Matrices, Jordan Normal Forms, and Spectral Radius Theory[*]

René Thiemann and Akihisa Yamada

May 14, 2024

Abstract

Matrix interpretations are useful as measure functions in termination proving. In order to use these interpretations also for complexity analysis, the growth rate of matrix powers has to examined. Here, we formalized an important result of spectral radius theory, namely that the growth rate is polynomially bounded if and only if the spectral radius of a matrix is at most one.

To formally prove this result we first studied the growth rates of matrices in Jordan normal form, and prove the result that every complex matrix has a Jordan normal form by means of two algorithms: we first convert matrices into similar ones via Schur decomposition, and then apply a second algorithm which converts an upper-triangular matrix into Jordan normal form. We further showed uniqueness of Jordan normal forms which then gives rise to a modular algorithm to compute individual blocks of a Jordan normal form.

The whole development is based on a new abstract type for matrices, which is also executable by a suitable setup of the code generator. It completely subsumes our former AFP-entry on executable matrices [6], and its main advantage is its close connection to the HMA-representation which allowed us to easily adapt existing proofs on determinants.

All the results have been applied to improve CeTA [7, 1], our certifier to validate termination and complexity proof certificates.

Contents

---

[*] Supported by FWF (Austrian Science Fund) project Y757.

1 Introduction

The spectral radius of a square, complex valued matrix A is defined as the largest norm of some eigenvalue c with eigenvector v. It is a central notion to estimate how the values in A^n for increasing n. If the spectral radius is larger than 1, clearly the values grow exponentially, since then $A^n \cdot v = c^n \cdot v$ becomes exponentially large.

The other results, namely that the values in A^n are bounded by a constant, if the spectral radius is smaller than 1, and that there is a polynomial bound if the spectral radius is exactly 1 are only immediate for matrices which have an eigenbasis, a precondition which is not satisfied by every matrix.

However, these results are derivable via Jordan normal forms (JNFs): If J is a JNF of A, then the growth rates of A^n and J^n are related by a constant as A and J are similar matrices. And for the values in J^n there is a closed formula which gives the desired complexity bounds. To be more precise, the values in J^n are bounded by $\mathcal{O}(|c|^n \cdot n^{k-1})$ where k is the size of the largest block of an eigenvalue c which has maximal norm w.r.t. the set of all eigenvalues. And since every complex matrix has a JNF, we can derive the polynomial (resp. constant bounds), if the spectral radius is 1 (resp. smaller than 1).

These results are already applied in current complexity tools, and the motivation of this development was to extend our certifier CeTA to be able to validate corresponding complexity proofs. To this end, we formalized the following main results:

- an algorithm to compute the characteristic polynomial, since the eigenvalues are exactly the roots of this polynomial;
- the complexity bounds for JNFs; and
- an algorithm which computes JNFs for every matrix, provided that the list of eigenvalues is given. With the help of the fundamental theorem of algebra this shows that every complex matrix has a JNF.

Since CeTA is generated from Isabelle/HOL via code-generation, all the algorithms and results need to be available at code-generation time. Especially there is no possibility to create types on the fly which are chosen to fit the matrix dimensions of the input. To this end, we cannot use the matrix-representation of HOL multivariate analysis (HMA).

Instead, we provide a new matrix library which is based on HOL-algebra with its explicit carriers. In contrast to our earlier development [6], we do not immediately formalize everything as lists of lists, but use a more mathematical notion as triples of the form (dimension, dimension, characteristic-function). This makes reasoning very similar to HMA, and a suitable implementation type can be chosen afterwards: we provide one via immutable arrays (we use IArray's from the HOL library), but one can also think of an implementation for sparse matrices, etc. Even the infinite carrier itself is executable where we rely upon Lochbihler's container framework [4] to have different set representations at the same time.

As a consequence of not using HMA, we could not directly reuse existing algorithms which have been formalized for this representation. For instance, we formalized our own version of Gauss-Jordan elimination which is not very different to the one of Divasón and Aransay in [2]: both define row-echelon form and apply elementary row transformations. Whereas Gauss-Jordan elimination has been developed from scratch as a case-study to see how suitable our matrix representation is, in other cases we often just copied and adjusted existing proofs from HMA. For instance, most of the library for determinants has been copied from the Isabelle distribution and adapted to our matrix representation.

As a result of our formalization, CeTA is now able to check polynomial bounds for matrix interpretations [3].

2 Material missing in the distribution

This theory provides some definitions and lemmas which we did not find in the Isabelle distribution.

theory *Missing-Misc*

imports
HOL−Library.FuncSet
HOL−Combinatorics.Permutations
begin

declare *finite-image-iff* [*simp*]

lemma *inj-on-finite*:
‹*finite* (*f* ‘ *A*) $\longleftrightarrow$ *finite A*› **if** ‹*inj-on f A*›
⟨*proof*⟩

The following lemma is slightly generalized from Determinants.thy in HMA.

lemma *finite-bounded-functions*:
assumes *fS*: *finite S*
shows *finite* $T \Longrightarrow$ *finite* $\{f.\ (\forall\, i \in T.\ f\ i \in S) \wedge (\forall\, i.\ i \notin T \longrightarrow f\ i = i)\}$
⟨*proof*⟩

lemma *finite-bounded-functions′*:
assumes *fS*: *finite S*
shows *finite* $T \Longrightarrow$ *finite* $\{f.\ (\forall\, i \in T.\ f\ i \in S) \wedge (\forall\, i.\ i \notin T \longrightarrow f\ i = j)\}$
⟨*proof*⟩

lemma *permutes-less* [*simp*]:
assumes *p*: *p permutes* $\{0..<(n :: nat)\}$
shows
$i < n \Longrightarrow p\ i < n$
$i < n \Longrightarrow inv\ p\ i < n$
$p\ (inv\ p\ i) = i$
$inv\ p\ (p\ i) = i$
⟨*proof*⟩

lemma *permutes-prod*:
assumes *p*: *p permutes S*
shows $(\prod s{\in}S.\ f\ (p\ s)\ s) = (\prod s{\in}S.\ f\ s\ (inv\ p\ s))$
(**is** *?l* = *?r*)
⟨*proof*⟩

lemma *permutes-sum*:
assumes *p*: *p permutes S*
shows $(\sum s{\in}S.\ f\ (p\ s)\ s) = (\sum s{\in}S.\ f\ s\ (inv\ p\ s))$
(**is** *?l* = *?r*)
⟨*proof*⟩

context
fixes *A* :: *′a set*
and *B* :: *′b set*
and *a-to-b* :: $'a \Rightarrow 'b$
and *b-to-a* :: $'b \Rightarrow 'a$

assumes *ab*: ⋀ *a*. *a* ∈ *A* ⟹ *a-to-b a* ∈ *B*
and *ba*: ⋀ *b*. *b* ∈ *B* ⟹ *b-to-a b* ∈ *A*
and *ab-ba*: ⋀ *a*. *a* ∈ *A* ⟹ *b-to-a* (*a-to-b a*) = *a*
and *ba-ab*: ⋀ *b*. *b* ∈ *B* ⟹ *a-to-b* (*b-to-a b*) = *b*
begin

qualified lemma *permutes-memb*: **fixes** *p* :: *'b* ⇒ *'b*
assumes *p*: *p permutes B*
and *a*: *a* ∈ *A*
defines *ip* ≡ *Hilbert-Choice.inv p*
shows *a* ∈ *A a-to-b a* ∈ *B ip* (*a-to-b a*) ∈ *B p* (*a-to-b a*) ∈ *B*
b-to-a (*p* (*a-to-b a*)) ∈ *A b-to-a* (*ip* (*a-to-b a*)) ∈ *A*
⟨*proof*⟩

lemma *permutes-bij-main*:
{*p* . *p permutes A*} ⊇ (λ *p a*. *if a* ∈ *A then b-to-a* (*p* (*a-to-b a*)) *else a*) ‘ {*p* . *p permutes B*}
(**is** *?A* ⊇ *?f* ‘ *?B*)
⟨*proof*⟩

end

lemma *permutes-bij'*: **assumes** *ab*: ⋀ *a*. *a* ∈ *A* ⟹ *a-to-b a* ∈ *B*
and *ba*: ⋀ *b*. *b* ∈ *B* ⟹ *b-to-a b* ∈ *A*
and *ab-ba*: ⋀ *a*. *a* ∈ *A* ⟹ *b-to-a* (*a-to-b a*) = *a*
and *ba-ab*: ⋀ *b*. *b* ∈ *B* ⟹ *a-to-b* (*b-to-a b*) = *b*
shows {*p* . *p permutes A*} = (λ *p a*. *if a* ∈ *A then b-to-a* (*p* (*a-to-b a*)) *else a*) ‘ {*p* . *p permutes B*}
(**is** *?A* = *?f* ‘ *?B*)
⟨*proof*⟩

lemma *permutes-others*:
assumes *p*: *p permutes S* **and** *x*: *x* ∉ *S* **shows** *p x* = *x*
⟨*proof*⟩

lemma *inj-on-nat-permutes*: **assumes** *i*: *inj-on f* (*S* :: *nat set*)
and *fS*: *f* ∈ *S* → *S*
and *fin*: *finite S*
and *f*: ⋀ *i*. *i* ∉ *S* ⟹ *f i* = *i*
shows *f permutes S*
⟨*proof*⟩

abbreviation (*input*) *signof* :: ‹(*nat* ⇒ *nat*) ⇒ *'a* :: *ring-1*›
where ‹*signof p* ≡ *of-int* (*sign p*)›

lemma *signof-id*:
signof id = *1*
signof (λ*x*. *x*) = *1*
⟨*proof*⟩

lemma *signof-inv*: *finite S* ⟹ *p permutes S* ⟹ *signof* (*inv p*) = *signof p*
⟨*proof*⟩

lemma *signof-pm-one*: *signof p* ∈ {*1*, − *1*}
⟨*proof*⟩

lemma *signof-compose*:
assumes *p permutes* {*0*..<(*n* :: *nat*)}
and *q permutes* {*0* ..<(*m* :: *nat*)}
shows *signof* (*p o q*) = *signof p* ∗ *signof q*
⟨*proof*⟩

end

3 Missing Ring

This theory contains several lemmas which might be of interest to the Isabelle distribution.

theory *Missing-Ring*
imports
Missing-Misc
HOL−Algebra.Ring
begin

context *ordered-cancel-semiring*
begin

subclass *ordered-cancel-ab-semigroup-add* ⟨*proof*⟩

end

partially ordered variant

class *ordered-semiring-strict* = *semiring* + *comm-monoid-add* + *ordered-cancel-ab-semigroup-add* +
assumes *mult-strict-left-mono*: $a < b \Longrightarrow 0 < c \Longrightarrow c * a < c * b$
assumes *mult-strict-right-mono*: $a < b \Longrightarrow 0 < c \Longrightarrow a * c < b * c$
begin

subclass *semiring-0-cancel* ⟨*proof*⟩

subclass *ordered-semiring*
⟨*proof*⟩

lemma *mult-pos-pos*[*simp*]: $0 < a \Longrightarrow 0 < b \Longrightarrow 0 < a * b$
⟨*proof*⟩

lemma *mult-pos-neg*: $0 < a \Longrightarrow b < 0 \Longrightarrow a * b < 0$

⟨*proof*⟩

lemma *mult-neg-pos*: $a < 0 \Longrightarrow 0 < b \Longrightarrow a * b < 0$
⟨*proof*⟩

Legacy - use *mult-neg-pos*

lemma *mult-pos-neg2*: $0 < a \Longrightarrow b < 0 \Longrightarrow b * a < 0$
⟨*proof*⟩

Strict monotonicity in both arguments

lemma *mult-strict-mono*:
assumes $a < b$ **and** $c < d$ **and** $0 < b$ **and** $0 \leq c$
shows $a * c < b * d$
⟨*proof*⟩

This weaker variant has more natural premises

lemma *mult-strict-mono'*:
assumes $a < b$ **and** $c < d$ **and** $0 \leq a$ **and** $0 \leq c$
shows $a * c < b * d$
⟨*proof*⟩

lemma *mult-less-le-imp-less*:
assumes $a < b$ **and** $c \leq d$ **and** $0 \leq a$ **and** $0 < c$
shows $a * c < b * d$
⟨*proof*⟩

lemma *mult-le-less-imp-less*:
assumes $a \leq b$ **and** $c < d$ **and** $0 < a$ **and** $0 \leq c$
shows $a * c < b * d$
⟨*proof*⟩

end

class *ordered-idom* = *idom* + *ordered-semiring-strict* +
assumes *zero-less-one* [*simp*]: $0 < 1$ **begin**

subclass *semiring-1* ⟨*proof*⟩
subclass *comm-ring-1* ⟨*proof*⟩
subclass *ordered-ring* ⟨*proof*⟩
subclass *ordered-comm-semiring* ⟨*proof*⟩
subclass *ordered-ab-semigroup-add* ⟨*proof*⟩

lemma *of-nat-ge-0*[*simp*]: *of-nat* $x \geq 0$
⟨*proof*⟩

lemma *of-nat-eq-0*[*simp*]: *of-nat* $x = 0 \longleftrightarrow x = 0$
⟨*proof*⟩

lemma *inj-of-nat*: *inj* (*of-nat* :: *nat* ⇒ *'a*)

⟨*proof*⟩

subclass *ring-char-0* ⟨*proof*⟩

end

context *comm-monoid*
begin

lemma *finprod-reindex-bij-betw*: *bij-betw h S T*
  $\Longrightarrow$ *g* $\in$ *h* ‘ *S* $\to$ *carrier G*
  $\Longrightarrow$ *finprod G* ($\lambda x.$ *g* (*h x*)) *S* = *finprod G g T*
  ⟨*proof*⟩

lemma *finprod-reindex-bij-witness*:
  assumes *witness*:
    $\bigwedge a.\ a \in S \Longrightarrow i\ (j\ a) = a$
    $\bigwedge a.\ a \in S \Longrightarrow j\ a \in T$
    $\bigwedge b.\ b \in T \Longrightarrow j\ (i\ b) = b$
    $\bigwedge b.\ b \in T \Longrightarrow i\ b \in S$
  assumes *eq*:
    $\bigwedge a.\ a \in S \Longrightarrow h\ (j\ a) = g\ a$
  assumes *g*: *g* $\in$ *S* $\to$ *carrier G*
  and *h*: *h* $\in$ *j* ‘ *S* $\to$ *carrier G*
  shows *finprod G g S* = *finprod G h T*
⟨*proof*⟩
end

lemmas (**in** *abelian-monoid*) *finsum-reindex-bij-witness* = *add.finprod-reindex-bij-witness*

locale *csemiring* = *semiring* + *comm-monoid R*

context *cring*
begin
sublocale *csemiring* ⟨*proof*⟩
end

lemma (**in** *comm-monoid*) *finprod-one'*:
  ($\bigwedge$ *a*. *a* $\in$ *A* $\Longrightarrow$ *f a* = $\mathbf{1}$) $\Longrightarrow$ *finprod G f A* = $\mathbf{1}$
  ⟨*proof*⟩

lemma (**in** *comm-monoid*) *finprod-split*:
  finite A $\Longrightarrow$ *f* ‘ *A* $\subseteq$ *carrier G* $\Longrightarrow$ *a* $\in$ *A* $\Longrightarrow$ *finprod G f A* = *f a* $\otimes$ *finprod G f*
($A - \{a\}$)
  ⟨*proof*⟩

lemma (**in** *comm-monoid*) *finprod-finprod*:

$\mathit{finite}\ A \Longrightarrow \mathit{finite}\ B \Longrightarrow (\bigwedge a\ b.\ a \in A\ \Longrightarrow b \in B \Longrightarrow g\ a\ b \in \mathit{carrier}\ G) \Longrightarrow$
$\mathit{finprod}\ G\ (\lambda\ a.\ \mathit{finprod}\ G\ (g\ a)\ B)\ A = \mathit{finprod}\ G\ (\lambda\ (a,b).\ g\ a\ b)\ (A \times B)$
⟨*proof*⟩

lemma (**in** *comm-monoid*) *finprod-swap*:
assumes $\mathit{finite}\ A\ \mathit{finite}\ B\ \bigwedge a\ b.\ a \in A\ \Longrightarrow b \in B \Longrightarrow g\ a\ b \in \mathit{carrier}\ G$
shows $\mathit{finprod}\ G\ (\lambda\ (b,a).\ g\ a\ b)\ (B \times A) = \mathit{finprod}\ G\ (\lambda\ (a,b).\ g\ a\ b)\ (A \times B)$
⟨*proof*⟩

lemma (**in** *comm-monoid*) *finprod-finprod-swap*:
$\mathit{finite}\ A \Longrightarrow \mathit{finite}\ B \Longrightarrow (\bigwedge a\ b.\ a \in A\ \Longrightarrow b \in B \Longrightarrow g\ a\ b \in \mathit{carrier}\ G) \Longrightarrow$
$\mathit{finprod}\ G\ (\lambda\ a.\ \mathit{finprod}\ G\ (g\ a)\ B)\ A = \mathit{finprod}\ G\ (\lambda\ b.\ \mathit{finprod}\ G\ (\lambda\ a.\ g\ a\ b)\ A)\ B$
⟨*proof*⟩

lemmas (**in** *semiring*) *finsum-zero'* = *add.finprod-one'*
lemmas (**in** *semiring*) *finsum-split* = *add.finprod-split*
lemmas (**in** *semiring*) *finsum-finsum-swap* = *add.finprod-finprod-swap*

lemma (**in** *csemiring*) *finprod-zero*:
$\mathit{finite}\ A \Longrightarrow f \in A \to \mathit{carrier}\ R \Longrightarrow \exists\, a{\in}A.\ f\ a = \mathbf{0}$
$\Longrightarrow \mathit{finprod}\ R\ f\ A = \mathbf{0}$
⟨*proof*⟩

lemma (**in** *semiring*) *finsum-product*:
assumes A: $\mathit{finite}\ A$ **and** B: $\mathit{finite}\ B$
and f: $f \in A \to \mathit{carrier}\ R$ **and** g: $g \in B \to \mathit{carrier}\ R$
shows $\mathit{finsum}\ R\ f\ A \otimes \mathit{finsum}\ R\ g\ B = (\bigoplus i{\in}A.\ \bigoplus j{\in}B.\ f\ i \otimes g\ j)$
⟨*proof*⟩

lemma (**in** *semiring*) *Units-one-side-I*:
$a \in \mathit{carrier}\ R \Longrightarrow p \in \mathit{Units}\ R \Longrightarrow p \otimes a = \mathbf{1} \Longrightarrow a \in \mathit{Units}\ R$
$a \in \mathit{carrier}\ R \Longrightarrow p \in \mathit{Units}\ R \Longrightarrow a \otimes p = \mathbf{1} \Longrightarrow a \in \mathit{Units}\ R$
⟨*proof*⟩

lemma *permutes-funcset*: $p\ \mathit{permutes}\ A \Longrightarrow (p\ `\ A \to B) = (A \to B)$
⟨*proof*⟩

context *comm-monoid*
begin
lemma *finprod-permute*:
assumes p: $p\ \mathit{permutes}\ S$
and f: $f \in S \to \mathit{carrier}\ G$
shows $\mathit{finprod}\ G\ f\ S = \mathit{finprod}\ G\ (f \circ p)\ S$
⟨*proof*⟩

lemma *finprod-singleton-set*[*simp*]: **assumes** *f a* ∈ *carrier G*
shows *finprod G f* {*a*} = *f a*
⟨*proof*⟩
end

lemmas (**in** *semiring*) *finsum-permute* = *add.finprod-permute*
lemmas (**in** *semiring*) *finsum-singleton-set* = *add.finprod-singleton-set*

context *cring*
begin

lemma *finsum-permutations-inverse*:
assumes *f*: *f* ∈ {*p. p permutes S*} → *carrier R*
shows *finsum R f* {*p. p permutes S*} = *finsum R* (λ*p. f*(*Hilbert-Choice.inv p*)) {*p. p permutes S*}
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *finsum-permutations-compose-right*: **assumes** *q*: *q permutes S*
and ∗: *f* ∈ {*p. p permutes S*} → *carrier R*
shows *finsum R f* {*p. p permutes S*} = *finsum R* (λ*p. f*(*p* ∘ *q*)) {*p. p permutes S*}
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

end

end

theory *Conjugate*
imports *HOL.Complex HOL−Library.Complex-Order*
begin

class *conjugate* =
fixes *conjugate* :: *′a* ⇒ *′a*
assumes *conjugate-id*[*simp*]: *conjugate* (*conjugate a*) = *a*
and *conjugate-cancel-iff*[*simp*]: *conjugate a* = *conjugate b* ⟷ *a* = *b*

class *conjugatable-ring* = *ring* + *conjugate* +
assumes *conjugate-dist-mul*: *conjugate* (*a* ∗ *b*) = *conjugate a* ∗ *conjugate b*
and *conjugate-dist-add*: *conjugate* (*a* + *b*) = *conjugate a* + *conjugate b*
and *conjugate-neg*: *conjugate* (−*a*) = − *conjugate a*
and *conjugate-zero*[*simp*]: *conjugate 0* = *0*
begin
lemma *conjugate-zero-iff*[*simp*]: *conjugate a* = *0* ⟷ *a* = *0*
⟨*proof*⟩
end

class *conjugatable-field* = *conjugatable-ring* + *field*

lemma *sum-conjugate*:
fixes *f* :: *'b* ⇒ *'a* :: *conjugatable-ring*
assumes *finX*: *finite X*
shows *conjugate* (*sum f X*) = *sum* (λ*x*. *conjugate* (*f x*)) *X*
⟨*proof*⟩

class *conjugatable-ordered-ring* = *conjugatable-ring* + *ordered-comm-monoid-add* +
assumes *conjugate-square-positive*: *a* ∗ *conjugate a* ≥ *0*

class *conjugatable-ordered-field* = *conjugatable-ordered-ring* + *field*
begin
subclass *conjugatable-field*⟨*proof*⟩
end

lemma *conjugate-square-0*:
fixes *a* :: *'a* :: {*conjugatable-ordered-ring*, *semiring-no-zero-divisors*}
shows *a* ∗ *conjugate a* = *0* ⟹ *a* = *0* ⟨*proof*⟩

3.1 Instantiations

instantiation *complex* :: *conjugatable-ordered-field*
begin
definition [*simp*]: *conjugate* ≡ *cnj*

instance
⟨*proof*⟩

end

instantiation *real* :: *conjugatable-ordered-field*
begin
definition [*simp*]: *conjugate* (*x*::*real*) ≡ *x*
instance ⟨*proof*⟩
end

instantiation *rat* :: *conjugatable-ordered-field*
begin
definition [*simp*]: *conjugate* (*x*::*rat*) ≡ *x*
instance ⟨*proof*⟩
end

instantiation *int* :: *conjugatable-ordered-ring*
begin
definition [*simp*]: *conjugate* (*x*::*int*) ≡ *x*
instance ⟨*proof*⟩
end

lemma *conjugate-square-eq-0* [*simp*]:
fixes $x :: 'a :: \{\mathit{conjugatable\text{-}ring}, \mathit{semiring\text{-}no\text{-}zero\text{-}divisors}\}$
shows $x * \mathit{conjugate}\ x = 0 \longleftrightarrow x = 0$ $\mathit{conjugate}\ x * x = 0 \longleftrightarrow x = 0$
⟨*proof*⟩

lemma *conjugate-square-greater-0* [*simp*]:
fixes $x :: 'a :: \{\mathit{conjugatable\text{-}ordered\text{-}ring}, \mathit{ring\text{-}no\text{-}zero\text{-}divisors}\}$
shows $x * \mathit{conjugate}\ x > 0 \longleftrightarrow x \neq 0$
⟨*proof*⟩

lemma *conjugate-square-smaller-0* [*simp*]:
fixes $x :: 'a :: \{\mathit{conjugatable\text{-}ordered\text{-}ring}, \mathit{ring\text{-}no\text{-}zero\text{-}divisors}\}$
shows $\neg\ x * \mathit{conjugate}\ x < 0$
⟨*proof*⟩

end

4 Vectors and Matrices

We define vectors as pairs of dimension and a characteristic function from natural numbers to elements. Similarly, matrices are defined as triples of two dimensions and one characteristic function from pairs of natural numbers to elements. Via a subtype we ensure that the characteristic function always behaves the same on indices outside the intended one. Hence, every matrix has a unique representation.

In this part we define basic operations like matrix-addition, -multiplication, scalar-product, etc. We connect these operations to HOL-Algebra with its explicit carrier sets.

theory *Matrix*
imports
Polynomial-Interpolation.Ring-Hom
Missing-Ring
Conjugate
HOL−Algebra.Module
begin

4.1 Vectors

Here we specify which value should be returned in case an index is out of bounds. The current solution has the advantage that in the implementation later on, no index comparison has to be performed.

definition $\mathit{undef\text{-}vec} :: \mathit{nat} \Rightarrow 'a$ **where**
$\mathit{undef\text{-}vec}\ i \equiv []\ !\ i$

definition $\mathit{mk\text{-}vec} :: \mathit{nat} \Rightarrow (\mathit{nat} \Rightarrow 'a) \Rightarrow (\mathit{nat} \Rightarrow 'a)$ **where**

$\textit{mk-vec}\ n\ f \equiv \lambda\ i.\ \textit{if}\ i < n\ \textit{then}\ f\ i\ \textit{else}\ \textit{undef-vec}\ (i - n)$

typedef $'a\ \textit{vec} = \{(n,\ \textit{mk-vec}\ n\ f) \mid n\ f :: \textit{nat} \Rightarrow 'a.\ \textit{True}\}$
$\langle\textit{proof}\rangle$

setup-lifting *type-definition-vec*

lift-definition $\textit{dim-vec} :: 'a\ \textit{vec} \Rightarrow \textit{nat}$ **is** *fst* $\langle\textit{proof}\rangle$
lift-definition $\textit{vec-index} :: 'a\ \textit{vec} \Rightarrow (\textit{nat} \Rightarrow 'a)$ (**infixl** \$ *100*) **is** *snd* $\langle\textit{proof}\rangle$
lift-definition $\textit{vec} :: \textit{nat} \Rightarrow (\textit{nat} \Rightarrow 'a) \Rightarrow 'a\ \textit{vec}$
is $\lambda\ n\ f.\ (n,\ \textit{mk-vec}\ n\ f)\ \langle\textit{proof}\rangle$

lift-definition $\textit{vec-of-list} :: 'a\ \textit{list} \Rightarrow 'a\ \textit{vec}$ **is**
$\lambda\ v.\ (\textit{length}\ v,\ \textit{mk-vec}\ (\textit{length}\ v)\ (\textit{nth}\ v))\ \langle\textit{proof}\rangle$

lift-definition $\textit{list-of-vec} :: 'a\ \textit{vec} \Rightarrow 'a\ \textit{list}$ **is**
$\lambda\ (n,v).\ \textit{map}\ v\ [0\ ..<\ n]\ \langle\textit{proof}\rangle$

definition $\textit{carrier-vec} :: \textit{nat} \Rightarrow 'a\ \textit{vec}\ \textit{set}$ **where**
$\textit{carrier-vec}\ n = \{\ v\ .\ \textit{dim-vec}\ v = n\}$

lemma *carrier-vec-dim-vec*[*simp*]: $v \in \textit{carrier-vec}\ (\textit{dim-vec}\ v)\ \langle\textit{proof}\rangle$

lemma *dim-vec*[*simp*]: $\textit{dim-vec}\ (\textit{vec}\ n\ f) = n\ \langle\textit{proof}\rangle$
lemma *vec-carrier*[*simp*]: $\textit{vec}\ n\ f \in \textit{carrier-vec}\ n\ \langle\textit{proof}\rangle$
lemma *index-vec*[*simp*]: $i < n \Longrightarrow \textit{vec}\ n\ f\ \$\ i = f\ i\ \langle\textit{proof}\rangle$
lemma *eq-vecI*[*intro*]: $(\bigwedge\ i.\ i < \textit{dim-vec}\ w \Longrightarrow v\ \$\ i = w\ \$\ i) \Longrightarrow \textit{dim-vec}\ v = \textit{dim-vec}\ w$
$\Longrightarrow v = w$
$\langle\textit{proof}\rangle$

lemma *carrier-dim-vec*: $v \in \textit{carrier-vec}\ n \longleftrightarrow \textit{dim-vec}\ v = n$
$\langle\textit{proof}\rangle$

lemma *carrier-vecD*[*simp*]: $v \in \textit{carrier-vec}\ n \Longrightarrow \textit{dim-vec}\ v = n\ \langle\textit{proof}\rangle$

lemma *carrier-vecI*: $\textit{dim-vec}\ v = n \Longrightarrow v \in \textit{carrier-vec}\ n\ \langle\textit{proof}\rangle$

instantiation $\textit{vec} :: (\textit{plus})\ \textit{plus}$
begin
definition $\textit{plus-vec} :: 'a\ \textit{vec} \Rightarrow 'a\ \textit{vec} \Rightarrow 'a :: \textit{plus}\ \textit{vec}$ **where**
$v_1 + v_2 \equiv \textit{vec}\ (\textit{dim-vec}\ v_2)\ (\lambda\ i.\ v_1\ \$\ i + v_2\ \$\ i)$
instance $\langle\textit{proof}\rangle$
end

instantiation $\textit{vec} :: (\textit{minus})\ \textit{minus}$
begin
definition $\textit{minus-vec} :: 'a\ \textit{vec} \Rightarrow 'a\ \textit{vec} \Rightarrow 'a :: \textit{minus}\ \textit{vec}$ **where**
$v_1 - v_2 \equiv \textit{vec}\ (\textit{dim-vec}\ v_2)\ (\lambda\ i.\ v_1\ \$\ i - v_2\ \$\ i)$

instance ⟨*proof*⟩
end

definition
zero-vec :: *nat* ⇒ *'a* :: *zero vec* (0_v)
where 0_v *n* ≡ *vec n* (λ *i*. *0*)

lemma *zero-carrier-vec*[*simp*]: 0_v *n* ∈ *carrier-vec n*
⟨*proof*⟩

lemma *index-zero-vec*[*simp*]: *i* < *n* ⟹ 0_v *n* \$ *i* = *0* *dim-vec* (0_v *n*) = *n*
⟨*proof*⟩

lemma *vec-of-dim-0*[*simp*]: *dim-vec v* = *0* ⟷ *v* = 0_v *0* ⟨*proof*⟩

definition
unit-vec :: *nat* ⇒ *nat* ⇒ (*'a* :: *zero-neq-one*) *vec*
where *unit-vec n i* = *vec n* (λ *j*. *if j* = *i then 1 else 0*)

lemma *index-unit-vec*[*simp*]:
i < *n* ⟹ *j* < *n* ⟹ *unit-vec n i* \$ *j* = (*if j* = *i then 1 else 0*)
i < *n* ⟹ *unit-vec n i* \$ *i* = *1*
dim-vec (*unit-vec n i*) = *n*
⟨*proof*⟩

lemma *unit-vec-eq*[*simp*]:
assumes *i*: *i* < *n*
shows (*unit-vec n i* = *unit-vec n j*) = (*i* = *j*)
⟨*proof*⟩

lemma *unit-vec-nonzero*[*simp*]:
assumes *i-n*: *i* < *n* **shows** *unit-vec n i* ≠ *zero-vec n* (**is** *?l* ≠ *?r*)
⟨*proof*⟩

lemma *unit-vec-carrier*[*simp*]: *unit-vec n i* ∈ *carrier-vec n*
⟨*proof*⟩

definition *unit-vecs*:: *nat* ⇒ *'a* :: *zero-neq-one vec list*
where *unit-vecs n* = *map* (*unit-vec n*) [*0*..<*n*]

List of first i units

fun *unit-vecs-first*:: *nat* ⇒ *nat* ⇒ *'a*::*zero-neq-one vec list*
where *unit-vecs-first n 0* = []
| *unit-vecs-first n* (*Suc i*) = *unit-vecs-first n i* @ [*unit-vec n i*]

lemma *unit-vecs-first*: *unit-vecs n* = *unit-vecs-first n n*
⟨*proof*⟩

list of last i units

fun *unit-vecs-last*:: *nat* ⇒ *nat* ⇒ *'a* :: *zero-neq-one vec list*

where *unit-vecs-last n 0* = []
  | *unit-vecs-last n* (*Suc i*) = *unit-vec n* (*n* − *Suc i*) # *unit-vecs-last n i*

lemma *unit-vecs-last-carrier*: *set* (*unit-vecs-last n i*) $\subseteq$ *carrier-vec n*
⟨*proof*⟩

lemma *unit-vecs-last*[*code*]: *unit-vecs n* = *unit-vecs-last n n*
⟨*proof*⟩

lemma *unit-vecs-carrier*: *set* (*unit-vecs n*) $\subseteq$ *carrier-vec n*
⟨*proof*⟩

lemma *unit-vecs-last-distinct*:
$j \le n \Longrightarrow i < n - j \Longrightarrow$ *unit-vec n i* $\notin$ *set* (*unit-vecs-last n j*)
⟨*proof*⟩

lemma *unit-vecs-first-distinct*:
$i \le j \Longrightarrow j < n \Longrightarrow$ *unit-vec n j* $\notin$ *set* (*unit-vecs-first n i*)
⟨*proof*⟩

definition *map-vec* **where** *map-vec f v* $\equiv$ *vec* (*dim-vec v*) ($\lambda i.$ *f* (*v* \$ *i*))

instantiation *vec* :: (*uminus*) *uminus*
begin
definition *uminus-vec* :: *'a* :: *uminus vec* $\Rightarrow$ *'a vec* **where**
− *v* $\equiv$ *vec* (*dim-vec v*) (λ *i.* − (*v* \$ *i*))
instance ⟨*proof*⟩
end

definition *smult-vec* :: *'a* :: *times* $\Rightarrow$ *'a vec* $\Rightarrow$ *'a vec* (**infixl** $\cdot_v$ *70*)
where *a* $\cdot_v$ *v* $\equiv$ *vec* (*dim-vec v*) (λ *i. a* $*$ *v* \$ *i*)

definition *scalar-prod* :: *'a vec* $\Rightarrow$ *'a vec* $\Rightarrow$ *'a* :: *semiring-0* (**infix** $\cdot$ *70*)
where *v* $\cdot$ *w* $\equiv \sum$ *i* $\in$ {*0* ..< *dim-vec w*}. *v* \$ *i* $*$ *w* \$ *i*

definition *monoid-vec* :: *'a itself* $\Rightarrow$ *nat* $\Rightarrow$ (*'a* :: *monoid-add vec*) *monoid* **where**
monoid-vec ty n $\equiv$ ⦇
carrier = *carrier-vec n*,
mult = (+),
one = 0_v *n*⦈

definition *module-vec* ::
'a :: *semiring-1 itself* $\Rightarrow$ *nat* $\Rightarrow$ (*'a*,*'a vec*) *module* **where**
module-vec ty n $\equiv$ ⦇
carrier = *carrier-vec n*,
mult = *undefined*,
one = *undefined*,
zero = 0_v *n*,
add = (+),

$\mathit{smult} = (\cdot_v)$⦈

lemma *monoid-vec-simps*:
$\mathit{mult}\ (\mathit{monoid\text{-}vec}\ \mathit{ty}\ n) = (+)$
$\mathit{carrier}\ (\mathit{monoid\text{-}vec}\ \mathit{ty}\ n) = \mathit{carrier\text{-}vec}\ n$
$\mathit{one}\ (\mathit{monoid\text{-}vec}\ \mathit{ty}\ n) = \mathit{0}_v\ n$
$\langle \mathit{proof} \rangle$

lemma *module-vec-simps*:
$\mathit{add}\ (\mathit{module\text{-}vec}\ \mathit{ty}\ n) = (+)$
$\mathit{zero}\ (\mathit{module\text{-}vec}\ \mathit{ty}\ n) = \mathit{0}_v\ n$
$\mathit{carrier}\ (\mathit{module\text{-}vec}\ \mathit{ty}\ n) = \mathit{carrier\text{-}vec}\ n$
$\mathit{smult}\ (\mathit{module\text{-}vec}\ \mathit{ty}\ n) = (\cdot_v)$
$\langle \mathit{proof} \rangle$

definition *finsum-vec* $:: {}'a :: \mathit{monoid\text{-}add}\ \mathit{itself} \Rightarrow \mathit{nat} \Rightarrow ({}'c \Rightarrow {}'a\ \mathit{vec}) \Rightarrow {}'c\ \mathit{set} \Rightarrow {}'a\ \mathit{vec}$ **where**
$\mathit{finsum\text{-}vec}\ \mathit{ty}\ n = \mathit{finprod}\ (\mathit{monoid\text{-}vec}\ \mathit{ty}\ n)$

lemma *index-add-vec*[*simp*]:
$i < \mathit{dim\text{-}vec}\ v_2 \Longrightarrow (v_1 + v_2)\ \$\ i = v_1\ \$\ i + v_2\ \$\ i\ \mathit{dim\text{-}vec}\ (v_1 + v_2) = \mathit{dim\text{-}vec}\ v_2$
$\langle \mathit{proof} \rangle$

lemma *index-minus-vec*[*simp*]:
$i < \mathit{dim\text{-}vec}\ v_2 \Longrightarrow (v_1 - v_2)\ \$\ i = v_1\ \$\ i - v_2\ \$\ i\ \mathit{dim\text{-}vec}\ (v_1 - v_2) = \mathit{dim\text{-}vec}\ v_2$
$\langle \mathit{proof} \rangle$

lemma *index-map-vec*[*simp*]:
$i < \mathit{dim\text{-}vec}\ v \Longrightarrow \mathit{map\text{-}vec}\ f\ v\ \$\ i = f\ (v\ \$\ i)$
$\mathit{dim\text{-}vec}\ (\mathit{map\text{-}vec}\ f\ v) = \mathit{dim\text{-}vec}\ v$
$\langle \mathit{proof} \rangle$

lemma *map-carrier-vec*[*simp*]: $\mathit{map\text{-}vec}\ h\ v \in \mathit{carrier\text{-}vec}\ n = (v \in \mathit{carrier\text{-}vec}\ n)$
$\langle \mathit{proof} \rangle$

lemma *index-uminus-vec*[*simp*]:
$i < \mathit{dim\text{-}vec}\ v \Longrightarrow (-\ v)\ \$\ i = -\ (v\ \$\ i)$
$\mathit{dim\text{-}vec}\ (-\ v) = \mathit{dim\text{-}vec}\ v$
$\langle \mathit{proof} \rangle$

lemma *index-smult-vec*[*simp*]:
$i < \mathit{dim\text{-}vec}\ v \Longrightarrow (a \cdot_v v)\ \$\ i = a * v\ \$\ i\ \mathit{dim\text{-}vec}\ (a \cdot_v v) = \mathit{dim\text{-}vec}\ v$
$\langle \mathit{proof} \rangle$

lemma *add-carrier-vec*[*simp*]:
$v_1 \in \mathit{carrier\text{-}vec}\ n \Longrightarrow v_2 \in \mathit{carrier\text{-}vec}\ n \Longrightarrow v_1 + v_2 \in \mathit{carrier\text{-}vec}\ n$
$\langle \mathit{proof} \rangle$

lemma *minus-carrier-vec*[*simp*]:
$v_1 \in \textit{carrier-vec } n \Longrightarrow v_2 \in \textit{carrier-vec } n \Longrightarrow v_1 - v_2 \in \textit{carrier-vec } n$
$\langle \textit{proof} \rangle$

lemma *comm-add-vec*[*ac-simps*]:
$(v_1 :: {'a} :: \textit{ab-semigroup-add vec}) \in \textit{carrier-vec } n \Longrightarrow v_2 \in \textit{carrier-vec } n \Longrightarrow v_1 + v_2 = v_2 + v_1$
$\langle \textit{proof} \rangle$

lemma *assoc-add-vec*[*simp*]:
$(v_1 :: {'a} :: \textit{semigroup-add vec}) \in \textit{carrier-vec } n \Longrightarrow v_2 \in \textit{carrier-vec } n \Longrightarrow v_3 \in \textit{carrier-vec } n$
$\Longrightarrow (v_1 + v_2) + v_3 = v_1 + (v_2 + v_3)$
$\langle \textit{proof} \rangle$

lemma *zero-minus-vec*[*simp*]: $(v :: {'a} :: \textit{group-add vec}) \in \textit{carrier-vec } n \Longrightarrow \mathit{0}_v\ n - v = - v$
$\langle \textit{proof} \rangle$

lemma *minus-zero-vec*[*simp*]: $(v :: {'a} :: \textit{group-add vec}) \in \textit{carrier-vec } n \Longrightarrow v - \mathit{0}_v\ n = v$
$\langle \textit{proof} \rangle$

lemma *minus-cancel-vec*[*simp*]: $(v :: {'a} :: \textit{group-add vec}) \in \textit{carrier-vec } n \Longrightarrow v - v = \mathit{0}_v\ n$
$\langle \textit{proof} \rangle$

lemma *minus-add-uminus-vec*: $(v :: {'a} :: \textit{group-add vec}) \in \textit{carrier-vec } n \Longrightarrow$
$w \in \textit{carrier-vec } n \Longrightarrow v - w = v + (- w)$
$\langle \textit{proof} \rangle$

lemma *comm-monoid-vec*: *comm-monoid* (*monoid-vec TYPE* ($'a$ :: *comm-monoid-add*) *n*)
$\langle \textit{proof} \rangle$

lemma *left-zero-vec*[*simp*]: $(v :: {'a} :: \textit{monoid-add vec}) \in \textit{carrier-vec } n \Longrightarrow \mathit{0}_v\ n + v = v$ $\langle \textit{proof} \rangle$

lemma *right-zero-vec*[*simp*]: $(v :: {'a} :: \textit{monoid-add vec}) \in \textit{carrier-vec } n \Longrightarrow v + \mathit{0}_v\ n = v$ $\langle \textit{proof} \rangle$

lemma *uminus-carrier-vec*[*simp*]:
$(- v \in \textit{carrier-vec } n) = (v \in \textit{carrier-vec } n)$
$\langle \textit{proof} \rangle$

lemma *uminus-r-inv-vec*[*simp*]:
$(v :: {'a} :: \textit{group-add vec}) \in \textit{carrier-vec } n \Longrightarrow (v + - v) = \mathit{0}_v\ n$

