof*⟩

corollary *invertible-mat-1*:
shows *invertible* (*mat* (*1*::*'a*::{*field*})) ⟨*proof*⟩

4.2 Definitions of elementary row and column operations

Definitions of elementary row operations

definition *interchange-rows* :: *'a* ^*'n*^*'m* => *'m* => *'m* ⇒ *'a* ^*'n*^*'m*
where *interchange-rows A a b* = (χ *i j. if i=a then A $ b $ j else if i=b then A $ a $ j else A $ i $ j*)

definition *mult-row* :: (*'a*::*times*) ^*'n*^*'m* => *'m* => *'a* ⇒ *'a* ^*'n*^*'m*
where *mult-row A a q* = (χ *i j. if i=a then q*∗(*A $ a $ j*) *else A $ i $ j*)

definition *row-add* :: (*'a*::{*plus*, *times*}) ^*'n*^*'m* => *'m* => *'m* ⇒ *'a* ⇒ *'a* ^*'n*^*'m*
where *row-add A a b q* = (χ *i j. if i=a then* (*A $ a $ j*) + *q*∗(*A $ b $ j*) *else A $ i $ j*)

Definitions of elementary column operations

definition *interchange-columns* :: *'a* ^*'n*^*'m* => *'n* => *'n* ⇒ *'a* ^*'n*^*'m*
where *interchange-columns A n m* = (χ *i j. if j=n then A $ i $ m else if j=m then A $ i $ n else A $ i $ j*)

definition *mult-column* :: (*'a*::*times*) ^*'n*^*'m* => *'n* => *'a* ⇒ *'a* ^*'n*^*'m*
where *mult-column A n q* = (χ *i j. if j=n then* (*A $ i $ j*)∗*q else A $ i $ j*)

definition *column-add* :: ('*a*::{*plus*, *times*}) ^'*n*^'*m* => '*n* => '*n* ⇒ '*a* ⇒ '*a* ^'*n*^'*m*
where *column-add A n m q* = (χ *i j*. *if j*=*n then* ((*A* $ *i* $ *n*) + (*A* $ *i* $ *m*)∗*q*) *else A* $ *i* $ *j*)

4.3 Properties about elementary row operations

4.3.1 Properties about interchanging rows

Properties about *interchange-rows*

lemma *interchange-same-rows*: *interchange-rows A a a* = *A*
⟨*proof*⟩

lemma *interchange-rows-i*[*simp*]: *interchange-rows A i j* $ *i* = *A* $ *j*
⟨*proof*⟩

lemma *interchange-rows-j*[*simp*]: *interchange-rows A i j* $ *j* = *A* $ *i*
⟨*proof*⟩

lemma *interchange-rows-preserves*:
assumes *i* ≠ *a* **and** *j* ≠ *a*
shows *interchange-rows A i j* $ *a* = *A* $ *a*
⟨*proof*⟩

lemma *interchange-rows-mat-1*:
shows *interchange-rows* (*mat 1*) *a b* ∗∗ *A* = *interchange-rows A a b*
⟨*proof*⟩

lemma *invertible-interchange-rows*: *invertible* (*interchange-rows* (*mat 1*) *a b*)
⟨*proof*⟩

4.3.2 Properties about multiplying a row by a constant

Properties about *mult-row*

lemma *mult-row-mat-1*: *mult-row* (*mat 1*) *a q* ∗∗ *A* = *mult-row A a q*
⟨*proof*⟩

lemma *invertible-mult-row*:
assumes *qk*: *q* ∗ *k* = *1* **and** *kq*: *k*∗*q*=*1*
shows *invertible* (*mult-row* (*mat 1*) *a q*)
⟨*proof*⟩

corollary *invertible-mult-row'*:
assumes *q-not-zero*: *q* ≠ *0*
shows *invertible* (*mult-row* (*mat* (*1*::'*a*::{*field*})) *a q*)
⟨*proof*⟩

4.3.3 Properties about adding a row multiplied by a constant to another row

Properties about *row-add*

lemma *row-add-mat-1*: *row-add* (*mat 1*) *a b q* ** *A* = *row-add A a b q*
⟨*proof*⟩

lemma *invertible-row-add*:
assumes *a-noteq-b*: *a*≠*b*
shows *invertible* (*row-add* (*mat* (*1*::'*a*::{*ring-1*})) *a b q*)
⟨*proof*⟩

4.4 Properties about elementary column operations

4.4.1 Properties about interchanging columns

Properties about *interchange-columns*

lemma *interchange-columns-mat-1*: *A* ** *interchange-columns* (*mat 1*) *a b* = *interchange-columns A a b*
⟨*proof*⟩

lemma *invertible-interchange-columns*: *invertible* (*interchange-columns* (*mat 1*) *a b*)
⟨*proof*⟩

4.4.2 Properties about multiplying a column by a constant

Properties about *mult-column*

lemma *mult-column-mat-1*: *A* ** *mult-column* (*mat 1*) *a q* = *mult-column A a q*
⟨*proof*⟩

lemma *invertible-mult-column*:
assumes *qk*: *q* * *k* = *1* **and** *kq*: *k* * *q* = *1*
shows *invertible* (*mult-column* (*mat 1*) *a q*)
⟨*proof*⟩

corollary *invertible-mult-column'*:
assumes *q-not-zero*: *q* ≠ *0*
shows *invertible* (*mult-column* (*mat* (*1*::'*a*::{*field*})) *a q*)
⟨*proof*⟩

4.4.3 Properties about adding a column multiplied by a constant to another column

Properties about *column-add*

lemma *column-add-mat-1*: *A* ** *column-add* (*mat 1*) *a b q* = *column-add A a b q*
⟨*proof*⟩

lemma *invertible-column-add*:
assumes *a-noteq-b*: $a \neq b$
shows *invertible* (*column-add* (*mat* (*1*::'*a*::{*ring-1*})) *a b q*)
⟨*proof*⟩

4.5 Relationships amongst the definitions

Relationships between *interchange-rows* and *interchange-columns*

lemma *interchange-rows-transpose*:
shows *interchange-rows* (*transpose A*) *a b* = *transpose* (*interchange-columns A a b*)
⟨*proof*⟩

lemma *interchange-rows-transpose'*:
shows *interchange-rows A a b* = *transpose* (*interchange-columns* (*transpose A*) *a b*)
⟨*proof*⟩

lemma *interchange-columns-transpose*:
shows *interchange-columns* (*transpose A*) *a b* = *transpose* (*interchange-rows A a b*)
⟨*proof*⟩

lemma *interchange-columns-transpose'*:
shows *interchange-columns A a b* = *transpose* (*interchange-rows* (*transpose A*) *a b*)
⟨*proof*⟩

4.6 Code Equations

Code equations for *interchange-rows ?A ?a ?b* = (*χi j. if i = ?a then ?A $?b $ j else if i = ?b then ?A $?a $ j else ?A $ i $ j*), *interchange-columns ?A ?n ?m* = (*χi j. if j = ?n then ?A $ i $?m else if j = ?m then ?A $ i $?n else ?A $ i $ j*), *row-add ?A ?a ?b ?q* = (*χi j. if i = ?a then ?A $?a $ j + ?q * ?A $?b $ j else ?A $ i $ j*), *column-add ?A ?n ?m ?q* = (*χi j. if j = ?n then ?A $ i $?n + ?A $ i $?m * ?q else ?A $ i $ j*), *mult-row ?A ?a ?q* = (*χi j. if i = ?a then ?q * ?A $?a $ j else ?A $ i $ j*) and *mult-column ?A ?n ?q* = (*χi j. if j = ?n then ?A $ i $ j * ?q else ?A $ i $ j*):

definition *interchange-rows-row*
where *interchange-rows-row A a b i* = *vec-lambda* (%*j. if i = a then A $ b $ j else if i = b then A $ a $ j else A $ i $ j*)

lemma *interchange-rows-code* [*code abstract*]:
vec-nth (*interchange-rows-row A a b i*) = (%*j. if i = a then A $ b $ j else if i = b then A $ a $ j else A $ i $ j*)

⟨*proof*⟩

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

definition *interchange-columns-row*
where *interchange-columns-row A n m i* = *vec-lambda* (%*j*. *if j* = *n then A* $ *i* $ *m else if j* = *m then A* $ *i* $ *n else A* $ *i* $ *j*)

lemma *interchange-columns-code* [*code abstract*]:
vec-nth (*interchange-columns-row A n m i*) = (%*j*. *if j* = *n then A* $ *i* $ *m else if j* = *m then A* $ *i* $ *n else A* $ *i* $ *j*)
⟨*proof*⟩

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

definition *row-add-row*
where *row-add-row A a b q i* = *vec-lambda* (%*j*. *if i* = *a then A* $ *a* $ *j* + *q* ∗ *A* $ *b* $ *j else A* $ *i* $ *j*)

lemma *row-add-code* [*code abstract*]:
vec-nth (*row-add-row A a b q i*) = (%*j*. *if i* = *a then A* $ *a* $ *j* + *q* ∗ *A* $ *b* $ *j else A* $ *i* $ *j*)
⟨*proof*⟩

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

definition *column-add-row*
where *column-add-row A n m q i* = *vec-lambda* (%*j*. *if j* = *n then A* $ *i* $ *n* + *A* $ *i* $ *m* ∗ *q else A* $ *i* $ *j*)

lemma *column-add-code* [*code abstract*]:
vec-nth (*column-add-row A n m q i*) = (%*j*. *if j* = *n then A* $ *i* $ *n* + *A* $ *i* $ *m* ∗ *q else A* $ *i* $ *j*)
⟨*proof*⟩

lemma *column-add-code-nth* [*code abstract*]: *vec-nth* (*column-add A a b q*) = *column-add-row A a b q*
⟨*proof*⟩

definition *mult-row-row*
where *mult-row-row A a q i* = *vec-lambda* (%*j*. *if i* = *a then q* ∗ *A* $ *a* $ *j else A* $ *i* $ *j*)

lemma *mult-row-code* [*code abstract*]:
vec-nth (*mult-row-row A a q i*) = (%*j*. *if i* = *a then q* ∗ *A* $ *a* $ *j else A* $ *i* $ *j*)
⟨*proof*⟩

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

definition *mult-column-row*
where *mult-column-row A n q i* = *vec-lambda* (%*j*. *if j* = *n then A* $ *i* $ *j* ∗ *q else A* $ *i* $ *j*)

lemma *mult-column-code* [*code abstract*]:
vec-nth (*mult-column-row A n q i*) = (%*j*. *if j* = *n then A* $ *i* $ *j* ∗ *q else A* $ *i* $ *j*)
⟨*proof*⟩

lemma *mult-column-code-nth* [*code abstract*]: *vec-nth* (*mult-column A a q*) = *mult-column-row A a q*
⟨*proof*⟩

end

5 Rank of a matrix

theory *Rank*
imports
Rank-Nullity-Theorem.Dim-Formula
begin

5.1 Row rank, column rank and rank

Definitions of row rank, column rank and rank

definition *row-rank* :: *'a*::{*field*}^*'n*^*'m*=>*nat*
where *row-rank A* = *vec.dim* (*row-space A*)

definition *col-rank* :: *'a*::{*field*}^*'n*^*'m*=>*nat*
where *col-rank A* = *vec.dim* (*col-space A*)

lemma *rank-def*: *rank A* = *row-rank A*
⟨*proof*⟩

5.2 Properties

lemma *rrk-is-preserved*:
fixes *A*::*'a*::{*field*}^*'cols*^*'rows*::{*finite*, *wellorder*}
and *P*::*'a*::{*field*}^*'rows*::{*finite*, *wellorder*}^*'rows*::{*finite*, *wellorder*}

assumes *inv-P*: *invertible P*
shows *row-rank A* = *row-rank* (*P****A*)
⟨*proof*⟩

lemma *crk-is-preserved*:
fixes *A*::*'a*::{*field*}^*cols*::{*finite*, *wellorder*}^*rows*
and *P*::*'a*::{*field*}^*rows*^*rows*
assumes *inv-P*: *invertible P*
shows *col-rank A* = *col-rank* (*P****A*)
⟨*proof*⟩

end

6 Gauss Jordan algorithm over abstract matrices

theory *Gauss-Jordan*
imports
Rref
Elementary-Operations
Rank
begin

6.1 The Gauss-Jordan Algorithm

Now, a computable version of the Gauss-Jordan algorithm is presented. The output will be a matrix in reduced row echelon form. We present an algorithm in which the reduction is applied by columns

Using this definition, zeros are made in the column j of a matrix A placing the pivot entry (a nonzero element) in the position (i,j). For that, a suitable row interchange is made to achieve a non-zero entry in position (i,j). Then, this pivot entry is multiplied by its inverse to make the pivot entry equals to 1. After that, are other entries of the j-th column are eliminated by subtracting suitable multiples of the i-th row from the other rows.

definition *Gauss-Jordan-in-ij* :: *'a*::{*semiring-1*, *inverse*, *one*, *uminus*}^*m*^*n*::{*finite*, *ord*}=> *'n*=>*'m*=>*'a*^*m*^*n*::{*finite*, *ord*}
where *Gauss-Jordan-in-ij A i j* = (*let n* = (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*);
interchange-A = (*interchange-rows A i n*);
A' = *mult-row interchange-A i* (*1*/*interchange-A*$*i*$*j*) *in*
vec-lambda(% *s. if s*=*i then A'* $ *s else* (*row-add A' s i* (−(*interchange-A*$*s*$*j*))) $ *s*))

lemma *Gauss-Jordan-in-ij-unfold*:
assumes ∃ *n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*
obtains *n* :: *'n*::{*finite*, *wellorder*} **and** *interchange-A* **and** *A'*
where
(*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) = *n*

and $A \$ n \$ j \neq 0$
and $i \leq n$
and *interchange-A = interchange-rows A i n*
and $A' =$ *mult-row interchange-A i (1 / interchange-A $ i $ j)*
and *Gauss-Jordan-in-ij A i j = vec-lambda (λs. if s = i then A' $ s*
else (row-add A' s i (− (interchange-A $ s $ j))) $ s)
⟨*proof*⟩

The following definition makes the step of Gauss-Jordan in a column. This function receives two input parameters: the column k where the step of Gauss-Jordan must be applied and a pair (which consists of the row where the pivot should be placed in the column k and the original matrix).

definition *Gauss-Jordan-column-k* :: (*nat* × ('*a*::{*zero*,*inverse*,*uminus*,*semiring-1*}^'*m*::{*mod-type*}^'*n*::{*mod-*

=> *nat* => (*nat* × ('*a*^'*m*::{*mod-type*}^'*n*::{*mod-type*}))
where *Gauss-Jordan-column-k A' k* = (*let i*=*fst A'*; *A*=(*snd A'*); *from-nat-i*=(*from-nat i*::'*n*); *from-nat-k*=(*from-nat k*::'*m*) *in*
if (∀ *m*≥(*from-nat-i*). *A* $ *m* $(*from-nat-k*)=*0*) ∨ (*i* = *nrows A*) *then* (*i*,*A*)
else (*i*+*1*, (*Gauss-Jordan-in-ij A* (*from-nat-i*) (*from-nat-k*))))

The following definition applies the Gauss-Jordan step from the first column up to the k one (included).

definition *Gauss-Jordan-upt-k* :: '*a*::{*inverse*,*uminus*,*semiring-1*}^'*columns*::{*mod-type*}^'*rows*::{*mod-type*}
=> *nat*
=> '*a*^'*columns*::{*mod-type*}^'*rows*::{*mod-type*}
where *Gauss-Jordan-upt-k A k* = *snd* (*foldl Gauss-Jordan-column-k* (*0*,*A*) [*0*..<*Suc k*])

Gauss-Jordan is to apply the *Gauss-Jordan-column-k* in all columns.

definition *Gauss-Jordan* :: '*a*::{*inverse*,*uminus*,*semiring-1*}^'*columns*::{*mod-type*}^'*rows*::{*mod-type*}

=> '*a*^'*columns*::{*mod-type*}^'*rows*::{*mod-type*}
where *Gauss-Jordan A* = *Gauss-Jordan-upt-k A* ((*ncols A*) − *1*)

6.2 Properties about rref and the greatest nonzero row.

lemma *greatest-plus-one-eq-0*:
fixes *A*::'*a*::{*field*}^'*columns*::{*mod-type*}^'*rows*::{*mod-type*} **and** *k*::*nat*
assumes *Suc* (*to-nat* (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*)) = *nrows A*
shows (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1* = *0*
⟨*proof*⟩

lemma *from-nat-to-nat-greatest*:
fixes *A*::'*a*::{*zero*}^'*columns*::{*mod-type*}^'*rows*::{*mod-type*}
shows *from-nat* (*Suc* (*to-nat* (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*))) = (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*
⟨*proof*⟩

lemma *greatest-less-zero-row*:
fixes $A::'a::\{one,\ zero\}^{\sim}{}'n::\{mod\text{-}type\}^{\sim}{}'m::\{finite,one,plus,linorder\}$
assumes *r*: *reduced-row-echelon-form-upt-k A k*
and *zero-i*: *is-zero-row-upt-k i k A*
and *not-all-zero*: $\neg\ (\forall\ a.\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ a\ k\ A)$
shows $(GREATEST\ m.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A) < i$
⟨*proof*⟩

lemma *rref-suc-if-zero-below-greatest*:
fixes $A::'a::\{one,\ zero\}^{\sim}{}'n::\{mod\text{-}type\}^{\sim}{}'m::\{finite,one,plus,linorder\}$
assumes *r*: *reduced-row-echelon-form-upt-k A k*
and *not-all-zero*: $\neg\ (\forall\ a.\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ a\ k\ A)$
and *all-zero-below-greatest*: $\forall\ a.\ a>(GREATEST\ m.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A)$
$\longrightarrow$ *is-zero-row-upt-k a* (*Suc k*) *A*
shows *reduced-row-echelon-form-upt-k A* (*Suc k*)
⟨*proof*⟩

lemma *rref-suc-if-all-rows-not-zero*:
fixes $A::'a::\{one,\ zero\}^{\sim}{}'n::\{mod\text{-}type\}^{\sim}{}'m::\{finite,one,plus,linorder\}$
assumes *r*: *reduced-row-echelon-form-upt-k A k*
and *all-not-zero*: $\forall\ n.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ n\ k\ A$
shows *reduced-row-echelon-form-upt-k A* (*Suc k*)
⟨*proof*⟩

lemma *greatest-ge-nonzero-row*:
fixes $A::'a::\{zero\}^{\sim}{}'n::\{mod\text{-}type\}^{\sim}{}'m::\{finite,linorder\}$
assumes $\neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ i\ k\ A$
shows $i \leq (GREATEST\ m.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A)$ ⟨*proof*⟩

lemma *greatest-ge-nonzero-row′*:
fixes $A::'a::\{zero,\ one\}^{\sim}{}'n::\{mod\text{-}type\}^{\sim}{}'m::\{finite,\ linorder,\ one,\ plus\}$
assumes *r*: *reduced-row-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 $\neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ i\ k\ A$
⟨*proof*⟩

corollary *row-greater-greatest-is-zero*:
fixes $A::'a::\{zero\}^{\sim}{}'n::\{mod\text{-}type\}^{\sim}{}'m::\{finite,linorder\}$
assumes $(GREATEST\ m.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A) < i$
shows *is-zero-row-upt-k i k A* ⟨*proof*⟩

6.3 The proof of its correctness

Properties of *Gauss-Jordan-in-ij*

lemma *Gauss-Jordan-in-ij-1*:
fixes $A::'a::\{field\}^{\sim}{}'m^{\sim}{}'n::\{finite,\ ord,\ wellorder\}$
assumes *ex*: $\exists\ n.\ A\ \$\ n\ \$\ j \neq 0\ \wedge\ i \leq n$

shows (*Gauss-Jordan-in-ij A i j*) $ *i* $ *j* = *1*
⟨*proof*⟩

lemma *Gauss-Jordan-in-ij-0*:
fixes *A*::*'a*::{*field*}^*'m*^*n*::{*finite*, *ord*, *wellorder*}
assumes *ex*: ∃ *n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n* **and** *a*: *a* ≠ *i*
shows (*Gauss-Jordan-in-ij A i j*) $ *a* $ *j* = *0*
⟨*proof*⟩

corollary *Gauss-Jordan-in-ij-0'*:
fixes *A*::*'a*::{*field*}^*'m*^*n*::{*finite*, *ord*, *wellorder*}
assumes *ex*: ∃ *n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*
shows ∀ *a*. *a* ≠ *i* ⟶ (*Gauss-Jordan-in-ij A i j*) $ *a* $ *j* = *0* ⟨*proof*⟩

lemma *Gauss-Jordan-in-ij-preserves-previous-elements*:
fixes *A*::*'a*::{*field*}^*columns*::{*mod-type*}^*rows*::{*mod-type*}
assumes *r*: *reduced-row-echelon-form-upt-k A k*
and *not-zero-a*: ¬ *is-zero-row-upt-k a k A*
and *exists-m*: ∃ *m*. *A* $ *m* $ (*from-nat k*) ≠ *0* ∧ (*GREATEST m*. ¬ *is-zero-row-upt-k m k A*) + *1* ≤ *m*
and *Greatest-plus-1*: (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1* ≠ *0*
and *j-le-k*: *to-nat j* < *k*
shows *Gauss-Jordan-in-ij A* ((*GREATEST m*. ¬ *is-zero-row-upt-k m k A*) + *1*) (*from-nat k*) $ *i* $ *j* = *A* $ *i* $ *j*
⟨*proof*⟩

lemma *Gauss-Jordan-in-ij-preserves-previous-elements'*:
fixes *A*::*'a*::{*field*}^*columns*::{*mod-type*}^*rows*::{*mod-type*}
assumes *all-zero*: ∀ *n*. *is-zero-row-upt-k n k A*
and *j-le-k*: *to-nat j* < *k*
and *A-nk-not-zero*: *A* $ *n* $ (*from-nat k*) ≠ *0*
shows *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *i* $ *j* = *A* $ *i* $ *j*
⟨*proof*⟩

lemma *is-zero-after-Gauss*:
fixes *A*::*'a*::{*field*}^*n*::{*mod-type*}^*m*::{*mod-type*}
assumes *zero-a*: *is-zero-row-upt-k a k A*
and *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *r*: *reduced-row-echelon-form-upt-k A k*
and *greatest-less-ma*: (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *ma*
and *A-ma-k-not-zero*: *A* $ *ma* $ *from-nat k* ≠ *0*
shows *is-zero-row-upt-k a k* (*Gauss-Jordan-in-ij A* ((*GREATEST m*. ¬ *is-zero-row-upt-k m k A*) + *1*) (*from-nat k*))
⟨*proof*⟩

lemma *all-zero-imp-Gauss-Jordan-column-not-zero-in-row-0*:

fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*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*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *all-zero*: ∀ *n*. *is-zero-row-upt-k n k A*
and *not-zero-i*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *Amk-zero*: *A* $ *m* $ *from-nat k* ≠ *0*
shows *i*=*0*
⟨*proof*⟩

Here we start to prove that the output of *Gauss Jordan A* is a matrix in reduced row echelon form.

lemma *condition-1-part-1*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
assumes *zero-column-k*: ∀ *m*≥*from-nat 0*. *A* $ *m* $ *from-nat k* = *0*
and *all-zero*: ∀ *m*. *is-zero-row-upt-k m k A*
shows *is-zero-row-upt-k j* (*Suc k*) *A*
⟨*proof*⟩

lemma *condition-1-part-2*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
assumes *j-not-zero*: *j* ≠ *0*
and *all-zero*: ∀ *m*. *is-zero-row-upt-k m k A*
and *Amk-not-zero*: *A* $ *m* $ *from-nat k* ≠ *0*
shows *is-zero-row-upt-k j* (*Suc k*) (*Gauss-Jordan-in-ij A* (*from-nat 0*) (*from-nat k*))
⟨*proof*⟩

lemma *condition-1-part-3*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*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* ≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *i-less-j*: *i*<*j*
and *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *zero-below-greatest*: ∀ *m*≥(*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*. *A* $ *m* $ *from-nat k* = *0*
and *zero-i-suc-k*: *is-zero-row-upt-k i* (*Suc k*) *B*
shows *is-zero-row-upt-k j* (*Suc k*) *A*
⟨*proof*⟩

lemma *condition-1-part-4*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*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* ≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *zero-i-suc-k*: *is-zero-row-upt-k i* (*Suc k*) *B*

and *i-less-j*: $i<j$
and *not-zero-m*: $\neg$ *is-zero-row-upt-k m k A*
and *greatest-eq-card*: *Suc* (*to-nat* (*GREATEST n*. $\neg$ *is-zero-row-upt-k n k A*)) = *nrows A*
shows *is-zero-row-upt-k j* (*Suc k*) *A*
⟨*proof*⟩

lemma *condition-1-part-5*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* ∀ *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* ≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *zero-i-suc-k*: *is-zero-row-upt-k i* (*Suc k*) *B*
and *i-less-j*: $i<j$
and *not-zero-m*: $\neg$ *is-zero-row-upt-k m k A*
and *greatest-not-card*: *Suc* (*to-nat* (*GREATEST n*. $\neg$ *is-zero-row-upt-k n k A*)) ≠ *nrows A*
and *greatest-less-ma*: (*GREATEST n*. $\neg$ *is-zero-row-upt-k n k A*) + *1* ≤ *ma*
and *A-ma-k-not-zero*: *A* $ *ma* $ *from-nat k* ≠ *0*
shows *is-zero-row-upt-k j* (*Suc k*) (*Gauss-Jordan-in-ij A* ((*GREATEST n*. $\neg$ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*))
⟨*proof*⟩

lemma *condition-1*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* ∀ *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*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *zero-i-suc-k*: *is-zero-row-upt-k i* (*Suc k*) *B* **and** *i-less-j*: $i < j$
shows *is-zero-row-upt-k j* (*Suc k*) *B*
⟨*proof*⟩

lemma *condition-2-part-1*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*if* ∀ *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*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *not-zero-i-suc-k*: $\neg$ *is-zero-row-upt-k i* (*Suc k*) *B*
and *all-zero*: ∀ *m*. *is-zero-row-upt-k m k A*
and *all-zero-k*: ∀ *m*. *A* $ *m* $ *from-nat k* = *0*
shows *A* $ *i* $ (*LEAST k*. *A* $ *i* $ *k* ≠ *0*) = *1*
⟨*proof*⟩

lemma *condition-2-part-2*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*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*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *all-zero*: ∀ *m*. *is-zero-row-upt-k m k A*
and *Amk-not-zero*: *A* $ *m* $ *from-nat k* ≠ *0*
shows *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *i* $ (*LEAST ka*. *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *i* $ *ka* ≠ *0*) = *1*
⟨*proof*⟩

lemma *condition-2-part-3*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*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*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *zero-below-greatest*: ∀ *m*≥(*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*. *A* $ *m* $ *from-nat k* = *0*
shows *A* $ *i* $ (*LEAST k*. *A* $ *i* $ *k* ≠ *0*) = *1*
⟨*proof*⟩

lemma *condition-2-part-4*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *greatest-eq-card*: *Suc* (*to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)) = *nrows A*
shows *A* $ *i* $ (*LEAST k*. *A* $ *i* $ *k* ≠ *0*) = *1*
⟨*proof*⟩

lemma *condition-2-part-5*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*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*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *greatest-noteq-card*: *Suc* (*to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)) ≠ *nrows A*

and *greatest-less-ma*: (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *ma*
and *A-ma-k-not-zero*: *A* $ *ma* $ *from-nat k* ≠ *0*
shows *Gauss-Jordan-in-ij A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $
(*LEAST ka. Gauss-Jordan-in-ij A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ *ka* ≠ *0*) = *1*
⟨*proof*⟩

lemma *condition-2*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*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*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
shows *B* $ *i* $ (*LEAST k. B* $ *i* $ *k* ≠ *0*) = *1*
⟨*proof*⟩

lemma *condition-3-part-1*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*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*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *all-zero*: ∀ *m. is-zero-row-upt-k m k A*
and *all-zero-k*: ∀ *m. A* $ *m* $ *from-nat k* = *0*
shows (*LEAST n. A* $ *i* $ *n* ≠ *0*) < (*LEAST n. A* $ (*i* + *1*) $ *n* ≠ *0*)
⟨*proof*⟩

lemma *condition-3-part-2*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*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*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *i-le*: *i* < *i* + *1*
and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *not-zero-suc-i-suc-k*: ¬ *is-zero-row-upt-k* (*i* + *1*) (*Suc k*) *B*
and *all-zero*: ∀ *m. is-zero-row-upt-k m k A*
and *Amk-notzero*: *A* $ *m* $ *from-nat k* ≠ *0*
shows (*LEAST n. Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *i* $ *n* ≠ *0*) < (*LEAST n. Gauss-Jordan-in-ij A 0* (*from-nat k*) $ (*i* + *1*) $ *n* ≠ *0*)
⟨*proof*⟩

lemma *condition-3-part-3*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*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*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *i-le*: *i* < *i* + *1*
and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *not-zero-suc-i-suc-k*: ¬ *is-zero-row-upt-k* (*i* + *1*) (*Suc k*) *B*
and *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *zero-below-greatest*: ∀ *m*≥(*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*. *A* $ *m* $ *from-nat k* = *0*
shows (*LEAST n*. *A* $ *i* $ *n* ≠ *0*) < (*LEAST n*. *A* $ (*i* + *1*) $ *n* ≠ *0*)
⟨*proof*⟩

lemma *condition-3-part-4*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*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*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k* **and** *i-le*: *i* < *i* + *1*
and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *not-zero-suc-i-suc-k*: ¬ *is-zero-row-upt-k* (*i* + *1*) (*Suc k*) *B*
and *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *greatest-eq-card*: *Suc* (*to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)) = *nrows A*
shows (*LEAST n*. *A* $ *i* $ *n* ≠ *0*) < (*LEAST n*. *A* $ (*i* + *1*) $ *n* ≠ *0*)
⟨*proof*⟩

lemma *condition-3-part-5*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*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*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *i-le*: *i* < *i* + *1*
and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *not-zero-suc-i-suc-k*: ¬ *is-zero-row-upt-k* (*i* + *1*) (*Suc k*) *B*
and *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *greatest-not-card*: *Suc* (*to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)) ≠ *nrows A*
and *greatest-less-ma*: (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *ma*
and *A-ma-k-not-zero*: *A* $ *ma* $ *from-nat k* ≠ *0*
shows (*LEAST n*. *Gauss-Jordan-in-ij A* ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ *n* ≠ *0*)
< (*LEAST n*. *Gauss-Jordan-in-ij A* ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ (*i* + *1*) $ *n* ≠ *0*)

