

Gauss-Jordan algorithm and its applications

By Jose Divasón and Jesús Aransay*

May 26, 2024

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

1	Reduced row echelon form	6
1.1	Defining the concept of Reduced Row Echelon Form	6

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

1.1.1	Previous definitions and properties	6
1.1.2	Definition of reduced row echelon form up to a column	7
1.1.3	The definition of reduced row echelon form	9
1.2	Properties of the reduced row echelon form of a matrix	9
2	Code set	11
3	Code generation for vectors and matrices	12
4	Elementary Operations over matrices	14
4.1	Some previous results:	14
4.2	Definitions of elementary row and column operations	14
4.3	Properties about elementary row operations	15
4.3.1	Properties about interchanging rows	15
4.3.2	Properties about multiplying a row by a constant	15
4.3.3	Properties about adding a row multiplied by a constant to another row	16
4.4	Properties about elementary column operations	16
4.4.1	Properties about interchanging columns	16
4.4.2	Properties about multiplying a column by a constant	16
4.4.3	Properties about adding a column multiplied by a constant to another column	16
4.5	Relationships amongst the definitions	17
4.6	Code Equations	17
5	Rank of a matrix	19
5.1	Row rank, column rank and rank	19
5.2	Properties	19
6	Gauss Jordan algorithm over abstract matrices	20
6.1	The Gauss-Jordan Algorithm	20
6.2	Properties about rref and the greatest nonzero row.	21
6.3	The proof of its correctness	22
6.4	Lemmas for code generation and rank computation	38
7	Linear Maps	38
7.1	Properties about ranks and linear maps	39
7.2	Invertible linear maps	39
7.3	Definition and properties of the set of a vector	41
7.4	Coordinates of a vector	43
7.5	Matrix of change of basis and coordinate matrix of a linear map	44
7.6	Equivalent Matrices	47
7.7	Similar matrices	48

8	Obtaining explicitly the invertible matrix which transforms a matrix to its reduced row echelon form	49
8.1	Definitions	49
8.2	Proofs	50
8.2.1	Properties about <i>Gauss-Jordan-in-ij-PA</i>	50
8.2.2	Properties about <i>Gauss-Jordan-column-k-PA</i>	50
8.2.3	Properties about <i>Gauss-Jordan-upt-k-PA</i>	51
8.2.4	Properties about <i>Gauss-Jordan-PA</i>	51
8.2.5	Proving that the transformation has been carried out by means of elementary operations	51
9	Computing determinants of matrices using the Gauss Jordan algorithm	53
9.1	Some previous properties	54
9.1.1	Relationships between determinants and elementary row operations	54
9.1.2	Relationships between determinants and elementary column operations	54
9.2	Proving that the determinant can be computed by means of the Gauss Jordan algorithm	55
9.2.1	Previous properties	55
9.2.2	Definitions	56
9.2.3	Proofs	56
10	Inverse of a matrix using the Gauss Jordan algorithm	60
10.1	Several properties	60
10.2	Computing the inverse of a matrix using the Gauss Jordan algorithm	61
11	Bases of the four fundamental subspaces	62
11.1	Computation of the bases of the fundamental subspaces	62
11.2	Relationships amongst the bases	62
11.3	Code equations	63
11.4	Demonstrations that they are bases	63
12	Solving systems of equations using the Gauss Jordan algorithm	65
12.1	Definitions	65
12.2	Relationship between <i>is-solution-def</i> and <i>solve-system-def</i>	65
12.3	Consistent and inconsistent systems of equations	65
12.4	Solution set of a system of equations. Dependent and independent systems.	68
12.5	Solving systems of linear equations	70

13 Code Generation for Z2	72
14 Examples of computations over abstract matrices	73
14.1 Transforming a list of lists to an abstract matrix	73
14.2 Examples	74
14.2.1 Ranks and dimensions	74
14.2.2 Inverse of a matrix	75
14.2.3 Determinant of a matrix	75
14.2.4 Bases of the fundamental subspaces	75
14.2.5 Consistency and inconsistency	76
14.2.6 Solving systems of linear equations	76
15 IArrays Addenda	77
15.1 Some previous instances	77
15.2 Some previous definitions and properties for IArrays	77
15.3 Code generation	77
16 Matrices as nested IArrays	78
16.1 Isomorphism between matrices implemented by vecs and matrices implemented by iarrays	78
16.1.1 Isomorphism between vec and iarray	78
16.1.2 Isomorphism between matrix and nested iarrays	79
16.2 Definition of operations over matrices implemented by iarrays	80
16.2.1 Properties of previous definitions	81
16.3 Definition of elementary operations	82
16.3.1 Code generator	83
17 Gauss Jordan algorithm over nested IArrays	84
17.1 Definitions and functions to compute the Gauss-Jordan algorithm over matrices represented as nested iarrays	85
17.2 Proving the equivalence between Gauss-Jordan algorithm over nested iarrays and over nested vecs (abstract matrices).	85
17.3 Implementation over IArrays of the computation of the <i>rank</i> of a matrix	87
17.3.1 Proving the equivalence between <i>rank</i> and <i>rank-iarray</i>	88
17.3.2 Code equations for computing the rank over nested iarrays and the dimensions of the elementary subspaces	88
18 Obtaining explicitly the invertible matrix which transforms a matrix to its reduced row echelon form over nested iarrays	89
18.1 Definitions	89
18.2 Proofs	90
18.2.1 Properties of <i>Gauss-Jordan-in-ij-iarrays-PA</i>	90
18.2.2 Properties about <i>Gauss-Jordan-column-k-iarrays-PA</i>	90

18.2.3	Properties about <i>Gauss-Jordan-upt-k-iarrays-PA</i> . . .	91
18.2.4	Properties about <i>Gauss-Jordan-iarrays-PA</i>	91
19	Bases of the four fundamental subspaces over IArrays	92
19.1	Computation of bases of the fundamental subspaces using IArrays	92
19.2	Code equations	93
20	Solving systems of equations using the Gauss Jordan algo- rithm over nested IArrays	94
20.1	Previous definitions and properties	94
20.2	Consistency and inconsistency	95
20.3	Independence and dependence	96
20.4	Solve a system of equations over nested IArrays	96
21	Computing determinants of matrices using the Gauss Jor- dan algorithm over nested IArrays	97
21.1	Definitions	97
21.2	Proofs	98
21.3	Code equations	99
22	Inverse of a matrix using the Gauss Jordan algorithm over nested IArrays	100
22.1	Definitions	100
22.2	Some lemmas and code generation	100
23	Examples of computations over matrices represented as nested IArrays	101
23.1	Transformations between nested lists nested IArrays	101
23.2	Examples	102
23.2.1	Ranks, dimensions and Gauss Jordan algorithm	102
23.2.2	Inverse of a matrix	103
23.2.3	Determinant of a matrix	103
23.2.4	Bases of the fundamental subspaces	103
23.2.5	Consistency and inconsistency	105
23.2.6	Solving systems of linear equations	105
24	Exporting code to SML and Haskell	106
24.1	Exporting code	106
24.2	The Mathematica bug	107
25	Exporting code to SML	108
26	Code Generation for rational numbers in Haskell	110
26.1	Serializations	110

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* $\langle \text{proof} \rangle$

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

shows $\forall m. \text{is-zero-row-upt-k } m \ 0 \ A \ \langle \text{proof} \rangle$

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*
 $\langle \text{proof} \rangle$

lemma *is-zero-row-imp-is-zero-row-upt*:

assumes *is-zero-row i A*

shows *is-zero-row-upt-k i k A*
 $\langle \text{proof} \rangle$

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* :: $'a::\{\text{zero, one}\}^m::\{\text{mod-type}\}^n::\{\text{finite, ord, plus, one}\} \Rightarrow \text{nat} \Rightarrow \text{bool}$

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

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

$\langle \text{proof} \rangle$

lemma *rref-upt-condition1*:

assumes *r: reduced-row-echelon-form-upt-k A k*

shows $(\forall i. \text{is-zero-row-upt-k } i \ k \ A \longrightarrow \neg (\exists j. j > i \wedge \neg \text{is-zero-row-upt-k } j \ k \ A))$
 $\langle \text{proof} \rangle$

lemma *rref-upt-condition2*:

assumes *r: reduced-row-echelon-form-upt-k A k*

shows $(\forall i. \neg (\text{is-zero-row-upt-k } i \ k \ A) \longrightarrow A \ \$ \ i \ \$ \ (\text{LEAST } k. A \ \$ \ i \ \$ \ k \neq 0) = 1)$
 $\langle \text{proof} \rangle$

lemma *rref-upt-condition3*:

assumes *r: reduced-row-echelon-form-upt-k A k*

shows $(\forall i. i < i+1 \wedge \neg (\text{is-zero-row-upt-k } i \ k \ A) \wedge \neg (\text{is-zero-row-upt-k } (i+1) \ k \ A))$

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

lemma *rref-upt-condition4*:

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

<proof>

lemma *rref-suc-imp-rref*:

fixes $A::'a::\{\text{semiring-1}\} \sim n::\{\text{mod-type}\} \sim m::\{\text{mod-type}\}$

assumes r : *reduced-row-echelon-form-upt-k A (Suc k)*

and k -le-card: $\text{Suc } k < \text{ncols } A$

shows *reduced-row-echelon-form-upt-k A k*

<proof>

lemma *reduced-row-echelon-if-all-zero*:

assumes all-zero : $\forall n. \text{is-zero-row-upt-k } n \ k \ A$

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* :: $'a::\{\text{zero, one}\} \sim m::\{\text{mod-type}\} \sim n::\{\text{finite, ord, plus, one}\} \Rightarrow \text{bool}$

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 =

(
 $(\forall i. \text{is-zero-row } i \ A \longrightarrow \neg (\exists j. j > i \wedge \neg \text{is-zero-row } j \ A)) \wedge$
 $(\forall i. \neg (\text{is-zero-row } i \ A) \longrightarrow A \ \$ \ i \ \$ \ (\text{LEAST } k. A \ \$ \ i \ \$ \ k \neq 0) = 1) \wedge$
 $(\forall i. i < i+1 \wedge \neg (\text{is-zero-row } i \ A) \wedge \neg (\text{is-zero-row } (i+1) \ A) \longrightarrow ((\text{LEAST } k. A \ \$ \ i \ \$ \ k \neq 0) < (\text{LEAST } k. A \ \$ \ (i+1) \ \$ \ k \neq 0))) \wedge$
 $(\forall i. \neg (\text{is-zero-row } i \ A) \longrightarrow (\forall j. i \neq j \longrightarrow A \ \$ \ j \ \$ \ (\text{LEAST } k. A \ \$ \ i \ \$ \ k \neq 0) = 0))$
) *<proof>*

1.2 Properties of the reduced row echelon form of a matrix

lemma *rref-condition1*:

assumes r : *reduced-row-echelon-form A*

shows $(\forall i. \text{is-zero-row } i \ A \longrightarrow \neg (\exists j. j > i \wedge \neg \text{is-zero-row } j \ A))$ *<proof>*

lemma *rref-condition2*:

assumes r : *reduced-row-echelon-form A*

shows $(\forall i. \neg (\text{is-zero-row } i \ A) \longrightarrow A \ \$ \ i \ \$ \ (\text{LEAST } k. A \ \$ \ i \ \$ \ k \neq 0) = 1)$

<proof>

lemma *rref-condition3*:

assumes r : *reduced-row-echelon-form A*

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

lemma *rref-condition4*:

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

lemma *rref-condition2-explicit*:

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

lemma *rref-condition3-explicit*:

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

lemma *rref-condition4-explicit*:

assumes *rref-A*: *reduced-row-echelon-form A*
and $\neg \text{is-zero-row } i \ A$
and $i \neq j$
shows $A \ \$ \ j \ \$ \ (\text{LEAST } n. A \ \$ \ i \ \$ \ n \neq 0) = 0$
 ⟨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*: $j < \text{nrows } A$
and *not-zero-i*: $\neg \text{is-zero-row } (\text{from-nat } i) \ A$
and *not-zero-j*: $\neg \text{is-zero-row } (\text{from-nat } j) \ A$
shows $(\text{LEAST } n. A \ \$ \ (\text{from-nat } i) \ \$ \ n \neq 0) < (\text{LEAST } n. A \ \$ \ (\text{from-nat } j) \ \$ \ n \neq 0)$
 ⟨proof⟩

corollary *rref-condition3-equiv*:
fixes $A::'a::\{one, zero\} \wedge cols::\{mod-type\} \wedge rows::\{mod-type\}$
assumes *rref*: *reduced-row-echelon-form* A
and *i-less-j*: $i < j$
and $i: \neg is-zero-row\ i\ A$
and $j: \neg is-zero-row\ j\ A$
shows $(LEAST\ n.\ A\ \$\ i\ \$\ n\ \neq\ 0) < (LEAST\ n.\ A\ \$\ j\ \$\ n\ \neq\ 0)$
 $\langle proof \rangle$

lemma *rref-implies-rref-upt*:
fixes $A::'a::\{one, zero\} \wedge cols::\{mod-type\} \wedge rows::\{mod-type\}$
assumes *rref*: *reduced-row-echelon-form* A
shows *reduced-row-echelon-form-upt-k* $A\ k$
 $\langle proof \rangle$

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

lemma *rref-first-element*:
fixes $A::'a::\{one, zero\} \wedge cols::\{mod-type\} \wedge rows::\{mod-type\}$
assumes *rref*: *reduced-row-echelon-form* A
and *column-not-0*: *column 0* $A \neq 0$
shows $A\ \$\ 0\ \$\ 0 = 1$
 $\langle proof \rangle$

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 functions 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) = (%i. n) <proof>
lemma [code abstract]: vec-nth (a * b) = (%i. a$i * b$i) <proof>
lemma [code abstract]: vec-nth (c *s x) = (%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*]: $vec\text{-}nth (m ** m') = mat\text{-}mult\text{-}row\ m\ m'$
<proof>

lemma *matrix-vector-mult-code* [*code abstract*]:
 $vec\text{-}nth (A *v\ x) = (\%i. (\sum j \in UNIV. A\ \$\ i\ \$\ j * x\ \$\ j))$ *<proof>*

lemma *vector-matrix-mult-code* [*code abstract*]:
 $vec\text{-}nth (x\ v* A) = (\%j. (\sum i \in UNIV. A\ \$\ i\ \$\ j * x\ \$\ i))$ *<proof>*

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

lemma *mat-row-code* [*code abstract*]:
 $vec\text{-}nth (mat\text{-}row\ k\ i) = (\%j. \text{if } i = j \text{ then } k \text{ else } 0)$ *<proof>*

lemma [*code abstract*]: $vec\text{-}nth (mat\ k) = mat\text{-}row\ k$
<proof>

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

lemma *transpose-row-code* [*code abstract*]:
 $vec\text{-}nth (transpose\text{-}row\ A\ i) = (\%j. A\ \$\ j\ \$\ i)$ *<proof>*

lemma *transpose-code* [*code abstract*]:
 $vec\text{-}nth (transpose\ A) = transpose\text{-}row\ A$
<proof>

lemma [*code abstract*]: $vec\text{-}nth (row\ i\ A) = ((\$) (A\ \$\ i))$ *<proof>*

lemma [*code abstract*]: $vec\text{-}nth (column\ j\ A) = (\%i. A\ \$\ i\ \$\ j)$ *<proof>*

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

lemma *rowvector-row-code* [*code abstract*]:
 $vec\text{-}nth (rowvector\text{-}row\ v\ i) = (\%j. (v\ \$\ j))$ *<proof>*

lemma [*code abstract*]: $vec\text{-}nth (rowvector\ v) = rowvector\text{-}row\ v$
<proof>

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

lemma *columnvector-row-code* [*code abstract*]:
 $vec\text{-}nth (columnvector\text{-}row\ v\ i) = (\%j. (v\ \$\ i))$ *<proof>*

lemma [*code abstract*]: $vec\text{-}nth (columnvector\ v) = columnvector\text{-}row\ v$
<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*: $\text{mat } 1 \ \$ \ a \ \$ \ b = (\lambda i \ j. \text{if } i=j \text{ then } 1 \text{ else } 0) \ a \ b$ *<proof>*

lemma *mat1-sum-eq*:
shows $(\sum_{k \in UNIV} \text{mat } (1::'a::\{\text{semiring-1}\}) \ \$ \ s \ \$ \ k * \text{mat } 1 \ \$ \ k \ \$ \ t) = \text{mat } 1 \ \$ \ s \ \$ \ t$
<proof>

lemma *invertible-mat-n*:
fixes $n::'a::\{\text{field}\}$
assumes $n: n \neq 0$
shows *invertible* $((\text{mat } n)::'a \ \wedge \ n \ \wedge \ n)$
<proof>

corollary *invertible-mat-1*:
shows *invertible* $(\text{mat } (1::'a::\{\text{field}\}))$ *<proof>*

4.2 Definitions of elementary row and column operations

Definitions of elementary row operations

definition *interchange-rows* :: $'a \ \wedge \ n \ \wedge \ m \Rightarrow 'm \Rightarrow 'm \Rightarrow 'a \ \wedge \ n \ \wedge \ m$
where *interchange-rows* $A \ a \ b = (\chi \ i \ j. \text{if } i=a \text{ then } A \ \$ \ b \ \$ \ j \text{ else if } i=b \text{ then } A \ \$ \ a \ \$ \ j \text{ else } A \ \$ \ i \ \$ \ j)$

definition *mult-row* :: $('a::\text{times}) \ \wedge \ n \ \wedge \ m \Rightarrow 'm \Rightarrow 'a \Rightarrow 'a \ \wedge \ n \ \wedge \ m$
where *mult-row* $A \ a \ q = (\chi \ i \ j. \text{if } i=a \text{ then } q*(A \ \$ \ a \ \$ \ j) \text{ else } A \ \$ \ i \ \$ \ j)$

definition *row-add* :: $('a::\{\text{plus}, \text{times}\}) \ \wedge \ n \ \wedge \ m \Rightarrow 'm \Rightarrow 'm \Rightarrow 'a \Rightarrow 'a \ \wedge \ n \ \wedge \ m$
where *row-add* $A \ a \ b \ q = (\chi \ i \ j. \text{if } i=a \text{ then } (A \ \$ \ a \ \$ \ j) + q*(A \ \$ \ b \ \$ \ j) \text{ else } A \ \$ \ i \ \$ \ j)$

Definitions of elementary column operations

definition *interchange-columns* :: $'a \ \wedge \ n \ \wedge \ m \Rightarrow 'n \Rightarrow 'n \Rightarrow 'a \ \wedge \ n \ \wedge \ m$
where *interchange-columns* $A \ n \ m = (\chi \ i \ j. \text{if } j=n \text{ then } A \ \$ \ i \ \$ \ m \text{ else if } j=m \text{ then } A \ \$ \ i \ \$ \ n \text{ else } A \ \$ \ i \ \$ \ j)$

definition *mult-column* :: $('a::\text{times}) \ \wedge \ n \ \wedge \ m \Rightarrow 'n \Rightarrow 'a \Rightarrow 'a \ \wedge \ n \ \wedge \ m$
where *mult-column* $A \ n \ q = (\chi \ i \ j. \text{if } j=n \text{ then } (A \ \$ \ i \ \$ \ j)*q \text{ else } A \ \$ \ i \ \$ \ j)$

definition *column-add* :: ('a::{plus, times}) $\hat{~}n\hat{~}m \Rightarrow 'n \Rightarrow 'n \Rightarrow 'a \Rightarrow 'a$
 $\hat{~}n\hat{~}m$

where *column-add* $A\ n\ m\ q = (\chi\ i\ j. \text{if } j=n \text{ then } ((A\ \$\ i\ \$\ n) + (A\ \$\ i\ \$\ m)*q)$
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 $i \neq a$ **and** $j \neq a$
shows *interchange-rows* $A\ i\ j\ \$\ a = A\ \$\ a$
<proof>

lemma *interchange-rows-mat-1*:
shows *interchange-rows* $(\text{mat } 1)\ a\ b\ **\ A = \text{interchange-rows } A\ a\ b$
<proof>

lemma *invertible-interchange-rows*: *invertible* $(\text{interchange-rows } (\text{mat } 1)\ a\ b)$
<proof>

4.3.2 Properties about multiplying a row by a constant

Properties about *mult-row*

lemma *mult-row-mat-1*: *mult-row* $(\text{mat } 1)\ a\ q\ **\ A = \text{mult-row } A\ a\ q$
<proof>

lemma *invertible-mult-row*:
assumes $qk: q * k = 1$ **and** $kq: k*q=1$
shows *invertible* $(\text{mult-row } (\text{mat } 1)\ a\ q)$
<proof>

corollary *invertible-mult-row'*:
assumes $q\text{-not-zero}: q \neq 0$
shows *invertible* $(\text{mult-row } (\text{mat } (1::'a::\{\text{field}\}))\ a\ q)$
<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*: $\text{row-add } (\text{mat } 1) \ a \ b \ q \ ** \ A = \text{row-add } A \ a \ b \ q$
(*proof*)

lemma *invertible-row-add*:

assumes *a-not-eq-b*: $a \neq b$

shows *invertible* ($\text{row-add } (\text{mat } (1::'a::\{\text{ring-1}\})) \ a \ b \ q$)
(*proof*)

4.4 Properties about elementary column operations

4.4.1 Properties about interchanging columns

Properties about *interchange-columns*

lemma *interchange-columns-mat-1*: $A \ ** \ \text{interchange-columns } (\text{mat } 1) \ a \ b = \text{interchange-columns } A \ a \ b$
(*proof*)

lemma *invertible-interchange-columns*: *invertible* ($\text{interchange-columns } (\text{mat } 1) \ a \ b$)
(*proof*)