⟨*proof*⟩

lemma *uminus-l-inv-vec*[*simp*]:
(*v* :: *'a* :: *group-add vec*) ∈ *carrier-vec n* ⟹ (− *v* + *v*) = 0_v *n*
⟨*proof*⟩

lemma *add-inv-exists-vec*:
(*v* :: *'a* :: *group-add vec*) ∈ *carrier-vec n* ⟹ ∃ *w* ∈ *carrier-vec n*. *w* + *v* = 0_v *n* ∧ *v* + *w* = 0_v *n*
⟨*proof*⟩

lemma *comm-group-vec*: *comm-group* (*monoid-vec TYPE* (*'a* :: *ab-group-add*) *n*)
⟨*proof*⟩

lemmas *finsum-vec-insert* =
comm-monoid.finprod-insert[*OF comm-monoid-vec*, *folded finsum-vec-def*, *unfolded monoid-vec-simps*]

lemmas *finsum-vec-closed* =
comm-monoid.finprod-closed[*OF comm-monoid-vec*, *folded finsum-vec-def*, *unfolded monoid-vec-simps*]

lemmas *finsum-vec-empty* =
comm-monoid.finprod-empty[*OF comm-monoid-vec*, *folded finsum-vec-def*, *unfolded monoid-vec-simps*]

lemma *smult-carrier-vec*[*simp*]: (*a* $\cdot_v$ *v* ∈ *carrier-vec n*) = (*v* ∈ *carrier-vec n*)
⟨*proof*⟩

lemma *scalar-prod-left-zero*[*simp*]: *v* ∈ *carrier-vec n* ⟹ 0_v *n* • *v* = *0*
⟨*proof*⟩

lemma *scalar-prod-right-zero*[*simp*]: *v* ∈ *carrier-vec n* ⟹ *v* • 0_v *n* = *0*
⟨*proof*⟩

lemma *scalar-prod-left-unit*[*simp*]: **assumes** *v*: (*v* :: *'a* :: *semiring-1 vec*) ∈ *carrier-vec n* **and** *i*: *i* < *n*
shows *unit-vec n i* • *v* = *v* \$ *i*
⟨*proof*⟩

lemma *scalar-prod-right-unit*[*simp*]: **assumes** *i*: *i* < *n*
shows (*v* :: *'a* :: *semiring-1 vec*) • *unit-vec n i* = *v* \$ *i*
⟨*proof*⟩

lemma *add-scalar-prod-distrib*: **assumes** *v*: v_1 ∈ *carrier-vec n* v_2 ∈ *carrier-vec n* v_3 ∈ *carrier-vec n*
shows (v_1 + v_2) • v_3 = v_1 • v_3 + v_2 • v_3
⟨*proof*⟩

lemma *scalar-prod-add-distrib*: **assumes** *v*: $v_1 \in \mathit{carrier\text{-}vec}\ n$ $v_2 \in \mathit{carrier\text{-}vec}\ n$ $v_3 \in \mathit{carrier\text{-}vec}\ n$
shows $v_1 \cdot (v_2 + v_3) = v_1 \cdot v_2 + v_1 \cdot v_3$
⟨*proof*⟩

lemma *smult-scalar-prod-distrib*[*simp*]: **assumes** *v*: $v_1 \in \mathit{carrier\text{-}vec}\ n$ $v_2 \in \mathit{carrier\text{-}vec}\ n$
shows $(a \cdot_v v_1) \cdot v_2 = a * (v_1 \cdot v_2)$
⟨*proof*⟩

lemma *scalar-prod-smult-distrib*[*simp*]: **assumes** *v*: $v_1 \in \mathit{carrier\text{-}vec}\ n$ $v_2 \in \mathit{carrier\text{-}vec}\ n$
shows $v_1 \cdot (a \cdot_v v_2) = (a :: 'a :: \mathit{comm\text{-}ring}) * (v_1 \cdot v_2)$
⟨*proof*⟩

lemma *comm-scalar-prod*: **assumes** $(v_1 :: 'a :: \mathit{comm\text{-}semiring\text{-}0}\ \mathit{vec}) \in \mathit{carrier\text{-}vec}\ n$ $v_2 \in \mathit{carrier\text{-}vec}\ n$
shows $v_1 \cdot v_2 = v_2 \cdot v_1$
⟨*proof*⟩

lemma *add-smult-distrib-vec*:
$((a::'a::\mathit{ring}) + b) \cdot_v v = a \cdot_v v + b \cdot_v v$
⟨*proof*⟩

lemma *smult-add-distrib-vec*:
assumes $v \in \mathit{carrier\text{-}vec}\ n$ $w \in \mathit{carrier\text{-}vec}\ n$
shows $(a::'a::\mathit{ring}) \cdot_v (v + w) = a \cdot_v v + a \cdot_v w$
⟨*proof*⟩

lemma *smult-smult-assoc*:
$a \cdot_v (b \cdot_v v) = (a * b::'a::\mathit{ring}) \cdot_v v$
⟨*proof*⟩

lemma *one-smult-vec* [*simp*]:
$(1::'a::\mathit{ring\text{-}1}) \cdot_v v = v$ ⟨*proof*⟩

lemma *uminus-zero-vec*[*simp*]: $-\ (0_v\ n) = (0_v\ n :: 'a :: \mathit{group\text{-}add}\ \mathit{vec})$
⟨*proof*⟩

lemma *index-finsum-vec*: **assumes** *finite F* **and** *i*: $i < n$
and *vs*: $vs \in F \rightarrow \mathit{carrier\text{-}vec}\ n$
shows $\mathit{finsum\text{-}vec}\ \mathit{TYPE}('a :: \mathit{comm\text{-}monoid\text{-}add})\ n\ vs\ F\ \$\ i = \mathit{sum}\ (\lambda\ f.\ vs\ f\ \$\ i)\ F$
⟨*proof*⟩

Definition of pointwise ordering on vectors for non-strict part, and strict version is defined in a way such that the *order* constraints are satisfied.

instantiation *vec* :: (*ord*) *ord*
begin

definition *less-eq-vec* :: *'a vec ⇒ 'a vec ⇒ bool* **where**
less-eq-vec v w = (*dim-vec v* = *dim-vec w* ∧ (∀ *i* < *dim-vec w*. *v* $ *i* ≤ *w* $ *i*))

definition *less-vec* :: *'a vec ⇒ 'a vec ⇒ bool* **where**
less-vec v w = (*v* ≤ *w* ∧ ¬ (*w* ≤ *v*))
instance ⟨*proof*⟩
end

instantiation *vec* :: (*preorder*) *preorder*
begin
instance
⟨*proof*⟩
end

instantiation *vec* :: (*order*) *order*
begin
instance
⟨*proof*⟩
end

4.2 Matrices

Similarly as for vectors, we specify which value should be returned in case an index is out of bounds. It is defined in a way that only few index comparisons have to be performed in the implementation.

definition *undef-mat* :: *nat ⇒ nat ⇒ (nat × nat ⇒ 'a) ⇒ nat × nat ⇒ 'a* **where**
undef-mat nr nc f ≡ λ (*i*,*j*). [[*f* (*i*,*j*). *j* <− [*0* ..< *nc*]] . *i* <− [*0* ..< *nr*]] ! *i* ! *j*

lemma *undef-cong-mat*: **assumes** ⋀ *i j*. *i* < *nr* ⟹ *j* < *nc* ⟹ *f* (*i*,*j*) = *f'* (*i*,*j*)
shows *undef-mat nr nc f x* = *undef-mat nr nc f' x*
⟨*proof*⟩

definition *mk-mat* :: *nat ⇒ nat ⇒ (nat × nat ⇒ 'a) ⇒ (nat × nat ⇒ 'a)* **where**
mk-mat nr nc f ≡ λ (*i*,*j*). *if i* < *nr* ∧ *j* < *nc then f* (*i*,*j*) *else undef-mat nr nc f* (*i*,*j*)

lemma *cong-mk-mat*: **assumes** ⋀ *i j*. *i* < *nr* ⟹ *j* < *nc* ⟹ *f* (*i*,*j*) = *f'* (*i*,*j*)
shows *mk-mat nr nc f* = *mk-mat nr nc f'*
⟨*proof*⟩

typedef *'a mat* = {(*nr*, *nc*, *mk-mat nr nc f*) | *nr nc f* :: *nat × nat ⇒ 'a*. *True*}
⟨*proof*⟩

setup-lifting *type-definition-mat*

lift-definition *dim-row* :: *'a mat ⇒ nat* **is** *fst* ⟨*proof*⟩
lift-definition *dim-col* :: *'a mat ⇒ nat* **is** *fst o snd* ⟨*proof*⟩
lift-definition *index-mat* :: *'a mat ⇒ (nat × nat ⇒ 'a)* (**infixl** $$ *100*) **is** *snd o*

snd ⟨*proof*⟩
lift-definition *mat* :: *nat* ⇒ *nat* ⇒ (*nat* × *nat* ⇒ *'a*) ⇒ *'a mat*
 is *λ nr nc f*. (*nr*, *nc*, *mk-mat nr nc f*) ⟨*proof*⟩
lift-definition *mat-of-row-fun* :: *nat* ⇒ *nat* ⇒ (*nat* ⇒ *'a vec*) ⇒ *'a mat* (mat_r)
 is *λ nr nc f*. (*nr*, *nc*, *mk-mat nr nc* (*λ* (*i*,*j*). *f i* $ *j*)) ⟨*proof*⟩

definition *mat-to-list* :: *'a mat* ⇒ *'a list list* **where**
 mat-to-list A = [[*A* $$ (*i*,*j*) . *j* <− [*0* ..< *dim-col A*]] . *i* <− [*0* ..< *dim-row A*]]

fun *square-mat* :: *'a mat* ⇒ *bool* **where** *square-mat A* = (*dim-col A* = *dim-row A*)

definition *upper-triangular* :: *'a*::*zero mat* ⇒ *bool*
 where *upper-triangular A* ≡
  ∀ *i* < *dim-row A*. ∀ *j* < *i*. *A* $$ (*i*,*j*) = *0*

lemma *upper-triangularD*[*elim*] :
 upper-triangular A ⟹ *j* < *i* ⟹ *i* < *dim-row A* ⟹ *A* $$ (*i*,*j*) = *0*
⟨*proof*⟩

lemma *upper-triangularI*[*intro*] :
 (⋀*i j*. *j* < *i* ⟹ *i* < *dim-row A* ⟹ *A* $$ (*i*,*j*) = *0*) ⟹ *upper-triangular A*
⟨*proof*⟩

lemma *dim-row-mat*[*simp*]: *dim-row* (*mat nr nc f*) = *nr dim-row* (mat_r *nr nc g*) = *nr*
 ⟨*proof*⟩

lemma *dim-col-mat*[*simp*]: *dim-col* (*mat nr nc f*) = *nc dim-col* (mat_r *nr nc g*) = *nc*
 ⟨*proof*⟩

definition *carrier-mat* :: *nat* ⇒ *nat* ⇒ *'a mat set*
 where *carrier-mat nr nc* = { *m* . *dim-row m* = *nr* ∧ *dim-col m* = *nc*}

lemma *carrier-mat-triv*[*simp*]: *m* ∈ *carrier-mat* (*dim-row m*) (*dim-col m*)
 ⟨*proof*⟩

lemma *mat-carrier*[*simp*]: *mat nr nc f* ∈ *carrier-mat nr nc*
 ⟨*proof*⟩

definition *elements-mat* :: *'a mat* ⇒ *'a set*
 where *elements-mat A* = *set* [*A* $$ (*i*,*j*). *i* <− [*0* ..< *dim-row A*], *j* <− [*0* ..< *dim-col A*]]

lemma *elements-matD* [*dest*]:
 a ∈ *elements-mat A* ⟹ ∃ *i j*. *i* < *dim-row A* ∧ *j* < *dim-col A* ∧ *a* = *A* $$ (*i*,*j*)
 ⟨*proof*⟩

lemma *elements-matI* [*intro*]:

A ∈ *carrier-mat nr nc* ⟹ *i* < *nr* ⟹ *j* < *nc* ⟹ *a* = *A* $$ (*i*,*j*) ⟹ *a* ∈ *elements-mat A*
⟨*proof*⟩

lemma *index-mat*[*simp*]: *i* < *nr* ⟹ *j* < *nc* ⟹ *mat nr nc f* $$ (*i*,*j*) = *f* (*i*,*j*)
i < *nr* ⟹ *j* < *nc* ⟹ mat_r *nr nc g* $$ (*i*,*j*) = *g i* $ *j*
⟨*proof*⟩

lemma *eq-matI*[*intro*]: (⋀ *i j* . *i* < *dim-row B* ⟹ *j* < *dim-col B* ⟹ *A* $$ (*i*,*j*) = *B* $$ (*i*,*j*))
⟹ *dim-row A* = *dim-row B*
⟹ *dim-col A* = *dim-col B*
⟹ *A* = *B*
⟨*proof*⟩

lemma *carrier-matI*[*intro*]:
assumes *dim-row A* = *nr dim-col A* = *nc* **shows** *A* ∈ *carrier-mat nr nc*
⟨*proof*⟩

lemma *carrier-matD*[*dest*,*simp*]: **assumes** *A* ∈ *carrier-mat nr nc*
shows *dim-row A* = *nr dim-col A* = *nc* ⟨*proof*⟩

lemma *cong-mat*: **assumes** *nr* = *nr*′ *nc* = *nc*′ ⋀ *i j*. *i* < *nr* ⟹ *j* < *nc* ⟹
f (*i*,*j*) = *f*′ (*i*,*j*) **shows** *mat nr nc f* = *mat nr*′ *nc*′ *f*′
⟨*proof*⟩

definition *row* :: ′*a mat* ⇒ *nat* ⇒ ′*a vec* **where**
row A i = *vec* (*dim-col A*) (λ *j*. *A* $$ (*i*,*j*))

definition *rows* :: ′*a mat* ⇒ ′*a vec list* **where**
rows A = *map* (*row A*) [*0*..<*dim-row A*]

lemma *row-carrier*[*simp*]: *row A i* ∈ *carrier-vec* (*dim-col A*) ⟨*proof*⟩

lemma *rows-carrier*[*simp*]: *set* (*rows A*) ⊆ *carrier-vec* (*dim-col A*) ⟨*proof*⟩

lemma *length-rows*[*simp*]: *length* (*rows A*) = *dim-row A* ⟨*proof*⟩

lemma *nth-rows*[*simp*]: *i* < *dim-row A* ⟹ *rows A* ! *i* = *row A i*
⟨*proof*⟩

lemma *row-mat-of-row-fun*[*simp*]: *i* < *nr* ⟹ *dim-vec* (*f i*) = *nc* ⟹ *row* (mat_r *nr nc f*) *i* = *f i*
⟨*proof*⟩

lemma *set-rows-carrier*:
assumes *A* ∈ *carrier-mat m n* **and** *v* ∈ *set* (*rows A*) **shows** *v* ∈ *carrier-vec n*
⟨*proof*⟩

definition *mat-of-rows* :: *nat* ⇒ *'a vec list* ⇒ *'a mat*
where *mat-of-rows n rs* = *mat* (*length rs*) *n* (λ(*i*,*j*). *rs* ! *i* $ *j*)

definition *mat-of-rows-list* :: *nat* ⇒ *'a list list* ⇒ *'a mat* **where**
mat-of-rows-list nc rs = *mat* (*length rs*) *nc* (λ (*i*,*j*). *rs* ! *i* ! *j*)

lemma *mat-of-rows-carrier*[*simp*]:
mat-of-rows n vs ∈ *carrier-mat* (*length vs*) *n*
dim-row (*mat-of-rows n vs*) = *length vs*
dim-col (*mat-of-rows n vs*) = *n*
⟨*proof*⟩

lemma *mat-of-rows-row*[*simp*]:
assumes *i*:*i* < *length vs* **and** *n*: *vs* ! *i* ∈ *carrier-vec n*
shows *row* (*mat-of-rows n vs*) *i* = *vs* ! *i*
⟨*proof*⟩

lemma *rows-mat-of-rows*[*simp*]:
assumes *set vs* ⊆ *carrier-vec n* **shows** *rows* (*mat-of-rows n vs*) = *vs*
⟨*proof*⟩

lemma *mat-of-rows-rows*[*simp*]:
mat-of-rows (*dim-col A*) (*rows A*) = *A*
⟨*proof*⟩

definition *col* :: *'a mat* ⇒ *nat* ⇒ *'a vec* **where**
col A j = *vec* (*dim-row A*) (λ *i*. *A* $$ (*i*,*j*))

definition *cols* :: *'a mat* ⇒ *'a vec list* **where**
cols A = *map* (*col A*) [*0*..<*dim-col A*]

definition *mat-of-cols* :: *nat* ⇒ *'a vec list* ⇒ *'a mat*
where *mat-of-cols n cs* = *mat n* (*length cs*) (λ(*i*,*j*). *cs* ! *j* $ *i*)

definition *mat-of-cols-list* :: *nat* ⇒ *'a list list* ⇒ *'a mat* **where**
mat-of-cols-list nr cs = *mat nr* (*length cs*) (λ (*i*,*j*). *cs* ! *j* ! *i*)

lemma *col-dim*[*simp*]: *col A i* ∈ *carrier-vec* (*dim-row A*) ⟨*proof*⟩

lemma *dim-col*[*simp*]: *dim-vec* (*col A i*) = *dim-row A* ⟨*proof*⟩

lemma *cols-dim*[*simp*]: *set* (*cols A*) ⊆ *carrier-vec* (*dim-row A*) ⟨*proof*⟩

lemma *cols-length*[*simp*]: *length* (*cols A*) = *dim-col A* ⟨*proof*⟩

lemma *cols-nth*[*simp*]: *i* < *dim-col A* ⟹ *cols A* ! *i* = *col A i*
⟨*proof*⟩

lemma *mat-of-cols-carrier*[*simp*]:
mat-of-cols n vs ∈ *carrier-mat n* (*length vs*)
dim-row (*mat-of-cols n vs*) = *n*
dim-col (*mat-of-cols n vs*) = *length vs*
⟨*proof*⟩

lemma *col-mat-of-cols*[*simp*]:
assumes *j*:*j* < *length vs* **and** *n*: *vs* ! *j* ∈ *carrier-vec n*
shows *col* (*mat-of-cols n vs*) *j* = *vs* ! *j*
⟨*proof*⟩

lemma *cols-mat-of-cols*[*simp*]:
assumes *set vs* ⊆ *carrier-vec n* **shows** *cols* (*mat-of-cols n vs*) = *vs*
⟨*proof*⟩

lemma *mat-of-cols-cols*[*simp*]:
mat-of-cols (*dim-row A*) (*cols A*) = *A*
⟨*proof*⟩

instantiation *mat* :: (*ord*) *ord*
begin

definition *less-eq-mat* :: ′*a mat* ⇒ ′*a mat* ⇒ *bool* **where**
less-eq-mat A B = (*dim-row A* = *dim-row B* ∧ *dim-col A* = *dim-col B* ∧
(∀ *i* < *dim-row B*. ∀ *j* < *dim-col B*. *A* $$ (*i*,*j*) ≤ *B* $$ (*i*,*j*)))

definition *less-mat* :: ′*a mat* ⇒ ′*a mat* ⇒ *bool* **where**
less-mat A B = (*A* ≤ *B* ∧ ¬ (*B* ≤ *A*))
instance ⟨*proof*⟩
end

instantiation *mat* :: (*preorder*) *preorder*
begin
instance
⟨*proof*⟩
end

instantiation *mat* :: (*order*) *order*
begin
instance
⟨*proof*⟩
end

instantiation *mat* :: (*plus*) *plus*
begin
definition *plus-mat* :: (′*a* :: *plus*) *mat* ⇒ ′*a mat* ⇒ ′*a mat* **where**
A + *B* ≡ *mat* (*dim-row B*) (*dim-col B*) (λ *ij*. *A* $$ *ij* + *B* $$ *ij*)
instance ⟨*proof*⟩

end

definition *map-mat* :: (*'a* ⇒ *'b*) ⇒ *'a mat* ⇒ *'b mat* **where**
map-mat f A ≡ *mat* (*dim-row A*) (*dim-col A*) (λ *ij*. *f* (*A* $$ *ij*))

definition *smult-mat* :: *'a* :: *times* ⇒ *'a mat* ⇒ *'a mat* (**infixl** $\cdot_m$ *70*)
where *a* $\cdot_m$ *A* ≡ *map-mat* (λ *b*. *a* ∗ *b*) *A*

definition *zero-mat* :: *nat* ⇒ *nat* ⇒ *'a* :: *zero mat* (0_m) **where**
0_m *nr nc* ≡ *mat nr nc* (λ *ij*. *0*)

lemma *elements-0-mat* [*simp*]: *elements-mat* (0_m *nr nc*) ⊆ {*0*}
⟨*proof*⟩

definition *transpose-mat* :: *'a mat* ⇒ *'a mat* **where**
transpose-mat A ≡ *mat* (*dim-col A*) (*dim-row A*) (λ (*i*,*j*). *A* $$ (*j*,*i*))

definition *one-mat* :: *nat* ⇒ *'a* :: {*zero*,*one*} *mat* (1_m) **where**
1_m *n* ≡ *mat n n* (λ (*i*,*j*). *if i* = *j then 1 else 0*)

instantiation *mat* :: (*uminus*) *uminus*
begin
definition *uminus-mat* :: *'a* :: *uminus mat* ⇒ *'a mat* **where**
− *A* ≡ *mat* (*dim-row A*) (*dim-col A*) (λ *ij*. − (*A* $$ *ij*))
instance ⟨*proof*⟩
end

instantiation *mat* :: (*minus*) *minus*
begin
definition *minus-mat* :: (*'a* :: *minus*) *mat* ⇒ *'a mat* ⇒ *'a mat* **where**
A − *B* ≡ *mat* (*dim-row B*) (*dim-col B*) (λ *ij*. *A* $$ *ij* − *B* $$ *ij*)
instance ⟨*proof*⟩
end

instantiation *mat* :: (*semiring-0*) *times*
begin
definition *times-mat* :: *'a* :: *semiring-0 mat* ⇒ *'a mat* ⇒ *'a mat*
where *A* ∗ *B* ≡ *mat* (*dim-row A*) (*dim-col B*) (λ (*i*,*j*). *row A i* · *col B j*)
instance ⟨*proof*⟩
end

definition *mult-mat-vec* :: *'a* :: *semiring-0 mat* ⇒ *'a vec* ⇒ *'a vec* (**infixl** $*_v$ *70*)
where *A* $*_v$ *v* ≡ *vec* (*dim-row A*) (λ *i*. *row A i* · *v*)

definition *inverts-mat* :: *'a* :: *semiring-1 mat* ⇒ *'a mat* ⇒ *bool* **where**
inverts-mat A B ≡ *A* ∗ *B* = 1_m (*dim-row A*)

definition *invertible-mat* :: *'a* :: *semiring-1 mat* ⇒ *bool*
where *invertible-mat A* ≡ *square-mat A* ∧ (∃ *B*. *inverts-mat A B* ∧ *inverts-mat*

B A)

definition *monoid-mat* :: *'a* :: *monoid-add itself* ⇒ *nat* ⇒ *nat* ⇒ *'a mat monoid*
where
  monoid-mat ty nr nc ≡ ⦇
    carrier = *carrier-mat nr nc*,
    mult = (+),
    one = 0_m *nr nc*⦈

definition *ring-mat* :: *'a* :: *semiring-1 itself* ⇒ *nat* ⇒ *'b* ⇒ (*'a mat*,*'b*) *ring-scheme*
where
  ring-mat ty n b ≡ ⦇
    carrier = *carrier-mat n n*,
    mult = (∗),
    one = 1_m *n*,
    zero = 0_m *n n*,
    add = (+),
    ... = *b*⦈

definition *module-mat* :: *'a* :: *semiring-1 itself* ⇒ *nat* ⇒ *nat* ⇒ (*'a*,*'a mat*)*module*
where
  module-mat ty nr nc ≡ ⦇
    carrier = *carrier-mat nr nc*,
    mult = (∗),
    one = 1_m *nr*,
    zero = 0_m *nr nc*,
    add = (+),
    smult = ($\cdot_m$)⦈

lemma *ring-mat-simps*:
  mult (*ring-mat ty n b*) = (∗)
  add (*ring-mat ty n b*) = (+)
  one (*ring-mat ty n b*) = 1_m *n*
  zero (*ring-mat ty n b*) = 0_m *n n*
  carrier (*ring-mat ty n b*) = *carrier-mat n n*
  ⟨*proof*⟩

lemma *module-mat-simps*:
  mult (*module-mat ty nr nc*) = (∗)
  add (*module-mat ty nr nc*) = (+)
  one (*module-mat ty nr nc*) = 1_m *nr*
  zero (*module-mat ty nr nc*) = 0_m *nr nc*
  carrier (*module-mat ty nr nc*) = *carrier-mat nr nc*
  smult (*module-mat ty nr nc*) = ($\cdot_m$)
  ⟨*proof*⟩

lemma *index-zero-mat*[*simp*]: *i* < *nr* ⟹ *j* < *nc* ⟹ 0_m *nr nc* $$ (*i*,*j*) = *0*
  dim-row (0_m *nr nc*) = *nr* *dim-col* (0_m *nr nc*) = *nc*
  ⟨*proof*⟩

lemma *index-one-mat*[*simp*]: $i < n \Longrightarrow j < n \Longrightarrow 1_m\ n\ \$\$\ (i,j) = (\textit{if } i = j \textit{ then } 1 \textit{ else } 0)$
$\textit{dim-row}\ (1_m\ n) = n$ $\textit{dim-col}\ (1_m\ n) = n$
⟨*proof*⟩

lemma *index-add-mat*[*simp*]:
$i < \textit{dim-row}\ B \Longrightarrow j < \textit{dim-col}\ B \Longrightarrow (A + B)\ \$\$\ (i,j) = A\ \$\$\ (i,j) + B\ \$\$\ (i,j)$
$\textit{dim-row}\ (A + B) = \textit{dim-row}\ B$ $\textit{dim-col}\ (A + B) = \textit{dim-col}\ B$
⟨*proof*⟩

lemma *index-minus-mat*[*simp*]:
$i < \textit{dim-row}\ B \Longrightarrow j < \textit{dim-col}\ B \Longrightarrow (A - B)\ \$\$\ (i,j) = A\ \$\$\ (i,j) - B\ \$\$\ (i,j)$
$\textit{dim-row}\ (A - B) = \textit{dim-row}\ B$ $\textit{dim-col}\ (A - B) = \textit{dim-col}\ B$
⟨*proof*⟩

lemma *index-map-mat*[*simp*]:
$i < \textit{dim-row}\ A \Longrightarrow j < \textit{dim-col}\ A \Longrightarrow \textit{map-mat}\ f\ A\ \$\$\ (i,j) = f\ (A\ \$\$\ (i,j))$
$\textit{dim-row}\ (\textit{map-mat}\ f\ A) = \textit{dim-row}\ A$ $\textit{dim-col}\ (\textit{map-mat}\ f\ A) = \textit{dim-col}\ A$
⟨*proof*⟩

lemma *index-smult-mat*[*simp*]:
$i < \textit{dim-row}\ A \Longrightarrow j < \textit{dim-col}\ A \Longrightarrow (a \cdot_m A)\ \$\$\ (i,j) = a * A\ \$\$\ (i,j)$
$\textit{dim-row}\ (a \cdot_m A) = \textit{dim-row}\ A$ $\textit{dim-col}\ (a \cdot_m A) = \textit{dim-col}\ A$
⟨*proof*⟩

lemma *index-uminus-mat*[*simp*]:
$i < \textit{dim-row}\ A \Longrightarrow j < \textit{dim-col}\ A \Longrightarrow (-\ A)\ \$\$\ (i,j) = -\ (A\ \$\$\ (i,j))$
$\textit{dim-row}\ (-\ A) = \textit{dim-row}\ A$ $\textit{dim-col}\ (-\ A) = \textit{dim-col}\ A$
⟨*proof*⟩

lemma *index-transpose-mat*[*simp*]:
$i < \textit{dim-col}\ A \Longrightarrow j < \textit{dim-row}\ A \Longrightarrow \textit{transpose-mat}\ A\ \$\$\ (i,j) = A\ \$\$\ (j,i)$
$\textit{dim-row}\ (\textit{transpose-mat}\ A) = \textit{dim-col}\ A$ $\textit{dim-col}\ (\textit{transpose-mat}\ A) = \textit{dim-row}\ A$
⟨*proof*⟩

lemma *index-mult-mat*[*simp*]:
$i < \textit{dim-row}\ A \Longrightarrow j < \textit{dim-col}\ B \Longrightarrow (A * B)\ \$\$\ (i,j) = \textit{row}\ A\ i \cdot \textit{col}\ B\ j$
$\textit{dim-row}\ (A * B) = \textit{dim-row}\ A$ $\textit{dim-col}\ (A * B) = \textit{dim-col}\ B$
⟨*proof*⟩

lemma *dim-mult-mat-vec*[*simp*]: $\textit{dim-vec}\ (A *_v v) = \textit{dim-row}\ A$
⟨*proof*⟩

lemma *index-mult-mat-vec*[*simp*]: $i < \textit{dim-row}\ A \Longrightarrow (A *_v v)\ \$\ i = \textit{row}\ A\ i \cdot v$
⟨*proof*⟩

lemma *index-row*[*simp*]:
$i < \textit{dim-row}\ A \Longrightarrow j < \textit{dim-col}\ A \Longrightarrow \textit{row}\ A\ i\ \$\ j = A\ \$\$\ (i,j)$

dim-vec (*row A i*) = *dim-col A*
⟨*proof*⟩

lemma *index-col*[*simp*]: $i < \textit{dim-row}\ A \Longrightarrow j < \textit{dim-col}\ A \Longrightarrow \textit{col}\ A\ j\ \$\ i = A\ \$\$\ (i,j)$
⟨*proof*⟩

lemma *upper-triangular-one*[*simp*]: *upper-triangular* (1_m *n*)
⟨*proof*⟩

lemma *upper-triangular-zero*[*simp*]: *upper-triangular* (0_m *n n*)
⟨*proof*⟩

lemma *mat-row-carrierI*[*intro*,*simp*]: $\textit{mat}_r$ *nr nc r* ∈ *carrier-mat nr nc*
⟨*proof*⟩

lemma *eq-rowI*: **assumes** *rows*: $\bigwedge i.\ i < \textit{dim-row}\ B \Longrightarrow \textit{row}\ A\ i = \textit{row}\ B\ i$
and *dims*: *dim-row A* = *dim-row B* *dim-col A* = *dim-col B*
shows *A* = *B*
⟨*proof*⟩

lemma *elements-mat-map*[*simp*]: *elements-mat* (*map-mat f A*) = *f* ‘ *elements-mat A*
⟨*proof*⟩

lemma *row-mat*[*simp*]: $i < nr \Longrightarrow \textit{row}\ (\textit{mat}\ nr\ nc\ f)\ i = \textit{vec}\ nc\ (\lambda\ j.\ f\ (i,j))$
⟨*proof*⟩

lemma *col-mat*[*simp*]: $j < nc \Longrightarrow \textit{col}\ (\textit{mat}\ nr\ nc\ f)\ j = \textit{vec}\ nr\ (\lambda\ i.\ f\ (i,j))$
⟨*proof*⟩

lemma *zero-carrier-mat*[*simp*]: 0_m *nr nc* ∈ *carrier-mat nr nc*
⟨*proof*⟩

lemma *smult-carrier-mat*[*simp*]:
$A \in \textit{carrier-mat}\ nr\ nc \Longrightarrow k \cdot_m A \in \textit{carrier-mat}\ nr\ nc$
⟨*proof*⟩

lemma *add-carrier-mat*[*simp*]:
$B \in \textit{carrier-mat}\ nr\ nc \Longrightarrow A + B \in \textit{carrier-mat}\ nr\ nc$
⟨*proof*⟩

lemma *one-carrier-mat*[*simp*]: 1_m *n* ∈ *carrier-mat n n*
⟨*proof*⟩

lemma *uminus-carrier-mat*:
$A \in \textit{carrier-mat}\ nr\ nc \Longrightarrow (-\ A \in \textit{carrier-mat}\ nr\ nc)$
⟨*proof*⟩

lemma *uminus-carrier-iff-mat*[*simp*]:
$(-\ A \in \mathit{carrier\text{-}mat}\ nr\ nc) = (A \in \mathit{carrier\text{-}mat}\ nr\ nc)$
⟨*proof*⟩

lemma *minus-carrier-mat*:
$B \in \mathit{carrier\text{-}mat}\ nr\ nc \Longrightarrow (A - B \in \mathit{carrier\text{-}mat}\ nr\ nc)$
⟨*proof*⟩

lemma *transpose-carrier-mat*[*simp*]: $(\mathit{transpose\text{-}mat}\ A \in \mathit{carrier\text{-}mat}\ nc\ nr) = (A \in \mathit{carrier\text{-}mat}\ nr\ nc)$
⟨*proof*⟩

lemma *row-carrier-vec*[*simp*]: $i < nr \Longrightarrow A \in \mathit{carrier\text{-}mat}\ nr\ nc \Longrightarrow \mathit{row}\ A\ i \in \mathit{carrier\text{-}vec}\ nc$
⟨*proof*⟩

lemma *col-carrier-vec*[*simp*]: $j < nc \Longrightarrow A \in \mathit{carrier\text{-}mat}\ nr\ nc \Longrightarrow \mathit{col}\ A\ j \in \mathit{carrier\text{-}vec}\ nr$
⟨*proof*⟩

lemma *mult-carrier-mat*[*simp*]:
$A \in \mathit{carrier\text{-}mat}\ nr\ n \Longrightarrow B \in \mathit{carrier\text{-}mat}\ n\ nc \Longrightarrow A * B \in \mathit{carrier\text{-}mat}\ nr\ nc$
⟨*proof*⟩

lemma *mult-mat-vec-carrier*[*simp*]:
$A \in \mathit{carrier\text{-}mat}\ nr\ n \Longrightarrow v \in \mathit{carrier\text{-}vec}\ n \Longrightarrow A *_v v \in \mathit{carrier\text{-}vec}\ nr$
⟨*proof*⟩

lemma *comm-add-mat*[*ac-simps*]:
$(A :: {'a} :: \mathit{comm\text{-}monoid\text{-}add}\ \mathit{mat}) \in \mathit{carrier\text{-}mat}\ nr\ nc \Longrightarrow B \in \mathit{carrier\text{-}mat}\ nr\ nc \Longrightarrow A + B = B + A$
⟨*proof*⟩

lemma *minus-r-inv-mat*[*simp*]:
$(A :: {'a} :: \mathit{group\text{-}add}\ \mathit{mat}) \in \mathit{carrier\text{-}mat}\ nr\ nc \Longrightarrow (A - A) = 0_m\ nr\ nc$
⟨*proof*⟩

lemma *uminus-l-inv-mat*[*simp*]:
$(A :: {'a} :: \mathit{group\text{-}add}\ \mathit{mat}) \in \mathit{carrier\text{-}mat}\ nr\ nc \Longrightarrow (-\ A + A) = 0_m\ nr\ nc$
⟨*proof*⟩

lemma *add-inv-exists-mat*:
$(A :: {'a} :: \mathit{group\text{-}add}\ \mathit{mat}) \in \mathit{carrier\text{-}mat}\ nr\ nc \Longrightarrow \exists\ B \in \mathit{carrier\text{-}mat}\ nr\ nc.\ B + A = 0_m\ nr\ nc \wedge A + B = 0_m\ nr\ nc$
⟨*proof*⟩

lemma *assoc-add-mat*[*simp*]:

(*A* :: *'a* :: *monoid-add mat*) ∈ *carrier-mat nr nc* ⟹ *B* ∈ *carrier-mat nr nc* ⟹ *C* ∈ *carrier-mat nr nc*
⟹ (*A* + *B*) + *C* = *A* + (*B* + *C*)
⟨*proof*⟩

lemma *uminus-add-mat*: **fixes** *A* :: *'a* :: *group-add mat*
assumes *A* ∈ *carrier-mat nr nc*
and *B* ∈ *carrier-mat nr nc*
shows − (*A* + *B*) = − *B* + − *A*
⟨*proof*⟩

lemma *transpose-transpose*[*simp*]:
transpose-mat (*transpose-mat A*) = *A*
⟨*proof*⟩

lemma *transpose-one*[*simp*]: *transpose-mat* (1_m *n*) = (1_m *n*)
⟨*proof*⟩

lemma *row-transpose*[*simp*]:
j < *dim-col A* ⟹ *row* (*transpose-mat A*) *j* = *col A j*
⟨*proof*⟩

lemma *col-transpose*[*simp*]:
i < *dim-row A* ⟹ *col* (*transpose-mat A*) *i* = *row A i*
⟨*proof*⟩

lemma *row-zero*[*simp*]:
i < *nr* ⟹ *row* (0_m *nr nc*) *i* = 0_v *nc*
⟨*proof*⟩

lemma *col-zero*[*simp*]:
j < *nc* ⟹ *col* (0_m *nr nc*) *j* = 0_v *nr*
⟨*proof*⟩

lemma *row-one*[*simp*]:
i < *n* ⟹ *row* (1_m *n*) *i* = *unit-vec n i*
⟨*proof*⟩

lemma *col-one*[*simp*]:
j < *n* ⟹ *col* (1_m *n*) *j* = *unit-vec n j*
⟨*proof*⟩

lemma *transpose-add*: *A* ∈ *carrier-mat nr nc* ⟹ *B* ∈ *carrier-mat nr nc*
⟹ *transpose-mat* (*A* + *B*) = *transpose-mat A* + *transpose-mat B*
⟨*proof*⟩

lemma *transpose-minus*: *A* ∈ *carrier-mat nr nc* ⟹ *B* ∈ *carrier-mat nr nc*
⟹ *transpose-mat* (*A* − *B*) = *transpose-mat A* − *transpose-mat B*
⟨*proof*⟩

lemma *transpose-uminus*: $\textit{transpose-mat}\ (-\ A) = -\ (\textit{transpose-mat}\ A)$
⟨*proof*⟩

lemma *row-add*[*simp*]:
$A \in \textit{carrier-mat}\ nr\ nc \Longrightarrow B \in \textit{carrier-mat}\ nr\ nc \Longrightarrow i < nr$
$\Longrightarrow \textit{row}\ (A + B)\ i = \textit{row}\ A\ i + \textit{row}\ B\ i$
$i < \textit{dim-row}\ A \Longrightarrow \textit{dim-row}\ B = \textit{dim-row}\ A \Longrightarrow \textit{dim-col}\ B = \textit{dim-col}\ A \Longrightarrow \textit{row}\ (A + B)\ i = \textit{row}\ A\ i + \textit{row}\ B\ i$
⟨*proof*⟩

lemma *col-add*[*simp*]:
$A \in \textit{carrier-mat}\ nr\ nc \Longrightarrow B \in \textit{carrier-mat}\ nr\ nc \Longrightarrow j < nc$
$\Longrightarrow \textit{col}\ (A + B)\ j = \textit{col}\ A\ j + \textit{col}\ B\ j$
⟨*proof*⟩

lemma *row-mult*[*simp*]: **assumes** *m*: $A \in \textit{carrier-mat}\ nr\ n\ B \in \textit{carrier-mat}\ n\ nc$
and *i*: $i < nr$
shows $\textit{row}\ (A * B)\ i = \textit{vec}\ nc\ (\lambda\ j.\ \textit{row}\ A\ i \cdot \textit{col}\ B\ j)$
⟨*proof*⟩

lemma *col-mult*[*simp*]: **assumes** *m*: $A \in \textit{carrier-mat}\ nr\ n\ B \in \textit{carrier-mat}\ n\ nc$
and *j*: $j < nc$
shows $\textit{col}\ (A * B)\ j = \textit{vec}\ nr\ (\lambda\ i.\ \textit{row}\ A\ i \cdot \textit{col}\ B\ j)$
⟨*proof*⟩

lemma *transpose-mult*:
$(A :: 'a :: \textit{comm-semiring-0 mat}) \in \textit{carrier-mat}\ nr\ n \Longrightarrow B \in \textit{carrier-mat}\ n\ nc$
$\Longrightarrow \textit{transpose-mat}\ (A * B) = \textit{transpose-mat}\ B * \textit{transpose-mat}\ A$
⟨*proof*⟩

lemma *left-add-zero-mat*[*simp*]:
$(A :: 'a :: \textit{monoid-add mat}) \in \textit{carrier-mat}\ nr\ nc \ \Longrightarrow 0_m\ nr\ nc + A = A$
⟨*proof*⟩

lemma *add-uminus-minus-mat*: $A \in \textit{carrier-mat}\ nr\ nc \Longrightarrow B \in \textit{carrier-mat}\ nr\ nc \Longrightarrow$
$A + (-\ B) = A - (B :: 'a :: \textit{group-add mat})$
⟨*proof*⟩

lemma *right-add-zero-mat*[*simp*]: $A \in \textit{carrier-mat}\ nr\ nc \Longrightarrow$
$A + 0_m\ nr\ nc = (A :: 'a :: \textit{monoid-add mat})$
⟨*proof*⟩

lemma *left-mult-zero-mat*:
$A \in \textit{carrier-mat}\ n\ nc \Longrightarrow 0_m\ nr\ n * A = 0_m\ nr\ nc$
⟨*proof*⟩

lemma *left-mult-zero-mat'*[*simp*]: $\textit{dim-row}\ A = n \Longrightarrow 0_m\ nr\ n * A = 0_m\ nr$

(*dim-col A*)
  ⟨*proof*⟩

lemma *right-mult-zero-mat*:
  $A \in$ *carrier-mat nr n* $\Longrightarrow A * 0_m$ *n nc* $= 0_m$ *nr nc*
  ⟨*proof*⟩

lemma *right-mult-zero-mat'*[*simp*]: *dim-col A* $= n \Longrightarrow A * 0_m$ *n nc* $= 0_m$ (*dim-row A*) *nc*
  ⟨*proof*⟩

lemma *left-mult-one-mat*:
  (*A* :: *'a* :: *semiring-1 mat*) $\in$ *carrier-mat nr nc* $\Longrightarrow 1_m$ *nr* $* A = A$
  ⟨*proof*⟩

lemma *left-mult-one-mat'*[*simp*]: *dim-row* (*A* :: *'a* :: *semiring-1 mat*) $= n \Longrightarrow 1_m$ $n * A = A$
  ⟨*proof*⟩

lemma *right-mult-one-mat*:
  (*A* :: *'a* :: *semiring-1 mat*) $\in$ *carrier-mat nr nc* $\Longrightarrow A * 1_m$ *nc* $= A$
  ⟨*proof*⟩

lemma *right-mult-one-mat'*[*simp*]: *dim-col* (*A* :: *'a* :: *semiring-1 mat*) $= n \Longrightarrow A * 1_m$ $n = A$
  ⟨*proof*⟩

lemma *one-mult-mat-vec*[*simp*]:
  (*v* :: *'a* :: *semiring-1 vec*) $\in$ *carrier-vec n* $\Longrightarrow 1_m$ $n *_v v = v$
  ⟨*proof*⟩

lemma *minus-add-uminus-mat*: **fixes** *A* :: *'a* :: *group-add mat*
  shows $A \in$ *carrier-mat nr nc* $\Longrightarrow B \in$ *carrier-mat nr nc* $\Longrightarrow$
  $A - B = A + (- B)$
  ⟨*proof*⟩

lemma *add-mult-distrib-mat*[*algebra-simps*]: **assumes** *m*: $A \in$ *carrier-mat nr n*
  $B \in$ *carrier-mat nr n* $C \in$ *carrier-mat n nc*
  shows $(A + B) * C = A * C + B * C$
  ⟨*proof*⟩

lemma *mult-add-distrib-mat*[*algebra-simps*]: **assumes** *m*: $A \in$ *carrier-mat nr n*
  $B \in$ *carrier-mat n nc* $C \in$ *carrier-mat n nc*
  shows $A * (B + C) = A * B + A * C$
  ⟨*proof*⟩

lemma *add-mult-distrib-mat-vec*[*algebra-simps*]: **assumes** *m*: $A \in$ *carrier-mat nr nc*
  $B \in$ *carrier-mat nr nc* $v \in$ *carrier-vec nc*

shows $(A + B) *_v v = A *_v v + B *_v v$
⟨*proof*⟩

lemma *mult-add-distrib-mat-vec*[*algebra-simps*]: **assumes** *m*: $A \in$ *carrier-mat nr nc*
$v_1 \in$ *carrier-vec nc* $v_2 \in$ *carrier-vec nc*
shows $A *_v (v_1 + v_2) = A *_v v_1 + A *_v v_2$
⟨*proof*⟩

lemma *mult-mat-vec*:
assumes *m*: (*A*::*'a*::*field mat*) $\in$ *carrier-mat nr nc* **and** *v*: $v \in$ *carrier-vec nc*
shows $A *_v (k \cdot_v v) = k \cdot_v (A *_v v)$ (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *assoc-scalar-prod*: **assumes** $*$: $v_1 \in$ *carrier-vec nr* $A \in$ *carrier-mat nr nc* $v_2 \in$ *carrier-vec nc*
shows *vec nc* ($\lambda j.$ $v_1 \cdot$ *col A j*) $\cdot\ v_2 = v_1 \cdot$ *vec nr* ($\lambda i.$ *row A i* $\cdot\ v_2$)
⟨*proof*⟩

lemma *transpose-vec-mult-scalar*:
fixes *A* :: *'a* :: *comm-semiring-0 mat*
assumes *A*: $A \in$ *carrier-mat nr nc*
and *x*: $x \in$ *carrier-vec nc*
and *y*: $y \in$ *carrier-vec nr*
shows (*transpose-mat A* $*_v$ *y*) $\cdot\ x = y \cdot (A *_v x)$
⟨*proof*⟩

lemma *assoc-mult-mat*[*simp*]:
$A \in$ *carrier-mat* $n_1\ n_2 \Longrightarrow B \in$ *carrier-mat* $n_2\ n_3 \Longrightarrow C \in$ *carrier-mat* $n_3\ n_4$
$\Longrightarrow (A * B) * C = A * (B * C)$
⟨*proof*⟩

lemma *assoc-mult-mat-vec*[*simp*]:
$A \in$ *carrier-mat* $n_1\ n_2 \Longrightarrow B \in$ *carrier-mat* $n_2\ n_3 \Longrightarrow v \in$ *carrier-vec* n_3
$\Longrightarrow (A * B) *_v v = A *_v (B *_v v)$
⟨*proof*⟩

lemma *comm-monoid-mat*: *comm-monoid* (*monoid-mat TYPE*(*'a* :: *comm-monoid-add*) *nr nc*)
⟨*proof*⟩

lemma *comm-group-mat*: *comm-group* (*monoid-mat TYPE*(*'a* :: *ab-group-add*) *nr nc*)
⟨*proof*⟩

lemma *semiring-mat*: *semiring* (*ring-mat TYPE*(*'a* :: *semiring-1*) *n b*)
⟨*proof*⟩

lemma *ring-mat*: *ring* (*ring-mat TYPE*(*'a* :: *comm-ring-1*) *n b*)

⟨*proof*⟩

lemma *abelian-group-mat*: *abelian-group* (*module-mat TYPE*(*'a* :: *comm-ring-1*) *nr nc*)
⟨*proof*⟩

lemma *row-smult*[*simp*]: **assumes** *i*: $i < dim\text{-}row\ A$
shows $row\ (k \cdot_m A)\ i = k \cdot_v (row\ A\ i)$
⟨*proof*⟩

lemma *col-smult*[*simp*]: **assumes** *i*: $i < dim\text{-}col\ A$
shows $col\ (k \cdot_m A)\ i = k \cdot_v (col\ A\ i)$
⟨*proof*⟩

lemma *row-uminus*[*simp*]: **assumes** *i*: $i < dim\text{-}row\ A$
shows $row\ (-\ A)\ i = -\ (row\ A\ i)$
⟨*proof*⟩

lemma *scalar-prod-uminus-left*[*simp*]: **assumes** *dim*: $dim\text{-}vec\ v = dim\text{-}vec\ (w :: 'a :: ring\ vec)$
shows $-\ v \cdot w = -\ (v \cdot w)$
⟨*proof*⟩

lemma *col-uminus*[*simp*]: **assumes** *i*: $i < dim\text{-}col\ A$
shows $col\ (-\ A)\ i = -\ (col\ A\ i)$
⟨*proof*⟩

lemma *scalar-prod-uminus-right*[*simp*]: **assumes** *dim*: $dim\text{-}vec\ v = dim\text{-}vec\ (w :: 'a :: ring\ vec)$
shows $v \cdot -\ w = -\ (v \cdot w)$
⟨*proof*⟩

context fixes $A\ B :: 'a :: ring\ mat$
assumes *dim*: $dim\text{-}col\ A = dim\text{-}row\ B$
begin
lemma *uminus-mult-left-mat*[*simp*]: $(-\ A * B) = -\ (A * B)$
⟨*proof*⟩

lemma *uminus-mult-right-mat*[*simp*]: $(A * -\ B) = -\ (A * B)$
⟨*proof*⟩
end

lemma *minus-mult-distrib-mat*[*algebra-simps*]: **fixes** $A :: 'a :: ring\ mat$
assumes *m*: $A \in carrier\text{-}mat\ nr\ n$ $B \in carrier\text{-}mat\ nr\ n$ $C \in carrier\text{-}mat\ n\ nc$
shows $(A - B) * C = A * C - B * C$
⟨*proof*⟩

lemma *minus-mult-distrib-mat-vec*[*algebra-simps*]: **assumes** *A*: $(A :: 'a :: ring\ mat) \in carrier\text{-}mat\ nr\ nc$

and B: $B \in \textit{carrier-mat nr nc}$
and v: $v \in \textit{carrier-vec nc}$
shows $(A - B) *_v v = A *_v v - B *_v v$
$\langle \textit{proof} \rangle$

lemma *mult-minus-distrib-mat-vec*[*algebra-simps*]: **assumes** A: $(A :: \textit{'a} :: \textit{ring mat}) \in \textit{carrier-mat nr nc}$
and v: $v \in \textit{carrier-vec nc}$
and w: $w \in \textit{carrier-vec nc}$
shows $A *_v (v - w) = A *_v v - A *_v w$
$\langle \textit{proof} \rangle$

lemma *mult-minus-distrib-mat*[*algebra-simps*]: **fixes** $A :: \textit{'a} :: \textit{ring mat}$
assumes m: $A \in \textit{carrier-mat nr n}$ $B \in \textit{carrier-mat n nc}$ $C \in \textit{carrier-mat n nc}$
shows $A * (B - C) = A * B - A * C$
$\langle \textit{proof} \rangle$

lemma *uminus-mult-mat-vec*[*simp*]: **assumes** v: $\textit{dim-vec } v = \textit{dim-col } (A :: \textit{'a} :: \textit{ring mat})$
shows $- A *_v v = - (A *_v v)$
$\langle \textit{proof} \rangle$

lemma *uminus-zero-vec-eq*: **assumes** v: $(v :: \textit{'a} :: \textit{group-add vec}) \in \textit{carrier-vec n}$
shows $(- v = 0_v\ n) = (v = 0_v\ n)$
$\langle \textit{proof} \rangle$

lemma *map-carrier-mat*[*simp*]:
$(\textit{map-mat } f\ A \in \textit{carrier-mat nr nc}) = (A \in \textit{carrier-mat nr nc})$
$\langle \textit{proof} \rangle$

lemma *col-map-mat*[*simp*]:
assumes $j < \textit{dim-col } A$ **shows** $\textit{col } (\textit{map-mat } f\ A)\ j = \textit{map-vec } f\ (\textit{col } A\ j)$
$\langle \textit{proof} \rangle$

lemma *scalar-vec-one*[*simp*]: $1 \cdot_v (v :: \textit{'a} :: \textit{semiring-1 vec}) = v$
$\langle \textit{proof} \rangle$

lemma *scalar-prod-smult-right*[*simp*]:
$\textit{dim-vec } w = \textit{dim-vec } v \Longrightarrow w \bullet (k \cdot_v v) = (k :: \textit{'a} :: \textit{comm-semiring-0}) * (w \bullet v)$
$\langle \textit{proof} \rangle$

lemma *scalar-prod-smult-left*[*simp*]:
$\textit{dim-vec } w = \textit{dim-vec } v \Longrightarrow (k \cdot_v w) \bullet v = (k :: \textit{'a} :: \textit{comm-semiring-0}) * (w \bullet v)$
$\langle \textit{proof} \rangle$

lemma *mult-smult-distrib*: **assumes** A: $A \in \textit{carrier-mat nr n}$ **and** B: $B \in \textit{carrier-mat n nc}$
shows $A * (k \cdot_m B) = (k :: \textit{'a} :: \textit{comm-semiring-0}) \cdot_m (A * B)$
$\langle \textit{proof} \rangle$

lemma *add-smult-distrib-left-mat*: **assumes** $A \in$ *carrier-mat nr nc* $B \in$ *carrier-mat nr nc*
shows $k \cdot_m (A + B) = (k :: {'a} :: \mathit{semiring}) \cdot_m A + k \cdot_m B$
⟨*proof*⟩

lemma *add-smult-distrib-right-mat*: **assumes** $A \in$ *carrier-mat nr nc*
shows $(k + l) \cdot_m A = (k :: {'a} :: \mathit{semiring}) \cdot_m A + l \cdot_m A$
⟨*proof*⟩

lemma *mult-smult-assoc-mat*: **assumes** A: $A \in$ *carrier-mat nr n* **and** B: $B \in$ *carrier-mat n nc*
shows $(k \cdot_m A) * B = (k :: {'a} :: \mathit{comm\text{-}semiring\text{-}0}) \cdot_m (A * B)$
⟨*proof*⟩

definition *similar-mat-wit* :: *'a* :: *semiring-1 mat* ⇒ *'a mat* ⇒ *'a mat* ⇒ *'a mat* ⇒ *bool* **where**
similar-mat-wit A B P Q = (*let n* = *dim-row A in* $\{A,B,P,Q\} \subseteq$ *carrier-mat n n* $\wedge P * Q = 1_m\ n \wedge Q * P = 1_m\ n \wedge$
$A = P * B * Q$)

definition *similar-mat* :: *'a* :: *semiring-1 mat* ⇒ *'a mat* ⇒ *bool* **where**
similar-mat A B = (∃ *P Q*. *similar-mat-wit A B P Q*)

lemma *similar-matD*: **assumes** *similar-mat A B*
shows ∃ *n P Q*. $\{A,B,P,Q\} \subseteq$ *carrier-mat n n* $\wedge P * Q = 1_m\ n \wedge Q * P = 1_m\ n \wedge A = P * B * Q$
⟨*proof*⟩

lemma *similar-matI*: **assumes** $\{A,B,P,Q\} \subseteq$ *carrier-mat n n* $P * Q = 1_m\ n$ $Q * P = 1_m\ n$ $A = P * B * Q$
shows *similar-mat A B* ⟨*proof*⟩

fun *pow-mat* :: *'a* :: *semiring-1 mat* ⇒ *nat* ⇒ *'a mat* (**infixr** $\widehat{}_m$ *75*) **where**
$A\ \widehat{}_m\ 0 = 1_m$ (*dim-row A*)
| $A\ \widehat{}_m$ (*Suc k*) = $A\ \widehat{}_m\ k * A$

lemma *pow-mat-dim*[*simp*]:
dim-row $(A\ \widehat{}_m\ k)$ = *dim-row A*
dim-col $(A\ \widehat{}_m\ k)$ = (*if k* = *0 then dim-row A else dim-col A*)
⟨*proof*⟩

lemma *pow-mat-dim-square*[*simp*]:
$A \in$ *carrier-mat n n* ⟹ *dim-row* $(A\ \widehat{}_m\ k) = n$
$A \in$ *carrier-mat n n* ⟹ *dim-col* $(A\ \widehat{}_m\ k) = n$
⟨*proof*⟩

lemma *pow-carrier-mat*[*simp*]: $A \in$ *carrier-mat n n* ⟹ $A\ \widehat{}_m\ k \in$ *carrier-mat n n*

⟨*proof*⟩

definition *diag-mat* :: *'a mat* ⇒ *'a list* **where**
diag-mat A = *map* (λ *i*. *A* \$\$ (*i*,*i*)) [*0* ..< *dim-row A*]

lemma *prod-list-diag-prod*: *prod-list* (*diag-mat A*) = ($\prod$ *i* = *0* ..< *dim-row A*. *A* \$\$ (*i*,*i*))
⟨*proof*⟩

lemma *diag-mat-transpose*[*simp*]: *dim-row A* = *dim-col A* ⟹
diag-mat (*transpose-mat A*) = *diag-mat A* ⟨*proof*⟩

lemma *diag-mat-zero*[*simp*]: *diag-mat* (0_m *n n*) = *replicate n 0*
⟨*proof*⟩

lemma *diag-mat-one*[*simp*]: *diag-mat* (1_m *n*) = *replicate n 1*
⟨*proof*⟩

lemma *pow-mat-ring-pow*: **assumes** *A*: (*A* :: (*'a* :: *semiring-1*)*mat*) ∈ *carrier-mat n n*
shows $A \;\widehat{}_m\; k = A\ [\widehat{}]_{\mathit{ring\text{-}mat}\ \mathit{TYPE}('a)\ n\ b}\ {}^{k}$
(**is** - = $A\ [\widehat{}]_{?C}\ k$)
⟨*proof*⟩

definition *diagonal-mat* :: *'a*::*zero mat* ⇒ *bool* **where**
diagonal-mat A ≡ ∀ *i*<*dim-row A*. ∀ *j*<*dim-col A*. *i* ≠ *j* ⟶ *A* \$\$ (*i*,*j*) = *0*

definition (**in** *comm-monoid-add*) *sum-mat* :: *'a mat* ⇒ *'a* **where**
sum-mat A = *sum* (λ *ij*. *A* \$\$ *ij*) ({*0* ..< *dim-row A*} × {*0* ..< *dim-col A*})

lemma *sum-mat-0*[*simp*]: *sum-mat* (0_m *nr nc*) = (*0* :: *'a* :: *comm-monoid-add*)
⟨*proof*⟩

lemma *sum-mat-add*: **assumes** *A*: (*A* :: *'a* :: *comm-monoid-add mat*) ∈ *carrier-mat nr nc* **and** *B*: *B* ∈ *carrier-mat nr nc*
shows *sum-mat* (*A* + *B*) = *sum-mat A* + *sum-mat B*
⟨*proof*⟩

4.3 Update Operators

definition *update-vec* :: *'a vec* ⇒ *nat* ⇒ *'a* ⇒ *'a vec* (- $|_v$ - ↦ - [*60*,*61*,*62*] *60*)
where *v* $|_v$ *i* ↦ *a* = *vec* (*dim-vec v*) (λ*i'*. *if i'* = *i then a else v* \$ *i'*)

definition *update-mat* :: *'a mat* ⇒ *nat* × *nat* ⇒ *'a* ⇒ *'a mat* (- $|_m$ - ↦ - [*60*,*61*,*62*] *60*)
where *A* $|_m$ *ij* ↦ *a* = *mat* (*dim-row A*) (*dim-col A*) (λ*ij'*. *if ij'* = *ij then a else A* \$\$ *ij'*)

lemma *dim-update-vec*[*simp*]:

$\textit{dim-vec}\ (v\ |_v\ i \mapsto a) = \textit{dim-vec}\ v$ ⟨*proof*⟩

lemma *index-update-vec1*[*simp*]:
assumes $i < \textit{dim-vec}\ v$ **shows** $(v\ |_v\ i \mapsto a)\ \$\ i = a$
⟨*proof*⟩

lemma *index-update-vec2*[*simp*]:
assumes $i' \neq i$ **shows** $(v\ |_v\ i \mapsto a)\ \$\ i' = v\ \$\ i'$
⟨*proof*⟩

lemma *dim-update-mat*[*simp*]:
$\textit{dim-row}\ (A\ |_m\ ij \mapsto a) = \textit{dim-row}\ A$
$\textit{dim-col}\ (A\ |_m\ ij \mapsto a) = \textit{dim-col}\ A$ ⟨*proof*⟩

lemma *index-update-mat1*[*simp*]:
assumes $i < \textit{dim-row}\ A\ j < \textit{dim-col}\ A$ **shows** $(A\ |_m\ (i,j) \mapsto a)\ \$\$\ (i,j) = a$
⟨*proof*⟩

lemma *index-update-mat2*[*simp*]:
assumes i': $i' < \textit{dim-row}\ A$ **and** j': $j' < \textit{dim-col}\ A$ **and** *neq*: $(i',j') \neq ij$
shows $(A\ |_m\ ij \mapsto a)\ \$\$\ (i',j') = A\ \$\$\ (i',j')$
⟨*proof*⟩

4.4 Block Vectors and Matrices

definition *append-vec* :: $'a\ vec \Rightarrow\ 'a\ vec \Rightarrow\ 'a\ vec$ (**infixr** $@_v$ *65*) **where**
$v\ @_v\ w \equiv \textit{let}\ n = \textit{dim-vec}\ v;\ m = \textit{dim-vec}\ w\ \textit{in}$
$\quad vec\ (n + m)\ (\lambda\ i.\ \textit{if}\ i < n\ \textit{then}\ v\ \$\ i\ \textit{else}\ w\ \$\ (i - n))$

lemma *index-append-vec*[*simp*]: $i < \textit{dim-vec}\ v + \textit{dim-vec}\ w$
$\Longrightarrow (v\ @_v\ w)\ \$\ i = (\textit{if}\ i < \textit{dim-vec}\ v\ \textit{then}\ v\ \$\ i\ \textit{else}\ w\ \$\ (i - \textit{dim-vec}\ v))$
$\textit{dim-vec}\ (v\ @_v\ w) = \textit{dim-vec}\ v + \textit{dim-vec}\ w$
⟨*proof*⟩

lemma *append-carrier-vec*[*simp,intro*]:
$v \in \textit{carrier-vec}\ n1 \Longrightarrow w \in \textit{carrier-vec}\ n2 \Longrightarrow v\ @_v\ w \in \textit{carrier-vec}\ (n1 + n2)$
⟨*proof*⟩

lemma *scalar-prod-append*: **assumes** $v1 \in \textit{carrier-vec}\ n1\ v2 \in \textit{carrier-vec}\ n2$
$w1 \in \textit{carrier-vec}\ n1\ w2 \in \textit{carrier-vec}\ n2$
shows $(v1\ @_v\ v2) \cdot (w1\ @_v\ w2) = v1 \cdot w1 + v2 \cdot w2$
⟨*proof*⟩

definition $\textit{vec-first}\ v\ n \equiv vec\ n\ (\lambda i.\ v\ \$\ i)$
definition $\textit{vec-last}\ v\ n \equiv vec\ n\ (\lambda i.\ v\ \$\ (\textit{dim-vec}\ v - n + i))$

lemma *dim-vec-first*[*simp*]: $\textit{dim-vec}\ (\textit{vec-first}\ v\ n) = n$ ⟨*proof*⟩
lemma *dim-vec-last*[*simp*]: $\textit{dim-vec}\ (\textit{vec-last}\ v\ n) = n$ ⟨*proof*⟩

lemma *vec-first-carrier*[*simp*]: *vec-first v n* ∈ *carrier-vec n* ⟨*proof*⟩
lemma *vec-last-carrier*[*simp*]: *vec-last v n* ∈ *carrier-vec n* ⟨*proof*⟩

lemma *vec-first-last-append*[*simp*]:
  assumes *v* ∈ *carrier-vec* (*n*+*m*) **shows** *vec-first v n* $@_v$ *vec-last v m* = *v*
  ⟨*proof*⟩

lemma *append-vec-le*: **assumes** *v* ∈ *carrier-vec n* **and** *w*: *w* ∈ *carrier-vec n*
  shows *v* $@_v$ *v'* ≤ *w* $@_v$ *w'* ⟷ *v* ≤ *w* ∧ *v'* ≤ *w'*
⟨*proof*⟩

lemma *all-vec-append*: (∀ *x* ∈ *carrier-vec* (*n* + *m*). *P x*) ⟷ (∀ *x1* ∈ *carrier-vec n*. ∀ *x2* ∈ *carrier-vec m*. *P* (*x1* $@_v$ *x2*))
⟨*proof*⟩

definition *four-block-mat* :: *'a mat* ⇒ *'a mat* ⇒ *'a mat* ⇒ *'a mat* ⇒ *'a mat* **where**
  four-block-mat A B C D =
    (*let nra* = *dim-row A*; *nrd* = *dim-row D*;
        nca = *dim-col A*; *ncd* = *dim-col D*
      in
    mat (*nra* + *nrd*) (*nca* + *ncd*) (λ (*i*,*j*). *if i* < *nra then*
      if j < *nca then A* \$\$ (*i*,*j*) *else B* \$\$ (*i*,*j* − *nca*)
      else if j < *nca then C* \$\$ (*i* − *nra*, *j*) *else D* \$\$ (*i* − *nra*, *j* − *nca*)))

lemma *index-mat-four-block*[*simp*]:
  i < *dim-row A* + *dim-row D* ⟹ *j* < *dim-col A* + *dim-col D* ⟹ *four-block-mat A B C D* \$\$ (*i*,*j*)
  = (*if i* < *dim-row A then*
      if j < *dim-col A then A* \$\$ (*i*,*j*) *else B* \$\$ (*i*,*j* − *dim-col A*)
      else if j < *dim-col A then C* \$\$ (*i* − *dim-row A*, *j*) *else D* \$\$ (*i* − *dim-row A*, *j* − *dim-col A*))
  dim-row (*four-block-mat A B C D*) = *dim-row A* + *dim-row D*
  dim-col (*four-block-mat A B C D*) = *dim-col A* + *dim-col D*
  ⟨*proof*⟩

lemma *four-block-carrier-mat*[*simp*]:
  A ∈ *carrier-mat nr1 nc1* ⟹ *D* ∈ *carrier-mat nr2 nc2* ⟹
  four-block-mat A B C D ∈ *carrier-mat* (*nr1* + *nr2*) (*nc1* + *nc2*)
  ⟨*proof*⟩

lemma *cong-four-block-mat*: *A1* = *B1* ⟹ *A2* = *B2* ⟹ *A3* = *B3* ⟹ *A4* = *B4* ⟹
  four-block-mat A1 A2 A3 A4 = *four-block-mat B1 B2 B3 B4* ⟨*proof*⟩

lemma *four-block-one-mat*[*simp*]:
  four-block-mat (1_m *n1*) (0_m *n1 n2*) (0_m *n2 n1*) (1_m *n2*) = 1_m (*n1* + *n2*)
  ⟨*proof*⟩

lemma *four-block-zero-mat*[*simp*]:
four-block-mat (0_m *nr1 nc1*) (0_m *nr1 nc2*) (0_m *nr2 nc1*) (0_m *nr2 nc2*) = 0_m (*nr1* + *nr2*) (*nc1* + *nc2*)
⟨*proof*⟩

lemma *row-four-block-mat*:
assumes *c*: *A* ∈ *carrier-mat nr1 nc1 B* ∈ *carrier-mat nr1 nc2*
C ∈ *carrier-mat nr2 nc1 D* ∈ *carrier-mat nr2 nc2*
shows
i < *nr1* ⟹ *row* (*four-block-mat A B C D*) *i* = *row A i* @$_v$ *row B i* (**is** - ⟹ *?AB*)
¬ *i* < *nr1* ⟹ *i* < *nr1* + *nr2* ⟹ *row* (*four-block-mat A B C D*) *i* = *row C* (*i* − *nr1*) @$_v$ *row D* (*i* − *nr1*)
(**is** - ⟹ - ⟹ *?CD*)
⟨*proof*⟩

lemma *col-four-block-mat*:
assumes *c*: *A* ∈ *carrier-mat nr1 nc1 B* ∈ *carrier-mat nr1 nc2*
C ∈ *carrier-mat nr2 nc1 D* ∈ *carrier-mat nr2 nc2*
shows
j < *nc1* ⟹ *col* (*four-block-mat A B C D*) *j* = *col A j* @$_v$ *col C j* (**is** - ⟹ *?AC*)
¬ *j* < *nc1* ⟹ *j* < *nc1* + *nc2* ⟹ *col* (*four-block-mat A B C D*) *j* = *col B* (*j* − *nc1*) @$_v$ *col D* (*j* − *nc1*)
(**is** - ⟹ - ⟹ *?BD*)
⟨*proof*⟩

lemma *mult-four-block-mat*: **assumes**
c1: *A1* ∈ *carrier-mat nr1 n1 B1* ∈ *carrier-mat nr1 n2 C1* ∈ *carrier-mat nr2 n1 D1* ∈ *carrier-mat nr2 n2* **and**
c2: *A2* ∈ *carrier-mat n1 nc1 B2* ∈ *carrier-mat n1 nc2 C2* ∈ *carrier-mat n2 nc1 D2* ∈ *carrier-mat n2 nc2*
shows *four-block-mat A1 B1 C1 D1* ∗ *four-block-mat A2 B2 C2 D2*
= *four-block-mat* (*A1* ∗ *A2* + *B1* ∗ *C2*) (*A1* ∗ *B2* + *B1* ∗ *D2*)
(*C1* ∗ *A2* + *D1* ∗ *C2*) (*C1* ∗ *B2* + *D1* ∗ *D2*) (**is** *?M1* ∗ *?M2* = -)
⟨*proof*⟩

definition *append-rows* :: *'a* :: *zero mat* ⇒ *'a mat* ⇒ *'a mat* (**infixr** @$_r$ *65*)**where**
A @$_r$ *B* = *four-block-mat A* (0_m (*dim-row A*) *0*) *B* (0_m (*dim-row B*) *0*)

lemma *carrier-append-rows*[*simp*,*intro*]: *A* ∈ *carrier-mat nr1 nc* ⟹ *B* ∈ *carrier-mat nr2 nc* ⟹
A @$_r$ *B* ∈ *carrier-mat* (*nr1* + *nr2*) *nc*
⟨*proof*⟩

lemma *col-mult2*[*simp*]:
assumes *A*: *A* : *carrier-mat nr n*
and *B*: *B* : *carrier-mat n nc*
and *j*: *j* < *nc*

shows $\mathit{col}\ (A * B)\ j = A *_v \mathit{col}\ B\ j$
⟨*proof*⟩

lemma *mat-vec-as-mat-mat-mult*: **assumes** A: $A \in \mathit{carrier\text{-}mat}\ nr\ nc$
and v: $v \in \mathit{carrier\text{-}vec}\ nc$
shows $A *_v v = \mathit{col}\ (A * \mathit{mat\text{-}of\text{-}cols}\ nc\ [v])\ 0$
⟨*proof*⟩

lemma *mat-mult-append*: **assumes** A: $A \in \mathit{carrier\text{-}mat}\ nr1\ nc$
and B: $B \in \mathit{carrier\text{-}mat}\ nr2\ nc$
and v: $v \in \mathit{carrier\text{-}vec}\ nc$
shows $(A\ @_r\ B) *_v v = (A *_v v)\ @_v\ (B *_v v)$
⟨*proof*⟩

lemma *append-rows-le*: **assumes** A: $A \in \mathit{carrier\text{-}mat}\ nr1\ nc$
and B: $B \in \mathit{carrier\text{-}mat}\ nr2\ nc$
and a: $a \in \mathit{carrier\text{-}vec}\ nr1$
and v: $v \in \mathit{carrier\text{-}vec}\ nc$
shows $(A\ @_r\ B) *_v v \le (a\ @_v\ b) \longleftrightarrow A *_v v \le a \wedge B *_v v \le b$
⟨*proof*⟩

lemma *elements-four-block-mat*:
assumes c: $A \in \mathit{carrier\text{-}mat}\ nr1\ nc1\ B \in \mathit{carrier\text{-}mat}\ nr1\ nc2$
$C \in \mathit{carrier\text{-}mat}\ nr2\ nc1\ D \in \mathit{carrier\text{-}mat}\ nr2\ nc2$
shows
$\mathit{elements\text{-}mat}\ (\mathit{four\text{-}block\text{-}mat}\ A\ B\ C\ D) \subseteq$
$\mathit{elements\text{-}mat}\ A \cup \mathit{elements\text{-}mat}\ B \cup \mathit{elements\text{-}mat}\ C \cup \mathit{elements\text{-}mat}\ D$
(**is** $\mathit{elements\text{-}mat}\ ?four \subseteq$ -)
⟨*proof*⟩

lemma *assoc-four-block-mat*: **fixes** FB :: $'a\ mat \Rightarrow\ 'a\ mat \Rightarrow\ 'a :: zero\ mat$
defines FB: $FB \equiv \lambda\ Bb\ Cc.\ \mathit{four\text{-}block\text{-}mat}\ Bb\ (0_m\ (\mathit{dim\text{-}row}\ Bb)\ (\mathit{dim\text{-}col}\ Cc))$
$(0_m\ (\mathit{dim\text{-}row}\ Cc)\ (\mathit{dim\text{-}col}\ Bb))\ Cc$
shows $FB\ A\ (FB\ B\ C) = FB\ (FB\ A\ B)\ C$ (**is** $?L = ?R$)
⟨*proof*⟩

definition *split-block* :: $'a\ mat \Rightarrow nat \Rightarrow nat \Rightarrow ('a\ mat \times\ 'a\ mat \times\ 'a\ mat \times\ 'a\ mat)$
where $\mathit{split\text{-}block}\ A\ sr\ sc = (\mathit{let}$
$nr = \mathit{dim\text{-}row}\ A;\ nc = \mathit{dim\text{-}col}\ A;$
$nr2 = nr - sr;\ nc2 = nc - sc;$
$A1 = \mathit{mat}\ sr\ sc\ (\lambda\ ij.\ A\ \$\$\ ij);$
$A2 = \mathit{mat}\ sr\ nc2\ (\lambda\ (i,j).\ A\ \$\$\ (i,j+sc));$
$A3 = \mathit{mat}\ nr2\ sc\ (\lambda\ (i,j).\ A\ \$\$\ (i+sr,j));$
$A4 = \mathit{mat}\ nr2\ nc2\ (\lambda\ (i,j).\ A\ \$\$\ (i+sr,j+sc))$
$\mathit{in}\ (A1,A2,A3,A4))$

lemma *split-block*: **assumes** *res*: $\mathit{split\text{-}block}\ A\ sr1\ sc1 = (A1,A2,A3,A4)$

and *dims*: *dim-row A = sr1 + sr2 dim-col A = sc1 + sc2*
shows *A1 ∈ carrier-mat sr1 sc1 A2 ∈ carrier-mat sr1 sc2*
A3 ∈ carrier-mat sr2 sc1 A4 ∈ carrier-mat sr2 sc2
A = four-block-mat A1 A2 A3 A4
⟨*proof*⟩

Using *four-block-mat* we define block-diagonal matrices.