⟨*proof*⟩

lemma *condition-3*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*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*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *i-le*: *i* < *i* + *1*
and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *not-zero-suc-i-suc-k*: ¬ *is-zero-row-upt-k* (*i* + *1*) (*Suc k*) *B*
shows (*LEAST n*. *B* $ *i* $ *n* ≠ *0*) < (*LEAST n*. *B* $ (*i* + *1*) $ *n* ≠ *0*)
⟨*proof*⟩

lemma *condition-4-part-1*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*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*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *all-zero*: ∀ *m*. *is-zero-row-upt-k m k A*
and *all-zero-k*: ∀ *m*. *A* $ *m* $ *from-nat k* = *0*
shows *A* $ *j* $ (*LEAST n*. *A* $ *i* $ *n* ≠ *0*) = *0*
⟨*proof*⟩

lemma *condition-4-part-2*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*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*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *i-not-j*: *i* ≠ *j*
and *all-zero*: ∀ *m*. *is-zero-row-upt-k m k A*
and *Amk-not-zero*: *A* $ *m* $ *from-nat k* ≠ *0*
shows *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *j* $ (*LEAST n*. *Gauss-Jordan-in-ij A 0* (*from-nat k*) $ *i* $ *n* ≠ *0*) = *0*
⟨*proof*⟩

lemma *condition-4-part-3*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*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*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *i-not-j*: *i* ≠ *j*

and *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *zero-below-greatest*: ∀ *m*≥(*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*. *A* $ *m* $ *from-nat k* = *0*
shows *A* $ *j* $ (*LEAST n. A* $ *i* $ *n* ≠ *0*) = *0*
⟨*proof*⟩

lemma *condition-4-part-4*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*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*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *i-not-j*: *i* ≠ *j*
and *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *greatest-eq-card*: *Suc* (*to-nat* (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*)) = *nrows A*
shows *A* $ *j* $ (*LEAST n. A* $ *i* $ *n* ≠ *0*) = *0*
⟨*proof*⟩

lemma *condition-4-part-5*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*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*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *i-not-j*: *i* ≠ *j*
and *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *greatest-not-card*: *Suc* (*to-nat* (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*)) ≠ *nrows A*
and *greatest-less-ma*: (*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *ma*
and *A-ma-k-not-zero*: *A* $ *ma* $ *from-nat k* ≠ *0*
shows *Gauss-Jordan-in-ij A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *j* $
(*LEAST n. Gauss-Jordan-in-ij A* ((*GREATEST n.* ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*) $ *i* $ *n* ≠ *0*) = *0*
⟨*proof*⟩

lemma *condition-4*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*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*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *not-zero-i-suc-k*: ¬ *is-zero-row-upt-k i* (*Suc k*) *B*
and *i-not-j*: *i* ≠ *j*
shows *B* $ *j* $ (*LEAST n. B* $ *i* $ *n* ≠ *0*) = *0*

⟨*proof*⟩

lemma *reduced-row-echelon-form-upt-k-Gauss-Jordan-column-k*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
defines *ia*:*ia*≡(*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*≡(*snd* (*Gauss-Jordan-column-k* (*ia*,*A*) *k*))
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
shows *reduced-row-echelon-form-upt-k B* (*Suc k*)
⟨*proof*⟩

lemma *foldl-Gauss-condition-1*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
assumes ∀ *m*. *is-zero-row-upt-k m k A*
and ∀ *m*≥*0*. *A* $ *m* $ *from-nat k* = *0*
shows *is-zero-row-upt-k m* (*Suc k*) *A*
⟨*proof*⟩

lemma *foldl-Gauss-condition-2*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
assumes *k*: *k* < *ncols A*
and *all-zero*: ∀ *m*. *is-zero-row-upt-k m k A*
and *Amk-not-zero*: *A* $ *m* $ *from-nat k* ≠ *0*
shows ∃ *m*. ¬ *is-zero-row-upt-k m* (*Suc k*) (*Gauss-Jordan-in-ij A 0* (*from-nat k*))
⟨*proof*⟩

lemma *foldl-Gauss-condition-3*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
assumes *k*: *k* < *ncols A*
and *all-zero*: ∀ *m*. *is-zero-row-upt-k m k A*
and *Amk-not-zero*: *A* $ *m* $ *from-nat k* ≠ *0*
and ¬ *is-zero-row-upt-k ma* (*Suc k*) (*Gauss-Jordan-in-ij A 0* (*from-nat k*))
shows *to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n* (*Suc k*) (*Gauss-Jordan-in-ij A 0* (*from-nat k*))) = *0*
⟨*proof*⟩

lemma *foldl-Gauss-condition-5*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*} **and** *k*::*nat*
assumes *rref-A*: *reduced-row-echelon-form-upt-k A k*
and *not-zero-a*:¬ *is-zero-row-upt-k a k A*
and *all-zero-below-greatest*: ∀ *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 *foldl-Gauss-condition-6*:
fixes *A*::*'a*::{*field*}^*columns*::{*mod-type*}^*rows*::{*mod-type*} **and** *k*::*nat*
assumes *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *eq-card*: *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 *foldl-Gauss-condition-8*:
fixes *A*::*'a*::{*field*}^*columns*::{*mod-type*}^*rows*::{*mod-type*} **and** *k*::*nat*
assumes *k*: *k* < *ncols A*
and *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *A-ma-k*: *A* $ *ma* $ *from-nat k* ≠ *0*
and *ma*: (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *ma*
shows ∃ *m*. ¬ *is-zero-row-upt-k m* (*Suc k*) (*Gauss-Jordan-in-ij A* ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*))
⟨*proof*⟩

lemma *foldl-Gauss-condition-9*:
fixes *A*::*'a*::{*field*}^*columns*::{*mod-type*}^*rows*::{*mod-type*} **and** *k*::*nat*
assumes *k*: *k* < *ncols A*
and *rref-A*: *reduced-row-echelon-form-upt-k A k*
assumes *not-zero-m*: ¬ *is-zero-row-upt-k m k A*
and *suc-greatest-not-card*: *Suc* (*to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*)) ≠ *nrows A*
and *greatest-less-ma*: (*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1* ≤ *ma*
and *A-ma-k*: *A* $ *ma* $ *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*) (*Gauss-Jordan-in-ij A* ((*GREATEST n*. ¬ *is-zero-row-upt-k n k A*) + *1*) (*from-nat k*)))
⟨*proof*⟩

The following lemma is one of most important ones in the verification of the Gauss-Jordan algorithm. The aim is to prove two statements about *Gauss-Jordan-upt-k ?A ?k* = *snd* (*foldl Gauss-Jordan-column-k* (*0*, *?A*) [*0*..<*Suc ?k*]) (one about the result is on rref and another about the index). The reason of doing that way is because both statements need them mutually to be proved. As the proof is made using induction, two base cases and two induction steps appear.

lemma *rref-and-index-Gauss-Jordan-upt-k*:
fixes *A*::*'a*::{*field*}^*columns*::{*mod-type*}^*rows*::{*mod-type*} **and** *k*::*nat*
assumes *k* < *ncols A*
shows *rref-Gauss-Jordan-upt-k*: *reduced-row-echelon-form-upt-k* (*Gauss-Jordan-upt-k*

A k) (*Suc k*)
and *snd-Gauss-Jordan-upt-k*:
foldl Gauss-Jordan-column-k (*0*, *A*) [*0*..<*Suc k*] =
(*if* ∀ *m*. *is-zero-row-upt-k m* (*Suc k*) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*])) *then 0*
else to-nat (*GREATEST n*. ¬ *is-zero-row-upt-k n* (*Suc k*) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*]))) + *1*,
snd (*foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*Suc k*]))
⟨*proof*⟩

corollary *rref-Gauss-Jordan*:
fixes *A*::*'a*::{*field*}^*columns*::{*mod-type*}^*rows*::{*mod-type*}
shows *reduced-row-echelon-form* (*Gauss-Jordan A*)
⟨*proof*⟩

lemma *independent-not-zero-rows-rref*:
fixes *A*::*'a*::{*field*}^*m*::{*mod-type*}^*n*::{*finite*,*one*,*plus*,*ord*}
assumes *rref-A*: *reduced-row-echelon-form A*
shows *vec.independent* {*row i A* |*i*. *row i A* ≠ *0*}
⟨*proof*⟩

Here we start to prove that the transformation from the original matrix to its reduced row echelon form has been carried out by means of elementary operations.

The following function eliminates all entries of the j-th column using the non-zero element situated in the position (i,j). It is introduced to make easier the proof that each Gauss-Jordan step consists in applying suitable elementary operations.

primrec *row-add-iterate* :: *'a*::{*semiring-1*, *uminus*}^*n*^*m*::{*mod-type*} => *nat* => *'m* => *'n* => *'a*^*n*^*m*::{*mod-type*}
where *row-add-iterate A 0 i j* = (*if i=0 then A else row-add A 0 i* (−*A* $ *0* $ *j*))
| *row-add-iterate A* (*Suc n*) *i j* = (*if* (*Suc n* = *to-nat i*) *then row-add-iterate A n i j*
else row-add-iterate (*row-add A* (*from-nat* (*Suc n*)) *i* (− *A* $ (*from-nat* (*Suc n*)) $ *j*)) *n i j*)

lemma *invertible-row-add-iterate*:
fixes *A*::*'a*::{*ring-1*}^*n*^*m*::{*mod-type*}
assumes *n*: *n*<*nrows A*
shows ∃ *P*. *invertible P* ∧ *row-add-iterate A n i j* = *P****A*
⟨*proof*⟩

lemma *row-add-iterate-preserves-greater-than-n*:
fixes *A*::*'a*::{*ring-1*}^*n*^*m*::{*mod-type*}
assumes *n*: *n*<*nrows A*

and *a*: *to-nat a* > *n*
shows (*row-add-iterate A n i j*) $ *a* $ *b* = *A* $ *a* $ *b*
⟨*proof*⟩

lemma *row-add-iterate-preserves-pivot-row*:
fixes *A*::'*a*::{*ring-1*}^'*n*^'*m*::{*mod-type*}
assumes *n*: *n*<*nrows A*
and *a*: *to-nat i* ≤ *n*
shows (*row-add-iterate A n i j*) $ *i* $ *b* = *A* $ *i* $ *b*
⟨*proof*⟩

lemma *row-add-iterate-eq-row-add*:
fixes *A*::'*a*::{*ring-1*}^'*n*^'*m*::{*mod-type*}
assumes *a-not-i*: *a* ≠ *i*
and *n*: *n*<*nrows A*
and *to-nat a* ≤ *n*
shows (*row-add-iterate A n i j*) $ *a* $ *b* = (*row-add A a i* (− *A* $ *a* $ *j*)) $ *a* $ *b*
⟨*proof*⟩

lemma *row-add-iterate-eq-Gauss-Jordan-in-ij*:
fixes *A*::'*a*::{*field*}^'*n*^'*m*::{*mod-type*} **and** *i*::'*m* **and** *j*::'*n*
defines *A'*: *A'*== *mult-row* (*interchange-rows A i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*)) *i* (*1* / (*interchange-rows A i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*)) $ *i* $ *j*)
shows *row-add-iterate A'* (*nrows A* − *1*) *i j* = *Gauss-Jordan-in-ij A i j*
⟨*proof*⟩

lemma *invertible-Gauss-Jordan-column-k*:
fixes *A*::'*a*::{*field*}^'*n*::{*mod-type*}^'*m*::{*mod-type*} **and** *k*::*nat*
shows ∃ *P*. *invertible P* ∧ (*snd* (*Gauss-Jordan-column-k* (*i*,*A*) *k*)) = *P****A*
⟨*proof*⟩

lemma *invertible-Gauss-Jordan-up-to-k*:
fixes *A*::'*a*::{*field*}^'*n*::{*mod-type*}^'*m*::{*mod-type*}
shows ∃ *P*. *invertible P* ∧ (*Gauss-Jordan-upt-k A k*) = *P****A*
⟨*proof*⟩

lemma *inj-index-independent-rows*:
fixes *A*::'*a*::{*field*}^'*m*::{*mod-type*}^'*n*::{*finite*,*one*,*plus*,*ord*}
assumes *rref-A*: *reduced-row-echelon-form A*
and *x*: *row x A* ∈ {*row i A* |*i*. *row i A* ≠ *0*}
and *eq*: *A* $ *x* = *A* $ *y*
shows *x* = *y*
⟨*proof*⟩

The final results:

lemma *invertible-Gauss-Jordan*:
fixes $A::'a::\{field\}\hat{}'n::\{mod\text{-}type\}\hat{}'m::\{mod\text{-}type\}$
shows $\exists P.\ invertible\ P \wedge (Gauss\text{-}Jordan\ A) = P{**}A$ ⟨*proof*⟩

lemma *Gauss-Jordan*:
fixes $A::'a::\{field\}\hat{}'n::\{mod\text{-}type\}\hat{}'m::\{mod\text{-}type\}$
shows $\exists P.\ invertible\ P \wedge (Gauss\text{-}Jordan\ A) = P{**}A \wedge reduced\text{-}row\text{-}echelon\text{-}form\ (Gauss\text{-}Jordan\ A)$
⟨*proof*⟩

Some properties about the rank of a matrix, obtained thanks to the Gauss-Jordan algorithm and the reduced row echelon form.

lemma *rref-rank*:
fixes $A::'a::\{field\}\hat{}'m::\{mod\text{-}type\}\hat{}'n::\{finite,one,plus,ord\}$
assumes *rref-A*: *reduced-row-echelon-form A*
shows $rank\ A = card\ \{row\ i\ A\ |i.\ row\ i\ A \neq 0\}$
⟨*proof*⟩

lemma *column-leading-coefficient-component-eq*:
fixes $A::'a::\{field\}\hat{}'m::\{mod\text{-}type\}\hat{}'n::\{finite,one,plus,ord\}$
assumes *rref-A*: *reduced-row-echelon-form A*
and *v*: $v \in \{column\ (LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0)\ A\ |i.\ row\ i\ A \neq 0\}$
and *vx*: $v\ \$\ x \neq 0$
and *vy*: $v\ \$\ y \neq 0$
shows $x = y$
⟨*proof*⟩

lemma *column-leading-coefficient-component-1*:
fixes $A::'a::\{field\}\hat{}'m::\{mod\text{-}type\}\hat{}'n::\{finite,one,plus,ord\}$
assumes *rref-A*: *reduced-row-echelon-form A*
and *v*: $v \in \{column\ (LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0)\ A\ |i.\ row\ i\ A \neq 0\}$
and *vx*: $v\ \$\ x \neq 0$
shows $v\ \$\ x = 1$
⟨*proof*⟩

lemma *column-leading-coefficient-component-0*:
fixes $A::'a::\{field\}\hat{}'m::\{mod\text{-}type\}\hat{}'n::\{finite,one,plus,ord\}$
assumes *rref-A*: *reduced-row-echelon-form A*
and *v*: $v \in \{column\ (LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0)\ A\ |i.\ row\ i\ A \neq 0\}$
and *vx*: $v\ \$\ x \neq 0$
and *x-not-y*: $x \neq y$
shows $v\ \$\ y = 0$ ⟨*proof*⟩

lemma *rref-col-rank*:
fixes $A::'a::\{field\}\hat{}'m::\{mod\text{-}type\}\hat{}'n::\{mod\text{-}type\}$
assumes *rref-A*: *reduced-row-echelon-form A*

shows *col-rank A = card {column (LEAST n. A $ i $ n ≠ 0) A | i. row i A ≠ 0}*
⟨*proof*⟩

lemma *rref-row-rank*:
fixes *A*::*'a*::{*field*}^*'m*::{*mod-type*}^*'n*::{*finite*,*one*,*plus*,*ord*}
assumes *rref-A*: *reduced-row-echelon-form A*
shows *row-rank A = card {column (LEAST n. A $ i $ n ≠ 0) A | i. row i A ≠ 0}*
⟨*proof*⟩

lemma *row-rank-eq-col-rank-rref*:
fixes *A*::*'a*::{*field*}^*'m*::{*mod-type*}^*'n*::{*mod-type*}
assumes *r*: *reduced-row-echelon-form A*
shows *row-rank A = col-rank A*
⟨*proof*⟩

lemma *row-rank-eq-col-rank*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'m*::{*mod-type*}
shows *row-rank A = col-rank A*
⟨*proof*⟩

theorem *rank-col-rank*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'m*::{*mod-type*}
shows *rank A = col-rank A* ⟨*proof*⟩

theorem *rank-eq-dim-image*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'m*::{*mod-type*}
shows *rank A = vec.dim (range (λx. A ∗v x))*
⟨*proof*⟩

theorem *rank-eq-dim-col-space*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'m*::{*mod-type*}
shows *rank A = vec.dim (col-space A)* ⟨*proof*⟩

lemma *rank-transpose*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'m*::{*mod-type*}
shows *rank (transpose A) = rank A*
⟨*proof*⟩

lemma *rank-le-nrows*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'m*::{*mod-type*}
shows *rank A ≤ nrows A*
⟨*proof*⟩

lemma *rank-le-ncols*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'m*::{*mod-type*}
shows *rank A* ≤ *ncols A*
⟨*proof*⟩

lemma *rank-Gauss-Jordan*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'m*::{*mod-type*}
shows *rank A* = *rank* (*Gauss-Jordan A*)
⟨*proof*⟩

Other interesting properties:

lemma *A-0-imp-Gauss-Jordan-0*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'m*::{*mod-type*}
assumes *A*=*0*
shows *Gauss-Jordan A* = *0*
⟨*proof*⟩

lemma *rank-0*: *rank 0* = *0*
⟨*proof*⟩

lemma *rank-greater-zero*:
assumes *A* ≠ *0*
shows *rank A* > *0*
⟨*proof*⟩

lemma *Gauss-Jordan-not-0*:
fixes *A*::*'a*::{*field*}^*cols*::{*mod-type*}^*rows*::{*mod-type*}
assumes *A* ≠ *0*
shows *Gauss-Jordan A* ≠ *0*
⟨*proof*⟩

lemma *rank-eq-suc-to-nat-greatest*:
assumes *A-not-0*: *A* ≠ *0*
shows *rank A* = *to-nat* (*GREATEST a*. ¬ *is-zero-row a* (*Gauss-Jordan A*)) + *1*
⟨*proof*⟩

lemma *rank-less-row-i-imp-i-is-zero*:
assumes *rank-less-i*: *to-nat i* ≥ *rank A*
shows *Gauss-Jordan A* $ *i* = *0*
⟨*proof*⟩

lemma *rank-Gauss-Jordan-eq*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'m*::{*mod-type*}
shows *rank A* = (*let A'*=(*Gauss-Jordan A*) *in card* {*row i A'* |*i*. *row i A'* ≠ *0*})
⟨*proof*⟩

6.4 Lemmas for code generation and rank computation

lemma [*code abstract*]:
shows *vec-nth* (*Gauss-Jordan-in-ij A i j*) = (*let n* = (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*);
interchange-A = (*interchange-rows A i n*);
A′ = *mult-row interchange-A i* (*1*/*interchange-A*$*i*$*j*) *in*
(% *s. if s*=*i then A′* $ *s else* (*row-add A′ s i* (−(*interchange-A*$*s*$*j*))) $ *s*))
⟨*proof*⟩

lemma *rank-Gauss-Jordan-code*[*code*]:
fixes *A*::*′a*::{*field*}^*′n*::{*mod-type*}^*′m*::{*mod-type*}
shows *rank A* = (*if A* = *0 then 0 else* (*let A′*=(*Gauss-Jordan A*) *in to-nat* (*GREATEST a. row a A′* ≠ *0*) + *1*))
⟨*proof*⟩

lemma *dim-null-space*[*code-unfold*]:
fixes *A*::*′a*::{*field*}^*′cols*::{*mod-type*}^*′rows*::{*mod-type*}
shows *vec.dim* (*null-space A*) = (*vec.dimension TYPE*(*′a*) *TYPE*(*′cols*)) − *rank* (*A*)
⟨*proof*⟩

lemma *rank-eq-dim-col-space′*[*code-unfold*]:
fixes *A*::*′a*::{*field*}^*′cols*::{*mod-type*}^*′rows*::{*mod-type*}
shows *vec.dim* (*col-space A*) = *rank A* ⟨*proof*⟩

lemma *dim-left-null-space*[*code-unfold*]:
fixes *A*::*′a*::{*field*}^*′cols*::{*mod-type*}^*′rows*::{*mod-type*}
shows *vec.dim* (*left-null-space A*) = (*vec.dimension TYPE*(*′a*) *TYPE*(*′rows*)) − *rank* (*A*)
⟨*proof*⟩

lemmas *rank-col-rank*[*symmetric*, *code-unfold*]
lemmas *rank-def*[*symmetric*, *code-unfold*]
lemmas *row-rank-def*[*symmetric*, *code-unfold*]
lemmas *col-rank-def*[*symmetric*, *code-unfold*]
lemmas *DIM-cart*[*code-unfold*]
lemmas *DIM-real*[*code-unfold*]

end

7 Linear Maps

theory *Linear-Maps*
imports
Gauss-Jordan
begin

lemma ((λ(*x*, *y*). (*x*::*real* , − *y*::*real*)) *has-derivative* (λ*h*. (*fst h*, − *snd h*))) (*at x*)

⟨*proof*⟩

7.1 Properties about ranks and linear maps

lemma *rank-matrix-dim-range*:
assumes *lf*: *linear* ((∗*s*)) ((∗*s*)) *f*
shows *rank* (*matrix f*::′*a*::{*field*}^′*cols*::{*mod-type*}^′*rows*::{*mod-type*}) = *vec.dim* (*range f*)
⟨*proof*⟩

The following two lemmas are the demonstration of theorem 2.11 that appears the book "Advanced Linear Algebra" by Steven Roman.

lemma *linear-injective-rank-eq-ncols*:
assumes *lf*: *linear* ((∗*s*)) ((∗*s*)) *f*
shows *inj f* ⟷ *rank* (*matrix f*::′*a*::{*field*}^′*cols*::{*mod-type*}^′*rows*::{*mod-type*}) = *ncols* (*matrix f*)
⟨*proof*⟩

lemma *linear-surjective-rank-eq-ncols*:
assumes *lf*: *linear* ((∗*s*)) ((∗*s*)) *f*
shows *surj f* ⟷ *rank* (*matrix f*::′*a*::{*field*}^′*cols*::{*mod-type*}^′*rows*::{*mod-type*}) = *nrows* (*matrix f*)
⟨*proof*⟩

lemma *linear-bij-rank-eq-ncols*:
fixes *f*::(′*a*::{*field*}^′*n*::{*mod-type*})=>(′*a*::{*field*}^′*n*::{*mod-type*})
assumes *lf*: *linear* ((∗*s*)) ((∗*s*)) *f*
shows *bij f* ⟷ *rank* (*matrix f*) = *ncols* (*matrix f*)
⟨*proof*⟩

7.2 Invertible linear maps

locale *invertible-lf* = *Vector-Spaces.linear* +
assumes *invertible*: (∃ *g*. *g* ∘ *f* = *id* ∧ *f* ∘ *g* = *id*)
begin

lemma *invertible-lf*: (∃ *g*. *linear* ((∗*b*)) ((∗*a*)) *g* ∧ (*g* ∘ *f* = *id*) ∧ (*f* ∘ *g* = *id*))
⟨*proof*⟩

end

lemma (**in** *Vector-Spaces.linear*) *invertible-lf-intro*[*intro*]:
assumes (*g* ∘ *f* = *id*) **and** (*f* ∘ *g* = *id*)
shows *invertible-lf* ((∗*a*)) ((∗*b*)) *f*
⟨*proof*⟩

lemma *invertible-imp-bijective*:
assumes *invertible-lf scaleB scaleC f*
shows *bij f*

⟨*proof*⟩

lemma *invertible-matrix-imp-invertible-lf*:
fixes $A::'a::\{field\}\hat{}'n\hat{}'n$
assumes *invertible-A*: *invertible A*
shows *invertible-lf* $((*s))$ $((*s))$ $(\lambda x.\ A *v x)$
⟨*proof*⟩

lemma *invertible-lf-imp-invertible-matrix*:
fixes $f::'a::\{field\}\hat{}'n \Rightarrow 'a\hat{}'n$
assumes *invertible-f*: *invertible-lf* $((*s))$ $((*s))$ *f*
shows *invertible* (*matrix f*)
⟨*proof*⟩

lemma *invertible-matrix-iff-invertible-lf*:
fixes $A::'a::\{field\}\hat{}'n\hat{}'n$
shows *invertible A* $\longleftrightarrow$ *invertible-lf* $((*s))$ $((*s))$ $(\lambda x.\ A *v x)$
⟨*proof*⟩

lemma *invertible-matrix-iff-invertible-lf'*:
fixes $f::'a::\{field\}\hat{}'n \Rightarrow 'a\hat{}'n$
assumes *linear-f*: *linear* $((*s))$ $((*s))$ *f*
shows *invertible* (*matrix f*) $\longleftrightarrow$ *invertible-lf* $((*s))$ $((*s))$ *f*
⟨*proof*⟩

lemma *invertible-matrix-mult-right-rank*:
fixes $A::'a::\{field\}\hat{}'n::\{mod\text{-}type\}\hat{}'m::\{mod\text{-}type\}$
and $Q::'a::\{field\}\hat{}'n::\{mod\text{-}type\}\hat{}'n::\{mod\text{-}type\}$
assumes *invertible-Q*: *invertible Q*
shows *rank* $(A**Q) =$ *rank A*
⟨*proof*⟩

lemma *subspace-image-invertible-mat*:
fixes $P::'a::\{field\}\hat{}'m\hat{}'m$
assumes *inv-P*: *invertible P*
and *sub-W*: *vec.subspace W*
shows *vec.subspace* $((\lambda x.\ P *v x)`\ W)$
⟨*proof*⟩

lemma *dim-image-invertible-mat*:
fixes $P::'a::\{field\}\hat{}'m\hat{}'m$
assumes *inv-P*: *invertible P*
and *sub-W*: *vec.subspace W*
shows *vec.dim* $((\lambda x.\ P *v x)`\ W) =$ *vec.dim W*
⟨*proof*⟩

lemma *invertible-matrix-mult-left-rank*:

fixes $A::'a::\{field\}\hat{}'n::\{mod\text{-}type\}\hat{}'m::\{mod\text{-}type\}$
and $P::'a::\{field\}\hat{}'m::\{mod\text{-}type\}\hat{}'m::\{mod\text{-}type\}$
assumes *invertible-P*: *invertible P*
shows *rank* $(P{**}A) =$ *rank A*
⟨*proof*⟩

corollary *invertible-matrices-mult-rank*:
fixes $A::'a::\{field\}\hat{}'n::\{mod\text{-}type\}\hat{}'m::\{mod\text{-}type\}$
and $P::'a\hat{}'m::\{mod\text{-}type\}\hat{}'m::\{mod\text{-}type\}$ **and** $Q::'a\hat{}'n::\{mod\text{-}type\}\hat{}'n::\{mod\text{-}type\}$
assumes *invertible-P*: *invertible P*
and *invertible-Q*: *invertible Q*
shows *rank* $(P{**}A{**}Q) =$ *rank A*
⟨*proof*⟩

lemma *invertible-matrix-mult-left-rank'*:
fixes $A::'a::\{field\}\hat{}'n::\{mod\text{-}type\}\hat{}'m::\{mod\text{-}type\}$ **and** $P::'a\hat{}'m::\{mod\text{-}type\}\hat{}'m::\{mod\text{-}type\}$
assumes *invertible-P*: *invertible P* **and** *B-eq-PA*: $B{=}P{**}A$
shows *rank B* = *rank A*
⟨*proof*⟩

lemma *invertible-matrix-mult-right-rank'*:
fixes $A::'a::\{field\}\hat{}'n::\{mod\text{-}type\}\hat{}'m::\{mod\text{-}type\}$
and $Q::'a::\{field\}\hat{}'n::\{mod\text{-}type\}\hat{}'n::\{mod\text{-}type\}$
assumes *invertible-Q*: *invertible Q* **and** *B-eq-PA*: $B{=}A{**}Q$
shows *rank B* = *rank A* ⟨*proof*⟩

lemma *invertible-matrices-rank'*:
fixes $A::'a::\{field\}\hat{}'n::\{mod\text{-}type\}\hat{}'m::\{mod\text{-}type\}$
and $P::'a\hat{}'m::\{mod\text{-}type\}\hat{}'m::\{mod\text{-}type\}$ **and** $Q::'a\hat{}'n::\{mod\text{-}type\}\hat{}'n::\{mod\text{-}type\}$
assumes *invertible-P*: *invertible P* **and** *invertible-Q*: *invertible Q* **and** *B-eq-PA*:
$B = P{**}A{**}Q$
shows *rank B* = *rank A* ⟨*proof*⟩

7.3 Definition and properties of the set of a vector

Some definitions:

In the file *Generalizations.thy* there exists the following definition: *cart-basis* = $\{axis\ i\ 1\ |i.\ i \in UNIV\}$.

cart-basis returns a set which is a basis and it works properly in my development. But in this file, I need to know the order of the elements of the basis, because is very important for the coordenates of a vector and the matrices of change of bases. So, I have defined a new *cart-basis'*, which will be a matrix. The columns of this matrix are the elements of the basis.

definition *set-of-vector* :: $'a\hat{}'n \Rightarrow 'a\ set$
where *set-of-vector A* = $\{A\ \$\ i\ |i.\ i \in UNIV\}$

definition *cart-basis'* :: *'a*::{*field*}^'*n*^'*n*
where *cart-basis'* = (*χ i. axis i 1*)

lemma *cart-basis-eq-set-of-vector-cart-basis'*:
cart-basis = *set-of-vector* (*cart-basis'*)
⟨*proof*⟩

lemma *basis-image-linear*:
fixes *f*::*'b*::{*field*}^'*n* => *'b*^'*n*
assumes *invertible-lf*: *invertible-lf* ((∗*s*)) ((∗*s*)) *f*
and *basis-X*: *is-basis* (*set-of-vector X*)
shows *is-basis* (*f*‘ (*set-of-vector X*))
⟨*proof*⟩

Properties about *cart-basis'* = (*χi. axis i 1*)

lemma *set-of-vector-cart-basis'*:
shows (*set-of-vector cart-basis'*) = {*axis i 1* :: *'a*::{*field*}^'*n* | *i. i* ∈ (*UNIV* :: *'n set*)}
⟨*proof*⟩

lemma *cart-basis'-i*: *cart-basis'* $ *i* = *axis i 1* ⟨*proof*⟩

lemma *finite-set-of-vector*[*intro*,*simp*]: *finite* (*set-of-vector X*)
⟨*proof*⟩

lemma *is-basis-cart-basis'*: *is-basis* (*set-of-vector cart-basis'*)
⟨*proof*⟩

lemma *basis-expansion-cart-basis'*:*sum* (*λi. x*$*i* ∗*s cart-basis'* $ *i*) *UNIV* = *x*
⟨*proof*⟩

lemma *basis-expansion-unique*:
sum (*λi. f i* ∗*s axis* (*i*::*'n*::*finite*) *1*) *UNIV* = (*x*::(*'a*::*comm-ring-1*) ^'*n*) ⟷ (∀ *i. f i* = *x*$*i*)
⟨*proof*⟩

lemma *basis-expansion-cart-basis'-unique*: *sum* (*λi. f* (*cart-basis'* $ *i*) ∗*s cart-basis'* $ *i*) *UNIV* = *x* ⟷ (∀ *i. f* (*cart-basis'* $ *i*) = *x*$*i*)
⟨*proof*⟩

lemma *basis-expansion-cart-basis'-unique'*: *sum* (*λi. f i* ∗*s cart-basis'* $ *i*) *UNIV* = *x* ⟷ (∀ *i. f i* = *x*$*i*)
⟨*proof*⟩