4.4.2 Properties about multiplying a column by a constant

Properties about *mult-column*

lemma *mult-column-mat-1*: $A \ ** \ \text{mult-column } (\text{mat } 1) \ a \ q = \text{mult-column } A \ a \ q$
(*proof*)

lemma *invertible-mult-column*:

assumes *qk*: $q * k = 1$ **and** *kq*: $k * q = 1$

shows *invertible* ($\text{mult-column } (\text{mat } 1) \ a \ q$)
(*proof*)

corollary *invertible-mult-column'*:

assumes *q-not-zero*: $q \neq 0$

shows *invertible* ($\text{mult-column } (\text{mat } (1::'a::\{\text{field}\})) \ a \ q$)
(*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*: $A \ ** \ \text{column-add } (\text{mat } 1) \ a \ b \ q = \text{column-add } A \ a \ b \ q$
(*proof*)

lemma *invertible-column-add*:
assumes *a-not-eq-b*: $a \neq b$
shows *invertible* (*column-add* (*mat* ($1::'a::\{\text{ring-1}\}$)) *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* $?A \ ?a \ ?b = (\chi i \ j. \text{if } i = ?a \ \text{then } ?A \ \$ \ ?b \ \$ \ j \ \text{else if } i = ?b \ \text{then } ?A \ \$ \ ?a \ \$ \ j \ \text{else } ?A \ \$ \ i \ \$ \ j)$, *interchange-columns* $?A \ ?n \ ?m = (\chi i \ j. \text{if } j = ?n \ \text{then } ?A \ \$ \ i \ \$ \ ?m \ \text{else if } j = ?m \ \text{then } ?A \ \$ \ i \ \$ \ ?n \ \text{else } ?A \ \$ \ i \ \$ \ j)$, *row-add* $?A \ ?a \ ?b \ ?q = (\chi i \ j. \text{if } i = ?a \ \text{then } ?A \ \$ \ ?a \ \$ \ j + ?q * ?A \ \$ \ ?b \ \$ \ j \ \text{else } ?A \ \$ \ i \ \$ \ j)$, *column-add* $?A \ ?n \ ?m \ ?q = (\chi i \ j. \text{if } j = ?n \ \text{then } ?A \ \$ \ i \ \$ \ ?n + ?A \ \$ \ i \ \$ \ ?m * ?q \ \text{else } ?A \ \$ \ i \ \$ \ j)$, *mult-row* $?A \ ?a \ ?q = (\chi i \ j. \text{if } i = ?a \ \text{then } ?q * ?A \ \$ \ ?a \ \$ \ j \ \text{else } ?A \ \$ \ i \ \$ \ j)$ and *mult-column* $?A \ ?n \ ?q = (\chi i \ j. \text{if } j = ?n \ \text{then } ?A \ \$ \ i \ \$ \ j * ?q \ \text{else } ?A \ \$ \ i \ \$ \ j)$:

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

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

<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]:
 $vec\text{-}nth (mult\text{-}row\text{-}row A a q i) = (\%j. \text{if } i = a \text{ then } q * A \$ a \$ j \text{ else } A \$ i \$ j)$
 ⟨proof⟩

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

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

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

lemma *mult-column-code-nth* [code abstract]: $vec\text{-}nth (mult\text{-}column A a q) = mult\text{-}column\text{-}row A a q$
 ⟨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* :: $'a::\{field\} \wedge n \wedge m \Rightarrow nat$
where $row\text{-}rank A = vec.\text{dim} (row\text{-}space A)$

definition *col-rank* :: $'a::\{field\} \wedge n \wedge m \Rightarrow nat$
where $col\text{-}rank A = vec.\text{dim} (col\text{-}space A)$