fun *diag-block-mat* :: *'a* :: *zero mat list ⇒ 'a mat* **where**
diag-block-mat [] = 0_m *0 0*
| *diag-block-mat* (*A # As*) = (*let*
B = diag-block-mat As
in four-block-mat A (0_m (*dim-row A*) (*dim-col B*)) (0_m (*dim-row B*) (*dim-col A*)) *B*)

lemma *dim-diag-block-mat*:
dim-row (*diag-block-mat As*) = *sum-list* (*map dim-row As*) (**is** *?row*)
dim-col (*diag-block-mat As*) = *sum-list* (*map dim-col As*) (**is** *?col*)
⟨*proof*⟩

lemma *diag-block-mat-singleton*[*simp*]: *diag-block-mat* [*A*] = *A*
⟨*proof*⟩

lemma *diag-block-mat-append*: *diag-block-mat* (*As* @ *Bs*) =
(*let A = diag-block-mat As*; *B = diag-block-mat Bs*
in four-block-mat A (0_m (*dim-row A*) (*dim-col B*)) (0_m (*dim-row B*) (*dim-col A*)) *B*)
⟨*proof*⟩

lemma *diag-block-mat-last*: *diag-block-mat* (*As* @ [*B*]) =
(*let A = diag-block-mat As*
in four-block-mat A (0_m (*dim-row A*) (*dim-col B*)) (0_m (*dim-row B*) (*dim-col A*)) *B*)
⟨*proof*⟩

lemma *diag-block-mat-square*:
Ball (*set As*) *square-mat* ⟹ *square-mat* (*diag-block-mat As*)
⟨*proof*⟩

lemma *diag-block-one-mat*[*simp*]:
diag-block-mat (*map* (λ*A*. 1_m (*dim-row A*)) *As*) = (1_m (*sum-list* (*map dim-row As*)))
⟨*proof*⟩

lemma *elements-diag-block-mat*:
elements-mat (*diag-block-mat As*) ⊆ {*0*} ∪ ⋃ (*set* (*map elements-mat As*))
⟨*proof*⟩

lemma *diag-block-pow-mat*: **assumes** *sq*: *Ball* (*set As*) *square-mat*

shows *diag-block-mat As* $\widehat{\ }_m$ *n* = *diag-block-mat* (*map* (λ *A*. *A* $\widehat{\ }_m$ *n*) *As*) (**is** *?As* $\widehat{\ }_m$ - = -)
⟨*proof*⟩

lemma *diag-block-upper-triangular*: **assumes**
⋀ *A i j*. *A* ∈ *set As* ⟹ *j* < *i* ⟹ *i* < *dim-row A* ⟹ *A* $$ (*i*,*j*) = *0*
and *Ball* (*set As*) *square-mat*
and *j* < *i i* < *dim-row* (*diag-block-mat As*)
shows *diag-block-mat As* $$ (*i*,*j*) = *0*
⟨*proof*⟩

lemma *smult-four-block-mat*: **assumes** *c*: *A* ∈ *carrier-mat nr1 nc1 B* ∈ *carrier-mat nr1 nc2*
C ∈ *carrier-mat nr2 nc1 D* ∈ *carrier-mat nr2 nc2*
shows *a* $\cdot_m$ *four-block-mat A B C D* = *four-block-mat* (*a* $\cdot_m$ *A*) (*a* $\cdot_m$ *B*) (*a* $\cdot_m$ *C*) (*a* $\cdot_m$ *D*)
⟨*proof*⟩

lemma *map-four-block-mat*: **assumes** *c*: *A* ∈ *carrier-mat nr1 nc1 B* ∈ *carrier-mat nr1 nc2*
C ∈ *carrier-mat nr2 nc1 D* ∈ *carrier-mat nr2 nc2*
shows *map-mat f* (*four-block-mat A B C D*) = *four-block-mat* (*map-mat f A*) (*map-mat f B*) (*map-mat f C*) (*map-mat f D*)
⟨*proof*⟩

lemma *add-four-block-mat*: **assumes**
c1: *A1* ∈ *carrier-mat nr1 nc1 B1* ∈ *carrier-mat nr1 nc2 C1* ∈ *carrier-mat nr2 nc1 D1* ∈ *carrier-mat nr2 nc2* **and**
c2: *A2* ∈ *carrier-mat nr1 nc1 B2* ∈ *carrier-mat nr1 nc2 C2* ∈ *carrier-mat nr2 nc1 D2* ∈ *carrier-mat nr2 nc2*
shows *four-block-mat A1 B1 C1 D1* + *four-block-mat A2 B2 C2 D2*
= *four-block-mat* (*A1* + *A2*) (*B1* + *B2*) (*C1* + *C2*) (*D1* + *D2*)
⟨*proof*⟩

lemma *diag-four-block-mat*: **assumes** *c*: *A* ∈ *carrier-mat n1 n1*
D ∈ *carrier-mat n2 n2*
shows *diag-mat* (*four-block-mat A B C D*) = *diag-mat A* @ *diag-mat D*
⟨*proof*⟩

definition *mk-diagonal* :: *'a::zero list* ⇒ *'a mat*
where *mk-diagonal as* = *diag-block-mat* (*map* (λ*a*. *mat* (*Suc 0*) (*Suc 0*) (λ-. *a*)) *as*)

lemma *mk-diagonal-dim*:
dim-row (*mk-diagonal as*) = *length as dim-col* (*mk-diagonal as*) = *length as*
⟨*proof*⟩

lemma *mk-diagonal-diagonal*: *diagonal-mat* (*mk-diagonal as*)

⟨*proof*⟩

definition *orthogonal-mat* :: $'a::semiring\text{-}0\ mat \Rightarrow bool$
where *orthogonal-mat A* ≡
let B = transpose-mat A ∗ A in
diagonal-mat B ∧ (∀ *i*<*dim-col A*. *B* $$ (*i*,*i*) ≠ *0*)

lemma *orthogonal-matD*[*elim*]:
orthogonal-mat A ⟹
i < *dim-col A* ⟹ *j* < *dim-col A* ⟹ (*col A i* · *col A j* = *0*) = (*i* ≠ *j*)
⟨*proof*⟩

lemma *orthogonal-matI*[*intro*]:
(⋀*i j*. *i* < *dim-col A* ⟹ *j* < *dim-col A* ⟹ (*col A i* · *col A j* = *0*) = (*i* ≠ *j*)) ⟹
orthogonal-mat A
⟨*proof*⟩

definition *orthogonal* :: $'a::semiring\text{-}0\ vec\ list \Rightarrow bool$
where *orthogonal vs* ≡
∀ *i j*. *i* < *length vs* ⟶ *j* < *length vs* ⟶
(*vs* ! *i* · *vs* ! *j* = *0*) = (*i* ≠ *j*)

lemma *orthogonalD*[*elim*]:
orthogonal vs ⟹ *i* < *length vs* ⟹ *j* < *length vs* ⟹
(*nth vs i* · *nth vs j* = *0*) = (*i* ≠ *j*)
⟨*proof*⟩

lemma *orthogonalI*[*intro*]:
(⋀*i j*. *i* < *length vs* ⟹ *j* < *length vs* ⟹ (*nth vs i* · *nth vs j* = *0*) = (*i* ≠ *j*)) ⟹
orthogonal vs
⟨*proof*⟩

lemma *transpose-four-block-mat*: **assumes** ∗: *A* ∈ *carrier-mat nr1 nc1 B* ∈ *carrier-mat nr1 nc2*
C ∈ *carrier-mat nr2 nc1 D* ∈ *carrier-mat nr2 nc2*
shows *transpose-mat* (*four-block-mat A B C D*) =
four-block-mat (*transpose-mat A*) (*transpose-mat C*) (*transpose-mat B*) (*transpose-mat D*)
⟨*proof*⟩

lemma *zero-transpose-mat*[*simp*]: *transpose-mat* (0_m *n m*) = (0_m *m n*)
⟨*proof*⟩

lemma *upper-triangular-four-block*: **assumes** *AD*: *A* ∈ *carrier-mat n n D* ∈ *carrier-mat m m*
and *ut*: *upper-triangular A upper-triangular D*

  shows *upper-triangular* (*four-block-mat A B* (0_m *m n*) *D*)
⟨*proof*⟩

lemma *pow-four-block-mat*: **assumes** *A*: $A \in$ *carrier-mat n n*
  and *B*: $B \in$ *carrier-mat m m*
  shows (*four-block-mat A* (0_m *n m*) (0_m *m n*) *B*) $\widehat{\ }_m$ *k* =
    four-block-mat (*A* $\widehat{\ }_m$ *k*) (0_m *n m*) (0_m *m n*) (*B* $\widehat{\ }_m$ *k*)
⟨*proof*⟩

lemma *uminus-scalar-prod*:
  assumes [*simp*]: *v* : *carrier-vec n w* : *carrier-vec n*
  shows − ((*v*::′*a*::*field vec*) · *w*) = (− *v*) · *w*
  ⟨*proof*⟩

lemma *append-vec-eq*:
  assumes [*simp*]: *v* : *carrier-vec n* v' : *carrier-vec n*
  shows [*simp*]: $v\ @_v\ w = v'\ @_v\ w' \longleftrightarrow v = v' \wedge w = w'$ (**is** *?L* ⟷ *?R*)
⟨*proof*⟩

lemma *append-vec-add*:
  assumes [*simp*]: *v* : *carrier-vec n* v' : *carrier-vec n*
    and [*simp*]: *w* : *carrier-vec m* w' : *carrier-vec m*
  shows $(v\ @_v\ w) + (v'\ @_v\ w') = (v + v')\ @_v\ (w + w')$ (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *four-block-mat-mult-vec*:
  assumes *A*: *A* : *carrier-mat nr1 nc1*
    and *B*: *B* : *carrier-mat nr1 nc2*
    and *C*: *C* : *carrier-mat nr2 nc1*
    and *D*: *D* : *carrier-mat nr2 nc2*
    and *a*: *a* : *carrier-vec nc1*
    and *d*: *d* : *carrier-vec nc2*
  shows *four-block-mat A B C D* $*_v$ ($a\ @_v\ d$) = ($A *_v a + B *_v d$) $@_v$ ($C *_v a$ + $D *_v d$)
    (**is** *?ABCD* $*_v$ - = *?r*)
⟨*proof*⟩

lemma *mult-mat-vec-split*:
  assumes *A*: *A* : *carrier-mat n n*
    and *D*: *D* : *carrier-mat m m*
    and *a*: *a* : *carrier-vec n*
    and *d*: *d* : *carrier-vec m*
  shows *four-block-mat A* (0_m *n m*) (0_m *m n*) *D* $*_v$ ($a\ @_v\ d$) = $A *_v a\ @_v\ D *_v d$
  ⟨*proof*⟩

lemma *similar-mat-witI*: **assumes** $P * Q = 1_m\ n$ $Q * P = 1_m\ n$ $A = P * B *$

Q
A ∈ *carrier-mat n n B* ∈ *carrier-mat n n P* ∈ *carrier-mat n n Q* ∈ *carrier-mat n n*
shows *similar-mat-wit A B P Q* ⟨*proof*⟩

lemma *similar-mat-witD*: **assumes** *n* = *dim-row A similar-mat-wit A B P Q*
shows *P* ∗ *Q* = 1_m *n Q* ∗ *P* = 1_m *n A* = *P* ∗ *B* ∗ *Q*
A ∈ *carrier-mat n n B* ∈ *carrier-mat n n P* ∈ *carrier-mat n n Q* ∈ *carrier-mat n n*
⟨*proof*⟩

lemma *similar-mat-witD2*: **assumes** *A* ∈ *carrier-mat n m similar-mat-wit A B P Q*
shows *P* ∗ *Q* = 1_m *n Q* ∗ *P* = 1_m *n A* = *P* ∗ *B* ∗ *Q*
A ∈ *carrier-mat n n B* ∈ *carrier-mat n n P* ∈ *carrier-mat n n Q* ∈ *carrier-mat n n*
⟨*proof*⟩

lemma *similar-mat-wit-sym*: **assumes** *sim*: *similar-mat-wit A B P Q*
shows *similar-mat-wit B A Q P*
⟨*proof*⟩

lemma *similar-mat-wit-refl*: **assumes** *A*: *A* ∈ *carrier-mat n n*
shows *similar-mat-wit A A* (1_m *n*) (1_m *n*)
⟨*proof*⟩

lemma *similar-mat-wit-trans*: **assumes** *AB*: *similar-mat-wit A B P Q*
and *BC*: *similar-mat-wit B C P′ Q′*
shows *similar-mat-wit A C* (*P* ∗ *P′*) (*Q′* ∗ *Q*)
⟨*proof*⟩

lemma *similar-mat-refl*: *A* ∈ *carrier-mat n n* ⟹ *similar-mat A A*
⟨*proof*⟩

lemma *similar-mat-trans*: *similar-mat A B* ⟹ *similar-mat B C* ⟹ *similar-mat A C*
⟨*proof*⟩

lemma *similar-mat-sym*: *similar-mat A B* ⟹ *similar-mat B A*
⟨*proof*⟩

lemma *similar-mat-wit-four-block*: **assumes**
1: *similar-mat-wit A1 B1 P1 Q1*
and *2*: *similar-mat-wit A2 B2 P2 Q2*
and *URA*: *URA* = (*P1* ∗ *UR* ∗ *Q2*)
and *LLA*: *LLA* = (*P2* ∗ *LL* ∗ *Q1*)
and *A1*: *A1* ∈ *carrier-mat n n*
and *A2*: *A2* ∈ *carrier-mat m m*
and *LL*: *LL* ∈ *carrier-mat m n*

and *UR*: *UR* ∈ *carrier-mat n m*
shows *similar-mat-wit* (*four-block-mat A1 URA LLA A2*) (*four-block-mat B1 UR LL B2*)
(*four-block-mat P1* (0_m *n m*) (0_m *m n*) *P2*) (*four-block-mat Q1* (0_m *n m*) (0_m *m n*) *Q2*)
(**is** *similar-mat-wit ?A ?B ?P ?Q*)
⟨*proof*⟩

lemma *similar-mat-four-block-0-ex*: **assumes**
1: *similar-mat A1 B1*
and *2*: *similar-mat A2 B2*
and *A0*: *A0* ∈ *carrier-mat n m*
and *A1*: *A1* ∈ *carrier-mat n n*
and *A2*: *A2* ∈ *carrier-mat m m*
shows ∃ *B0*. *B0* ∈ *carrier-mat n m* ∧ *similar-mat* (*four-block-mat A1 A0* (0_m *m n*) *A2*)
(*four-block-mat B1 B0* (0_m *m n*) *B2*)
⟨*proof*⟩

lemma *similar-mat-four-block-0-0*: **assumes**
1: *similar-mat A1 B1*
and *2*: *similar-mat A2 B2*
and *A1*: *A1* ∈ *carrier-mat n n*
and *A2*: *A2* ∈ *carrier-mat m m*
shows *similar-mat* (*four-block-mat A1* (0_m *n m*) (0_m *m n*) *A2*)
(*four-block-mat B1* (0_m *n m*) (0_m *m n*) *B2*)
⟨*proof*⟩

lemma *similar-diag-mat-block-mat*: **assumes** ⋀ *A B*. (*A*,*B*) ∈ *set Ms* ⟹ *similar-mat A B*
shows *similar-mat* (*diag-block-mat* (*map fst Ms*)) (*diag-block-mat* (*map snd Ms*))
⟨*proof*⟩

lemma *similar-mat-wit-pow*: **assumes** *wit*: *similar-mat-wit A B P Q*
shows *similar-mat-wit* (*A* $\hat{}_m$ *k*) (*B* $\hat{}_m$ *k*) *P Q*
⟨*proof*⟩

lemma *similar-mat-wit-pow-id*: *similar-mat-wit A B P Q* ⟹ *A* $\hat{}_m$ *k* = *P* ∗ *B* $\hat{}_m$ *k* ∗ *Q*
⟨*proof*⟩

4.5 Homomorphism properties

context *semiring-hom*
begin
abbreviation *mat-hom* :: *'a mat* ⇒ *'b mat* (mat_h)
where mat_h ≡ *map-mat hom*

abbreviation *vec-hom* :: *'a vec* ⇒ *'b vec* (vec_h)
where vec_h ≡ *map-vec hom*

lemma *vec-hom-zero*: vec_h (0_v *n*) = 0_v *n*
⟨*proof*⟩

lemma *mat-hom-one*: mat_h (1_m *n*) = 1_m *n*
⟨*proof*⟩

lemma *mat-hom-mult*: **assumes** *A*: *A* ∈ *carrier-mat nr n* **and** *B*: *B* ∈ *carrier-mat n nc*
shows mat_h (*A* ∗ *B*) = mat_h *A* ∗ mat_h *B*
⟨*proof*⟩

lemma *mult-mat-vec-hom*: **assumes** *A*: *A* ∈ *carrier-mat nr n* **and** *v*: *v* ∈ *carrier-vec n*
shows vec_h (*A* $*_v$ *v*) = mat_h *A* $*_v$ vec_h *v*
⟨*proof*⟩
end

lemma *vec-eq-iff*: (*x* = *y*) = (*dim-vec x* = *dim-vec y* ∧ (∀ *i* < *dim-vec y*. *x* \$ *i* = *y* \$ *i*)) (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *mat-eq-iff*: (*x* = *y*) = (*dim-row x* = *dim-row y* ∧ *dim-col x* = *dim-col y* ∧
(∀ *i j*. *i* < *dim-row y* ⟶ *j* < *dim-col y* ⟶ *x* \$\$ (*i*,*j*) = *y* \$\$ (*i*,*j*))) (**is** *?l* = *?r*)
⟨*proof*⟩

lemma (**in** *inj-semiring-hom*) *vec-hom-zero-iff*[*simp*]: (vec_h *x* = 0_v *n*) = (*x* = 0_v *n*)
⟨*proof*⟩

lemma (**in** *inj-semiring-hom*) *mat-hom-inj*: mat_h *A* = mat_h *B* ⟹ *A* = *B*
⟨*proof*⟩

lemma (**in** *inj-semiring-hom*) *vec-hom-inj*: vec_h *v* = vec_h *w* ⟹ *v* = *w*
⟨*proof*⟩

lemma (**in** *semiring-hom*) *mat-hom-pow*: **assumes** *A*: *A* ∈ *carrier-mat n n*
shows mat_h (*A* $\widehat{}_m$ *k*) = (mat_h *A*) $\widehat{}_m$ *k*
⟨*proof*⟩

lemma (**in** *semiring-hom*) *hom-sum-mat*: *hom* (*sum-mat A*) = *sum-mat* (mat_h *A*)
⟨*proof*⟩

lemma (**in** *semiring-hom*) *vec-hom-smult*: vec_h (*ev* $\cdot_v$ *v*) = *hom ev* $\cdot_v$ vec_h *v*
⟨*proof*⟩

lemma *minus-scalar-prod-distrib*: **fixes** v_1 :: *'a* :: *ring vec*

assumes v: $v_1 \in \textit{carrier-vec}\ n\ v_2 \in \textit{carrier-vec}\ n\ v_3 \in \textit{carrier-vec}\ n$
shows $(v_1 - v_2) \cdot v_3 = v_1 \cdot v_3 - v_2 \cdot v_3$
⟨*proof*⟩

lemma *scalar-prod-minus-distrib*: **fixes** $v_1 :: {'a} :: \textit{ring vec}$
assumes v: $v_1 \in \textit{carrier-vec}\ n\ v_2 \in \textit{carrier-vec}\ n\ v_3 \in \textit{carrier-vec}\ n$
shows $v_1 \cdot (v_2 - v_3) = v_1 \cdot v_2 - v_1 \cdot v_3$
⟨*proof*⟩

lemma *uminus-add-minus-vec*:
assumes $l \in \textit{carrier-vec}\ n\ r \in \textit{carrier-vec}\ n$
shows $- ((l::{'a} :: \textit{ab-group-add vec}) + r) = (- l - r)$
⟨*proof*⟩

lemma *minus-add-minus-vec*: **fixes** $u :: {'a} :: \textit{ab-group-add vec}$
assumes $u \in \textit{carrier-vec}\ n\ v \in \textit{carrier-vec}\ n\ w \in \textit{carrier-vec}\ n$
shows $u - (v + w) = u - v - w$
⟨*proof*⟩

lemma *uminus-add-minus-mat*:
assumes $l \in \textit{carrier-mat}\ nr\ nc\ r \in \textit{carrier-mat}\ nr\ nc$
shows $- ((l::{'a} :: \textit{ab-group-add mat}) + r) = (- l - r)$
⟨*proof*⟩

lemma *minus-add-minus-mat*: **fixes** $u :: {'a} :: \textit{ab-group-add mat}$
assumes $u \in \textit{carrier-mat}\ nr\ nc\ v \in \textit{carrier-mat}\ nr\ nc\ w \in \textit{carrier-mat}\ nr\ nc$
shows $u - (v + w) = u - v - w$
⟨*proof*⟩

lemma *uminus-uminus-vec*[*simp*]: $- (- (v::{'a}:: \textit{group-add vec})) = v$
⟨*proof*⟩

lemma *uminus-eq-vec*[*simp*]: $- (v::{'a}:: \textit{group-add vec}) = - w \longleftrightarrow v = w$
⟨*proof*⟩

lemma *uminus-uminus-mat*[*simp*]: $- (- (A::{'a}:: \textit{group-add mat})) = A$
⟨*proof*⟩

lemma *uminus-eq-mat*[*simp*]: $- (A::{'a}:: \textit{group-add mat}) = - B \longleftrightarrow A = B$
⟨*proof*⟩

lemma *smult-zero-mat*[*simp*]: $(k :: {'a} :: \textit{mult-zero}) \cdot_m 0_m\ nr\ nc = 0_m\ nr\ nc$
⟨*proof*⟩

lemma *similar-mat-wit-smult*: **fixes** $A :: {'a} :: \textit{comm-ring-1 mat}$
assumes *similar-mat-wit* $A\ B\ P\ Q$
shows *similar-mat-wit* $(k \cdot_m A)\ (k \cdot_m B)\ P\ Q$
⟨*proof*⟩

lemma *similar-mat-smult*: **fixes** *A* :: *'a* :: *comm-ring-1 mat*
  assumes *similar-mat A B*
  shows *similar-mat* $(k \cdot_m A)$ $(k \cdot_m B)$
  ⟨*proof*⟩

definition *mat-diag* :: *nat* ⇒ (*nat* ⇒ *'a* :: *zero*) ⇒ *'a mat* **where**
  mat-diag n f = *Matrix.mat n n* (λ (*i*,*j*). *if i* = *j then f j else 0*)

lemma *mat-diag-dim*[*simp*]: *mat-diag n f* ∈ *carrier-mat n n*
  ⟨*proof*⟩

lemma *mat-diag-mult-left*: **assumes** *A*: *A* ∈ *carrier-mat n nr*
  shows *mat-diag n f* ∗ *A* = *Matrix.mat n nr* (λ (*i*,*j*). *f i* ∗ *A* $$ (*i*,*j*))
⟨*proof*⟩

lemma *mat-diag-mult-right*: **assumes** *A*: *A* ∈ *carrier-mat nr n*
  shows *A* ∗ *mat-diag n f* = *Matrix.mat nr n* (λ (*i*,*j*). *A* $$ (*i*,*j*) ∗ *f j*)
⟨*proof*⟩

lemma *mat-diag-diag*[*simp*]: *mat-diag n f* ∗ *mat-diag n g* = *mat-diag n* (λ *i*. *f i* ∗ *g i*)
  ⟨*proof*⟩

lemma *mat-diag-one*[*simp*]: *mat-diag n* (λ *x*. *1*) = 1_m *n* ⟨*proof*⟩

Interpret vector as row-matrix

definition *mat-of-row y* = *mat 1* (*dim-vec y*) (λ *ij*. *y* $ (*snd ij*))

lemma *mat-of-row-carrier*[*simp*,*intro*]:
  y ∈ *carrier-vec n* ⟹ *mat-of-row y* ∈ *carrier-mat 1 n*
  y ∈ *carrier-vec n* ⟹ *mat-of-row y* ∈ *carrier-mat* (*Suc 0*) *n*
  ⟨*proof*⟩

lemma *mat-of-row-dim*[*simp*]: *dim-row* (*mat-of-row y*) = *1*
  dim-col (*mat-of-row y*) = *dim-vec y*
  ⟨*proof*⟩

lemma *mat-of-row-index*[*simp*]: *x* < *dim-vec y* ⟹ *mat-of-row y* $$ (*0*,*x*) = *y* $ *x*

  ⟨*proof*⟩

lemma *row-mat-of-row*[*simp*]: *row* (*mat-of-row y*) *0* = *y*
  ⟨*proof*⟩

lemma *mat-of-row-mult-append-rows*: **assumes** *y1*: *y1* ∈ *carrier-vec nr1*
  and *y2*: *y2* ∈ *carrier-vec nr2*
  and *A1*: *A1* ∈ *carrier-mat nr1 nc*
  and *A2*: *A2* ∈ *carrier-mat nr2 nc*
shows *mat-of-row* $(y1 \mathbin{@_v} y2)$ ∗ $(A1 \mathbin{@_r} A2)$ =

mat-of-row y1 ∗ A1 + mat-of-row y2 ∗ A2
⟨*proof*⟩

lemma *mat-of-row-uminus*: *mat-of-row* (− *v*) = − *mat-of-row v*
⟨*proof*⟩

Allowing to construct and deconstruct vectors like lists

abbreviation *vNil* **where** *vNil* ≡ *vec 0* ((!) [])
definition *vCons* **where** *vCons a v* ≡ *vec* (*Suc* (*dim-vec v*)) (*λi. case i of 0* ⇒ *a* | *Suc i* ⇒ *v* $ *i*)

lemma *vec-index-vCons-0* [*simp*]: *vCons a v* $ *0* = *a*
⟨*proof*⟩

lemma *vec-index-vCons-Suc* [*simp*]:
fixes *v* :: *'a vec*
shows *vCons a v* $ *Suc n* = *v* $ *n*
⟨*proof*⟩

lemma *vec-index-vCons*: *vCons a v* $ *n* = (*if n* = *0 then a else v* $ (*n* − *1*))
⟨*proof*⟩

lemma *dim-vec-vCons* [*simp*]: *dim-vec* (*vCons a v*) = *Suc* (*dim-vec v*)
⟨*proof*⟩

lemma *vCons-carrier-vec*[*simp*]: *vCons a v* ∈ *carrier-vec* (*Suc n*) ⟷ *v* ∈ *carrier-vec n*
⟨*proof*⟩

lemma *vec-Suc*: *vec* (*Suc n*) *f* = *vCons* (*f 0*) (*vec n* (*f* ∘ *Suc*)) (**is** *?l* = *?r*)
⟨*proof*⟩

declare *Abs-vec-cases*[*cases del*]

lemma *vec-cases* [*case-names vNil vCons*, *cases type*: *vec*]:
assumes *v* = *vNil* ⟹ *thesis* **and** ⋀*a w. v* = *vCons a w* ⟹ *thesis*
shows *thesis*
⟨*proof*⟩

lemma *vec-induct* [*case-names vNil vCons*, *induct type*: *vec*]:
assumes *P vNil* **and** ⋀*a v. P v* ⟹ *P* (*vCons a v*)
shows *P v*
⟨*proof*⟩

lemma *carrier-vec-induct* [*consumes 1*, *case-names 0 Suc*, *induct set*:*carrier-vec*]:
assumes *v*: *v* ∈ *carrier-vec n*
and *1*: *P 0 vNil* **and** *2*: ⋀*n a v. v* ∈ *carrier-vec n* ⟹ *P n v* ⟹ *P* (*Suc n*) (*vCons a v*)
shows *P n v*

⟨*proof*⟩

lemma *vec-of-list-Cons*[*simp*]: *vec-of-list* (*a#as*) = *vCons a* (*vec-of-list as*)
⟨*proof*⟩

lemma *vec-of-list-Nil*[*simp*]: *vec-of-list* [] = *vNil*
⟨*proof*⟩

lemma *scalar-prod-vCons*[*simp*]:
vCons a v · *vCons b w* = *a* ∗ *b* + *v* · *w*
⟨*proof*⟩

lemma *zero-vec-Suc*: 0_v (*Suc n*) = *vCons 0* (0_v *n*)
⟨*proof*⟩

lemma *zero-vec-zero*[*simp*]: 0_v *0* = *vNil* ⟨*proof*⟩

lemma *vCons-eq-vCons*[*simp*]: *vCons a v* = *vCons b w* ⟷ *a* = *b* ∧ *v* = *w* (**is** *?l* ⟷ *?r*)
⟨*proof*⟩

lemma *vec-carrier-vec*[*simp*]: *vec n f* ∈ *carrier-vec m* ⟷ *n* = *m*
⟨*proof*⟩

notation *transpose-mat* ((-^{T}) [*1000*])

lemma *map-mat-transpose*: (*map-mat f A*)^T = *map-mat f A*^{T} ⟨*proof*⟩

lemma *cols-transpose*[*simp*]: *cols A*^{T} = *rows A* ⟨*proof*⟩
lemma *rows-transpose*[*simp*]: *rows A*^{T} = *cols A* ⟨*proof*⟩
lemma *list-of-vec-vec* [*simp*]: *list-of-vec* (*vec n f*) = *map f* [*0..<n*]
⟨*proof*⟩

lemma *list-of-vec-0* [*simp*]: *list-of-vec* (0_v *n*) = *replicate n 0*
⟨*proof*⟩

lemma *diag-mat-map*:
assumes *M-carrier*: *M* ∈ *carrier-mat n n*
shows *diag-mat* (*map-mat f M*) = *map f* (*diag-mat M*)
⟨*proof*⟩

lemma *mat-of-rows-map* [*simp*]:
assumes *x*: *set vs* ⊆ *carrier-vec n*
shows *mat-of-rows n* (*map* (*map-vec f*) *vs*) = *map-mat f* (*mat-of-rows n vs*)
⟨*proof*⟩

lemma *mat-of-cols-map* [*simp*]:
assumes *x*: *set vs* ⊆ *carrier-vec n*
shows *mat-of-cols n* (*map* (*map-vec f*) *vs*) = *map-mat f* (*mat-of-cols n vs*)

⟨*proof*⟩

lemma *vec-of-list-map* [*simp*]: $\mathit{vec\text{-}of\text{-}list}\ (\mathit{map}\ f\ \mathit{xs}) = \mathit{map\text{-}vec}\ f\ (\mathit{vec\text{-}of\text{-}list}\ \mathit{xs})$
⟨*proof*⟩

lemma *map-vec*: $\mathit{map\text{-}vec}\ f\ (\mathit{vec}\ n\ g) = \mathit{vec}\ n\ (f\ o\ g)$ ⟨*proof*⟩

lemma *mat-of-cols-Cons-index-0*: $i < n \Longrightarrow \mathit{mat\text{-}of\text{-}cols}\ n\ (w\ \#\ \mathit{ws})\ \$\$\ (i,\ 0) = w\ \$\ i$
⟨*proof*⟩

lemma *nth-map-out-of-bound*: $i \geq \mathit{length}\ \mathit{xs} \Longrightarrow \mathit{map}\ f\ \mathit{xs}\ !\ i = []\ !\ (i - \mathit{length}\ \mathit{xs})$
⟨*proof*⟩

lemma *mat-of-cols-Cons-index-Suc*:
$i < n \Longrightarrow \mathit{mat\text{-}of\text{-}cols}\ n\ (w\ \#\ \mathit{ws})\ \$\$\ (i,\ \mathit{Suc}\ j) = \mathit{mat\text{-}of\text{-}cols}\ n\ \mathit{ws}\ \$\$\ (i,j)$
⟨*proof*⟩

lemma *mat-of-cols-index*: $i < n \Longrightarrow j < \mathit{length}\ \mathit{ws} \Longrightarrow \mathit{mat\text{-}of\text{-}cols}\ n\ \mathit{ws}\ \$\$\ (i,j) = \mathit{ws}\ !\ j\ \$\ i$
⟨*proof*⟩

lemma *mat-of-rows-index*: $i < \mathit{length}\ \mathit{rs} \Longrightarrow j < n \Longrightarrow \mathit{mat\text{-}of\text{-}rows}\ n\ \mathit{rs}\ \$\$\ (i,j) = \mathit{rs}\ !\ i\ \$\ j$
⟨*proof*⟩

lemma *transpose-mat-of-rows*: $(\mathit{mat\text{-}of\text{-}rows}\ n\ \mathit{vs})^T = \mathit{mat\text{-}of\text{-}cols}\ n\ \mathit{vs}$
⟨*proof*⟩

lemma *transpose-mat-of-cols*: $(\mathit{mat\text{-}of\text{-}cols}\ n\ \mathit{vs})^T = \mathit{mat\text{-}of\text{-}rows}\ n\ \mathit{vs}$
⟨*proof*⟩

lemma *nth-list-of-vec* [*simp*]:
assumes $i < \mathit{dim\text{-}vec}\ v$ **shows** $\mathit{list\text{-}of\text{-}vec}\ v\ !\ i = v\ \$\ i$
⟨*proof*⟩

lemma *length-list-of-vec* [*simp*]:
$\mathit{length}\ (\mathit{list\text{-}of\text{-}vec}\ v) = \mathit{dim\text{-}vec}\ v$ ⟨*proof*⟩

lemma *vec-eq-0-iff*:
$v = 0_v\ n \longleftrightarrow n = \mathit{dim\text{-}vec}\ v \wedge (n = 0 \vee \mathit{set}\ (\mathit{list\text{-}of\text{-}vec}\ v) = \{0\})$ (**is** $?l \longleftrightarrow ?r$)
⟨*proof*⟩

lemma *list-of-vec-vCons*[*simp*]: $\mathit{list\text{-}of\text{-}vec}\ (\mathit{vCons}\ a\ v) = a\ \#\ \mathit{list\text{-}of\text{-}vec}\ v$ (**is** $?l = ?r$)
⟨*proof*⟩

lemma *append-vec-vCons*[*simp*]: $\mathit{vCons}\ a\ v\ @_v\ w = \mathit{vCons}\ a\ (v\ @_v\ w)$ (**is** $?l =$

?r)
⟨*proof*⟩

lemma *append-vec-vNil*[*simp*]: $vNil\ @_v\ v = v$
⟨*proof*⟩

lemma *list-of-vec-append*[*simp*]: *list-of-vec* ($v\ @_v\ w$) = *list-of-vec v* @ *list-of-vec w*
⟨*proof*⟩

lemma *transpose-mat-eq*[*simp*]: $A^T = B^T \longleftrightarrow A = B$
⟨*proof*⟩

lemma *mat-col-eqI*: **assumes** *cols*: $\bigwedge i.\ i <$ *dim-col B* $\Longrightarrow$ *col A i* = *col B i*
and *dims*: *dim-row A* = *dim-row B dim-col A* = *dim-col B*
shows $A = B$
⟨*proof*⟩

lemma *upper-triangular-imp-distinct*:
assumes *A*: $A \in$ *carrier-mat n n*
and *tri*: *upper-triangular A*
and *diag*: $0 \notin$ *set* (*diag-mat A*)
shows *distinct* (*rows A*)
⟨*proof*⟩

lemma *dim-vec-of-list*[*simp*] :*dim-vec* (*vec-of-list as*) = *length as* ⟨*proof*⟩

lemma *list-vec*: *list-of-vec* (*vec-of-list xs*) = *xs*
⟨*proof*⟩

lemma *vec-list*: *vec-of-list* (*list-of-vec v*) = *v*
⟨*proof*⟩

lemma *index-vec-of-list*: *i*<*length xs* $\Longrightarrow$ (*vec-of-list xs*) \$ *i* = *xs* ! *i*
⟨*proof*⟩

lemma *vec-of-list-index*: *vec-of-list xs* \$ *j* = *xs* ! *j*
⟨*proof*⟩

lemma *list-of-vec-index*: *list-of-vec v* ! *j* = *v* \$ *j*
⟨*proof*⟩

lemma *list-of-vec-map*: *list-of-vec xs* = *map* ((\$) *xs*) [*0* ..<*dim-vec xs*] ⟨*proof*⟩

definition *component-mult v w* = *vec* (*min* (*dim-vec v*) (*dim-vec w*)) ($\lambda i.$ *v* \$ *i* ∗ *w* \$ *i*)

definition *vec-set*::*'a vec* ⇒ *'a set* (set_v)
where *vec-set v* = *vec-index v* ‘ {..<*dim-vec v*}

lemma *vec-set-map*[*simp*]: set_v (*map-vec f v*) = *f* ‘ set_v *v*
⟨*proof*⟩

lemma *index-component-mult*:
assumes *i* < *dim-vec v i* < *dim-vec w*
shows *component-mult v w* \$ *i* = *v* \$ *i* ∗ *w* \$ *i*
⟨*proof*⟩

lemma *dim-component-mult*:
dim-vec (*component-mult v w*) = *min* (*dim-vec v*) (*dim-vec w*)
⟨*proof*⟩

lemma *vec-setE*:
assumes *a* ∈ set_v *v*
obtains *i* **where** *v*\$*i* = *a i*<*dim-vec v* ⟨*proof*⟩

lemma *vec-setI*:
assumes *v*\$*i* = *a i*<*dim-vec v*
shows *a* ∈ set_v *v* ⟨*proof*⟩

lemma *set-list-of-vec*: *set* (*list-of-vec v*) = set_v *v* ⟨*proof*⟩

instantiation *vec* :: (*conjugate*) *conjugate*
begin

definition *conjugate-vec* :: ′*a* :: *conjugate vec* ⇒ ′*a vec*
where *conjugate v* = *vec* (*dim-vec v*) (λ*i*. *conjugate* (*v* \$ *i*))

lemma *conjugate-vCons* [*simp*]:
conjugate (*vCons a v*) = *vCons* (*conjugate a*) (*conjugate v*)
⟨*proof*⟩

lemma *dim-vec-conjugate*[*simp*]: *dim-vec* (*conjugate v*) = *dim-vec v*
⟨*proof*⟩

lemma *carrier-vec-conjugate*[*simp*]: *v* ∈ *carrier-vec n* ⟹ *conjugate v* ∈ *carrier-vec n*
⟨*proof*⟩

lemma *vec-index-conjugate*[*simp*]:
shows *i* < *dim-vec v* ⟹ *conjugate v* \$ *i* = *conjugate* (*v* \$ *i*)
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *conjugate-add-vec*:
fixes *v w* :: *'a* :: *conjugatable-ring vec*
assumes *dim*: *v* : *carrier-vec n w* : *carrier-vec n*
shows *conjugate* (*v* + *w*) = *conjugate v* + *conjugate w*
⟨*proof*⟩

lemma *uminus-conjugate-vec*:
fixes *v w* :: *'a* :: *conjugatable-ring vec*
shows − (*conjugate v*) = *conjugate* (− *v*)
⟨*proof*⟩

lemma *conjugate-zero-vec*[*simp*]:
conjugate (0_v *n* :: *'a* :: *conjugatable-ring vec*) = 0_v *n* ⟨*proof*⟩

lemma *conjugate-vec-0*[*simp*]:
conjugate (*vec 0 f*) = *vec 0 f* ⟨*proof*⟩

lemma *sprod-vec-0*[*simp*]: *v* · *vec 0 f* = *0*
⟨*proof*⟩

lemma *conjugate-zero-iff-vec*[*simp*]:
fixes *v* :: *'a* :: *conjugatable-ring vec*
shows *conjugate v* = 0_v *n* ⟷ *v* = 0_v *n*
⟨*proof*⟩

lemma *conjugate-smult-vec*:
fixes *k* :: *'a* :: *conjugatable-ring*
shows *conjugate* (*k* $\cdot_v$ *v*) = *conjugate k* $\cdot_v$ *conjugate v*
⟨*proof*⟩

lemma *conjugate-sprod-vec*:
fixes *v w* :: *'a* :: *conjugatable-ring vec*
assumes *v*: *v* : *carrier-vec n* **and** *w*: *w* : *carrier-vec n*
shows *conjugate* (*v* · *w*) = *conjugate v* · *conjugate w*
⟨*proof*⟩

abbreviation *cscalar-prod* :: *'a vec* ⇒ *'a vec* ⇒ *'a* :: *conjugatable-ring* (**infix** ·*c* *70*)
where (·*c*) ≡ λ*v w*. *v* · *conjugate w*

lemma *conjugate-conjugate-sprod*[*simp*]:
assumes *v*[*simp*]: *v* : *carrier-vec n* **and** *w*[*simp*]: *w* : *carrier-vec n*
shows *conjugate* (*conjugate v* · *w*) = *v* ·*c* *w*
⟨*proof*⟩

lemma *conjugate-vec-sprod-comm*:
fixes *v w* :: *'a* :: {*conjugatable-ring*, *comm-ring*} *vec*
assumes *v* : *carrier-vec n* **and** *w* : *carrier-vec n*
shows *v* ·*c* *w* = (*conjugate w* · *v*)

⟨*proof*⟩

lemma *conjugate-square-ge-0-vec*[*intro*!]:
fixes $v :: 'a :: \mathit{conjugatable\text{-}ordered\text{-}ring}\ \mathit{vec}$
shows $v \cdot c\ v \geq 0$
⟨*proof*⟩

lemma *conjugate-square-eq-0-vec*[*simp*]:
fixes $v :: 'a :: \{\mathit{conjugatable\text{-}ordered\text{-}ring}, \mathit{semiring\text{-}no\text{-}zero\text{-}divisors}\}\ \mathit{vec}$
assumes $v \in \mathit{carrier\text{-}vec}\ n$
shows $v \cdot c\ v = 0 \longleftrightarrow v = 0_v\ n$
⟨*proof*⟩

lemma *conjugate-square-greater-0-vec*[*simp*]:
fixes $v :: 'a :: \{\mathit{conjugatable\text{-}ordered\text{-}ring}, \mathit{semiring\text{-}no\text{-}zero\text{-}divisors}\}\ \mathit{vec}$
assumes $v \in \mathit{carrier\text{-}vec}\ n$
shows $v \cdot c\ v > 0 \longleftrightarrow v \neq 0_v\ n$
⟨*proof*⟩

lemma *vec-conjugate-rat*[*simp*]: $(\mathit{conjugate} :: \mathit{rat}\ \mathit{vec} \Rightarrow \mathit{rat}\ \mathit{vec}) = (\lambda x.\ x)$ ⟨*proof*⟩
lemma *vec-conjugate-real*[*simp*]: $(\mathit{conjugate} :: \mathit{real}\ \mathit{vec} \Rightarrow \mathit{real}\ \mathit{vec}) = (\lambda x.\ x)$ ⟨*proof*⟩

end

5 Code Generation for Basic Matrix Operations

In this theory we implement matrices as arrays of arrays. Due to the target language serialization, access to matrix entries should be constant time. Hence operations like matrix addition, multiplication, etc. should all have their standard complexity.