Properties of *is-basis ?S* ≡ *vec.independent ?S* ∧ *vec.span ?S* = *UNIV*.

lemma *sum-basis-eq*:
fixes *X*::*'a*::{*field*}^'*n*^'*n*
assumes *is-basis*:*is-basis* (*set-of-vector X*)

shows *sum* ($\lambda x.\ f\ x *s\ x$) (*set-of-vector* X) = *sum* ($\lambda i.\ f$ ($X\$i$) $*s$ ($X\$i$)) *UNIV*
⟨*proof*⟩

corollary *sum-basis-eq2*:
fixes $X::'a::\{field\}\hat{}'n\hat{}n$
assumes *is-basis*:*is-basis* (*set-of-vector* X)
shows *sum* ($\lambda x.\ f\ x *s\ x$) (*set-of-vector* X) = *sum* ($\lambda i.$ ($f \circ (\$)\ X$) $i *s$ ($X\$i$)) *UNIV*
⟨*proof*⟩

lemma *inj-op-nth*:
fixes $X::'a::\{field\}\hat{}'n\hat{}n$
assumes *is-basis*: *is-basis* (*set-of-vector* X)
shows *inj* ((\$) X)
⟨*proof*⟩

lemma *basis-UNIV*:
fixes $X::'a::\{field\}\hat{}'n\hat{}n$
assumes *is-basis*: *is-basis* (*set-of-vector* X)
shows $UNIV = \{x.\ \exists g.\ (\sum i{\in}UNIV.\ g\ i *s\ X\$i) = x\}$
⟨*proof*⟩

lemma *scalars-zero-if-basis*:
fixes $X::'a::\{field\}\hat{}'n\hat{}n$
assumes *is-basis*: *is-basis* (*set-of-vector* X) **and** *sum*: $(\sum i{\in}(UNIV::'n\ set).\ f\ i *s\ X\$i) = 0$
shows $\forall\, i{\in}(UNIV::'n\ set).\ f\ i = 0$
⟨*proof*⟩

lemma *basis-combination-unique*:
fixes $X::'a::\{field\}\hat{}'n\hat{}n$
assumes *basis-X*: *is-basis* (*set-of-vector* X) **and** *sum-eq*: $(\sum i{\in}UNIV.\ g\ i *s\ X\$i) = (\sum i{\in}UNIV.\ f\ i *s\ X\$i)$
shows $f{=}g$
⟨*proof*⟩

7.4 Coordinates of a vector

Definition and properties of the coordinates of a vector (in terms of a particular ordered basis).

definition *coord* :: $'a::\{field\}\hat{}'n\hat{}n \Rightarrow 'a::\{field\}\hat{}'n \Rightarrow 'a::\{field\}\hat{}n$
where *coord* $X\ v = (\chi\ i.\ (THE\ f.\ v = sum\ (\lambda x.\ f\ x *s\ X\$x)\ UNIV)\ i)$

coord $X\ v$ are the coordinates of vector v with respect to the basis X

lemma *bij-coord*:
fixes $X::'a::\{field\}\hat{}'n\hat{}n$
assumes *basis-X*: *is-basis* (*set-of-vector* X)
shows *bij* (*coord* X)

⟨*proof*⟩

lemma *linear-coord*:
fixes $X::'a::\{field\}\hat{}'n\hat{}'n$
assumes *basis-X*: *is-basis* (*set-of-vector X*)
shows *linear* ((∗*s*)) ((∗*s*)) (*coord X*)
⟨*proof*⟩

lemma *coord-eq*:
assumes *basis-X*:*is-basis* (*set-of-vector X*)
and *coord-eq*: *coord X v* = *coord X w*
shows *v* = *w*
⟨*proof*⟩

7.5 Matrix of change of basis and coordinate matrix of a linear map

Definitions of matrix of change of basis and matrix of a linear transformation with respect to two bases:

definition *matrix-change-of-basis* :: $'a::\{field\}\hat{}'n\hat{}'n \Rightarrow 'a\hat{}'n\hat{}'n \Rightarrow 'a\hat{}'n\hat{}'n$
where *matrix-change-of-basis X Y* = (χ *i j*. (*coord Y* (*X*$*j*)) $ *i*)

There exists in the library the definition *matrix ?f* = (χ*i j*. *?f* (*axis j 1*) $ *i*), which is the coordinate matrix of a linear map with respect to the standard bases. Now we generalise that concept to the coordinate matrix of a linear map with respect to any two bases.

definition *matrix′* :: $'a::\{field\}\hat{}'n\hat{}'n \Rightarrow 'a\hat{}'m\hat{}'m \Rightarrow ('a\hat{}'n => 'a\hat{}'m) \Rightarrow 'a\hat{}'n\hat{}'m$
where *matrix′ X Y f*= (χ *i j*. (*coord Y* (*f*(*X*$*j*))) $ *i*)

Properties of *matrix′ ?X ?Y ?f* = (χ*i j*. *coord ?Y* (*?f* (*?X* $ *j*)) $ *i*)

lemma *matrix′-eq-matrix*:
defines *cart-basis-Rn*: *cart-basis-Rn* == (*cart-basis′*)::$'a::\{field\}\hat{}'n\hat{}'n$
and *cart-basis-Rm*:*cart-basis-Rm* == (*cart-basis′*)::$'a\hat{}'m\hat{}'m$
shows *matrix′* (*cart-basis-Rn*) (*cart-basis-Rm*) *f* = *matrix f*
⟨*proof*⟩

lemma *matrix′*:
assumes *basis-X*: *is-basis* (*set-of-vector X*) **and** *basis-Y*: *is-basis* (*set-of-vector Y*)
shows *f* (*X*$*i*) = *sum* (λ*j*. (*matrix′ X Y f*) $ *j* $ *i* ∗*s* (*Y*$*j*)) *UNIV*
⟨*proof*⟩

corollary *matrix′2*:
assumes *basis-X*: *is-basis* (*set-of-vector X*) **and** *basis-Y*: *is-basis* (*set-of-vector Y*)
and *eq-f*: ∀ *i*. *f* (*X*$*i*) = *sum* (λ*j*. *A* $ *j* $ *i* ∗*s* (*Y*$*j*)) *UNIV*

shows *matrix′ X Y f* = *A*
⟨*proof*⟩

This is the theorem 2.14 in the book "Advanced Linear Algebra" by Steven Roman.

lemma *coord-matrix′*:
fixes *X*::′*a*::{*field*}^′*n*^′*n* **and** *Y*::′*a*^′*m*^′*m*
assumes *basis-X*: *is-basis* (*set-of-vector X*) **and** *basis-Y*: *is-basis* (*set-of-vector Y*)
and *linear-f*: *linear* ((∗*s*)) ((∗*s*)) *f*
shows *coord Y* (*f v*) = (*matrix′ X Y f*) ∗*v* (*coord X v*)
⟨*proof*⟩

This is the second part of the theorem 2.15 in the book "Advanced Linear Algebra" by Steven Roman.

lemma *matrix′-compose*:
fixes *X*::′*a*::{*field*}^′*n*^′*n* **and** *Y*::′*a*^′*m*^′*m* **and** *Z*::′*a*^′*p*^′*p*
assumes *basis-X*: *is-basis* (*set-of-vector X*) **and** *basis-Y*: *is-basis* (*set-of-vector Y*) **and** *basis-Z*: *is-basis* (*set-of-vector Z*)
and *linear-f*: *linear* ((∗*s*)) ((∗*s*)) *f* **and** *linear-g*: *linear* ((∗*s*)) ((∗*s*)) *g*
shows *matrix′ X Z* (*g* ∘ *f*) = (*matrix′ Y Z g*) ∗∗ (*matrix′ X Y f*)
⟨*proof*⟩

lemma *exists-linear-eq-matrix′*:
fixes *A*::′*a*::{*field*}^′*m*^′*n* **and** *X*::′*a*^′*m*^′*m* **and** *Y*::′*a*^′*n*^′*n*
assumes *basis-X*: *is-basis* (*set-of-vector X*) **and** *basis-Y*: *is-basis* (*set-of-vector Y*)
shows ∃*f*. *matrix′ X Y f* = *A* ∧ *linear* ((∗*s*)) ((∗*s*)) *f*
⟨*proof*⟩

lemma *matrix′-surj*:
assumes *basis-X*: *is-basis* (*set-of-vector X*) **and** *basis-Y*: *is-basis* (*set-of-vector Y*)
shows *surj* (*matrix′ X Y*)
⟨*proof*⟩

Properties of *matrix-change-of-basis ?X ?Y* = (χ*i j*. *coord ?Y* (*?X* $ *j*) $ *i*).

This is the first part of the theorem 2.12 in the book "Advanced Linear Algebra" by Steven Roman.

lemma *matrix-change-of-basis-works*:
fixes *X*::′*a*::{*field*}^′*n*^′*n* **and** *Y*::′*a*^′*n*^′*n*
assumes *basis-X*: *is-basis* (*set-of-vector X*)
and *basis-Y*: *is-basis* (*set-of-vector Y*)
shows (*matrix-change-of-basis X Y*) ∗*v* (*coord X v*) = (*coord Y v*)

⟨*proof*⟩

lemma *matrix-change-of-basis-mat-1*:
fixes *X*::*'a*::{*field*}^*'n*^*n*
assumes *basis-X*: *is-basis* (*set-of-vector X*)
shows *matrix-change-of-basis X X* = *mat 1*
⟨*proof*⟩

Relationships between *matrix' ?X ?Y ?f* = (χ*i j*. *coord ?Y* (*?f* (*?X* $ *j*)) $ *i*) and *matrix-change-of-basis ?X ?Y* = (χ*i j*. *coord ?Y* (*?X* $ *j*) $ *i*). This is the theorem 2.16 in the book "Advanced Linear Algebra" by Steven Roman.

lemma *matrix'-matrix-change-of-basis*:
fixes *B*::*'a*::{*field*}^*'n*^*n* **and** *B'*::*'a*^*'n*^*n* **and** *C*::*'a*^*'m*^*m* **and** *C'*::*'a*^*'m*^*m*
assumes *basis-B*: *is-basis* (*set-of-vector B*) **and** *basis-B'*: *is-basis* (*set-of-vector B'*)
and *basis-C*: *is-basis* (*set-of-vector C*) **and** *basis-C'*: *is-basis* (*set-of-vector C'*)
and *linear-f*: *linear* ((∗*s*)) ((∗*s*)) *f*
shows *matrix' B' C' f* = *matrix-change-of-basis C C'* ∗∗ *matrix' B C f* ∗∗ *matrix-change-of-basis B' B*
⟨*proof*⟩

lemma *matrix'-id-eq-matrix-change-of-basis*:
fixes *X*::*'a*::{*field*}^*'n*^*n* **and** *Y*::*'a*^*'n*^*n*
assumes *basis-X*: *is-basis* (*set-of-vector X*) **and** *basis-Y*: *is-basis* (*set-of-vector Y*)
shows *matrix' X Y* (*id*) = *matrix-change-of-basis X Y*
⟨*proof*⟩

Relationships among *invertible-lf ?s1.0 ?s2.0 ?f* ≡ *linear ?s1.0 ?s2.0 ?f* ∧ *invertible-lf-axioms ?f*, *matrix-change-of-basis ?X ?Y* = (χ*i j*. *coord ?Y* (*?X* $ *j*) $ *i*), *matrix' ?X ?Y ?f* = (χ*i j*. *coord ?Y* (*?f* (*?X* $ *j*)) $ *i*) and *invertible ?A* = (∃ *A'*. *?A* ∗∗ *A'* = *mat 1* ∧ *A'* ∗∗ *?A* = *mat 1*).

This is the second part of the theorem 2.12 in the book "Advanced Linear Algebra" by Steven Roman.

lemma *matrix-inv-matrix-change-of-basis*:
fixes *X*::*'a*::{*field*}^*'n*^*n* **and** *Y*::*'a*^*'n*^*n*
assumes *basis-X*: *is-basis* (*set-of-vector X*) **and** *basis-Y*: *is-basis* (*set-of-vector Y*)
shows*matrix-change-of-basis Y X* = *matrix-inv* (*matrix-change-of-basis X Y*)
⟨*proof*⟩

The following four lemmas are the proof of the theorem 2.13 in the book "Advanced Linear Algebra" by Steven Roman.

corollary *invertible-matrix-change-of-basis*:

fixes $X::'a::\{field\}\hat{}'n\hat{}'n$ **and** $Y::'a\hat{}'n\hat{}'n$
assumes *basis-X*: *is-basis* (*set-of-vector X*) **and** *basis-Y*: *is-basis* (*set-of-vector Y*)
shows *invertible* (*matrix-change-of-basis X Y*)
⟨*proof*⟩

lemma *invertible-lf-imp-invertible-matrix'*:
fixes $f::'a::\{field\}\hat{}'b \Rightarrow 'a\hat{}'b$
assumes *invertible-lf* ((∗*s*)) ((∗*s*)) *f* **and** *basis-X*: *is-basis* (*set-of-vector X*) **and** *basis-Y*: *is-basis* (*set-of-vector Y*)
shows *invertible* (*matrix' X Y f*)
⟨*proof*⟩

lemma *invertible-matrix'-imp-invertible-lf*:
fixes $f::'a::\{field\}\hat{}'b \Rightarrow 'a\hat{}'b$
assumes *invertible* (*matrix' X Y f*) **and** *basis-X*: *is-basis* (*set-of-vector X*)
and *linear-f*: *linear* ((∗*s*)) ((∗*s*)) *f* **and** *basis-Y*: *is-basis* (*set-of-vector Y*)
shows *invertible-lf* ((∗*s*)) ((∗*s*)) *f*
⟨*proof*⟩

lemma *invertible-matrix-is-change-of-basis*:
assumes *invertible-P*: *invertible P* **and** *basis-X*: *is-basis* (*set-of-vector X*)
shows ∃!*Y*. *matrix-change-of-basis Y X* = *P* ∧ *is-basis* (*set-of-vector Y*)
⟨*proof*⟩

7.6 Equivalent Matrices

Next definition follows the one presented in Modern Algebra by Seth Warner.

definition *equivalent-matrices A B* = (∃ *P Q*. *invertible P* ∧ *invertible Q* ∧ *B* = (*matrix-inv P*)∗∗*A*∗∗*Q*)

lemma *exists-basis*: ∃ *X*::'*a*::{*field*}^'*n*^'*n*. *is-basis* (*set-of-vector X*)
⟨*proof*⟩

lemma *equivalent-implies-exist-matrix'*:
assumes *equivalent*: *equivalent-matrices A B*
shows ∃ *X Y X' Y' f*::'*a*::{*field*}^'*n*⇒'*a*^'*m*.
linear ((∗*s*)) ((∗*s*)) *f* ∧ *matrix' X Y f* = *A* ∧ *matrix' X' Y' f* = *B* ∧ *is-basis* (*set-of-vector X*)
∧ *is-basis* (*set-of-vector Y*) ∧ *is-basis* (*set-of-vector X'*) ∧ *is-basis* (*set-of-vector Y'*)
⟨*proof*⟩

lemma *exist-matrix'-implies-equivalent*:
assumes *A*: *matrix' X Y f* = *A*
and *B*: *matrix' X' Y' f* = *B*
and *X*: *is-basis* (*set-of-vector X*)
and *Y*: *is-basis* (*set-of-vector Y*)

and X': *is-basis* (*set-of-vector* X')
and Y': *is-basis* (*set-of-vector* Y')
and *linear-f*: *linear* $((*s))$ $((*s))$ f
shows *equivalent-matrices* A B
⟨*proof*⟩

This is the proof of the theorem 2.18 in the book "Advanced Linear Algebra" by Steven Roman.

corollary *equivalent-iff-exist-matrix'*:
shows *equivalent-matrices* A $B \longleftrightarrow (\exists X\ Y\ X'\ Y'\ f::'a::\{field\}\hat{}'n \Rightarrow 'a\hat{}'m.$
linear $((*s))$ $((*s))$ $f \wedge$ *matrix'* X Y $f = A \wedge$ *matrix'* X' Y' $f = B$
$\wedge$ *is-basis* (*set-of-vector* X) $\wedge$ *is-basis* (*set-of-vector* Y)
$\wedge$ *is-basis* (*set-of-vector* X') $\wedge$ *is-basis* (*set-of-vector* Y'))
⟨*proof*⟩

7.7 Similar matrices

definition *similar-matrices* :: $'a::\{semiring\text{-}1\}\hat{}'n\hat{}'n \Rightarrow 'a::\{semiring\text{-}1\}\hat{}'n\hat{}'n \Rightarrow$ *bool*
where *similar-matrices* A $B = (\exists P.$ *invertible* $P \wedge B=$(*matrix-inv* P)$**A**$$P$)

lemma *similar-implies-exist-matrix'*:
fixes A $B::'a::\{field\}\hat{}'n\hat{}'n$
assumes *similar*: *similar-matrices* A B
shows $\exists X\ Y\ f.$ *linear* $((*s))$ $((*s))$ $f \wedge$ *matrix'* X X $f = A \wedge$ *matrix'* Y Y $f = B$
$\wedge$ *is-basis* (*set-of-vector* X) $\wedge$ *is-basis* (*set-of-vector* Y)
⟨*proof*⟩

lemma *exist-matrix'-implies-similar*:
fixes A $B::'a::\{field\}\hat{}'n\hat{}'n$
assumes *linear-f*: *linear* $((*s))$ $((*s))$ f **and** A: *matrix'* X X $f = A$ **and** B: *matrix'* Y Y $f = B$
and X: *is-basis* (*set-of-vector* X) **and** Y: *is-basis* (*set-of-vector* Y)
shows *similar-matrices* A B
⟨*proof*⟩

This is the proof of the theorem 2.19 in the book "Advanced Linear Algebra" by Steven Roman.

corollary *similar-iff-exist-matrix'*:
fixes A $B::'a::\{field\}\hat{}'n\hat{}'n$
shows *similar-matrices* A $B \longleftrightarrow (\exists X\ Y\ f.$ *linear* $((*s))$ $((*s))$ $f \wedge$ *matrix'* X X $f = A$
$\wedge$ *matrix'* Y Y $f = B \wedge$ *is-basis* (*set-of-vector* X) $\wedge$ *is-basis* (*set-of-vector* Y))
⟨*proof*⟩

end

8 Obtaining explicitly the invertible matrix which transforms a matrix to its reduced row echelon form

theory *Gauss-Jordan-PA*
imports
Gauss-Jordan
Rank-Nullity-Theorem.Miscellaneous
Linear-Maps
begin

8.1 Definitions

The following algorithm is similar to *Gauss-Jordan*, but in this case we will also return the P matrix which makes *Gauss-Jordan* $A = P ** A$. If A is invertible, this matrix P will be the inverse of it.

definition *Gauss-Jordan-in-ij-PA* :: (('*a*::{*semiring-1*, *inverse*, *one*, *uminus*}^'*rows*::{*finite*, *ord*}^'*rows*::{*finite*, *ord*}) × ('*a*^'*cols*^'*rows*::{*finite*, *ord*})) => '*rows*=>'*cols* =>(('*a*^'*rows*::{*finite*, *ord*}^'*rows*::{*finite*, *ord*}) × ('*a*^'*cols*^'*rows*::{*finite*, *ord*}))
where *Gauss-Jordan-in-ij-PA A' i j* = (*let P*=*fst A'*; *A*=*snd A'*;
n = (*LEAST n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*);
interchange-A = (*interchange-rows A i n*);
interchange-P = (*interchange-rows P i n*);
P' = *mult-row interchange-P i* (*1*/*interchange-A*$*i*$*j*)
in
(*vec-lambda*(% *s*. *if s*=*i then P'* $ *s else* (*row-add P' s i* (−(*interchange-A*$*s*$*j*))) $ *s*), *Gauss-Jordan-in-ij A i j*))

definition *Gauss-Jordan-column-k-PA*
where *Gauss-Jordan-column-k-PA A' k* =
(*let P* = *fst A'*;
i = *fst* (*snd A'*);
A = *snd* (*snd A'*);
from-nat-i=*from-nat i*;
from-nat-k=*from-nat k*
in
if (∀ *m*≥*from-nat-i*. *A* $ *m* $ *from-nat-k* = *0*) ∨ *i* = *nrows A then* (*P*, *i*, *A*)
else (*let Gauss* = *Gauss-Jordan-in-ij-PA* (*P*,*A*) (*from-nat-i*) (*from-nat-k*)
in (*fst Gauss*, *i* + *1*, *snd Gauss*)))

definition *Gauss-Jordan-upt-k-PA A k* = (*let foldl*=(*foldl Gauss-Jordan-column-k-PA* (*mat 1*,*0*, *A*) [*0*..<*Suc k*]) *in* (*fst foldl*, *snd* (*snd foldl*)))
definition *Gauss-Jordan-PA A* = *Gauss-Jordan-upt-k-PA A* (*ncols A* − *1*)

8.2 Proofs

8.2.1 Properties about *Gauss-Jordan-in-ij-PA*

The following lemmas are very important in order to improve the efficience of the code

We define the following function to obtain an efficient code for *Gauss-Jordan-in-ij-PA A i j*.

definition *Gauss-Jordan-wrapper i j A B = vec-lambda*(%*s. if s=i then A $ s else* (*row-add A s i* (−(*B*$*s*$*j*))) $ *s*)

lemma *Gauss-Jordan-wrapper-code*[*code abstract*]:
vec-nth (*Gauss-Jordan-wrapper i j A B*) = (%*s. if s=i then A $ s else* (*row-add A s i* (−(*B*$*s*$*j*))) $ *s*)
⟨*proof*⟩

lemma *Gauss-Jordan-in-ij-PA-def'*[*code*]:
Gauss-Jordan-in-ij-PA A' i j = (*let P*=*fst A'*; *A*=*snd A'*;
n = (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*);
interchange-A = (*interchange-rows A i n*);
A' = *mult-row interchange-A i* (*1*/*interchange-A*$*i*$*j*);
interchange-P = (*interchange-rows P i n*);
P' = *mult-row interchange-P i* (*1*/*interchange-A*$*i*$*j*)
in
(*Gauss-Jordan-wrapper i j P' interchange-A*,
Gauss-Jordan-wrapper i j A' interchange-A))
⟨*proof*⟩

The second component is equal to *Gauss-Jordan-in-ij*

lemma *snd-Gauss-Jordan-in-ij-PA-eq*[*code-unfold*]: *snd* (*Gauss-Jordan-in-ij-PA* (*P*,*A*) *i j*) = *Gauss-Jordan-in-ij A i j*
⟨*proof*⟩

lemma *fst-Gauss-Jordan-in-ij-PA*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *PB-A*: *P* ** *B* = *A*
shows *fst* (*Gauss-Jordan-in-ij-PA* (*P*,*A*) *i j*) ** *B* = *snd* (*Gauss-Jordan-in-ij-PA* (*P*,*A*) *i j*)
⟨*proof*⟩

8.2.2 Properties about *Gauss-Jordan-column-k-PA*

lemma *fst-Gauss-Jordan-column-k*:
assumes *i*≤*nrows A*
shows *fst* (*Gauss-Jordan-column-k* (*i*, *A*) *k*) ≤ *nrows A*
⟨*proof*⟩

lemma *fst-Gauss-Jordan-column-k-PA*:

fixes $A::'a::\{field\}^{'}cols::\{mod\text{-}type\}^{'}rows::\{mod\text{-}type\}$
assumes *PB-A*: $P ** B = A$
shows *fst* (*Gauss-Jordan-column-k-PA* (*P*,*i*,*A*) *k*) ** *B* = *snd* (*snd* (*Gauss-Jordan-column-k-PA* (*P*,*i*,*A*) *k*))
⟨*proof*⟩

lemma *snd-snd-Gauss-Jordan-column-k-PA-eq*:
shows *snd* (*snd* (*Gauss-Jordan-column-k-PA* (*P*,*i*,*A*) *k*)) = *snd* (*Gauss-Jordan-column-k* (*i*,*A*) *k*)
⟨*proof*⟩

lemma *fst-snd-Gauss-Jordan-column-k-PA-eq*:
shows *fst* (*snd* (*Gauss-Jordan-column-k-PA* (*P*,*i*,*A*) *k*)) = *fst* (*Gauss-Jordan-column-k* (*i*,*A*) *k*)
⟨*proof*⟩

8.2.3 Properties about *Gauss-Jordan-upt-k-PA*

lemma *fst-Gauss-Jordan-upt-k-PA*:
fixes $A::'a::\{field\}^{'}cols::\{mod\text{-}type\}^{'}rows::\{mod\text{-}type\}$
shows *fst* (*Gauss-Jordan-upt-k-PA A k*) ** *A* = *snd* (*Gauss-Jordan-upt-k-PA A k*)
⟨*proof*⟩

lemma *snd-foldl-Gauss-Jordan-column-k-eq*:
snd (*foldl Gauss-Jordan-column-k-PA* (*mat 1*, *0*, *A*) [*0*..<*k*]) = *foldl Gauss-Jordan-column-k* (*0*, *A*) [*0*..<*k*]
⟨*proof*⟩

lemma *snd-Gauss-Jordan-upt-k-PA*:
shows *snd* (*Gauss-Jordan-upt-k-PA A k*) = (*Gauss-Jordan-upt-k A k*)
⟨*proof*⟩

8.2.4 Properties about *Gauss-Jordan-PA*

lemma *fst-Gauss-Jordan-PA*:
fixes $A::'a::\{field\}^{'}cols::\{mod\text{-}type\}^{'}rows::\{mod\text{-}type\}$
shows *fst* (*Gauss-Jordan-PA A*) ** *A* = *snd* (*Gauss-Jordan-PA A*)
⟨*proof*⟩

lemma *Gauss-Jordan-PA-eq*:
shows *snd* (*Gauss-Jordan-PA A*)= (*Gauss-Jordan A*)
⟨*proof*⟩

8.2.5 Proving that the transformation has been carried out by means of elementary operations

This function is very similar to *row-add-iterate* one. It allows us to prove that *fst* (*Gauss-Jordan-PA A*) is an invertible matrix. Concretly, it has been

defined to demonstrate that *fst* (*Gauss-Jordan-PA A*) has been obtained by means of elementary operations applied to the identity matrix

fun *row-add-iterate-PA* :: (('*a*::{*semiring-1*, *uminus*}^'*m*::{*mod-type*} ^'*m*::{*mod-type*}) × ('*a*^'*n*^'*m*::{*mod-type*}))=> *nat* => '*m* => '*n* =>
((('*a*^'*m*::{*mod-type*} ^'*m*::{*mod-type*}) × ('*a*^'*n*^'*m*::{*mod-type*}))
where *row-add-iterate-PA* (*P*,*A*) *0 i j* = (*if i=0 then* (*P*,*A*) *else* (*row-add P 0 i* (−*A* $ *0* $ *j*), *row-add A 0 i* (−*A* $ *0* $ *j*)))
| *row-add-iterate-PA* (*P*,*A*) (*Suc n*) *i j* = (*if* (*Suc n* = *to-nat i*) *then row-add-iterate-PA* (*P*,*A*) *n i j*
else row-add-iterate-PA ((*row-add P* (*from-nat* (*Suc n*)) *i* (− *A* $ (*from-nat* (*Suc n*)) $ *j*)), (*row-add A* (*from-nat* (*Suc n*)) *i* (− *A* $ (*from-nat* (*Suc n*)) $ *j*))) *n i j*)

lemma *fst-row-add-iterate-PA-preserves-greater-than-n*:
assumes *n*: *n*<*nrows A*
and *a*: *to-nat a* > *n*
shows *fst* (*row-add-iterate-PA* (*P*,*A*) *n i j*) $ *a* $ *b* = *P* $ *a* $ *b*
⟨*proof*⟩

lemma *snd-row-add-iterate-PA-eq-row-add-iterate*:
shows *snd* (*row-add-iterate-PA* (*P*,*A*) *n i j*) = *row-add-iterate A n i j*
⟨*proof*⟩

lemma *row-add-iterate-PA-preserves-pivot-row*:
assumes *n*: *n*<*nrows A*
and *a*: *to-nat i* ≤ *n*
shows *fst* (*row-add-iterate-PA* (*P*,*A*) *n i j*) $ *i* $ *b* = *P* $ *i* $ *b*
⟨*proof*⟩

lemma *fst-row-add-iterate-PA-eq-row-add*:
fixes *A*::'*a*::{*ring-1*}^'*n*^'*m*::{*mod-type*}
assumes *a-not-i*: *a* ≠ *i*
and *n*: *n*<*nrows A*
and *to-nat a* ≤ *n*
shows *fst* (*row-add-iterate-PA* (*P*,*A*) *n i j*) $ *a* $ *b* = (*row-add P a i* (− *A* $ *a* $ *j*)) $ *a* $ *b*
⟨*proof*⟩

lemma *fst-row-add-iterate-PA-eq-fst-Gauss-Jordan-in-ij-PA*:
fixes *A*::'*a*::{*field*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
and *i*::'*rows* **and** *j*::'*cols*
and *P*::'*a*::{*field*}^'*rows*::{*mod-type*}^'*rows*::{*mod-type*}
defines *A'*: *A'*== *mult-row* (*interchange-rows A i* (*LEAST n*. *A* $ *n* $ *j* ≠ *0* ∧ *i*