lemma *rank-def*: $rank A = row\text{-}rank A$
 ⟨proof⟩

5.2 Properties

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

```

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
  Ref
  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 ≠ 0
and i ≤ 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 = nrow 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)) = nrow 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 $A::'a::\{one, zero\}^{\wedge}n::\{mod\text{-}type\}^{\wedge}m::\{finite, one, plus, linorder\}$
assumes $r: reduced\text{-}row\text{-}echelon\text{-}form\text{-}upt\text{-}k\ A\ k$
and $zero\text{-}i: is\text{-}zero\text{-}row\text{-}upt\text{-}k\ i\ k\ A$
and $not\text{-}all\text{-}zero: \neg (\forall a. is\text{-}zero\text{-}row\text{-}upt\text{-}k\ a\ k\ A)$
shows $(GREATEST\ m. \neg is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A) < i$
 $\langle proof \rangle$

lemma *rref-suc-if-zero-below-greatest*:

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

lemma *rref-suc-if-all-rows-not-zero*:

fixes $A::'a::\{one, zero\}^{\wedge}n::\{mod\text{-}type\}^{\wedge}m::\{finite, one, plus, linorder\}$
assumes $r: reduced\text{-}row\text{-}echelon\text{-}form\text{-}upt\text{-}k\ A\ k$
and $all\text{-}not\text{-}zero: \forall n. \neg is\text{-}zero\text{-}row\text{-}upt\text{-}k\ n\ k\ A$
shows $reduced\text{-}row\text{-}echelon\text{-}form\text{-}upt\text{-}k\ A\ (Suc\ k)$
 $\langle proof \rangle$

lemma *greatest-ge-nonzero-row*:

fixes $A::'a::\{zero\}^{\wedge}n::\{mod\text{-}type\}^{\wedge}m::\{finite, linorder\}$
assumes $\neg is\text{-}zero\text{-}row\text{-}upt\text{-}k\ i\ k\ A$
shows $i \leq (GREATEST\ m. \neg is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A)$ $\langle proof \rangle$

lemma *greatest-ge-nonzero-row'*:

fixes $A::'a::\{zero, one\}^{\wedge}n::\{mod\text{-}type\}^{\wedge}m::\{finite, linorder, one, plus\}$
assumes $r: reduced\text{-}row\text{-}echelon\text{-}form\text{-}upt\text{-}k\ A\ k$
and $i: i \leq (GREATEST\ m. \neg is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A)$
and $not\text{-}all\text{-}zero: \neg (\forall a. is\text{-}zero\text{-}row\text{-}upt\text{-}k\ a\ k\ A)$
shows $\neg is\text{-}zero\text{-}row\text{-}upt\text{-}k\ i\ k\ A$
 $\langle proof \rangle$

corollary *row-greater-greatest-is-zero*:

fixes $A::'a::\{zero\}^{\wedge}n::\{mod\text{-}type\}^{\wedge}m::\{finite, linorder\}$
assumes $(GREATEST\ m. \neg is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ k\ A) < i$
shows $is\text{-}zero\text{-}row\text{-}upt\text{-}k\ i\ k\ A$ $\langle proof \rangle$

6.3 The proof of its correctness

Properties of *Gauss-Jordan-in-ij*

lemma *Gauss-Jordan-in-ij-1*:

fixes $A::'a::\{field\}^{\wedge}m^{\wedge}n::\{finite, ord, wellorder\}$
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 $A::'a::\{\text{field}\}^m^n::\{\text{finite, ord, wellorder}\}$
assumes $ex: \exists n. A \$ n \$ j \neq 0 \wedge i \leq n$ **and** $a: a \neq i$
shows (*Gauss-Jordan-in-ij* A i j) $\$ a \$ j = 0$
 ⟨*proof*⟩

corollary *Gauss-Jordan-in-ij-0'*:
fixes $A::'a::\{\text{field}\}^m^n::\{\text{finite, ord, wellorder}\}$
assumes $ex: \exists n. A \$ n \$ j \neq 0 \wedge i \leq n$
shows $\forall a. a \neq i \longrightarrow$ (*Gauss-Jordan-in-ij* A i j) $\$ a \$ j = 0$ ⟨*proof*⟩

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

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

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

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

fixes $A::'a::\{\text{field}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$ **and** $k::\text{nat}$
defines $ia:ia\equiv(\text{if } \forall m. \text{is-zero-row-upt-}k\ m\ k\ A \text{ then } 0 \text{ else to-nat } (GREATEST$
 $n. \neg \text{is-zero-row-upt-}k\ n\ k\ A) + 1)$
defines $B:B\equiv(\text{snd } (Gauss-Jordan-column-k\ (ia,A)\ k))$
assumes $\text{all-zero}: \forall n. \text{is-zero-row-upt-}k\ n\ k\ A$
and $\text{not-zero-}i: \neg \text{is-zero-row-upt-}k\ i\ (Suc\ k)\ B$
and $\text{Amk-zero}: A\ \$\ m\ \$\ \text{from-nat}\ k\ \neq\ 0$
shows $i=0$
 $\langle\text{proof}\rangle$

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 $A::'a::\{\text{field}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$ **and** $k::\text{nat}$
assumes $\text{zero-column-}k: \forall m \geq \text{from-nat } 0. A\ \$\ m\ \$\ \text{from-nat}\ k = 0$
and $\text{all-zero}: \forall m. \text{is-zero-row-upt-}k\ m\ k\ A$
shows $\text{is-zero-row-upt-}k\ j\ (Suc\ k)\ A$
 $\langle\text{proof}\rangle$

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

lemma condition-1-part-3:
fixes $A::'a::\{\text{field}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$ **and** $k::\text{nat}$
defines $ia:ia\equiv(\text{if } \forall m. \text{is-zero-row-upt-}k\ m\ k\ A \text{ then } 0 \text{ else to-nat } (GREATEST$
 $n. \neg \text{is-zero-row-upt-}k\ n\ k\ A) + 1)$
defines $B:B\equiv(\text{snd } (Gauss-Jordan-column-k\ (ia,A)\ k))$
assumes $\text{rref}: \text{reduced-row-echelon-form-upt-}k\ A\ k$
and $\text{i-less-j}: i < j$
and $\text{not-zero-}m: \neg \text{is-zero-row-upt-}k\ m\ k\ A$
and $\text{zero-below-greatest}: \forall m \geq (GREATEST\ n. \neg \text{is-zero-row-upt-}k\ n\ k\ A) + 1.$
 $A\ \$\ m\ \$\ \text{from-nat}\ k = 0$
and $\text{zero-i-suc-}k: \text{is-zero-row-upt-}k\ i\ (Suc\ k)\ B$
shows $\text{is-zero-row-upt-}k\ j\ (Suc\ k)\ A$
 $\langle\text{proof}\rangle$

lemma condition-1-part-4:
fixes $A::'a::\{\text{field}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$ **and** $k::\text{nat}$
defines $ia:ia\equiv(\text{if } \forall m. \text{is-zero-row-upt-}k\ m\ k\ A \text{ then } 0 \text{ else to-nat } (GREATEST$
 $n. \neg \text{is-zero-row-upt-}k\ n\ k\ A) + 1)$
defines $B:B\equiv(\text{snd } (Gauss-Jordan-column-k\ (ia,A)\ k))$
assumes $\text{rref}: \text{reduced-row-echelon-form-upt-}k\ A\ k$
and $\text{zero-i-suc-}k: \text{is-zero-row-upt-}k\ i\ (Suc\ k)\ B$

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

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

lemma condition-1:
fixes $A::'a::\{\text{field}\}^{\sim} \text{columns}::\{\text{mod-type}\}^{\sim} \text{rows}::\{\text{mod-type}\}$ **and** $k::nat$
defines $ia:ia \equiv (\text{if } \forall m. \text{is-zero-row-upt-}k\ m\ k\ A \text{ then } 0 \text{ else } \text{to-nat } (\text{GREATEST } n. \neg \text{is-zero-row-upt-}k\ n\ k\ A) + 1)$
defines $B:B \equiv (\text{snd } (\text{Gauss-Jordan-column-}k\ (ia,A)\ k))$
assumes *rref*: *reduced-row-echelon-form-upt-}k\ A\ k*
and *zero-i-suc-}k*: *is-zero-row-upt-}k\ i\ (\text{Suc } k)\ B* **and** *i-less-j*: $i < j$
shows *is-zero-row-upt-}k\ j\ (\text{Suc } k)\ B*
 $\langle proof \rangle$

lemma condition-2-part-1:
fixes $A::'a::\{\text{field}\}^{\sim} \text{columns}::\{\text{mod-type}\}^{\sim} \text{rows}::\{\text{mod-type}\}$ **and** $k::nat$
defines $ia:ia \equiv (\text{if } \forall m. \text{is-zero-row-upt-}k\ m\ k\ A \text{ then } 0 \text{ else } \text{to-nat } (\text{GREATEST } n. \neg \text{is-zero-row-upt-}k\ n\ k\ A) + 1)$
defines $B:B \equiv (\text{snd } (\text{Gauss-Jordan-column-}k\ (ia,A)\ k))$
assumes *not-zero-i-suc-}k*: $\neg \text{is-zero-row-upt-}k\ i\ (\text{Suc } k)\ B$
and *all-zero*: $\forall m. \text{is-zero-row-upt-}k\ m\ k\ A$
and *all-zero-k*: $\forall m. A\ \$\ m\ \$\ \text{from-nat } k = 0$
shows $A\ \$\ i\ \$\ (\text{LEAST } k. A\ \$\ i\ \$\ k \neq 0) = 1$
 $\langle proof \rangle$

lemma condition-2-part-2:
fixes $A::'a::\{\text{field}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$ **and** $k::\text{nat}$
defines $ia:ia\equiv(\text{if } \forall m. \text{is-zero-row-upt-}k\ m\ k\ A \text{ then } 0 \text{ else to-nat } (GREATEST\ n. \neg \text{is-zero-row-upt-}k\ n\ k\ A) + 1)$
defines $B:B\equiv(\text{snd } (Gauss-Jordan-column-k\ (ia,A)\ k))$
assumes $\text{not-zero-i-suc-}k: \neg \text{is-zero-row-upt-}k\ i\ (Suc\ k)\ B$
and $\text{all-zero}: \forall m. \text{is-zero-row-upt-}k\ m\ k\ A$
and $\text{Amk-not-zero}: A\ \$\ m\ \$\ \text{from-nat}\ k \neq 0$
shows $Gauss-Jordan-in-ij\ A\ 0\ (\text{from-nat}\ k)\ \$\ i\ \$\ (LEAST\ ka. Gauss-Jordan-in-ij\ A\ 0\ (\text{from-nat}\ k)\ \$\ i\ \$\ ka \neq 0) = 1$
 $\langle\text{proof}\rangle$

lemma condition-2-part-3:
fixes $A::'a::\{\text{field}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$ **and** $k::\text{nat}$
defines $ia:ia\equiv(\text{if } \forall m. \text{is-zero-row-upt-}k\ m\ k\ A \text{ then } 0 \text{ else to-nat } (GREATEST\ n. \neg \text{is-zero-row-upt-}k\ n\ k\ A) + 1)$
defines $B:B\equiv(\text{snd } (Gauss-Jordan-column-k\ (ia,A)\ k))$
assumes $\text{rref}: \text{reduced-row-echelon-form-upt-}k\ A\ k$
and $\text{not-zero-i-suc-}k: \neg \text{is-zero-row-upt-}k\ i\ (Suc\ k)\ B$
and $\text{not-zero-m}: \neg \text{is-zero-row-upt-}k\ m\ k\ A$
and $\text{zero-below-greatest}: \forall m \geq (GREATEST\ n. \neg \text{is-zero-row-upt-}k\ n\ k\ A) + 1. A\ \$\ m\ \$\ \text{from-nat}\ k = 0$
shows $A\ \$\ i\ \$\ (LEAST\ k. A\ \$\ i\ \$\ k \neq 0) = 1$
 $\langle\text{proof}\rangle$

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

lemma condition-2-part-5:
fixes $A::'a::\{\text{field}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$ **and** $k::\text{nat}$
defines $ia:ia\equiv(\text{if } \forall m. \text{is-zero-row-upt-}k\ m\ k\ A \text{ then } 0 \text{ else to-nat } (GREATEST\ n. \neg \text{is-zero-row-upt-}k\ n\ k\ A) + 1)$
defines $B:B\equiv(\text{snd } (Gauss-Jordan-column-k\ (ia,A)\ k))$
assumes $\text{rref}: \text{reduced-row-echelon-form-upt-}k\ A\ k$
and $\text{not-zero-i-suc-}k: \neg \text{is-zero-row-upt-}k\ i\ (Suc\ k)\ B$
and $\text{not-zero-m}: \neg \text{is-zero-row-upt-}k\ m\ k\ A$
and $\text{greatest-not-eq-card}: Suc\ (\text{to-nat } (GREATEST\ n. \neg \text{is-zero-row-upt-}k\ n\ k\ A)) \neq \text{nrows } A$

and *greatest-less-ma*: $(\text{GREATEST } n. \neg \text{is-zero-row-upt-}k \ n \ k \ A) + 1 \leq ma$
and *A-ma-k-not-zero*: $A \ \$ \ ma \ \$ \ \text{from-nat } k \neq 0$
shows *Gauss-Jordan-in-ij A* $((\text{GREATEST } n. \neg \text{is-zero-row-upt-}k \ n \ k \ A) + 1)$
 $(\text{from-nat } k) \ \$ \ i \ \$$
 $(\text{LEAST } ka. \ \text{Gauss-Jordan-in-ij } A \ ((\text{GREATEST } n. \neg \text{is-zero-row-upt-}k \ n \ k \ A)$
 $+ 1) (\text{from-nat } k) \ \$ \ i \ \$ \ ka \neq 0) = 1$
 $\langle \text{proof} \rangle$

lemma condition-2:

fixes $A::'a::\{\text{field}\} \wedge \text{columns}::\{\text{mod-type}\} \wedge \text{rows}::\{\text{mod-type}\}$ **and** $k::\text{nat}$
defines $ia:ia \equiv (\text{if } \forall m. \text{is-zero-row-upt-}k \ m \ k \ A \ \text{then } 0 \ \text{else } \text{to-nat } (\text{GREATEST}$
 $n. \neg \text{is-zero-row-upt-}k \ n \ k \ A) + 1)$
defines $B:B \equiv (\text{snd } (\text{Gauss-Jordan-column-}k \ (ia, A) \ k))$
assumes *rref*: *reduced-row-echelon-form-upt-k A k*
and *not-zero-i-suc-k*: $\neg \text{is-zero-row-upt-}k \ i \ (\text{Suc } k) \ B$
shows $B \ \$ \ i \ \$ \ (\text{LEAST } k. B \ \$ \ i \ \$ \ k \neq 0) = 1$
 $\langle \text{proof} \rangle$

lemma condition-3-part-1:

fixes $A::'a::\{\text{field}\} \wedge \text{columns}::\{\text{mod-type}\} \wedge \text{rows}::\{\text{mod-type}\}$ **and** $k::\text{nat}$
defines $ia:ia \equiv (\text{if } \forall m. \text{is-zero-row-upt-}k \ m \ k \ A \ \text{then } 0 \ \text{else } \text{to-nat } (\text{GREATEST}$
 $n. \neg \text{is-zero-row-upt-}k \ n \ k \ A) + 1)$
defines $B:B \equiv (\text{snd } (\text{Gauss-Jordan-column-}k \ (ia, A) \ k))$
assumes *not-zero-i-suc-k*: $\neg \text{is-zero-row-upt-}k \ i \ (\text{Suc } k) \ B$
and *all-zero*: $\forall m. \text{is-zero-row-upt-}k \ m \ k \ A$
and *all-zero-k*: $\forall m. A \ \$ \ m \ \$ \ \text{from-nat } k = 0$
shows $(\text{LEAST } n. A \ \$ \ i \ \$ \ n \neq 0) < (\text{LEAST } n. A \ \$ \ (i + 1) \ \$ \ n \neq 0)$
 $\langle \text{proof} \rangle$

lemma condition-3-part-2:

fixes $A::'a::\{\text{field}\} \wedge \text{columns}::\{\text{mod-type}\} \wedge \text{rows}::\{\text{mod-type}\}$ **and** $k::\text{nat}$
defines $ia:ia \equiv (\text{if } \forall m. \text{is-zero-row-upt-}k \ m \ k \ A \ \text{then } 0 \ \text{else } \text{to-nat } (\text{GREATEST}$
 $n. \neg \text{is-zero-row-upt-}k \ n \ k \ A) + 1)$
defines $B:B \equiv (\text{snd } (\text{Gauss-Jordan-column-}k \ (ia, A) \ k))$
assumes *i-le*: $i < i + 1$
and *not-zero-i-suc-k*: $\neg \text{is-zero-row-upt-}k \ i \ (\text{Suc } k) \ B$
and *not-zero-suc-i-suc-k*: $\neg \text{is-zero-row-upt-}k \ (i + 1) \ (\text{Suc } k) \ B$
and *all-zero*: $\forall m. \text{is-zero-row-upt-}k \ m \ k \ A$
and *Amk-notzero*: $A \ \$ \ m \ \$ \ \text{from-nat } k \neq 0$
shows $(\text{LEAST } n. \ \text{Gauss-Jordan-in-ij } A \ 0 \ (\text{from-nat } k) \ \$ \ i \ \$ \ n \neq 0) < (\text{LEAST}$
 $n. \ \text{Gauss-Jordan-in-ij } A \ 0 \ (\text{from-nat } k) \ \$ \ (i + 1) \ \$ \ n \neq 0)$
 $\langle \text{proof} \rangle$

lemma condition-3-part-3:
fixes $A::'a::\{\text{field}\}^{\wedge'}\text{columns}::\{\text{mod-type}\}^{\wedge'}\text{rows}::\{\text{mod-type}\}$ **and** $k::\text{nat}$
defines $ia:ia\equiv(\text{if } \forall m. \text{is-zero-row-upt-}k\ m\ k\ A \text{ then } 0 \text{ else to-nat } (GREATEST\ n. \neg \text{is-zero-row-upt-}k\ n\ k\ A) + 1)$
defines $B:B\equiv(\text{snd } (Gauss-Jordan-column-}k\ (ia,A)\ k))$
assumes $rref: \text{reduced-row-echelon-form-upt-}k\ A\ k$
and $i-le: i < i + 1$
and $\text{not-zero-i-suc-}k: \neg \text{is-zero-row-upt-}k\ i\ (\text{Suc } k)\ B$
and $\text{not-zero-suc-i-suc-}k: \neg \text{is-zero-row-upt-}k\ (i + 1)\ (\text{Suc } k)\ B$
and $\text{not-zero-}m: \neg \text{is-zero-row-upt-}k\ m\ k\ A$
and $\text{zero-below-greatest: } \forall m \geq (GREATEST\ n. \neg \text{is-zero-row-upt-}k\ n\ k\ A) + 1. A\ \$\ m\ \$\ \text{from-nat } k = 0$
shows $(LEAST\ n. A\ \$\ i\ \$\ n \neq 0) < (LEAST\ n. A\ \$\ (i + 1)\ \$\ n \neq 0)$
 $\langle \text{proof} \rangle$

lemma condition-3-part-4:
fixes $A::'a::\{\text{field}\}^{\wedge'}\text{columns}::\{\text{mod-type}\}^{\wedge'}\text{rows}::\{\text{mod-type}\}$ **and** $k::\text{nat}$
defines $ia:ia\equiv(\text{if } \forall m. \text{is-zero-row-upt-}k\ m\ k\ A \text{ then } 0 \text{ else to-nat } (GREATEST\ n. \neg \text{is-zero-row-upt-}k\ n\ k\ A) + 1)$
defines $B:B\equiv(\text{snd } (Gauss-Jordan-column-}k\ (ia,A)\ k))$
assumes $rref: \text{reduced-row-echelon-form-upt-}k\ A\ k$ **and** $i-le: i < i + 1$
and $\text{not-zero-i-suc-}k: \neg \text{is-zero-row-upt-}k\ i\ (\text{Suc } k)\ B$
and $\text{not-zero-suc-i-suc-}k: \neg \text{is-zero-row-upt-}k\ (i + 1)\ (\text{Suc } k)\ B$
and $\text{not-zero-}m: \neg \text{is-zero-row-upt-}k\ m\ k\ A$
and $\text{greatest-eq-card: } \text{Suc } (\text{to-nat } (GREATEST\ n. \neg \text{is-zero-row-upt-}k\ n\ k\ A)) = \text{nrows } A$
shows $(LEAST\ n. A\ \$\ i\ \$\ n \neq 0) < (LEAST\ n. A\ \$\ (i + 1)\ \$\ n \neq 0)$
 $\langle \text{proof} \rangle$

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

<proof>

lemma condition-3:

fixes $A::'a::\{\text{field}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$ **and** $k::\text{nat}$
defines $ia:ia\equiv(\text{if } \forall m. \text{is-zero-row-upt-}k\ m\ k\ A \text{ then } 0 \text{ else to-nat } (\text{GREATEST } n. \neg \text{is-zero-row-upt-}k\ n\ k\ A) + 1)$
defines $B:B\equiv(\text{snd } (\text{Gauss-Jordan-column-}k\ (ia,A)\ k))$
assumes $rref: \text{reduced-row-echelon-form-upt-}k\ A\ k$
and $i-le: i < i + 1$
and $\text{not-zero-i-suc-}k: \neg \text{is-zero-row-upt-}k\ i\ (\text{Suc } k)\ B$
and $\text{not-zero-suc-i-suc-}k: \neg \text{is-zero-row-upt-}k\ (i + 1)\ (\text{Suc } k)\ B$
shows $(\text{LEAST } n. B\ \$\ i\ \$\ n \neq 0) < (\text{LEAST } n. B\ \$\ (i + 1)\ \$\ n \neq 0)$
<proof>

lemma condition-4-part-1:

fixes $A::'a::\{\text{field}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$ **and** $k::\text{nat}$
defines $ia:ia\equiv(\text{if } \forall m. \text{is-zero-row-upt-}k\ m\ k\ A \text{ then } 0 \text{ else to-nat } (\text{GREATEST } n. \neg \text{is-zero-row-upt-}k\ n\ k\ A) + 1)$
defines $B:B\equiv(\text{snd } (\text{Gauss-Jordan-column-}k\ (ia,A)\ k))$
assumes $\text{not-zero-i-suc-}k: \neg \text{is-zero-row-upt-}k\ i\ (\text{Suc } k)\ B$
and $\text{all-zero}: \forall m. \text{is-zero-row-upt-}k\ m\ k\ A$
and $\text{all-zero-}k: \forall m. A\ \$\ m\ \$\ \text{from-nat } k = 0$
shows $A\ \$\ j\ \$\ (\text{LEAST } n. A\ \$\ i\ \$\ n \neq 0) = 0$
<proof>

lemma condition-4-part-2:

fixes $A::'a::\{\text{field}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$ **and** $k::\text{nat}$
defines $ia:ia\equiv(\text{if } \forall m. \text{is-zero-row-upt-}k\ m\ k\ A \text{ then } 0 \text{ else to-nat } (\text{GREATEST } n. \neg \text{is-zero-row-upt-}k\ n\ k\ A) + 1)$
defines $B:B\equiv(\text{snd } (\text{Gauss-Jordan-column-}k\ (ia,A)\ k))$
assumes $\text{not-zero-i-suc-}k: \neg \text{is-zero-row-upt-}k\ i\ (\text{Suc } k)\ B$
and $i\text{-not-}j: i \neq j$
and $\text{all-zero}: \forall m. \text{is-zero-row-upt-}k\ m\ k\ A$
and $\text{Amk-not-zero}: A\ \$\ m\ \$\ \text{from-nat } k \neq 0$
shows $\text{Gauss-Jordan-in-ij } A\ 0\ (\text{from-nat } k)\ \$\ j\ \$\ (\text{LEAST } n. \text{Gauss-Jordan-in-ij } A\ 0\ (\text{from-nat } k)\ \$\ i\ \$\ n \neq 0) = 0$
<proof>

lemma condition-4-part-3:

fixes $A::'a::\{\text{field}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$ **and** $k::\text{nat}$
defines $ia:ia\equiv(\text{if } \forall m. \text{is-zero-row-upt-}k\ m\ k\ A \text{ then } 0 \text{ else to-nat } (\text{GREATEST } n. \neg \text{is-zero-row-upt-}k\ n\ k\ A) + 1)$
defines $B:B\equiv(\text{snd } (\text{Gauss-Jordan-column-}k\ (ia,A)\ k))$
assumes $rref: \text{reduced-row-echelon-form-upt-}k\ A\ k$
and $\text{not-zero-i-suc-}k: \neg \text{is-zero-row-upt-}k\ i\ (\text{Suc } k)\ B$
and $i\text{-not-}j: i \neq j$

and *not-zero-m*: $\neg \text{is-zero-row-upt-k } m \ k \ A$
and *zero-below-greatest*: $\forall m \geq (\text{GREATEST } n. \neg \text{is-zero-row-upt-k } n \ k \ A) + 1.$
 $A \ \$ \ m \ \$ \ \text{from-nat } k = 0$
shows $A \ \$ \ j \ \$ \ (\text{LEAST } n. A \ \$ \ i \ \$ \ n \neq 0) = 0$
<proof>

lemma condition-4-part-4:
fixes $A::'a::\{\text{field}\}^{\wedge} \text{columns}::\{\text{mod-type}\}^{\wedge} \text{rows}::\{\text{mod-type}\}$ **and** $k::\text{nat}$
defines $ia:ia \equiv (\text{if } \forall m. \text{is-zero-row-upt-k } m \ k \ A \ \text{then } 0 \ \text{else } \text{to-nat } (\text{GREATEST } n. \neg \text{is-zero-row-upt-k } n \ k \ A) + 1)$
defines $B:B \equiv (\text{snd } (\text{Gauss-Jordan-column-k } (ia, A) \ k))$
assumes *rref*: *reduced-row-echelon-form-upt-k* $A \ k$
and *not-zero-i-suc-k*: $\neg \text{is-zero-row-upt-k } i \ (\text{Suc } k) \ B$
and *i-not-j*: $i \neq j$
and *not-zero-m*: $\neg \text{is-zero-row-upt-k } m \ k \ A$
and *greatest-eq-card*: $\text{Suc } (\text{to-nat } (\text{GREATEST } n. \neg \text{is-zero-row-upt-k } n \ k \ A)) = \text{nrows } A$
shows $A \ \$ \ j \ \$ \ (\text{LEAST } n. A \ \$ \ i \ \$ \ n \neq 0) = 0$
<proof>

lemma condition-4-part-5:
fixes $A::'a::\{\text{field}\}^{\wedge} \text{columns}::\{\text{mod-type}\}^{\wedge} \text{rows}::\{\text{mod-type}\}$ **and** $k::\text{nat}$
defines $ia:ia \equiv (\text{if } \forall m. \text{is-zero-row-upt-k } m \ k \ A \ \text{then } 0 \ \text{else } \text{to-nat } (\text{GREATEST } n. \neg \text{is-zero-row-upt-k } n \ k \ A) + 1)$
defines $B:B \equiv (\text{snd } (\text{Gauss-Jordan-column-k } (ia, A) \ k))$
assumes *rref*: *reduced-row-echelon-form-upt-k* $A \ k$
and *not-zero-i-suc-k*: $\neg \text{is-zero-row-upt-k } i \ (\text{Suc } k) \ B$
and *i-not-j*: $i \neq j$
and *not-zero-m*: $\neg \text{is-zero-row-upt-k } m \ k \ A$
and *greatest-not-card*: $\text{Suc } (\text{to-nat } (\text{GREATEST } n. \neg \text{is-zero-row-upt-k } n \ k \ A)) \neq \text{nrows } A$
and *greatest-less-ma*: $(\text{GREATEST } n. \neg \text{is-zero-row-upt-k } n \ k \ A) + 1 \leq ma$
and *A-ma-k-not-zero*: $A \ \$ \ ma \ \$ \ \text{from-nat } k \neq 0$
shows $\text{Gauss-Jordan-in-ij } A \ ((\text{GREATEST } n. \neg \text{is-zero-row-upt-k } n \ k \ A) + 1) \ (\text{from-nat } k) \ \$ \ j \ \$ \ (\text{LEAST } n. \text{Gauss-Jordan-in-ij } A \ ((\text{GREATEST } n. \neg \text{is-zero-row-upt-k } n \ k \ A) + 1) \ (\text{from-nat } k) \ \$ \ i \ \$ \ n \neq 0) = 0$
<proof>

lemma condition-4:
fixes $A::'a::\{\text{field}\}^{\wedge} \text{columns}::\{\text{mod-type}\}^{\wedge} \text{rows}::\{\text{mod-type}\}$ **and** $k::\text{nat}$
defines $ia:ia \equiv (\text{if } \forall m. \text{is-zero-row-upt-k } m \ k \ A \ \text{then } 0 \ \text{else } \text{to-nat } (\text{GREATEST } n. \neg \text{is-zero-row-upt-k } n \ k \ A) + 1)$
defines $B:B \equiv (\text{snd } (\text{Gauss-Jordan-column-k } (ia, A) \ k))$
assumes *rref*: *reduced-row-echelon-form-upt-k* $A \ k$
and *not-zero-i-suc-k*: $\neg \text{is-zero-row-upt-k } i \ (\text{Suc } k) \ B$
and *i-not-j*: $i \neq j$
shows $B \ \$ \ j \ \$ \ (\text{LEAST } n. B \ \$ \ i \ \$ \ n \neq 0) = 0$

<proof>

lemma *reduced-row-echelon-form-upt-k-Gauss-Jordan-column-k:*

fixes $A::'a::\{\text{field}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$ **and** $k::\text{nat}$
defines $ia:ia\equiv(\text{if } \forall m. \text{is-zero-row-upt-k } m \ k \ A \ \text{then } 0 \ \text{else } \text{to-nat } (\text{GREATEST } n. \neg \text{is-zero-row-upt-k } n \ k \ A) + 1)$
defines $B:B\equiv(\text{snd } (\text{Gauss-Jordan-column-k } (ia,A) \ k))$
assumes *rref*: *reduced-row-echelon-form-upt-k* $A \ k$
shows *reduced-row-echelon-form-upt-k* $B \ (\text{Suc } k)$
<proof>

lemma *foldl-Gauss-condition-1:*

fixes $A::'a::\{\text{field}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$ **and** $k::\text{nat}$
assumes $\forall m. \text{is-zero-row-upt-k } m \ k \ A$
and $\forall m \geq 0. A \ \$ \ m \ \$ \ \text{from-nat } k = 0$
shows *is-zero-row-upt-k* $m \ (\text{Suc } k) \ A$
<proof>

lemma *foldl-Gauss-condition-2:*

fixes $A::'a::\{\text{field}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$ **and** $k::\text{nat}$
assumes $k: k < \text{ncols } A$
and *all-zero*: $\forall m. \text{is-zero-row-upt-k } m \ k \ A$
and *Amk-not-zero*: $A \ \$ \ m \ \$ \ \text{from-nat } k \neq 0$
shows $\exists m. \neg \text{is-zero-row-upt-k } m \ (\text{Suc } k) \ (\text{Gauss-Jordan-in-ij } A \ 0 \ (\text{from-nat } k))$
<proof>

lemma *foldl-Gauss-condition-3:*

fixes $A::'a::\{\text{field}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$ **and** $k::\text{nat}$
assumes $k: k < \text{ncols } A$
and *all-zero*: $\forall m. \text{is-zero-row-upt-k } m \ k \ A$
and *Amk-not-zero*: $A \ \$ \ m \ \$ \ \text{from-nat } k \neq 0$
and $\neg \text{is-zero-row-upt-k } m \ a \ (\text{Suc } k) \ (\text{Gauss-Jordan-in-ij } A \ 0 \ (\text{from-nat } k))$
shows $\text{to-nat } (\text{GREATEST } n. \neg \text{is-zero-row-upt-k } n \ (\text{Suc } k) \ (\text{Gauss-Jordan-in-ij } A \ 0 \ (\text{from-nat } k))) = 0$
<proof>

lemma *foldl-Gauss-condition-5:*

fixes $A::'a::\{\text{field}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$ **and** $k::\text{nat}$
assumes *rref-A*: *reduced-row-echelon-form-upt-k* $A \ k$
and *not-zero-a*: $\neg \text{is-zero-row-upt-k } a \ k \ A$
and *all-zero-below-greatest*: $\forall m \geq (\text{GREATEST } n. \neg \text{is-zero-row-upt-k } n \ k \ A) + 1. A \ \$ \ m \ \$ \ \text{from-nat } k = 0$
shows $(\text{GREATEST } n. \neg \text{is-zero-row-upt-k } n \ k \ A) = (\text{GREATEST } n. \neg \text{is-zero-row-upt-k } n \ (\text{Suc } k) \ A)$

<proof>

lemma *foldl-Gauss-condition-6:*

fixes $A::'a::\{\text{field}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$ **and** $k::\text{nat}$
assumes $\text{not-zero-}m: \neg \text{is-zero-row-upt-}k\ m\ k\ A$
and $\text{eq-card}: \text{Suc} (\text{to-nat} (\text{GREATEST } n. \neg \text{is-zero-row-upt-}k\ n\ k\ A)) = \text{nrows } A$
shows $\text{nrows } A = \text{Suc} (\text{to-nat} (\text{GREATEST } n. \neg \text{is-zero-row-upt-}k\ n\ (\text{Suc } k)\ A))$
<proof>

lemma *foldl-Gauss-condition-8:*

fixes $A::'a::\{\text{field}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$ **and** $k::\text{nat}$
assumes $k: k < \text{ncols } A$
and $\text{not-zero-}m: \neg \text{is-zero-row-upt-}k\ m\ k\ A$
and $A\text{-}ma\text{-}k: A\ \$\ ma\ \$\ \text{from-nat } k \neq 0$
and $ma: (\text{GREATEST } n. \neg \text{is-zero-row-upt-}k\ n\ k\ A) + 1 \leq ma$
shows $\exists m. \neg \text{is-zero-row-upt-}k\ m\ (\text{Suc } k)\ (\text{Gauss-Jordan-in-ij } A\ ((\text{GREATEST } n. \neg \text{is-zero-row-upt-}k\ n\ k\ A) + 1)\ (\text{from-nat } k))$
<proof>

lemma *foldl-Gauss-condition-9:*

fixes $A::'a::\{\text{field}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$ **and** $k::\text{nat}$
assumes $k: k < \text{ncols } A$
and $\text{rref-}A: \text{reduced-row-echelon-form-upt-}k\ A\ k$
assumes $\text{not-zero-}m: \neg \text{is-zero-row-upt-}k\ m\ k\ A$
and $\text{suc-greatest-not-card}: \text{Suc} (\text{to-nat} (\text{GREATEST } n. \neg \text{is-zero-row-upt-}k\ n\ k\ A)) \neq \text{nrows } A$
and $\text{greatest-less-}ma: (\text{GREATEST } n. \neg \text{is-zero-row-upt-}k\ n\ k\ A) + 1 \leq ma$
and $A\text{-}ma\text{-}k: A\ \$\ ma\ \$\ \text{from-nat } k \neq 0$
shows $\text{Suc} (\text{to-nat} (\text{GREATEST } n. \neg \text{is-zero-row-upt-}k\ n\ k\ A)) = \text{to-nat}(\text{GREATEST } n. \neg \text{is-zero-row-upt-}k\ n\ (\text{Suc } k)\ (\text{Gauss-Jordan-in-ij } A\ ((\text{GREATEST } n. \neg \text{is-zero-row-upt-}k\ n\ k\ A) + 1)\ (\text{from-nat } k)))$
<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 $\text{Gauss-Jordan-upt-}k\ ?A\ ?k = \text{snd} (\text{foldl } \text{Gauss-Jordan-column-}k\ (0, ?A)\ [0..<\text{Suc } ?k])$ (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 $A::'a::\{\text{field}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$ **and** $k::\text{nat}$
assumes $k < \text{ncols } A$
shows $\text{rref-Gauss-Jordan-upt-}k: \text{reduced-row-echelon-form-upt-}k\ (\text{Gauss-Jordan-upt-}k\ A)$

$A\ k\ (Suc\ k)$
and *snd-Gauss-Jordan-upt-k*:
 $foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k\ (0,\ A)\ [0..\lt Suc\ k] =$
 $(if\ \forall m.\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ m\ (Suc\ k)\ (snd\ (foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k\ (0,\ A)$
 $[0..\lt Suc\ k]))\ then\ 0$
 $else\ to\text{-}nat\ (GREATEST\ n.\ \neg\ is\text{-}zero\text{-}row\text{-}upt\text{-}k\ n\ (Suc\ k)\ (snd\ (foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k$
 $(0,\ A)\ [0..\lt Suc\ k])))\ +\ 1,$
 $snd\ (foldl\ Gauss\text{-}Jordan\text{-}column\text{-}k\ (0,\ A)\ [0..\lt Suc\ k]))$
 $\langle proof \rangle$

corollary *rref-Gauss-Jordan*:
fixes $A::'a::\{field\}\ \wedge\ columns::\{mod\text{-}type\}\ \wedge\ rows::\{mod\text{-}type\}$
shows *reduced-row-echelon-form* (*Gauss-Jordan* A)
 $\langle proof \rangle$

lemma *independent-not-zero-rows-rref*:
fixes $A::'a::\{field\}\ \wedge\ m::\{mod\text{-}type\}\ \wedge\ n::\{finite,\ one,\ plus,\ ord\}$
assumes *rref-A*: *reduced-row-echelon-form* A
shows *vec.independent* $\{row\ i\ A\ | i.\ row\ i\ A\ \neq\ 0\}$
 $\langle proof \rangle$

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

lemma *invertible-row-add-iterate*:
fixes $A::'a::\{ring\text{-}1\}\ \wedge\ n\ \wedge\ m::\{mod\text{-}type\}$
assumes $n < n\ rows\ A$
shows $\exists P.\ invertible\ P \wedge row\text{-}add\text{-}iterate\ A\ n\ i\ j = P**A$
 $\langle proof \rangle$

lemma *row-add-iterate-preserves-greater-than-n*:
fixes $A::'a::\{ring\text{-}1\}\ \wedge\ n\ \wedge\ m::\{mod\text{-}type\}$
assumes $n < n\ rows\ A$

and a : $\text{to-nat } a > n$
shows $(\text{row-add-iterate } A \ n \ i \ j) \ \$ \ a \ \$ \ b = A \ \$ \ a \ \$ \ b$
 $\langle \text{proof} \rangle$

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

lemma *row-add-iterate-eq-row-add*:
fixes $A::'a::\{\text{ring-1}\}^{\wedge n}{}^{\wedge m}::\{\text{mod-type}\}$
assumes $a\text{-not-}i$: $a \neq i$
and n : $n < \text{nrows } A$
and $\text{to-nat } a \leq n$
shows $(\text{row-add-iterate } A \ n \ i \ j) \ \$ \ a \ \$ \ b = (\text{row-add } A \ a \ i \ (- \ A \ \$ \ a \ \$ \ j)) \ \$ \ a \ \$ \ b$
 $\langle \text{proof} \rangle$

lemma *row-add-iterate-eq-Gauss-Jordan-in-ij*:
fixes $A::'a::\{\text{field}\}^{\wedge n}{}^{\wedge m}::\{\text{mod-type}\}$ **and** $i::'m$ **and** $j::'n$
defines A' : $A' = \text{mult-row } (\text{interchange-rows } A \ i \ (\text{LEAST } n. A \ \$ \ n \ \$ \ j \neq 0 \wedge i \leq n)) \ i \ (1 / (\text{interchange-rows } A \ i \ (\text{LEAST } n. A \ \$ \ n \ \$ \ j \neq 0 \wedge i \leq n)) \ \$ \ i \ \$ \ j)$
shows $\text{row-add-iterate } A' \ (\text{nrows } A - 1) \ i \ j = \text{Gauss-Jordan-in-ij } A \ i \ j$
 $\langle \text{proof} \rangle$

lemma *invertible-Gauss-Jordan-column-k*:
fixes $A::'a::\{\text{field}\}^{\wedge n}::\{\text{mod-type}\}^{\wedge m}::\{\text{mod-type}\}$ **and** $k::\text{nat}$
shows $\exists P. \text{invertible } P \wedge (\text{snd } (\text{Gauss-Jordan-column-k } (i,A) \ k)) = P**A$
 $\langle \text{proof} \rangle$

lemma *invertible-Gauss-Jordan-up-to-k*:
fixes $A::'a::\{\text{field}\}^{\wedge n}::\{\text{mod-type}\}^{\wedge m}::\{\text{mod-type}\}$
shows $\exists P. \text{invertible } P \wedge (\text{Gauss-Jordan-upt-k } A \ k) = P**A$
 $\langle \text{proof} \rangle$

lemma *inj-index-independent-rows*:
fixes $A::'a::\{\text{field}\}^{\wedge m}::\{\text{mod-type}\}^{\wedge n}::\{\text{finite,one,plus,ord}\}$
assumes $\text{rref-}A$: $\text{reduced-row-echelon-form } A$
and x : $\text{row } x \ A \in \{\text{row } i \ A \mid i. \text{row } i \ A \neq 0\}$
and eq : $A \ \$ \ x = A \ \$ \ y$
shows $x = y$
 $\langle \text{proof} \rangle$

The final results:

lemma *invertible-Gauss-Jordan*:

fixes $A::'a::\{field\}^{\wedge}n::\{mod-type\}^{\wedge}m::\{mod-type\}$
shows $\exists P. invertible P \wedge (Gauss-Jordan A) = P**A$ $\langle proof \rangle$

lemma *Gauss-Jordan*:

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

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 $A::'a::\{field\}^{\wedge}m::\{mod-type\}^{\wedge}n::\{finite,one,plus,ord\}$
assumes *rref-A: reduced-row-echelon-form A*
shows $rank A = card \{row i A \mid i. row i A \neq 0\}$
 $\langle proof \rangle$

lemma *column-leading-coefficient-component-eq*:

fixes $A::'a::\{field\}^{\wedge}m::\{mod-type\}^{\wedge}n::\{finite,one,plus,ord\}$
assumes *rref-A: reduced-row-echelon-form A*
and $v: v \in \{column (LEAST n. A \$ i \$ n \neq 0) A \mid i. row i A \neq 0\}$
and $vx: v \$ x \neq 0$
and $vy: v \$ y \neq 0$
shows $x = y$
 $\langle proof \rangle$

lemma *column-leading-coefficient-component-1*:

fixes $A::'a::\{field\}^{\wedge}m::\{mod-type\}^{\wedge}n::\{finite,one,plus,ord\}$
assumes *rref-A: reduced-row-echelon-form A*
and $v: v \in \{column (LEAST n. A \$ i \$ n \neq 0) A \mid i. row i A \neq 0\}$
and $vx: v \$ x \neq 0$
shows $v \$ x = 1$
 $\langle proof \rangle$

lemma *column-leading-coefficient-component-0*:

fixes $A::'a::\{field\}^{\wedge}m::\{mod-type\}^{\wedge}n::\{finite,one,plus,ord\}$
assumes *rref-A: reduced-row-echelon-form A*
and $v: v \in \{column (LEAST n. A \$ i \$ n \neq 0) A \mid i. row i A \neq 0\}$
and $vx: v \$ x \neq 0$
and *x-not-y: $x \neq y$*
shows $v \$ y = 0$ $\langle proof \rangle$

lemma *rref-col-rank*:

fixes $A::'a::\{field\}^{\wedge}m::\{mod-type\}^{\wedge}n::\{mod-type\}$
assumes *rref-A: reduced-row-echelon-form A*

shows $\text{col-rank } A = \text{card } \{ \text{column } (LEAST\ n.\ A\ \$\ i\ \$\ n \neq 0)\ A \mid i.\ \text{row } i\ A \neq 0 \}$
 ⟨*proof*⟩

lemma *rref-row-rank*:

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

lemma *row-rank-eq-col-rank-rref*:

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

lemma *row-rank-eq-col-rank*:

fixes $A::'a::\{\text{field}\}^{\wedge}n::\{\text{mod-type}\}^{\wedge}m::\{\text{mod-type}\}$
shows $\text{row-rank } A = \text{col-rank } A$
 ⟨*proof*⟩

theorem *rank-col-rank*:

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

theorem *rank-eq-dim-image*:

fixes $A::'a::\{\text{field}\}^{\wedge}n::\{\text{mod-type}\}^{\wedge}m::\{\text{mod-type}\}$
shows $\text{rank } A = \text{vec.dim } (\text{range } (\lambda x.\ A *v\ x))$
 ⟨*proof*⟩

theorem *rank-eq-dim-col-space*:

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

lemma *rank-transpose*:

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

lemma *rank-le-nrows*:

fixes $A::'a::\{\text{field}\}^{\wedge}n::\{\text{mod-type}\}^{\wedge}m::\{\text{mod-type}\}$
shows $\text{rank } A \leq \text{nrows } A$
 ⟨*proof*⟩

lemma *rank-le-ncols*:
fixes $A::'a::\{\text{field}\}^{\wedge n}::\{\text{mod-type}\}^{\wedge m}::\{\text{mod-type}\}$
shows $\text{rank } A \leq \text{ncols } A$
 $\langle \text{proof} \rangle$

lemma *rank-Gauss-Jordan*:
fixes $A::'a::\{\text{field}\}^{\wedge n}::\{\text{mod-type}\}^{\wedge m}::\{\text{mod-type}\}$
shows $\text{rank } A = \text{rank } (\text{Gauss-Jordan } A)$
 $\langle \text{proof} \rangle$

Other interesting properties:

lemma *A-0-imp-Gauss-Jordan-0*:
fixes $A::'a::\{\text{field}\}^{\wedge n}::\{\text{mod-type}\}^{\wedge m}::\{\text{mod-type}\}$
assumes $A=0$
shows $\text{Gauss-Jordan } A = 0$
 $\langle \text{proof} \rangle$

lemma *rank-0*: $\text{rank } 0 = 0$
 $\langle \text{proof} \rangle$

lemma *rank-greater-zero*:
assumes $A \neq 0$
shows $\text{rank } A > 0$
 $\langle \text{proof} \rangle$

lemma *Gauss-Jordan-not-0*:
fixes $A::'a::\{\text{field}\}^{\wedge \text{cols}}::\{\text{mod-type}\}^{\wedge \text{rows}}::\{\text{mod-type}\}$
assumes $A \neq 0$
shows $\text{Gauss-Jordan } A \neq 0$
 $\langle \text{proof} \rangle$

lemma *rank-eq-suc-to-nat-greatest*:
assumes $A\text{-not-0}$: $A \neq 0$
shows $\text{rank } A = \text{to-nat } (\text{GREATEST } a. \neg \text{is-zero-row } a \ (\text{Gauss-Jordan } A)) + 1$
 $\langle \text{proof} \rangle$

lemma *rank-less-row-i-imp-i-is-zero*:
assumes *rank-less-i*: $\text{to-nat } i \geq \text{rank } A$
shows $\text{Gauss-Jordan } A \$ i = 0$
 $\langle \text{proof} \rangle$

lemma *rank-Gauss-Jordan-eq*:
fixes $A::'a::\{\text{field}\}^{\wedge n}::\{\text{mod-type}\}^{\wedge m}::\{\text{mod-type}\}$
shows $\text{rank } A = (\text{let } A'=(\text{Gauss-Jordan } A) \text{ in card } \{\text{row } i \ A' \mid i. \text{row } i \ A' \neq 0\})$
 $\langle \text{proof} \rangle$

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 (λh. (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* \longleftrightarrow *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* \longleftrightarrow *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* \longleftrightarrow *rank* (*matrix* *f*) = *ncols* (*matrix* *f*)

<proof>

7.2 Invertible linear maps

locale *invertible-lf* = *Vector-Spaces.linear* +

assumes *invertible*: (\exists *g*. *g* \circ *f* = *id* \wedge *f* \circ *g* = *id*)

begin

lemma *invertible-lf*: (\exists *g*. *linear* ((*b)) ((*a)) *g* \wedge (*g* \circ *f* = *id*) \wedge (*f* \circ *g* = *id*))

<proof>

end

lemma (**in** *Vector-Spaces.linear*) *invertible-lf-intro*[*intro*]:

assumes (*g* \circ *f* = *id*) **and** (*f* \circ *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 $A::'a::\{\text{field}\}^{\wedge n}$
assumes *invertible-A*: *invertible* A
shows *invertible-lf* $((*)$) $((*)$) $(\lambda x. A * v x)$
<proof>

lemma *invertible-lf-imp-invertible-matrix*:
fixes $f::'a::\{\text{field}\}^{\wedge n} \Rightarrow 'a^{\wedge n}$
assumes *invertible-f*: *invertible-lf* $((*)$) $((*)$) f
shows *invertible* (*matrix* f)
<proof>

lemma *invertible-matrix-iff-invertible-lf*:
fixes $A::'a::\{\text{field}\}^{\wedge n}$
shows *invertible* $A \longleftrightarrow$ *invertible-lf* $((*)$) $((*)$) $(\lambda x. A * v x)$
<proof>

lemma *invertible-matrix-iff-invertible-lf'*:
fixes $f::'a::\{\text{field}\}^{\wedge n} \Rightarrow 'a^{\wedge n}$
assumes *linear-f*: *linear* $((*)$) $((*)$) f
shows *invertible* (*matrix* f) \longleftrightarrow *invertible-lf* $((*)$) $((*)$) f
<proof>

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

lemma *subspace-image-invertible-mat*:
fixes $P::'a::\{\text{field}\}^{\wedge m}$
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 $P::'a::\{\text{field}\}^{\wedge m}$
assumes *inv-P*: *invertible* P
and *sub-W*: *vec.subspace* W
shows *vec.dim* $((\lambda x. P * v x) ' W) = \text{vec.dim } W$
<proof>

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

fixes $A::'a::\{\text{field}\}^{\wedge n}::\{\text{mod-type}\}^{\wedge m}::\{\text{mod-type}\}$
and $P::'a::\{\text{field}\}^{\wedge m}::\{\text{mod-type}\}^{\wedge m}::\{\text{mod-type}\}$
assumes *invertible-P*: *invertible P*
shows $\text{rank } (P**A) = \text{rank } A$
<proof>

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

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

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

lemma *invertible-matrices-rank'*:
fixes $A::'a::\{\text{field}\}^{\wedge n}::\{\text{mod-type}\}^{\wedge m}::\{\text{mod-type}\}$
and $P::'a^{\wedge m}::\{\text{mod-type}\}^{\wedge m}::\{\text{mod-type}\}$ **and** $Q::'a^{\wedge n}::\{\text{mod-type}\}^{\wedge n}::\{\text{mod-type}\}$
assumes *invertible-P*: *invertible P* **and** *invertible-Q*: *invertible Q* **and** *B-eq-PA*:
 $B = P**A**Q$
shows $\text{rank } B = \text{rank } A$ *<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* = $\{\text{axis } i \ (1::?'a) \ | \ i. \ i \in \text{UNIV}\}$.

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 coordinates 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* :: $'a^{\wedge n} \Rightarrow 'a \text{ set}$
where *set-of-vector* $A = \{A \ \$ \ i \ | \ i. \ i \in \text{UNIV}\}$

definition *cart-basis'* :: 'a::{'field'}ⁿⁿ
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 => 'bⁿ
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'* = (χ *i*. *axis i (1::?'a)*)

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

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* (λ *i*. *x \$ i * s cart-basis' \$ i*) *UNIV* = *x*
 ⟨*proof*⟩

lemma *basis-expansion-unique*:
 $\text{sum } (\lambda i. f i * s \text{ axis } (i::'n::\text{finite}) 1) \text{ UNIV} = (x::('a::\text{comm-ring-1})^n) \iff$
 $(\forall i. f i = x $ i)$
 ⟨*proof*⟩

lemma *basis-expansion-cart-basis'-unique*: $\text{sum } (\lambda i. f (\text{cart-basis}' \$ i) * s \text{ cart-basis}' \$ i) \text{ UNIV} = x \iff (\forall i. f (\text{cart-basis}' \$ i) = x $ i)$
 ⟨*proof*⟩

lemma *basis-expansion-cart-basis'-unique'*: $\text{sum } (\lambda i. f i * s \text{ cart-basis}' \$ i) \text{ UNIV} = x \iff (\forall i. f i = x $ i)$
 ⟨*proof*⟩

Properties of *is-basis ?S* \equiv *vec.independent ?S* \wedge *vec.span ?S* = *UNIV*.

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

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

corollary *sum-basis-eq2*:

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

lemma *inj-op-nth*:

fixes $X::'a::\{\text{field}\}^{\wedge n}$
assumes *is-basis: is-basis* (set-of-vector X)
shows $\text{inj } ((\$) X)$
 ⟨proof⟩

lemma *basis-UNIV*:

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

lemma *scalars-zero-if-basis*:

fixes $X::'a::\{\text{field}\}^{\wedge n}$
assumes *is-basis: is-basis* (set-of-vector X) **and** *sum:* $(\sum_{i \in (\text{UNIV}::'n \text{ set})} f i *s X\$i) = 0$
shows $\forall i \in (\text{UNIV}::'n \text{ set}). f i = 0$
 ⟨proof⟩

lemma *basis-combination-unique*:

fixes $X::'a::\{\text{field}\}^{\wedge n}$
assumes *basis-X: is-basis* (set-of-vector X) **and** *sum-eq:* $(\sum_{i \in \text{UNIV}} g i *s X\$i) = (\sum_{i \in \text{UNIV}} f i *s X\$i)$
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* $:: 'a::\{\text{field}\}^{\wedge n} \Rightarrow 'a::\{\text{field}\}^{\wedge n} \Rightarrow 'a::\{\text{field}\}^{\wedge n}$
where $\text{coord } X v = (\chi i. (\text{THE } f. v = \text{sum } (\lambda x. f x *s X\$x) \text{ UNIV}) i)$

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

lemma *bij-coord*:

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

<proof>

lemma *linear-coord*:

fixes $X::'a::\{\text{field}\}^n$

assumes *basis-X*: *is-basis* (*set-of-vector* X)

shows *linear* $((*)$) $((*)$) (*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* :: $'a::\{\text{field}\}^n \Rightarrow 'a^n \Rightarrow 'a^n$

where *matrix-change-of-basis* X Y = $(\chi$ i j . (*coord* Y (X $\$$ j)) $\$$ i)

There exists in the library the definition *matrix* $?f$ = $(\chi$ i j . $?f$ (*axis* j ($1::?'a$)) $\$$ 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'* :: $'a::\{\text{field}\}^n \Rightarrow 'a^m \Rightarrow ('a^n \Rightarrow 'a^m) \Rightarrow 'a^n$

where *matrix'* X Y f = $(\chi$ i j . (*coord* Y (f (X $\$$ j))) $\$$ i)

Properties of *matrix'* $?X$ $?Y$ $?f$ = $(\chi$ i j . *coord* $?Y$ ($?f$ ($?X$ $\$$ j)) $\$$ i)

lemma *matrix'-eq-matrix*:

defines *cart-basis-Rn*: *cart-basis-Rn* == (*cart-basis'*) :: $'a::\{\text{field}\}^n$

and *cart-basis-Rm*: *cart-basis-Rm* == (*cart-basis'*) :: $'a^m$

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* $(\lambda j$. (*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*: $\forall i$. f (X $\$$ i) = *sum* $(\lambda j$. A $\$$ j $\$$ i $*s$ (Y $\$$ j)) *UNIV*

shows $\text{matrix}' X Y f = A$
 ⟨proof⟩

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

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

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

lemma *matrix'-compose*:
fixes $X::'a::\{\text{field}\}^{\wedge n} \wedge^{\wedge n}$ **and** $Y::'a^{\wedge m} \wedge^{\wedge m}$ **and** $Z::'a^{\wedge p} \wedge^{\wedge p}$
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 ((*) ((*)) f)* **and** *linear-g: linear ((*) ((*)) g)*
shows $\text{matrix}' X Z (g \circ f) = (\text{matrix}' Y Z g) ** (\text{matrix}' X Y f)$
 ⟨proof⟩

lemma *exists-linear-eq-matrix'*:
fixes $A::'a::\{\text{field}\}^{\wedge m} \wedge^{\wedge n}$ **and** $X::'a^{\wedge m} \wedge^{\wedge m}$ **and** $Y::'a^{\wedge n} \wedge^{\wedge n}$
assumes *basis-X: is-basis (set-of-vector X)* **and** *basis-Y: is-basis (set-of-vector Y)*
shows $\exists f. \text{matrix}' X Y f = A \wedge \text{linear } ((*) ((*)) f$
 ⟨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* $?X ?Y = (\chi i j. \text{coord } ?Y (?X \$ j) \$ i)$.

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 $X::'a::\{\text{field}\}^{\wedge n} \wedge^{\wedge n}$ **and** $Y::'a^{\wedge n} \wedge^{\wedge n}$
assumes *basis-X: is-basis (set-of-vector X)*
and *basis-Y: is-basis (set-of-vector Y)*
shows $(\text{matrix-change-of-basis } X Y) * v (\text{coord } X v) = (\text{coord } Y v)$

<proof>

lemma *matrix-change-of-basis-mat-1*:
fixes $X::'a::\{\text{field}\}^{\wedge n} \wedge n$
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 = ($\chi i j.$ coord ?Y (?f (?X \$ j)) \$ i)* and *matrix-change-of-basis ?X ?Y = ($\chi i j.$ 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 $B::'a::\{\text{field}\}^{\wedge n} \wedge n$ and $B'::'a^{\wedge n} \wedge n$ and $C::'a^{\wedge m} \wedge m$ and $C'::'a^{\wedge m} \wedge m$
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)) 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 $X::'a::\{\text{field}\}^{\wedge n} \wedge n$ and $Y::'a^{\wedge n} \wedge n$
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 \equiv linear ?s1.0 ?s2.0 ?f \wedge invertible-lf-axioms ?f*, *matrix-change-of-basis ?X ?Y = ($\chi i j.$ coord ?Y (?X \$ j) \$ i)*, *matrix' ?X ?Y ?f = ($\chi i j.$ coord ?Y (?f (?X \$ j)) \$ i)* and *invertible ?A = ($\exists A'. ?A ** A' = mat (1::?'a) \wedge A' ** ?A = mat (1::?'a)$)*.

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 $X::'a::\{\text{field}\}^{\wedge n} \wedge n$ and $Y::'a^{\wedge n} \wedge n$
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 $X::'a::\{\text{field}\}^{\wedge n} \wedge n$ **and** $Y::'a^{\wedge n} \wedge n$
assumes $\text{basis-}X: \text{is-basis (set-of-vector } X)$ **and** $\text{basis-}Y: \text{is-basis (set-of-vector } Y)$
shows $\text{invertible (matrix-change-of-basis } X Y)$
 $\langle \text{proof} \rangle$

lemma $\text{invertible-}lf\text{-imp-invertible-matrix'}$:
fixes $f::'a::\{\text{field}\}^{\wedge b} \Rightarrow 'a^{\wedge b}$
assumes $\text{invertible-}lf ((*) (**) f$ **and** $\text{basis-}X: \text{is-basis (set-of-vector } X)$ **and**
 $\text{basis-}Y: \text{is-basis (set-of-vector } Y)$
shows $\text{invertible (matrix' } X Y f)$
 $\langle \text{proof} \rangle$

lemma $\text{invertible-matrix'-imp-invertible-}lf$:
fixes $f::'a::\{\text{field}\}^{\wedge b} \Rightarrow 'a^{\wedge b}$
assumes $\text{invertible (matrix' } X Y f)$ **and** $\text{basis-}X: \text{is-basis (set-of-vector } X)$
and $\text{linear-}f: \text{linear ((*) ((*) f$ **and** $\text{basis-}Y: \text{is-basis (set-of-vector } Y)$
shows $\text{invertible-}lf ((*) ((*) f)$
 $\langle \text{proof} \rangle$

lemma $\text{invertible-matrix-is-change-of-basis}$:
assumes $\text{invertible-}P: \text{invertible } P$ **and** $\text{basis-}X: \text{is-basis (set-of-vector } X)$
shows $\exists! Y. \text{matrix-change-of-basis } Y X = P \wedge \text{is-basis (set-of-vector } Y)$
 $\langle \text{proof} \rangle$

7.6 Equivalent Matrices

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

definition $\text{equivalent-matrices } A B = (\exists P Q. \text{invertible } P \wedge \text{invertible } Q \wedge B =$
 $(\text{matrix-inv } P)**A**Q)$

lemma $\text{exists-basis: } \exists X::'a::\{\text{field}\}^{\wedge n} \wedge n. \text{is-basis (set-of-vector } X)$
 $\langle \text{proof} \rangle$

lemma $\text{equivalent-implies-exist-matrix'}$:
assumes $\text{equivalent: equivalent-matrices } A B$
shows $\exists X Y X' Y' f::'a::\{\text{field}\}^{\wedge n} \Rightarrow 'a^{\wedge m}.$
 $\text{linear ((*) ((*) f} \wedge \text{matrix' } X Y f = A \wedge \text{matrix' } X' Y' f = B \wedge \text{is-basis}$
 $(\text{set-of-vector } X)$
 $\wedge \text{is-basis (set-of-vector } Y) \wedge \text{is-basis (set-of-vector } X') \wedge \text{is-basis (set-of-vector } Y')$
 $\langle \text{proof} \rangle$

lemma $\text{exist-matrix'-implies-equivalent}$:
assumes $A: \text{matrix' } X Y f = A$
and $B: \text{matrix' } X' Y' f = B$
and $X: \text{is-basis (set-of-vector } X)$
and $Y: \text{is-basis (set-of-vector } Y)$

and X' : *is-basis* (*set-of-vector* X')
and Y' : *is-basis* (*set-of-vector* Y')
and *linear-f*: *linear* $((*)$) $((*)$) 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* $A B \longleftrightarrow (\exists X Y X' Y' f :: 'a :: \{\text{field}\}^{\wedge n} \Rightarrow 'a^{\wedge m}.$
linear $((*)$) $((*)$) $f \wedge \text{matrix}' X Y f = A \wedge \text{matrix}' X' Y' f = B$
 \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* $:: 'a :: \{\text{semiring-1}\}^{\wedge n} \Rightarrow 'a :: \{\text{semiring-1}\}^{\wedge n} \Rightarrow \text{bool}$

where *similar-matrices* $A B = (\exists P. \text{invertible } P \wedge B = (\text{matrix-inv } P) ** A ** P)$

lemma *similar-implies-exist-matrix'*:

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

lemma *exist-matrix'-implies-similar*:

fixes $A B :: 'a :: \{\text{field}\}^{\wedge n} \Rightarrow 'a^{\wedge n}$
assumes *linear-f*: *linear* $((*)$) $((*)$) 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 $A B :: 'a :: \{\text{field}\}^{\wedge n} \Rightarrow 'a^{\wedge n}$
shows *similar-matrices* $A B \longleftrightarrow (\exists X Y f. \text{linear} ((*)$) $((*)$) $f \wedge \text{matrix}' X X f = A$
 $\wedge \text{matrix}' Y Y f = B \wedge$ *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 = nrow 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..