There might be room for optimizations.

To implement the infinite carrier set, we use A. Lochbihler's container framework [4].

theory *Matrix-IArray-Impl*
imports
Matrix
HOL−Library.IArray
Containers.Set-Impl
begin

typedef $'a\ \mathit{vec\text{-}impl} = \{(n,v :: 'a\ \mathit{iarray}).\ \mathit{IArray.length}\ v = n\}$ ⟨*proof*⟩
typedef $'a\ \mathit{mat\text{-}impl} = \{(nr,nc,m :: 'a\ \mathit{iarray}\ \mathit{iarray}).$
$\mathit{IArray.length}\ m = nr \wedge \mathit{IArray.all}\ (\lambda\ r.\ \mathit{IArray.length}\ r = nc)\ m\}$
⟨*proof*⟩

setup-lifting *type-definition-vec-impl*

setup-lifting *type-definition-mat-impl*

lift-definition *vec-impl* :: *'a vec-impl* ⇒ *'a vec* **is**
λ (*n*,*v*). (*n*,*mk-vec n* (*IArray.sub v*)) ⟨*proof*⟩

lift-definition *vec-add-impl* :: *'a::plus vec-impl* ⇒ *'a vec-impl* ⇒ *'a vec-impl* **is**
λ (*n*,*v*) (*m*,*w*).
(*n*, *IArray.of-fun* (*λi. IArray.sub v i* + *IArray.sub w i*) *n*)
⟨*proof*⟩

lift-definition *mat-impl* :: *'a mat-impl* ⇒ *'a mat* **is**
λ (*nr*,*nc*,*m*). (*nr*,*nc*,*mk-mat nr nc* (*λ* (*i*,*j*). *IArray.sub* (*IArray.sub m i*) *j*)) ⟨*proof*⟩

lift-definition *vec-of-list-impl* :: *'a list* ⇒ *'a vec-impl* **is**
λ v. (*length v*, *IArray v*) ⟨*proof*⟩

lift-definition *list-of-vec-impl* :: *'a vec-impl* ⇒ *'a list* **is**
λ (*n*,*v*). *IArray.list-of v* ⟨*proof*⟩

lift-definition *vec-of-fun* :: *nat* ⇒ (*nat* ⇒ *'a*) ⇒ *'a vec-impl* **is**
λ n f. (*n*, *IArray.of-fun f n*) ⟨*proof*⟩

lift-definition *mat-of-fun* :: *nat* ⇒ *nat* ⇒ (*nat* × *nat* ⇒ *'a*) ⇒ *'a mat-impl* **is**
λ nr nc f. (*nr*, *nc*, *IArray.of-fun* (*λ i. IArray.of-fun* (*λ j. f* (*i*,*j*)) *nc*) *nr*) ⟨*proof*⟩

lift-definition *vec-index-impl* :: *'a vec-impl* ⇒ *nat* ⇒ *'a*
is *λ* (*n*,*v*). *IArray.sub v* ⟨*proof*⟩

lift-definition *index-mat-impl* :: *'a mat-impl* ⇒ *nat* × *nat* ⇒ *'a*
is *λ* (*nr*,*nc*,*m*) (*i*,*j*). *if i* < *nr then IArray.sub* (*IArray.sub m i*) *j*
else IArray.sub (*IArray* ([] ! (*i* − *nr*))) *j* ⟨*proof*⟩

lift-definition *vec-equal-impl* :: *'a vec-impl* ⇒ *'a vec-impl* ⇒ *bool*
is *λ* (*n1*,*v1*) (*n2*,*v2*). *n1* = *n2* ∧ *v1* = *v2* ⟨*proof*⟩

lift-definition *mat-equal-impl* :: *'a mat-impl* ⇒ *'a mat-impl* ⇒ *bool*
is *λ* (*nr1*,*nc1*,*m1*) (*nr2*,*nc2*,*m2*). *nr1* = *nr2* ∧ *nc1* = *nc2* ∧ *m1* = *m2* ⟨*proof*⟩

lift-definition *dim-vec-impl* :: *'a vec-impl* ⇒ *nat* **is** *fst* ⟨*proof*⟩

lift-definition *dim-row-impl* :: *'a mat-impl* ⇒ *nat* **is** *fst* ⟨*proof*⟩
lift-definition *dim-col-impl* :: *'a mat-impl* ⇒ *nat* **is** *fst o snd* ⟨*proof*⟩

code-datatype *vec-impl*
code-datatype *mat-impl*

lemma *vec-code*[*code*]: *vec n f* = *vec-impl* (*vec-of-fun n f*)
⟨*proof*⟩

lemma *mat-code*[*code*]: *mat nr nc f* = *mat-impl* (*mat-of-fun nr nc f*)
⟨*proof*⟩

lemma *vec-of-list*[*code*]: *vec-of-list v* = *vec-impl* (*vec-of-list-impl v*)
⟨*proof*⟩

lemma *list-of-vec-code*[*code*]: *list-of-vec* (*vec-impl v*) = *list-of-vec-impl v*
⟨*proof*⟩

lemma *empty-nth*: $\neg$ *i* < *length x* $\Longrightarrow$ *x* ! *i* = [] ! (*i* − *length x*)
⟨*proof*⟩

lemma *undef-vec*: $\neg$ *i* < *length x* $\Longrightarrow$ *undef-vec* (*i* − *length x*) = *x* ! *i*
⟨*proof*⟩

lemma *vec-index-code*[*code*]: (*vec-impl v*) $ *i* = *vec-index-impl v i*
⟨*proof*⟩

lemma *index-mat-code*[*code*]: (*mat-impl m*) $$ *ij* = (*index-mat-impl m ij* :: *'a*)
⟨*proof*⟩

lift-definition (*code-dt*) *mat-of-rows-list-impl* :: *nat* ⇒ *'a list list* ⇒ *'a mat-impl option* **is**
λ *n rows. if list-all* (λ *r. length r* = *n*) *rows then Some* (*length rows*, *n*, *IArray* (*map IArray rows*))
else None
⟨*proof*⟩

lemma *mat-of-rows-list-impl*: *mat-of-rows-list-impl n rs* = *Some A* $\Longrightarrow$ *mat-impl A* = *mat-of-rows-list n rs*
⟨*proof*⟩

lemma *mat-of-rows-list-code*[*code*]: *mat-of-rows-list nc vs* =
(*case mat-of-rows-list-impl nc vs of Some A* ⇒ *mat-impl A*
| *None* ⇒ *mat-of-rows nc* (*map* (λ *v. vec nc* (*nth v*)) *vs*))
⟨*proof*⟩

lemma *dim-vec-code*[*code*]: *dim-vec* (*vec-impl v*) = *dim-vec-impl v*
⟨*proof*⟩

lemma *dim-row-code*[*code*]: *dim-row* (*mat-impl m*) = *dim-row-impl m*
⟨*proof*⟩

lemma *dim-col-code*[*code*]: *dim-col* (*mat-impl m*) = *dim-col-impl m*
⟨*proof*⟩

instantiation *vec* :: (*type*)*equal*
begin
definition (*equal-vec* :: (*'a vec* ⇒ *'a vec* ⇒ *bool*)) = (=)

instance
⟨*proof*⟩
end

instantiation *mat* :: (*type*)*equal*
begin
definition (*equal-mat* :: (*'a mat* ⇒ *'a mat* ⇒ *bool*)) = (=)
instance
⟨*proof*⟩
end

lemma *veq-equal-code*[*code*]: *HOL.equal* (*vec-impl* (*v1* :: *'a vec-impl*)) (*vec-impl* *v2*) = *vec-equal-impl v1 v2*
⟨*proof*⟩

lemma *mat-equal-code*[*code*]: *HOL.equal* (*mat-impl* (*m1* :: *'a mat-impl*)) (*mat-impl* *m2*) = *mat-equal-impl m1 m2*
⟨*proof*⟩

declare *prod.set-conv-list*[*code del*, *code-unfold*]

derive (*eq*) *ceq mat vec*
derive (*no*) *ccompare mat vec*
derive (*dlist*) *set-impl mat vec*
derive (*no*) *cenum mat vec*

lemma *carrier-mat-code*[*code*]: *carrier-mat nr nc* = *Collect-set* (*λ A. dim-row A* = *nr* ∧ *dim-col A* = *nc*) ⟨*proof*⟩
lemma *carrier-vec-code*[*code*]: *carrier-vec n* = *Collect-set* (*λ v. dim-vec v* = *n*)
⟨*proof*⟩

end

6 Gauss-Jordan Algorithm

We define the elementary row operations and use them to implement the Gauss-Jordan algorithm to transform matrices into row-echelon-form. This algorithm is used to implement the inverse of a matrix and to derive certain results on determinants, as well as determine a basis of the kernel of a matrix.

theory *Gauss-Jordan-Elimination*
imports *Matrix*
begin

6.1 Row Operations

definition *mat-multrow-gen* :: (*'a* ⇒ *'a* ⇒ *'a*) ⇒ *nat* ⇒ *'a* ⇒ *'a mat* ⇒ *'a mat*
where
mat-multrow-gen mul k a A = *mat* (*dim-row A*) (*dim-col A*)

(λ (*i*,*j*). *if k* = *i then mul a* (*A* $$ (*i*,*j*)) *else A* $$ (*i*,*j*))

abbreviation *mat-multrow* :: *nat* ⇒ *'a* :: *semiring-1* ⇒ *'a mat* ⇒ *'a mat* (*multrow*)
where
multrow ≡ *mat-multrow-gen* ((∗))

lemmas *mat-multrow-def* = *mat-multrow-gen-def*

definition *multrow-mat* :: *nat* ⇒ *nat* ⇒ *'a* :: *semiring-1* ⇒ *'a mat* **where**
multrow-mat n k a = *mat n n*
(λ (*i*,*j*). *if k* = *i* ∧ *k* = *j then a else if i* = *j then 1 else 0*)

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

definition *swaprows-mat* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *'a* :: *semiring-1 mat* **where**
swaprows-mat n k l = *mat n n*
(λ (*i*,*j*). *if k* = *i* ∧ *l* = *j* ∨ *k* = *j* ∧ *l* = *i* ∨ *i* = *j* ∧ *i* ≠ *k* ∧ *i* ≠ *l then 1 else 0*)

definition *mat-addrow-gen* :: (*'a* ⇒ *'a* ⇒ *'a*) ⇒ (*'a* ⇒ *'a* ⇒ *'a*) ⇒ *'a* ⇒ *nat* ⇒ *nat* ⇒ *'a mat* ⇒ *'a mat* **where**
mat-addrow-gen ad mul a k l A = *mat* (*dim-row A*) (*dim-col A*)
(λ (*i*,*j*). *if k* = *i then ad* (*mul a* (*A* $$ (*l*,*j*))) (*A* $$ (*i*,*j*)) *else A* $$ (*i*,*j*))

abbreviation *mat-addrow* :: *'a* :: *semiring-1* ⇒ *nat* ⇒ *nat* ⇒ *'a mat* ⇒ *'a mat* (*addrow*) **where**
addrow ≡ *mat-addrow-gen* (+) ((∗))

lemmas *mat-addrow-def* = *mat-addrow-gen-def*

definition *addrow-mat* :: *nat* ⇒ *'a* :: *semiring-1* ⇒ *nat* ⇒ *nat* ⇒ *'a mat* **where**
addrow-mat n a k l = *mat n n* (λ (*i*,*j*).
(*if k* = *i* ∧ *l* = *j then* (+) *a else id*) (*if i* = *j then 1 else 0*))

lemma *index-mat-multrow*[*simp*]:
i < *dim-row A* ⟹ *j* < *dim-col A* ⟹ *mat-multrow-gen mul k a A* $$ (*i*,*j*) = (*if k* = *i then mul a* (*A* $$ (*i*,*j*)) *else A* $$ (*i*,*j*))
i < *dim-row A* ⟹ *j* < *dim-col A* ⟹ *mat-multrow-gen mul i a A* $$ (*i*,*j*) = *mul a* (*A* $$ (*i*,*j*))
i < *dim-row A* ⟹ *j* < *dim-col A* ⟹ *k* ≠ *i* ⟹ *mat-multrow-gen mul k a A* $$ (*i*,*j*) = *A* $$ (*i*,*j*)
dim-row (*mat-multrow-gen mul k a A*) = *dim-row A dim-col* (*mat-multrow-gen mul k a A*) = *dim-col A*
⟨*proof*⟩

lemma *index-mat-multrow-mat*[*simp*]:
i < *n* ⟹ *j* < *n* ⟹ *multrow-mat n k a* $$ (*i*,*j*) = (*if k* = *i* ∧ *k* = *j then a else if i* = *j*

then 1 else 0)
dim-row (multrow-mat n k a) = n dim-col (multrow-mat n k a) = n
⟨proof⟩

lemma index-mat-swaprows[simp]:
i < dim-row A ⟹ j < dim-col A ⟹ swaprows k l A $$ (i,j) = (if k = i then A $$ (l,j) else
if l = i then A $$ (k,j) else A $$ (i,j))
dim-row (swaprows k l A) = dim-row A dim-col (swaprows k l A) = dim-col A
⟨proof⟩

lemma index-mat-swaprows-mat[simp]:
i < n ⟹ j < n ⟹ swaprows-mat n k l $$ (i,j) =
(if k = i ∧ l = j ∨ k = j ∧ l = i ∨ i = j ∧ i ≠ k ∧ i ≠ l then 1 else 0)
dim-row (swaprows-mat n k l) = n dim-col (swaprows-mat n k l) = n
⟨proof⟩

lemma index-mat-addrow[simp]:
i < dim-row A ⟹ j < dim-col A ⟹ mat-addrow-gen ad mul a k l A $$ (i,j) = (if k = i then
ad (mul a (A $$ (l,j))) (A $$ (i,j)) else A $$ (i,j))
i < dim-row A ⟹ j < dim-col A ⟹ mat-addrow-gen ad mul a i l A $$ (i,j) = ad (mul a (A $$ (l,j))) (A $$ (i,j))
i < dim-row A ⟹ j < dim-col A ⟹ k ≠ i ⟹ mat-addrow-gen ad mul a k l A $$ (i,j) = A $$(i,j)
dim-row (mat-addrow-gen ad mul a k l A) = dim-row A dim-col (mat-addrow-gen ad mul a k l A) = dim-col A
⟨proof⟩

lemma index-mat-addrow-mat[simp]:
i < n ⟹ j < n ⟹ addrow-mat n a k l $$ (i,j) =
(if k = i ∧ l = j then (+) a else id) (if i = j then 1 else 0)
dim-row (addrow-mat n a k l) = n dim-col (addrow-mat n a k l) = n
⟨proof⟩

lemma multrow-carrier[simp]: (mat-multrow-gen mul k a A ∈ carrier-mat n nc) = (A ∈ carrier-mat n nc)
⟨proof⟩

lemma multrow-mat-carrier[simp]: multrow-mat n k a ∈ carrier-mat n n
⟨proof⟩

lemma addrow-mat-carrier[simp]: addrow-mat n a k l ∈ carrier-mat n n
⟨proof⟩

lemma swaprows-mat-carrier[simp]: swaprows-mat n k l ∈ carrier-mat n n
⟨proof⟩

lemma swaprows-carrier[simp]: (swaprows k l A ∈ carrier-mat n nc) = (A ∈ car-

rier-mat n nc)
⟨*proof*⟩

lemma *addrow-carrier*[*simp*]: (*mat-addrow-gen ad mul a k l A* $\in$ *carrier-mat n nc*) = (*A* $\in$ *carrier-mat n nc*)
⟨*proof*⟩

lemma *row-multrow*: $k \neq i \Longrightarrow i < n \Longrightarrow$ *row* (*multrow-mat n k a*) *i* = *unit-vec n i*
$k < n \Longrightarrow$ *row* (*multrow-mat n k a*) *k* = *a* $\cdot_v$ *unit-vec n k*
⟨*proof*⟩

lemma *multrow-mat*: **assumes** *A*: *A* $\in$ *carrier-mat n nc*
shows *multrow k a A* = *multrow-mat n k a* $*$ *A*
⟨*proof*⟩

lemma *row-addrow*:
$k \neq i \Longrightarrow i < n \Longrightarrow$ *row* (*addrow-mat n a k l*) *i* = *unit-vec n i*
$k < n \Longrightarrow l < n \Longrightarrow$ *row* (*addrow-mat n a k l*) *k* = *a* $\cdot_v$ *unit-vec n l* + *unit-vec n k*
⟨*proof*⟩

lemma *addrow-mat*: **assumes** *A*: *A* $\in$ *carrier-mat n nc*
and *l*: $l < n$
shows *addrow a k l A* = *addrow-mat n a k l* $*$ *A*
⟨*proof*⟩

lemma *row-swaprows*:
$l < n \Longrightarrow$ *row* (*swaprows-mat n l l*) *l* = *unit-vec n l*
$i \neq k \Longrightarrow i \neq l \Longrightarrow i < n \Longrightarrow$ *row* (*swaprows-mat n k l*) *i* = *unit-vec n i*
$k < n \Longrightarrow l < n \Longrightarrow$ *row* (*swaprows-mat n k l*) *l* = *unit-vec n k*
$k < n \Longrightarrow l < n \Longrightarrow$ *row* (*swaprows-mat n k l*) *k* = *unit-vec n l*
⟨*proof*⟩

lemma *swaprows-mat*: **assumes** *A*: *A* $\in$ *carrier-mat n nc* **and** *k*: $k < n$ $l < n$
shows *swaprows k l A* = *swaprows-mat n k l* $*$ *A*
⟨*proof*⟩

lemma *swaprows-mat-inv*: **assumes** *k*: $k < n$ **and** *l*: $l < n$
shows *swaprows-mat n k l* $*$ *swaprows-mat n k l* = 1_m *n*
⟨*proof*⟩

lemma *swaprows-mat-Unit*: **assumes** *k*: $k < n$ **and** *l*: $l < n$
shows *swaprows-mat n k l* $\in$ *Units* (*ring-mat TYPE*(*'a* :: *semiring-1*) *n b*)
⟨*proof*⟩

lemma *addrow-mat-inv*: **assumes** *k*: $k < n$ **and** *l*: $l < n$ **and** *neq*: $k \neq l$
shows *addrow-mat n a k l* $*$ *addrow-mat n* ($-$ (*a* :: *'a* :: *comm-ring-1*)) *k l* = 1_m *n*

⟨*proof*⟩

lemma *addrow-mat-Unit*: **assumes** *k*: *k* < *n* **and** *l*: *l* < *n* **and** *neq*: *k* ≠ *l*
shows *addrow-mat n a k l* ∈ *Units* (*ring-mat TYPE*(*'a* :: *comm-ring-1*) *n b*)
⟨*proof*⟩

lemma *multrow-mat-inv*: **assumes** *k*: *k* < *n* **and** *a*: (*a* :: *'a* :: *division-ring*) ≠ *0*
shows *multrow-mat n k a* ∗ *multrow-mat n k* (*inverse a*) = 1_m *n*
⟨*proof*⟩

lemma *multrow-mat-Unit*: **assumes** *k*: *k* < *n* **and** *a*: (*a* :: *'a* :: *division-ring*) ≠ *0*
shows *multrow-mat n k a* ∈ *Units* (*ring-mat TYPE*(*'a*) *n b*)
⟨*proof*⟩

6.2 Gauss-Jordan Elimination

fun *eliminate-entries-rec* **where**
eliminate-entries-rec B i [] = *B*
| *eliminate-entries-rec B i* ((*ai'j*,*i'*) # *is*) = (
eliminate-entries-rec (*mat-addrow-gen* ((+) :: *'b* :: *ring-1* ⇒ *'b* ⇒ *'b*) (∗) *ai'j i' i B*) *i is*)

context
fixes *minus* :: *'a* ⇒ *'a* ⇒ *'a*
and *times* :: *'a* ⇒ *'a* ⇒ *'a*
begin

definition *eliminate-entries-gen* :: (*nat* ⇒ *'a*) ⇒ *'a mat* ⇒ *nat* ⇒ *nat* ⇒ *'a mat* **where**
eliminate-entries-gen v A I J = *mat* (*dim-row A*) (*dim-col A*) (λ (*i*, *j*).
if i ≠ *I then minus* (*A* $$ (*i*,*j*)) (*times* (*v i*) (*A* $$ (*I*,*j*))) *else A* $$ (*i*,*j*))

lemma *dim-eliminate-entries-gen*[*simp*]: *dim-row* (*eliminate-entries-gen v B i as*) = *dim-row B*
dim-col (*eliminate-entries-gen v B i as*) = *dim-col B*
⟨*proof*⟩

lemma *dimc-eliminate-entries-rec*[*simp*]: *dim-col* (*eliminate-entries-rec B i as*) = *dim-col B*
⟨*proof*⟩

lemma *dimr-eliminate-entries-rec*[*simp*]: *dim-row* (*eliminate-entries-rec B i as*) = *dim-row B*
⟨*proof*⟩

lemma *carrier-eliminate-entries*: *A* ∈ *carrier-mat nr nc* ⟹ *eliminate-entries-gen v A i bs* ∈ *carrier-mat nr nc*
B ∈ *carrier-mat nr nc* ⟹ *eliminate-entries-rec B i as* ∈ *carrier-mat nr nc*
⟨*proof*⟩

end

abbreviation *eliminate-entries* ≡ *eliminate-entries-gen* (−) ((∗) :: *'a* :: *ring-1* ⇒ *'a* ⇒ *'a*)

lemma *eliminate-entries-convert*:
assumes *jA*: *J* < *dim-col A* **and** ∗: *I* < *dim-row A dim-row B* = *dim-row A*
shows *eliminate-entries* (λ *i*. *A* $$ (*i*,*J*)) *B I J* =
eliminate-entries-rec B I (*map* (λ *i*. (− *A* $$ (*i*, *J*), *i*)) (*filter* (λ *i*. *i* ≠ *I*) [*0* ..< *dim-row A*]))
⟨*proof*⟩

lemma *Unit-prod-eliminate-entries*: *i* < *nr* ⟹ (⋀ *a i'*. (*a*, *i'*) ∈ *set is* ⟹ *i'* < *nr* ∧ *i'* ≠ *i*)
⟹ ∃ *P* ∈ *Units* (*ring-mat TYPE*(*'a* :: *comm-ring-1*) *nr b*) . ∀ *B nc*. *B* ∈ *carrier-mat nr nc* ⟶ *eliminate-entries-rec B i is* = *P* ∗ *B*
⟨*proof*⟩

function *gauss-jordan-main* :: *'a* :: *field mat* ⇒ *'a mat* ⇒ *nat* ⇒ *nat* ⇒ *'a mat* × *'a mat* **where**
gauss-jordan-main A B i j = (*let nr* = *dim-row A*; *nc* = *dim-col A in*
if i < *nr* ∧ *j* < *nc then let aij* = *A* $$ (*i*,*j*) *in if aij* = *0 then*
(*case* [*i'* . *i'* <− [*Suc i* ..< *nr*], *A* $$ (*i'*,*j*) ≠ *0*]
of [] ⇒ *gauss-jordan-main A B i* (*Suc j*)
| (*i'* # -) ⇒ *gauss-jordan-main* (*swaprows i i' A*) (*swaprows i i' B*) *i j*)
else if aij = *1 then let*
v = (λ *i*. *A* $$ (*i*,*j*)) *in*
gauss-jordan-main
(*eliminate-entries v A i j*) (*eliminate-entries v B i j*) (*Suc i*) (*Suc j*)
else let iaij = *inverse aij in gauss-jordan-main* (*multrow i iaij A*) (*multrow i iaij B*) *i j*
else (*A*,*B*))
⟨*proof*⟩

termination
⟨*proof*⟩

declare *gauss-jordan-main.simps*[*simp del*]

definition *gauss-jordan A B* ≡ *gauss-jordan-main A B 0 0*

lemma *gauss-jordan-transform*: **assumes** *A*: *A* ∈ *carrier-mat nr nc* **and** *B*: *B* ∈ *carrier-mat nr nc'*
and *res*: *gauss-jordan* (*A* :: *'a* :: *field mat*) *B* = (*A'*,*B'*)
shows ∃ *P* ∈ *Units* (*ring-mat TYPE*(*'a*) *nr b*). *A'* = *P* ∗ *A* ∧ *B'* = *P* ∗ *B*
⟨*proof*⟩

lemma *gauss-jordan-carrier*: **assumes** *A*: (*A* :: *'a* :: *field mat*) ∈ *carrier-mat nr nc*

and B: $B \in \textit{carrier-mat nr nc}'$
and *res*: $\textit{gauss-jordan } A\ B = (A',B')$
shows $A' \in \textit{carrier-mat nr nc } B' \in \textit{carrier-mat nr nc}'$
$\langle \textit{proof} \rangle$

definition *pivot-fun* :: $'a :: \{\textit{zero},\textit{one}\}\ \textit{mat} \Rightarrow (\textit{nat} \Rightarrow \textit{nat}) \Rightarrow \textit{nat} \Rightarrow \textit{bool}$ **where**
$\textit{pivot-fun } A\ f\ nc \equiv \textit{let } nr = \textit{dim-row } A \textit{ in}$
$(\forall\ i < nr.\ f\ i \leq nc\ \wedge$
$(f\ i < nc \longrightarrow A\ \$\$\ (i,\ f\ i) = 1 \wedge (\forall\ i' < nr.\ i' \neq i \longrightarrow A\ \$\$\ (i',f\ i) = 0)) \wedge$
$(\forall\ j < f\ i.\ A\ \$\$\ (i,\ j) = 0) \wedge$
$(\textit{Suc } i < nr \longrightarrow f\ (\textit{Suc } i) > f\ i \vee f\ (\textit{Suc } i) = nc))$

lemma *pivot-funI*: **assumes** d: $\textit{dim-row } A = nr$
and $*$: $\bigwedge\ i.\ i < nr \Longrightarrow f\ i \leq nc$
$\bigwedge\ i\ j.\ i < nr \Longrightarrow j < f\ i \Longrightarrow A\ \$\$\ (i,j) = 0$
$\bigwedge\ i.\ i < nr \Longrightarrow \textit{Suc } i < nr \Longrightarrow f\ (\textit{Suc } i) > f\ i \vee f\ (\textit{Suc } i) = nc$
$\bigwedge\ i.\ i < nr \Longrightarrow f\ i < nc \Longrightarrow A\ \$\$\ (i,\ f\ i) = 1$
$\bigwedge\ i\ i'.\ i < nr \Longrightarrow f\ i < nc \Longrightarrow i' < nr \Longrightarrow i' \neq i \Longrightarrow A\ \$\$\ (i',f\ i) = 0$
shows $\textit{pivot-fun } A\ f\ nc$
$\langle \textit{proof} \rangle$

lemma *pivot-funD*: **assumes** d: $\textit{dim-row } A = nr$
and p: $\textit{pivot-fun } A\ f\ nc$
shows $\bigwedge\ i.\ i < nr \Longrightarrow f\ i \leq nc$
$\bigwedge\ i\ j.\ i < nr \Longrightarrow j < f\ i \Longrightarrow A\ \$\$\ (i,j) = 0$
$\bigwedge\ i.\ i < nr \Longrightarrow \textit{Suc } i < nr \Longrightarrow f\ (\textit{Suc } i) > f\ i \vee f\ (\textit{Suc } i) = nc$
$\bigwedge\ i.\ i < nr \Longrightarrow f\ i < nc \Longrightarrow A\ \$\$\ (i,\ f\ i) = 1$
$\bigwedge\ i\ i'.\ i < nr \Longrightarrow f\ i < nc \Longrightarrow i' < nr \Longrightarrow i' \neq i \Longrightarrow A\ \$\$\ (i',f\ i) = 0$
$\langle \textit{proof} \rangle$

lemma *pivot-fun-multrow*: **assumes** p: $\textit{pivot-fun } A\ f\ jj$
and d: $\textit{dim-row } A = nr\ \textit{dim-col } A = nc$
and *fi*: $f\ i0 = jj$
and *jj*: $jj \leq nc$
shows $\textit{pivot-fun } (\textit{multrow } i0\ a\ A)\ f\ jj$
$\langle \textit{proof} \rangle$

lemma *pivot-fun-swaprows*: **assumes** p: $\textit{pivot-fun } A\ f\ jj$
and d: $\textit{dim-row } A = nr\ \textit{dim-col } A = nc$
and *flk*: $f\ l = jj\ f\ k = jj$
and *nr*: $l < nr\ k < nr$
and *jj*: $jj \leq nc$
shows $\textit{pivot-fun } (\textit{swaprows } l\ k\ A)\ f\ jj$
$\langle \textit{proof} \rangle$

lemma *pivot-fun-eliminate-entries*: **assumes** p: $\textit{pivot-fun } A\ f\ jj$
and d: $\textit{dim-row } A = nr\ \textit{dim-col } A = nc$
and *fl*: $f\ l = jj$

and *nr*: $l < nr$
and *jj*: $jj \leq nc$
shows *pivot-fun* (*eliminate-entries vs A l j*) *f jj*
⟨*proof*⟩

definition *row-echelon-form* :: *'a* :: {*zero*,*one*} *mat* ⇒ *bool* **where**
row-echelon-form A ≡ ∃ *f*. *pivot-fun A f* (*dim-col A*)

lemma *pivot-fun-init*: *pivot-fun A* (λ -. *0*) *0*
⟨*proof*⟩

lemma *gauss-jordan-main-row-echelon*:
assumes
$A \in$ *carrier-mat nr nc*
gauss-jordan-main A B i j = (A',B')
pivot-fun A f j
$\bigwedge i'. \; i' < i \Longrightarrow f\, i' < j \; \bigwedge i'. \; i' \geq i \Longrightarrow f\, i' = j$
$i \leq nr\; j \leq nc$
shows *row-echelon-form* A'
⟨*proof*⟩

lemma *gauss-jordan-row-echelon*:
assumes *A*: $A \in$ *carrier-mat nr nc*
and *res*: *gauss-jordan A B* = (A', B')
shows *row-echelon-form* A'
⟨*proof*⟩

lemma *pivot-bound*: **assumes** *dim*: *dim-row A* = *nr*
and *pivot*: *pivot-fun A f n*
shows $i + j < nr \Longrightarrow f\ (i + j) = n \lor f\ (i + j) \geq j + f\, i$
⟨*proof*⟩

context
fixes *zero* :: *'a*
and *A* :: *'a mat*
and *nr nc* :: *nat*
begin
function *pivot-positions-main-gen* :: *nat* ⇒ *nat* ⇒ (*nat* × *nat*) *list* **where**
pivot-positions-main-gen i j = (
if $i < nr$ *then*
if $j < nc$ *then*
if A $$ (*i*,*j*) = *zero then*
pivot-positions-main-gen i (*Suc j*)
else (*i*,*j*) # *pivot-positions-main-gen* (*Suc i*) (*Suc j*)
else []
else []) ⟨*proof*⟩

termination ⟨*proof*⟩

declare *pivot-positions-main-gen.simps*[*simp del*]
end

context
  fixes $A :: {}'a :: \mathit{semiring\text{-}1\ mat}$
  and $\mathit{nr}\ \mathit{nc} :: \mathit{nat}$
begin

abbreviation $\mathit{pivot\text{-}positions\text{-}main} \equiv \mathit{pivot\text{-}positions\text{-}main\text{-}gen}\ (0 :: {}'a)\ A\ \mathit{nr}\ \mathit{nc}$

lemma *pivot-positions-main*: **assumes** *A*: $A \in \mathit{carrier\text{-}mat}\ \mathit{nr}\ \mathit{nc}$
  and *pivot*: $\mathit{pivot\text{-}fun}\ A\ f\ \mathit{nc}$
  shows $j \le f\ i \vee i \ge \mathit{nr} \Longrightarrow$
    $\mathit{set}\ (\mathit{pivot\text{-}positions\text{-}main}\ i\ j) = \{(i', f\ i') \mid i'.\ i \le i' \wedge i' < \mathit{nr}\} - \mathit{UNIV} \times \{\mathit{nc}\}$
    $\wedge\ \mathit{distinct}\ (\mathit{map}\ \mathit{snd}\ (\mathit{pivot\text{-}positions\text{-}main}\ i\ j))$
    $\wedge\ \mathit{distinct}\ (\mathit{map}\ \mathit{fst}\ (\mathit{pivot\text{-}positions\text{-}main}\ i\ j))$
⟨*proof*⟩
end

lemma *pivot-fun-zero-row-iff*: **assumes** *pivot*: $\mathit{pivot\text{-}fun}\ (A :: {}'a :: \mathit{semiring\text{-}1\ mat})\ f\ \mathit{nc}$
  and *A*: $A \in \mathit{carrier\text{-}mat}\ \mathit{nr}\ \mathit{nc}$
  and *i*: $i < \mathit{nr}$
  shows $f\ i = \mathit{nc} \longleftrightarrow \mathit{row}\ A\ i = 0_v\ \mathit{nc}$
⟨*proof*⟩

definition $\mathit{pivot\text{-}positions\text{-}gen} :: {}'a \Rightarrow {}'a\ \mathit{mat} \Rightarrow (\mathit{nat} \times \mathit{nat})\ \mathit{list}$ **where**
  $\mathit{pivot\text{-}positions\text{-}gen}\ \mathit{zer}\ A \equiv \mathit{pivot\text{-}positions\text{-}main\text{-}gen}\ \mathit{zer}\ A\ (\mathit{dim\text{-}row}\ A)\ (\mathit{dim\text{-}col}\ A)\ 0\ 0$

abbreviation $\mathit{pivot\text{-}positions} :: {}'a :: \mathit{semiring\text{-}1\ mat} \Rightarrow (\mathit{nat} \times \mathit{nat})\ \mathit{list}$ **where**
  $\mathit{pivot\text{-}positions} \equiv \mathit{pivot\text{-}positions\text{-}gen}\ 0$

lemmas $\mathit{pivot\text{-}positions\text{-}def} = \mathit{pivot\text{-}positions\text{-}gen\text{-}def}$

lemma *pivot-positions*: **assumes** *A*: $A \in \mathit{carrier\text{-}mat}\ \mathit{nr}\ \mathit{nc}$
  and *pivot*: $\mathit{pivot\text{-}fun}\ A\ f\ \mathit{nc}$
  shows
    $\mathit{set}\ (\mathit{pivot\text{-}positions}\ A) = \{(i, f\ i) \mid i.\ i < \mathit{nr} \wedge f\ i \neq \mathit{nc}\}$
    $\mathit{distinct}\ (\mathit{map}\ \mathit{fst}\ (\mathit{pivot\text{-}positions}\ A))$
    $\mathit{distinct}\ (\mathit{map}\ \mathit{snd}\ (\mathit{pivot\text{-}positions}\ A))$
    $\mathit{length}\ (\mathit{pivot\text{-}positions}\ A) = \mathit{card}\ \{\ i.\ i < \mathit{nr} \wedge \mathit{row}\ A\ i \neq 0_v\ \mathit{nc}\}$
⟨*proof*⟩

context
  fixes $\mathit{uminus} :: {}'a \Rightarrow {}'a$
  and $\mathit{zero} :: {}'a$
  and $\mathit{one} :: {}'a$

begin
definition *non-pivot-base-gen* :: *'a mat* ⇒ (*nat* × *nat*)*list* ⇒ *nat* ⇒ *'a vec* **where**
non-pivot-base-gen A pivots ≡ *let nr* = *dim-row A*; *nc* = *dim-col A*;
invers = *map-of* (*map prod.swap pivots*)
in (λ *qj*. *vec nc* (λ *i*.
if i = *qj then one else* (*case invers i of Some j* => *uminus* (*A* $$ (*j*,*qj*)) | *None* ⇒ *zero*)))

definition *find-base-vectors-gen* :: *'a mat* ⇒ *'a vec list* **where**
find-base-vectors-gen A ≡
let
pp = *pivot-positions-gen zero A*;
cands = *filter* (λ *j*. *j* ∉ *set* (*map snd pp*)) [*0* ..< *dim-col A*]
in map (*non-pivot-base-gen A pp*) *cands*
end

abbreviation *non-pivot-base* ≡ *non-pivot-base-gen uminus 0* (*1* :: *'a* :: *comm-ring-1*)
abbreviation *find-base-vectors* ≡ *find-base-vectors-gen uminus 0* (*1* :: *'a* :: *comm-ring-1*)

lemmas *non-pivot-base-def* = *non-pivot-base-gen-def*
lemmas *find-base-vectors-def* = *find-base-vectors-gen-def*

The soundness of *find-base-vectors* is proven in theory Matrix-Kern, where it is shown that *find-base-vectors* is a basis of the kern of *A*.

definition *find-base-vector* :: *'a* :: *comm-ring-1 mat* ⇒ *'a vec* **where**
find-base-vector A ≡
let
pp = *pivot-positions A*;
cands = *filter* (λ *j*. *j* ∉ *set* (*map snd pp*)) [*0* ..< *dim-col A*]
in non-pivot-base A pp (*hd cands*)

context
fixes *A* :: *'a* :: *field mat* **and** *nr nc* :: *nat* **and** *p* :: *nat* ⇒ *nat*
assumes *ref*: *row-echelon-form A*
and *A*: *A* ∈ *carrier-mat nr nc*
begin

lemma *non-pivot-base*:
defines *pp*: *pp* ≡ *pivot-positions A*
assumes *qj*: *qj* < *nc qj* ∉ *snd* ‘ *set pp*
shows *non-pivot-base A pp qj* ∈ *carrier-vec nc*
non-pivot-base A pp qj $ *qj* = *1*
A $*_v$ *non-pivot-base A pp qj* = 0_v *nr*
⋀ *qj'*. *qj'* < *nc* ⟹ *qj'* ∉ *snd* ‘ *set pp* ⟹ *qj* ≠ *qj'* ⟹ *non-pivot-base A pp qj* $ *qj'* = *0*
⟨*proof*⟩

lemma *find-base-vector*: **assumes** *snd* ‘ *set* (*pivot-positions A*) ≠ {*0* ..< *nc*}
shows

find-base-vector $A \in$ *carrier-vec nc*
find-base-vector $A \neq 0_v$ *nc*
$A *_v$ *find-base-vector* $A = 0_v$ *nr*
⟨*proof*⟩
end

lemma *row-echelon-form-imp-1-or-0-row*: **assumes** *A*: $A \in$ *carrier-mat n n*
and *row*: *row-echelon-form A*
shows $A = 1_m\ n \vee (n > 0 \wedge row\ A\ (n - 1) = 0_v\ n)$
⟨*proof*⟩

context
fixes *A* :: *'a* :: *field mat* **and** *n* :: *nat* **and** *p* :: *nat* ⇒ *nat*
assumes *ref*: *row-echelon-form A*
and *A*: $A \in$ *carrier-mat n n*
and *1*: $A \neq 1_m\ n$
begin

lemma *find-base-vector-not-1-pivot-positions*: *snd* ‘ *set* (*pivot-positions A*) ≠ {*0* ..< *n*}
⟨*proof*⟩

lemma *find-base-vector-not-1*:
find-base-vector $A \in$ *carrier-vec n*
find-base-vector $A \neq 0_v\ n$
$A *_v$ *find-base-vector* $A = 0_v\ n$
⟨*proof*⟩
end

lemma *gauss-jordan*: **assumes** *A*: $A \in$ *carrier-mat nr nc*
and *B*: $B \in$ *carrier-mat nr nc2*
and *gauss*: *gauss-jordan A B* = (*C*,*D*)
shows $x \in$ *carrier-vec nc* $\Longrightarrow (A *_v x = 0_v\ nr) = (C *_v x = 0_v\ nr)$ (**is** - $\Longrightarrow$ *?l* = *?r*)
$X \in$ *carrier-mat nc nc2* $\Longrightarrow (A * X = B) = (C * X = D)$ (**is** - $\Longrightarrow$ *?l2* = *?r2*)
$C \in$ *carrier-mat nr nc*
$D \in$ *carrier-mat nr nc2*
⟨*proof*⟩

definition *gauss-jordan-single* :: *'a* :: *field mat* ⇒ *'a mat* **where**
gauss-jordan-single A = *fst* (*gauss-jordan A* (0_m (*dim-row A*) *0*))

lemma *gauss-jordan-single*: **assumes** *A*: $A \in$ *carrier-mat nr nc*
and *gauss*: *gauss-jordan-single A* = *C*
shows $x \in$ *carrier-vec nc* $\Longrightarrow (A *_v x = 0_v\ nr) = (C *_v x = 0_v\ nr)$
$C \in$ *carrier-mat nr nc*
row-echelon-form C
∃ *P Q*. $C = P * A \wedge P \in$ *carrier-mat nr nr* ∧ $Q \in$ *carrier-mat nr nr* ∧ $P *$