$\leq n))$ i $(1$ / $(interchange\text{-}rows$ A i $(LEAST$ $n.$ A \$ n \$ $j \neq 0 \wedge i \leq n))$ \$ i \$ $j)$
defines P': $P' == mult\text{-}row$ $(interchange\text{-}rows$ P i $(LEAST$ $n.$ A \$ n \$ $j \neq 0 \wedge i$
$\leq n))$ i $(1$ / $(interchange\text{-}rows$ A i $(LEAST$ $n.$ A \$ n \$ $j \neq 0 \wedge i \leq n))$ \$ i \$ $j)$
shows fst $(row\text{-}add\text{-}iterate\text{-}PA$ (P',A') $(nrows$ $A - 1)$ i $j)$ $= fst$ $(Gauss\text{-}Jordan\text{-}in\text{-}ij\text{-}PA$
(P,A) i $j)$
⟨*proof*⟩

lemma *invertible-fst-row-add-iterate-PA*:
fixes $A::'a::\{ring\text{-}1\}\hat{}'n\hat{}'m::\{mod\text{-}type\}$
assumes n: $n<nrows$ A
and *inv-P*: *invertible* P
shows *invertible* $(fst$ $(row\text{-}add\text{-}iterate\text{-}PA$ (P,A) n i $j))$
⟨*proof*⟩

lemma *invertible-fst-Gauss-Jordan-in-ij-PA*:
fixes $A::'a::\{field\}\hat{}'n::\{mod\text{-}type\}\hat{}'m::\{mod\text{-}type\}$
assumes *inv-P*: *invertible* P
and *not-all-zero*: $\neg$ $(\forall$ $m \geq i.$ A \$ m \$ $j = 0)$
shows *invertible* $(fst$ $(Gauss\text{-}Jordan\text{-}in\text{-}ij\text{-}PA$ (P,A) i $j))$
⟨*proof*⟩

lemma *invertible-fst-Gauss-Jordan-column-k-PA*:
fixes $A::'a::\{field\}\hat{}'n::\{mod\text{-}type\}\hat{}'m::\{mod\text{-}type\}$
assumes *inv-P*: *invertible* P
shows *invertible* $(fst$ $(Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}PA$ (P,i,A) $k))$
⟨*proof*⟩

lemma *invertible-fst-Gauss-Jordan-upt-k-PA*:
fixes $A::'a::\{field\}\hat{}'n::\{mod\text{-}type\}\hat{}'m::\{mod\text{-}type\}$
shows *invertible* $(fst$ $(Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}PA$ A $k))$
⟨*proof*⟩

lemma *invertible-fst-Gauss-Jordan-PA*:
fixes $A::'a::\{field\}\hat{}'n::\{mod\text{-}type\}\hat{}'m::\{mod\text{-}type\}$
shows *invertible* $(fst$ $(Gauss\text{-}Jordan\text{-}PA$ $A))$
⟨*proof*⟩

definition $P\text{-}Gauss\text{-}Jordan$ $A = fst$ $(Gauss\text{-}Jordan\text{-}PA$ $A)$

end

9 Computing determinants of matrices using the Gauss Jordan algorithm

theory *Determinants2*

imports
Gauss-Jordan-PA
begin

9.1 Some previous properties

9.1.1 Relationships between determinants and elementary row operations

lemma *det-interchange-rows*:
shows *det* (*interchange-rows A i j*) = *of-int* (*if i* = *j then 1 else* −*1*) ∗ *det A*
⟨*proof*⟩

corollary *det-interchange-different-rows*:
assumes *i-not-j*: *i* ≠ *j*
shows *det* (*interchange-rows A i j*) = − *det A* ⟨*proof*⟩

corollary *det-interchange-same-rows*:
assumes *i-eq-j*: *i* = *j*
shows *det* (*interchange-rows A i j*) = *det A* ⟨*proof*⟩

lemma *det-mult-row*:
shows *det* (*mult-row A a k*) = *k* ∗ *det A*
⟨*proof*⟩

lemma *det-row-add′*:
assumes *i-not-j*: *i* ≠ *j*
shows *det* (*row-add A i j q*) = *det A*
⟨*proof*⟩

9.1.2 Relationships between determinants and elementary column operations

lemma *det-interchange-columns*:
shows *det* (*interchange-columns A i j*) = *of-int* (*if i* = *j then 1 else* −*1*) ∗ *det A*
⟨*proof*⟩

corollary *det-interchange-different-columns*:
assumes *i-not-j*: *i* ≠ *j*
shows *det* (*interchange-columns A i j*) = − *det A* ⟨*proof*⟩

corollary *det-interchange-same-columns*:
assumes *i-eq-j*: *i* = *j*
shows *det* (*interchange-columns A i j*) = *det A* ⟨*proof*⟩

lemma *det-mult-columns*:
shows *det* (*mult-column A a k*) = *k* ∗ *det A*
⟨*proof*⟩

lemma *det-column-add*:
assumes *i-not-j*: $i \neq j$
shows *det* (*column-add A i j q*) = *det A*
⟨*proof*⟩

9.2 Proving that the determinant can be computed by means of the Gauss Jordan algorithm

9.2.1 Previous properties

lemma *det-row-add-iterate-upt-n*:
fixes *A*::*'a*::{*comm-ring-1*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
assumes *n*: *n*<*nrows A*
shows *det* (*row-add-iterate A n i j*) = *det A*
⟨*proof*⟩

corollary *det-row-add-iterate*:
fixes *A*::*'a*::{*comm-ring-1*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *det* (*row-add-iterate A* (*nrows A* − *1*) *i j*) = *det A*
⟨*proof*⟩

lemma *det-Gauss-Jordan-in-ij*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*} **and** *i j*::*'n*
defines *A'*: *A'*== *mult-row* (*interchange-rows A i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*)) *i* (*1* / (*interchange-rows A i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*)) $ *i* $ *j*)
shows *det* (*Gauss-Jordan-in-ij A i j*) = *det A'*
⟨*proof*⟩

lemma *det-Gauss-Jordan-in-ij-1*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*} **and** *i j*::*'n*
defines *A'*: *A'*== *mult-row* (*interchange-rows A i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*)) *i* (*1* / (*interchange-rows A i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*)) $ *i* $ *j*)
assumes *i*: (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) = *i*
shows *det* (*Gauss-Jordan-in-ij A i j*) = *1*/(*A*$*i*$*j*) ∗ *det A*
⟨*proof*⟩

lemma *det-Gauss-Jordan-in-ij-2*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*} **and** *i j*::*'n*
defines *A'*: *A'*== *mult-row* (*interchange-rows A i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*)) *i* (*1* / (*interchange-rows A i* (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*)) $ *i* $ *j*)
assumes *i*: (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) ≠ *i*
shows *det* (*Gauss-Jordan-in-ij A i j*) = − *1*/(*A* $ (*LEAST n. A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) $ *j*) ∗ *det A*
⟨*proof*⟩

9.2.2 Definitions

The following definitions allow the computation of the determinant of a matrix using the Gauss-Jordan algorithm. In the first component the determinant of each transformation is accumulated and the second component contains the matrix transformed into a reduced row echelon form matrix

definition *Gauss-Jordan-in-ij-det-P* :: *'a*::{*semiring-1*, *inverse*, *one*, *uminus*}^*'m*^*'n*::{*finite*, *ord*}=> *'n*=>*'m*=>(*'a* × (*'a*^*'m*^*'n*::{*finite*, *ord*}))
where *Gauss-Jordan-in-ij-det-P A i j* = (*let n* = (*LEAST n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*) *in* (*if i* = *n then 1*/(*A* $ *i* $ *j*) *else* − *1*/(*A* $ *n* $ *j*), *Gauss-Jordan-in-ij A i j*))

definition *Gauss-Jordan-column-k-det-P* **where** *Gauss-Jordan-column-k-det-P A' k* =
(*let det-P*= *fst A'*; *i* = *fst* (*snd A'*); *A* = *snd* (*snd A'*); *from-nat-i* = *from-nat i*; *from-nat-k* = *from-nat k*
in if (∀ *m*≥*from-nat-i*. *A* $ *m* $ *from-nat-k* = *0*) ∨ *i* = *nrows A then* (*det-P*, *i*, *A*)
else let gauss = *Gauss-Jordan-in-ij-det-P A* (*from-nat-i*) (*from-nat-k*) *in* (*fst gauss* ∗ *det-P*, *i* + *1*, *snd gauss*))

definition *Gauss-Jordan-upt-k-det-P*
where *Gauss-Jordan-upt-k-det-P A k* = (*let foldl* = *foldl Gauss-Jordan-column-k-det-P* (*1*, *0*, *A*) [*0*..<*Suc k*] *in* (*fst foldl*, *snd* (*snd foldl*)))
definition *Gauss-Jordan-det-P*
where *Gauss-Jordan-det-P A* = *Gauss-Jordan-upt-k-det-P A* (*ncols A* − *1*)

9.2.3 Proofs

This is an equivalent definition created to achieve a more efficient computation.

lemma *Gauss-Jordan-in-ij-det-P-code*[*code*]:
shows *Gauss-Jordan-in-ij-det-P A i j* =
(*let n* = (*LEAST n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*);
interchange-A = *interchange-rows A i n*;
A' = *mult-row interchange-A i* (*1* / *interchange-A* $ *i* $ *j*) *in* (*if i* = *n then 1*/(*A* $ *i* $ *j*) *else* − *1*/(*A* $ *n* $ *j*), *Gauss-Jordan-wrapper i j A' interchange-A*))
⟨*proof*⟩

lemma *det-Gauss-Jordan-in-ij-det-P*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*} **and** *i j*::*'n*
shows (*fst* (*Gauss-Jordan-in-ij-det-P A i j*)) ∗ *det A* = *det* (*snd* (*Gauss-Jordan-in-ij-det-P A i j*))
⟨*proof*⟩

lemma *det-Gauss-Jordan-column-k-det-P*:

fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
assumes *det*: *det-P* ∗ *det B* = *det A*
shows (*fst* (*Gauss-Jordan-column-k-det-P* (*det-P*,*i*,*A*) *k*)) ∗ *det B* = *det* (*snd* (*snd* (*Gauss-Jordan-column-k-det-P* (*det-P*,*i*,*A*) *k*)))
⟨*proof*⟩

lemma *det-Gauss-Jordan-upt-k-det-P*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows (*fst* (*Gauss-Jordan-upt-k-det-P A k*)) ∗ *det A* = *det* (*snd* (*Gauss-Jordan-upt-k-det-P A k*))
⟨*proof*⟩

lemma *det-Gauss-Jordan-det-P*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows (*fst* (*Gauss-Jordan-det-P A*)) ∗ *det A* = *det* (*snd* (*Gauss-Jordan-det-P A*))
⟨*proof*⟩

definition *upper-triangular-upt-k* **where** *upper-triangular-upt-k A k* = (∀ *i j*. *j*<*i* ∧ *to-nat j* < *k* ⟶ *A* $ *i* $ *j* = *0*)
definition *upper-triangular* **where** *upper-triangular A* = (∀ *i j*. *j*<*i* ⟶ *A* $ *i* $ *j* = *0*)

lemma *upper-triangular-upt-imp-upper-triangular*:
assumes *upper-triangular-upt-k A* (*nrows A*)
shows *upper-triangular A*
⟨*proof*⟩

lemma *rref-imp-upper-triagular-upt*:
fixes *A*::*'a*::{*one*, *zero*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
assumes *reduced-row-echelon-form A*
shows *upper-triangular-upt-k A k*
⟨*proof*⟩

lemma *rref-imp-upper-triagular*:
assumes *reduced-row-echelon-form A*
shows *upper-triangular A*
⟨*proof*⟩

lemma *det-Gauss-Jordan*[*code-unfold*]:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *det* (*Gauss-Jordan A*) = *prod* (λ*i*. (*Gauss-Jordan A*)$*i*$*i*) (*UNIV*:: *'n set*)
⟨*proof*⟩

lemma *snd-Gauss-Jordan-in-ij-det-P-is-snd-Gauss-Jordan-in-ij-PA*:

shows *snd* (*Gauss-Jordan-in-ij-det-P A i j*) = *snd* (*Gauss-Jordan-in-ij-PA* (*P*,*A*) *i j*)
⟨*proof*⟩

lemma *snd-Gauss-Jordan-column-k-det-P-is-snd-Gauss-Jordan-column-k-PA*:
shows *snd* (*Gauss-Jordan-column-k-det-P* (*n*,*i*,*A*) *k*) = *snd* (*Gauss-Jordan-column-k-PA* (*P*,*i*,*A*) *k*)
⟨*proof*⟩

lemma *det-fst-row-add-iterate-PA*:
fixes *A*::*'a*::{*comm-ring-1*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
assumes *n*: *n*<*nrows A*
shows *det* (*fst* (*row-add-iterate-PA* (*P*,*A*) *n i j*)) = *det P*
⟨*proof*⟩

lemma *det-fst-Gauss-Jordan-in-ij-PA-eq-fst-Gauss-Jordan-in-ij-det-P*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *fst* (*Gauss-Jordan-in-ij-det-P A i j*) * *det P* = *det* (*fst* (*Gauss-Jordan-in-ij-PA* (*P*,*A*) *i j*))
⟨*proof*⟩

lemma *det-fst-Gauss-Jordan-column-k-PA-eq-fst-Gauss-Jordan-column-k-det-P*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *fst* (*Gauss-Jordan-column-k-det-P* (*det P*,*i*,*A*) *k*) = *det* (*fst* (*Gauss-Jordan-column-k-PA* (*P*,*i*,*A*) *k*))
⟨*proof*⟩

lemma *fst-snd-Gauss-Jordan-column-k-det-P-eq-fst-snd-Gauss-Jordan-column-k-PA*:
shows *fst* (*snd* (*Gauss-Jordan-column-k-det-P* (*n*,*i*,*A*) *k*)) = *fst* (*snd* (*Gauss-Jordan-column-k-PA* (*P*,*i*,*A*) *k*))
⟨*proof*⟩

The way of proving the following lemma is very similar to the demonstration of *?k* < *ncols ?A* ⟹ *reduced-row-echelon-form-upt-k* (*Gauss-Jordan-upt-k ?A ?k*) (*Suc ?k*)

?k < *ncols ?A* ⟹ *foldl Gauss-Jordan-column-k* (*0*, *?A*) [*0*..<*Suc ?k*] = (*if* ∀ *m*. *is-zero-row-upt-k m* (*Suc ?k*) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *?A*) [*0*..<*Suc ?k*])) *then 0 else mod-type-class.to-nat* (*GREATEST n*. ¬ *is-zero-row-upt-k n* (*Suc ?k*) (*snd* (*foldl Gauss-Jordan-column-k* (*0*, *?A*) [*0*..<*Suc ?k*]))) + *1*, *snd* (*foldl Gauss-Jordan-column-k* (*0*, *?A*) [*0*..<*Suc ?k*])).

lemma *foldl-Gauss-Jordan-column-k-det-P*:

fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *det-fst-Gauss-Jordan-upt-k-PA-eq-fst-Gauss-Jordan-upt-k-det-P*: *fst* (*Gauss-Jordan-upt-k-det-P A k*) = *det* (*fst* (*Gauss-Jordan-upt-k-PA A k*))
and *snd-Gauss-Jordan-upt-k-det-P-is-snd-Gauss-Jordan-upt-k-PA*: *snd* (*Gauss-Jordan-upt-k-det-P A k*) = *snd* (*Gauss-Jordan-upt-k-PA A k*)
and *fst-snd-foldl-Gauss-det-P-PA*: *fst* (*snd* (*foldl Gauss-Jordan-column-k-det-P* (*1*, *0*, *A*) [*0*..<*Suc k*])) = *fst* (*snd* (*foldl Gauss-Jordan-column-k-PA* (*mat 1*, *0*, *A*) [*0*..<*Suc k*]))
⟨*proof*⟩

lemma *snd-Gauss-Jordan-det-P-is-Gauss-Jordan*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *snd* (*Gauss-Jordan-det-P A*) = (*Gauss-Jordan A*)
⟨*proof*⟩

lemma *det-snd-Gauss-Jordan-det-P*[*code-unfold*]:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *det* (*snd* (*Gauss-Jordan-det-P A*)) = *prod* (λ*i*. (*snd* (*Gauss-Jordan-det-P A*))$*i*$*i*) (*UNIV*:: *'n set*)
⟨*proof*⟩

lemma *det-fst-Gauss-Jordan-PA-eq-fst-Gauss-Jordan-det-P*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *fst* (*Gauss-Jordan-det-P A*) = *det* (*fst* (*Gauss-Jordan-PA A*))
⟨*proof*⟩

lemma *fst-Gauss-Jordan-det-P-not-0*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *fst* (*Gauss-Jordan-det-P A*) ≠ *0*
⟨*proof*⟩

lemma *det-code-equation*[*code-unfold*]:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *det A* = (*let A'* = *Gauss-Jordan-det-P A in prod* (λ*i*. (*snd* (*A'*))$*i*$*i*) (*UNIV*::*'n set*)/(*fst* (*A'*)))
⟨*proof*⟩

end

10 Inverse of a matrix using the Gauss Jordan algorithm

theory *Inverse*
imports
Gauss-Jordan-PA
begin

10.1 Several properties

Properties about Gauss Jordan algorithm, reduced row echelon form, rank, identity matrix and invertibility

lemma *rref-id-implies-invertible*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
assumes *Gauss-mat-1*: *Gauss-Jordan A* = *mat 1*
shows *invertible A*
⟨*proof*⟩

In the following case, nrows is equivalent to ncols due to we are working with a square matrix

lemma *full-rank-implies-invertible*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
assumes *rank-n*: *rank A* = *nrows A*
shows *invertible A*
⟨*proof*⟩

lemma *invertible-implies-full-rank*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
assumes *inv-A*: *invertible A*
shows *rank A* = *nrows A*
⟨*proof*⟩

definition *id-upt-k* :: *'a*::{*zero*, *one*}^*'n*::{*mod-type*}^*'n*::{*mod-type*} ⇒ *nat* => *bool*
where *id-upt-k A k* = (∀ *i j*. *to-nat i* < *k* ∧ *to-nat j* < *k* ⟶ ((*i* = *j* ⟶ *A* $ *i* $ *j* = *1*) ∧ (*i* ≠ *j* ⟶ *A* $ *i* $ *j* = *0*)))

lemma *id-upt-nrows-mat-1*:
assumes *id-upt-k A* (*nrows A*)
shows *A* = *mat 1*
⟨*proof*⟩

10.2 Computing the inverse of a matrix using the Gauss Jordan algorithm

This lemma is essential to demonstrate that the Gauss Jordan form of an invertible matrix is the identity. The proof is made by induction and it is explained in http://www.unirioja.es/cu/jodivaso/Isabelle/Gauss-Jordan-2013-2-Generalized/Demonstration_invertible.pdf

lemma *id-upt-k-Gauss-Jordan*:
fixes $A::'a::\{field\}\ \hat{}'n::\{mod\text{-}type\}\ \hat{}'n::\{mod\text{-}type\}$
assumes *inv-A*: *invertible A*
shows *id-upt-k* (*Gauss-Jordan A*) *k*
⟨*proof*⟩

lemma *invertible-implies-rref-id*:
fixes $A::'a::\{field\}\ \hat{}'n::\{mod\text{-}type\}\ \hat{}'n::\{mod\text{-}type\}$
assumes *inv-A*: *invertible A*
shows *Gauss-Jordan A* = *mat 1*
⟨*proof*⟩

lemma *matrix-inv-Gauss*:
fixes $A::'a::\{field\}\ \hat{}'n::\{mod\text{-}type\}\ \hat{}'n::\{mod\text{-}type\}$
assumes *inv-A*: *invertible A* **and** *Gauss-eq*: *Gauss-Jordan A* = *P* ∗∗ *A*
shows *matrix-inv A* = *P*
⟨*proof*⟩

lemma *matrix-inv-Gauss-Jordan-PA*:
fixes $A::'a::\{field\}\ \hat{}'n::\{mod\text{-}type\}\ \hat{}'n::\{mod\text{-}type\}$
assumes *inv-A*: *invertible A*
shows *matrix-inv A* = *fst* (*Gauss-Jordan-PA A*)
⟨*proof*⟩

lemma *invertible-eq-full-rank*[*code-unfold*]:
fixes $A::'a::\{field\}\ \hat{}'n::\{mod\text{-}type\}\ \hat{}'n::\{mod\text{-}type\}$
shows *invertible A* = (*rank A* = *nrows A*)
⟨*proof*⟩

definition *inverse-matrix A* = (*if invertible A then Some* (*matrix-inv A*) *else None*)

lemma *the-inverse-matrix*:
fixes $A::'a::\{field\}\ \hat{}'n::\{mod\text{-}type\}\ \hat{}'n::\{mod\text{-}type\}$
assumes *invertible A*
shows *the* (*inverse-matrix A*) = *P-Gauss-Jordan A*
⟨*proof*⟩

lemma *inverse-matrix*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *inverse-matrix A* = (*if invertible A then Some* (*P-Gauss-Jordan A*) *else None*)
⟨*proof*⟩

lemma *inverse-matrix-code*[*code-unfold*]:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *inverse-matrix A* = (*let GJ* = *Gauss-Jordan-PA A*;
rank-A = (*if A* = *0 then 0 else to-nat* (*GREATEST a. row a* (*snd GJ*) ≠ *0*) + *1*) *in*
if nrows A = *rank-A then Some* (*fst*(*GJ*)) *else None*)
⟨*proof*⟩

end

11 Bases of the four fundamental subspaces

theory *Bases-Of-Fundamental-Subspaces*
imports
Gauss-Jordan-PA
begin

11.1 Computation of the bases of the fundamental subspaces

definition *basis-null-space A* = {*row i* (*P-Gauss-Jordan* (*transpose A*)) | *i. to-nat i* ≥ *rank A*}
definition *basis-row-space A* = {*row i* (*Gauss-Jordan A*) |*i. row i* (*Gauss-Jordan A*) ≠ *0*}
definition *basis-col-space A* = {*row i* (*Gauss-Jordan* (*transpose A*)) |*i. row i* (*Gauss-Jordan* (*transpose A*)) ≠ *0*}
definition *basis-left-null-space A* = {*row i* (*P-Gauss-Jordan A*) | *i. to-nat i* ≥ *rank A*}

11.2 Relatioships amongst the bases

lemma *basis-null-space-eq-basis-left-null-space-transpose*:
basis-null-space A = *basis-left-null-space* (*transpose A*)
⟨*proof*⟩

lemma *basis-null-space-transpose-eq-basis-left-null-space*:
shows *basis-null-space* (*transpose A*) = *basis-left-null-space A*
⟨*proof*⟩

lemma *basis-col-space-eq-basis-row-space-transpose*:
basis-col-space A = *basis-row-space* (*transpose A*)
⟨*proof*⟩

11.3 Code equations

Code equations to make more efficient the computations.

lemma *basis-null-space-code*[*code*]: *basis-null-space A* = (*let GJ* = *Gauss-Jordan-PA* (*transpose A*);
rank-A = (*if A* = *0 then 0 else to-nat* (*GREATEST a. row a* (*snd GJ*) ≠ *0*) + *1*)
in {*row i* (*fst GJ*) | *i. to-nat i* ≥ *rank-A*})
⟨*proof*⟩

lemma *basis-row-space-code*[*code*]: *basis-row-space A* = (*let A'* = *Gauss-Jordan A in* {*row i A'* |*i. row i A'* ≠ *0*})
⟨*proof*⟩

lemma *basis-col-space-code*[*code*]: *basis-col-space A* = (*let A'* = *Gauss-Jordan* (*transpose A*) *in* {*row i A'* |*i. row i A'* ≠ *0*})
⟨*proof*⟩

lemma *basis-left-null-space-code*[*code*]: *basis-left-null-space A* = (*let GJ* = *Gauss-Jordan-PA A*;
rank-A = (*if A* = *0 then 0 else to-nat* (*GREATEST a. row a* (*snd GJ*) ≠ *0*) + *1*)
in {*row i* (*fst GJ*) | *i. to-nat i* ≥ *rank-A*})
⟨*proof*⟩

11.4 Demonstrations that they are bases

We prove that we have obtained a basis for each subspace

lemma *independent-basis-left-null-space*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
shows *vec.independent* (*basis-left-null-space A*)
⟨*proof*⟩

lemma *card-basis-left-null-space-eq-dim*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
shows *card* (*basis-left-null-space A*) = *vec.dim* (*left-null-space A*)
⟨*proof*⟩

lemma *basis-left-null-space-in-left-null-space*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
shows *basis-left-null-space A* ⊆ *left-null-space A*
⟨*proof*⟩

lemma *left-null-space-subset-span-basis*:

fixes *A*::′*a*::{*field*} ^′*cols*::{*mod-type*} ^′*rows*::{*mod-type*}
shows *left-null-space A* ⊆ *vec.span* (*basis-left-null-space A*)
⟨*proof*⟩

corollary *basis-left-null-space*:
fixes *A*::′*a*::{*field*} ^′*cols*::{*mod-type*} ^′*rows*::{*mod-type*}
shows *vec.independent* (*basis-left-null-space A*) ∧
left-null-space A = *vec.span* (*basis-left-null-space A*)
⟨*proof*⟩

corollary *basis-null-space*:
fixes *A*::′*a*::{*field*} ^′*cols*::{*mod-type*} ^′*rows*::{*mod-type*}
shows *vec.independent* (*basis-null-space A*) ∧
null-space A = *vec.span* (*basis-null-space A*)
⟨*proof*⟩

lemma *basis-row-space-subset-row-space*:
fixes *A*::′*a*::{*field*} ^′*cols*::{*mod-type*} ^′*rows*::{*mod-type*}
shows *basis-row-space A* ⊆ *row-space A*
⟨*proof*⟩

lemma *row-space-subset-span-basis-row-space*:
fixes *A*::′*a*::{*field*} ^′*cols*::{*mod-type*} ^′*rows*::{*mod-type*}
shows *row-space A* ⊆ *vec.span* (*basis-row-space A*)
⟨*proof*⟩

lemma *basis-row-space*:
fixes *A*::′*a*::{*field*} ^′*cols*::{*mod-type*} ^′*rows*::{*mod-type*}
shows *vec.independent* (*basis-row-space A*)
∧ *vec.span* (*basis-row-space A*) = *row-space A*
⟨*proof*⟩

corollary *basis-col-space*:
fixes *A*::′*a*::{*field*} ^′*cols*::{*mod-type*} ^′*rows*::{*mod-type*}
shows *vec.independent* (*basis-col-space A*)
∧ *vec.span* (*basis-col-space A*) = *col-space A*
⟨*proof*⟩

end

12 Solving systems of equations using the Gauss Jordan algorithm

theory *System-Of-Equations*
imports
Gauss-Jordan-PA
Bases-Of-Fundamental-Subspaces
begin

12.1 Definitions

Given a system of equations $A *v x = b$, the following function returns the pair $(P ** A, P *v b)$, where P is the matrix which states *Gauss-Jordan A* $= P ** A$. That matrix is computed by means of *Gauss-Jordan-PA*.

definition *solve-system* :: ('*a*::{*field*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}) ⇒ ('*a*^'*rows*::{*mod-type*})

⇒ (('*a*^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}) × ('*a*^'*rows*::{*mod-type*}))
where *solve-system A b* = (*let A'* = *Gauss-Jordan-PA A in* (*snd A'*, (*fst A'*) *∗v b*))

definition *is-solution* **where** *is-solution x A b* = (*A ∗v x* = *b*)

12.2 Relationship between *is-solution-def* and *solve-system-def*

lemma *is-solution-imp-solve-system*:
fixes *A*::'*a*::{*field*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
assumes *xAb*:*is-solution x A b*
shows *is-solution x* (*fst* (*solve-system A b*)) (*snd* (*solve-system A b*))
⟨*proof*⟩

lemma *solve-system-imp-is-solution*:
fixes *A*::'*a*::{*field*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
assumes *xAb*: *is-solution x* (*fst* (*solve-system A b*)) (*snd* (*solve-system A b*))
shows *is-solution x A b*
⟨*proof*⟩

lemma *is-solution-solve-system*:
fixes *A*::'*a*::{*field*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
shows *is-solution x A b* = *is-solution x* (*fst* (*solve-system A b*)) (*snd* (*solve-system A b*))
⟨*proof*⟩

12.3 Consistent and inconsistent systems of equations

definition *consistent* :: '*a*::{*field*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*} ⇒ '*a*::{*field*}^'*rows*::{*mod-type*} ⇒ *bool*
where *consistent A b* = (∃ *x*. *is-solution x A b*)

definition *inconsistent* **where** *inconsistent A b* = (¬ (*consistent A b*))

lemma *inconsistent*: *inconsistent A b* = (¬ (∃ *x*. *is-solution x A b*))
⟨*proof*⟩

The following function will be use to solve consistent systems which are already in the reduced row echelon form.

definition *solve-consistent-rref* :: *'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*} ⇒ *'a*::{*field*}^*'rows*::{*mod-type*} ⇒ *'a*::{*field*}^*'cols*::{*mod-type*}
where *solve-consistent-rref A b* = (χ *j*. *if* (∃ *i*. *A* $ *i* $ *j* = *1* ∧ *j*=(*LEAST n*. *A* $ *i* $ *n* ≠ *0*)) *then b* $ (*THE i*. *A* $ *i* $ *j* = *1*) *else 0*)

lemma *solve-consistent-rref-code*[*code abstract*]:
shows *vec-nth* (*solve-consistent-rref A b*) = (% *j*. *if* (∃ *i*. *A* $ *i* $ *j* = *1* ∧ *j*=(*LEAST n*. *A* $ *i* $ *n* ≠ *0*)) *then b* $ (*THE i*. *A* $ *i* $ *j* = *1*) *else 0*)
⟨*proof*⟩

lemma *rank-ge-imp-is-solution*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *con*: *rank A* ≥ (*if* (∃ *a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*) *then* (*to-nat* (*GREATEST a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*) + *1*) *else 0*)
shows *is-solution* (*solve-consistent-rref* (*Gauss-Jordan A*) (*P-Gauss-Jordan A* ∗*v b*)) *A b*
⟨*proof*⟩

corollary *rank-ge-imp-consistent*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *rank A* ≥ (*if* (∃ *a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*) *then* (*to-nat* (*GREATEST a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*) + *1*) *else 0*)
shows *consistent A b*
⟨*proof*⟩

lemma *inconsistent-imp-rank-less*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *inc*: *inconsistent A b*
shows *rank A* < (*if* (∃ *a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*) *then* (*to-nat* (*GREATEST a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*) + *1*) *else 0*)
⟨*proof*⟩

lemma *rank-less-imp-inconsistent*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *inc*: *rank A* < (*if* (∃ *a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*) *then* (*to-nat* (*GREATEST a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*) + *1*) *else 0*)
shows *inconsistent A b*

⟨*proof*⟩

corollary *consistent-imp-rank-ge*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *consistent A b*
shows *rank A* ≥ (*if* (∃ *a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*) *then* (*to-nat* (*GREATEST a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*) + *1*) *else 0*)
⟨*proof*⟩

lemma *inconsistent-eq-rank-less*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
shows *inconsistent A b* = (*rank A* < (*if* (∃ *a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*)
then (*to-nat* (*GREATEST a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*) + *1*) *else 0*))
⟨*proof*⟩

lemma *consistent-eq-rank-ge*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
shows *consistent A b* = (*rank A* ≥ (*if* (∃ *a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*)
then (*to-nat* (*GREATEST a*. (*P-Gauss-Jordan A* ∗*v b*) $ *a* ≠ *0*) + *1*) *else 0*))
⟨*proof*⟩

corollary *consistent-imp-is-solution*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *consistent A b*
shows *is-solution* (*solve-consistent-rref* (*Gauss-Jordan A*) (*P-Gauss-Jordan A* ∗*v b*)) *A b*
⟨*proof*⟩

corollary *consistent-imp-is-solution'*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *consistent A b*
shows *is-solution* (*solve-consistent-rref* (*fst* (*solve-system A b*)) (*snd* (*solve-system A b*))) *A b*
⟨*proof*⟩