```

```

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 efficiency 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 $B = \text{vec-lambda}(\%s. \text{if } s=i \text{ then } A \ \$ \ s \ \text{else } (\text{row-add } A \ s \ i \ (- (B\ \$ \ s\ \$ \ j))) \ \$ \ s)$

lemma *Gauss-Jordan-wrapper-code*[code abstract]:

$\text{vec-nth } (\text{Gauss-Jordan-wrapper } i \ j \ A \ B) = (\%s. \text{if } s=i \text{ then } A \ \$ \ s \ \text{else } (\text{row-add } A \ s \ i \ (- (B\ \$ \ s\ \$ \ j))) \ \$ \ s)$
 ⟨proof⟩

lemma *Gauss-Jordan-in-ij-PA-def'*[code]:

$\text{Gauss-Jordan-in-ij-PA } A' \ i \ j = (\text{let } P = \text{fst } A'; \ A = \text{snd } A';$
 $n = (\text{LEAST } n. \ A \ \$ \ n \ \$ \ j \neq 0 \wedge i \leq n);$
 $\text{interchange-A} = (\text{interchange-rows } A \ i \ n);$
 $A' = \text{mult-row interchange-A } i \ (1 / \text{interchange-A } \$ \ i \ \$ \ j);$
 $\text{interchange-P} = (\text{interchange-rows } P \ i \ n);$
 $P' = \text{mult-row interchange-P } i \ (1 / \text{interchange-A } \$ \ i \ \$ \ j)$
 in
 $(\text{Gauss-Jordan-wrapper } i \ j \ P' \ \text{interchange-A},$
 $\text{Gauss-Jordan-wrapper } i \ j \ A' \ \text{interchange-A}))$

⟨proof⟩

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

lemma *snd-Gauss-Jordan-in-ij-PA-eq*[code-unfold]: $\text{snd } (\text{Gauss-Jordan-in-ij-PA } (P, A) \ i \ j) = \text{Gauss-Jordan-in-ij } A \ i \ j$

⟨proof⟩

lemma *fst-Gauss-Jordan-in-ij-PA*:

fixes $A :: 'a :: \{\text{field}\} \ \sim \ \text{cols} :: \{\text{mod-type}\} \ \sim \ \text{rows} :: \{\text{mod-type}\}$

assumes $PB-A: P ** B = A$

shows $\text{fst } (\text{Gauss-Jordan-in-ij-PA } (P, A) \ i \ j) ** B = \text{snd } (\text{Gauss-Jordan-in-ij-PA } (P, A) \ i \ j)$

⟨proof⟩

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

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

assumes $i \leq \text{nrows } A$

shows $\text{fst } (\text{Gauss-Jordan-column-k } (i, A) \ k) \leq \text{nrows } A$

⟨proof⟩

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

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

lemma $\text{snd-snd-Gauss-Jordan-column-k-PA-eq}$:
shows $\text{snd} (\text{snd} (\text{Gauss-Jordan-column-k-PA} (P, i, A) k)) = \text{snd} (\text{Gauss-Jordan-column-k} (i, A) k)$
 $\langle \text{proof} \rangle$

lemma $\text{fst-snd-Gauss-Jordan-column-k-PA-eq}$:
shows $\text{fst} (\text{snd} (\text{Gauss-Jordan-column-k-PA} (P, i, A) k)) = \text{fst} (\text{Gauss-Jordan-column-k} (i, A) k)$
 $\langle \text{proof} \rangle$

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

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

lemma $\text{snd-foldl-Gauss-Jordan-column-k-eq}$:
 $\text{snd} (\text{foldl Gauss-Jordan-column-k-PA} (\text{mat } 1, 0, A) [0..<k]) = \text{foldl Gauss-Jordan-column-k} (0, A) [0..<k]$
 $\langle \text{proof} \rangle$

lemma $\text{snd-Gauss-Jordan-upt-k-PA}$:
shows $\text{snd} (\text{Gauss-Jordan-upt-k-PA} A k) = (\text{Gauss-Jordan-upt-k} A k)$
 $\langle \text{proof} \rangle$

8.2.4 Properties about *Gauss-Jordan-PA*

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

lemma $\text{Gauss-Jordan-PA-eq}$:
shows $\text{snd} (\text{Gauss-Jordan-PA} A) = (\text{Gauss-Jordan} A)$
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

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

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 $\det (\text{interchange-rows } A \ i \ j) = \text{of-int } (\text{if } i = j \text{ then } 1 \text{ else } -1) * \det A$
<proof>

corollary *det-interchange-different-rows*:
assumes *i-not-j*: $i \neq j$
shows $\det (\text{interchange-rows } A \ i \ j) = - \det A$ *<proof>*

corollary *det-interchange-same-rows*:
assumes *i-eq-j*: $i = j$
shows $\det (\text{interchange-rows } A \ i \ j) = \det A$ *<proof>*

lemma *det-mult-row*:
shows $\det (\text{mult-row } A \ a \ k) = k * \det A$
<proof>

lemma *det-row-add'*:
assumes *i-not-j*: $i \neq j$
shows $\det (\text{row-add } A \ i \ j \ q) = \det A$
<proof>

9.1.2 Relationships between determinants and elementary column operations

lemma *det-interchange-columns*:
shows $\det (\text{interchange-columns } A \ i \ j) = \text{of-int } (\text{if } i = j \text{ then } 1 \text{ else } -1) * \det A$
<proof>

corollary *det-interchange-different-columns*:
assumes *i-not-j*: $i \neq j$
shows $\det (\text{interchange-columns } A \ i \ j) = - \det A$ *<proof>*

corollary *det-interchange-same-columns*:
assumes *i-eq-j*: $i = j$
shows $\det (\text{interchange-columns } A \ i \ j) = \det A$ *<proof>*

lemma *det-mult-columns*:
shows $\det (\text{mult-column } A \ a \ k) = k * \det A$
<proof>

lemma *det-column-add*:
assumes *i-not-j*: $i \neq j$
shows $\det (\text{column-add } A \ i \ j \ q) = \det A$
 $\langle \text{proof} \rangle$

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 $A::'a::\{\text{comm-ring-1}\}^{\sim n}::\{\text{mod-type}\}^{\sim n}::\{\text{mod-type}\}$
assumes $n: n < n\text{rows } A$
shows $\det (\text{row-add-iterate } A \ n \ i \ j) = \det A$
 $\langle \text{proof} \rangle$

corollary *det-row-add-iterate*:
fixes $A::'a::\{\text{comm-ring-1}\}^{\sim n}::\{\text{mod-type}\}^{\sim n}::\{\text{mod-type}\}$
shows $\det (\text{row-add-iterate } A \ (n\text{rows } A - 1) \ i \ j) = \det A$
 $\langle \text{proof} \rangle$

lemma *det-Gauss-Jordan-in-ij*:
fixes $A::'a::\{\text{field}\}^{\sim n}::\{\text{mod-type}\}^{\sim n}::\{\text{mod-type}\}$ **and** $i \ j::'n$
defines A' : $A' == \text{mult-row } (\text{interchange-rows } A \ i \ (\text{LEAST } n. A \ \$ \ n \ \$ \ j \neq 0 \wedge i \leq n)) \ i \ (1 / (\text{interchange-rows } A \ i \ (\text{LEAST } n. A \ \$ \ n \ \$ \ j \neq 0 \wedge i \leq n)) \ \$ \ i \ \$ \ j)$
shows $\det (\text{Gauss-Jordan-in-ij } A \ i \ j) = \det A'$
 $\langle \text{proof} \rangle$

lemma *det-Gauss-Jordan-in-ij-1*:
fixes $A::'a::\{\text{field}\}^{\sim n}::\{\text{mod-type}\}^{\sim n}::\{\text{mod-type}\}$ **and** $i \ j::'n$
defines A' : $A' == \text{mult-row } (\text{interchange-rows } A \ i \ (\text{LEAST } n. A \ \$ \ n \ \$ \ j \neq 0 \wedge i \leq n)) \ i \ (1 / (\text{interchange-rows } A \ i \ (\text{LEAST } n. A \ \$ \ n \ \$ \ j \neq 0 \wedge i \leq n)) \ \$ \ i \ \$ \ j)$
assumes $i: (\text{LEAST } n. A \ \$ \ n \ \$ \ j \neq 0 \wedge i \leq n) = i$
shows $\det (\text{Gauss-Jordan-in-ij } A \ i \ j) = 1 / (A \ \$ \ i \ \$ \ j) * \det A$
 $\langle \text{proof} \rangle$

lemma *det-Gauss-Jordan-in-ij-2*:
fixes $A::'a::\{\text{field}\}^{\sim n}::\{\text{mod-type}\}^{\sim n}::\{\text{mod-type}\}$ **and** $i \ j::'n$
defines A' : $A' == \text{mult-row } (\text{interchange-rows } A \ i \ (\text{LEAST } n. A \ \$ \ n \ \$ \ j \neq 0 \wedge i \leq n)) \ i \ (1 / (\text{interchange-rows } A \ i \ (\text{LEAST } n. A \ \$ \ n \ \$ \ j \neq 0 \wedge i \leq n)) \ \$ \ i \ \$ \ j)$
assumes $i: (\text{LEAST } n. A \ \$ \ n \ \$ \ j \neq 0 \wedge i \leq n) \neq i$
shows $\det (\text{Gauss-Jordan-in-ij } A \ i \ j) = -1 / (A \ \$ \ (\text{LEAST } n. A \ \$ \ n \ \$ \ j \neq 0 \wedge i \leq n) \ \$ \ j) * \det A$
 $\langle \text{proof} \rangle$

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 = nrow 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 (ncol 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 $A::'a::\{\text{field}\}^{\wedge n}::\{\text{mod-type}\}^{\wedge n}::\{\text{mod-type}\}$
assumes $\text{det}: \text{det-}P * \text{det } B = \text{det } A$
shows $(\text{fst } (\text{Gauss-Jordan-column-}k\text{-det-}P (\text{det-}P, i, A) k)) * \text{det } B = \text{det } (\text{snd } (\text{snd } (\text{Gauss-Jordan-column-}k\text{-det-}P (\text{det-}P, i, A) k)))$
 $\langle \text{proof} \rangle$

lemma *det-Gauss-Jordan-upt- k -det- P :*
fixes $A::'a::\{\text{field}\}^{\wedge n}::\{\text{mod-type}\}^{\wedge n}::\{\text{mod-type}\}$
shows $(\text{fst } (\text{Gauss-Jordan-upt-}k\text{-det-}P A k)) * \text{det } A = \text{det } (\text{snd } (\text{Gauss-Jordan-upt-}k\text{-det-}P A k))$
 $\langle \text{proof} \rangle$

lemma *det-Gauss-Jordan-det- P :*
fixes $A::'a::\{\text{field}\}^{\wedge n}::\{\text{mod-type}\}^{\wedge n}::\{\text{mod-type}\}$
shows $(\text{fst } (\text{Gauss-Jordan-det-}P A)) * \text{det } A = \text{det } (\text{snd } (\text{Gauss-Jordan-det-}P A))$
 $\langle \text{proof} \rangle$

definition *upper-triangular-upt- k* **where** $\text{upper-triangular-upt-}k A k = (\forall i j. j < i \wedge \text{to-nat } j < k \longrightarrow A \$ i \$ j = 0)$

definition *upper-triangular* **where** $\text{upper-triangular } A = (\forall i j. j < i \longrightarrow A \$ i \$ j = 0)$

lemma *upper-triangular-upt-imp-upper-triangular:*
assumes $\text{upper-triangular-upt-}k A (nrows A)$
shows $\text{upper-triangular } A$
 $\langle \text{proof} \rangle$

lemma *rref-imp-upper-triangular-upt:*
fixes $A::'a::\{\text{one, zero}\}^{\wedge n}::\{\text{mod-type}\}^{\wedge n}::\{\text{mod-type}\}$
assumes $\text{reduced-row-echelon-form } A$
shows $\text{upper-triangular-upt-}k A k$
 $\langle \text{proof} \rangle$

lemma *rref-imp-upper-triangular:*
assumes $\text{reduced-row-echelon-form } A$
shows $\text{upper-triangular } A$
 $\langle \text{proof} \rangle$

lemma *det-Gauss-Jordan[`code-unfold`]:*
fixes $A::'a::\{\text{field}\}^{\wedge n}::\{\text{mod-type}\}^{\wedge n}::\{\text{mod-type}\}$
shows $\text{det } (\text{Gauss-Jordan } A) = \text{prod } (\lambda i. (\text{Gauss-Jordan } A) \$ i \$ i) (\text{UNIV}:: 'n \text{ set})$
 $\langle \text{proof} \rangle$

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

shows $\text{snd} (\text{Gauss-Jordan-in-ij-det-P } A \ i \ j) = \text{snd} (\text{Gauss-Jordan-in-ij-PA } (P, A) \ i \ j)$
 ⟨proof⟩

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

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

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

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

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

The way of proving the following lemma is very similar to the demonstration of $?k < n\text{cols } ?A \implies \text{reduced-row-echelon-form-upt-k} (\text{Gauss-Jordan-upt-k } ?A \ ?k) (\text{Suc } ?k)$

$?k < n\text{cols } ?A \implies \text{foldl Gauss-Jordan-column-k } (0, ?A) [0..<\text{Suc } ?k] =$
 (if $\forall m. \text{is-zero-row-upt-k } m (\text{Suc } ?k) (\text{snd} (\text{foldl Gauss-Jordan-column-k } (0, ?A) [0..<\text{Suc } ?k]))$ then 0 else $\text{mod-type-class.to-nat} (\text{GREATEST } n. \neg \text{is-zero-row-upt-k } n (\text{Suc } ?k) (\text{snd} (\text{foldl Gauss-Jordan-column-k } (0, ?A) [0..<\text{Suc } ?k]))) + 1, \text{snd} (\text{foldl Gauss-Jordan-column-k } (0, ?A) [0..<\text{Suc } ?k]))$).

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

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

lemma $snd\text{-Gauss-Jordan-det-P-is-Gauss-Jordan}$:
fixes $A::'a::\{field\}^{\wedge n}::\{mod-type\}^{\wedge n}::\{mod-type\}$
shows $snd (Gauss-Jordan-det-P A) = (Gauss-Jordan A)$
 $\langle proof \rangle$

lemma $det\text{-snd-Gauss-Jordan-det-P}[code-unfold]$:
fixes $A::'a::\{field\}^{\wedge n}::\{mod-type\}^{\wedge n}::\{mod-type\}$
shows $det (snd (Gauss-Jordan-det-P A)) = prod (\lambda i. (snd (Gauss-Jordan-det-P A))\$i\$i) (UNIV::'n set)$
 $\langle proof \rangle$

lemma $det\text{-fst-Gauss-Jordan-PA-eq-fst-Gauss-Jordan-det-P}$:
fixes $A::'a::\{field\}^{\wedge n}::\{mod-type\}^{\wedge n}::\{mod-type\}$
shows $fst (Gauss-Jordan-det-P A) = det (fst (Gauss-Jordan-PA A))$
 $\langle proof \rangle$

lemma $fst\text{-Gauss-Jordan-det-P-not-0}$:
fixes $A::'a::\{field\}^{\wedge n}::\{mod-type\}^{\wedge n}::\{mod-type\}$
shows $fst (Gauss-Jordan-det-P A) \neq 0$
 $\langle proof \rangle$

lemma $det\text{-code-equation}[code-unfold]$:
fixes $A::'a::\{field\}^{\wedge n}::\{mod-type\}^{\wedge n}::\{mod-type\}$
shows $det A = (let A' = Gauss-Jordan-det-P A in prod (\lambda i. (snd (A'))\$i\$i) (UNIV::'n set))/(fst (A'))$
 $\langle proof \rangle$

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 ::  $'a::\{zero, one\}^n::\{mod-type\}^n::\{mod-type\} \Rightarrow nat \Rightarrow$ 
  bool
where id-upt-k  $A\ k = (\forall i\ j. to-nat\ i < k \wedge to-nat\ j < k \longrightarrow ((i = j \longrightarrow A\ \$\ i\ \$$ 
   $j = 1) \wedge (i \neq j \longrightarrow A\ \$\ i\ \$\ j = 0)))$ 