$Q = 1_m\ nr \wedge Q * P = 1_m\ nr$ (**is** *?ex*)
⟨*proof*⟩

lemma *gauss-jordan-inverse-one-direction*:
  assumes *A*: $A \in$ *carrier-mat n n* **and** *B*: $B \in$ *carrier-mat n nc*
  and *res*: *gauss-jordan A B* $= (1_m\ n,\ B')$
  shows $A \in$ *Units* (*ring-mat TYPE*(*'a* :: *field*) *n b*)
  $B = 1_m\ n \Longrightarrow A * B' = 1_m\ n \wedge B' * A = 1_m\ n$
⟨*proof*⟩

lemma *gauss-jordan-inverse-other-direction*:
 assumes *AU*: $A \in$ *Units* (*ring-mat TYPE*(*'a* :: *field*) *n b*) **and** *B*: $B \in$ *carrier-mat n nc*
  shows *fst* (*gauss-jordan A B*) $= 1_m\ n$
⟨*proof*⟩

lemma *gauss-jordan-compute-inverse*:
  assumes *A*: $A \in$ *carrier-mat n n*
  and *res*: *gauss-jordan A* $(1_m\ n) = (1_m\ n,\ B')$
  shows $A * B' = 1_m\ n$ $B' * A = 1_m\ n$ $B' \in$ *carrier-mat n n*
⟨*proof*⟩

lemma *gauss-jordan-check-invertable*: **assumes** *A*: $A \in$ *carrier-mat n n* **and** *B*: $B \in$ *carrier-mat n nc*
  shows ($A \in$ *Units* (*ring-mat TYPE*(*'a* :: *field*) *n b*)) $\longleftrightarrow$ *fst* (*gauss-jordan A B*) $= 1_m\ n$
  (**is** *?l* = *?r*)
⟨*proof*⟩

definition *mat-inverse* :: *'a* :: *field mat* ⇒ *'a mat option* **where**
  mat-inverse A = (*if dim-row A* = *dim-col A then*
    let one = 1_m (*dim-row A*) *in*
    (*case gauss-jordan A one of*
      (*B*, *C*) ⇒ *if B* = *one then Some C else None*) *else None*)

lemma *mat-inverse*: **assumes** *A*: $A \in$ *carrier-mat n n*
  shows *mat-inverse A* = *None* $\Longrightarrow A \notin$ *Units* (*ring-mat TYPE*(*'a* :: *field*) *n b*)
   mat-inverse A = *Some B* $\Longrightarrow A * B = 1_m\ n \wedge B * A = 1_m\ n \wedge B \in$ *carrier-mat n n*
⟨*proof*⟩
end

7 Code Generation for Basic Matrix Operations

In this theory we provide efficient implementations for the elementary row-transformations. These are necessary since the default implementations would construct a whole new matrix in every step.

theory *Gauss-Jordan-IArray-Impl*
imports
  Polynomial-Interpolation.Missing-Unsorted
  Matrix-IArray-Impl
  Gauss-Jordan-Elimination
begin

lift-definition *mat-swaprows-impl* :: *nat* ⇒ *nat* ⇒ *'a mat-impl* ⇒ *'a mat-impl* **is**
  λ i j (nr,nc,A). if i < nr ∧ j < nr then
  let Ai = IArray.sub A i;
      Aj = IArray.sub A j;
      Arows = IArray.list-of A;
      A' = IArray.IArray (Arows [i := Aj, j := Ai])
   in (nr,nc,A')
     else (nr,nc,A)
  ⟨*proof*⟩

lemma [*code*]: *mat-swaprows k l (mat-impl A) = (let nr = dim-row-impl A in*
  if l < nr ∧ k < nr then
  mat-impl (mat-swaprows-impl k l A) else Code.abort (STR ''index out of bounds in mat-swaprows'')
  (λ -. mat-swaprows k l (mat-impl A))) (**is** *?l = ?r*)
⟨*proof*⟩

lift-definition *mat-multrow-gen-impl* :: *('a ⇒ 'a ⇒ 'a) ⇒ nat ⇒ 'a ⇒ 'a mat-impl ⇒ 'a mat-impl* **is**
  λ mul k a (nr,nc,A). let Ak = IArray.sub A k; Arows = IArray.list-of A;
     Ak' = IArray.IArray (map (mul a) (IArray.list-of Ak));
     A' = IArray.IArray (Arows [k := Ak'])
     in (nr,nc,A')
⟨*proof*⟩

lemma [*code*]: *mat-multrow-gen mul k a (mat-impl A) = mat-impl (mat-multrow-gen-impl mul k a A)*
  ⟨*proof*⟩

lift-definition *mat-addrow-gen-impl*
  :: *('a ⇒ 'a ⇒ 'a) ⇒ ('a ⇒ 'a ⇒ 'a) ⇒ 'a ⇒ nat ⇒ nat ⇒ 'a mat-impl ⇒ 'a mat-impl* **is**
  λ ad mul a k l (nr,nc,A). if l < nr then let Ak = IArray.sub A k; Al = IArray.sub A l;
      Ak' = IArray.of-fun (λ i. ad (mul a (Al !! i)) (Ak !! i)) (min (IArray.length Ak) (IArray.length Al));
     A' = IArray.of-fun (λ i. if i = k then Ak' else A !! i) (IArray.length A)
     in (nr,nc,A') else (nr,nc,A)
⟨*proof*⟩

lemma mat-addrow-gen-impl[code]: mat-addrow-gen ad mul a k l (mat-impl A) =
(if l < dim-row-impl A then
 mat-impl (mat-addrow-gen-impl ad mul a k l A) else Code.abort (STR ''index out of bounds in mat-addrow'')
 (λ -. mat-addrow-gen ad mul a k l (mat-impl A))) (**is** ?l = ?r)
⟨proof⟩

lemma gauss-jordan-main-code[code]:
 gauss-jordan-main A B i j = (let nr = dim-row A; nc = dim-col A in
  if i < nr ∧ j < nc then let aij = A $$ (i,j) in if aij = 0 then
   (case [i' . i' <− [Suc i ..< nr], A $$ (i',j) ≠ 0]
    of [] ⇒ gauss-jordan-main A B i (Suc j)
     | (i' # -) ⇒ gauss-jordan-main (swaprows i i' A) (swaprows i i' B) i j)
   else if aij = 1 then let v = (λ i. A $$ (i,j)) in
    gauss-jordan-main
    (eliminate-entries v A i j) (eliminate-entries v B i j) (Suc i) (Suc j)
   else let iaij = inverse aij; A' = multrow i iaij A; B' = multrow i iaij B;
    v = (λ i. A' $$ (i,j)) in gauss-jordan-main
    (eliminate-entries v A' i j) (eliminate-entries v B' i j) (Suc i) (Suc j)
  else (A,B)) (**is** ?l = ?r)
⟨proof⟩

end

8 Elementary Column Operations

We define elementary column operations and also combine them with elementary row operations. These combined operations are the basis to perform operations which preserve similarity of matrices. They are applied later on to convert upper triangular matrices into Jordan normal form.

theory Column-Operations
imports
 Gauss-Jordan-Elimination
begin

definition mat-multcol :: nat ⇒ 'a :: semiring-1 ⇒ 'a mat ⇒ 'a mat (multcol)
where
 multcol k a A = mat (dim-row A) (dim-col A)
  (λ (i,j). if k = j then a ∗ A $$ (i,j) else A $$ (i,j))

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

definition mat-addcol-vec :: nat ⇒ 'a :: plus vec ⇒ 'a mat ⇒ 'a mat **where**
 mat-addcol-vec k v A = mat (dim-row A) (dim-col A)
 (λ (i,j). if k = j then v $ i + A $$ (i,j) else A $$ (i,j))

definition *mat-addcol* :: *'a* :: *semiring-1* ⇒ *nat* ⇒ *nat* ⇒ *'a mat* ⇒ *'a mat* (*addcol*)
where
addcol a k l A = *mat* (*dim-row A*) (*dim-col A*)
(λ (*i*,*j*). *if k* = *j then a* ∗ *A* $$ (*i*,*l*) + *A* $$ (*i*,*j*) *else A* $$ (*i*,*j*))

lemma *index-mat-multcol*[*simp*]:
i < *dim-row A* ⟹ *j* < *dim-col A* ⟹ *multcol k a A* $$ (*i*,*j*) = (*if k* = *j then a* ∗ *A* $$ (*i*,*j*) *else A* $$ (*i*,*j*))
i < *dim-row A* ⟹ *j* < *dim-col A* ⟹ *multcol j a A* $$ (*i*,*j*) = *a* ∗ *A* $$ (*i*,*j*)
i < *dim-row A* ⟹ *j* < *dim-col A* ⟹ *k* ≠ *j* ⟹ *multcol k a A* $$ (*i*,*j*) = *A* $$ (*i*,*j*)
dim-row (*multcol k a A*) = *dim-row A* *dim-col* (*multcol k a A*) = *dim-col A*
⟨*proof*⟩

lemma *index-mat-swapcols*[*simp*]:
i < *dim-row A* ⟹ *j* < *dim-col A* ⟹ *swapcols k l A* $$ (*i*,*j*) = (*if k* = *j then A* $$ (*i*,*l*) *else*
if l = *j then A* $$ (*i*,*k*) *else A* $$ (*i*,*j*))
dim-row (*swapcols k l A*) = *dim-row A* *dim-col* (*swapcols k l A*) = *dim-col A*
⟨*proof*⟩

lemma *index-mat-addcol*[*simp*]:
i < *dim-row A* ⟹ *j* < *dim-col A* ⟹ *addcol a k l A* $$ (*i*,*j*) = (*if k* = *j then*
a ∗ *A* $$ (*i*,*l*) + *A* $$ (*i*,*j*) *else A* $$ (*i*,*j*))
i < *dim-row A* ⟹ *j* < *dim-col A* ⟹ *addcol a j l A* $$ (*i*,*j*) = *a* ∗ *A* $$(*i*,*l*) + *A*$$(*i*,*j*)
i < *dim-row A* ⟹ *j* < *dim-col A* ⟹ *k* ≠ *j* ⟹ *addcol a k l A* $$ (*i*,*j*) = *A* $$(*i*,*j*)
dim-row (*addcol a k l A*) = *dim-row A* *dim-col* (*addcol a k l A*) = *dim-col A*
⟨*proof*⟩

Each column-operation can be seen as a multiplication of an elementary matrix from the right

lemma *col-addrow*:
l ≠ *i* ⟹ *i* < *n* ⟹ *col* (*addrow-mat n a k l*) *i* = *unit-vec n i*
k < *n* ⟹ *l* < *n* ⟹ *col* (*addrow-mat n a k l*) *l* = *a* $\cdot_v$ *unit-vec n k* + *unit-vec n l*
⟨*proof*⟩

lemma *col-addcol*[*simp*]:
k < *dim-col A* ⟹ *l* < *dim-col A* ⟹ *col* (*addcol a k l A*) *k* = *a* $\cdot_v$ *col A l* + *col A k*
⟨*proof*⟩

lemma *addcol-mat*: **assumes** *A*: *A* ∈ *carrier-mat nr n*
and *k*: *k* < *n*
shows *addcol* (*a* :: *'a* :: *comm-semiring-1*) *l k A* = *A* ∗ *addrow-mat n a k l*
⟨*proof*⟩

lemma *col-multrow*: $k \neq i \Longrightarrow i < n \Longrightarrow \mathit{col}\ (\mathit{multrow\text{-}mat}\ n\ k\ a)\ i = \mathit{unit\text{-}vec}\ n\ i$
$k < n \Longrightarrow \mathit{col}\ (\mathit{multrow\text{-}mat}\ n\ k\ a)\ k = a \cdot_v \mathit{unit\text{-}vec}\ n\ k$
⟨*proof*⟩

lemma *multcol-mat*: **assumes** A: $(A :: {}'a :: \mathit{comm\text{-}ring\text{-}1}\ \mathit{mat}) \in \mathit{carrier\text{-}mat}\ \mathit{nr}\ n$
shows $\mathit{multcol}\ k\ a\ A = A * \mathit{multrow\text{-}mat}\ n\ k\ a$
⟨*proof*⟩

lemma *col-swaprows*:
$l < n \Longrightarrow \mathit{col}\ (\mathit{swaprows\text{-}mat}\ n\ l\ l)\ l = \mathit{unit\text{-}vec}\ n\ l$
$i \neq k \Longrightarrow i \neq l \Longrightarrow i < n \Longrightarrow \mathit{col}\ (\mathit{swaprows\text{-}mat}\ n\ k\ l)\ i = \mathit{unit\text{-}vec}\ n\ i$
$k < n \Longrightarrow l < n \Longrightarrow \mathit{col}\ (\mathit{swaprows\text{-}mat}\ n\ k\ l)\ l = \mathit{unit\text{-}vec}\ n\ k$
$k < n \Longrightarrow l < n \Longrightarrow \mathit{col}\ (\mathit{swaprows\text{-}mat}\ n\ k\ l)\ k = \mathit{unit\text{-}vec}\ n\ l$
⟨*proof*⟩

lemma *swapcols-mat*: **assumes** A: $A \in \mathit{carrier\text{-}mat}\ \mathit{nr}\ n$ **and** k: $k < n\ l < n$
shows $\mathit{swapcols}\ k\ l\ A = A * \mathit{swaprows\text{-}mat}\ n\ k\ l$
⟨*proof*⟩

Combining row and column-operations yields similarity transformations.

definition *add-col-sub-row* $:: {}'a :: \mathit{ring\text{-}1} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow {}'a\ \mathit{mat} \Rightarrow {}'a\ \mathit{mat}$ **where**
$\mathit{add\text{-}col\text{-}sub\text{-}row}\ a\ k\ l\ A = \mathit{addrow}\ (-\ a)\ k\ l\ (\mathit{addcol}\ a\ l\ k\ A)$

definition *mult-col-div-row* $:: {}'a :: \mathit{field} \Rightarrow \mathit{nat} \Rightarrow {}'a\ \mathit{mat} \Rightarrow {}'a\ \mathit{mat}$ **where**
$\mathit{mult\text{-}col\text{-}div\text{-}row}\ a\ k\ A = \mathit{multrow}\ k\ (\mathit{inverse}\ a)\ (\mathit{multcol}\ k\ a\ A)$

definition *swap-cols-rows* $:: \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow {}'a\ \mathit{mat} \Rightarrow {}'a\ \mathit{mat}$ **where**
$\mathit{swap\text{-}cols\text{-}rows}\ k\ l\ A = \mathit{swaprows}\ k\ l\ (\mathit{swapcols}\ k\ l\ A)$

lemma *add-col-sub-row-carrier*[*simp*]:
$\mathit{dim\text{-}row}\ (\mathit{add\text{-}col\text{-}sub\text{-}row}\ a\ k\ l\ A) = \mathit{dim\text{-}row}\ A$
$\mathit{dim\text{-}col}\ (\mathit{add\text{-}col\text{-}sub\text{-}row}\ a\ k\ l\ A) = \mathit{dim\text{-}col}\ A$
$A \in \mathit{carrier\text{-}mat}\ n\ n \Longrightarrow \mathit{add\text{-}col\text{-}sub\text{-}row}\ a\ k\ l\ A \in \mathit{carrier\text{-}mat}\ n\ n$
⟨*proof*⟩

lemma *add-col-sub-index-row*[*simp*]:
$i < \mathit{dim\text{-}row}\ A \Longrightarrow i < \mathit{dim\text{-}col}\ A \Longrightarrow j < \mathit{dim\text{-}row}\ A \Longrightarrow j < \mathit{dim\text{-}col}\ A \Longrightarrow l < \mathit{dim\text{-}row}\ A$
$\Longrightarrow \mathit{add\text{-}col\text{-}sub\text{-}row}\ a\ k\ l\ A\ \$\$\ (i,j) = (\mathit{if}$
$i = k \wedge j = l\ \mathit{then}\ A\ \$\$\ (i,\ j) + a * A\ \$\$\ (i,\ i) - a * a * A\ \$\$\ (j,\ i) - a * A\ \$\$\ (j,\ j)\ \mathit{else\ if}$
$i = k \wedge j \neq l\ \mathit{then}\ A\ \$\$\ (i,\ j) - a * A\ \$\$\ (l,\ j)\ \mathit{else\ if}$
$i \neq k \wedge j = l\ \mathit{then}\ A\ \$\$\ (i,\ j) + a * A\ \$\$\ (i,\ k)\ \mathit{else}\ A\ \$\$\ (i,j))$
⟨*proof*⟩

lemma *mult-col-div-index-row*[*simp*]:

i < *dim-row A* ⟹ *i* < *dim-col A* ⟹ *j* < *dim-row A* ⟹ *j* < *dim-col A* ⟹ *a* ≠ *0*
⟹ *mult-col-div-row a k A* $$ (*i*,*j*) = (*if*
i = *k* ∧ *j* ≠ *i then inverse a* ∗ *A* $$ (*i*, *j*) *else if*
j = *k* ∧ *j* ≠ *i then a* ∗ *A* $$ (*i*, *j*) *else A* $$ (*i*,*j*))
⟨*proof*⟩

lemma *mult-col-div-row-carrier*[*simp*]:
dim-row (*mult-col-div-row a k A*) = *dim-row A*
dim-col (*mult-col-div-row a k A*) = *dim-col A*
A ∈ *carrier-mat n n* ⟹ *mult-col-div-row a k A* ∈ *carrier-mat n n*
⟨*proof*⟩

lemma *swap-cols-rows-carrier*[*simp*]:
dim-row (*swap-cols-rows k l A*) = *dim-row A*
dim-col (*swap-cols-rows k l A*) = *dim-col A*
A ∈ *carrier-mat n n* ⟹ *swap-cols-rows k l A* ∈ *carrier-mat n n*
⟨*proof*⟩

lemma *swap-cols-rows-index*[*simp*]:
i < *dim-row A* ⟹ *i* < *dim-col A* ⟹ *j* < *dim-row A* ⟹ *j* < *dim-col A* ⟹ *a* < *dim-row A* ⟹ *b* < *dim-row A*
⟹ *swap-cols-rows a b A* $$ (*i*,*j*) = *A* $$ (*if i* = *a then b else if i* = *b then a else i*,
if j = *a then b else if j* = *b then a else j*)
⟨*proof*⟩

lemma *add-col-sub-row-similar*: **assumes** *A*: *A* ∈ *carrier-mat n n* **and** *kl*: *k* < *n* *l* < *n* *k* ≠ *l*
shows *similar-mat* (*add-col-sub-row a k l A*) (*A* :: *'a* :: *comm-ring-1 mat*)
⟨*proof*⟩

lemma *mult-col-div-row-similar*: **assumes** *A*: *A* ∈ *carrier-mat n n* **and** *ak*: *k* < *n* *a* ≠ *0*
shows *similar-mat* (*mult-col-div-row a k A*) *A*
⟨*proof*⟩

lemma *swap-cols-rows-similar*: **assumes** *A*: *A* ∈ *carrier-mat n n* **and** *kl*: *k* < *n* *l* < *n*
shows *similar-mat* (*swap-cols-rows k l A*) *A*
⟨*proof*⟩

lemma *swapcols-carrier*[*simp*]: (*swapcols l k A* ∈ *carrier-mat n m*) = (*A* ∈ *carrier-mat n m*)
⟨*proof*⟩

fun *swap-row-to-front* :: *'a mat* ⇒ *nat* ⇒ *'a mat* **where**

swap-row-to-front A $0 = A$
| *swap-row-to-front* A (*Suc* I) $=$ *swap-row-to-front* (*swaprows* I (*Suc* I) A) I

fun *swap-col-to-front* :: *'a mat* $\Rightarrow$ *nat* $\Rightarrow$ *'a mat* **where**
swap-col-to-front A $0 = A$
| *swap-col-to-front* A (*Suc* I) $=$ *swap-col-to-front* (*swapcols* I (*Suc* I) A) I

lemma *swap-row-to-front-result*: $A \in$ *carrier-mat* n $m \Longrightarrow I < n \Longrightarrow$ *swap-row-to-front* A $I =$
mat n m (λ (i,j). *if* $i = 0$ *then* A \$\$ (I,j)
else if $i \leq I$ *then* A \$\$ $(i - 1, j)$ *else* A \$\$ (i,j))
⟨*proof*⟩

lemma *swap-col-to-front-result*: $A \in$ *carrier-mat* n $m \Longrightarrow J < m \Longrightarrow$ *swap-col-to-front* A $J =$
mat n m (λ (i,j). *if* $j = 0$ *then* A \$\$ (i,J)
else if $j \leq J$ *then* A \$\$ $(i, j-1)$ *else* A \$\$ (i,j))
⟨*proof*⟩

lemma *swapcols-is-transp-swap-rows*: **assumes** A: $A \in$ *carrier-mat* n m $k < m$ $l < m$
shows *swapcols* k l $A =$ *transpose-mat* (*swaprows* k l (*transpose-mat* A))
⟨*proof*⟩

end

9 Determinants

Most of the following definitions and proofs on determinants have been copied and adapted from /src/HOL/Multivariate-Analysis/Determinants.thy.

Exceptions are *det-identical-rows*.

We further generalized some lemmas, e.g., that the determinant is 0 iff the kernel of a matrix is non-empty is available for integral domains, not just for fields.

theory *Determinant*
imports
Missing-Misc
Column-Operations
HOL−Computational-Algebra.Polynomial-Factorial
Polynomial-Interpolation.Ring-Hom
Polynomial-Interpolation.Missing-Unsorted
begin

definition *det*:: *'a mat* $\Rightarrow$ *'a* :: *comm-ring-1* **where**

det A = (*if dim-row A* = *dim-col A then* (∑ *p* ∈ {*p*. *p permutes* {*0* ..< *dim-row A*}}.
signof p ∗ (∏ *i* = *0* ..< *dim-row A*. *A* $$ (*i*, *p i*))) *else 0*)

lemma(**in** *ring-hom*) *hom-signof*[*simp*]: *hom* (*signof p*) = *signof p*
⟨*proof*⟩

lemma(**in** *comm-ring-hom*) *hom-det*[*simp*]: *det* (*map-mat hom A*) = *hom* (*det A*)
⟨*proof*⟩

lemma *det-def'*: *A* ∈ *carrier-mat n n* ⟹
det A = (∑ *p* ∈ {*p*. *p permutes* {*0* ..< *n*}}.
signof p ∗ (∏ *i* = *0* ..< *n*. *A* $$ (*i*, *p i*))) ⟨*proof*⟩

lemma *det-smult*[*simp*]: *det* (*a* $\cdot_m$ *A*) = *a* ^ *dim-col A* ∗ *det A*
⟨*proof*⟩

lemma *det-transpose*: **assumes** *A*: *A* ∈ *carrier-mat n n*
shows *det* (*transpose-mat A*) = *det A*
⟨*proof*⟩

lemma *det-col*:
assumes *A*: *A* ∈ *carrier-mat n n*
shows *det A* = (∑ *p* | *p permutes* {*0* ..< *n*}. *signof p* ∗ (∏*j*<*n*. *A* $$ (*p j*, *j*)))
(**is** - = (*sum* (λ*p*. - ∗ *?prod p*) *?P*))
⟨*proof*⟩

lemma *mat-det-left-def*: **assumes** *A*: *A* ∈ *carrier-mat n n*
shows *det A* = (∑*p*∈{*p*. *p permutes* {*0*..<*dim-row A*}}. *signof p* ∗ (∏*i* = *0* ..< *dim-row A*. *A* $$ (*p i*, *i*)))
⟨*proof*⟩

lemma *det-upper-triangular*:
assumes *ut*: *upper-triangular A*
and *m*: *A* ∈ *carrier-mat n n*
shows *det A* = *prod-list* (*diag-mat A*)
⟨*proof*⟩

lemma *det-single*: **assumes** *A* ∈ *carrier-mat 1 1*
shows *det A* = *A* $$ (*0*,*0*)
⟨*proof*⟩

lemma *det-one*[*simp*]: *det* (1_m *n*) = *1*
⟨*proof*⟩

lemma *det-zero*[*simp*]: **assumes** *n* > *0* **shows** *det* (0_m *n n*) = *0*
⟨*proof*⟩

lemma *det-dim-zero*[*simp*]: *A* ∈ *carrier-mat 0 0* ⟹ *det A* = *1*

⟨*proof*⟩

lemma *det-lower-triangular*:
assumes *ld*: $\bigwedge i\ j.\ i < j \Longrightarrow j < n \Longrightarrow A\ \$\$\ (i,j) = 0$
and *m*: $A \in \mathit{carrier\text{-}mat}\ n\ n$
shows $\mathit{det}\ A = \mathit{prod\text{-}list}\ (\mathit{diag\text{-}mat}\ A)$
⟨*proof*⟩

lemma *det-permute-rows*: **assumes** *A*: $A \in \mathit{carrier\text{-}mat}\ n\ n$
and *p*: $p\ \mathit{permutes}\ \{0\ ..{<}\ (n :: \mathit{nat})\}$
shows $\mathit{det}\ (\mathit{mat}\ n\ n\ (\lambda\ (i,j).\ A\ \$\$\ (p\ i,\ j))) = \mathit{signof}\ p * \mathit{det}\ A$
⟨*proof*⟩

lemma *det-multrow-mat*: **assumes** *k*: $k < n$
shows $\mathit{det}\ (\mathit{multrow\text{-}mat}\ n\ k\ a) = a$
⟨*proof*⟩

lemma *swap-rows-mat-eq-permute*:
$k < n \Longrightarrow l < n \Longrightarrow \mathit{swaprows\text{-}mat}\ n\ k\ l = \mathit{mat}\ n\ n\ (\lambda(i,\ j).\ 1_m\ n\ \$\$\ (\mathit{transpose}\ k\ l\ i,\ j))$
⟨*proof*⟩

lemma *det-swaprows-mat*: **assumes** *k*: $k < n$ **and** *l*: $l < n$ **and** *kl*: $k \neq l$
shows $\mathit{det}\ (\mathit{swaprows\text{-}mat}\ n\ k\ l) = -\ 1$
⟨*proof*⟩

lemma *det-addrow-mat*:
assumes *l*: $k \neq l$
shows $\mathit{det}\ (\mathit{addrow\text{-}mat}\ n\ a\ k\ l) = 1$
⟨*proof*⟩

The following proof is new, as it does not use $2 \neq 0$ as in Multivariate-Analysis.

lemma *det-identical-rows*:
assumes *A*: $A \in \mathit{carrier\text{-}mat}\ n\ n$
and *ij*: $i \neq j$
and *i*: $i < n$ **and** *j*: $j < n$
and *r*: $\mathit{row}\ A\ i = \mathit{row}\ A\ j$
shows $\mathit{det}\ A = 0$
⟨*proof*⟩

lemma *det-row-0*: **assumes** *k*: $k < n$
and *c*: $c \in \{0\ ..{<}\ n\} \rightarrow \mathit{carrier\text{-}vec}\ n$
shows $\mathit{det}\ (\mathit{mat}_r\ n\ n\ (\lambda i.\ \mathit{if}\ i = k\ \mathit{then}\ 0_v\ n\ \mathit{else}\ c\ i)) = 0$
⟨*proof*⟩

lemma *det-row-add*:
assumes *abc*: $a\ k \in \mathit{carrier\text{-}vec}\ n\ b\ k \in \mathit{carrier\text{-}vec}\ n\ c \in \{0..{<}n\} \rightarrow \mathit{carrier\text{-}vec}\ n$

and k: $k < n$
shows $\mathit{det}(\mathit{mat}_r\ n\ n\ (\lambda\ i.\ \mathit{if}\ i = k\ \mathit{then}\ a\ i + b\ i\ \mathit{else}\ c\ i)) =$
$\mathit{det}(\mathit{mat}_r\ n\ n\ (\lambda\ i.\ \mathit{if}\ i = k\ \mathit{then}\ a\ i\ \mathit{else}\ c\ i)) +$
$\mathit{det}(\mathit{mat}_r\ n\ n\ (\lambda\ i.\ \mathit{if}\ i = k\ \mathit{then}\ b\ i\ \mathit{else}\ c\ i))$
(**is** $?\mathit{lhs} = ?\mathit{rhs}$)
⟨*proof*⟩

lemma *det-linear-row-finsum*:
assumes fS: $\mathit{finite}\ S$ **and** c: $c \in \{0..{<}n\} \to \mathit{carrier\text{-}vec}\ n$ **and** k: $k < n$
and a: $a\ k \in S \to \mathit{carrier\text{-}vec}\ n$
shows $\mathit{det}\ (\mathit{mat}_r\ n\ n\ (\lambda\ i.\ \mathit{if}\ i = k\ \mathit{then}\ \mathit{finsum\text{-}vec}\ \mathit{TYPE}('a :: \mathit{comm\text{-}ring\text{-}1})\ n$ $(a\ i)\ S\ \mathit{else}\ c\ i)) =$
$\mathit{sum}\ (\lambda j.\ \mathit{det}\ (\mathit{mat}_r\ n\ n\ (\lambda\ i.\ \mathit{if}\ i = k\ \mathit{then}\ a\ \ i\ j\ \mathit{else}\ c\ i)))\ S$
⟨*proof*⟩

lemma *det-linear-rows-finsum-lemma*:
assumes fS: $\mathit{finite}\ S$
and fT: $\mathit{finite}\ T$ **and** c: $c \in \{0..{<}n\} \to \mathit{carrier\text{-}vec}\ n$
and T: $T \subseteq \{0\ ..{<}\ n\}$
and a: $a \in T \to S \to \mathit{carrier\text{-}vec}\ n$
shows $\mathit{det}\ (\mathit{mat}_r\ n\ n\ (\lambda\ i.\ \mathit{if}\ i \in T\ \mathit{then}\ \mathit{finsum\text{-}vec}\ \mathit{TYPE}('a :: \mathit{comm\text{-}ring\text{-}1})\ n$ $(a\ i)\ S\ \mathit{else}\ c\ i)) =$
$\mathit{sum}\ (\lambda f.\ \mathit{det}(\mathit{mat}_r\ n\ n\ (\lambda\ i.\ \mathit{if}\ i \in T\ \mathit{then}\ a\ i\ (f\ i)\ \mathit{else}\ c\ i)))$
$\{f.\ (\forall\, i \in T.\ f\ i \in S) \wedge (\forall\, i.\ i \notin T \longrightarrow f\ i = i)\}$
⟨*proof*⟩

lemma *det-linear-rows-sum*:
assumes fS: $\mathit{finite}\ S$
and a: $a \in \{0..{<}n\} \to S \to \mathit{carrier\text{-}vec}\ n$
shows $\mathit{det}\ (\mathit{mat}_r\ n\ n\ (\lambda\ i.\ \mathit{finsum\text{-}vec}\ \mathit{TYPE}('a :: \mathit{comm\text{-}ring\text{-}1})\ n\ (a\ i)\ S)) =$
$\mathit{sum}\ (\lambda f.\ \mathit{det}\ (\mathit{mat}_r\ n\ n\ (\lambda\ i.\ a\ i\ (f\ i))))$
$\{f.\ (\forall\, i{\in}\{0..{<}n\}.\ f\ i \in S) \wedge (\forall\, i.\ i \notin \{0..{<}n\} \longrightarrow f\ i = i)\}$
⟨*proof*⟩

lemma *det-rows-mul*:
assumes a: $a \in \{0..{<}n\} \to \mathit{carrier\text{-}vec}\ n$
shows $\mathit{det}(\mathit{mat}_r\ n\ n\ (\lambda\ i.\ c\ i \cdot_v\ a\ i)) =$
$\mathit{prod}\ c\ \{0..{<}n\} * \mathit{det}(\mathit{mat}_r\ n\ n\ (\lambda\ i.\ a\ i))$
⟨*proof*⟩

lemma *mat-mul-finsum-alt*:
assumes A: $A \in \mathit{carrier\text{-}mat}\ \mathit{nr}\ n$ **and** B: $B \in \mathit{carrier\text{-}mat}\ n\ \mathit{nc}$
shows $A * B = \mathit{mat}_r\ \mathit{nr}\ \mathit{nc}\ (\lambda\ i.\ \mathit{finsum\text{-}vec}\ \mathit{TYPE}('a :: \mathit{semiring\text{-}0})\ \mathit{nc}\ (\lambda k.\ A$ $\$\$\ (i,k) \cdot_v\ \mathit{row}\ B\ k)\ \{0\ ..{<}\ n\})$
⟨*proof*⟩

lemma *det-mult*:
assumes *A*: $A \in \mathit{carrier\text{-}mat}\ n\ n$ **and** *B*: $B \in \mathit{carrier\text{-}mat}\ n\ n$
shows $\mathit{det}\ (A * B) = \mathit{det}\ A * \mathit{det}\ (B :: {}'a :: \mathit{comm\text{-}ring\text{-}1}\ \mathit{mat})$
⟨*proof*⟩

lemma *unit-imp-det-non-zero*: **assumes** $A \in \mathit{Units}\ (\mathit{ring\text{-}mat}\ \mathit{TYPE}({}'a :: \mathit{comm\text{-}ring\text{-}1})$
$n\ b)$
shows $\mathit{det}\ A \neq 0$
⟨*proof*⟩

The following proof is based on the Gauss-Jordan algorithm.

lemma *det-non-zero-imp-unit*: **assumes** *A*: $A \in \mathit{carrier\text{-}mat}\ n\ n$
and *dA*: $\mathit{det}\ A \neq (0 :: {}'a :: \mathit{field})$
shows $A \in \mathit{Units}\ (\mathit{ring\text{-}mat}\ \mathit{TYPE}({}'a)\ n\ b)$
⟨*proof*⟩

lemma *mat-mult-left-right-inverse*: **assumes** *A*: $(A :: {}'a :: \mathit{field}\ \mathit{mat}) \in \mathit{carrier\text{-}mat}$
$n\ n$
and *B*: $B \in \mathit{carrier\text{-}mat}\ n\ n$ **and** *AB*: $A * B = 1_m\ n$
shows $B * A = 1_m\ n$
⟨*proof*⟩

lemma *det-zero-imp-zero-row*: **assumes** *A*: $(A :: {}'a :: \mathit{field}\ \mathit{mat}) \in \mathit{carrier\text{-}mat}\ n$
n
and *det*: $\mathit{det}\ A = 0$
shows $\exists\ P.\ P \in \mathit{Units}\ (\mathit{ring\text{-}mat}\ \mathit{TYPE}({}'a)\ n\ b) \wedge \mathit{row}\ (P * A)\ (n - 1) = 0_v$
$n \wedge 0 < n$
$\wedge\ \mathit{row\text{-}echelon\text{-}form}\ (P * A)$
⟨*proof*⟩

lemma *det-0-iff-vec-prod-zero-field*: **assumes** *A*: $(A :: {}'a :: \mathit{field}\ \mathit{mat}) \in \mathit{carrier\text{-}mat}$
$n\ n$
shows $\mathit{det}\ A = 0 \longleftrightarrow (\exists\ v.\ v \in \mathit{carrier\text{-}vec}\ n \wedge v \neq 0_v\ n \wedge A *_v v = 0_v\ n)$ (**is**
$?l = (\exists\ v.\ ?P\ v))$
⟨*proof*⟩

In order to get the result for integral domains, we embed the domain in its fraction field, and then apply the result for fields.