Code equations optimized using Lets

lemma *inconsistent-eq-rank-less-code*[*code*]:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
shows *inconsistent A b* = (*let GJ-P*=*Gauss-Jordan-PA A*;
P-mult-b = (*fst*(*GJ-P*) ∗*v b*);
rank-A = (*if A* = *0 then 0 else to-nat* (*GREATEST a*. *row a* (*snd GJ-P*) ≠ *0*) + *1*) *in* (*rank-A* < (*if* (∃ *a*. *P-mult-b* $ *a* ≠ *0*)
then (*to-nat* (*GREATEST a*. *P-mult-b* $ *a* ≠ *0*) + *1*) *else 0*)))

⟨*proof*⟩

lemma *consistent-eq-rank-ge-code*[*code*]:
fixes *A*::*'a*::{*field*}^*cols*::{*mod-type*}^*rows*::{*mod-type*}
shows *consistent A b* = (*let GJ-P*=*Gauss-Jordan-PA A*;
P-mult-b = (*fst*(*GJ-P*) ∗*v b*);
rank-A = (*if A* = *0 then 0 else to-nat* (*GREATEST a. row a* (*snd GJ-P*) ≠ *0*) + *1*) *in* (*rank-A* ≥ (*if* (∃ *a. P-mult-b* $ *a* ≠ *0*)
then (*to-nat* (*GREATEST a. P-mult-b* $ *a* ≠ *0*) + *1*) *else 0*)))
⟨*proof*⟩

12.4 Solution set of a system of equations. Dependent and independent systems.

definition *solution-set* **where** *solution-set A b* = {*x. is-solution x A b*}

lemma *null-space-eq-solution-set*:
shows *null-space A* = *solution-set A 0* ⟨*proof*⟩

corollary *dim-solution-set-homogeneous-eq-dim-null-space*[*code-unfold*]:
shows *vec.dim* (*solution-set A 0*) = *vec.dim* (*null-space A*) ⟨*proof*⟩

lemma *zero-is-solution-homogeneous-system*:
shows *0* ∈ (*solution-set A 0*)
⟨*proof*⟩

lemma *homogeneous-solution-set-subspace*:
fixes *A*::*'a*::{*field*}^*n*^*rows*
shows *vec.subspace* (*solution-set A 0*)
⟨*proof*⟩

lemma *solution-set-rel*:
fixes *A*::*'a*::{*field*}^*n*^*rows*
assumes *p*: *is-solution p A b*
shows *solution-set A b* = {*p*} + (*solution-set A 0*)
⟨*proof*⟩

lemma *independent-and-consistent-imp-uniqueness-solution*:
fixes *A*::*'a*::{*field*}^*n*::{*mod-type*}^*rows*::{*mod-type*}
assumes *dim-0*: *vec.dim* (*solution-set A 0*) = *0*
and *con*: *consistent A b*
shows ∃!*x. is-solution x A b*
⟨*proof*⟩

lemma *card-1-exists*: *card s = 1* $\longleftrightarrow$ $(\exists!x.\ x \in s)$
⟨*proof*⟩

corollary *independent-and-consistent-imp-card-1*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *dim-0*: *vec.dim* (*solution-set A 0*) = *0*
and *con*: *consistent A b*
shows *card* (*solution-set A b*) = *1*
⟨*proof*⟩

lemma *uniqueness-solution-imp-independent*:
fixes *A*::*'a*::{*field*}^*'n*^*'rows*
assumes *ex1-sol*: $\exists!x.$ *is-solution x A b*
shows *vec.dim* (*solution-set A 0*) = *0*
⟨*proof*⟩

corollary *uniqueness-solution-eq-independent-and-consistent*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'rows*::{*mod-type*}
shows ($\exists!x.$ *is-solution x A b*) = (*consistent A b* $\wedge$ *vec.dim* (*solution-set A 0*) = *0*)
⟨*proof*⟩

lemma *consistent-homogeneous*:
shows *consistent A 0* ⟨*proof*⟩

lemma *dim-solution-set-0*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'rows*::{*mod-type*}
shows (*vec.dim* (*solution-set A 0*) = *0*) = (*solution-set A 0* = {*0*})
⟨*proof*⟩

We have to impose the restriction *semiring-char-0* in the following lemma, because it may not hold over a general field (for instance, in Z2 there is a finite number of elements, so the solution set can't be infinite.

lemma *dim-solution-set-not-zero-imp-infinite-solutions-homogeneous*:
fixes *A*::*'a*::{*field*, *semiring-char-0*}^*'n*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *dim-not-zero*: *vec.dim* (*solution-set A 0*) > *0*
shows *infinite* (*solution-set A 0*)
⟨*proof*⟩

lemma *infinite-solutions-homogeneous-imp-dim-solution-set-not-zero*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *i*: *infinite* (*solution-set A 0*)
shows *vec.dim* (*solution-set A 0*) > *0*
⟨*proof*⟩

corollary *infinite-solution-set-homogeneous-eq*:

fixes A::'a::{*field*,*semiring-char-0*}^'n::{*mod-type*}^'*rows*::{*mod-type*}
shows *infinite* (*solution-set A 0*) = (*vec.dim* (*solution-set A 0*) > *0*)
⟨*proof*⟩

corollary *infinite-solution-set-homogeneous-eq'*:
fixes A::'a::{*field*,*semiring-char-0*}^'n::{*mod-type*}^'*rows*::{*mod-type*}
shows (∃$_\infty x$. *is-solution x A 0*) = (*vec.dim* (*solution-set A 0*) > *0*)
⟨*proof*⟩

lemma *infinite-solution-set-imp-consistent*:
infinite (*solution-set A b*) ⟹ *consistent A b*
⟨*proof*⟩

lemma *dim-solution-set-not-zero-imp-infinite-solutions-no-homogeneous*:
fixes A::'a::{*field*, *semiring-char-0*}^'n::{*mod-type*}^'*rows*::{*mod-type*}
assumes *dim-not-0*: *vec.dim* (*solution-set A 0*) > *0*
and *con*: *consistent A b*
shows *infinite* (*solution-set A b*)
⟨*proof*⟩

lemma *infinite-solutions-no-homogeneous-imp-dim-solution-set-not-zero-imp*:
fixes A::'a::{*field*}^'n::{*mod-type*}^'*rows*::{*mod-type*}
assumes *i*: *infinite* (*solution-set A b*)
shows *vec.dim* (*solution-set A 0*) > *0*
⟨*proof*⟩

corollary *infinite-solution-set-no-homogeneous-eq*:
fixes A::'a::{*field*, *semiring-char-0*}^'n::{*mod-type*}^'*rows*::{*mod-type*}
shows *infinite* (*solution-set A b*) = (*consistent A b* ∧ *vec.dim* (*solution-set A 0*) > *0*)
⟨*proof*⟩

corollary *infinite-solution-set-no-homogeneous-eq'*:
fixes A::'a::{*field*, *semiring-char-0*}^'n::{*mod-type*}^'*rows*::{*mod-type*}
shows (∃$_\infty x$. *is-solution x A b*) = (*consistent A b* ∧ *vec.dim* (*solution-set A 0*) > *0*)
⟨*proof*⟩

definition *independent-and-consistent A b* = (*consistent A b* ∧ *vec.dim* (*solution-set A 0*) = *0*)
definition *dependent-and-consistent A b* = (*consistent A b* ∧ *vec.dim* (*solution-set A 0*) > *0*)

12.5 Solving systems of linear equations

The following function will solve any system of linear equations. Given a matrix A and a vector b, Firstly it makes use of the funcion *solve-system* to transform the original matrix A and the vector b into another ones in

reduced row echelon form. Then, that system will have the same solution than the original one but it is easier to be solved. So we make use of the function *solve-consistent-rref* to obtain one solution of the system.

We will prove that any solution of the system can be rewritten as a linear combination of elements of a basis of the null space plus a particular solution of the system. So the function *solve* will return an option type, depending on the consistency of the system:

- If the system is consistent (so there exists at least one solution), the function will return the *Some* of a pair. In the first component of that pair will be one solution of the system and the second one will be a basis of the null space of the matrix. Hence:
  1. If the system is consistent and independent (so there exists one and only one solution), the pair will consist of the solution and the empty set (this empty set is the basis of the null space).
  2. If the system is consistent and dependent (so there exists more than one solution, maybe an infinite number), the pair will consist of one particular solution and a basis of the null space (which will not be the empty set).
- If the system is inconsistent (so there exists no solution), the function will return *None*.

definition *solve A b* = (*if consistent A b then*
Some (*solve-consistent-rref* (*fst* (*solve-system A b*)) (*snd* (*solve-system A b*)), *basis-null-space A*)
else None)

lemma *solve-code*[*code*]:
shows *solve A b* = (*let GJ-P*=*Gauss-Jordan-PA A*;
P-times-b=*fst*(*GJ-P*) ∗*v b*;
rank-A = (*if A* = *0 then 0 else to-nat* (*GREATEST a. row a* (*snd GJ-P*) ≠ *0*) + *1*);
consistent-Ab = (*rank-A* ≥ (*if* (∃ *a*. (*P-times-b*) $ *a* ≠ *0*) *then* (*to-nat* (*GREATEST a*. (*P-times-b*) $ *a* ≠ *0*) + *1*) *else 0*));
GJ-transpose = *Gauss-Jordan-PA* (*transpose A*);
basis = {*row i* (*fst GJ-transpose*) | *i*. *to-nat i* ≥ *rank-A*}
in (*if consistent-Ab then Some* (*solve-consistent-rref* (*snd GJ-P*) *P-times-b*,*basis*) *else None*))
⟨*proof*⟩

lemma *consistent-imp-is-solution-solve*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *con*: *consistent A b*
shows *is-solution* (*fst* (*the* (*solve A b*))) *A b*
⟨*proof*⟩

corollary *consistent-eq-solution-solve*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
shows *consistent A b* = *is-solution* (*fst* (*the* (*solve A b*))) *A b*
⟨*proof*⟩

lemma *inconsistent-imp-solve-eq-none*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *con*: *inconsistent A b*
shows *solve A b* = *None* ⟨*proof*⟩

corollary *inconsistent-eq-solve-eq-none*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
shows *inconsistent A b* = (*solve A b* = *None*)
⟨*proof*⟩

We demonstrate that all solutions of a system of linear equations can be expressed as a linear combination of the basis of the null space plus a particular solution obtained. The basis and the particular solution are obtained by means of the function *solve A b*

lemma *solution-set-rel-solve*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *con*: *consistent A b*
shows *solution-set A b* = {*fst* (*the* (*solve A b*))} + *vec.span* (*snd* (*the* (*solve A b*)))
⟨*proof*⟩

lemma *is-solution-eq-in-span-solve*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *con*: *consistent A b*
shows (*is-solution x A b*) = (*x* ∈ {*fst* (*the* (*solve A b*))} + *vec.span* (*snd* (*the* (*solve A b*))))
⟨*proof*⟩

end

13 Code Generation for Z2

theory *Code-Z2*
imports *HOL−Library.Z2*
begin

Implementation for the field of integer numbers module 2. Experimentally we have checked that the implementation by means of booleans is the fastest one.

code-datatype *0*::*bit* (*1*::*bit*)
code-printing
type-constructor *bit* ⇀ (*SML*) *Bool.bool*

| **constant** *0*::*bit* ⇀ (*SML*) *false*
| **constant** *1*::*bit* ⇀ (*SML*) *true*

code-printing
type-constructor *bit* ⇀ (*Haskell*) *Bool*
| **constant** *0*::*bit* ⇀ (*Haskell*) *False*
| **constant** *1*::*bit* ⇀ (*Haskell*) *True*
| **class-instance** *bit* :: *HOL.equal* => (*Haskell*) –

end

14 Examples of computations over abstract matrices

theory *Examples-Gauss-Jordan-Abstract*
imports
Determinants2
Inverse
System-Of-Equations
Code-Z2
HOL−Library.Code-Target-Numeral
begin

14.1 Transforming a list of lists to an abstract matrix

Definitions to transform a matrix to a list of list and vice versa

definition *vec-to-list* :: *'a^'n*::{*finite*, *enum*} => *'a list*
where *vec-to-list A* = *map* ((\$) *A*) (*enum-class.enum*::*'n list*)

definition *matrix-to-list-of-list* :: *'a^'n*::{*finite*, *enum*}*^'m*::{*finite*, *enum*} => *'a list list*
where *matrix-to-list-of-list A* = *map* (*vec-to-list*) (*map* ((\$) *A*) (*enum-class.enum*::*'m list*))

This definition should be equivalent to *vector-def* (in suitable types)

definition *list-to-vec* :: *'a list* => *'a^'n*::{*finite*, *enum*, *mod-type*}
where *list-to-vec xs* = *vec-lambda* (% *i*. *xs* ! (*to-nat i*))

lemma [*code abstract*]: *vec-nth* (*list-to-vec xs*) = (%*i*. *xs* ! (*to-nat i*))

⟨*proof*⟩

definition *list-of-list-to-matrix* :: *'a list list => 'a^'n::{finite, enum, mod-type}^'m::{finite, enum, mod-type}*
where *list-of-list-to-matrix xs = vec-lambda (%i. list-to-vec (xs ! (to-nat i)))*

lemma [*code abstract*]: *vec-nth (list-of-list-to-matrix xs) = (%i. list-to-vec (xs ! (to-nat i)))*
⟨*proof*⟩

14.2 Examples

The following three lemmas are presented in both this file and in the *Examples-Gauss-Jordan-IArrays* one. They allow a more convenient printing of rational and real numbers after evaluation. They have already been added to the repository version of Isabelle, so after Isabelle2014 they should be removed from here.

lemma [*code-post*]:
int-of-integer (− 1) = − 1
⟨*proof*⟩

lemma [*code-abbrev*]:
(of-rat (− 1) :: real) = − 1
⟨*proof*⟩

lemma [*code-post*]:
(of-rat (− (1 / numeral k)) :: real) = − 1 / numeral k
(of-rat (− (numeral k / numeral l)) :: real) = − numeral k / numeral l
⟨*proof*⟩

14.2.1 Ranks and dimensions

Examples on computing ranks, dimensions of row space, null space and col space and the Gauss Jordan algorithm

value *matrix-to-list-of-list (Gauss-Jordan (list-of-list-to-matrix ([[1,0,0,0,0,0],[0,1,0,0,0,0],[0,0,0,0,0,0],[0,*
value *matrix-to-list-of-list (Gauss-Jordan (list-of-list-to-matrix ([[1,−2,1,−3,0],[3,−6,2,−7,0]])::rat^5^2))*
value *matrix-to-list-of-list (Gauss-Jordan (list-of-list-to-matrix ([[1,0,0,1,1],[1,0,1,1,1]])::bit^5^2))*
value *(reduced-row-echelon-form-upt-k (list-of-list-to-matrix ([[1,0,8],[0,1,9],[0,0,0]])::real^3^3))*
3
value *matrix-to-list-of-list (Gauss-Jordan (list-of-list-to-matrix [[Complex 1 1,Complex 1 (− 1), Complex 0 0],[Complex 2 (− 1),Complex 1 3, Complex 7 3]]::complex^3^2))*
value *DIM(real^5)*
value *vec.dimension (TYPE(bit)) (TYPE(5))*
value *vec.dimension (TYPE(real)) (TYPE(2))*

value *DIM(real^5^4)*

value *row-rank* (*list-of-list-to-matrix* [[*1*,*0*,*0*,*7*,*5*],[*1*,*0*,*4*,*8*,−*1*],[*1*,*0*,*0*,*9*,*8*],[*1*,*2*,*3*,*6*,*5*]]::*real*^*5*^*4*)
value *vec.dim* (*row-space* (*list-of-list-to-matrix* [[*1*,*0*,*0*,*7*,*5*],[*1*,*0*,*4*,*8*,−*1*],[*1*,*0*,*0*,*9*,*8*],[*1*,*2*,*3*,*6*,*5*]]::*real*^*5*^*4*))
value *col-rank* (*list-of-list-to-matrix* [[*1*,*0*,*0*,*7*,*5*],[*1*,*0*,*4*,*8*,−*1*],[*1*,*0*,*0*,*9*,*8*],[*1*,*2*,*3*,*6*,*5*]]::*real*^*5*^*4*)
value *vec.dim* (*col-space* (*list-of-list-to-matrix* [[*1*,*0*,*0*,*7*,*5*],[*1*,*0*,*4*,*8*,−*1*],[*1*,*0*,*0*,*9*,*8*],[*1*,*2*,*3*,*6*,*5*]]::*real*^*5*^*4*))
value *rank* (*list-of-list-to-matrix* [[*1*,*0*,*0*,*7*,*5*],[*1*,*0*,*4*,*8*,−*1*],[*1*,*0*,*0*,*9*,*8*],[*1*,*2*,*3*,*6*,*5*]]::*real*^*5*^*4*)
value *vec.dim* (*null-space* (*list-of-list-to-matrix* [[*1*,*0*,*0*,*7*,*5*],[*1*,*0*,*4*,*8*,−*1*],[*1*,*0*,*0*,*9*,*8*],[*1*,*2*,*3*,*6*,*5*]]::*real*^*5*^*4*))
value *rank* (*list-of-list-to-matrix* [[*Complex 1 1*,*Complex 1* (− *1*), *Complex 0 0*],[*Complex 2* (− *1*),*Complex 1 3*, *Complex 7 3*]]::*complex*^*3*^*2*)

14.2.2 Inverse of a matrix

Examples on computing the inverse of matrices

value *let A*=(*list-of-list-to-matrix* [[*1*,*1*,*2*,*4*,*5*,*9*,*8*],[*3*,*0*,*8*,*4*,*5*,*0*,*8*],[*3*,*2*,*0*,*4*,*5*,*9*,*8*],
[*3*,*2*,*8*,*0*,*5*,*9*,*8*] ,[*3*,*2*,*8*,*4*,*0*,*9*,*8*] ,[*3*,*2*,*8*,*4*,*5*,*0*,*8*], [*3*,*2*,*8*,*4*,*5*,*9*,*0*]]::*real*^*7*^*7*)
in matrix-to-list-of-list (*P-Gauss-Jordan A*)
value *let A*=(*list-of-list-to-matrix* [[*1*,*1*,*2*,*4*,*5*,*9*,*8*],[*3*,*0*,*8*,*4*,*5*,*0*,*8*],[*3*,*2*,*0*,*4*,*5*,*9*,*8*],
[*3*,*2*,*8*,*0*,*5*,*9*,*8*] ,[*3*,*2*,*8*,*4*,*0*,*9*,*8*] ,[*3*,*2*,*8*,*4*,*5*,*0*,*8*], [*3*,*2*,*8*,*4*,*5*,*9*,*0*]]::*real*^*7*^*7*) *in*
matrix-to-list-of-list (*A* ∗∗ (*P-Gauss-Jordan A*))
value *let A*=(*list-of-list-to-matrix* [[*1*,*1*,*2*,*4*,*5*,*9*,*8*],[*3*,*0*,*8*,*4*,*5*,*0*,*8*],[*3*,*2*,*0*,*4*,*5*,*9*,*8*],
[*3*,*2*,*8*,*0*,*5*,*9*,*8*] ,[*3*,*2*,*8*,*4*,*0*,*9*,*8*] ,[*3*,*2*,*8*,*4*,*5*,*0*,*8*], [*3*,*2*,*8*,*4*,*5*,*9*,*0*]]::*real*^*7*^*7*)
in (*inverse-matrix A*)
value *let A*=(*list-of-list-to-matrix* [[*1*,*1*,*2*,*4*,*5*,*9*,*8*],[*3*,*0*,*8*,*4*,*5*,*0*,*8*],[*3*,*2*,*0*,*4*,*5*,*9*,*8*],
[*3*,*2*,*8*,*0*,*5*,*9*,*8*] ,[*3*,*2*,*8*,*4*,*0*,*9*,*8*] ,[*3*,*2*,*8*,*4*,*5*,*0*,*8*], [*3*,*2*,*8*,*4*,*5*,*9*,*0*]]::*real*^*7*^*7*)
in matrix-to-list-of-list (*the* (*inverse-matrix A*))
value *let A*=(*list-of-list-to-matrix* [[*1*,*1*,*1*,*1*,*1*,*1*,*1*],[*2*,*2*,*2*,*2*,*2*,*2*,*2*],[*3*,*2*,*0*,*4*,*5*,*9*,*8*],
[*3*,*2*,*8*,*0*,*5*,*9*,*8*] ,[*3*,*2*,*8*,*4*,*0*,*9*,*8*] ,[*3*,*2*,*8*,*4*,*5*,*0*,*8*], [*3*,*2*,*8*,*4*,*5*,*9*,*0*]]::*real*^*7*^*7*)
in (*inverse-matrix A*)
value *let A*=(*list-of-list-to-matrix* [[*Complex 1 1*,*Complex 1* (− *1*), *Complex 0 0*],[*Complex 1 1*,*Complex 1* (− *1*), *Complex 8 0*],[*Complex 2* (− *1*),*Complex 1 3*, *Complex 7 3*]]::*complex*^*3*^*3*)
in matrix-to-list-of-list (*the* (*inverse-matrix A*))

14.2.3 Determinant of a matrix

Examples on computing determinants of matrices

value (*let A* = *list-of-list-to-matrix*[[*1*,*2*,*7*,*8*,*9*],[*3*,*4*,*12*,*10*,*7*],[−*5*,*4*,*8*,*7*,*4*],[*0*,*1*,*2*,*4*,*8*],[*9*,*8*,*7*,*13*,*11*]]::*real*^*5*^
in det A)
value *det* (*list-of-list-to-matrix* ([[*1*,*0*,*0*],[*0*,*1*,*0*],[*0*,*0*,*1*]])::*real*^*3*^*3*)
value *det* (*list-of-list-to-matrix* ([[*1*,*8*,*9*,*1*,*47*],[*7*,*2*,*2*,*5*,*9*],[*3*,*2*,*7*,*7*,*4*],[*9*,*8*,*7*,*5*,*1*],[*1*,*2*,*6*,*4*,*5*]])::*rat*^*5*^*5*)

14.2.4 Bases of the fundamental subspaces

Examples on computing basis for null space, row space, column space and left null space

value *let A* = (*list-of-list-to-matrix* ([[*1*,*3*,−*2*,*0*,*2*,*0*],[*2*,*6*,−*5*,−*2*,*4*,−*3*],[*0*,*0*,*5*,*10*,*0*,*15*],[*2*,*6*,*0*,*8*,*4*,*18*]])::*real*

in vec-to-list‘ (*basis-null-space A*)

value *let A = (list-of-list-to-matrix ([[3,4,0,7],[1,−5,2,−2],[−1,4,0,3],[1,−1,2,2]])::real^4^4)*

in vec-to-list‘ (basis-null-space A)

value *let A = (list-of-list-to-matrix ([[1,3,−2,0,2,0],[2,6,−5,−2,4,−3],[0,0,5,10,0,15],[2,6,0,8,4,18]])::real*

in vec-to-list‘ (basis-row-space A)
value *let A = (list-of-list-to-matrix ([[3,4,0,7],[1,−5,2,−2],[−1,4,0,3],[1,−1,2,2]])::real^4^4)*

in vec-to-list‘ (basis-row-space A)

value *let A = (list-of-list-to-matrix ([[1,3,−2,0,2,0],[2,6,−5,−2,4,−3],[0,0,5,10,0,15],[2,6,0,8,4,18]])::real*

in vec-to-list‘ (basis-col-space A)
value *let A = (list-of-list-to-matrix ([[3,4,0,7],[1,−5,2,−2],[−1,4,0,3],[1,−1,2,2]])::real^4^4)*

in vec-to-list‘ (basis-col-space A)

value *let A = (list-of-list-to-matrix ([[1,3,−2,0,2,0],[2,6,−5,−2,4,−3],[0,0,5,10,0,15],[2,6,0,8,4,18]])::real*

in vec-to-list‘ (basis-left-null-space A)
value *let A = (list-of-list-to-matrix ([[3,4,0,7],[1,−5,2,−2],[−1,4,0,3],[1,−1,2,2]])::real^4^4)*

in vec-to-list‘ (basis-left-null-space A)

14.2.5 Consistency and inconsistency

Examples on checking the consistency/inconsistency of a system of equations

value *independent-and-consistent (list-of-list-to-matrix ([[1,0,0],[0,1,0],[0,0,1],[0,0,0],[0,0,0]])::real^3^5)*
(list-to-vec([2,3,4,0,0])::real^5)
value *consistent (list-of-list-to-matrix ([[1,0,0],[0,1,0],[0,0,1],[0,0,0],[0,0,0]])::real^3^5)*
(list-to-vec([2,3,4,0,0])::real^5)
value *inconsistent (list-of-list-to-matrix ([[1,0,0],[0,1,0],[3,0,1],[0,7,0],[0,0,9]])::real^3^5)*
(list-to-vec([2,0,4,0,0])::real^5)
value *dependent-and-consistent (list-of-list-to-matrix ([[1,0,0],[0,1,0]])::real^3^2)*
(list-to-vec([3,4])::real^2)
value *independent-and-consistent (mat 1::real^3^3) (list-to-vec([3,4,5])::real^3)*

14.2.6 Solving systems of linear equations

Examples on solving linear systems.

definition *print-result-solve*
where *print-result-solve A = (if A = None then None else Some (vec-to-list (fst (the A)), vec-to-list‘ (snd (the A))))*

value *let A = (list-of-list-to-matrix [[4,5,8],[9,8,7],[4,6,1]]::real^3^3);*
b=(list-to-vec [4,5,8]::real^3)
in (print-result-solve (solve A b))

value *let A = (list-of-list-to-matrix [[0,0,0],[0,0,0],[0,0,1]]::real^3^3);*
b=(list-to-vec [4,5,0]::real^3)
in (print-result-solve (solve A b))

value *let A = (list-of-list-to-matrix [[3,2,5,2,7],[6,4,7,4,5],[3,2,−1,2,−11],[6,4,1,4,−13]]::real^5^4);*

b=(list-to-vec [0,0,0,0]::real^4)
in (print-result-solve (solve A b))

value *let A = (list-of-list-to-matrix [[1,2,1],[−2,−3,−1],[2,4,2]]::real^3^3);*
b=(list-to-vec [−2,1,−4]::real^3)
in (print-result-solve (solve A b))

value *let A = (list-of-list-to-matrix [[1,1,−4,10],[3,−2,−2,6]]::real^4^2);*
b=(list-to-vec [24,15]::real^2)
in (print-result-solve (solve A b))

end

15 IArrays Addenda

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

15.1 Some previous instances

instantiation *iarray* :: (*plus*) *plus*
begin
definition *plus-iarray* :: *'a iarray ⇒ 'a iarray ⇒ 'a iarray*
where *plus-iarray A B = IArray.of-fun (λn. A!!n + B !! n) (IArray.length A)*
instance ⟨*proof*⟩
end

instantiation *iarray* :: (*minus*) *minus*
begin
definition *minus-iarray* :: *'a iarray ⇒ 'a iarray ⇒ 'a iarray*
where *minus-iarray A B = IArray.of-fun (λn. A!!n − B !! n) (IArray.length A)*
instance ⟨*proof*⟩
end

15.2 Some previous definitions and properties for IArrays

15.3 Code generation

end

16 Matrices as nested IArrays

theory *Matrix-To-IArray*
imports
Rank-Nullity-Theorem.Mod-Type
Elementary-Operations
IArray-Addenda
begin

16.1 Isomorphism between matrices implemented by vecs and matrices implemented by iarrays

16.1.1 Isomorphism between vec and iarray

definition *vec-to-iarray* :: $'a\hat{}'n::\{mod\text{-}type\} \Rightarrow 'a$ *iarray*
where *vec-to-iarray A* = *IArray.of-fun* (λi. *A* $ (*from-nat i*)) (*CARD*($'n$))

definition *iarray-to-vec* :: $'a$ *iarray* $\Rightarrow 'a\hat{}'n::\{mod\text{-}type\}$
where *iarray-to-vec A* = (χ *i*. *A* !! (*to-nat i*))

lemma *vec-to-iarray-nth*:
fixes *A*::$'a\hat{}'n::\{finite, mod\text{-}type\}$
assumes *i*: *i*<*CARD*($'n$)
shows (*vec-to-iarray A*) !! *i* = *A* $ (*from-nat i*)
⟨*proof*⟩

lemma *vec-to-iarray-nth'*:
fixes *A*::$'a\hat{}'n::\{mod\text{-}type\}$
shows (*vec-to-iarray A*) !! (*to-nat i*) = *A* $ *i*
⟨*proof*⟩

lemma *iarray-to-vec-nth*:
shows (*iarray-to-vec A*) $ *i* = *A* !! (*to-nat i*)
⟨*proof*⟩

lemma *vec-to-iarray-morph*:
fixes *A*::$'a\hat{}'n::\{mod\text{-}type\}$
shows (*A* = *B*) = (*vec-to-iarray A* = *vec-to-iarray B*)
⟨*proof*⟩

lemma *inj-vec-to-iarray*:
shows *inj vec-to-iarray*
⟨*proof*⟩

lemma *iarray-to-vec-vec-to-iarray*:
fixes *A*::$'a\hat{}'n::\{mod\text{-}type\}$
shows *iarray-to-vec* (*vec-to-iarray A*)=*A*

⟨*proof*⟩

lemma *vec-to-iarray-iarray-to-vec*:
assumes *length-eq*: *IArray.length A* = *CARD*('*n*::{*mod-type*})
shows *vec-to-iarray* (*iarray-to-vec A*::'*a*^'*n*::{*mod-type*}) = *A*
⟨*proof*⟩

lemma *length-vec-to-iarray*:
fixes *xa*::'*a*^'*n*::{*mod-type*}
shows *IArray.length* (*vec-to-iarray xa*) = *CARD*('*n*)
⟨*proof*⟩