```

```

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 $A::'a::\{field\}^n::\{mod-type\}^n::\{mod-type\}$
assumes *inv-A*: *invertible A*
shows *id-upt-k (Gauss-Jordan A) k*
<proof>

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

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

lemma *invertible-eq-full-rank**[code-unfold]*:
fixes $A::'a::\{field\}^n::\{mod-type\}^n::\{mod-type\}$
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 $A::'a::\{field\}^n::\{mod-type\}^n::\{mod-type\}$
assumes *invertible A*
shows *the (inverse-matrix A) = P-Gauss-Jordan A*
<proof>

11.3 Code equations

Code equations to make more efficient the computations.

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

lemma *basis-row-space-code*[code]: *basis-row-space* $A = (\text{let } A' = \text{Gauss-Jordan } A$
 $\text{in } \{\text{row } i A' \mid i. \text{row } i A' \neq 0\})$
 ⟨proof⟩

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

lemma *basis-left-null-space-code*[code]: *basis-left-null-space* $A = (\text{let } GJ = \text{Gauss-Jordan-PA}$
 $A;$
 $\text{rank-}A = (\text{if } A = 0 \text{ then } 0 \text{ else}$
 $\text{to-nat } (\text{GREATEST } a. \text{row } a (\text{snd } GJ) \neq 0) + 1)$
 $\text{in } \{\text{row } i (\text{fst } GJ) \mid i. \text{to-nat } i \geq$
 $\text{rank-}A\}$)
 ⟨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 $A::'a::\{\text{field}\} \wedge \text{cols}::\{\text{mod-type}\} \wedge \text{rows}::\{\text{mod-type}\}$
shows *vec.independent* (*basis-left-null-space* A)
 ⟨proof⟩

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

lemma *basis-left-null-space-in-left-null-space*:
fixes $A::'a::\{\text{field}\} \wedge \text{cols}::\{\text{mod-type}\} \wedge \text{rows}::\{\text{mod-type}\}$
shows $\text{basis-left-null-space } A \subseteq \text{left-null-space } A$
 ⟨proof⟩

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

fixes $A::'a::\{\text{field}\} \sim \text{cols}::\{\text{mod-type}\} \sim \text{rows}::\{\text{mod-type}\}$
shows $\text{left-null-space } A \subseteq \text{vec.span } (\text{basis-left-null-space } A)$
 $\langle \text{proof} \rangle$

corollary *basis-left-null-space*:
fixes $A::'a::\{\text{field}\} \sim \text{cols}::\{\text{mod-type}\} \sim \text{rows}::\{\text{mod-type}\}$
shows $\text{vec.independent } (\text{basis-left-null-space } A) \wedge$
 $\text{left-null-space } A = \text{vec.span } (\text{basis-left-null-space } A)$
 $\langle \text{proof} \rangle$

corollary *basis-null-space*:
fixes $A::'a::\{\text{field}\} \sim \text{cols}::\{\text{mod-type}\} \sim \text{rows}::\{\text{mod-type}\}$
shows $\text{vec.independent } (\text{basis-null-space } A) \wedge$
 $\text{null-space } A = \text{vec.span } (\text{basis-null-space } A)$
 $\langle \text{proof} \rangle$

lemma *basis-row-space-subset-row-space*:
fixes $A::'a::\{\text{field}\} \sim \text{cols}::\{\text{mod-type}\} \sim \text{rows}::\{\text{mod-type}\}$
shows $\text{basis-row-space } A \subseteq \text{row-space } A$
 $\langle \text{proof} \rangle$

lemma *row-space-subset-span-basis-row-space*:
fixes $A::'a::\{\text{field}\} \sim \text{cols}::\{\text{mod-type}\} \sim \text{rows}::\{\text{mod-type}\}$
shows $\text{row-space } A \subseteq \text{vec.span } (\text{basis-row-space } A)$
 $\langle \text{proof} \rangle$

lemma *basis-row-space*:
fixes $A::'a::\{\text{field}\} \sim \text{cols}::\{\text{mod-type}\} \sim \text{rows}::\{\text{mod-type}\}$
shows $\text{vec.independent } (\text{basis-row-space } A)$
 $\wedge \text{vec.span } (\text{basis-row-space } A) = \text{row-space } A$
 $\langle \text{proof} \rangle$

corollary *basis-col-space*:
fixes $A::'a::\{\text{field}\} \sim \text{cols}::\{\text{mod-type}\} \sim \text{rows}::\{\text{mod-type}\}$
shows $\text{vec.independent } (\text{basis-col-space } A)$
 $\wedge \text{vec.span } (\text{basis-col-space } A) = \text{col-space } A$
 $\langle \text{proof} \rangle$

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* $\text{inconsistent } A \ b = (\neg (\text{consistent } A \ b))$

lemma *inconsistent:* $\text{inconsistent } A \ b = (\neg (\exists x. \text{is-solution } x \ A \ b))$
 ⟨proof⟩

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

definition *solve-consistent-rref* :: $'a::\{\text{field}\} \wedge 'cols::\{\text{mod-type}\} \wedge 'rows::\{\text{mod-type}\} \Rightarrow 'a::\{\text{field}\} \wedge 'rows::\{\text{mod-type}\} \Rightarrow 'a::\{\text{field}\} \wedge 'cols::\{\text{mod-type}\}$
where *solve-consistent-rref* $A \ b = (\chi \ j. \text{if } (\exists i. A \ \$ \ i \ \$ \ j = 1 \wedge j=(\text{LEAST } n. A \ \$ \ i \ \$ \ n \neq 0)) \text{ then } b \ \$ \ (\text{THE } i. A \ \$ \ i \ \$ \ j = 1) \text{ else } 0)$

lemma *solve-consistent-rref-code*[code abstract]:
shows *vec-nth* $(\text{solve-consistent-rref } A \ b) = (\% \ j. \text{if } (\exists i. A \ \$ \ i \ \$ \ j = 1 \wedge j=(\text{LEAST } n. A \ \$ \ i \ \$ \ n \neq 0)) \text{ then } b \ \$ \ (\text{THE } i. A \ \$ \ i \ \$ \ j = 1) \text{ else } 0)$
 ⟨proof⟩

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

corollary *rank-ge-imp-consistent:*
fixes $A::'a::\{\text{field}\} \wedge 'cols::\{\text{mod-type}\} \wedge 'rows::\{\text{mod-type}\}$
assumes $\text{rank } A \geq (\text{if } (\exists a. (P\text{-Gauss-Jordan } A \ *v \ b) \ \$ \ a \neq 0) \text{ then } (\text{to-nat } (\text{GREATEST } a. (P\text{-Gauss-Jordan } A \ *v \ b) \ \$ \ a \neq 0) + 1) \text{ else } 0)$
shows *consistent* $A \ b$
 ⟨proof⟩

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

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

<proof>

corollary *consistent-imp-rank-ge:*

fixes $A::'a::\{\text{field}\} \sim \text{cols}::\{\text{mod-type}\} \sim \text{rows}::\{\text{mod-type}\}$
assumes *consistent A b*
shows $\text{rank } A \geq (\text{if } (\exists a. (\text{P-Gauss-Jordan } A *v b) \$ a \neq 0) \text{ then } (\text{to-nat } (\text{GREATEST } a. (\text{P-Gauss-Jordan } A *v b) \$ a \neq 0) + 1) \text{ else } 0)$
<proof>

lemma *inconsistent-eq-rank-less:*

fixes $A::'a::\{\text{field}\} \sim \text{cols}::\{\text{mod-type}\} \sim \text{rows}::\{\text{mod-type}\}$
shows $\text{inconsistent } A b = (\text{rank } A < (\text{if } (\exists a. (\text{P-Gauss-Jordan } A *v b) \$ a \neq 0) \text{ then } (\text{to-nat } (\text{GREATEST } a. (\text{P-Gauss-Jordan } A *v b) \$ a \neq 0) + 1) \text{ else } 0))$
<proof>

lemma *consistent-eq-rank-ge:*

fixes $A::'a::\{\text{field}\} \sim \text{cols}::\{\text{mod-type}\} \sim \text{rows}::\{\text{mod-type}\}$
shows $\text{consistent } A b = (\text{rank } A \geq (\text{if } (\exists a. (\text{P-Gauss-Jordan } A *v b) \$ a \neq 0) \text{ then } (\text{to-nat } (\text{GREATEST } a. (\text{P-Gauss-Jordan } A *v b) \$ a \neq 0) + 1) \text{ else } 0))$
<proof>

corollary *consistent-imp-is-solution:*

fixes $A::'a::\{\text{field}\} \sim \text{cols}::\{\text{mod-type}\} \sim \text{rows}::\{\text{mod-type}\}$
assumes *consistent A b*
shows $\text{is-solution } (\text{solve-consistent-rref } (\text{Gauss-Jordan } A) (\text{P-Gauss-Jordan } A *v b)) A b$
<proof>

corollary *consistent-imp-is-solution':*

fixes $A::'a::\{\text{field}\} \sim \text{cols}::\{\text{mod-type}\} \sim \text{rows}::\{\text{mod-type}\}$
assumes *consistent A b*
shows $\text{is-solution } (\text{solve-consistent-rref } (\text{fst } (\text{solve-system } A b)) (\text{snd } (\text{solve-system } A b))) A b$
<proof>

Code equations optimized using Lets

lemma *inconsistent-eq-rank-less-code[code]:*

fixes $A::'a::\{\text{field}\} \sim \text{cols}::\{\text{mod-type}\} \sim \text{rows}::\{\text{mod-type}\}$
shows $\text{inconsistent } A b = (\text{let } GJ\text{-}P = \text{Gauss-Jordan-PA } A;$
 $P\text{-mult-}b = (\text{fst}(GJ\text{-}P) *v b);$
 $\text{rank-}A = (\text{if } A = 0 \text{ then } 0 \text{ else } \text{to-nat } (\text{GREATEST } a.$
 $\text{row } a (\text{snd } GJ\text{-}P) \neq 0) + 1) \text{ in } (\text{rank-}A < (\text{if } (\exists a. P\text{-mult-}b \$ a \neq 0)$
 $\text{then } (\text{to-nat } (\text{GREATEST } a. P\text{-mult-}b \$ a \neq 0) + 1) \text{ else } 0))$

<proof>

lemma *consistent-eq-rank-ge-code*[code]:
fixes $A::'a::\{\text{field}\}^{\wedge}n::\{\text{mod-type}\}^{\wedge}rows::\{\text{mod-type}\}$
shows $consistent\ A\ b = (let\ GJ-P=Gauss-Jordan-PA\ A;$
 $\quad P-mult-b = (fst(GJ-P) *v\ b);$
 $\quad rank-A = (if\ A = 0\ then\ 0\ else\ to-nat\ (GREATEST\ a.\ row$
 $a\ (snd\ GJ-P) \neq 0) + 1)\ in\ (rank-A \geq (if\ (\exists\ a.\ P-mult-b\ \$\ a \neq 0)$
 $\quad then\ (to-nat\ (GREATEST\ a.\ P-mult-b\ \$\ a \neq$
 $0) + 1)\ else\ 0)))$
<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 $0 \in (solution-set\ A\ 0)$
<proof>

lemma *homogeneous-solution-set-subspace*:
fixes $A::'a::\{\text{field}\}^{\wedge}n::\{\text{mod-type}\}^{\wedge}rows$
shows $vec.subspace\ (solution-set\ A\ 0)$
<proof>

lemma *solution-set-rel*:
fixes $A::'a::\{\text{field}\}^{\wedge}n::\{\text{mod-type}\}^{\wedge}rows$
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 $A::'a::\{\text{field}\}^{\wedge}n::\{\text{mod-type}\}^{\wedge}rows::\{\text{mod-type}\}$
assumes $dim-0: vec.dim\ (solution-set\ A\ 0) = 0$
and $con: consistent\ A\ b$
shows $\exists!x.\ is-solution\ x\ A\ b$
<proof>

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

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

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

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

lemma *consistent-homogeneous*:
shows $\text{consistent } A \ 0$ ⟨proof⟩

lemma *dim-solution-set-0*:
fixes $A::'a::\{\text{field}\}^{\wedge n}::\{\text{mod-type}\}^{\wedge \text{rows}}::\{\text{mod-type}\}$
shows $(\text{vec.dim } (\text{solution-set } A \ 0) = 0) = (\text{solution-set } A \ 0 = \{0\})$
 ⟨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 $\mathbb{Z}2$ 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 $A::'a::\{\text{field}, \text{semiring-char-0}\}^{\wedge n}::\{\text{mod-type}\}^{\wedge \text{rows}}::\{\text{mod-type}\}$
assumes dim-not-zero : $\text{vec.dim } (\text{solution-set } A \ 0) > 0$
shows $\text{infinite } (\text{solution-set } A \ 0)$
 ⟨proof⟩

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

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

fixes $A::'a::\{\text{field, semiring-char-0}\}^{\wedge}n::\{\text{mod-type}\}^{\wedge}rows::\{\text{mod-type}\}$
shows $\text{infinite } (\text{solution-set } A \ 0) = (\text{vec.dim } (\text{solution-set } A \ 0) > 0)$
 $\langle \text{proof} \rangle$

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

lemma *infinite-solution-set-imp-consistent*:
 $\text{infinite } (\text{solution-set } A \ b) \implies \text{consistent } A \ b$
 $\langle \text{proof} \rangle$

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

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

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

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

definition *independent-and-consistent* $A \ b = (\text{consistent } A \ b \wedge \text{vec.dim } (\text{solution-set } A \ 0) = 0)$

definition *dependent-and-consistent* $A \ b = (\text{consistent } A \ b \wedge \text{vec.dim } (\text{solution-set } A \ 0) > 0)$

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 function *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 $GJ-P = \text{Gauss-Jordan-PA } A$;
 $P\text{-times-}b = \text{fst}(GJ-P) *v b$;
 $\text{rank-}A = (\text{if } A = 0 \text{ then } 0 \text{ else to-nat } (GREATEST a. \text{row } a \text{ (snd } GJ-P) \neq 0) + 1)$;
 $\text{consistent-}Ab = (\text{rank-}A \geq (\text{if } (\exists a. (P\text{-times-}b) \$ a \neq 0) \text{ then } (\text{to-nat } (GREATEST a. (P\text{-times-}b) \$ a \neq 0) + 1) \text{ else } 0))$;
 $GJ\text{-transpose} = \text{Gauss-Jordan-PA } (\text{transpose } A)$;
 $\text{basis} = \{\text{row } i \text{ (fst } GJ\text{-transpose)} \mid i. \text{to-nat } i \geq \text{rank-}A\}$
 in (if consistent- Ab then *Some* (*solve-consistent-rref* (*snd* $GJ-P$)
 $P\text{-times-}b, \text{basis}$) else *None*))
 ⟨*proof*⟩

lemma *consistent-imp-is-solution-solve*:

fixes $A::'a::\{\text{field}\} \sim \text{cols}::\{\text{mod-type}\} \sim \text{rows}::\{\text{mod-type}\}$

assumes *con*: consistent A b

shows *is-solution* (*fst* (*the* (*solve* A b))) A b

⟨*proof*⟩

corollary *consistent-eq-solution-solve*:
fixes $A::'a::\{field\} \wedge cols::\{mod-type\} \wedge rows::\{mod-type\}$
shows $consistent\ A\ b = is-solution\ (fst\ (the\ (solve\ A\ b)))\ A\ b$
 $\langle proof \rangle$

lemma *inconsistent-imp-solve-eq-none*:
fixes $A::'a::\{field\} \wedge cols::\{mod-type\} \wedge rows::\{mod-type\}$
assumes $con: inconsistent\ A\ b$
shows $solve\ A\ b = None\ \langle proof \rangle$

corollary *inconsistent-eq-solve-eq-none*:
fixes $A::'a::\{field\} \wedge cols::\{mod-type\} \wedge rows::\{mod-type\}$
shows $inconsistent\ A\ b = (solve\ A\ b = None)$
 $\langle proof \rangle$

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 $A::'a::\{field\} \wedge cols::\{mod-type\} \wedge rows::\{mod-type\}$
assumes $con: consistent\ A\ b$
shows $solution-set\ A\ b = \{fst\ (the\ (solve\ A\ b))\} + vec.span\ (snd\ (the\ (solve\ A\ b)))$
 $\langle proof \rangle$

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

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 $bit \rightarrow (SML)\ Bool.bool$