lemma *det-0-iff-vec-prod-zero*: **assumes** *A*: $(A :: {}'a :: \mathit{idom}\ \mathit{mat}) \in \mathit{carrier\text{-}mat}\ n$
n
shows $\mathit{det}\ A = 0 \longleftrightarrow (\exists\ v.\ v \in \mathit{carrier\text{-}vec}\ n \wedge v \neq 0_v\ n \wedge A *_v v = 0_v\ n)$
⟨*proof*⟩

lemma *det-0-negate*: **assumes** *A*: $(A :: {}'a :: \mathit{field}\ \mathit{mat}) \in \mathit{carrier\text{-}mat}\ n\ n$
shows $(\mathit{det}\ (-\ A) = 0) = (\mathit{det}\ A = 0)$
⟨*proof*⟩

lemma *det-multrow*:

assumes *k*: *k* < *n* **and** *A*: *A* ∈ *carrier-mat n n*
shows *det* (*multrow k a A*) = *a* ∗ *det A*
⟨*proof*⟩

lemma *det-multrow-div*:
assumes *k*: *k* < *n* **and** *A*: *A* ∈ *carrier-mat n n* **and** *a0*: *a* ≠ *0*
shows *det* (*multrow k a A* :: *'a* :: *idom-divide mat*) *div a* = *det A*
⟨*proof*⟩

lemma *det-addrow*:
assumes *l*: *l* < *n* **and** *k*: *k* ≠ *l* **and** *A*: *A* ∈ *carrier-mat n n*
shows *det* (*addrow a k l A*) = *det A*
⟨*proof*⟩

lemma *det-swaprows*:
assumes ∗: *k* < *n l* < *n* **and** *k*: *k* ≠ *l* **and** *A*: *A* ∈ *carrier-mat n n*
shows *det* (*swaprows k l A*) = − *det A*
⟨*proof*⟩

lemma *det-similar*: **assumes** *similar-mat A B*
shows *det A* = *det B*
⟨*proof*⟩

lemma *det-four-block-mat-upper-right-zero-col*: **assumes** *A1*: *A1* ∈ *carrier-mat n n*
and *A20*: *A2* = (0_m *n 1*) **and** *A3*: *A3* ∈ *carrier-mat 1 n*
and *A4*: *A4* ∈ *carrier-mat 1 1*
shows *det* (*four-block-mat A1 A2 A3 A4*) = *det A1* ∗ *det A4* (**is** *det ?A* = -)
⟨*proof*⟩

lemma *det-swap-initial-rows*: **assumes** *A*: *A* ∈ *carrier-mat m m*
and *lt*: *k* + *n* ≤ *m*
shows *det A* = (− *1*) ^ (*k* ∗ *n*) ∗
det (*mat m m* (λ(*i*, *j*). *A* $$ (*if i* < *n then i* + *k else if i* < *k* + *n then i* − *n else i*, *j*)))
⟨*proof*⟩

lemma *det-swap-rows*: **assumes** *A*: *A* ∈ *carrier-mat* (*k* + *n*) (*k* + *n*)
shows *det A* = (−*1*)^(*k* ∗ *n*) ∗ *det* (*mat* (*k* + *n*) (*k* + *n*) (λ (*i*,*j*).
A $$ ((*if i* < *k then i* + *n else i* − *k*),*j*)))
⟨*proof*⟩

lemma *det-swap-final-rows*: **assumes** *A*: *A* ∈ *carrier-mat m m*
and *m*: *m* = *l* + *k* + *n*
shows *det A* = (− *1*) ^ (*k* ∗ *n*) ∗
det (*mat m m* (λ(*i*, *j*). *A* $$ (*if i* < *l then i else if i* < *l* + *n then i* + *k else i* − *n*, *j*)))
(**is** - = - ∗ *det ?M*)
⟨*proof*⟩

lemma *det-swap-final-cols*: **assumes** *A*: $A \in$ *carrier-mat m m*
  and *m*: $m = l + k + n$
  shows $\mathit{det}\ A = (-\ 1)\ \hat{}\ (k * n) *$
    $\mathit{det}\ (\mathit{mat}\ m\ m\ (\lambda(i,\ j).\ A\ \$\$\ (i,\ \mathit{if}\ j < l\ \mathit{then}\ j\ \mathit{else}\ \mathit{if}\ j < l + n\ \mathit{then}\ j + k\ \mathit{else}$
$j - n)))$
⟨*proof*⟩

lemma *det-swap-initial-cols*: **assumes** *A*: $A \in$ *carrier-mat m m*
  and *lt*: $k + n \le m$
  shows $\mathit{det}\ A = (-\ 1)\ \hat{}\ (k * n) *$
    $\mathit{det}\ (\mathit{mat}\ m\ m\ (\lambda(i,\ j).\ A\ \$\$\ (i,\ \mathit{if}\ j < n\ \mathit{then}\ j + k\ \mathit{else}\ \mathit{if}\ j < k + n\ \mathit{then}\ j -$
$n\ \mathit{else}\ j)))$
⟨*proof*⟩

lemma *det-swap-cols*: **assumes** *A*: $A \in$ *carrier-mat* $(k + n)\ (k + n)$
  shows $\mathit{det}\ A = (-1)\hat{}(k * n) * \mathit{det}\ (\mathit{mat}\ (k + n)\ (k + n)\ (\lambda\ (i,j).$
   $A\ \$\$\ (i,(\mathit{if}\ j < k\ \mathit{then}\ j + n\ \mathit{else}\ j - k))))$ (**is** - = - ∗ *det ?B*)
⟨*proof*⟩

lemma *det-four-block-mat-upper-right-zero*: **fixes** *A1* :: *'a* :: *idom mat*
  assumes *A1*: $A1 \in$ *carrier-mat n n*
  and *A20*: $A2 = (0_m\ n\ m)$ **and** *A3*: $A3 \in$ *carrier-mat m n*
  and *A4*: $A4 \in$ *carrier-mat m m*
shows *det* (*four-block-mat A1 A2 A3 A4*) = *det A1* ∗ *det A4*
  ⟨*proof*⟩

lemma *det-four-block-mat-lower-left-zero*: **fixes** *A1* :: *'a* :: *idom mat*
  assumes *A1*: $A1 \in$ *carrier-mat n n*
  and *A2*: $A2 \in$ *carrier-mat n m* **and** *A30*: $A3 = 0_m\ m\ n$
  and *A4*: $A4 \in$ *carrier-mat m m*
shows *det* (*four-block-mat A1 A2 A3 A4*) = *det A1* ∗ *det A4*
⟨*proof*⟩

context
begin
private lemma *det-four-block-mat-preliminary*: **assumes** *A*: (*A* :: *'a* :: *idom mat*)
∈ *carrier-mat n n*
  and *B*: $B \in$ *carrier-mat n n*
  and *C*: $C \in$ *carrier-mat n n*
  and *D*: $D \in$ *carrier-mat n n*
  and *commute*: $C * D = D * C$
  and *detD*: $\mathit{det}\ D \neq 0$
shows *det* (*four-block-mat A B C D*) = $\mathit{det}\ (A * D - B * C)$
⟨*proof*⟩

lemma *det-four-block-mat*: **assumes** *A*: (*A* :: *'a* :: *idom mat*) ∈ *carrier-mat n n*
  and *B*: $B \in$ *carrier-mat n n*
  and *C*: $C \in$ *carrier-mat n n*

  and *D*: *D* ∈ *carrier-mat n n*
  and *commute*: *C* ∗ *D* = *D* ∗ *C*
shows *det* (*four-block-mat A B C D*) = *det* (*A* ∗ *D* − *B* ∗ *C*)
⟨*proof*⟩
end

lemma *det-swapcols*:
  assumes ∗: *k* < *n l* < *n k* ≠ *l* **and** *A*: *A* ∈ *carrier-mat n n*
  shows *det* (*swapcols k l A*) = − *det A*
⟨*proof*⟩

lemma *swap-row-to-front-det*: *A* ∈ *carrier-mat n n* ⟹ *I* < *n* ⟹ *det* (*swap-row-to-front A I*)
  = (−*1*)^*I* ∗ *det A*
⟨*proof*⟩

lemma *swap-col-to-front-det*: *A* ∈ *carrier-mat n n* ⟹ *I* < *n* ⟹ *det* (*swap-col-to-front A I*)
  = (−*1*)^*I* ∗ *det A*
⟨*proof*⟩

lemma *swap-row-to-front-four-block*: **assumes** *A1*: *A1* ∈ *carrier-mat n m1*
  and *A2*: *A2* ∈ *carrier-mat n m2*
  and *A3*: *A3* ∈ *carrier-mat 1 m1*
  and *A4*: *A4* ∈ *carrier-mat 1 m2*
  shows *swap-row-to-front* (*four-block-mat A1 A2 A3 A4*) *n* = *four-block-mat A3 A4 A1 A2*
  ⟨*proof*⟩

lemma *swap-col-to-front-four-block*: **assumes** *A1*: *A1* ∈ *carrier-mat n1 m*
  and *A2*: *A2* ∈ *carrier-mat n1 1*
  and *A3*: *A3* ∈ *carrier-mat n2 m*
  and *A4*: *A4* ∈ *carrier-mat n2 1*
  shows *swap-col-to-front* (*four-block-mat A1 A2 A3 A4*) *m* = *four-block-mat A2 A1 A4 A3*
  ⟨*proof*⟩

lemma *det-four-block-mat-lower-right-zero-col*: **assumes** *A1*: *A1* ∈ *carrier-mat 1 n*
  and *A2*: *A2* ∈ *carrier-mat 1 1*
  and *A3*: *A3* ∈ *carrier-mat n n*
  and *A40*: *A4* = (0_m *n 1*)
  shows *det* (*four-block-mat A1 A2 A3 A4*) = (−*1*)^*n* ∗ *det A2* ∗ *det A3* (**is** *det ?A* = -)
⟨*proof*⟩

lemma *det-four-block-mat-lower-left-zero-col*: **assumes** *A1*: *A1* ∈ *carrier-mat 1 1*
and *A2*: *A2* ∈ *carrier-mat 1 n*
and *A30*: *A3* = (0_m *n 1*)
and *A4*: *A4* ∈ *carrier-mat n n*
shows *det* (*four-block-mat A1 A2 A3 A4*) = *det A1* ∗ *det A4* (**is** *det ?A* = -)
⟨*proof*⟩

lemma *det-addcol*[*simp*]:
assumes *l*: *l* < *n* **and** *k*: *k* ≠ *l* **and** *A*: *A* ∈ *carrier-mat n n*
shows *det* (*addcol a k l A*) = *det A*
⟨*proof*⟩

definition *insert-index i* ≡ λ*i'*. *if i'* < *i then i' else Suc i'*

definition *delete-index i* ≡ λ*i'*. *if i'* < *i then i' else i'* − *Suc 0*

lemma *insert-index*[*simp*]:
i' < *i* ⟹ *insert-index i i'* = *i'*
i' ≥ *i* ⟹ *insert-index i i'* = *Suc i'*
⟨*proof*⟩

lemma *delete-insert-index*[*simp*]:
delete-index i (*insert-index i i'*) = *i'*
⟨*proof*⟩

lemma *insert-delete-index*:
assumes *i'i*: *i'* ≠ *i*
shows *insert-index i* (*delete-index i i'*) = *i'*
⟨*proof*⟩

definition *delete-dom p i* ≡ λ*i'*. *p* (*insert-index i i'*)

definition *delete-ran p j* ≡ λ*i*. *delete-index j* (*p i*)

definition *permutation-delete p i* = *delete-ran* (*delete-dom p i*) (*p i*)

definition *insert-ran p j* ≡ λ*i*. *insert-index j* (*p i*)

definition *insert-dom p i j* ≡
λ*i'*. *if i'* < *i then p i' else if i'* = *i then j else p* (*i'*−*1*)

definition *permutation-insert i j p* ≡ *insert-dom* (*insert-ran p j*) *i j*

lemmas *permutation-delete-expand* =
permutation-delete-def[*unfolded delete-dom-def delete-ran-def insert-index-def delete-index-def*]

lemmas *permutation-insert-expand* =
permutation-insert-def[*unfolded insert-dom-def insert-ran-def insert-index-def delete-index-def*]

lemma *permutation-insert-inserted*[*simp*]:
permutation-insert (*i*::*nat*) *j p i* = *j*
⟨*proof*⟩

lemma *permutation-insert-base*:
assumes *p*: *p permutes* {*0*..<*n*}
shows *permutation-insert n n p* = *p*
⟨*proof*⟩

lemma *permutation-insert-row-step*:
‹*permutation-insert* (*Suc i*) *j p* ∘ *transpose i* (*Suc i*) = *permutation-insert i j p*›
(**is** ‹*?l* = *?r*›)
⟨*proof*⟩

lemma *permutation-insert-column-step*:
assumes *p*: *p permutes* {*0*..<*n*} **and** *j* < *n*
shows *transpose j* (*Suc j*) ∘ *permutation-insert i* (*Suc j*) *p* = *permutation-insert i j p*
(**is** *?l* = *?r*)
⟨*proof*⟩

lemma *delete-dom-image*:
assumes *i*: *i* ∈ {*0*..<*Suc n*} (**is** - ∈ *?N*)
assumes *iff*: ∀ *i'* ∈ *?N*. *f i'* = *f i* ⟶ *i'* = *i*
shows *delete-dom f i* ‘ {*0*..<*n*} = *f* ‘ *?N* − {*f i*} (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *delete-ran-image*:
assumes *j*: *j* ∈ {*0*..<*Suc n*} (**is** - ∈ *?N*)
assumes *fimg*: *f* ‘ {*0*..<*n*} = *?N* − {*j*}
shows *delete-ran f j* ‘ {*0*..<*n*} = {*0*..<*n*} (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *delete-index-inj-on*:
assumes *iS*: *i* ∉ *S*
shows *inj-on* (*delete-index i*) *S*
⟨*proof*⟩

lemma *insert-index-inj-on*:
shows *inj-on* (*insert-index i*) *S*
⟨*proof*⟩

lemma *delete-dom-inj-on*:
assumes *i*: *i* ∈ {*0*..<*Suc n*} (**is** - ∈ *?N*)
assumes *inj*: *inj-on f ?N*
shows *inj-on* (*delete-dom f i*) {*0*..<*n*}
⟨*proof*⟩

lemma *delete-ran-inj-on*:
  assumes *j*: *j* ∈ {*0..<Suc n*} (**is** - ∈ *?N*)
  assumes *img*: *f* ‘ {*0..<n*} = *?N* − {*j*}
  shows *inj-on* (*delete-ran f j*) {*0..<n*}
  ⟨*proof*⟩

lemma *permutation-delete-bij-betw*:
  assumes *i*: *i* ∈ {*0 ..< Suc n*} (**is** - ∈ *?N*)
  assumes *bij*: *bij-betw p ?N ?N*
  shows *bij-betw* (*permutation-delete p i*) {*0..<n*} {*0..<n*} (**is** *bij-betw ?p* - -)
⟨*proof*⟩

lemma *permutation-delete-permutes*:
  assumes *p*: *p permutes* {*0 ..< Suc n*} (**is** - *permutes ?N*)
    and *i*: *i* < *Suc n*
  shows *permutation-delete p i permutes* {*0..<n*} (**is** *?p permutes ?N'*)
⟨*proof*⟩

lemma *permutation-insert-delete*:
  assumes *p*: *p permutes* {*0..<Suc n*}
    and *i*: *i* < *Suc n*
  shows *permutation-insert i* (*p i*) (*permutation-delete p i*) = *p*
   (**is** *?l* = -)
⟨*proof*⟩

lemma *insert-index-exclude*[*simp*]:
  insert-index i i' ≠ *i* ⟨*proof*⟩

lemma *insert-index-image*:
  assumes *i*: *i* < *Suc n*
  shows *insert-index i* ‘ {*0..<n*} = {*0..<Suc n*} − {*i*} (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *insert-ran-image*:
  assumes *j*: *j* < *Suc n*
  assumes *img*: *f* ‘ {*0..<n*} = {*0..<n*}
  shows *insert-ran f j* ‘ {*0..<n*} = {*0..<Suc n*} − {*j*} (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *insert-dom-image*:
  assumes *i*: *i* < *Suc n* **and** *j*: *j* < *Suc n*
   and *img*: *f* ‘ {*0..<n*} = {*0..<Suc n*} − {*j*} (**is** - = *?N* − -)
  shows *insert-dom f i j* ‘ *?N* = *?N* (**is** *?f* ‘ - = -)
⟨*proof*⟩

lemma *insert-ran-inj-on*:
  assumes *inj*: *inj-on f* {*0..<n*} **and** *j*: *j* < *Suc n*
  shows *inj-on* (*insert-ran f j*) {*0..<n*} (**is** *inj-on ?f* -)

⟨*proof*⟩

lemma *insert-dom-inj-on*:
assumes *inj*: *inj-on f* $\{0..<n\}$
and *i*: $i < Suc\ n$ **and** *j*: $j < Suc\ n$
and *img*: f ‘ $\{0..<n\} = \{0..<Suc\ n\} - \{j\}$ (**is** - = *?N* − -)
shows *inj-on* (*insert-dom f i j*) *?N*
⟨*proof*⟩

lemma *permutation-insert-bij-betw*:
assumes *q*: *q permutes* $\{0..<n\}$ **and** *i*: $i < Suc\ n$ **and** *j*: $j < Suc\ n$
shows *bij-betw* (*permutation-insert i j q*) $\{0..<Suc\ n\}$ $\{0..<Suc\ n\}$
(**is** *bij-betw ?q ?N* -)
⟨*proof*⟩

lemma *permutation-insert-permutes*:
assumes *q*: *q permutes* $\{0..<n\}$
and *i*: $i < Suc\ n$ **and** *j*: $j < Suc\ n$
shows *permutation-insert i j q permutes* $\{0..<Suc\ n\}$ (**is** *?p permutes ?N*)
⟨*proof*⟩

lemma *permutation-fix*:
assumes *i*: $i < Suc\ n$ **and** *j*: $j < Suc\ n$
shows $\{$ *p*. *p permutes* $\{0..<Suc\ n\} \wedge p\ i = j\ \} =$
permutation-insert i j ‘ $\{$ *q*. *q permutes* $\{0..<n\}\ \}$
(**is** *?L* = *?R*)
⟨*proof*⟩

lemma *permutation-split-ran*:
assumes *j*: $j \in S$
shows $\{$ *p*. *p permutes S* $\} = (\bigcup i \in S.\ \{$ *p*. *p permutes S* $\wedge\ p\ i = j\ \})$
(**is** *?L* = *?R*)
⟨*proof*⟩

lemma *permutation-disjoint-dom*:
assumes *i*: $i \in S$ **and** *i'*: $i' \in S$ **and** *j*: $j \in S$ **and** *ii'*: $i \neq i'$
shows $\{$ *p*. *p permutes S* $\wedge\ p\ i = j\ \} \cap \{$ *p*. *p permutes S* $\wedge\ p\ i' = j\ \} = \{\}$
(**is** *?L* ∩ *?R* = {})
⟨*proof*⟩

lemma *permutation-disjoint-ran*:
assumes *i*: $i \in S$ **and** *j*: $j \in S$ **and** *j'*: $j' \in S$ **and** *jj'*: $j \neq j'$
shows $\{$ *p*. *p permutes S* $\wedge\ p\ i = j\ \} \cap \{$ *p*. *p permutes S* $\wedge\ p\ i = j'\ \} = \{\}$
(**is** *?L* ∩ *?R* = {})
⟨*proof*⟩

lemma *permutation-insert-inj-on*:
assumes $i < Suc\ n$
assumes $j < Suc\ n$

shows *inj-on* (*permutation-insert i j*) { *q*. *q permutes* {*0*..<*n*} }
(**is** *inj-on ?f ?S*)
⟨*proof*⟩

lemma *signof-permutation-insert*:
assumes *p*: *p permutes* {*0*..<*n*} **and** *i*: *i* < *Suc n* **and** *j*: *j* < *Suc n*
shows *signof* (*permutation-insert i j p*) = (−*1*::′*a*::*ring-1*)^(*i*+*j*) ∗ *signof p*
⟨*proof*⟩

lemma *foo*:
assumes *i*: *i* < *Suc n* **and** *j*: *j* < *Suc n*
assumes *q*: *q permutes* {*0*..<*n*}
shows {(*i*′, *permutation-insert i j q i*′) | *i*′. *i*′ ∈ {*0*..<*Suc n*} − {*i*} } =
{ (*insert-index i i*′′, *insert-index j* (*q i*′′)) | *i*′′. *i*′′ < *n* } (**is** *?L* = *?R*)
⟨*proof*⟩

definition *mat-delete A i j* ≡
mat (*dim-row A* − *1*) (*dim-col A* − *1*) (λ(*i*′,*j*′).
A $$ (*if i*′ < *i then i*′ *else Suc i*′, *if j*′ < *j then j*′ *else Suc j*′))

lemma *mat-delete-dim*[*simp*]:
dim-row (*mat-delete A i j*) = *dim-row A* − *1*
dim-col (*mat-delete A i j*) = *dim-col A* − *1*
⟨*proof*⟩

lemma *mat-delete-carrier*:
assumes *A*: *A* ∈ *carrier-mat m n*
shows *mat-delete A i j* ∈ *carrier-mat* (*m*−*1*) (*n*−*1*) ⟨*proof*⟩

lemma *mat-delete-index*:
assumes *A*: *A* ∈ *carrier-mat* (*Suc n*) (*Suc n*)
and *i*: *i* < *Suc n* **and** *j*: *j* < *Suc n*
and *i*′: *i*′ < *n* **and** *j*′: *j*′ < *n*
shows *A* $$ (*insert-index i i*′, *insert-index j j*′) = *mat-delete A i j* $$ (*i*′, *j*′)
⟨*proof*⟩

definition *cofactor A i j* = (−*1*)^(*i*+*j*) ∗ *det* (*mat-delete A i j*)

lemma *laplace-expansion-column*:
assumes *A*: (*A* :: ′*a* :: *comm-ring-1 mat*) ∈ *carrier-mat n n*
and *j*: *j* < *n*
shows *det A* = (∑*i*<*n*. *A* $$ (*i*,*j*) ∗ *cofactor A i j*)
⟨*proof*⟩

lemma *laplace-expansion-row*:
assumes *A*: (*A* :: ′*a* :: *comm-ring-1 mat*) ∈ *carrier-mat n n*
and *i*: *i* < *n*
shows *det A* = (∑*j*<*n*. *A* $$ (*i*,*j*) ∗ *cofactor A i j*)

⟨*proof*⟩

lemma *degree-det-le*: **assumes** $\bigwedge i\ j.\ i < n \Longrightarrow j < n \Longrightarrow \mathit{degree}\ (A\ \$\$\ (i,j)) \leq k$
  and *A*: $A \in \mathit{carrier\text{-}mat}\ n\ n$
shows $\mathit{degree}\ (\mathit{det}\ A) \leq k * n$
⟨*proof*⟩

lemma *upper-triangular-imp-det-eq-0-iff*:
  fixes $A :: {'a} :: \mathit{idom}\ \mathit{mat}$
  assumes $A \in \mathit{carrier\text{-}mat}\ n\ n$ **and** $\mathit{upper\text{-}triangular}\ A$
  shows $\mathit{det}\ A = 0 \longleftrightarrow 0 \in \mathit{set}\ (\mathit{diag\text{-}mat}\ A)$
  ⟨*proof*⟩

lemma *det-identical-columns*:
  assumes *A*: $A \in \mathit{carrier\text{-}mat}\ n\ n$
    and *ij*: $i \neq j$
    and *i*: $i < n$ **and** *j*: $j < n$
    and *r*: $\mathit{col}\ A\ i = \mathit{col}\ A\ j$
  shows $\mathit{det}\ A = 0$
⟨*proof*⟩

definition $\mathit{adj\text{-}mat} :: {'a} :: \mathit{comm\text{-}ring\text{-}1}\ \mathit{mat} \Rightarrow {'a}\ \mathit{mat}$ **where**
  $\mathit{adj\text{-}mat}\ A = \mathit{mat}\ (\mathit{dim\text{-}row}\ A)\ (\mathit{dim\text{-}col}\ A)\ (\lambda\ (i,j).\ \mathit{cofactor}\ A\ j\ i)$

lemma *adj-mat*: **assumes** *A*: $A \in \mathit{carrier\text{-}mat}\ n\ n$
  shows $\mathit{adj\text{-}mat}\ A \in \mathit{carrier\text{-}mat}\ n\ n$
  $A * \mathit{adj\text{-}mat}\ A = \mathit{det}\ A \cdot_m 1_m\ n$
  $\mathit{adj\text{-}mat}\ A * A = \mathit{det}\ A \cdot_m 1_m\ n$
⟨*proof*⟩

definition $\mathit{replace\text{-}col}\ A\ b\ k = \mathit{mat}\ (\mathit{dim\text{-}row}\ A)\ (\mathit{dim\text{-}col}\ A)\ (\lambda\ (i,j).\ \mathit{if}\ j = k\ \mathit{then}\ b\ \$\ i\ \mathit{else}\ A\ \$\$\ (i,j))$

lemma *cramer-lemma-mat*:
  assumes *A*: $A \in \mathit{carrier\text{-}mat}\ n\ n$
  and *x*: $x \in \mathit{carrier\text{-}vec}\ n$
  and *k*: $k < n$
shows $\mathit{det}\ (\mathit{replace\text{-}col}\ A\ (A *_v x)\ k) = x\ \$\ k * \mathit{det}\ A$
⟨*proof*⟩

end

10 Code Equations for Determinants

We compute determinants on arbitrary rings by applying elementary row-operations to bring a matrix on upper-triangular form. Then the determinant can be determined by multiplying all entries on the diagonal. Moreover the final result has to be divided by a factor which is determined by the row-operations that we performed. To this end, we require a division operation on the element type.

The algorithm is parametric in a selection function for the pivot-element, e.g., for matrices over polynomials it turned out that selecting a polynomial of minimal degree is beneficial.

theory *Determinant-Impl*
imports
Polynomial-Interpolation.Missing-Polynomial
HOL−Computational-Algebra.Polynomial-Factorial
Determinant
begin

type-synonym *'a det-selection-fun* = (*nat* × *'a*)*list* ⇒ *nat*

definition *det-selection-fun* :: *'a det-selection-fun* ⇒ *bool* **where**
det-selection-fun f = (∀ *xs*. *xs* ≠ [] ⟶ *f xs* ∈ *fst* ‘ *set xs*)

lemma *det-selection-funD*: *det-selection-fun f* ⟹ *xs* ≠ [] ⟹ *f xs* ∈ *fst* ‘ *set xs*
⟨*proof*⟩

definition *mute-fun* :: (*'a* :: *comm-ring-1* ⇒ *'a* ⇒ *'a* × *'a* × *'a*) ⇒ *bool* **where**
mute-fun f = (∀ *x y x' y' g*. *f x y* = (*x'*,*y'*,*g*) ⟶ *y* ≠ *0*
⟶ *x* = *x'* ∗ *g* ∧ *y* ∗ *x'* = *x* ∗ *y'*)

context
fixes *sel-fun* :: *'a* ::*idom-divide det-selection-fun*
begin

10.1 Properties of triangular matrices

Each column of a triangular matrix should satisfy the following property.

definition *triangular-column*::*nat* ⇒ *'a mat* ⇒ *bool*
where *triangular-column j A* ≡ ∀ *i*. *j* < *i* ⟶ *i* < *dim-row A* ⟶ *A* $$ (*i*,*j*) = *0*

lemma *triangular-columnD* [*dest*]:
triangular-column j A ⟹ *j* < *i* ⟹ *i* < *dim-row A* ⟹ *A* $$ (*i*,*j*) = *0*
⟨*proof*⟩

lemma *triangular-columnI* [*intro*]:
(⋀*i*. *j* < *i* ⟹ *i* < *dim-row A* ⟹ *A* $$ (*i*,*j*) = *0*) ⟹ *triangular-column j A*
⟨*proof*⟩

The following predicate states that the first k columns satisfy triangularity.

definition *triangular-to*:: *nat* ⇒ *'a mat* ⇒ *bool*

where *triangular-to k A* == ∀ *j*. *j*<*k* ⟶ *triangular-column j A*

lemma *triangular-to-triangular*: *upper-triangular A* = *triangular-to* (*dim-row A*) *A*
⟨*proof*⟩

lemma *triangular-toD* [*dest*]:
triangular-to k A ⟹ *j* < *k* ⟹ *j* < *i* ⟹ *i* < *dim-row A* ⟹ *A* $$ (*i*,*j*) = *0*
⟨*proof*⟩

lemma *triangular-toI* [*intro*]:
(⋀*i j*. *j* < *k* ⟹ *j* < *i* ⟹ *i* < *dim-row A* ⟹ *A* $$ (*i*,*j*) = *0*) ⟹ *triangular-to k A*
⟨*proof*⟩

lemma *triangle-growth*:
assumes *tri*:*triangular-to k A*
and *col*:*triangular-column k A*
shows *triangular-to* (*Suc k*) *A*
⟨*proof*⟩

lemma *triangle-trans*: *triangular-to k A* ⟹ *k* > *k*′ ⟹ *triangular-to k*′ *A*
⟨*proof*⟩

10.2 Algorithms for Triangulization

context
fixes *mf* :: ′*a* ⇒ ′*a* ⇒ ′*a* × ′*a* × ′*a*
begin

private fun *mute* :: ′*a* ⇒ *nat* ⇒ *nat* ⇒ ′*a* × ′*a mat* ⇒ ′*a* × ′*a mat* **where**
mute A-ll k l (*r*,*A*) = (*let p* = *A* $$ (*k*,*l*) *in if p* = *0 then* (*r*,*A*) *else*
case mf A-ll p of (*q*′,*p*′,*g*) ⇒
(*r* ∗ *q*′, *addrow* (−*p*′) *k l* (*multrow k q*′ *A*)))

lemma *mute-preserves-dimensions*:
assumes *mute q k l* (*r*,*A*) = (*r*′,*A*′)
shows [*simp*]: *dim-row A*′ = *dim-row A* **and** [*simp*]: *dim-col A*′ = *dim-col A*
⟨*proof*⟩

Algorithm *mute k l* makes *k*-th row *l*-th column element to 0.

lemma *mute-makes-0* :
assumes *mute-fun*: *mute-fun mf*
assumes *mute* (*A* $$ (*l*,*l*)) *k l* (*r*,*A*) = (*r*′,*A*′)
l < *dim-row A*
l < *dim-col A*
k < *dim-row A*
k ≠ *l*
shows *A*′ $$ (*k*,*l*) = *0*
⟨*proof*⟩

It will not touch unexpected rows.

lemma *mute-preserves*:
  mute q k l (r,A) = (r′,A′) ⟹
   i < dim-row A ⟹
   j < dim-col A ⟹
   l < dim-row A ⟹
   k < dim-row A ⟹
   i ≠ k ⟹
   A′ $$ (i,j) = A $$ (i,j)
   ⟨*proof*⟩

It preserves 0s in the touched row.

lemma *mute-preserves-0*:
  mute q k l (r,A) = (r′,A′) ⟹
   i < dim-row A ⟹
   j < dim-col A ⟹
   l < dim-row A ⟹
   k < dim-row A ⟹
   A $$ (i,j) = 0 ⟹
   A $$ (l,j) = 0 ⟹
   A′ $$ (i,j) = 0
   ⟨*proof*⟩

Hence, it will respect partially triangular matrix.

lemma *mute-preserves-triangle*:
 assumes *rA′* : *mute q k l (r,A) = (r′,A′)*
 and *triA*: *triangular-to l A*
 and *lk*: *l < k*
 and *kr*: *k < dim-row A*
 and *lr*: *l < dim-row A*
 and *lc*: *l < dim-col A*
 shows *triangular-to l A′*
⟨*proof*⟩

Recursive application of *mute*

private fun *sub1* :: *′a ⇒ nat ⇒ nat ⇒ ′a × ′a mat ⇒ ′a × ′a mat*
where *sub1 q 0 l rA = rA*
  | *sub1 q (Suc k) l rA = mute q (l + Suc k) l (sub1 q k l rA)*

lemma *sub1-preserves-dimensions*[*simp*]:
  sub1 q k l (r,A) = (r′,A′) ⟹ *dim-row A′ = dim-row A*
  sub1 q k l (r,A) = (r′,A′) ⟹ *dim-col A′ = dim-col A*
⟨*proof*⟩

lemma *sub1-closed* [*simp*]:
  sub1 q k l (r,A) = (r′,A′) ⟹ *A ∈ carrier-mat m n* ⟹ *A′ ∈ carrier-mat m n*
  ⟨*proof*⟩

lemma *sub1-preserves-diagnal*:

assumes *sub1 q k l* (*r*,*A*) = (*r*′,*A*′)
and *l* < *dim-col A*
and *k* + *l* < *dim-row A*
shows *A*′ $$ (*l*,*l*) = *A* $$ (*l*,*l*)
⟨*proof*⟩

Triangularity is respected by *sub1*.

lemma *sub1-preserves-triangle*:
assumes *sub1 q k l* (*r*,*A*) = (*r*′,*A*′)
and *tri*: *triangular-to l A*
and *lr*: *l* < *dim-row A*
and *lc*: *l* < *dim-col A*
and *lkr*: *l* + *k* < *dim-row A*
shows *triangular-to l A*′
⟨*proof*⟩

context
assumes *mf*: *mute-fun mf*
begin
lemma *sub1-makes-0s*:
assumes *sub1* (*A* $$ (*l*,*l*)) *k l* (*r*,*A*) = (*r*′,*A*′)
and *lr*: *l* < *dim-row A*
and *lc*: *l* < *dim-col A*
and *li*: *l* < *i*
and *i* ≤ *k* + *l*
and *k* + *l* < *dim-row A*
shows *A*′ $$ (*i*,*l*) = *0*
⟨*proof*⟩

lemma *sub1-triangulizes-column*:
assumes *rA*′: *sub1* (*A* $$ (*l*,*l*)) (*dim-row A* − *Suc l*) *l* (*r*,*A*) = (*r*′,*A*′)
and *tri*:*triangular-to l A*
and *r*: *dim-row A* > *0*
and *lr*: *l* < *dim-row A*
and *lc*: *l* < *dim-col A*
shows *triangular-column l A*′
⟨*proof*⟩

The algorithm *sub1* increases the number of columns that form triangle.

lemma *sub1-grows-triangle*:
assumes *rA*′: *sub1* (*A* $$ (*l*,*l*)) (*dim-row A* − *Suc l*) *l* (*r*,*A*) = (*r*′,*A*′)
and *r*: *dim-row A* > *0*
and *tri*:*triangular-to l A*
and *lr*: *l* < *dim-row A*
and *lc*: *l* < *dim-col A*
shows *triangular-to* (*Suc l*) *A*′
⟨*proof*⟩
end

10.3 Finding Non-Zero Elements

private definition *find-non0* :: *nat* ⇒ *'a mat* ⇒ *nat option* **where**
find-non0 l A = (*let is* = [*Suc l* ..< *dim-row A*];
Ais = *filter* (λ (*i*,*Ail*). *Ail* ≠ *0*) (*map* (λ *i*. (*i*, *A* \$\$ (*i*,*l*))) *is*)
in case Ais of [] ⇒ *None* | - ⇒ *Some* (*sel-fun Ais*))

lemma *find-non0*: **assumes** *sel-fun*: *det-selection-fun sel-fun*
and *res*: *find-non0 l A* = *Some m*
shows *A* \$\$ (*m*,*l*) ≠ *0 l* < *m m* < *dim-row A*
⟨*proof*⟩

If *find-non0 l A* fails, then *A* is already triangular to *l*-th column.

lemma *find-non0-all0*:
find-non0 l A = *None* ⟹ *triangular-column l A*
⟨*proof*⟩

10.4 Determinant Preserving Growth of Triangle

The algorithm *sub1* does not preserve determinants when it hits a 0-valued diagonal element. To avoid this case, we introduce the following operation:

private fun *sub2* :: *nat* ⇒ *nat* ⇒ *'a* × *'a mat* ⇒ *'a* × *'a mat*
where *sub2 d l* (*r*,*A*) = (
case find-non0 l A of None ⇒ (*r*,*A*)
| *Some m* ⇒ *let A'* = *swaprows m l A in sub1* (*A'* \$\$ (*l*,*l*)) (*d* − *Suc l*) *l* (−*r*, *A'*))

lemma *sub2-preserves-dimensions*[*simp*]:
assumes *rA'*: *sub2 d l* (*r*,*A*) = (*r'*,*A'*)
shows *dim-row A'* = *dim-row A* ∧ *dim-col A'* = *dim-col A*
⟨*proof*⟩

lemma *sub2-closed* [*simp*]:
sub2 d l (*r*,*A*) = (*r'*,*A'*) ⟹ *A* ∈ *carrier-mat m n* ⟹ *A'* ∈ *carrier-mat m n*
⟨*proof*⟩

context
assumes *sel-fun*: *det-selection-fun sel-fun*
begin

lemma *sub2-preserves-triangle*:
assumes *rA'*: *sub2 d l* (*r*,*A*) = (*r'*,*A'*)
and *tri*: *triangular-to l A*
and *lc*: *l* < *dim-col A*
and *ld*: *l* < *d*
and *dr*: *d* ≤ *dim-row A*
shows *triangular-to l A'*
⟨*proof*⟩

lemma *sub2-grows-triangle*:
  assumes *mf*: *mute-fun mf*
  and *rA′*: *sub2* (*dim-row A*) *l* (*r*,*A*) = (*r′*,*A′*)
  and *tri*: *triangular-to l A*
  and *lc*: *l* < *dim-col A*
  and *lr*: *l* < *dim-row A*
  shows *triangular-to* (*Suc l*) *A′*
⟨*proof*⟩
end

10.5 Recursive Triangulization of Columns

Now we recursively apply *sub2* to make the entire matrix to be triangular.

private fun *sub3* :: *nat* ⇒ *nat* ⇒ *′a* × *′a mat* ⇒ *′a* × *′a mat*
  where *sub3 d 0 rA* = *rA*
  | *sub3 d* (*Suc l*) *rA* = *sub2 d l* (*sub3 d l rA*)

lemma *sub3-preserves-dimensions*[*simp*]:
  sub3 d l (*r*,*A*) = (*r′*,*A′*) ⟹ *dim-row A′* = *dim-row A*
  sub3 d l (*r*,*A*) = (*r′*,*A′*) ⟹ *dim-col A′* = *dim-col A*
⟨*proof*⟩

lemma *sub3-closed*[*simp*]:
  sub3 k l (*r*,*A*) = (*r′*,*A′*) ⟹ *A* ∈ *carrier-mat m n* ⟹ *A′* ∈ *carrier-mat m n*
  ⟨*proof*⟩

lemma *sub3-makes-triangle*:
  assumes *mf*: *mute-fun mf*
  and *sel-fun*: *det-selection-fun sel-fun*
  and *sub3* (*dim-row A*) *l* (*r*,*A*) = (*r′*,*A′*)
  and *l* ≤ *dim-row A*
  and *l* ≤ *dim-col A*
  shows *triangular-to l A′*
  ⟨*proof*⟩

10.6 Triangulization

definition *triangulize* :: *′a mat* ⇒ *′a* × *′a mat*
where *triangulize A* = *sub3* (*dim-row A*) (*dim-row A*) (*1*,*A*)

lemma *triangulize-preserves-dimensions*[*simp*]:
  triangulize A = (*r′*,*A′*) ⟹ *dim-row A′* = *dim-row A*
  triangulize A = (*r′*,*A′*) ⟹ *dim-col A′* = *dim-col A*
  ⟨*proof*⟩

lemma *triangulize-closed*[*simp*]:
  triangulize A = (*r′*,*A′*) ⟹ *A* ∈ *carrier-mat m n* ⟹ *A′* ∈ *carrier-mat m n*
  ⟨*proof*⟩

context
  assumes *mf*: *mute-fun mf*
  and *sel-fun*: *det-selection-fun sel-fun*
begin

theorem *triangulized*:
  assumes $A \in$ *carrier-mat n n*
  and *triangulize* $A = (r',A')$
  shows *upper-triangular* A'
⟨*proof*⟩

10.7 Divisor will not be 0

Here we show that each sub-algorithm will not make r of the input/output pair (r, A) to 0. The algorithm *sub1 A-ll k l* $(r,\ A)$ requires $A_{l,l} \neq 0$.

lemma *sub1-divisor* [*simp*]:
  assumes *rA′*: *sub1 q k l* $(r,\ A) = (r',A')$
  and *r0*: $r \neq 0$
  and *All*: $q \neq 0$
  and $k + l <$ *dim-row* A
  and *lc*: $l <$ *dim-col* A
  shows $r' \neq 0$
⟨*proof*⟩

The algorithm *sub2* will not require such a condition.

lemma *sub2-divisor* [*simp*]:
  assumes *rA′*: *sub2 k l* $(r,\ A) = (r',A')$
  and *lk*: $l < k$
  and *kr*: $k \leq$ *dim-row* A
  and *lc*: $l <$ *dim-col* A
  and *r0*: $r \neq 0$
  shows $r' \neq 0$
⟨*proof*⟩

lemma *sub3-divisor* [*simp*]:
  assumes *sub3 d l* $(r,A) = (r'',A'')$
  and $l \leq d$
  and $d \leq$ *dim-row* A
  and $l \leq$ *dim-col* A
  and *r0*: $r \neq 0$
  shows $r'' \neq 0$
  ⟨*proof*⟩

theorem *triangulize-divisor*:
  assumes *A*: $A \in$ *carrier-mat d d*
  shows *triangulize* $A = (r',A') \Longrightarrow r' \neq 0$
⟨*proof*⟩

10.8 Determinant Preservation Results

For each sub-algorithm f, we show $f(r, A) = (r', A')$ implies $r * \mathit{det}\ A' = r' * \mathit{det}\ A$.

lemma *mute-det*:
  assumes $A \in \mathit{carrier\text{-}mat}\ n\ n$
  and rA': $\mathit{mute}\ q\ k\ l\ (r,A) = (r',A')$
  and $k < n$
  and $l < n$
  and $k \neq l$
  shows $r * \mathit{det}\ A' = r' * \mathit{det}\ A$
⟨*proof*⟩

lemma *sub1-det*:
  assumes A: $A \in \mathit{carrier\text{-}mat}\ n\ n$
  and *sub1*: $\mathit{sub1}\ q\ k\ l\ (r,A) = (r'',A'')$
  and *r0*: $r \neq 0$
  and *All0*: $q \neq 0$
  and *l*: $l + k < n$
  shows $r * \mathit{det}\ A'' = r'' * \mathit{det}\ A$
  ⟨*proof*⟩

lemma *sub2-det*:
  assumes A: $A \in \mathit{carrier\text{-}mat}\ d\ d$
  and rA': $\mathit{sub2}\ d\ l\ (r,A) = (r',A')$
  and *r0*: $r \neq 0$
  and *ld*: $l < d$
  shows $r * \mathit{det}\ A' = r' * \mathit{det}\ A$
⟨*proof*⟩

lemma *sub3-det*:
  assumes A:$A \in \mathit{carrier\text{-}mat}\ d\ d$
  and $\mathit{sub3}\ d\ l\ (r,A) = (r'',A'')$
  and *r0*: $r \neq 0$
  and $l \leq d$
  shows $r * \mathit{det}\ A'' = r'' * \mathit{det}\ A$
  ⟨*proof*⟩

theorem *triangulize-det*:
  assumes A: $A \in \mathit{carrier\text{-}mat}\ d\ d$
  and rA': $\mathit{triangulize}\ A = (r',A')$
  shows $\mathit{det}\ A * r' = \mathit{det}\ A'$
⟨*proof*⟩
end

10.9 Determinant Computation

definition *det-code* :: $'a\ \mathit{mat} \Rightarrow\ 'a$ **where**
  $\mathit{det\text{-}code}\ A = (\mathit{if}\ \mathit{dim\text{-}row}\ A = \mathit{dim\text{-}col}\ A\ \mathit{then}$

```
    case triangulize A of (m,A′) ⇒
      prod-list (diag-mat A′) div m
  else 0)
```

lemma *det-code*[*simp*]: **assumes** *sel-fun*: *det-selection-fun sel-fun*
  and *mf*: *mute-fun mf*
  shows *det-code A* = *det A*
  ⟨*proof*⟩

end
end

Now we can select an arbitrary selection and mute function. This will be important for computing resultants over polynomials, where usually a polynomial with small degree is preferable.