16.1.2 Isomorphism between matrix and nested iarrays

definition *matrix-to-iarray* :: '*a*^'*n*::{*mod-type*}^'*m*::{*mod-type*} => '*a iarray iarray*
where *matrix-to-iarray A* = *IArray* (*map* (*vec-to-iarray* ∘ (($) *A*) ∘ (*from-nat*::*nat*=>'*m*)) [*0*..<*CARD*('*m*)])

definition *iarray-to-matrix* :: '*a iarray iarray* ⇒ '*a*^'*n*::{*mod-type*}^'*m*::{*mod-type*}
where *iarray-to-matrix A* = (χ *i j*. *A* !! (*to-nat i*) !! (*to-nat j*))

lemma *matrix-to-iarray-morph*:
fixes *A*::'*a*^'*n*::{*mod-type*}^'*m*::{*mod-type*}
shows (*A* = *B*) = (*matrix-to-iarray A* = *matrix-to-iarray B*)
⟨*proof*⟩

lemma *matrix-to-iarray-eq-of-fun*:
fixes *A*::'*a*^'*columns*::{*mod-type*}^'*rows*::{*mod-type*}
assumes *vec-eq-f*: ∀ *i*. *vec-to-iarray* (*A* $ *i*) = *f* (*to-nat i*)
and *n-eq-length*: *n*=*IArray.length* (*matrix-to-iarray A*)
shows *matrix-to-iarray A* = *IArray.of-fun f n*
⟨*proof*⟩

lemma *map-vec-to-iarray-rw*[*simp*]:
fixes *A*::'*a*^'*columns*::{*mod-type*}^'*rows*::{*mod-type*}
shows *map* (λ*x*. *vec-to-iarray* (*A* $ *from-nat x*)) [*0*..<*CARD*('*rows*)] ! *to-nat i* = *vec-to-iarray* (*A* $ *i*)
⟨*proof*⟩

lemma *matrix-to-iarray-nth*:
matrix-to-iarray A !! *to-nat i* !! *to-nat j* = *A* $ *i* $ *j*
⟨*proof*⟩

lemma *vec-matrix*: *vec-to-iarray* (*A*$*i*) = (*matrix-to-iarray A*) !! (*to-nat i*)
⟨*proof*⟩

lemma *iarray-to-matrix-matrix-to-iarray*:

fixes *A*::*'a*^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}
shows *iarray-to-matrix* (*matrix-to-iarray A*)=*A* ⟨*proof*⟩

16.2 Definition of operations over matrices implemented by iarrays

definition *mult-iarray* :: *'a*::{*times*} *iarray* => *'a* => *'a iarray*
where *mult-iarray A q* = *IArray.of-fun* (λ*n*. *q* ∗ *A*!!*n*) (*IArray.length A*)

definition *row-iarray* :: *nat* => *'a iarray iarray* => *'a iarray*
where *row-iarray k A* = *A* !! *k*

definition *column-iarray* :: *nat* => *'a iarray iarray* => *'a iarray*
where *column-iarray k A* = *IArray.of-fun* (λ*m*. *A* !! *m* !! *k*) (*IArray.length A*)

definition *nrows-iarray* :: *'a iarray iarray* => *nat*
where *nrows-iarray A* = *IArray.length A*

definition *ncols-iarray* :: *'a iarray iarray* => *nat*
where *ncols-iarray A* = *IArray.length* (*A*!!*0*)

definition *rows-iarray A* = {*row-iarray i A* | *i*. *i* ∈ {..<*nrows-iarray A*}}
definition *columns-iarray A* = {*column-iarray i A* | *i*. *i* ∈ {..<*ncols-iarray A*}}

definition *tabulate2* :: *nat* => *nat* => (*nat* => *nat* => *'a*) => *'a iarray iarray*
where *tabulate2 m n f* = *IArray.of-fun* (λ*i*. *IArray.of-fun* (*f i*) *n*) *m*

definition *transpose-iarray* :: *'a iarray iarray* => *'a iarray iarray*
where *transpose-iarray A* = *tabulate2* (*ncols-iarray A*) (*nrows-iarray A*) (λ*a b*. *A*!!*b*!!*a*)

definition *matrix-matrix-mult-iarray* :: *'a*::{*times*, *comm-monoid-add*} *iarray iarray* => *'a iarray iarray* => *'a iarray iarray* (**infixl** ‹∗∗*i*› *70*)
where *A* ∗∗*i B* = *tabulate2* (*nrows-iarray A*) (*ncols-iarray B*) (λ*i j*. *sum* (λ*k*. ((*A*!!*i*)!!*k*) ∗ ((*B*!!*k*)!!*j*)) {*0*..<*ncols-iarray A*})

definition *matrix-vector-mult-iarray* :: *'a*::{*semiring-1*} *iarray iarray* => *'a iarray* => *'a iarray* (**infixl** ‹∗*iv*› *70*)
where *A* ∗*iv x* = *IArray.of-fun* (λ*i*. *sum* (λ*j*. ((*A*!!*i*)!!*j*) ∗ (*x*!!*j*)) {*0*..<*IArray.length x*}) (*nrows-iarray A*)

definition *vector-matrix-mult-iarray* :: *'a*::{*semiring-1*} *iarray* => *'a iarray iarray* => *'a iarray* (**infixl** ‹*v*∗*i*› *70*)
where *x v*∗*i A* = *IArray.of-fun* (λ*j*. *sum* (λ*i*. (*x*!!*i*) ∗ ((*A*!!*i*)!!*j*)) {*0*..<*IArray.length x*}) (*ncols-iarray A*)

definition *mat-iarray* :: *'a*::{*zero*} => *nat* => *'a iarray iarray*
where *mat-iarray k n* = *tabulate2 n n* (λ *i j*. *if i* = *j then k else 0*)

definition *is-zero-iarray* :: *'a*::{*zero*} *iarray* ⇒ *bool*
  where *is-zero-iarray A* = *IArray.all* (λ*i*. *A* !! *i* = *0*) (*IArray*[*0*..<*IArray.length A*])

16.2.1 Properties of previous definitions

lemma *is-zero-iarray-eq-iff*:
  fixes *A*::*'a*::{*zero*}^*'n*::{*mod-type*}
  shows (*A* = *0*) = (*is-zero-iarray* (*vec-to-iarray A*))
⟨*proof*⟩

lemma *mult-iarray-works*:
  assumes *a*<*IArray.length A* **shows** *mult-iarray A q* !! *a* = *q* ∗ *A*!!*a*
  ⟨*proof*⟩

lemma *length-eq-card-rows*:
  fixes *A*::*'a*^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}
  shows *IArray.length* (*matrix-to-iarray A*) = *CARD*(*'rows*)
  ⟨*proof*⟩

lemma *nrows-eq-card-rows*:
  fixes *A*::*'a*^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}
  shows *nrows-iarray* (*matrix-to-iarray A*) = *CARD*(*'rows*)
  ⟨*proof*⟩

lemma *length-eq-card-columns*:
  fixes *A*::*'a*^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}
  shows *IArray.length* (*matrix-to-iarray A* !! *0*) = *CARD* (*'columns*)
  ⟨*proof*⟩

lemma *ncols-eq-card-columns*:
  fixes *A*::*'a*^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}
  shows *ncols-iarray* (*matrix-to-iarray A*) = *CARD*(*'columns*)
  ⟨*proof*⟩

lemma *matrix-to-iarray-nrows*:
  fixes *A*::*'a*^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}
  shows *nrows A* = *nrows-iarray* (*matrix-to-iarray A*)
  ⟨*proof*⟩

lemma *matrix-to-iarray-ncols*:
  fixes *A*::*'a*^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}
  shows *ncols A* = *ncols-iarray* (*matrix-to-iarray A*)
  ⟨*proof*⟩

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

lemma *vec-to-iarray-row'*: *vec-to-iarray* (*row i A*) = (*matrix-to-iarray A*) !! (*to-nat i*)
⟨*proof*⟩

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

lemma *vec-to-iarray-column'*:
assumes *k*: *k*<*ncols A*
shows (*vec-to-iarray* (*column* (*from-nat k*) *A*)) = (*column-iarray k* (*matrix-to-iarray A*))
⟨*proof*⟩

lemma *column-iarray-nth*:
assumes *i*: *i*<*nrows-iarray A*
shows *column-iarray j A* !! *i* = *A* !! *i* !! *j*
⟨*proof*⟩

lemma *vec-to-iarray-rows*: *vec-to-iarray'* (*rows A*) = *rows-iarray* (*matrix-to-iarray A*)
⟨*proof*⟩

lemma *vec-to-iarray-columns*: *vec-to-iarray'* (*columns A*) = *columns-iarray* (*matrix-to-iarray A*)
⟨*proof*⟩

16.3 Definition of elementary operations

definition *interchange-rows-iarray* :: *'a iarray iarray => nat => nat => 'a iarray iarray*
where *interchange-rows-iarray A a b* = *IArray.of-fun* (λ*n*. *if n=a then A!!b else if n=b then A!!a else A!!n*) (*IArray.length A*)

definition *mult-row-iarray* :: *'a*::{*times*} *iarray iarray => nat => 'a => 'a iarray iarray*
where *mult-row-iarray A a q* = *IArray.of-fun* (λ*n*. *if n=a then mult-iarray* (*A!!a*) *q else A!!n*) (*IArray.length A*)

definition *row-add-iarray* :: *'a*::{*plus*, *times*} *iarray iarray => nat => nat => 'a => 'a iarray iarray*
where *row-add-iarray A a b q* = *IArray.of-fun* (λ*n*. *if n=a then A!!a* + *mult-iarray* (*A!!b*) *q else A!!n*) (*IArray.length A*)

definition *interchange-columns-iarray* :: *'a iarray iarray => nat => nat => 'a iarray iarray*
where *interchange-columns-iarray A a b* = *tabulate2* (*nrows-iarray A*) (*ncols-iarray A*) (λ*i j*. *if j = a then A* !! *i* !! *b else if j = b then A* !! *i* !! *a else A* !! *i* !! *j*)

definition *mult-column-iarray* :: *'a*::{*times*} *iarray iarray* => *nat* => *'a* => *'a iarray iarray*
where *mult-column-iarray A n q* = *tabulate2* (*nrows-iarray A*) (*ncols-iarray A*) (λ*i j*. *if j* = *n then A* !! *i* !! *j* ∗ *q else A* !! *i* !! *j*)

definition *column-add-iarray* :: *'a*::{*plus*, *times*} *iarray iarray* => *nat* => *nat* => *'a* => *'a iarray iarray*
where *column-add-iarray A n m q* = *tabulate2* (*nrows-iarray A*) (*ncols-iarray A*) (λ*i j*. *if j* = *n then A* !! *i* !! *n* + *A* !! *i* !! *m* ∗ *q else A* !! *i* !! *j*)

16.3.1 Code generator

lemma *vec-to-iarray-plus*[*code-unfold*]: *vec-to-iarray* (*a* + *b*) = (*vec-to-iarray a*) + (*vec-to-iarray b*)
⟨*proof*⟩

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

lemma *matrix-to-iarray-mat*[*code-unfold*]:
matrix-to-iarray (*mat k* ::*'a*::{*zero*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}) = *mat-iarray k CARD*(*'n*::{*mod-type*})
⟨*proof*⟩

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

lemma *matrix-to-iarray-matrix-matrix-mult*[*code-unfold*]:
fixes *A*::*'a*::{*semiring-1*}^*'m*::{*mod-type*}^*'n*::{*mod-type*} **and** *B*::*'a*^*'b*::{*mod-type*}^*'m*::{*mod-type*}
shows *matrix-to-iarray* (*A* ∗∗ *B*) = (*matrix-to-iarray A*) ∗∗*i* (*matrix-to-iarray B*)
⟨*proof*⟩

lemma *vec-to-iarray-matrix-matrix-mult*[*code-unfold*]:
fixes *A*::*'a*::{*semiring-1*}^*'m*::{*mod-type*}^*'n*::{*mod-type*} **and** *x*::*'a*^*'m*::{*mod-type*}
shows *vec-to-iarray* (*A* ∗*v x*) = (*matrix-to-iarray A*) ∗*iv* (*vec-to-iarray x*)
⟨*proof*⟩

lemma *vec-to-iarray-vector-matrix-mult*[*code-unfold*]:
fixes *A*::*'a*::{*semiring-1*}^*'m*::{*mod-type*}^*'n*::{*mod-type*} **and** *x*::*'a*^*'n*::{*mod-type*}
shows *vec-to-iarray* (*x v*∗ *A*) = (*vec-to-iarray x*) *v*∗*i* (*matrix-to-iarray A*)
⟨*proof*⟩

lemma *matrix-to-iarray-interchange-rows*[*code-unfold*]:

fixes *A*::*'a*::{*semiring-1*}^*columns*::{*mod-type*}^*rows*::{*mod-type*}
shows *matrix-to-iarray* (*interchange-rows A i j*) = *interchange-rows-iarray* (*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*)
⟨*proof*⟩

lemma *matrix-to-iarray-mult-row*[*code-unfold*]:
fixes *A*::*'a*::{*semiring-1*}^*columns*::{*mod-type*}^*rows*::{*mod-type*}
shows *matrix-to-iarray* (*mult-row A i q*) = *mult-row-iarray* (*matrix-to-iarray A*) (*to-nat i*) *q*
⟨*proof*⟩

lemma *matrix-to-iarray-row-add*[*code-unfold*]:
fixes *A*::*'a*::{*semiring-1*}^*columns*::{*mod-type*}^*rows*::{*mod-type*}
shows *matrix-to-iarray* (*row-add A i j q*) = *row-add-iarray* (*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*) *q*
⟨*proof*⟩

lemma *matrix-to-iarray-interchange-columns*[*code-unfold*]:
fixes *A*::*'a*::{*semiring-1*}^*columns*::{*mod-type*}^*rows*::{*mod-type*}
shows *matrix-to-iarray* (*interchange-columns A i j*) = *interchange-columns-iarray* (*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*)
⟨*proof*⟩

lemma *matrix-to-iarray-mult-columns*[*code-unfold*]:
fixes *A*::*'a*::{*semiring-1*}^*columns*::{*mod-type*}^*rows*::{*mod-type*}
shows *matrix-to-iarray* (*mult-column A i q*) = *mult-column-iarray* (*matrix-to-iarray A*) (*to-nat i*) *q*
⟨*proof*⟩

lemma *matrix-to-iarray-column-add*[*code-unfold*]:
fixes *A*::*'a*::{*semiring-1*}^*columns*::{*mod-type*}^*rows*::{*mod-type*}
shows *matrix-to-iarray* (*column-add A i j q*) = *column-add-iarray* (*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*) *q*
⟨*proof*⟩

end

17 Gauss Jordan algorithm over nested IArrays

theory *Gauss-Jordan-IArrays*
imports
Matrix-To-IArray
Gauss-Jordan
begin

17.1 Definitions and functions to compute the Gauss-Jordan algorithm over matrices represented as nested iarrays

definition *least-non-zero-position-of-vector-from-index A i = the* (*List.find* (λ*x*. *A* !! *x* ≠ *0*) [*i*..<*IArray.length A*])
definition *least-non-zero-position-of-vector A = least-non-zero-position-of-vector-from-index A 0*

definition *vector-all-zero-from-index* :: (*nat* × *'a*::{*zero*} *iarray*) => *bool*
where *vector-all-zero-from-index A' = (let i=fst A'; A=(snd A') in IArray.all* (λ*x*. *A*!!*x = 0*) (*IArray* [*i*..<(*IArray.length A*)]))

definition *Gauss-Jordan-in-ij-iarrays* :: *'a*::{*field*} *iarray iarray => nat => nat => 'a iarray iarray*
where *Gauss-Jordan-in-ij-iarrays A i j = (let n = least-non-zero-position-of-vector-from-index* (*column-iarray j A*) *i*;
interchange-A = interchange-rows-iarray A i n;
A' = mult-row-iarray interchange-A i (*1*/*interchange-A*!!*i*!!*j*)
in IArray.of-fun (λ*s*. *if s = i then A'* !! *s else row-add-iarray A' s i* (− *interchange-A* !! *s* !! *j*) !! *s*) (*nrows-iarray A*))

definition *Gauss-Jordan-column-k-iarrays* :: (*nat* × *'a*::{*field*} *iarray iarray*) => *nat* => (*nat* × *'a iarray iarray*)
where *Gauss-Jordan-column-k-iarrays A' k=*(*let A=*(*snd A'*); *i=*(*fst A'*) *in*
if ((*vector-all-zero-from-index* (*i*, (*column-iarray k A*)))) ∨ *i* = (*nrows-iarray A*) *then* (*i*,*A*) *else* (*Suc i*, (*Gauss-Jordan-in-ij-iarrays A i k*)))

definition *Gauss-Jordan-upt-k-iarrays* :: *'a*::{*field*} *iarray iarray => nat => 'a*::{*field*} *iarray iarray*
where *Gauss-Jordan-upt-k-iarrays A k = snd* (*foldl Gauss-Jordan-column-k-iarrays* (*0*,*A*) [*0*..<*Suc k*])

definition *Gauss-Jordan-iarrays* :: *'a*::{*field*} *iarray iarray => 'a*::{*field*} *iarray iarray*
where *Gauss-Jordan-iarrays A = Gauss-Jordan-upt-k-iarrays A* (*ncols-iarray A* − *1*)

17.2 Proving the equivalence between Gauss-Jordan algorithm over nested iarrays and over nested vecs (abstract matrices).

lemma *vector-all-zero-from-index-eq*:
fixes *A*::*'a*::{*zero*}^*n*::{*mod-type*}
shows (∀ *m*≥*i*. *A* $ *m = 0*) = (*vector-all-zero-from-index* (*to-nat i*, *vec-to-iarray A*))
⟨*proof*⟩

lemma *matrix-vector-all-zero-from-index*:
fixes *A*::*'a*::{*zero*}^*columns*::{*mod-type*}^*rows*::{*mod-type*}

shows $(\forall\, m{\geq}i.\ A\ \$\ m\ \$\ k = 0) = (\textit{vector-all-zero-from-index}\ (\textit{to-nat}\ i,\ \textit{vec-to-iarray}\ (\textit{column}\ k\ A)))$
⟨*proof*⟩

lemma *vec-to-iarray-least-non-zero-position-of-vector-from-index*:
fixes $A{::}'a{::}\{\textit{zero}\}\ \hat{}'n{::}\{\textit{mod-type}\}$
assumes *not-all-zero*: $\neg\ (\textit{vector-all-zero-from-index}\ (\textit{to-nat}\ i,\ \ \textit{vec-to-iarray}\ A))$
shows $\textit{least-non-zero-position-of-vector-from-index}\ (\textit{vec-to-iarray}\ A)\ (\textit{to-nat}\ i) = \textit{to-nat}\ (\textit{LEAST}\ n.\ A\ \$\ n \neq 0 \wedge i \leq n)$
⟨*proof*⟩

corollary *vec-to-iarray-least-non-zero-position-of-vector-from-index′*:
fixes $A{::}'a{::}\{\textit{zero}\}\ \hat{}'\textit{cols}{::}\{\textit{mod-type}\}\ \hat{}'\textit{rows}{::}\{\textit{mod-type}\}$
assumes *not-all-zero*: $\neg\ (\textit{vector-all-zero-from-index}\ (\textit{to-nat}\ i,\ \textit{vec-to-iarray}\ (\textit{column}\ j\ A)))$
shows $\textit{least-non-zero-position-of-vector-from-index}\ (\textit{vec-to-iarray}\ (\textit{column}\ j\ A))\ (\textit{to-nat}\ i) = \textit{to-nat}\ (\textit{LEAST}\ n.\ A\ \$\ n\ \$\ j \neq 0 \wedge i \leq n)$
⟨*proof*⟩

corollary *vec-to-iarray-least-non-zero-position-of-vector-from-index″*:
fixes $A{::}'a{::}\{\textit{zero}\}\ \hat{}'\textit{cols}{::}\{\textit{mod-type}\}\ \hat{}'\textit{rows}{::}\{\textit{mod-type}\}$
assumes *not-all-zero*: $\neg\ (\textit{vector-all-zero-from-index}\ (\textit{to-nat}\ j,\ \textit{vec-to-iarray}\ (\textit{row}\ i\ A)))$
shows $\textit{least-non-zero-position-of-vector-from-index}\ (\textit{vec-to-iarray}\ (\textit{row}\ i\ A))\ (\textit{to-nat}\ j) = \textit{to-nat}\ (\textit{LEAST}\ n.\ A\ \$\ i\ \$\ n \neq 0 \wedge j \leq n)$
⟨*proof*⟩

lemma *matrix-to-iarray-Gauss-Jordan-in-ij*[*code-unfold*]:
fixes $A{::}'a{::}\{\textit{field}\}\ \hat{}'\textit{columns}{::}\{\textit{mod-type}\}\ \hat{}'\textit{rows}{::}\{\textit{mod-type}\}$
assumes *not-all-zero*: $\neg\ (\textit{vector-all-zero-from-index}\ (\textit{to-nat}\ i,\ \textit{vec-to-iarray}\ (\textit{column}\ j\ A)))$
shows $\textit{matrix-to-iarray}\ (\textit{Gauss-Jordan-in-ij}\ A\ i\ j) = \textit{Gauss-Jordan-in-ij-iarrays}\ (\textit{matrix-to-iarray}\ A)\ (\textit{to-nat}\ i)\ (\textit{to-nat}\ j)$
⟨*proof*⟩

lemma *matrix-to-iarray-Gauss-Jordan-column-k-1*:
fixes $A{::}'a{::}\{\textit{field}\}\ \hat{}'\textit{columns}{::}\{\textit{mod-type}\}\ \hat{}'\textit{rows}{::}\{\textit{mod-type}\}$
assumes k: $k{<}\textit{ncols}\ A$
and i: $i{\leq}\textit{nrows}\ A$
shows $(\textit{fst}\ (\textit{Gauss-Jordan-column-k}\ (i,\ A)\ k)) = \textit{fst}\ (\textit{Gauss-Jordan-column-k-iarrays}\ (i,\ \textit{matrix-to-iarray}\ A)\ k)$
⟨*proof*⟩

lemma *matrix-to-iarray-Gauss-Jordan-column-k-2*:

fixes $A::'a::\{field\}\ ^'columns::\{mod\text{-}type\}\ ^'rows::\{mod\text{-}type\}$
assumes k: $k<ncols\ A$
and i: $i\leq nrows\ A$
shows $matrix\text{-}to\text{-}iarray\ (snd\ (Gauss\text{-}Jordan\text{-}column\text{-}k\ (i,\ A)\ k)) = snd\ (Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}iarrays\ (i,\ matrix\text{-}to\text{-}iarray\ A)\ k)$
⟨*proof*⟩

Due to the assumptions presented in $[\![?k < ncols\ ?A;\ ?i \leq nrows\ ?A]\!] \Longrightarrow matrix\text{-}to\text{-}iarray\ (snd\ (Gauss\text{-}Jordan\text{-}column\text{-}k\ (?i,\ ?A)\ ?k)) = snd\ (Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}iarrays\ (?i,\ matrix\text{-}to\text{-}iarray\ ?A)\ ?k)$, the following lemma must have three shows. The proof style is similar to $?k < ncols\ ?A \Longrightarrow reduced\text{-}row\text{-}echelon\text{-}form\text{-}upt\text{-}k\ (Gauss\text{-}Jordan\text{-}upt\text{-}k\ ?A\ ?k)\ (Suc\ ?k)$
$?k < ncols\ ?A \Longrightarrow foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k\ (0,\ ?A)\ [0..<Suc\ ?k] = (if\ \forall m.\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ (Suc\ ?k)\ (snd\ (foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k\ (0,\ ?A)\ [0..<Suc\ ?k]))\ then\ 0\ else\ mod\text{-}type\text{-}class.to\text{-}nat\ (GREATEST\ n.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ n\ (Suc\ ?k)\ (snd\ (foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k\ (0,\ ?A)\ [0..<Suc\ ?k])))\ +\ 1,\ snd\ (foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k\ (0,\ ?A)\ [0..<Suc\ ?k]))$.

lemma *foldl-Gauss-Jordan-column-k-eq*:
fixes $A::'a::\{field\}\ ^'columns::\{mod\text{-}type\}\ ^'rows::\{mod\text{-}type\}$
assumes k: $k<ncols\ A$
shows *matrix-to-iarray-Gauss-Jordan-upt-k*[*code-unfold*]: $matrix\text{-}to\text{-}iarray\ (Gauss\text{-}Jordan\text{-}upt\text{-}k\ A\ k) = Gauss\text{-}Jordan\text{-}upt\text{-}k\text{-}iarrays\ (matrix\text{-}to\text{-}iarray\ A)\ k$
and *fst-foldl-Gauss-Jordan-column-k-eq*: $fst\ (foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k\text{-}iarrays\ (0,\ matrix\text{-}to\text{-}iarray\ A)\ [0..<Suc\ k]) = fst\ (foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k\ (0,\ A)\ [0..<Suc\ k])$
and *fst-foldl-Gauss-Jordan-column-k-less*: $fst\ (foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k\ (0,\ A)\ [0..<Suc\ k]) \leq nrows\ A$
⟨*proof*⟩

lemma *matrix-to-iarray-Gauss-Jordan*[*code-unfold*]:
fixes $A::'a::\{field\}\ ^'columns::\{mod\text{-}type\}\ ^'rows::\{mod\text{-}type\}$
shows $matrix\text{-}to\text{-}iarray\ (Gauss\text{-}Jordan\ A) = Gauss\text{-}Jordan\text{-}iarrays\ (matrix\text{-}to\text{-}iarray\ A)$
⟨*proof*⟩

17.3 Implementation over IArrays of the computation of the *rank* of a matrix

definition $rank\text{-}iarray :: 'a::\{field\}\ iarray\ iarray => nat$
where $rank\text{-}iarray\ A = (let\ A' = (Gauss\text{-}Jordan\text{-}iarrays\ A);\ nrows = (IArray.length\ A')\ in\ card\ \{i.\ i<nrows \wedge \neg\ is\text{-}zero\text{-}iarray\ (A' !!\ i)\})$

17.3.1 Proving the equivalence between *rank* and *rank-iarray*.

First of all, some code equations are removed to allow the execution of Gauss-Jordan algorithm using iarrays

lemmas *card'-code*(*2*)[*code del*]
lemmas *rank-Gauss-Jordan-code*[*code del*]

lemma *rank-eq-card-iarrays*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}
shows *rank A* = *card* {*vec-to-iarray* (*row i* (*Gauss-Jordan A*)) |*i*. ¬ *is-zero-iarray* (*vec-to-iarray* (*row i* (*Gauss-Jordan A*)))}
⟨*proof*⟩

lemma *rank-eq-card-iarrays'*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}
shows *rank A* = (*let A'* = (*Gauss-Jordan-iarrays* (*matrix-to-iarray A*)) *in card* {*row-iarray* (*to-nat i*) *A'* |*i*::*'rows*. ¬ *is-zero-iarray* (*A'* !! (*to-nat i*))})
⟨*proof*⟩

lemma *rank-eq-card-iarrays-code*:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}
shows *rank A* = (*let A'* = (*Gauss-Jordan-iarrays* (*matrix-to-iarray A*)) *in card* {*i*::*'rows*. ¬ *is-zero-iarray* (*A'* !! (*to-nat i*))})
⟨*proof*⟩

17.3.2 Code equations for computing the rank over nested iarrays and the dimensions of the elementary subspaces

lemma *rank-iarrays-code*[*code*]:
rank-iarray A = *length* (*filter* (λ*x*. ¬ *is-zero-iarray x*) (*IArray.list-of* (*Gauss-Jordan-iarrays A*)))
⟨*proof*⟩

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

lemma *dim-null-space-iarray*[*code-unfold*]:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}
shows *vec.dim* (*null-space A*) = *ncols-iarray* (*matrix-to-iarray A*) − *rank-iarray* (*matrix-to-iarray A*)
⟨*proof*⟩

lemma *dim-col-space-iarray*[*code-unfold*]:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}
shows *vec.dim* (*col-space A*) = *rank-iarray* (*matrix-to-iarray A*)
⟨*proof*⟩

lemma *dim-row-space-iarray*[*code-unfold*]:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}
shows *vec.dim* (*row-space A*) = *rank-iarray* (*matrix-to-iarray A*)
⟨*proof*⟩

lemma *dim-left-null-space-space-iarray*[*code-unfold*]:
fixes *A*::*'a*::{*field*}^*'columns*::{*mod-type*}^*'rows*::{*mod-type*}
shows *vec.dim* (*left-null-space A*) = *nrows-iarray* (*matrix-to-iarray A*) − *rank-iarray* (*matrix-to-iarray A*)
⟨*proof*⟩

end

18 Obtaining explicitly the invertible matrix which transforms a matrix to its reduced row echelon form over nested iarrays

theory *Gauss-Jordan-PA-IArrays*
imports
Gauss-Jordan-PA
Gauss-Jordan-IArrays
begin

18.1 Definitions

definition *Gauss-Jordan-in-ij-iarrays-PA A' i j* =
(*let P* = *fst A'*; *A* = *snd A'*; *n* = *least-non-zero-position-of-vector-from-index* (*column-iarray j A*) *i*; *interchange-A* = *interchange-rows-iarray A i n*;
interchange-P = *interchange-rows-iarray P i n*; *P'* = *mult-row-iarray interchange-P i* (*1* / *interchange-A* !! *i* !! *j*)
in (*IArray.of-fun* (λ*s*. *if s* = *i then P'* !! *s else row-add-iarray P' s i* (− *interchange-A* !! *s* !! *j*) !! *s*) (*nrows-iarray A*), *Gauss-Jordan-in-ij-iarrays A i j*))

definition *Gauss-Jordan-column-k-iarrays-PA* :: (*'a*::{*field*} *iarray iarray* × *nat* × *'a*::{*field*} *iarray iarray*) => *nat* => (*'a iarray iarray* × *nat* × *'a iarray iarray*)
where *Gauss-Jordan-column-k-iarrays-PA A' k*=(*let P* = *fst A'*; *i* = *fst* (*snd A'*); *A* = *snd* (*snd A'*) *in*
if (*vector-all-zero-from-index* (*i*, (*column-iarray k A*))) ∨ *i* = (*nrows-iarray A*) *then* (*P*,*i*,*A*) *else let Gauss* = *Gauss-Jordan-in-ij-iarrays-PA* (*P*, *A*) *i k*
in (*fst Gauss*, *i* + *1*, *snd Gauss*))

definition *Gauss-Jordan-upt-k-iarrays-PA* :: *'a*::{*field*} *iarray iarray* => *nat* => (*'a*::{*field*} *iarray iarray* × *'a iarray iarray*)
where *Gauss-Jordan-upt-k-iarrays-PA A k* = (*let foldl* = *foldl Gauss-Jordan-column-k-iarrays-PA* (*mat-iarray 1* (*nrows-iarray A*), *0*, *A*) [*0*..<*Suc k*] *in* (*fst foldl*, *snd* (*snd foldl*)))

definition *Gauss-Jordan-iarrays-PA* :: *'a*::{*field*} *iarray iarray* => (*'a iarray iarray* × *'a iarray iarray*)
where *Gauss-Jordan-iarrays-PA A* = *Gauss-Jordan-upt-k-iarrays-PA A* (*ncols-iarray A* − *1*)