```
| constant 0::bit  $\rightarrow$  (SML) false
| constant 1::bit  $\rightarrow$  (SML) true
```

code-printing

```
type-constructor bit  $\rightarrow$  (Haskell) Bool
| constant 0::bit  $\rightarrow$  (Haskell) False
| constant 1::bit  $\rightarrow$  (Haskell) True
| class-instance bit :: HOL.equal  $\Rightarrow$  (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 :: 'an::{finite, enum}  $\Rightarrow$  'a list
  where vec-to-list A = map (( $\$$ ) A) (enum-class.enum::n list)
```

```
definition matrix-to-list-of-list :: 'an::{finite, enum}m::{finite, enum}  $\Rightarrow$  '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  $\Rightarrow$  'an::{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 => 'aⁿ::{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],[0,1,0,0,0],[0,0,0,0,0],[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*⁵²)))
value *matrix-to-list-of-list* (*Gauss-Jordan* (*list-of-list-to-matrix* ([[1,0,0,1,1],[1,0,1,1,1]]::*bit*⁵²)))
value (*reduced-row-echelon-form-upt-k* (*list-of-list-to-matrix* ([[1,0,8],[0,1,9],[0,0,0]]::*real*³³)))
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*³²)))
value *DIM*(*real*⁵)
value *vec.dimension* (*TYPE*(*bit*)) (*TYPE*(5))
value *vec.dimension* (*TYPE*(*real*)) (*TYPE*(2))

value *DIM*(*real*⁵⁴)

```

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]]::real54)
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]]::real54))
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]]::real54)
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]]::real54))
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]]::real54)
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]]::real54))
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]]::complex32)

```

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]]::real77)
      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]]::real77) 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]]::real77)
      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]]::real77)
      in matrix-to-list-of-list (the (inverse-matrix A))
value let A=(list-of-list-to-matrix [[1,1,1,1,1,1],[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]]::real77)
      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]]::complex33)
      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]]::real55
in det A)
value det (list-of-list-to-matrix ([[1,0,0],[0,1,0],[0,0,1]]))::real33)
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]]))::rat55)

```

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]]))::real46
      in vec-to-list' (basis-null-space A)

```

value let $A = (\text{list-of-list-to-matrix } ([[3,4,0,7],[1,-5,2,-2],[-1,4,0,3],[1,-1,2,2]])::\text{real}^4^4)$
in $\text{vec-to-list}' (\text{basis-null-space } A)$

value let $A = (\text{list-of-list-to-matrix } ([[1,3,-2,0,2,0],[2,6,-5,-2,4,-3],[0,0,5,10,0,15],[2,6,0,8,4,18]])::\text{real}^4^6)$
in $\text{vec-to-list}' (\text{basis-row-space } A)$

value let $A = (\text{list-of-list-to-matrix } ([[3,4,0,7],[1,-5,2,-2],[-1,4,0,3],[1,-1,2,2]])::\text{real}^4^4)$
in $\text{vec-to-list}' (\text{basis-row-space } A)$

value let $A = (\text{list-of-list-to-matrix } ([[1,3,-2,0,2,0],[2,6,-5,-2,4,-3],[0,0,5,10,0,15],[2,6,0,8,4,18]])::\text{real}^6^4)$
in $\text{vec-to-list}' (\text{basis-col-space } A)$

value let $A = (\text{list-of-list-to-matrix } ([[3,4,0,7],[1,-5,2,-2],[-1,4,0,3],[1,-1,2,2]])::\text{real}^4^4)$
in $\text{vec-to-list}' (\text{basis-col-space } A)$

value let $A = (\text{list-of-list-to-matrix } ([[1,3,-2,0,2,0],[2,6,-5,-2,4,-3],[0,0,5,10,0,15],[2,6,0,8,4,18]])::\text{real}^6^4)$
in $\text{vec-to-list}' (\text{basis-left-null-space } A)$

value let $A = (\text{list-of-list-to-matrix } ([[3,4,0,7],[1,-5,2,-2],[-1,4,0,3],[1,-1,2,2]])::\text{real}^4^4)$
in $\text{vec-to-list}' (\text{basis-left-null-space } A)$

14.2.5 Consistency and inconsistency

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

value $\text{independent-and-consistent } (\text{list-of-list-to-matrix } ([[1,0,0],[0,1,0],[0,0,1],[0,0,0],[0,0,0]])::\text{real}^3^5)$
 $(\text{list-to-vec}([2,3,4,0,0])::\text{real}^5)$

value $\text{consistent } (\text{list-of-list-to-matrix } ([[1,0,0],[0,1,0],[0,0,1],[0,0,0],[0,0,0]])::\text{real}^3^5)$
 $(\text{list-to-vec}([2,3,4,0,0])::\text{real}^5)$

value $\text{inconsistent } (\text{list-of-list-to-matrix } ([[1,0,0],[0,1,0],[3,0,1],[0,7,0],[0,0,9]])::\text{real}^3^5)$
 $(\text{list-to-vec}([2,0,4,0,0])::\text{real}^5)$

value $\text{dependent-and-consistent } (\text{list-of-list-to-matrix } ([[1,0,0],[0,1,0]])::\text{real}^3^2)$
 $(\text{list-to-vec}([3,4])::\text{real}^2)$

value $\text{independent-and-consistent } (\text{mat } 1::\text{real}^3^3) (\text{list-to-vec}([3,4,5])::\text{real}^3)$

14.2.6 Solving systems of linear equations

Examples on solving linear systems.

definition $\text{print-result-solve}$

where $\text{print-result-solve } A = (\text{if } A = \text{None} \text{ then } \text{None} \text{ else } \text{Some } (\text{vec-to-list } (\text{fst } (\text{the } A)), \text{vec-to-list}' (\text{snd } (\text{the } A))))$

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

```

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  $\Rightarrow$  'a iarray  $\Rightarrow$  'a iarray
  where plus-iarray A B = IArray.of-fun ( $\lambda n. A !! n + B !! n$ ) (IArray.length A)
instance <proof>
end

```

```

instantiation iarray :: (minus) minus
begin
definition minus-iarray :: 'a iarray  $\Rightarrow$  'a iarray  $\Rightarrow$  'a iarray
  where minus-iarray A B = IArray.of-fun ( $\lambda n. A !! n - B !! n$ ) (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 :: 'an::{mod-type} ⇒ 'a iarray
  where vec-to-iarray A = IArray.of-fun (λi. A $ (from-nat i)) (CARD('n))
```

```
definition iarray-to-vec :: 'a iarray ⇒ 'an::{mod-type}
  where iarray-to-vec A = (χ i. A !! (to-nat i))
```

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

```
lemma vec-to-iarray-nth':
  fixes A::'an::{mod-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::'an::{mod-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::'an::{mod-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* :: $'a\ ^n::\{mod-type\} \ ^m::\{mod-type\} \Rightarrow 'a\ iarray\ iarray$

where $matrix-to-iarray\ A = IArray\ (map\ (vec-to-iarray\ \circ\ ((\$)\ A)\ \circ\ (from-nat::nat\ \Rightarrow\ 'm))\ [0..<CARD('m)])$

definition *iarray-to-matrix* :: $'a\ iarray\ iarray \Rightarrow 'a\ ^n::\{mod-type\} \ ^m::\{mod-type\}$

where $iarray-to-matrix\ A = (\chi\ i\ j.\ A\ !!\ (to-nat\ i)\ !!\ (to-nat\ j))$

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*: $\forall i.\ vec-to-iarray\ (A\ \$\ i) = f\ (to-nat\ i)$

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 $map\ (\lambda x.\ vec-to-iarray\ (A\ \$\ from-nat\ x))\ [0..<CARD('rows)]\ !\ to-nat\ i = vec-to-iarray\ (A\ \$\ i)$

<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 $A::'a \sim \text{columns}::\{\text{mod-type}\} \sim \text{rows}::\{\text{mod-type}\}$
shows $\text{iarray-to-matrix} (\text{matrix-to-iarray } A) = A$ $\langle \text{proof} \rangle$

16.2 Definition of operations over matrices implemented by iarrays

definition $\text{mult-iarray} :: 'a::\{\text{times}\} \text{iarray} \Rightarrow 'a \Rightarrow 'a \text{iarray}$
where $\text{mult-iarray } A \ q = \text{IArray.of-fun } (\lambda n. q * A!!n) (\text{IArray.length } A)$

definition $\text{row-iarray} :: \text{nat} \Rightarrow 'a \text{iarray} \text{iarray} \Rightarrow 'a \text{iarray}$
where $\text{row-iarray } k \ A = A !! k$

definition $\text{column-iarray} :: \text{nat} \Rightarrow 'a \text{iarray} \text{iarray} \Rightarrow 'a \text{iarray}$
where $\text{column-iarray } k \ A = \text{IArray.of-fun } (\lambda m. A !! m !! k) (\text{IArray.length } A)$

definition $\text{nrows-iarray} :: 'a \text{iarray} \text{iarray} \Rightarrow \text{nat}$
where $\text{nrows-iarray } A = \text{IArray.length } A$

definition $\text{ncols-iarray} :: 'a \text{iarray} \text{iarray} \Rightarrow \text{nat}$
where $\text{ncols-iarray } A = \text{IArray.length } (A!!0)$

definition $\text{rows-iarray } A = \{\text{row-iarray } i \ A \mid i. i \in \{..<\text{nrows-iarray } A\}\}$

definition $\text{columns-iarray } A = \{\text{column-iarray } i \ A \mid i. i \in \{..<\text{ncols-iarray } A\}\}$

definition $\text{tabulate2} :: \text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \Rightarrow \text{nat} \Rightarrow 'a) \Rightarrow 'a \text{iarray} \text{iarray}$
where $\text{tabulate2 } m \ n \ f = \text{IArray.of-fun } (\lambda i. \text{IArray.of-fun } (f \ i) \ n) \ m$

definition $\text{transpose-iarray} :: 'a \text{iarray} \text{iarray} \Rightarrow 'a \text{iarray} \text{iarray}$
where $\text{transpose-iarray } A = \text{tabulate2 } (\text{ncols-iarray } A) (\text{nrows-iarray } A) (\lambda a \ b. A!!b!!a)$

definition $\text{matrix-matrix-mult-iarray} :: 'a::\{\text{times}, \text{comm-monoid-add}\} \text{iarray} \text{iarray} \Rightarrow 'a \text{iarray} \text{iarray} \Rightarrow 'a \text{iarray} \text{iarray} \Rightarrow 'a \text{iarray} \text{iarray}$ (**infixl** $**i$ 70)

where $A **i \ B = \text{tabulate2 } (\text{nrows-iarray } A) (\text{ncols-iarray } B) (\lambda i \ j. \text{sum } (\lambda k. ((A!!i)!!k) * ((B!!k)!!j)) \{0..<\text{ncols-iarray } A\})$

definition $\text{matrix-vector-mult-iarray} :: 'a::\{\text{semiring-1}\} \text{iarray} \text{iarray} \Rightarrow 'a \text{iarray} \Rightarrow 'a \text{iarray}$ (**infixl** $*iv$ 70)

where $A *iv \ x = \text{IArray.of-fun } (\lambda i. \text{sum } (\lambda j. ((A!!i)!!j) * (x!!j)) \{0..<\text{IArray.length } x\}) (\text{nrows-iarray } A)$

definition $\text{vector-matrix-mult-iarray} :: 'a::\{\text{semiring-1}\} \text{iarray} \Rightarrow 'a \text{iarray} \text{iarray} \Rightarrow 'a \text{iarray}$ (**infixl** $v*i$ 70)

where $x v*i \ A = \text{IArray.of-fun } (\lambda j. \text{sum } (\lambda i. ((A!!i)!!j) * (x!!i)) \{0..<\text{IArray.length } x\}) (\text{ncols-iarray } A)$

definition $\text{mat-iarray} :: 'a::\{\text{zero}\} \Rightarrow \text{nat} \Rightarrow 'a \text{iarray} \text{iarray}$

where $\text{mat-iarray } k \ n = \text{tabulate2 } n \ n (\lambda i \ j. \text{if } i = j \ \text{then } k \ \text{else } 0)$

definition *is-zero-iarray* :: 'a::{zero} iarray \Rightarrow bool
 where *is-zero-iarray* A = IArray.all ($\lambda i. A !! i = 0$) (IArray[0..

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'*: $vec\text{-to-iarray} (row\ i\ A) = (matrix\text{-to-iarray}\ A) !! (to\text{-nat}\ i)$
 ⟨*proof*⟩

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

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

lemma *column-iarray-nth*:
assumes *i*: $i < nrows\text{-iarray}\ A$
shows $column\text{-iarray}\ j\ A !! i = A !! i !! j$
 ⟨*proof*⟩

lemma *vec-to-iarray-rows*: $vec\text{-to-iarray}' (rows\ A) = rows\text{-iarray} (matrix\text{-to-iarray}\ A)$
 ⟨*proof*⟩

lemma *vec-to-iarray-columns*: $vec\text{-to-iarray}' (columns\ A) = columns\text{-iarray} (matrix\text{-to-iarray}\ A)$
 ⟨*proof*⟩

16.3 Definition of elementary operations

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

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

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

definition *interchange-columns-iarray* :: $'a\ iarray\ iarray \Rightarrow nat \Rightarrow nat \Rightarrow 'a\ iarray\ iarray$
where *interchange-columns-iarray* *A a b* = *tabulate2* $(nrows\text{-iarray}\ A) (ncols\text{-iarray}\ A) (\lambda i\ j. if\ j = a\ then\ A !! i !! b\ else\ if\ j = b\ then\ A !! i !! a\ else\ A !! i !! j)$

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) ($\lambda i j.$ 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) ($\lambda i j.$ 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 $A::'a::\{\text{semiring-1}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$
shows $\text{matrix-to-iarray} (\text{interchange-rows } A \ i \ j) = \text{interchange-rows-iarray} (\text{matrix-to-iarray } A) \ (to\text{-nat } i) \ (to\text{-nat } j)$
 $\langle\text{proof}\rangle$

lemma $\text{matrix-to-iarray-mult-row}[\text{code-unfold}]$:
fixes $A::'a::\{\text{semiring-1}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$
shows $\text{matrix-to-iarray} (\text{mult-row } A \ i \ q) = \text{mult-row-iarray} (\text{matrix-to-iarray } A) \ (to\text{-nat } i) \ q$
 $\langle\text{proof}\rangle$

lemma $\text{matrix-to-iarray-row-add}[\text{code-unfold}]$:
fixes $A::'a::\{\text{semiring-1}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$
shows $\text{matrix-to-iarray} (\text{row-add } A \ i \ j \ q) = \text{row-add-iarray} (\text{matrix-to-iarray } A) \ (to\text{-nat } i) \ (to\text{-nat } j) \ q$
 $\langle\text{proof}\rangle$

lemma $\text{matrix-to-iarray-interchange-columns}[\text{code-unfold}]$:
fixes $A::'a::\{\text{semiring-1}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$
shows $\text{matrix-to-iarray} (\text{interchange-columns } A \ i \ j) = \text{interchange-columns-iarray} (\text{matrix-to-iarray } A) \ (to\text{-nat } i) \ (to\text{-nat } j)$
 $\langle\text{proof}\rangle$

lemma $\text{matrix-to-iarray-mult-columns}[\text{code-unfold}]$:
fixes $A::'a::\{\text{semiring-1}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$
shows $\text{matrix-to-iarray} (\text{mult-column } A \ i \ q) = \text{mult-column-iarray} (\text{matrix-to-iarray } A) \ (to\text{-nat } i) \ q$
 $\langle\text{proof}\rangle$

lemma $\text{matrix-to-iarray-column-add}[\text{code-unfold}]$:
fixes $A::'a::\{\text{semiring-1}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$
shows $\text{matrix-to-iarray} (\text{column-add } A \ i \ j \ q) = \text{column-add-iarray} (\text{matrix-to-iarray } A) \ (to\text{-nat } i) \ (to\text{-nat } j) \ q$
 $\langle\text{proof}\rangle$

end

17 Gauss Jordan algorithm over nested IArrays

theory $\text{Gauss-Jordan-IArrays}$
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* $A\ i = \text{the } (\text{List.find } (\lambda x. A\ !!\ x \neq 0))\ [i..<IArray.length\ A])$

definition *least-non-zero-position-of-vector* $A = \text{least-non-zero-position-of-vector-from-index } A\ 0$

definition *vector-all-zero-from-index* $:: (\text{nat} \times 'a::\{\text{zero}\}\ \text{iarray}) \Rightarrow \text{bool}$

where *vector-all-zero-from-index* $A' = (\text{let } i = \text{fst } A';\ A = (\text{snd } A')\ \text{in } IArray.all\ (\lambda x. A\ !!\ x = 0)\ (IArray\ [i..<(IArray.length\ A)]))$

definition *Gauss-Jordan-in-ij-iarrays* $:: 'a::\{\text{field}\}\ \text{iarray}\ \text{iarray} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow 'a\ \text{iarray}\ \text{iarray}$

where *Gauss-Jordan-in-ij-iarrays* $A\ i\ j = (\text{let } n = \text{least-non-zero-position-of-vector-from-index}\ (\text{column-iarray}\ j\ A)\ i;$

interchange-A $= \text{interchange-rows-iarray } A\ i\ n;$

$A' = \text{mult-row-iarray } \text{interchange-A } i\ (1/\text{interchange-A}\ !!\ i!!\ j)$

in $IArray.of-fun\ (\lambda s. \text{if } s = i\ \text{then } A' \ !!\ s\ \text{else } \text{row-add-iarray } A' \ s\ i\ (-\ \text{interchange-A}\ !!\ s!!\ j)\ !!\ s)\ (\text{nrows-iarray } A))$

definition *Gauss-Jordan-column-k-iarrays* $:: (\text{nat} \times 'a::\{\text{field}\}\ \text{iarray}\ \text{iarray}) \Rightarrow \text{nat} \Rightarrow (\text{nat} \times 'a\ \text{iarray}\ \text{iarray})$

where *Gauss-Jordan-column-k-iarrays* $A' \ k = (\text{let } A = (\text{snd } A');\ i = (\text{fst } A')\ \text{in}\ \text{if } ((\text{vector-all-zero-from-index}\ (i,\ (\text{column-iarray}\ k\ A)))) \vee i = (\text{nrows-iarray } A)\ \text{then } (i, A)\ \text{else } (\text{Suc } i,\ (\text{Gauss-Jordan-in-ij-iarrays } A\ i\ k)))$

definition *Gauss-Jordan-upt-k-iarrays* $:: 'a::\{\text{field}\}\ \text{iarray}\ \text{iarray} \Rightarrow \text{nat} \Rightarrow 'a::\{\text{field}\}\ \text{iarray}\ \text{iarray}$

where *Gauss-Jordan-upt-k-iarrays* $A\ k = \text{snd } (\text{foldl } \text{Gauss-Jordan-column-k-iarrays}\ (0, A)\ [0..<\text{Suc } k])$

definition *Gauss-Jordan-iarrays* $:: 'a::\{\text{field}\}\ \text{iarray}\ \text{iarray} \Rightarrow 'a::\{\text{field}\}\ \text{iarray}\ \text{iarray}$

where *Gauss-Jordan-iarrays* $A = \text{Gauss-Jordan-upt-k-iarrays } A\ (\text{ncols-iarray } A - 1)$

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 $A::'a::\{\text{zero}\}\ ^n::\{\text{mod-type}\}$

shows $(\forall m \geq i. A\ \$\ m = 0) = (\text{vector-all-zero-from-index } (\text{to-nat } i,\ \text{vec-to-iarray } A))$

<proof>

lemma *matrix-vector-all-zero-from-index:*

fixes $A::'a::\{\text{zero}\}\ ^\text{columns}::\{\text{mod-type}\}\ ^\text{rows}::\{\text{mod-type}\}$

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

lemma *vec-to-iarray-least-non-zero-position-of-vector-from-index*:

fixes $A::'a::\{\text{zero}\} \wedge n::\{\text{mod-type}\}$

assumes *not-all-zero*: $\neg (\text{vector-all-zero-from-index } (\text{to-nat } i, \text{vec-to-iarray } A))$

shows *least-non-zero-position-of-vector-from-index* $(\text{vec-to-iarray } A) (\text{to-nat } i) = \text{to-nat } (\text{LEAST } n. A \$ n \neq 0 \wedge i \leq n)$
 ⟨proof⟩

corollary *vec-to-iarray-least-non-zero-position-of-vector-from-index'*:

fixes $A::'a::\{\text{zero}\} \wedge \text{cols}::\{\text{mod-type}\} \wedge \text{rows}::\{\text{mod-type}\}$

assumes *not-all-zero*: $\neg (\text{vector-all-zero-from-index } (\text{to-nat } i, \text{vec-to-iarray } (\text{column } j A)))$

shows *least-non-zero-position-of-vector-from-index* $(\text{vec-to-iarray } (\text{column } j A)) (\text{to-nat } i) = \text{to-nat } (\text{LEAST } n. A \$ n \$ j \neq 0 \wedge i \leq n)$
 ⟨proof⟩

corollary *vec-to-iarray-least-non-zero-position-of-vector-from-index''*:

fixes $A::'a::\{\text{zero}\} \wedge \text{cols}::\{\text{mod-type}\} \wedge \text{rows}::\{\text{mod-type}\}$

assumes *not-all-zero*: $\neg (\text{vector-all-zero-from-index } (\text{to-nat } j, \text{vec-to-iarray } (\text{row } i A)))$

shows *least-non-zero-position-of-vector-from-index* $(\text{vec-to-iarray } (\text{row } i A)) (\text{to-nat } j) = \text{to-nat } (\text{LEAST } n. A \$ i \$ n \neq 0 \wedge j \leq n)$
 ⟨proof⟩

lemma *matrix-to-iarray-Gauss-Jordan-in-ij*[code-unfold]:

fixes $A::'a::\{\text{field}\} \wedge \text{columns}::\{\text{mod-type}\} \wedge \text{rows}::\{\text{mod-type}\}$

assumes *not-all-zero*: $\neg (\text{vector-all-zero-from-index } (\text{to-nat } i, \text{vec-to-iarray } (\text{column } j A)))$

shows *matrix-to-iarray* $(\text{Gauss-Jordan-in-ij } A i j) = \text{Gauss-Jordan-in-ij-iarrays } (\text{matrix-to-iarray } A) (\text{to-nat } i) (\text{to-nat } j)$
 ⟨proof⟩

lemma *matrix-to-iarray-Gauss-Jordan-column-k-1*:

fixes $A::'a::\{\text{field}\} \wedge \text{columns}::\{\text{mod-type}\} \wedge \text{rows}::\{\text{mod-type}\}$

assumes $k: k < \text{ncols } A$

and $i: i \leq \text{nrows } A$

shows $(\text{fst } (\text{Gauss-Jordan-column-k } (i, A) k)) = \text{fst } (\text{Gauss-Jordan-column-k-iarrays } (i, \text{matrix-to-iarray } A) k)$
 ⟨proof⟩

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

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

Due to the assumptions presented in $\llbracket ?k < \text{ncols } ?A; ?i \leq \text{nrows } ?A \rrbracket$
 $\implies \text{matrix-to-iarray} (\text{snd} (\text{Gauss-Jordan-column-k} (?i, ?A) ?k)) = \text{snd} (\text{Gauss-Jordan-column-k-iarrays} (?i, \text{matrix-to-iarray } ?A) ?k)$, the following lemma must have three shows. The proof style is similar to $?k < \text{ncols } ?A \implies \text{reduced-row-echelon-form-upt-k} (\text{Gauss-Jordan-upt-k } ?A ?k) (\text{Suc } ?k)$

$?k < \text{ncols } ?A \implies \text{foldl Gauss-Jordan-column-k} (0, ?A) [0..<\text{Suc } ?k] =$
(if $\forall m. \text{is-zero-row-upt-k } m (\text{Suc } ?k) (\text{snd} (\text{foldl Gauss-Jordan-column-k} (0, ?A) [0..<\text{Suc } ?k]))$ then 0 else mod-type-class.to-nat (GREATEST $n. \neg \text{is-zero-row-upt-k } n (\text{Suc } ?k) (\text{snd} (\text{foldl Gauss-Jordan-column-k} (0, ?A) [0..<\text{Suc } ?k]))$) + 1, $\text{snd} (\text{foldl Gauss-Jordan-column-k} (0, ?A) [0..<\text{Suc } ?k]))$.

lemma *foldl-Gauss-Jordan-column-k-eq:*

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

lemma *matrix-to-iarray-Gauss-Jordan[code-unfold]:*

fixes $A::'a::\{\text{field}\} \sim \text{columns}::\{\text{mod-type}\} \sim \text{rows}::\{\text{mod-type}\}$
shows $\text{matrix-to-iarray} (\text{Gauss-Jordan } A) = \text{Gauss-Jordan-iarrays} (\text{matrix-to-iarray } A)$
<proof>

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

definition $\text{rank-iarray} :: 'a::\{\text{field}\} \text{ iarray iarray} \Rightarrow \text{nat}$

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

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 $A::'a::\{\text{field}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$
shows $\text{rank } A = \text{card } \{\text{vec-to-iarray } (\text{row } i \text{ (Gauss-Jordan } A)) \mid i. \neg \text{is-zero-iarray } (\text{vec-to-iarray } (\text{row } i \text{ (Gauss-Jordan } A)))\}$
 $\langle \text{proof} \rangle$

lemma *rank-eq-card-iarrays'*:
fixes $A::'a::\{\text{field}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$
shows $\text{rank } A = (\text{let } A' = (\text{Gauss-Jordan-iarrays } (\text{matrix-to-iarray } A)) \text{ in card } \{\text{row-iarray } (\text{to-nat } i) A' \mid i::'\text{rows}. \neg \text{is-zero-iarray } (A' !! (\text{to-nat } i))\})$
 $\langle \text{proof} \rangle$

lemma *rank-eq-card-iarrays-code*:
fixes $A::'a::\{\text{field}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$
shows $\text{rank } A = (\text{let } A' = (\text{Gauss-Jordan-iarrays } (\text{matrix-to-iarray } A)) \text{ in card } \{i::'\text{rows}. \neg \text{is-zero-iarray } (A' !! (\text{to-nat } i))\})$
 $\langle \text{proof} \rangle$

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

lemma *rank-iarrays-code*[code]:
 $\text{rank-iarray } A = \text{length } (\text{filter } (\lambda x. \neg \text{is-zero-iarray } x) (\text{IArray.list-of } (\text{Gauss-Jordan-iarrays } A)))$
 $\langle \text{proof} \rangle$

lemma *matrix-to-iarray-rank*[code-unfold]:
shows $\text{rank } A = \text{rank-iarray } (\text{matrix-to-iarray } A)$
 $\langle \text{proof} \rangle$

lemma *dim-null-space-iarray*[code-unfold]:
fixes $A::'a::\{\text{field}\}^{\wedge}\text{columns}::\{\text{mod-type}\}^{\wedge}\text{rows}::\{\text{mod-type}\}$
shows $\text{vec.dim } (\text{null-space } A) = \text{ncols-iarray } (\text{matrix-to-iarray } A) - \text{rank-iarray } (\text{matrix-to-iarray } A)$
 $\langle \text{proof} \rangle$

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

lemma *dim-row-space-iarray*[code-unfold]:
fixes $A::'a::\{\text{field}\} \wedge \text{columns}::\{\text{mod-type}\} \wedge \text{rows}::\{\text{mod-type}\}$
shows $\text{vec.dim} (\text{row-space } A) = \text{rank-iarray} (\text{matrix-to-iarray } A)$
<proof>

lemma *dim-left-null-space-space-iarray*[code-unfold]:
fixes $A::'a::\{\text{field}\} \wedge \text{columns}::\{\text{mod-type}\} \wedge \text{rows}::\{\text{mod-type}\}$
shows $\text{vec.dim} (\text{left-null-space } A) = \text{nrows-iarray} (\text{matrix-to-iarray } A) - \text{rank-iarray} (\text{matrix-to-iarray } A)$
<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* $A' i j =$
 $(\text{let } P = \text{fst } A'; A = \text{snd } A'; n = \text{least-non-zero-position-of-vector-from-index} (\text{column-iarray } j A) i; \text{interchange-A} = \text{interchange-rows-iarray } A i n;$
 $\text{interchange-P} = \text{interchange-rows-iarray } P i n; P' = \text{mult-row-iarray } \text{interchange-P}$
 $i (1 / \text{interchange-A} !! i !! j)$
 $\text{in } (\text{IArray.of-fun } (\lambda s. \text{if } s = i \text{ then } P' !! s \text{ else } \text{row-add-iarray } P' s i (- \text{interchange-A} !! s !! j) !! s) (\text{nrows-iarray } A), \text{Gauss-Jordan-in-ij-iarrays } A i j))$

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

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

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 (λs. if s = i then P' !! s else row-add-iarray P' s i (- interchange-A !! s !! j) !! s) (nrows-iarray A),
 (IArray.of-fun (λs. 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 $i: i \leq \text{nrows } A$ **and** $k: k < \text{ncols } A$

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 $i: i \leq \text{nrows } A$ **and** $k: k < \text{ncols } A$

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 $k < \text{ncols } A$

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 $fst (snd (foldl Gauss-Jordan-column-k-PA (mat 1, 0, A) [0..<Suc k])) = fst (snd (foldl Gauss-Jordan-column-k-iarrays-PA (matrix-to-iarray 1 (nrows-iarray (matrix-to-iarray A)), 0, matrix-to-iarray A) [0..<Suc k]))$

and $fst (snd (foldl Gauss-Jordan-column-k-PA (mat 1, 0, A) [0..<Suc k])) \leq \text{nrows } A$
<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 $A::'a::\{field\} \wedge \text{cols}::\{mod-type\} \wedge \text{rows}::\{mod-type\}$

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 $\text{snd} (\text{snd} (\text{Gauss-Jordan-column-k-iarrays-PA} (P, i, A) k)) = \text{snd} (\text{Gauss-Jordan-column-k-iarrays} (i, A) k)$
<proof>

lemma *fst-snd-Gauss-Jordan-column-k-iarrays-PA-eq*:
shows $\text{fst} (\text{snd} (\text{Gauss-Jordan-column-k-iarrays-PA} (P, i, A) k)) = \text{fst} (\text{Gauss-Jordan-column-k-iarrays} (i, A) k)$
<proof>

lemma *foldl-Gauss-Jordan-column-k-iarrays-eq*:
 $\text{snd} (\text{foldl} \text{Gauss-Jordan-column-k-iarrays-PA} (B, 0, A) [0..<k]) = \text{foldl} \text{Gauss-Jordan-column-k-iarrays} (0, A) [0..<k]$
<proof>

lemma *snd-Gauss-Jordan-upt-k-iarrays-PA*:
shows $\text{snd} (\text{Gauss-Jordan-upt-k-iarrays-PA} A k) = (\text{Gauss-Jordan-upt-k-iarrays} A k)$
<proof>

lemma *snd-Gauss-Jordan-iarrays-PA-eq*: $\text{snd} (\text{Gauss-Jordan-iarrays-PA} A) = \text{Gauss-Jordan-iarrays} A$
<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
 $= (\text{let } GJ = \text{Gauss-Jordan-iarrays-PA} A;$
 $\text{rank-}A = \text{length} [x \leftarrow \text{IArray.list-of} (\text{snd } GJ) . \neg \text{is-zero-iarray } x]$
 $\text{in set } (\text{map } (\lambda i. \text{row-iarray } i (\text{fst } GJ)) [(rank-A)..<(nrows-iarray A)]))$

definition *basis-null-space-iarrays* A

$$= (\text{let } GJ = \text{Gauss-Jordan-iarrays-PA } (\text{transpose-iarray } A);$$

$$\text{rank-}A = \text{length } [x \leftarrow \text{IArray.list-of } (\text{snd } GJ) . \neg \text{is-zero-iarray } x]$$

$$\text{in set } (\text{map } (\lambda i. \text{row-iarray } i (\text{fst } GJ)) [(\text{rank-}A)..<(\text{ncols-iarray } A)]))$$
definition *basis-row-space-iarrays* $A =$

$$(\text{let } GJ = \text{Gauss-Jordan-iarrays } A;$$

$$\text{rank-}A = \text{length } [x \leftarrow \text{IArray.list-of } (GJ) . \neg \text{is-zero-iarray } x]$$

$$\text{in set } (\text{map } (\lambda i. \text{row-iarray } i (GJ)) [0..<\text{rank-}A]))$$
definition *basis-col-space-iarrays* $A = \text{basis-row-space-iarrays } (\text{transpose-iarray } A)$

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 $A::'a::\{\text{field}\} \sim \text{cols}::\{\text{mod-type}\} \sim \text{rows}::\{\text{mod-type}\}$
shows *basis-null-space-iarrays* (*matrix-to-iarray* A)
 $= \text{set } (\text{map } (\lambda i. \text{row-iarray } i (\text{fst } (\text{Gauss-Jordan-iarrays-PA } (\text{transpose-iarray } (\text{matrix-to-iarray } A)))))) [(\text{rank-iarray } (\text{matrix-to-iarray } A)..<(\text{ncols-iarray } (\text{matrix-to-iarray } A)))]$
 $\langle \text{proof} \rangle$

lemma *basis-row-space-iarrays-eq*:
fixes $A::'a::\{\text{field}\} \sim \text{cols}::\{\text{mod-type}\} \sim \text{rows}::\{\text{mod-type}\}$
shows *basis-row-space-iarrays* (*matrix-to-iarray* A) $= \text{set } (\text{map } (\lambda i. \text{row-iarray } i (\text{Gauss-Jordan-iarrays } (\text{matrix-to-iarray } A))) [0..<(\text{rank-iarray } (\text{matrix-to-iarray } A))])$
 $\langle \text{proof} \rangle$

lemma *basis-left-null-space-iarrays-eq*:
fixes $A::'a::\{\text{field}\} \sim \text{cols}::\{\text{mod-type}\} \sim \text{rows}::\{\text{mod-type}\}$
shows *basis-left-null-space-iarrays* (*matrix-to-iarray* A) $= \text{basis-null-space-iarrays } (\text{transpose-iarray } (\text{matrix-to-iarray } A))$
 $\langle \text{proof} \rangle$

19.2 Code equations

lemma *vec-to-iarray-basis-null-space[code-unfold]*:
fixes $A::'a::\{\text{field}\} \sim \text{cols}::\{\text{mod-type}\} \sim \text{rows}::\{\text{mod-type}\}$
shows *vec-to-iarray'* (*basis-null-space* A) $= \text{basis-null-space-iarrays } (\text{matrix-to-iarray } A)$
 $\langle \text{proof} \rangle$

corollary *vec-to-iarray-basis-left-null-space[code-unfold]*:
fixes $A::'a::\{\text{field}\} \sim \text{cols}::\{\text{mod-type}\} \sim \text{rows}::\{\text{mod-type}\}$
shows *vec-to-iarray'* (*basis-left-null-space* A) $= \text{basis-left-null-space-iarrays } (\text{matrix-to-iarray } A)$
 $\langle \text{proof} \rangle$

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

fixes $A::'a::\{field\} \sim cols::\{mod-type\} \sim rows::\{mod-type\}$
shows $vec-to-iarray' (basis-row-space A) = basis-row-space-iarrays (matrix-to-iarray A)$
 $\langle proof \rangle$

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

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 $greatest-not-zero :: 'a::\{zero\} iarray \Rightarrow nat$
where $greatest-not-zero A = the (List.find (\lambda n. A !! n \neq 0) (rev [0..<IArray.length A]))$

lemma $vec-to-iarray-exists:$

shows $(\exists b. A \$ b \neq 0) = IArray.exists (\lambda b. (vec-to-iarray A) !! b \neq 0) (IArray[0..<IArray.length (vec-to-iarray A)])$
 $\langle proof \rangle$

corollary $vec-to-iarray-exists':$

shows $(\exists b. A \$ b \neq 0) = IArray.exists (\lambda b. (vec-to-iarray A) !! b \neq 0) (IArray (rev [0..<IArray.length (vec-to-iarray A)]))$
 $\langle proof \rangle$

lemma $not-is-zero-iarray-eq-iff: (\exists b. A \$ b \neq 0) = (\neg is-zero-iarray (vec-to-iarray A))$
 $\langle proof \rangle$

lemma $vec-to-iarray-greatest-not-zero:$

assumes $ex-b: (\exists b. A \$ b \neq 0)$

shows $greatest-not-zero (vec-to-iarray A) = to-nat (GREATEST b. A \$ b \neq 0)$
 $\langle proof \rangle$

20.2 Consistency and inconsistency

definition *consistent-iarrays* $A\ b = (\text{let } GJ = \text{Gauss-Jordan-iarrays-PA } A;$
 $\text{rank-}A = \text{length } [x \leftarrow \text{IArray.list-of } (\text{snd } GJ) . \neg$
 $\text{is-zero-iarray } x];$

$P\text{-mult-}b = \text{fst}(GJ) *iv\ b$
 $\text{in } (\text{rank-}A \geq (\text{if } (\neg \text{is-zero-iarray } P\text{-mult-}b)$
 $\text{then } (\text{greatest-not-zero } P\text{-mult-}b + 1) \text{ else } 0)))$

definition *inconsistent-iarrays* $A\ b = (\neg \text{consistent-iarrays } A\ b)$

lemma *matrix-to-iarray-consistent*[code]: $\text{consistent } A\ b = \text{consistent-iarrays } (\text{matrix-to-iarray } A)$ $(\text{vec-to-iarray } b)$
 <proof>

lemma *matrix-to-iarray-inconsistent*[code]: $\text{inconsistent } A\ b = \text{inconsistent-iarrays } (\text{matrix-to-iarray } A)$ $(\text{vec-to-iarray } b)$
 <proof>

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

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

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

lemma *iarray-exhaust2*:
 $(xs = ys) = (\text{IArray.list-of } xs = \text{IArray.list-of } ys)$
 <proof>

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

20.3 Independence and dependence

definition *independent-and-consistent-iarrays A b* =
 $(\text{let } GJ = \text{Gauss-Jordan-iarrays-PA } A;$
 $\text{rank-}A = \text{length } [x \leftarrow \text{IArray.list-of } (\text{snd } GJ) . \neg \text{is-zero-iarray } x];$
 $P\text{-mult-}b = \text{fst } GJ *iv \ b;$
 $\text{consistent-}A = ((\text{if } \neg \text{is-zero-iarray } P\text{-mult-}b \text{ then greatest-not-zero } P\text{-mult-}b$
 $+ 1 \text{ else } 0) \leq \text{rank-}A);$
 $\text{dim-solution-set} = \text{ncols-iarray } A - \text{rank-}A$
 $\text{in } \text{consistent-}A \wedge \text{dim-solution-set} = 0)$

definition *dependent-and-consistent-iarrays A b* =
 $(\text{let } GJ = \text{Gauss-Jordan-iarrays-PA } A;$
 $\text{rank-}A = \text{length } [x \leftarrow \text{IArray.list-of } (\text{snd } GJ) . \neg \text{is-zero-iarray } x];$
 $P\text{-mult-}b = \text{fst } GJ *iv \ b;$
 $\text{consistent-}A = ((\text{if } \neg \text{is-zero-iarray } P\text{-mult-}b \text{ then greatest-not-zero } P\text{-mult-}b$
 $+ 1 \text{ else } 0) \leq \text{rank-}A);$
 $\text{dim-solution-set} = \text{ncols-iarray } A - \text{rank-}A$
 $\text{in } \text{consistent-}A \wedge \text{dim-solution-set} > 0)$

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*)
 $\langle \text{proof} \rangle$

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*)
 $\langle \text{proof} \rangle$

20.4 Solve a system of equations over nested IArrays

definition *solve-system-iarrays A b* = $(\text{let } A' = \text{Gauss-Jordan-iarrays-PA } A \text{ in } (\text{snd } A', \text{fst } A' *iv \ b))$

lemma *matrix-to-iarray-fst-solve-system*: $\text{matrix-to-iarray} (\text{fst } (\text{solve-system } A \ b)) = \text{fst } (\text{solve-system-iarrays} (\text{matrix-to-iarray } A) (\text{vec-to-iarray } b))$
 $\langle \text{proof} \rangle$

lemma *vec-to-iarray-snd-solve-system*: $\text{vec-to-iarray} (\text{snd } (\text{solve-system } A \ b)) = \text{snd } (\text{solve-system-iarrays} (\text{matrix-to-iarray } A) (\text{vec-to-iarray } b))$
 $\langle \text{proof} \rangle$

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

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

lemma *pair-vec-vecset-solve*^[code-unfold]:
shows *pair-vec-vecset* (*solve* $A\ b$) = *solve-iarrays* (*matrix-to-iarray* A) (*vec-to-iarray* b)
 $\langle \text{proof} \rangle$

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* $A\ i\ j = (\text{let } n = \text{least-non-zero-position-of-vector-from-index}$
 $(\text{column-iarray } j\ A)\ i$
 $\text{in } (\text{if } i = n \text{ then } 1 / A\ !!\ i\ !!\ j \text{ else } -1 / A\ !!\ n\ !!\ j, \text{Gauss-Jordan-in-ij-iarrays}$
 $A\ i\ j))$

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

definition *Gauss-Jordan-upt-k-det-P-iarrays* $A\ k = (\text{let foldl} = \text{foldl Gauss-Jordan-column-k-det-P-iarrays}$

(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 (λs. 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 $i \leq \text{nrows } A$ **and** $k < \text{ncols } A$
shows $\text{matrix-to-iarray} (\text{snd} (\text{snd} (\text{Gauss-Jordan-column-k-det-P } (n, i, A) k))) =$
 $\text{snd} (\text{snd} (\text{Gauss-Jordan-column-k-det-P-iarrays } (n, i, \text{matrix-to-iarray } A) k))$
 $\langle \text{proof} \rangle$

lemma *fst-snd-Gauss-Jordan-column-k-det-P-le-nrows*:
assumes $i \leq \text{nrows } A$
shows $\text{fst} (\text{snd} (\text{Gauss-Jordan-column-k-det-P } (n, i, A) k)) \leq \text{nrows } A$
 $\langle \text{proof} \rangle$

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 $k < \text{ncols } A$
shows $\text{matrix-to-iarray-fst-Gauss-Jordan-upt-k-det-P} : \text{fst} (\text{Gauss-Jordan-upt-k-det-P } A k) =$
 $\text{fst} (\text{Gauss-Jordan-upt-k-det-P-iarrays } (\text{matrix-to-iarray } A) k)$
and $\text{matrix-to-iarray-snd-Gauss-Jordan-upt-k-det-P} : \text{matrix-to-iarray} (\text{snd} (\text{Gauss-Jordan-upt-k-det-P } A k)) =$
 $\text{snd} (\text{Gauss-Jordan-upt-k-det-P-iarrays } (\text{matrix-to-iarray } A) k)$
and $\text{fst} (\text{snd} (\text{foldl Gauss-Jordan-column-k-det-P } (1, 0, A) [0..<\text{Suc } k])) \leq \text{nrows } A$
and $\text{fst} (\text{snd} (\text{foldl Gauss-Jordan-column-k-det-P-iarrays } (1, 0, \text{matrix-to-iarray } A) [0..<\text{Suc } k])) =$
 $\text{fst} (\text{snd} (\text{foldl Gauss-Jordan-column-k-det-P } (1, 0, A) [0..<\text{Suc } k]))$
 $\langle \text{proof} \rangle$

lemma *matrix-to-iarray-fst-Gauss-Jordan-det-P*:
shows $\text{fst} (\text{Gauss-Jordan-det-P } A) = \text{fst} (\text{Gauss-Jordan-det-P-iarrays } (\text{matrix-to-iarray } A))$
 $\langle \text{proof} \rangle$

lemma *matrix-to-iarray-snd-Gauss-Jordan-det-P*:
shows $\text{matrix-to-iarray} (\text{snd} (\text{Gauss-Jordan-det-P } A)) = \text{snd} (\text{Gauss-Jordan-det-P-iarrays } (\text{matrix-to-iarray } A))$
 $\langle \text{proof} \rangle$

21.3 Code equations

definition $\text{det-iarrays } A = (\text{let } A' = \text{Gauss-Jordan-det-P-iarrays } A \text{ in prod-list } (\text{map } (\lambda i. (\text{snd } A') !! i !! i) [0..<\text{nrows-iarray } A]) / \text{fst } A')$

lemma *matrix-to-iarray-det[code-unfold]*:
fixes $A :: 'a :: \{\text{field}\} \sim 'n :: \{\text{mod-type}\} \sim 'n :: \{\text{mod-type}\}$
shows $\text{det } A = \text{det-iarrays} (\text{matrix-to-iarray } A)$
 $\langle \text{proof} \rangle$

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* $A = (\text{rank-iarray } A = \text{nrows-iarray } A)$

definition *inverse-matrix-iarray* $A = (\text{if invertible-iarray } A \text{ then Some}(\text{fst}(\text{Gauss-Jordan-iarrays-PA } A)) \text{ else None})$

definition *matrix-to-iarray-option* $A = (\text{if } A \neq \text{None} \text{ then Some}(\text{matrix-to-iarray } (\text{the } A)) \text{ else None})$

22.2 Some lemmas and code generation

lemma *matrix-inv-Gauss-Jordan-iarrays-PA*:

fixes $A::'a::\{\text{field}\}^{\sim n}::\{\text{mod-type}\}^{\sim n}::\{\text{mod-type}\}$

assumes *inv-A*: *invertible* A

shows $\text{matrix-to-iarray}(\text{matrix-inv } A) = \text{fst}(\text{Gauss-Jordan-iarrays-PA}(\text{matrix-to-iarray } A))$

<proof>

lemma *matrix-to-iarray-invertible*[*code-unfold*]:

fixes $A::'a::\{\text{field}\}^{\sim n}::\{\text{mod-type}\}^{\sim n}::\{\text{mod-type}\}$

shows $\text{invertible } A = \text{invertible-iarray}(\text{matrix-to-iarray } A)$

<proof>

lemma *matrix-to-iarray-option-inverse-matrix*:

fixes $A::'a::\{\text{field}\}^{\sim n}::\{\text{mod-type}\}^{\sim n}::\{\text{mod-type}\}$

shows $\text{matrix-to-iarray-option}(\text{inverse-matrix } A) = (\text{inverse-matrix-iarray}(\text{matrix-to-iarray } A))$

<proof>

lemma *matrix-to-iarray-option-inverse-matrix-code*[*code-unfold*]:

fixes $A::'a::\{\text{field}\}^{\sim n}::\{\text{mod-type}\}^{\sim n}::\{\text{mod-type}\}$

shows $\text{matrix-to-iarray-option}(\text{inverse-matrix } A) = (\text{let matrix-to-iarray-A} = \text{matrix-to-iarray } A; GJ = \text{Gauss-Jordan-iarrays-PA } \text{matrix-to-iarray-A}$

in if nrows-iarray matrix-to-iarray-A = length [x←IArray.list-of (snd GJ) . \neg 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); nrow
= IArray.length A in
      (if length [x←IArray.list-of (snd A') . ¬ is-zero-iarray x]
= nrow
      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-Numerals