18.2 Proofs

18.2.1 Properties of *Gauss-Jordan-in-ij-iarrays-PA*

lemma *Gauss-Jordan-in-ij-iarrays-PA-def'*[*code*]:
Gauss-Jordan-in-ij-iarrays-PA A' i j =
(*let P* = *fst A'*; *A* = *snd A'*; *n* = *least-non-zero-position-of-vector-from-index* (*column-iarray j A*) *i*;
interchange-A = *interchange-rows-iarray A i n*; *A'* = *mult-row-iarray interchange-A i* (*1 / interchange-A* !! *i* !! *j*);
interchange-P = *interchange-rows-iarray P i n*; *P'* = *mult-row-iarray interchange-P i* (*1 / interchange-A* !! *i* !! *j*)
in (*IArray.of-fun* (λ*s*. *if s* = *i then P'* !! *s else row-add-iarray P' s i* (− *interchange-A* !! *s* !! *j*) !! *s*) (*nrows-iarray A*),
(*IArray.of-fun* (λ*s*. *if s* = *i then A'* !! *s else row-add-iarray A' s i* (− *interchange-A* !! *s* !! *j*) !! *s*) (*nrows-iarray A*))))
⟨*proof*⟩

lemma *snd-Gauss-Jordan-in-ij-iarrays-PA*:
shows *snd* (*Gauss-Jordan-in-ij-iarrays-PA* (*P*, *A*) *i j*) = *Gauss-Jordan-in-ij-iarrays A i j*
⟨*proof*⟩

lemma *matrix-to-iarray-snd-Gauss-Jordan-in-ij-iarrays-PA*:
assumes ¬ *vector-all-zero-from-index* (*to-nat i*, *vec-to-iarray* (*column j A*))
shows *matrix-to-iarray* (*snd* (*Gauss-Jordan-in-ij-PA* (*P*,*A*) *i j*)) = *snd* (*Gauss-Jordan-in-ij-iarrays-PA* (*matrix-to-iarray P*,*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*))
⟨*proof*⟩

lemma *matrix-to-iarray-fst-Gauss-Jordan-in-ij-iarrays-PA*:
assumes *not-all-zero*: ¬ *vector-all-zero-from-index* (*to-nat i*, *vec-to-iarray* (*column j A*))
shows *matrix-to-iarray* (*fst* (*Gauss-Jordan-in-ij-PA* (*P*,*A*) *i j*)) = *fst* (*Gauss-Jordan-in-ij-iarrays-PA* (*matrix-to-iarray P*,*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*))
⟨*proof*⟩

18.2.2 Properties about *Gauss-Jordan-column-k-iarrays-PA*

lemma *matrix-to-iarray-fst-Gauss-Jordan-column-k-PA*:
assumes *i*: *i*≤*nrows A* **and** *k*: *k*<*ncols A*
shows *matrix-to-iarray* (*fst* (*Gauss-Jordan-column-k-PA* (*P*,*i*,*A*) *k*)) = *fst* (*Gauss-Jordan-column-k-iarrays-PA* (*matrix-to-iarray P*, *i*, *matrix-to-iarray A*) *k*)
⟨*proof*⟩

lemma *matrix-to-iarray-snd-Gauss-Jordan-column-k-PA*:
assumes *i*: $i \leq nrows\ A$ **and** *k*: $k < ncols\ A$
shows (*fst* (*snd* (*Gauss-Jordan-column-k-PA* (*P*,*i*,*A*) *k*))) = *fst* (*snd* (*Gauss-Jordan-column-k-iarrays-PA* (*matrix-to-iarray P*, *i*, *matrix-to-iarray A*) *k*))
⟨*proof*⟩

lemma *matrix-to-iarray-third-Gauss-Jordan-column-k-PA*:
assumes *i*: $i \leq nrows\ A$ **and** *k*: $k < ncols\ A$
shows *matrix-to-iarray* (*snd* (*snd* (*Gauss-Jordan-column-k-PA* (*P*,*i*,*A*) *k*))) = *snd* (*snd* (*Gauss-Jordan-column-k-iarrays-PA* (*matrix-to-iarray P*, *i*, *matrix-to-iarray A*) *k*))
⟨*proof*⟩

18.2.3 Properties about *Gauss-Jordan-upt-k-iarrays-PA*

lemma
assumes $k < ncols\ A$
shows *matrix-to-iarray-fst-Gauss-Jordan-upt-k-PA*: *matrix-to-iarray* (*fst* (*Gauss-Jordan-upt-k-PA A k*)) = *fst* (*Gauss-Jordan-upt-k-iarrays-PA* (*matrix-to-iarray A*) *k*)
and *matrix-to-iarray-snd-Gauss-Jordan-upt-k-PA*: *matrix-to-iarray* (*snd* (*Gauss-Jordan-upt-k-PA A k*)) = (*snd* (*Gauss-Jordan-upt-k-iarrays-PA* (*matrix-to-iarray A*) *k*))
and *fst* (*snd* (*foldl Gauss-Jordan-column-k-PA* (*mat 1*, *0*, *A*) [*0*..<*Suc k*])) =
fst (*snd* (*foldl Gauss-Jordan-column-k-iarrays-PA* (*mat-iarray 1* (*nrows-iarray* (*matrix-to-iarray A*)), *0*, *matrix-to-iarray A*) [*0*..<*Suc k*]))
and *fst* (*snd* (*foldl Gauss-Jordan-column-k-PA* (*mat 1*, *0*, *A*) [*0*..<*Suc k*])) ≤ *nrows A*
⟨*proof*⟩

18.2.4 Properties about *Gauss-Jordan-iarrays-PA*

lemma *matrix-to-iarray-fst-Gauss-Jordan-PA*:
shows *matrix-to-iarray* (*fst* (*Gauss-Jordan-PA A*)) = *fst* (*Gauss-Jordan-iarrays-PA* (*matrix-to-iarray A*))
⟨*proof*⟩

lemma *matrix-to-iarray-snd-Gauss-Jordan-PA*:
shows *matrix-to-iarray* (*snd* (*Gauss-Jordan-PA A*)) = *snd* (*Gauss-Jordan-iarrays-PA* (*matrix-to-iarray A*))
⟨*proof*⟩

lemma *Gauss-Jordan-iarrays-PA-mult*:
fixes *A*::'*a*::{*field*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
shows *snd* (*Gauss-Jordan-iarrays-PA* (*matrix-to-iarray A*)) = *fst* (*Gauss-Jordan-iarrays-PA* (*matrix-to-iarray A*)) ∗∗*i* (*matrix-to-iarray A*)
⟨*proof*⟩

lemma *snd-snd-Gauss-Jordan-column-k-iarrays-PA-eq*:
shows *snd* (*snd* (*Gauss-Jordan-column-k-iarrays-PA* (*P*,*i*,*A*) *k*)) = *snd* (*Gauss-Jordan-column-k-iarrays* (*i*,*A*) *k*)
⟨*proof*⟩

lemma *fst-snd-Gauss-Jordan-column-k-iarrays-PA-eq*:
shows *fst* (*snd* (*Gauss-Jordan-column-k-iarrays-PA* (*P*,*i*,*A*) *k*)) = *fst* (*Gauss-Jordan-column-k-iarrays* (*i*,*A*) *k*)
⟨*proof*⟩

lemma *foldl-Gauss-Jordan-column-k-iarrays-eq*:
snd (*foldl Gauss-Jordan-column-k-iarrays-PA* (*B*, *0* , *A*) [*0*..<*k*]) = *foldl Gauss-Jordan-column-k-iarrays* (*0* , *A*) [*0*..<*k*]
⟨*proof*⟩

lemma *snd-Gauss-Jordan-upt-k-iarrays-PA*:
shows *snd* (*Gauss-Jordan-upt-k-iarrays-PA A k*) = (*Gauss-Jordan-upt-k-iarrays A k*)
⟨*proof*⟩

lemma *snd-Gauss-Jordan-iarrays-PA-eq*: *snd* (*Gauss-Jordan-iarrays-PA A*) = *Gauss-Jordan-iarrays A*
⟨*proof*⟩

end

19 Bases of the four fundamental subspaces over IArrays

theory *Bases-Of-Fundamental-Subspaces-IArrays*
imports
Bases-Of-Fundamental-Subspaces
Gauss-Jordan-PA-IArrays
begin

19.1 Computation of bases of the fundamental subspaces using IArrays

We have made the definitions as efficient as possible.

definition *basis-left-null-space-iarrays A*
= (*let GJ* = *Gauss-Jordan-iarrays-PA A*;
rank-A = *length* [*x*←*IArray.list-of* (*snd GJ*) . ¬ *is-zero-iarray x*]
in set (*map* (λ*i*. *row-iarray i* (*fst GJ*)) [(*rank-A*)..<(*nrows-iarray A*)]))
definition *basis-null-space-iarrays A*

= (*let GJ*= *Gauss-Jordan-iarrays-PA* (*transpose-iarray A*);
rank-A = *length* [*x*←*IArray.list-of* (*snd GJ*) . ¬ *is-zero-iarray x*]
in set (*map* (λ*i*. *row-iarray i* (*fst GJ*)) [(*rank-A*)..<(*ncols-iarray A*)]))
definition *basis-row-space-iarrays A* =
(*let GJ*= *Gauss-Jordan-iarrays A*;
rank-A = *length* [*x*←*IArray.list-of* (*GJ*) . ¬ *is-zero-iarray x*]
in set (*map* (λ*i*. *row-iarray i* (*GJ*)) [*0*..<*rank-A*]))
definition *basis-col-space-iarrays A* = *basis-row-space-iarrays* (*transpose-iarray A*)

The following lemmas make easier the proofs of equivalence between abstract versions and concrete versions. They are false if we remove *matrix-to-iarray*

lemma *basis-null-space-iarrays-eq*:
fixes *A*::'*a*::{*field*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
shows *basis-null-space-iarrays* (*matrix-to-iarray A*)
= *set* (*map* (λ*i*. *row-iarray i* (*fst* (*Gauss-Jordan-iarrays-PA* (*transpose-iarray* (*matrix-to-iarray A*))))) [(*rank-iarray* (*matrix-to-iarray A*))..<(*ncols-iarray* (*matrix-to-iarray A*))])
⟨*proof*⟩

lemma *basis-row-space-iarrays-eq*:
fixes *A*::'*a*::{*field*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
shows *basis-row-space-iarrays* (*matrix-to-iarray A*) = *set* (*map* (λ*i*. *row-iarray i* (*Gauss-Jordan-iarrays* (*matrix-to-iarray A*))) [*0*..<(*rank-iarray* (*matrix-to-iarray A*))])
⟨*proof*⟩

lemma *basis-left-null-space-iarrays-eq*:
fixes *A*::'*a*::{*field*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
shows *basis-left-null-space-iarrays* (*matrix-to-iarray A*) = *basis-null-space-iarrays* (*transpose-iarray* (*matrix-to-iarray A*))
⟨*proof*⟩

19.2 Code equations

lemma *vec-to-iarray-basis-null-space*[*code-unfold*]:
fixes *A*::'*a*::{*field*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
shows *vec-to-iarray*‘ (*basis-null-space A*) = *basis-null-space-iarrays* (*matrix-to-iarray A*)
⟨*proof*⟩

corollary *vec-to-iarray-basis-left-null-space*[*code-unfold*]:
fixes *A*::'*a*::{*field*}^'*cols*::{*mod-type*}^'*rows*::{*mod-type*}
shows *vec-to-iarray*‘ (*basis-left-null-space A*) = *basis-left-null-space-iarrays* (*matrix-to-iarray A*)
⟨*proof*⟩

lemma *vec-to-iarray-basis-row-space*[*code-unfold*]:

fixes $A::'a::\{field\}\ \widehat{}'cols::\{mod\text{-}type\}\ \widehat{}'rows::\{mod\text{-}type\}$
shows *vec-to-iarray'* (*basis-row-space A*) = *basis-row-space-iarrays* (*matrix-to-iarray A*)
⟨*proof*⟩

corollary *vec-to-iarray-basis-col-space*[*code-unfold*]:
fixes $A::'a::\{field\}\ \widehat{}'cols::\{mod\text{-}type\}\ \widehat{}'rows::\{mod\text{-}type\}$
shows *vec-to-iarray'* (*basis-col-space A*) = *basis-col-space-iarrays* (*matrix-to-iarray A*)
⟨*proof*⟩

end

20 Solving systems of equations using the Gauss Jordan algorithm over nested IArrays

theory *System-Of-Equations-IArrays*
imports
System-Of-Equations
Bases-Of-Fundamental-Subspaces-IArrays
begin

20.1 Previous definitions and properties

definition *greatest-not-zero* :: $'a::\{zero\}$ *iarray* => *nat*
where *greatest-not-zero A* = *the* (*List.find* (λ*n*. *A* !! *n* ≠ *0*) (*rev* [*0*..<*IArray.length A*]))

lemma *vec-to-iarray-exists*:
shows (∃ *b*. *A* $ *b* ≠ *0*) = *IArray.exists* (λ*b*. (*vec-to-iarray A*) !! *b* ≠ *0*) (*IArray*[*0*..<*IArray.length* (*vec-to-iarray A*)])
⟨*proof*⟩

corollary *vec-to-iarray-exists'*:
shows (∃ *b*. *A* $ *b* ≠ *0*) = *IArray.exists* (λ*b*. (*vec-to-iarray A*) !! *b* ≠ *0*) (*IArray* (*rev* [*0*..<*IArray.length* (*vec-to-iarray A*)]))
⟨*proof*⟩

lemma *not-is-zero-iarray-eq-iff*: (∃ *b*. *A* $ *b* ≠ *0*) = (¬ *is-zero-iarray* (*vec-to-iarray A*))
⟨*proof*⟩

lemma *vec-to-iarray-greatest-not-zero*:
assumes *ex-b*: (∃ *b*. *A* $ *b* ≠ *0*)
shows *greatest-not-zero* (*vec-to-iarray A*) = *to-nat* (*GREATEST b*. *A* $ *b* ≠ *0*)
⟨*proof*⟩

20.2 Consistency and inconsistency

definition *consistent-iarrays A b* = (*let GJ*=*Gauss-Jordan-iarrays-PA A*;
rank-A = *length* [*x*←*IArray.list-of* (*snd GJ*) . ¬ *is-zero-iarray x*];
P-mult-b = *fst*(*GJ*) ∗*iv b*
in (*rank-A* ≥ (*if* (¬ *is-zero-iarray P-mult-b*)
then (*greatest-not-zero P-mult-b* + *1*) *else 0*)))

definition *inconsistent-iarrays A b* = (¬ *consistent-iarrays A b*)

lemma *matrix-to-iarray-consistent*[*code*]: *consistent A b* = *consistent-iarrays* (*matrix-to-iarray A*) (*vec-to-iarray b*)
⟨*proof*⟩

lemma *matrix-to-iarray-inconsistent*[*code*]: *inconsistent A b* = *inconsistent-iarrays* (*matrix-to-iarray A*) (*vec-to-iarray b*)
⟨*proof*⟩

definition *solve-consistent-rref-iarrays A b*
= *IArray.of-fun* (λ*j*. *if* (*IArray.exists* (λ*i*. *A* !! *i* !! *j* = *1* ∧ *j*=*least-non-zero-position-of-vector* (*row-iarray i A*)) (*IArray*[*0*..<*nrows-iarray A*]))
then b !! (*least-non-zero-position-of-vector* (*column-iarray j A*)) *else 0*) (*ncols-iarray A*)

lemma *exists-solve-consistent-rref*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *rref*: *reduced-row-echelon-form A*
shows (∃ *i*. *A* $ *i* $ *j* = *1* ∧ *j* = (*LEAST n*. *A* $ *i* $ *n* ≠ *0*))
= (*IArray.exists* (λ*i*. (*matrix-to-iarray A*) !! *i* !! (*to-nat j*) = *1*
∧ (*to-nat j*)=*least-non-zero-position-of-vector* (*row-iarray i* (*matrix-to-iarray A*)))
(*IArray*[*0*..<*nrows-iarray* (*matrix-to-iarray A*)]))
⟨*proof*⟩

lemma *to-nat-the-solve-consistent-rref*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *rref*: *reduced-row-echelon-form A*
and *exists*: (∃ *i*. *A* $ *i* $ *j* = *1* ∧ *j* = (*LEAST n*. *A* $ *i* $ *n* ≠ *0*))
shows *to-nat* (*THE i*. *A* $ *i* $ *j* = *1*) = *least-non-zero-position-of-vector* (*column-iarray* (*to-nat j*) (*matrix-to-iarray A*))
⟨*proof*⟩

lemma *iarray-exhaust2*:
(*xs* = *ys*) = (*IArray.list-of xs* = *IArray.list-of ys*)
⟨*proof*⟩

lemma *vec-to-iarray-solve-consistent-rref*:
fixes *A*::*'a*::{*field*}^*'cols*::{*mod-type*}^*'rows*::{*mod-type*}
assumes *rref*: *reduced-row-echelon-form A*
shows *vec-to-iarray* (*solve-consistent-rref A b*) = *solve-consistent-rref-iarrays* (*matrix-to-iarray A*) (*vec-to-iarray b*)
⟨*proof*⟩

20.3 Independence and dependence

definition *independent-and-consistent-iarrays A b* =
(*let GJ* = *Gauss-Jordan-iarrays-PA A*;
rank-A = *length* [*x*←*IArray.list-of* (*snd GJ*) . ¬ *is-zero-iarray x*];
P-mult-b = *fst GJ* ∗*iv b*;
consistent-A = ((*if* ¬ *is-zero-iarray P-mult-b then greatest-not-zero P-mult-b* + *1 else 0*) ≤ *rank-A*);
dim-solution-set = *ncols-iarray A* − *rank-A*
in consistent-A ∧ *dim-solution-set* = *0*)

definition *dependent-and-consistent-iarrays A b* =
(*let GJ* = *Gauss-Jordan-iarrays-PA A*;
rank-A = *length* [*x*←*IArray.list-of* (*snd GJ*) . ¬ *is-zero-iarray x*];
P-mult-b = *fst GJ* ∗*iv b*;
consistent-A = ((*if* ¬ *is-zero-iarray P-mult-b then greatest-not-zero P-mult-b* + *1 else 0*) ≤ *rank-A*);
dim-solution-set = *ncols-iarray A* − *rank-A*
in consistent-A ∧ *dim-solution-set* > *0*)

lemma *matrix-to-iarray-independent-and-consistent*[*code*]:
shows *independent-and-consistent A b* = *independent-and-consistent-iarrays* (*matrix-to-iarray A*) (*vec-to-iarray b*)
⟨*proof*⟩

lemma *matrix-to-iarray-dependent-and-consistent*[*code*]:
shows *dependent-and-consistent A b* = *dependent-and-consistent-iarrays* (*matrix-to-iarray A*) (*vec-to-iarray b*)
⟨*proof*⟩

20.4 Solve a system of equations over nested IArrays

definition *solve-system-iarrays A b* = (*let A'* = *Gauss-Jordan-iarrays-PA A in* (*snd A'*, *fst A'* ∗*iv b*))

lemma *matrix-to-iarray-fst-solve-system*: *matrix-to-iarray* (*fst* (*solve-system A b*)) = *fst* (*solve-system-iarrays* (*matrix-to-iarray A*) (*vec-to-iarray b*))
⟨*proof*⟩

lemma *vec-to-iarray-snd-solve-system*: *vec-to-iarray* (*snd* (*solve-system A b*)) = *snd* (*solve-system-iarrays* (*matrix-to-iarray A*) (*vec-to-iarray b*))
⟨*proof*⟩

definition *solve-iarrays A b* = (*let GJ-P*=*Gauss-Jordan-iarrays-PA A*;
P-mult-b = *fst GJ-P* ∗*iv b*;
rank-A = *length* [*x*←*IArray.list-of* (*snd GJ-P*) . ¬ *is-zero-iarray x*];
consistent-Ab = (*if* ¬ *is-zero-iarray P-mult-b then greatest-not-zero P-mult-b* + *1 else 0*) ≤ *rank-A*;
GJ-transpose = *Gauss-Jordan-iarrays-PA* (*transpose-iarray A*);
basis = *set* (*map* (λ*i*. *row-iarray i* (*fst GJ-transpose*)) [*rank-A*..<*ncols-iarray A*])
in (*if consistent-Ab then Some* (*solve-consistent-rref-iarrays* (*snd GJ-P*) *P-mult-b*,*basis*) *else None*))

definition *pair-vec-vecset A* = (*if Option.is-none A then None else Some* (*vec-to-iarray* (*fst* (*the A*)), *vec-to-iarray'* (*snd* (*the A*))))

lemma *pair-vec-vecset-solve*[*code-unfold*]:
shows *pair-vec-vecset* (*solve A b*) = *solve-iarrays* (*matrix-to-iarray A*) (*vec-to-iarray b*)
⟨*proof*⟩

end

21 Computing determinants of matrices using the Gauss Jordan algorithm over nested IArrays

theory *Determinants-IArrays*
imports
Determinants2
Gauss-Jordan-IArrays
begin

21.1 Definitions

definition *Gauss-Jordan-in-ij-det-P-iarrays A i j* = (*let n* = *least-non-zero-position-of-vector-from-index* (*column-iarray j A*) *i*
in (*if i* = *n then 1* / *A* !! *i* !! *j else* − *1* / *A* !! *n* !! *j*, *Gauss-Jordan-in-ij-iarrays A i j*))

definition *Gauss-Jordan-column-k-det-P-iarrays A' k* = (*let det-P* = *fst A'*; *i* = *fst* (*snd A'*); *A* = *snd* (*snd A'*)
in if vector-all-zero-from-index (*i*, *column-iarray k A*) ∨ *i* = *nrows-iarray A then* (*det-P*, *i*, *A*)
else let gauss = *Gauss-Jordan-in-ij-det-P-iarrays A i k in* (*fst gauss* ∗ *det-P*, *i* + *1*, *snd gauss*))

definition *Gauss-Jordan-upt-k-det-P-iarrays A k* = (*let foldl* = *foldl Gauss-Jordan-column-k-det-P-iarrays*

(*1*, *0*, *A*) [*0*..<*Suc k*] *in* (*fst foldl*, *snd* (*snd foldl*)))
definition *Gauss-Jordan-det-P-iarrays A* = *Gauss-Jordan-upt-k-det-P-iarrays A* (*ncols-iarray A* − *1*)

21.2 Proofs

A more efficient equation for *Gauss-Jordan-in-ij-det-P-iarrays A i j*.

lemma *Gauss-Jordan-in-ij-det-P-iarrays-code*[*code*]: *Gauss-Jordan-in-ij-det-P-iarrays A i j*
= (*let n* = *least-non-zero-position-of-vector-from-index* (*column-iarray j A*) *i*;
interchange-A = *interchange-rows-iarray A i n*;
A' = *mult-row-iarray interchange-A i* (*1* / *interchange-A* !! *i* !! *j*)
in (*if i* = *n then 1* / *A* !! *i* !! *j else* − *1* / *A* !! *n* !! *j*, *IArray.of-fun* (λ*s*. *if s* = *i then A'* !! *s else row-add-iarray A' s i* (− *interchange-A* !! *s* !! *j*) !! *s*) (*nrows-iarray A*)))
⟨*proof*⟩

lemma *matrix-to-iarray-fst-Gauss-Jordan-in-ij-det-P*:
assumes *ex-n*: ∃ *n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*
shows *fst* (*Gauss-Jordan-in-ij-det-P A i j*) = *fst* (*Gauss-Jordan-in-ij-det-P-iarrays* (*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*))
⟨*proof*⟩

corollary *matrix-to-iarray-fst-Gauss-Jordan-in-ij-det-P'*:
assumes ¬ (*vector-all-zero-from-index* (*to-nat i*, *vec-to-iarray* (*column j A*)))
shows *fst* (*Gauss-Jordan-in-ij-det-P A i j*) = *fst* (*Gauss-Jordan-in-ij-det-P-iarrays* (*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*))
⟨*proof*⟩

lemma *matrix-to-iarray-snd-Gauss-Jordan-in-ij-det-P*:
assumes *ex-n*: ∃ *n*. *A* $ *n* $ *j* ≠ *0* ∧ *i* ≤ *n*
shows *matrix-to-iarray* (*snd* (*Gauss-Jordan-in-ij-det-P A i j*)) = *snd* (*Gauss-Jordan-in-ij-det-P-iarrays* (*matrix-to-iarray A*) (*to-nat i*) (*to-nat j*))
⟨*proof*⟩

lemma *matrix-to-iarray-fst-Gauss-Jordan-column-k-det-P*:
assumes *i*: *i*≤*nrows A* **and** *k*: *k*<*ncols A*
shows *fst* (*Gauss-Jordan-column-k-det-P* (*n*, *i*, *A*) *k*) = *fst* (*Gauss-Jordan-column-k-det-P-iarrays* (*n*, *i*, *matrix-to-iarray A*) *k*)
⟨*proof*⟩

lemma *matrix-to-iarray-fst-snd-Gauss-Jordan-column-k-det-P*:
assumes *i*: *i*≤*nrows A* **and** *k*: *k*<*ncols A*
shows *fst* (*snd* (*Gauss-Jordan-column-k-det-P* (*n*, *i*, *A*) *k*)) = *fst* (*snd* (*Gauss-Jordan-column-k-det-P-iarrays* (*n*, *i*, *matrix-to-iarray A*) *k*))
⟨*proof*⟩

lemma *matrix-to-iarray-snd-snd-Gauss-Jordan-column-k-det-P*:
assumes *i*: *i*≤*nrows A* **and** *k*: *k*<*ncols A*
shows *matrix-to-iarray* (*snd* (*snd* (*Gauss-Jordan-column-k-det-P* (*n*, *i*, *A*) *k*))) =
snd (*snd* (*Gauss-Jordan-column-k-det-P-iarrays* (*n*, *i*, *matrix-to-iarray A*) *k*))
⟨*proof*⟩

lemma *fst-snd-Gauss-Jordan-column-k-det-P-le-nrows*:
assumes *i*: *i*≤*nrows A*
shows *fst* (*snd* (*Gauss-Jordan-column-k-det-P* (*n*, *i*, *A*) *k*)) ≤ *nrows A*
⟨*proof*⟩

The proof of the following theorem is very similar to the ones of *foldl-Gauss-Jordan-column-k-eq*, *rref-and-index-Gauss-Jordan-upt-k* and *foldl-Gauss-Jordan-column-k-det-P*.

lemma
assumes *k*<*ncols A*
shows *matrix-to-iarray-fst-Gauss-Jordan-upt-k-det-P*: *fst* (*Gauss-Jordan-upt-k-det-P*
A k) = *fst* (*Gauss-Jordan-upt-k-det-P-iarrays* (*matrix-to-iarray A*) *k*)
and *matrix-to-iarray-snd-Gauss-Jordan-upt-k-det-P*: *matrix-to-iarray* (*snd* (*Gauss-Jordan-upt-k-det-P*
A k)) = *snd* (*Gauss-Jordan-upt-k-det-P-iarrays* (*matrix-to-iarray A*) *k*)
and *fst* (*snd* (*foldl Gauss-Jordan-column-k-det-P* (*1*, *0*, *A*) [*0*..<*Suc k*])) ≤ *nrows*
A
and *fst* (*snd* (*foldl Gauss-Jordan-column-k-det-P-iarrays* (*1*, *0*, *matrix-to-iarray A*)
[*0*..<*Suc k*])) = *fst* (*snd* (*foldl Gauss-Jordan-column-k-det-P* (*1*, *0*, *A*) [*0*..<*Suc*
k]))
⟨*proof*⟩

lemma *matrix-to-iarray-fst-Gauss-Jordan-det-P*:
shows *fst* (*Gauss-Jordan-det-P A*) = *fst* (*Gauss-Jordan-det-P-iarrays* (*matrix-to-iarray*
A))
⟨*proof*⟩

lemma *matrix-to-iarray-snd-Gauss-Jordan-det-P*:
shows *matrix-to-iarray* (*snd* (*Gauss-Jordan-det-P A*)) = *snd* (*Gauss-Jordan-det-P-iarrays*
(*matrix-to-iarray A*))
⟨*proof*⟩

21.3 Code equations

definition *det-iarrays A* = (*let A′* = *Gauss-Jordan-det-P-iarrays A in prod-list*
(*map* (λ*i*. (*snd A′*) !! *i* !! *i*) [*0*..<*nrows-iarray A*]) / *fst A′*)

lemma *matrix-to-iarray-det*[*code-unfold*]:
fixes *A*::′*a*::{*field*}^′*n*::{*mod-type*}^′*n*::{*mod-type*}
shows *det A* = *det-iarrays* (*matrix-to-iarray A*)
⟨*proof*⟩

end

22 Inverse of a matrix using the Gauss Jordan algorithm over nested IArrays

theory *Inverse-IArrays*
imports
Inverse
Gauss-Jordan-PA-IArrays
begin

22.1 Definitions

definition *invertible-iarray A* = (*rank-iarray A* = *nrows-iarray A*)
definition *inverse-matrix-iarray A* = (*if invertible-iarray A then Some*(*fst*(*Gauss-Jordan-iarrays-PA A*)) *else None*)
definition *matrix-to-iarray-option A* = (*if A* ≠ *None then Some* (*matrix-to-iarray* (*the A*)) *else None*)

22.2 Some lemmas and code generation

lemma *matrix-inv-Gauss-Jordan-iarrays-PA*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
assumes *inv-A*: *invertible A*
shows *matrix-to-iarray* (*matrix-inv A*) = *fst* (*Gauss-Jordan-iarrays-PA* (*matrix-to-iarray A*))
⟨*proof*⟩

lemma *matrix-to-iarray-invertible*[*code-unfold*]:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *invertible A* = *invertible-iarray* (*matrix-to-iarray A*)
⟨*proof*⟩

lemma *matrix-to-iarray-option-inverse-matrix*:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *matrix-to-iarray-option* (*inverse-matrix A*) = (*inverse-matrix-iarray* (*matrix-to-iarray A*))
⟨*proof*⟩

lemma *matrix-to-iarray-option-inverse-matrix-code*[*code-unfold*]:
fixes *A*::*'a*::{*field*}^*'n*::{*mod-type*}^*'n*::{*mod-type*}
shows *matrix-to-iarray-option* (*inverse-matrix A*) = (*let matrix-to-iarray-A* = *matrix-to-iarray A*; *GJ* = *Gauss-Jordan-iarrays-PA matrix-to-iarray-A*
in if nrows-iarray matrix-to-iarray-A = *length* [*x*←*IArray.list-of* (*snd GJ*) . ¬ *is-zero-iarray x*] *then Some* (*fst GJ*) *else None*)
⟨*proof*⟩

lemma[*code-unfold*]:

shows *inverse-matrix-iarray A* = (*let A'* = (*Gauss-Jordan-iarrays-PA A*); *nrows* = *IArray.length A in*
(*if length* [*x*←*IArray.list-of* (*snd A'*) . ¬ *is-zero-iarray x*] = *nrows*
then Some (*fst A'*) *else None*))
⟨*proof*⟩

end

23 Examples of computations over matrices represented as nested IArrays

theory *Examples-Gauss-Jordan-IArrays*
imports
System-Of-Equations-IArrays
Determinants-IArrays
Inverse-IArrays
Code-Z2
HOL−Library.Code-Target-Numeral

begin

23.1 Transformations between nested lists nested IArrays

definition *iarray-of-iarray-to-list-of-list* :: *'a iarray iarray* => *'a list list*
where *iarray-of-iarray-to-list-of-list A* = *map IArray.list-of* (*map* ((!!) *A*) [*0..<IArray.length A*])

The following definitions are also in the file *Examples-on-Gauss-Jordan-Abstract.*

Definitions to transform a matrix to a list of list and vice versa

definition *vec-to-list* :: *'a^'n*::{*finite*, *enum*} => *'a list*
where *vec-to-list A* = *map* (($) *A*) (*enum-class.enum*::*'n list*)

definition *matrix-to-list-of-list* :: *'a^'n*::{*finite*, *enum*}*^'m*::{*finite*, *enum*} => *'a list list*
where *matrix-to-list-of-list A* = *map* (*vec-to-list*) (*map* (($) *A*) (*enum-class.enum*::*'m list*))

This definition should be equivalent to *vector-def* (in suitable types)

definition *list-to-vec* :: *'a list* => *'a^'n*::{*enum*, *mod-type*}
where *list-to-vec xs* = *vec-lambda* (% *i*. *xs* ! (*to-nat i*))

lemma [*code abstract*]: *vec-nth* (*list-to-vec xs*) = (%*i*. *xs* ! (*to-nat i*))
⟨*proof*⟩

definition *list-of-list-to-matrix* :: *'a list list* => *'a^'n*::{*enum*, *mod-type*}*^'m*::{*enum*, *mod-type*}

where *list-of-list-to-matrix xs* = *vec-lambda* (%*i*. *list-to-vec* (*xs* ! (*to-nat i*)))

lemma [*code abstract*]: *vec-nth* (*list-of-list-to-matrix xs*) = (%*i*. *list-to-vec* (*xs* ! (*to-nat i*)))
⟨*proof*⟩

23.2 Examples

The following three lemmas are presented in both this file and in the *Examples-Gauss-Jordan-Abstract* one. They allow a more convenient printing of rational and real numbers after evaluation. They have already been added to the repository version of Isabelle, so after Isabelle2014 they should be removed from here.

lemma [*code-post*]:
int-of-integer (− *1*) = − *1*
⟨*proof*⟩

lemma [*code-abbrev*]:
(*of-rat* (− *1*) :: *real*) = − *1*
⟨*proof*⟩

lemma [*code-post*]:
(*of-rat* (− (*1* / *numeral k*)) :: *real*) = − *1* / *numeral k*
(*of-rat* (− (*numeral k* / *numeral l*)) :: *real*) = − *numeral k* / *numeral l*
⟨*proof*⟩

From here on, we do the computations in two ways. The first one consists of executing the abstract functions (which internally will execute the ones over iarrays). The second one runs directly the functions over iarrays.