```

```

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*⁵⁴))

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*⁵⁴)

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*⁵⁴))

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* $[[\text{Complex } 1 \ 1, \text{Complex } 1 \ (-1), \text{Complex } 0 \ 0], [\text{Complex } 2 \ (-1), \text{Complex } 1 \ 3, \text{Complex } 7 \ 3]]::\text{complex}^{\wedge 3 \wedge 2}$)))

value *iarray-of-iarray-to-list-of-list* (*Gauss-Jordan-iarrays* (*IArray* [*IArray* [*Complex* $1 \ 1, \text{Complex } 1 \ (-1), \text{Complex } 0 \ 0$], *IArray* [*Complex* $2 \ (-1), \text{Complex } 1 \ 3, \text{Complex } 7 \ 3$]]))

23.2.2 Inverse of a matrix

Examples on inverting matrices

definition *print-result-some-iarrays* $A =$ (*if* $A = \text{None}$ then None else Some (*iarray-of-iarray-to-list-of-list* (*the* A)))

value *let* $A =$ (*list-of-list-to-matrix* $[[1, 1, 2, 4, 5, 9, 8], [3, 0, 8, 4, 5, 0, 8], [3, 2, 0, 4, 5, 9, 8], [3, 2, 8, 0, 5, 9, 8], [3, 2, 8, 4, 0, 5, 9, 8]]::\text{rat}^{\wedge 5 \wedge 5}$)
in *print-result-some-iarrays* (*matrix-to-iarray-option* (*inverse-matrix* A))

value *let* $A =$ (*IArray* [*IArray* [$1::\text{real}, 1, 2, 4, 5, 9, 8$], *IArray* [$3, 0, 8, 4, 5, 0, 8$], *IArray* [$3, 2, 0, 4, 5, 9, 8$], *IArray* [$3, 2, 8, 0, 5, 9, 8$], *IArray* [$3, 2, 8, 4, 0, 5, 9, 8$]]]
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* $([[1, 8, 9, 1, 4, 7], [7, 2, 2, 5, 9], [3, 2, 7, 7, 4], [9, 8, 7, 5, 1], [1, 2, 6, 4, 5]]::\text{rat}^{\wedge 5 \wedge 5})$)
value *det* (*list-of-list-to-matrix* $[[1, 2, 7, 8, 9], [3, 4, 12, 10, 7], [-5, 4, 8, 7, 4], [0, 1, 2, 4, 8], [9, 8, 7, 13, 11]]::\text{rat}^{\wedge 5 \wedge 5})$)

value *det-iarrays* (*IArray* [*IArray* [$1::\text{real}, 2, 7, 8, 9$], *IArray* [$3, 4, 12, 10, 7$], *IArray* [$-5, 4, 8, 7, 4$], *IArray* [$0, 1, 2, 4, 8$], *IArray* [$9, 8, 7, 13, 11$]]]
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* [$900, 737, 280, 370, 121, 195, 958, 862, 957, 754::\text{real}$]])

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* $([[1, 3, -2, 0, 2, 0], [2, 6, -5, -2, 4, -3], [0, 0, 5, 10, 0, 15], [2, 6, 0, 8, 4, 18]]::\text{real}^{\wedge 4 \wedge 6})$)
in *vec-to-list*' (*basis-null-space* A)

value *let* $A =$ (*list-of-list-to-matrix* $([[3, 4, 0, 7], [1, -5, 2, -2], [-1, 4, 0, 3], [1, -1, 2, 2]]::\text{real}^{\wedge 4 \wedge 4})$)

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

value let $A = (\text{IArray}[\text{IArray}[1::\text{real}, 3, -2, 0, 2, 0], \text{IArray}[2, 6, -5, -2, 4, -3], \text{IArray}[0, 0, 5, 10, 0, 15], \text{IArray}[2, 6, 0, 8, 4, 18]])::\text{real}^4$
in $\text{IArray}.\text{list-of}' (\text{basis-null-space-iarrays } A)$
value let $A = (\text{IArray}[\text{IArray}[3::\text{real}, 4, 0, 7], \text{IArray}[1, -5, 2, -2], \text{IArray}[-1, 4, 0, 3], \text{IArray}[1, -1, 2, 2]])$
in $\text{IArray}.\text{list-of}' (\text{basis-null-space-iarrays } A)$

value let $A = (\text{list-of-list-to-matrix } ([[1, 3, -2, 0, 2, 0], [2, 6, -5, -2, 4, -3], [0, 0, 5, 10, 0, 15], [2, 6, 0, 8, 4, 18]])::\text{real}^4)$
in $\text{vec-to-list}' (\text{basis-row-space } A)$
value let $A = (\text{list-of-list-to-matrix } ([[3, 4, 0, 7], [1, -5, 2, -2], [-1, 4, 0, 3], [1, -1, 2, 2]])::\text{real}^4)$
in $\text{vec-to-list}' (\text{basis-row-space } A)$

value let $A = (\text{IArray}[\text{IArray}[1::\text{real}, 3, -2, 0, 2, 0], \text{IArray}[2, 6, -5, -2, 4, -3], \text{IArray}[0, 0, 5, 10, 0, 15], \text{IArray}[2, 6, 0, 8, 4, 18]])::\text{real}^4$
in $\text{IArray}.\text{list-of}' (\text{basis-row-space-iarrays } A)$
value let $A = (\text{IArray}[\text{IArray}[3::\text{real}, 4, 0, 7], \text{IArray}[1, -5, 2, -2], \text{IArray}[-1, 4, 0, 3], \text{IArray}[1, -1, 2, 2]])$
in $\text{IArray}.\text{list-of}' (\text{basis-row-space-iarrays } A)$

value let $A = (\text{list-of-list-to-matrix } ([[1, 3, -2, 0, 2, 0], [2, 6, -5, -2, 4, -3], [0, 0, 5, 10, 0, 15], [2, 6, 0, 8, 4, 18]])::\text{real}^4)$
in $\text{vec-to-list}' (\text{basis-col-space } A)$
value let $A = (\text{list-of-list-to-matrix } ([[3, 4, 0, 7], [1, -5, 2, -2], [-1, 4, 0, 3], [1, -1, 2, 2]])::\text{real}^4)$
in $\text{vec-to-list}' (\text{basis-col-space } A)$

value let $A = (\text{IArray}[\text{IArray}[1::\text{real}, 3, -2, 0, 2, 0], \text{IArray}[2, 6, -5, -2, 4, -3], \text{IArray}[0, 0, 5, 10, 0, 15], \text{IArray}[2, 6, 0, 8, 4, 18]])::\text{real}^4$
in $\text{IArray}.\text{list-of}' (\text{basis-col-space-iarrays } A)$
value let $A = (\text{IArray}[\text{IArray}[3::\text{real}, 4, 0, 7], \text{IArray}[1, -5, 2, -2], \text{IArray}[-1, 4, 0, 3], \text{IArray}[1, -1, 2, 2]])$
in $\text{IArray}.\text{list-of}' (\text{basis-col-space-iarrays } A)$

value let $A = (\text{list-of-list-to-matrix } ([[1, 3, -2, 0, 2, 0], [2, 6, -5, -2, 4, -3], [0, 0, 5, 10, 0, 15], [2, 6, 0, 8, 4, 18]])::\text{real}^4)$
in $\text{vec-to-list}' (\text{basis-left-null-space } A)$
value let $A = (\text{list-of-list-to-matrix } ([[3, 4, 0, 7], [1, -5, 2, -2], [-1, 4, 0, 3], [1, -1, 2, 2]])::\text{real}^4)$
in $\text{vec-to-list}' (\text{basis-left-null-space } A)$

value let $A = (\text{IArray}[\text{IArray}[1::\text{real}, 3, -2, 0, 2, 0], \text{IArray}[2, 6, -5, -2, 4, -3], \text{IArray}[0, 0, 5, 10, 0, 15], \text{IArray}[2, 6, 0, 8, 4, 18]])::\text{real}^4$
in $\text{IArray}.\text{list-of}' (\text{basis-left-null-space-iarrays } A)$
value let $A = (\text{IArray}[\text{IArray}[3::\text{real}, 4, 0, 7], \text{IArray}[1, -5, 2, -2], \text{IArray}[-1, 4, 0, 3], \text{IArray}[1, -1, 2, 2]])$
in $\text{IArray}.\text{list-of}' (\text{basis-left-null-space-iarrays } A)$

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* $A = (\text{if } A = \text{None then None else Some } (IArray.\text{list-of } (\text{fst } (\text{the } A)), IArray.\text{list-of}' (\text{snd } (\text{the } A))))$

```

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[10123,0,0,0,0,0,0,0,0,0,0,0,0,0],
IArray[0,10152,0,0,0,0,0,0,0,0,0,0,0,0,0],
IArray[0,0,10185,0,0,0,0,0,0,0,0,0,0,0,0],
IArray[0,0,0,10220,0,0,0,0,0,0,0,0,0,0,0],
IArray[0,0,0,0,10397,0,0,0,0,0,0,0,0,0,0],
IArray[0,0,0,0,0,10449,0,0,0,0,0,0,0,0,0],
IArray[0,0,0,0,0,0,10503,0,0,0,0,0,0,0,0],
IArray[0,0,0,0,0,0,0,10563,0,0,0,0,0,0,0],
IArray[0,0,0,0,0,0,0,0,10979,0,0,0,0,0,0],
IArray[0,0,0,0,0,0,0,0,0,101059,0,0,0,0,0],
IArray[0,0,0,0,0,0,0,0,0,0,101143,0,0,0,0],
IArray[0,0,0,0,0,0,0,0,0,0,0,101229,0,0,0],
IArray[0,0,0,0,0,0,0,0,0,0,0,0,101319,0],
IArray[0,0,0,0,0,0,0,0,0,0,0,0,0,(101412)::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 $gcd :: int \Rightarrow int \Rightarrow int$ $\langle proof \rangle$

lemma $gcd\text{-integer-code}[code]:$

$gcd\text{-integer } l\ k = |if\ l = (0::integer)\ then\ k\ else\ gcd\text{-integer } l\ (|k| \bmod |l|)|$
 $\langle proof \rangle$

end

code-printing

constant $abs :: integer \Rightarrow - \rightarrow (SML)\ IntInf.abs$

| **constant** $gcd\text{-integer} :: integer \Rightarrow - \Rightarrow - \rightarrow (SML)\ (PolyML.IntInf.gcd\ ((-),(-)))$

lemma $gcd\text{-code}[code]:$

$gcd\ a\ b = int\text{-of-integer}\ (gcd\text{-integer}\ (of\text{-int } a)\ (of\text{-int } b))$
 $\langle proof \rangle$

code-printing

constant $abs :: real \Rightarrow real \rightarrow$

$(SML)\ Real.abs$

declare $[[code\ drop:\ abs :: real \Rightarrow real]]$

There are several ways to serialize div and mod. The following ones are four examples of it:

code-printing

constant $divmod\text{-integer} :: integer \Rightarrow - \Rightarrow - \rightarrow (SML)\ (IntInf.quotRem\ ((-),(-)))$

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* \rightarrow (*Haskell*)
 \langle 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* \times *integer* \Rightarrow *rat*
 is *Frct* :: *int* \times *int* \Rightarrow *rat* \langle *proof* \rangle
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  $\rightarrow$  (Haskell) Prelude.Rational
| class-instance rat :: HOL.equal  $\Rightarrow$  (Haskell)  $-$ 
| constant 0 :: rat  $\rightarrow$  (Haskell) (Prelude.toRational (0::Integer))
| constant 1 :: rat  $\rightarrow$  (Haskell) (Prelude.toRational (1::Integer))
| constant Frct-integer  $\rightarrow$  (Haskell) Rational.fract  $(-)$ 
| constant numerator-integer  $\rightarrow$  (Haskell) Rational.numerator  $(-)$ 
| constant denominator-integer  $\rightarrow$  (Haskell) Rational.denominator  $(-)$ 
| constant HOL.equal :: rat  $\Rightarrow$  rat  $\Rightarrow$  bool  $\rightarrow$ 
(Haskell)  $(-)$   $==$   $(-)$ 
| constant  $(<)$  :: rat  $\Rightarrow$  rat  $\Rightarrow$  bool  $\rightarrow$ 
(Haskell)  $-$   $<$   $-$ 
| constant  $(\leq)$  :: rat  $\Rightarrow$  rat  $\Rightarrow$  bool  $\rightarrow$ 
(Haskell)  $-$   $\leq$   $-$ 
| constant  $(+)$  :: rat  $\Rightarrow$  rat  $\Rightarrow$  rat  $\rightarrow$ 
(Haskell)  $(-)$   $+$   $(-)$ 
| constant  $(-)$  :: rat  $\Rightarrow$  rat  $\Rightarrow$  rat  $\rightarrow$ 

```

```

    (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>.