23.2.1 Ranks, dimensions and Gauss Jordan algorithm

In the following examples, the theorem *matrix-to-iarray-rank* (which is the file *Gauss-Jordan-IArrays* and it is a code unfold theorem) assures that the computation will be carried out using the iarrays representation.

value *vec.dim* (*col-space* (*list-of-list-to-matrix* [[*1*,*0*,*0*,*7*,*5*],[*1*,*0*,*4*,*8*,−*1*],[*1*,*0*,*0*,*9*,*8*],[*1*,*2*,*3*,*6*,*5*]]::*real^5^4*))
value *rank* (*list-of-list-to-matrix* [[*1*,*0*,*0*,*7*,*5*],[*1*,*0*,*4*,*8*,−*1*],[*1*,*0*,*0*,*9*,*8*],[*1*,*2*,*3*,*6*,*5*]]::*real^5^4*)

value *vec.dim* (*null-space* (*list-of-list-to-matrix* [[*1*,*0*,*0*,*7*,*5*],[*1*,*0*,*4*,*8*,−*1*],[*1*,*0*,*0*,*9*,*8*],[*1*,*2*,*3*,*6*,*5*]]::*rat^5^4*))

value *rank-iarray* (*IArray*[*IArray*[*1*::*rat*,*0*,*0*,*7*,*5*],*IArray*[*1*,*0*,*4*,*8*,−*1*],*IArray*[*1*,*0*,*0*,*9*,*8*],*IArray*[*1*,*2*,*3*,*6*,*5*]])

value *rank-iarray* (*IArray*[*IArray*[*1*::*real*,*0*,*1*],*IArray*[*1*,*1*,*0*],*IArray*[*0*,*1*,*1*]])
value *rank-iarray* (*IArray*[*IArray*[*1*::*bit*,*0*,*1*],*IArray*[*1*,*1*,*0*],*IArray*[*0*,*1*,*1*]])

Examples on computing the Gauss Jordan algorithm.

value *iarray-of-iarray-to-list-of-list* (*matrix-to-iarray* (*Gauss-Jordan* (*list-of-list-to-matrix* [[*Complex 1 1*,*Complex 1* (− *1*), *Complex 0 0*],[*Complex 2* (− *1*),*Complex 1 3*, *Complex 7 3*]]::*complex*^*3*^*2*)))
value *iarray-of-iarray-to-list-of-list* (*Gauss-Jordan-iarrays*(*IArray*[*IArray*[*Complex 1 1*,*Complex 1* (− *1*),*Complex 0 0*],*IArray*[*Complex 2* (− *1*),*Complex 1 3*,*Complex 7 3*]]))

23.2.2 Inverse of a matrix

Examples on inverting matrices

definition *print-result-some-iarrays A* = (*if A* = *None then None else Some* (*iarray-of-iarray-to-list-of-list* (*the A*)))

value *let A*=(*list-of-list-to-matrix* [[*1*,*1*,*2*,*4*,*5*,*9*,*8*],[*3*,*0*,*8*,*4*,*5*,*0*,*8*],[*3*,*2*,*0*,*4*,*5*,*9*,*8*],[*3*,*2*,*8*,*0*,*5*,*9*,*8*],[*3*,*2*,*8*,*4*,*0
in print-result-some-iarrays (*matrix-to-iarray-option* (*inverse-matrix A*))
value *let A*=(*IArray*[*IArray*[*1*::*real*,*1*,*2*,*4*,*5*,*9*,*8*],*IArray*[*3*,*0*,*8*,*4*,*5*,*0*,*8*],*IArray*[*3*,*2*,*0*,*4*,*5*,*9*,*8*],*IArray*[*3*,*2*,*8*,*0
in print-result-some-iarrays (*inverse-matrix-iarray A*)

23.2.3 Determinant of a matrix

Examples on computing determinants of matrices

value *det* (*list-of-list-to-matrix* ([[*1*,*8*,*9*,*1*,*47*],[*7*,*2*,*2*,*5*,*9*],[*3*,*2*,*7*,*7*,*4*],[*9*,*8*,*7*,*5*,*1*],[*1*,*2*,*6*,*4*,*5*]])::*rat*^*5*^*5*)
value *det* (*list-of-list-to-matrix* [[*1*,*2*,*7*,*8*,*9*],[*3*,*4*,*12*,*10*,*7*],[−*5*,*4*,*8*,*7*,*4*],[*0*,*1*,*2*,*4*,*8*],[*9*,*8*,*7*,*13*,*11*]]::*rat*^*5*^*5*)

value *det-iarrays* (*IArray*[*IArray*[*1*::*real*,*2*,*7*,*8*,*9*],*IArray*[*3*,*4*,*12*,*10*,*7*],*IArray*[−*5*,*4*,*8*,*7*,*4*],*IArray*[*0*,*1*,*2*,*4*,*8*],
value *det-iarrays* (*IArray*[*IArray*[*286*,*662*,*263*,*246*,*642*,*656*,*351*,*454*,*339*,*848*],
IArray[*307*,*489*,*667*,*908*,*103*,*47*,*120*,*133*,*85*,*834*],
IArray[*69*,*732*,*285*,*147*,*527*,*655*,*732*,*661*,*846*,*202*],
IArray[*463*,*855*,*78*,*338*,*786*,*954*,*593*,*550*,*913*,*378*],
IArray[*90*,*926*,*201*,*362*,*985*,*341*,*540*,*912*,*494*,*427*],
IArray[*384*,*511*,*12*,*627*,*131*,*620*,*987*,*996*,*445*,*216*],
IArray[*385*,*538*,*362*,*643*,*567*,*804*,*499*,*914*,*332*,*512*],
IArray[*879*,*159*,*312*,*187*,*827*,*503*,*823*,*893*,*139*,*546*],
IArray[*800*,*376*,*331*,*363*,*840*,*737*,*911*,*886*,*456*,*848*],
IArray[*900*,*737*,*280*,*370*,*121*,*195*,*958*,*862*,*957*,*754*::*real*]])

23.2.4 Bases of the fundamental subspaces

Examples on computing basis for null space, row space, column space and left null space.

value *let A* = (*list-of-list-to-matrix* ([[*1*,*3*,−*2*,*0*,*2*,*0*],[*2*,*6*,−*5*,−*2*,*4*,−*3*],[*0*,*0*,*5*,*10*,*0*,*15*],[*2*,*6*,*0*,*8*,*4*,*18*]])::*real*

in vec-to-list‘ (*basis-null-space A*)
value *let A* = (*list-of-list-to-matrix* ([[*3*,*4*,*0*,*7*],[*1*,−*5*,*2*,−*2*],[−*1*,*4*,*0*,*3*],[*1*,−*1*,*2*,*2*]])::*real*^*4*^*4*)

in vec-to-list‘ (*basis-null-space A*)

value *let A = (IArray[IArray[1::real,3,−2,0,2,0],IArray[2,6,−5,−2,4,−3],IArray[0,0,5,10,0,15],IArray[2,6*

in IArray.list-of‘ (basis-null-space-iarrays A)
value *let A = (IArray[IArray[3::real,4,0,7],IArray[1,−5,2,−2],IArray[−1,4,0,3],IArray[1,−1,2,2]])*
in IArray.list-of‘ (basis-null-space-iarrays A)

value *let A = (list-of-list-to-matrix ([[1,3,−2,0,2,0],[2,6,−5,−2,4,−3],[0,0,5,10,0,15],[2,6,0,8,4,18]])::real*

in vec-to-list‘ (basis-row-space A)
value *let A = (list-of-list-to-matrix ([[3,4,0,7],[1,−5,2,−2],[−1,4,0,3],[1,−1,2,2]])::real^4^4)*

in vec-to-list‘ (basis-row-space A)

value *let A = (IArray[IArray[1::real,3,−2,0,2,0],IArray[2,6,−5,−2,4,−3],IArray[0,0,5,10,0,15],IArray[2,6*

in IArray.list-of‘ (basis-row-space-iarrays A)
value *let A = (IArray[IArray[3::real,4,0,7],IArray[1,−5,2,−2],IArray[−1,4,0,3],IArray[1,−1,2,2]])*
in IArray.list-of‘ (basis-row-space-iarrays A)

value *let A = (list-of-list-to-matrix ([[1,3,−2,0,2,0],[2,6,−5,−2,4,−3],[0,0,5,10,0,15],[2,6,0,8,4,18]])::real*

in vec-to-list‘ (basis-col-space A)
value *let A = (list-of-list-to-matrix ([[3,4,0,7],[1,−5,2,−2],[−1,4,0,3],[1,−1,2,2]])::real^4^4)*

in vec-to-list‘ (basis-col-space A)

value *let A = (IArray[IArray[1::real,3,−2,0,2,0],IArray[2,6,−5,−2,4,−3],IArray[0,0,5,10,0,15],IArray[2,6*

in IArray.list-of‘ (basis-col-space-iarrays A)
value *let A = (IArray[IArray[3::real,4,0,7],IArray[1,−5,2,−2],IArray[−1,4,0,3],IArray[1,−1,2,2]])*
in IArray.list-of‘ (basis-col-space-iarrays A)

value *let A = (list-of-list-to-matrix ([[1,3,−2,0,2,0],[2,6,−5,−2,4,−3],[0,0,5,10,0,15],[2,6,0,8,4,18]])::real*

in vec-to-list‘ (basis-left-null-space A)
value *let A = (list-of-list-to-matrix ([[3,4,0,7],[1,−5,2,−2],[−1,4,0,3],[1,−1,2,2]])::real^4^4)*

in vec-to-list‘ (basis-left-null-space A)

value *let A = (IArray[IArray[1::real,3,−2,0,2,0],IArray[2,6,−5,−2,4,−3],IArray[0,0,5,10,0,15],IArray[2,6*

in IArray.list-of‘ (basis-left-null-space-iarrays A)
value *let A = (IArray[IArray[3::real,4,0,7],IArray[1,−5,2,−2],IArray[−1,4,0,3],IArray[1,−1,2,2]])*
in IArray.list-of‘ (basis-left-null-space-iarrays A)

23.2.5 Consistency and inconsistency

Examples on checking the consistency/inconsistency of a system of equations. The theorems *matrix-to-iarray-independent-and-consistent* and *matrix-to-iarray-dependent-and-consistent* which are code theorems and they are in the file *System-Of-Equations-IArrays* assure the execution using the iarrays representation.

value *independent-and-consistent* (*list-of-list-to-matrix* ([[*1*,*0*,*0*],[*0*,*1*,*0*],[*0*,*0*,*1*],[*0*,*0*,*0*],[*0*,*0*,*0*]])::*real^3^5*)
(*list-to-vec*([*2*,*3*,*4*,*0*,*0*])::*real^5*)
value *consistent* (*list-of-list-to-matrix* ([[*1*,*0*,*0*],[*0*,*1*,*0*],[*0*,*0*,*1*],[*0*,*0*,*0*],[*0*,*0*,*0*]])::*real^3^5*)
(*list-to-vec*([*2*,*3*,*4*,*0*,*0*])::*real^5*)
value *inconsistent* (*list-of-list-to-matrix* ([[*1*,*0*,*0*],[*0*,*1*,*0*],[*3*,*0*,*1*],[*0*,*7*,*0*],[*0*,*0*,*9*]])::*real^3^5*)
(*list-to-vec*([*2*,*0*,*4*,*0*,*0*])::*real^5*)
value *dependent-and-consistent* (*list-of-list-to-matrix* ([[*1*,*0*,*0*],[*0*,*1*,*0*]])::*real^3^2*)
(*list-to-vec*([*3*,*4*])::*real^2*)
value *independent-and-consistent* (*mat 1*::*real^3^3*) (*list-to-vec*([*3*,*4*,*5*])::*real^3*)

23.2.6 Solving systems of linear equations

Examples on solving linear systems.

definition *print-result-system-iarrays A* = (*if A* = *None then None else Some*
(*IArray.list-of* (*fst* (*the A*)), *IArray.list-of'* (*snd* (*the A*))))

value *let A* = (*list-of-list-to-matrix* [[*0*,*0*,*0*],[*0*,*0*,*0*],[*0*,*0*,*1*]]::*real^3^3*); *b*=(*list-to-vec*
[*4*,*5*,*0*]::*real^3*);
result = *pair-vec-vecset* (*solve A b*)
in print-result-system-iarrays (*result*)
value *let A* = (*list-of-list-to-matrix* [[*3*,*2*,*5*,*2*,*7*],[*6*,*4*,*7*,*4*,*5*],[*3*,*2*,−*1*,*2*,−*11*],[*6*,*4*,*1*,*4*,−*13*]]::*real^5^4*);
b=(*list-to-vec* [*0*,*0*,*0*,*0*]::*real^4*);
result = *pair-vec-vecset* (*solve A b*)
in print-result-system-iarrays (*result*)
value *let A* = (*list-of-list-to-matrix* [[*4*,*5*,*8*],[*9*,*8*,*7*],[*4*,*6*,*1*]]::*real^3^3*); *b*=(*list-to-vec*
[*4*,*5*,*8*]::*real^3*);
result = *pair-vec-vecset* (*solve A b*)
in print-result-system-iarrays (*result*)

value *let A* = (*IArray*[*IArray*[*0*::*real*,*0*,*0*],*IArray*[*0*,*0*,*0*],*IArray*[*0*,*0*,*1*]]); *b*=(*IArray*[*4*,*5*,*0*]);
result = (*solve-iarrays A b*)
in print-result-system-iarrays (*result*)
value *let A* = (*IArray*[*IArray*[*3*::*real*,*2*,*5*,*2*,*7*],*IArray*[*6*,*4*,*7*,*4*,*5*],*IArray*[*3*,*2*,−*1*,*2*,−*11*],*IArray*[*6*,*4*,*1*,*4*,−*13*]])
b=(*IArray*[*0*,*0*,*0*,*0*]);
result = (*solve-iarrays A b*)
in print-result-system-iarrays (*result*)
value *let A* = (*IArray*[*IArray*[*4*,*5*,*8*],*IArray*[*9*::*real*,*8*,*7*],*IArray*[*4*,*6*::*real*,*1*]]); *b*=(*IArray*[*4*,*5*,*8*]);
result = (*solve-iarrays A b*)
in print-result-system-iarrays (*result*)

export-code

rank-iarray
inverse-matrix-iarray
det-iarrays
consistent-iarrays
inconsistent-iarrays
independent-and-consistent-iarrays
dependent-and-consistent-iarrays
basis-left-null-space-iarrays
basis-null-space-iarrays
basis-col-space-iarrays
basis-row-space-iarrays
solve-iarrays
in *SML*
end

24 Exporting code to SML and Haskell

theory *Code-Generation-IArrays*
imports
Examples-Gauss-Jordan-IArrays
begin

24.1 Exporting code

The following two equations are necessary to execute code. If we don't remove them from code unfold, the exported code will not work (there exist problems with the number 1 and number 0. Those problems appear when the HMA library is imported).

lemma [*code-unfold del*]: *1* ≡ *real-of-rat 1* ⟨*proof*⟩
lemma [*code-unfold del*]: *0* ≡ *real-of-rat 0* ⟨*proof*⟩

definition *matrix-z2* = *IArray*[*IArray*[*0*,*1*], *IArray*[*1*,*1*::*bit*], *IArray*[*1*,*0*::*bit*]]
definition *matrix-rat* = *IArray*[*IArray*[*1*,*0*,*8*], *IArray*[*5.7*,*22*,*1*], *IArray*[*41*,−*58*/*7*,*78*::*rat*]]
definition *matrix-real* = *IArray*[*IArray*[*0*,*1*], *IArray*[*1*,−*2*::*real*]]
definition *vec-rat* = *IArray*[*21*,*5*,*7*::*rat*]

definition *print-result-Gauss A* = *iarray-of-iarray-to-list-of-list* (*Gauss-Jordan-iarrays A*)
definition *print-rank A* = *rank-iarray A*
definition *print-det A* = *det-iarrays A*

definition *print-result-z2* = *print-result-Gauss* (*matrix-z2*)
definition *print-result-rat* = *print-result-Gauss* (*matrix-rat*)
definition *print-result-real* = *print-result-Gauss* (*matrix-real*)

definition *print-rank-z2* = *print-rank* (*matrix-z2*)
definition *print-rank-rat* = *print-rank* (*matrix-rat*)
definition *print-rank-real* = *print-rank* (*matrix-real*)

definition *print-det-rat* = *print-det* (*matrix-rat*)
definition *print-det-real* = *print-det* (*matrix-real*)

definition *print-inverse A* = *inverse-matrix-iarray A*
definition *print-inverse-real A* = *print-inverse* (*matrix-real*)
definition *print-inverse-rat A* = *print-inverse* (*matrix-rat*)

definition *print-system A b* = *print-result-system-iarrays* (*solve-iarrays A b*)
definition *print-system-rat* = *print-result-system-iarrays* (*solve-iarrays matrix-rat vec-rat*)

24.2 The Mathematica bug

Varona et al [1] detected that the commercial software *Mathematica*®, in its versions 8.0, 9.0 and 9.0.1, was computing erroneously determinants of matrices of big integers, even for small dimensions (in their work they present an example of a matrix of dimension 14×14). The situation is such that even the same determinant, computed twice, produces two different and wrong results.

Here we reproduce that example. The computation of the determinant is correctly carried out by the exported code from this formalization, in both SML and Haskell.

definition *powersMatrix* =
IArray[
IArray[*10^123*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*],
IArray[*0*,*10^152*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*],
IArray[*0*,*0*,*10^185*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*],
IArray[*0*,*0*,*0*,*10^220*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*],
IArray[*0*,*0*,*0*,*0*,*10^397*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*],
IArray[*0*,*0*,*0*,*0*,*0*,*10^449*,*0*,*0*,*0*,*0*,*0*,*0*,*0*],
IArray[*0*,*0*,*0*,*0*,*0*,*0*,*10^503*,*0*,*0*,*0*,*0*,*0*,*0*],
IArray[*0*,*0*,*0*,*0*,*0*,*0*,*0*,*10^563*,*0*,*0*,*0*,*0*,*0*],
IArray[*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*10^979*,*0*,*0*,*0*,*0*],
IArray[*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*10^1059*,*0*,*0*,*0*],
IArray[*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*10^1143*,*0*,*0*],
IArray[*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*10^1229*,*0*,*0*],
IArray[*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*10^1319*,*0*],
IArray[*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,*0*,(*10^1412*)::*rat*]]

definition *basicMatrix* =
IArray[
IArray[*−32*, *69*, *89*,*−60*,*−83*,*−22*,*−14*,*−58*,*85*,*56*,*−65*,*−30*,*−86*,*−9*],
IArray[*6*, *99*, *11*, *57*, *47*,*−42*,*−48*,*−65*, *25*, *50*,*−70*,*−3*,*−90*, *31*],
IArray[*78*, *38*, *12*, *64*,*−67*,*−4*,*−52*,*−65*, *19*, *71*, *38*,*−17*, *51*,*−3*],
IArray[*−93*, *30*, *89*, *22*, *13*, *48*,*−73*, *93*, *11*,*−97*,*−49*, *61*,*−25*,*−4*],
IArray[*54*,*−22*, *54*,*−53*,*−52*, *64*, *19*, *1*, *81*,*−72*,*−11*, *50*, *0*,*−81*],

IArray[*65*,−*58*, *3*, *57*, *19*, *77*, *76*,−*57*,−*80*, *22*, *93*,−*85*, *67*, *58*],
IArray[*29*,−*58*, *47*, *87*, *3*,−*6*,−*81*, *5*, *98*, *86*,−*98*, *51*,−*62*,−*66*],
IArray[*93*,−*77*, *16*,−*64*, *48*, *84*, *97*, *75*, *89*, *63*, *34*,−*98*,−*94*, *19*],
IArray[*45*,−*99*, *3*,−*57*, *32*, *60*, *74*, *4*, *69*, *98*,−*40*,−*69*,−*28*,−*26*],
IArray[−*13*, *51*,−*99*,−*2*, *48*, *71*,−*81*,−*32*, *78*, *27*,−*28*,−*22*, *22*, *94*],
IArray[*11*, *72*,−*74*, *86*, *79*,−*58*,−*89*, *80*, *70*, *55*,−*49*, *51*,−*42*, *66*],
IArray[−*72*, *53*, *49*,−*46*, *17*,−*22*,−*48*,−*40*,−*28*,−*85*, *88*,−*30*, *74*, *32*],
IArray[−*92*,−*22*,−*90*, *67*,−*25*,−*28*,−*91*,−*8*, *32*,−*41*, *10*, *6*, *85*, *21*],
IArray[*47*,−*73*,−*30*,−*60*, *99*, *9*,−*86*,−*70*, *84*, *55*, *19*, *69*, *11*,−*84*::*rat*]]

definition *smallMatrix*=
IArray[
IArray[*528*, *853*,−*547*,−*323*, *393*,−*916*,−*11*,−*976*, *279*,−*665*, *906*,−*277*, *103*,−*485*],
IArray[*878*, *910*,−*306*,−*260*, *575*,−*765*,−*32*, *94*, *254*, *276*,−*156*, *625*,−*8*,−*566*],
IArray[−*357*, *451*,−*475*, *327*,−*84*, *237*, *647*, *505*,−*137*, *363*,−*808*, *332*, *222*,−*998*],
IArray[−*76*, *26*,−*778*, *505*, *942*,−*561*,−*350*, *698*,−*532*,−*507*,−*78*,−*758*, *346*,−*545*],
IArray[−*358*, *18*,−*229*,−*880*,−*955*,−*346*, *550*,−*958*, *867*,−*541*,−*962*, *646*, *932*,
168],
IArray[*192*, *233*, *620*,*955*,−*877*, *281*, *357*,−*226*,−*820*, *513*,−*882*, *536*,−*237*, *877*],
IArray[−*234*,−*71*,−*831*, *880*,−*135*,−*249*,−*427*, *737*, *664*, *298*,−*552*,−*1*,−*712*,−*691*],
IArray[*80*, *748*, *684*, *332*, *730*,−*111*,−*643*, *102*,−*242*,−*82*,−*28*, *585*, *207*,−*986*],
IArray[*967*, *1*,−*494*, *633*, *891*,−*907*,−*586*, *129*, *688*, *150*,−*501*,−*298*, *704*,−*68*],
IArray[*406*,−*944*,−*533*,−*827*, *615*, *907*,−*443*,−*350*, *700*,−*878*, *706*, *1*,*800*, *120*],
IArray[*33*,−*328*,−*543*, *583*,−*443*,−*635*,*904*,−*745*,−*398*,−*110*, *751*, *660*, *474*, *255*],
IArray[−*537*,−*311*, *829*, *28*, *175*, *182*,−*930*, *258*,−*808*,−*399*,−*43*,−*68*,−*553*, *421*],
IArray[−*373*,−*447*,−*252*,−*619*,−*418*, *764*, *994*,−*543*,−*37*,−*845*, *30*,−*704*, *147*,−*534*],
IArray[*638*,−*33*, *932*,−*335*,−*75*,−*676*,−*934*, *239*, *210*, *665*, *414*,−*803*, *564*,−*805*::*rat*]]

definition *bigMatrix*=(*basicMatrix* ∗∗*i* *powersMatrix*) + *smallMatrix*

end

25 Exporting code to SML

theory *Code-Generation-IArrays-SML*
imports
HOL−Library.Code-Real-Approx-By-Float
Code-Generation-IArrays
begin

Some serializations of gcd and abs in SML. Since gcd is not part of the standard SML library, we have serialized it to the corresponding operation in PolyML and MLton.

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*]]

There are several ways to serialize div and mod. The following ones are four examples of it:

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

export-code
print-rank-real
print-rank-rat
print-rank-z2
print-rank
print-result-real
print-result-rat
print-result-z2
print-result-Gauss
print-det-rat
print-det-real
print-det
print-inverse-real
print-inverse-rat
print-inverse
print-system-rat
print-system
in *SML* **module-name** *Gauss-SML*

end

26 Code Generation for rational numbers in Haskell

theory *Code-Rational*
imports

HOL−Library.Code-Real-Approx-By-Float
Code-Generation-IArrays

HOL−Library.Code-Target-Int
begin

26.1 Serializations

The following *code-printing code-module* module is the usual way to import libraries in Haskell. In this case, we rebind some functions from Data.Ratio. See https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2013-June/msg00007.html

code-printing code-module *Rational* ⇀ (*Haskell*)
‹*module Rational*(*fract*, *numerator*, *denominator*) *where*

import qualified Data.Ratio
import Data.Ratio(*numerator*, *denominator*)

fract (*a*, *b*) = *a Data.Ratio.*% *b*›

context
includes *integer.lifting*
begin
lift-definition *Frct-integer* :: *integer* × *integer* => *rat*
is *Frct* :: *int* × *int* => *rat* ⟨*proof*⟩
end

consts *Foo*::*rat*
code-datatype *Foo*

context
includes *integer.lifting*
begin

lemma [*code*]:

Frct a = Frct-integer ((of-int (fst a)), (of-int (snd a)))
⟨*proof*⟩

lemma [*code*]:
Rat.of-int a = Frct-integer (of-int a, 1)
⟨*proof*⟩

definition *numerator* :: *rat => int*
where *numerator x = fst (quotient-of x)*

definition *denominator* :: *rat => int*
where *denominator x = snd (quotient-of x)*

lift-definition *numerator-integer* :: *rat => integer*
is *numerator* ⟨*proof*⟩

lift-definition *denominator-integer* :: *rat => integer*
is *denominator* ⟨*proof*⟩

lemma [*code*]:
inverse x = Frct-integer (denominator-integer x, numerator-integer x)
⟨*proof*⟩

lemma *quotient-of-num-den*: *quotient-of x = ((numerator x), (denominator x))*
⟨*proof*⟩

lemma [*code*]: *quotient-of x = (int-of-integer (numerator-integer x), int-of-integer(denominator-integer x))*
⟨*proof*⟩
end

code-printing
type-constructor *rat* → (*Haskell*) *Prelude.Rational*
| **class-instance** *rat* :: *HOL.equal* => (*Haskell*) −
| **constant** *0* :: *rat* → (*Haskell*) (*Prelude.toRational* (*0*::*Integer*))
| **constant** *1* :: *rat* → (*Haskell*) (*Prelude.toRational* (*1*::*Integer*))
| **constant** *Frct-integer* → (*Haskell*) *Rational.fract* (-)
| **constant** *numerator-integer* → (*Haskell*) *Rational.numerator* (-)
| **constant** *denominator-integer* → (*Haskell*) *Rational.denominator* (-)
| **constant** *HOL.equal* :: *rat* ⇒ *rat* ⇒ *bool* →
(*Haskell*) (-) == (-)
| **constant** (<) :: *rat => rat => bool* →
(*Haskell*) - < -
| **constant** (≤) :: *rat => rat => bool* →
(*Haskell*) - <= -
| **constant** (+) :: *rat* ⇒ *rat* ⇒ *rat* →
(*Haskell*) (-) + (-)
| **constant** (−) :: *rat* ⇒ *rat* ⇒ *rat* →

(*Haskell*) (-) − (-)
| **constant** (∗) :: *rat* ⇒ *rat* ⇒ *rat* ⇀
(*Haskell*) (-) ∗ (-)
| **constant** (/) :: *rat* ⇒ *rat* ⇒ *rat* ⇀
(*Haskell*) (-) ′/ (-)
| **constant** *uminus* :: *rat* => *rat* ⇀
(*Haskell*) *Prelude.negate*

end

27 Exporting code to Haskell

theory *Code-Generation-IArrays-Haskell*
imports
Code-Rational
begin

export-code
print-rank-real
print-rank-rat
print-rank-z2
print-rank
print-result-real
print-result-rat
print-result-z2
print-result-Gauss
print-det-rat
print-det-real
print-det
print-inverse-real
print-inverse-rat
print-inverse
print-system-rat
print-system
in *Haskell*
module-name *Gauss-Haskell*

end

References

[1] M. P. Antonio J. Durán and J. L. Varona. Misfortunes of a mathematicians' trio using computer algebra systems: Can we trust? *CoRR*, abs/1312.3270, 2013. http://arxiv.org/abs/1312.3270.