The default however is to use the first element.

definition *trivial-mute-fun* :: *′a* :: *comm-ring-1* ⇒ *′a* ⇒ *′a* × *′a* × *′a* **where**
  trivial-mute-fun x y = (*x*,*y*,*1*)

lemma *trivial-mute-fun*[*simp*,*intro*]: *mute-fun trivial-mute-fun*
  ⟨*proof*⟩

definition *fst-sel-fun* :: *′a det-selection-fun* **where**
  fst-sel-fun x = *fst* (*hd x*)

lemma *fst-sel-fun*[*simp*]: *det-selection-fun fst-sel-fun*
  ⟨*proof*⟩

context
  fixes *measure* :: *′a* ⇒ *nat*
begin
private fun *select-min-main* **where**
  select-min-main m i ((*j*,*p*) # *xs*) = (*let n* = *measure p in if n* < *m then select-min-main n j xs*
    else select-min-main m i xs)
| *select-min-main m i* [] = *i*

definition *select-min* :: (*nat* × *′a*) *list* ⇒ *nat* **where**
  select-min xs = (*case xs of* ((*i*,*p*) # *ys*) ⇒ (*select-min-main* (*measure p*) *i ys*))

lemma *select-min*[*simp*]: *det-selection-fun select-min*
  ⟨*proof*⟩
end

For the code equation we use the trivial mute and selection function as this does not impose any further class restrictions.

lemma *det-code-fst-sel-fun*[*code*]: *det A* = *det-code fst-sel-fun trivial-mute-fun A*
⟨*proof*⟩

But we also provide specialiced functions for more specific carriers.

definition *field-mute-fun* :: $'a$ :: *field* ⇒ $'a$ ⇒ $'a$ × $'a$ × $'a$ **where**
field-mute-fun x y = (*x*/*y*,*1*,*y*)

lemma *field-mute-fun*[*simp*,*intro*]: *mute-fun field-mute-fun*
⟨*proof*⟩

definition *det-field* :: $'a$ :: *field mat* ⇒ $'a$ **where**
det-field A = *det-code fst-sel-fun field-mute-fun A*

lemma *det-field*[*simp*]: *det-field* = *det*
⟨*proof*⟩

definition *gcd-mute-fun* :: $'a$ :: *ring-gcd* ⇒ $'a$ ⇒ $'a$ × $'a$ × $'a$ **where**
gcd-mute-fun x y = (*let g* = *gcd x y in* (*x div g*, *y div g*,*g*))

lemma *gcd-mute-fun*[*simp*,*intro*]: *mute-fun gcd-mute-fun*
⟨*proof*⟩

definition *det-int* :: *int mat* ⇒ *int* **where**
det-int A = *det-code* (*select-min* (λ *x*. *nat* (*abs x*))) *gcd-mute-fun A*

lemma *det-int*[*simp*]: *det-int* = *det*
⟨*proof*⟩

definition *det-field-poly* :: $'a$ :: {*field*,*field-gcd*} *poly mat* ⇒ $'a$ *poly* **where**
det-field-poly A = *det-code* (*select-min degree*) *gcd-mute-fun A*

lemma *det-field-poly*[*simp*]: *det-field-poly* = *det*
⟨*proof*⟩

end

11 Converting Matrices to Strings

We just instantiate matrices in the show-class by printing them as lists of lists.

theory *Show-Matrix*
imports
Show.Show
Matrix
begin

11.1 For the *show*-class

definition *shows-vec* :: $'a$ :: *show vec* ⇒ *shows* **where**
shows-vec v ≡ *shows* (*list-of-vec v*)

instantiation *vec* :: (*show*) *show*

begin

definition *shows-prec p* (*v* :: *'a vec*) ≡ *shows-vec v*
definition *shows-list* (*vs* :: *'a vec list*) ≡ *shows-sep shows-vec* (*shows* '', '') *vs*

instance
⟨*proof*⟩
end

definition *shows-mat* :: *'a* :: *show mat* ⇒ *shows* **where**
shows-mat A ≡ *shows* (*mat-to-list A*)

instantiation *mat* :: (*show*) *show*
begin

definition *shows-prec p* (*A* :: *'a mat*) ≡ *shows-mat A*
definition *shows-list* (*As* :: *'a mat list*) ≡ *shows-sep shows-mat* (*shows* '', '') *As*

instance
⟨*proof*⟩
end

end

theory *Shows-Literal-Matrix*
imports
Jordan-Normal-Form.Matrix
Show.Shows-Literal
begin

11.2 For the *showl*-class

instantiation *Matrix.vec* :: (*showl*)*showl* **begin**
definition *showsl-vec* :: *'a Matrix.vec* ⇒ *showsl* **where**
showsl-vec v ≡ *showsl-list* (*list-of-vec v*)
definition *showsl-list* (*xs* :: *'a Matrix.vec list*) = *default-showsl-list showsl xs*
instance ⟨*proof*⟩
end

instantiation *mat* :: (*showl*)*showl* **begin**
definition *showsl-mat* :: *'a Matrix.mat* ⇒ *showsl* **where**
showsl-mat a ≡ *default-showsl-list id* (*map showsl-list* (*mat-to-list a*))
definition *showsl-list* (*xs* :: *'a Matrix.mat list*) = *default-showsl-list showsl xs*
instance ⟨*proof*⟩
end

value *showsl* (*one-mat 3* :: *nat mat*) (*STR* ''*is the identity matrix of dimension 3*'')

end

12 Characteristic Polynomial

We define eigenvalues, eigenvectors, and the characteristic polynomial. We further prove that the eigenvalues are exactly the roots of the characteristic polynomial. Finally, we apply the fundamental theorem of algebra to show that the characteristic polynomial of a complex matrix can always be represented as product of linear factors $x - a$.

theory *Char-Poly*
imports
  Polynomial-Factorization.Fundamental-Theorem-Algebra-Factorized
  Polynomial-Interpolation.Missing-Polynomial
  Polynomial-Interpolation.Ring-Hom-Poly
  Determinant
  Complex-Main
begin

definition *eigenvector* :: *'a* :: *comm-ring-1 mat* ⇒ *'a vec* ⇒ *'a* ⇒ *bool* **where**
  eigenvector A v k = (*v* ∈ *carrier-vec* (*dim-row A*) ∧ *v* ≠ 0_v (*dim-row A*) ∧ *A* $*_v$ *v* = *k* $\cdot_v$ *v*)

lemma *eigenvector-pow*: **assumes** *A*: *A* ∈ *carrier-mat n n*
  and *ev*: *eigenvector A v* (*k* :: *'a* :: *comm-ring-1*)
  shows *A* $\widehat{}_m$ *i* $*_v$ *v* = *k*$\widehat{}$*i* $\cdot_v$ *v*
⟨*proof*⟩

definition *eigenvalue* :: *'a* :: *comm-ring-1 mat* ⇒ *'a* ⇒ *bool* **where**
  eigenvalue A k = (∃ *v*. *eigenvector A v k*)

definition *char-matrix* :: *'a* :: *field mat* ⇒ *'a* ⇒ *'a mat* **where**
  char-matrix A e = *A* + (($-e$) $\cdot_m$ (1_m (*dim-row A*)))

lemma *char-matrix-closed*[*simp*]: *A* ∈ *carrier-mat n n* ⟹ *char-matrix A e* ∈ *carrier-mat n n*
  ⟨*proof*⟩

lemma *eigenvector-char-matrix*: **assumes** *A*: (*A* :: *'a* :: *field mat*) ∈ *carrier-mat n n*
  shows *eigenvector A v e* = (*v* ∈ *carrier-vec n* ∧ *v* ≠ 0_v *n* ∧ *char-matrix A e* $*_v$ *v* = 0_v *n*)
⟨*proof*⟩

lemma *eigenvalue-char-matrix*: **assumes** *A*: (*A* :: *'a* :: *field mat*) ∈ *carrier-mat n n*

shows *eigenvalue A e* = (∃ *v*. *v* ∈ *carrier-vec n* ∧ *v* ≠ 0_v *n* ∧ *char-matrix A e* $*_v$ *v* = 0_v *n*)
⟨*proof*⟩

definition *find-eigenvector* :: *'a::field mat* ⇒ *'a* ⇒ *'a vec* **where**
find-eigenvector A e =
find-base-vector (*fst* (*gauss-jordan* (*char-matrix A e*) (0_m (*dim-row A*) *0*)))

lemma *find-eigenvector*: **assumes** *A*: *A* ∈ *carrier-mat n n*
and *ev*: *eigenvalue A e*
shows *eigenvector A* (*find-eigenvector A e*) *e*
⟨*proof*⟩

lemma *eigenvalue-imp-nonzero-dim*: **assumes** *A* ∈ *carrier-mat n n*
and *eigenvalue A ev*
shows *n* > *0*
⟨*proof*⟩

lemma *eigenvalue-det*: **assumes** *A*: (*A* :: *'a* :: *field mat*) ∈ *carrier-mat n n* **shows**
eigenvalue A e = (*det* (*char-matrix A e*) = *0*)
⟨*proof*⟩

definition *char-poly-matrix* :: *'a* :: *comm-ring-1 mat* ⇒ *'a poly mat* **where**
char-poly-matrix A = (([:*0*,*1*:] $\cdot_m$ 1_m (*dim-row A*)) + *map-mat* (λ *a*. [: − *a* :]) *A*)

lemma *char-poly-matrix-closed*[*simp*]: *A* ∈ *carrier-mat n n* ⟹ *char-poly-matrix A* ∈ *carrier-mat n n*
⟨*proof*⟩

definition *char-poly* :: *'a* :: *comm-ring-1 mat* ⇒ *'a poly* **where**
char-poly A = (*det* (*char-poly-matrix A*))

lemmas *char-poly-defs* = *char-poly-def char-poly-matrix-def*

lemma (**in** *comm-ring-hom*) *char-poly-matrix-hom*: **assumes** *A*: *A* ∈ *carrier-mat n n*
shows *char-poly-matrix* (mat_h *A*) = *map-mat* (*map-poly hom*) (*char-poly-matrix A*)
⟨*proof*⟩

lemma (**in** *comm-ring-hom*) *char-poly-hom*: **assumes** *A*: *A* ∈ *carrier-mat n n*
shows *char-poly* (*map-mat hom A*) = *map-poly hom* (*char-poly A*)
⟨*proof*⟩

context *inj-comm-ring-hom*
begin

lemma *eigenvector-hom*: **assumes** *A*: *A* ∈ *carrier-mat n n*

and *ev*: *eigenvector A v ev*
shows *eigenvector* (*mat*$_h$ *A*) (*vec*$_h$ *v*) (*hom ev*)
⟨*proof*⟩

lemma *eigenvalue-hom*: **assumes** *A*: *A* ∈ *carrier-mat n n*
and *ev*: *eigenvalue A ev*
shows *eigenvalue* (*mat*$_h$ *A*) (*hom ev*)
⟨*proof*⟩

lemma *eigenvector-hom-rev*: **assumes** *A*: *A* ∈ *carrier-mat n n*
and *ev*: *eigenvector* (*mat*$_h$ *A*) (*vec*$_h$ *v*) (*hom ev*)
shows *eigenvector A v ev*
⟨*proof*⟩

end

lemma *poly-det-cong*: **assumes** *A*: *A* ∈ *carrier-mat n n*
and *B*: *B* ∈ *carrier-mat n n*
and *poly*: ⋀ *i j*. *i* < *n* ⟹ *j* < *n* ⟹ *poly* (*B* $$ (*i*,*j*)) *k* = *A* $$ (*i*,*j*)
shows *poly* (*det B*) *k* = *det A*
⟨*proof*⟩

lemma *char-poly-matrix*: **assumes** *A*: (*A* :: *'a* :: *field mat*) ∈ *carrier-mat n n*
shows *poly* (*char-poly A*) *k* = *det* (− (*char-matrix A k*)) ⟨*proof*⟩

lemma *eigenvalue-root-char-poly*: **assumes** *A*: (*A* :: *'a* :: *field mat*) ∈ *carrier-mat n n*
shows *eigenvalue A k* ⟷ *poly* (*char-poly A*) *k* = *0*
⟨*proof*⟩

context
fixes *A* :: *'a* :: *comm-ring-1 mat* **and** *n* :: *nat*
assumes *A*: *A* ∈ *carrier-mat n n*
and *ut*: *upper-triangular A*
begin
lemma *char-poly-matrix-upper-triangular*: *upper-triangular* (*char-poly-matrix A*)
⟨*proof*⟩

lemma *char-poly-upper-triangular*:
char-poly A = (∏ *a* ← *diag-mat A*. [:− *a*, *1*:])
⟨*proof*⟩
end

lemma *map-poly-mult*: **assumes** *A*: *A* ∈ *carrier-mat nr n*
and *B*: *B* ∈ *carrier-mat n nc*
shows
map-mat (λ *a*. [: *a* :]) (*A* ∗ *B*) = *map-mat* (λ *a*. [: *a* :]) *A* ∗ *map-mat* (λ *a*. [: *a* :]) *B* (**is** *?id*)

map-mat (λ *a*. [: *a* :] ∗ *p*) (*A* ∗ *B*) = *map-mat* (λ *a*. [: *a* :] ∗ *p*) *A* ∗ *map-mat* (λ *a*. [: *a* :]) *B* (**is** *?left*)
map-mat (λ *a*. [: *a* :] ∗ *p*) (*A* ∗ *B*) = *map-mat* (λ *a*. [: *a* :]) *A* ∗ *map-mat* (λ *a*. [: *a* :] ∗ *p*) *B* (**is** *?right*)
⟨*proof*⟩

lemma *char-poly-similar*: **assumes** *similar-mat A* (*B* :: *'a* :: *comm-ring-1 mat*)
shows *char-poly A* = *char-poly B*
⟨*proof*⟩

lemma *degree-signof-mult*[*simp*]: *degree* (*signof p* ∗ *q*) = *degree q*
⟨*proof*⟩

lemma *degree-monic-char-poly*: **assumes** *A*: *A* ∈ *carrier-mat n n*
shows *degree* (*char-poly A*) = *n* ∧ *coeff* (*char-poly A*) *n* = *1*
⟨*proof*⟩

lemma *char-poly-factorized*: **fixes** *A* :: *complex mat*
assumes *A*: *A* ∈ *carrier-mat n n*
shows ∃ *as*. *char-poly A* = (∏ *a* ← *as*. [:− *a*, *1*:]) ∧ *length as* = *n*
⟨*proof*⟩

lemma *char-poly-four-block-zeros-col*: **assumes** *A1*: *A1* ∈ *carrier-mat 1 1*
and *A2*: *A2* ∈ *carrier-mat 1 n* **and** *A3*: *A3* ∈ *carrier-mat n n*
shows *char-poly* (*four-block-mat A1 A2* (0_m *n 1*) *A3*) = *char-poly A1* ∗ *char-poly A3*
(**is** *char-poly ?A* = *?cp1* ∗ *?cp3*)
⟨*proof*⟩

lemma *char-poly-transpose-mat*[*simp*]: **assumes** *A*: *A* ∈ *carrier-mat n n*
shows *char-poly* (*transpose-mat A*) = *char-poly A*
⟨*proof*⟩

lemma *pderiv-char-poly*: **fixes** *A* :: *'a* :: *idom mat*
assumes *A*: *A* ∈ *carrier-mat n n*
shows *pderiv* (*char-poly A*) = (∑ *i* < *n*. *char-poly* (*mat-delete A i i*))
⟨*proof*⟩

lemma *char-poly-0-column*: **fixes** *A* :: *'a* :: *idom mat*
assumes *0*: ⋀ *j*. *j* < *n* ⟹ *A* $$ (*j*,*i*) = *0*
and *A*: *A* ∈ *carrier-mat n n*
and *i*: *i* < *n*
shows *char-poly A* = *monom 1 1* ∗ *char-poly* (*mat-delete A i i*)
⟨*proof*⟩

definition *mat-erase* :: *'a* :: *zero mat* ⇒ *nat* ⇒ *nat* ⇒ *'a mat* **where**
mat-erase A i j = *Matrix.mat* (*dim-row A*) (*dim-col A*)
(λ (*i'*,*j'*). *if i'* = *i* ∨ *j'* = *j then 0 else A* $$ (*i'*,*j'*))

lemma *mat-erase-carrier*[*simp*]: (*mat-erase A i j*) ∈ *carrier-mat nr nc* ⟷ *A* ∈ *carrier-mat nr nc*
⟨*proof*⟩

lemma *pderiv-char-poly-mat-erase*: **fixes** *A* :: *'a* :: *idom mat*
assumes *A*: *A* ∈ *carrier-mat n n*
shows *monom 1 1* ∗ *pderiv* (*char-poly A*) = ($\sum$ *i* < *n*. *char-poly* (*mat-erase A i i*))
⟨*proof*⟩

end

13 Jordan Normal Form

This theory defines Jordan normal forms (JNFs) in a sparse representation, i.e., as block-diagonal matrices. We also provide a closed formula for powers of JNFs, which allows to estimate the growth rates of JNFs.

theory *Jordan-Normal-Form*
imports
Matrix
Char-Poly
Polynomial-Interpolation.Missing-Unsorted
begin

definition *jordan-block* :: *nat* ⇒ *'a* :: {*zero*,*one*} ⇒ *'a mat* **where**
jordan-block n a = *mat n n* (λ (*i*,*j*). *if i* = *j then a else if Suc i* = *j then 1 else 0*)

lemma *jordan-block-index*[*simp*]: *i* < *n* ⟹ *j* < *n* ⟹
jordan-block n a $$ (*i*,*j*) = (*if i* = *j then a else if Suc i* = *j then 1 else 0*)
dim-row (*jordan-block n k*) = *n*
dim-col (*jordan-block n k*) = *n*
⟨*proof*⟩

lemma *jordan-block-carrier*[*simp*]: *jordan-block n k* ∈ *carrier-mat n n*
⟨*proof*⟩

lemma *jordan-block-char-poly*: *char-poly* (*jordan-block n a*) = [: −*a*, *1*:]$\widehat{}$*n*
⟨*proof*⟩

lemma *jordan-block-pow-carrier*[*simp*]:
jordan-block n a $\widehat{}_m$ *r* ∈ *carrier-mat n n* ⟨*proof*⟩
lemma *jordan-block-pow-dim*[*simp*]:
dim-row (*jordan-block n a* $\widehat{}_m$ *r*) = *n* *dim-col* (*jordan-block n a* $\widehat{}_m$ *r*) = *n* ⟨*proof*⟩

lemma *jordan-block-pow*: (*jordan-block n* (*a* :: *'a* :: *comm-ring-1*)) $\widehat{}_m$ *r* =
mat n n (λ (*i*,*j*). *if i* ≤ *j then of-nat* (*r choose* (*j* − *i*)) ∗ *a* $\widehat{}$ (*r* + *i* − *j*) *else 0*)
⟨*proof*⟩

definition *jordan-matrix* :: (*nat* × *'a* :: {*zero*,*one*})*list* ⇒ *'a mat* **where**
jordan-matrix n-as = *diag-block-mat* (*map* (λ (*n*,*a*). *jordan-block n a*) *n-as*)

lemma *jordan-matrix-dim*[*simp*]:
dim-row (*jordan-matrix n-as*) = *sum-list* (*map fst n-as*)
dim-col (*jordan-matrix n-as*) = *sum-list* (*map fst n-as*)
⟨*proof*⟩

lemma *jordan-matrix-carrier*[*simp*]:
jordan-matrix n-as ∈ *carrier-mat* (*sum-list* (*map fst n-as*)) (*sum-list* (*map fst n-as*))
⟨*proof*⟩

lemma *jordan-matrix-upper-triangular*: *i* < *sum-list* (*map fst n-as*)
⟹ *j* < *i* ⟹ *jordan-matrix n-as* $$ (*i*,*j*) = *0*
⟨*proof*⟩

lemma *jordan-matrix-pow*: (*jordan-matrix n-as*) $\widehat{\ }_m$ *r* =
diag-block-mat (*map* (λ (*n*,*a*). (*jordan-block n a*) $\widehat{\ }_m$ *r*) *n-as*)
⟨*proof*⟩

lemma *jordan-matrix-char-poly*:
char-poly (*jordan-matrix n-as*) = (∏ (*n*, *a*)←*n-as*. [:− *a*, *1*:] ^ *n*)
⟨*proof*⟩

definition *jordan-nf* :: *'a* :: *semiring-1 mat* ⇒ (*nat* × *'a*)*list* ⇒ *bool* **where**
jordan-nf A n-as ≡ (*0* ∉ *fst* ‘ *set n-as* ∧ *similar-mat A* (*jordan-matrix n-as*))

lemma *jordan-nf-powE*: **assumes** *A*: *A* ∈ *carrier-mat n n* **and** *jnf*: *jordan-nf A n-as*
obtains *P Q* **where** *P* ∈ *carrier-mat n n Q* ∈ *carrier-mat n n* **and**
char-poly A = (∏ (*na*, *a*)←*n-as*. [:− *a*, *1*:] ^ *na*)
⋀ *k*. *A* $\widehat{\ }_m$ *k* = *P* ∗ (*jordan-matrix n-as*)$\widehat{\ }_m$ *k* ∗ *Q*
⟨*proof*⟩

lemma *choose-poly-bound*: **assumes** *i* ≤ *d*
shows *r choose i* ≤ *max 1* (*r*^*d*)
⟨*proof*⟩

context
fixes *b* :: *'a* :: *archimedean-field*
assumes *b*: *0* < *b b* < *1*
begin

lemma *poly-exp-constant-bound*: ∃ *p*. ∀ *x*. *c* ∗ *b* ^ *x* ∗ *of-nat x* ^ *deg* ≤ *p*
⟨*proof*⟩

lemma *poly-exp-max-constant-bound*: ∃ *p*. ∀ *x*. *c* ∗ *b* ^ *x* ∗ *max 1* (*of-nat x* ^ *deg*)

≤ *p*
⟨*proof*⟩
end

context
  fixes *a* :: *'a* :: *real-normed-field*
begin
lemma *jordan-block-bound*:
  assumes *i*: *i* < *n* **and** *j*: *j* < *n*
  shows *norm* ((*jordan-block n a* ^m *k*) $$ (*i*,*j*))
    ≤ *norm a* ^ (*k* + *i* − *j*) ∗ *max 1* (*of-nat k* ^ (*n* − *1*))
    (**is** *?lhs* ≤ *?rhs*)
⟨*proof*⟩

lemma *jordan-block-poly-bound*:
  assumes *i*: *i* < *n* **and** *j*: *j* < *n* **and** *a*: *norm a* = *1*
  shows *norm* ((*jordan-block n a* ^m *k*) $$ (*i*,*j*)) ≤ *max 1* (*of-nat k* ^ (*n* − *1*))
    (**is** *?lhs* ≤ *?rhs*)
⟨*proof*⟩

theorem *jordan-block-constant-bound*: **assumes** *a*: *norm a* < *1*
  shows ∃ *p*. ∀ *i j k*. *i* < *n* ⟶ *j* < *n* ⟶ *norm* ((*jordan-block n a* ^m *k*) $$ (*i*,*j*))
≤ *p*
⟨*proof*⟩

definition *norm-bound* :: *'a mat* ⇒ *real* ⇒ *bool* **where**
  norm-bound A b ≡ ∀ *i j*. *i* < *dim-row A* ⟶ *j* < *dim-col A* ⟶ *norm* (*A* $$ (*i*,*j*)) ≤ *b*

lemma *norm-boundI*[*intro*]:
  assumes ⋀ *i j*. *i* < *dim-row A* ⟹ *j* < *dim-col A* ⟹ *norm* (*A* $$ (*i*,*j*)) ≤ *b*
  shows *norm-bound A b*
  ⟨*proof*⟩

lemma *jordan-block-constant-bound2*:
∃ *p*. *norm* (*a* :: *'a* :: *real-normed-field*) < *1* ⟶
    (∀ *i j k*. *i* < *n* ⟶ *j* < *n* ⟶ *norm* ((*jordan-block n a* ^m *k*) $$ (*i*, *j*)) ≤ *p*)
⟨*proof*⟩

lemma *jordan-matrix-poly-bound2*:
  fixes *n-as* :: (*nat* × *'a*) *list*
  assumes *n-as*: ⋀ *n a*. (*n*,*a*) ∈ *set n-as* ⟹ *n* > *0* ⟹ *norm a* ≤ *1*
  and *N*: ⋀ *n a*. (*n*,*a*) ∈ *set n-as* ⟹ *norm a* = *1* ⟹ *n* ≤ *N*
  shows ∃ *c1*. ∀ *k*. ∀ *e* ∈ *elements-mat* (*jordan-matrix n-as* ^m *k*).
    norm e ≤ *c1* + *of-nat k* ^ (*N* − *1*)
⟨*proof*⟩

lemma *norm-bound-bridge*:

$\forall e \in$ *elements-mat A*. *norm e* $\leq b \Longrightarrow$ *norm-bound A b*
⟨*proof*⟩

lemma *norm-bound-mult*: **assumes** *A1*: *A1* $\in$ *carrier-mat nr n*
and *A2*: *A2* $\in$ *carrier-mat n nc*
and *b1*: *norm-bound A1 b1*
and *b2*: *norm-bound A2 b2*
shows *norm-bound* (*A1* $*$ *A2*) (*b1* $*$ *b2* $*$ *of-nat n*)
⟨*proof*⟩

lemma *norm-bound-max*: *norm-bound A* (*Max* {*norm* (*A* \$\$ (*i*,*j*)) | *i j*. *i* $<$ *dim-row A* $\wedge$ *j* $<$ *dim-col A*})
(**is** *norm-bound A* (*Max ?norms*))
⟨*proof*⟩

lemma *jordan-matrix-poly-bound*: **fixes** *n-as* :: (*nat* × ′*a*)*list*
assumes *n-as*: $\bigwedge$ *n a*. (*n*,*a*) $\in$ *set n-as* $\Longrightarrow$ *n* $>$ *0* $\Longrightarrow$ *norm a* $\leq$ *1*
and *N*: $\bigwedge$ *n a*. (*n*,*a*) $\in$ *set n-as* $\Longrightarrow$ *norm a* $=$ *1* $\Longrightarrow$ *n* $\leq$ *N*
shows $\exists$ *c1*. $\forall$ *k*. *norm-bound* (*jordan-matrix n-as* $\widehat{\ }_m$ *k*) (*c1* $+$ *of-nat k* $\widehat{\ }$ (*N* $-$ *1*))
⟨*proof*⟩

lemma *jordan-nf-matrix-poly-bound*: **fixes** *n-as* :: (*nat* × ′*a*)*list*
assumes *A*: *A* $\in$ *carrier-mat n n*
and *n-as*: $\bigwedge$ *n a*. (*n*,*a*) $\in$ *set n-as* $\Longrightarrow$ *n* $>$ *0* $\Longrightarrow$ *norm a* $\leq$ *1*
and *N*: $\bigwedge$ *n a*. (*n*,*a*) $\in$ *set n-as* $\Longrightarrow$ *norm a* $=$ *1* $\Longrightarrow$ *n* $\leq$ *N*
and *jnf*: *jordan-nf A n-as*
shows $\exists$ *c1 c2*. $\forall$ *k*. *norm-bound* (*A* $\widehat{\ }_m$ *k*) (*c1* $+$ *c2* $*$ *of-nat k* $\widehat{\ }$ (*N* $-$ *1*))
⟨*proof*⟩
end

context
fixes *f-ty* :: ′*a* :: *field itself*
begin
lemma *char-matrix-jordan-block*: *char-matrix* (*jordan-block n a*) *b* $=$ (*jordan-block n* (*a* $-$ *b*))
⟨*proof*⟩

lemma *diag-jordan-block-pow*: *diag-mat* (*jordan-block n* (*a* :: ′*a*) $\widehat{\ }_m$ *k*) $=$ *replicate n* (*a* $\widehat{\ }$ *k*)
⟨*proof*⟩

lemma *jordan-block-zero-pow*: (*jordan-block n* (*0* :: ′*a*)) $\widehat{\ }_m$ *k* $=$
(*mat n n* (λ (*i*,*j*). *if j* $\geq$ *i* $\wedge$ *j* $-$ *i* $=$ *k then 1 else 0*))
⟨*proof*⟩
end

lemma *jordan-matrix-concat-diag-block-mat*: *jordan-matrix* (*concat jbs*) $=$ *diag-block-mat* (*map jordan-matrix jbs*)

⟨*proof*⟩

lemma *jordan-nf-diag-block-mat*: **assumes** *Ms*: $\bigwedge$ *A jbs*. (*A*,*jbs*) $\in$ *set Ms* $\Longrightarrow$ *jordan-nf A jbs*
shows *jordan-nf* (*diag-block-mat* (*map fst Ms*)) (*concat* (*map snd Ms*))
⟨*proof*⟩

lemma *jordan-nf-char-poly*: **assumes** *jordan-nf A n-as*
shows *char-poly A* = ($\prod$ (*n*,*a*) $\leftarrow$ *n-as*. [:$-$ *a*, *1*:] ^ *n*)
⟨*proof*⟩

lemma *jordan-nf-block-size-order-bound*: **assumes** *jnf*: *jordan-nf A n-as*
and *mem*: (*n*,*a*) $\in$ *set n-as*
shows *n* $\leq$ *order a* (*char-poly A*)
⟨*proof*⟩

lemma *similar-mat-jordan-block-smult*: **fixes** *A* :: *'a* :: *field mat*
assumes *similar-mat A* (*jordan-block n a*)
and *k*: *k* $\neq$ *0*
shows *similar-mat* ($k \cdot_m A$) (*jordan-block n* ($k * a$))
⟨*proof*⟩

lemma *jordan-matrix-Cons*: *jordan-matrix* (*Cons* (*n*,*a*) *n-as*) = *four-block-mat*
(*jordan-block n a*) (0_m *n* (*sum-list* (*map fst n-as*)))
(0_m (*sum-list* (*map fst n-as*)) *n*) (*jordan-matrix n-as*)
⟨*proof*⟩

lemma *similar-mat-jordan-matrix-smult*: **fixes** *n-as* :: (*nat* $\times$ *'a* :: *field*) *list*
assumes *k*: *k* $\neq$ *0*
shows *similar-mat* ($k \cdot_m$ *jordan-matrix n-as*) (*jordan-matrix* (*map* (λ (*n*,*a*). (*n*, $k * a$)) *n-as*))
⟨*proof*⟩

lemma *jordan-nf-smult*: **fixes** *k* :: *'a* :: *field*
assumes *jn*: *jordan-nf A n-as*
and *k*: *k* $\neq$ *0*
shows *jordan-nf* ($k \cdot_m A$) (*map* (λ (*n*,*a*). (*n*, $k * a$)) *n-as*)
⟨*proof*⟩

lemma *jordan-nf-order*: **assumes** *jordan-nf A n-as*
shows *order a* (*char-poly A*) = *sum-list* (*map fst* (*filter* (λ *na*. *snd na* = *a*) *n-as*))
⟨*proof*⟩

13.1 Application for Complexity

lemma *factored-char-poly-norm-bound*: **assumes** *A*: *A* $\in$ *carrier-mat n n*

and *linear-factors*: *char-poly A* = ($\prod$ (*a* :: *'a* :: *real-normed-field*) ← *as*. [:− *a*, *1*:])
and *jnf-exists*: ∃ *n-as*. *jordan-nf A n-as*
and *le-1*: $\bigwedge$ *a*. *a* ∈ *set as* ⟹ *norm a* ≤ *1*
and *le-N*: $\bigwedge$ *a*. *a* ∈ *set as* ⟹ *norm a* = *1* ⟹ *length* (*filter* ((=) *a*) *as*) ≤ *N*
shows ∃ *c1 c2*. ∀ *k*. *norm-bound* (*A* $\hat{}_m$ *k*) (*c1* + *c2* ∗ *of-nat k* ^ (*N* − *1*))
⟨*proof*⟩

end

14 Missing Vector Spaces

This theory provides some lemmas which we required when working with vector spaces.

theory *Missing-VectorSpace*
imports
VectorSpace.VectorSpace
Missing-Ring
HOL−Library.Multiset
begin

locale *comp-fun-commute-on* =
fixes *f* :: *'a* ⇒ *'a* ⇒ *'a* **and** *A*::*'a set*
assumes *comp-fun-commute-restrict*: ∀ *y*∈*A*. ∀ *x*∈*A*. ∀ *z*∈*A*. *f y* (*f x z*) = *f x* (*f y z*)
and *f*: *f* : *A* → *A* → *A*
begin

lemma *comp-fun-commute-on-UNIV*:
assumes *A* = (*UNIV* :: *'a set*)
shows *comp-fun-commute f*
⟨*proof*⟩

lemma *fun-left-comm*:
assumes *y* ∈ *A* **and** *x* ∈ *A* **and** *z* ∈ *A* **shows** *f y* (*f x z*) = *f x* (*f y z*)
⟨*proof*⟩

lemma *commute-left-comp*:
assumes *y* ∈ *A* **and** *x*∈*A* **and** *z*∈*A* **and** *g* ∈ *A* → *A*
shows *f y* (*f x* (*g z*)) = *f x* (*f y* (*g z*))
⟨*proof*⟩

lemma *fold-graph-finite*:
assumes *fold-graph f z B y*

shows *finite B*
⟨*proof*⟩

lemma *fold-graph-closed*:
assumes *fold-graph f z B y* **and** $B \subseteq A$ **and** $z \in A$
shows $y \in A$
⟨*proof*⟩

lemma *fold-graph-insertE-aux*:
fold-graph f z B y $\Longrightarrow a \in B \Longrightarrow z{\in}A$
$\Longrightarrow B \subseteq A$
$\Longrightarrow \exists y'.\ y = f\ a\ y' \wedge$ *fold-graph f z* $(B - \{a\})\ y' \wedge y' \in A$
⟨*proof*⟩

lemma *fold-graph-insertE*:
assumes *fold-graph f z* (*insert x B*) *v* **and** $x \notin B$ **and** *insert x B* $\subseteq A$ **and** $z{\in}A$
obtains *y* **where** $v = f\ x\ y$ **and** *fold-graph f z B y*
⟨*proof*⟩

lemma *fold-graph-determ*: *fold-graph f z B x* $\Longrightarrow$ *fold-graph f z B y* $\Longrightarrow\ B \subseteq A \Longrightarrow z{\in}A \Longrightarrow y = x$
⟨*proof*⟩

lemma *fold-equality*: *fold-graph f z B y* $\Longrightarrow B \subseteq A \Longrightarrow z \in A \Longrightarrow$ *Finite-Set.fold f z B* $= y$
⟨*proof*⟩

lemma *fold-graph-fold*:
assumes *f*: *finite B* **and** *BA*: $B{\subseteq}A$ **and** *z*: $z \in A$
shows *fold-graph f z B* (*Finite-Set.fold f z B*)
⟨*proof*⟩

lemma *fold-insert* [*simp*]:
assumes *finite B* **and** $x \notin B$ **and** *BA*: *insert x B* $\subseteq A$ **and** *z*: $z \in A$
shows *Finite-Set.fold f z* (*insert x B*) $=$ *f x* (*Finite-Set.fold f z B*)
⟨*proof*⟩
end

lemma *fold-cong*:
assumes *f*: *comp-fun-commute-on f A* **and** *g*: *comp-fun-commute-on g A*
and *finite S*
and *cong*: $\bigwedge x.\ x \in S \Longrightarrow f\ x = g\ x$
and $s = t$ **and** $S = T$
and *SA*: $S \subseteq A$ **and** *s*: $s{\in}A$
shows *Finite-Set.fold f s S* $=$ *Finite-Set.fold g t T*
⟨*proof*⟩

context *comp-fun-commute-on*
begin

lemma *comp-fun-Pi*: $(\lambda x.\ f\ x\ \widehat{\ }\widehat{\ }\ g\ x) \in A \to A \to A$
⟨*proof*⟩

lemma *comp-fun-commute-funpow*: *comp-fun-commute-on* $(\lambda x.\ f\ x\ \widehat{\ }\widehat{\ }\ g\ x)\ A$
⟨*proof*⟩

lemma *fold-mset-add-mset*:
  assumes *MA*: *set-mset M* ⊆ *A* **and** *s*: *s* ∈ *A* **and** *x*: *x* ∈ *A*
  shows *fold-mset f s* (*add-mset x M*) = *f x* (*fold-mset f s M*)
⟨*proof*⟩
end

lemma *Diff-not-in*: *a* ∉ *A* − {*a*} ⟨*proof*⟩

context *abelian-group* **begin**

lemma *finsum-restrict*:
  assumes *fA*: *f* : *A* → *carrier G*
    and *restr*: *restrict f A* = *restrict g A*
  shows *finsum G f A* = *finsum G g A*
⟨*proof*⟩

lemma *minus-nonzero*: *x* : *carrier G* ⟹ *x* ≠ **0** ⟹ ⊖ *x* ≠ **0**
  ⟨*proof*⟩

end

lemma (**in** *ordered-comm-monoid-add*) *positive-sum*:
  assumes *X* : *finite X*
    and *f* : *X* → { *y* :: *'a*. *y* ≥ *0* }
  shows *sum f X* ≥ *0* ∧ (*sum f X* = *0* ⟶ *f* ‘ *X* ⊆ {*0*})
  ⟨*proof*⟩

lemma *insert-union*: *insert x X* = *X* ∪ {*x*} ⟨*proof*⟩

context *vectorspace* **begin**

lemmas *lincomb-insert2* = *lincomb-insert*[*unfolded insert-union*[*symmetric*]]

lemma *lincomb-restrict*:
assumes *U*: $U \subseteq carrier\ V$
and *a*: $a : U \to carrier\ K$
and *restr*: $restrict\ a\ U = restrict\ b\ U$
shows $lincomb\ a\ U = lincomb\ b\ U$
⟨*proof*⟩

lemma *lindep-span*:
assumes *U*: $U \subseteq carrier\ V$ **and** *finU*: *finite U*
shows $lin\text{-}dep\ U = (\exists\, u \in U.\ u \in span\ (U - \{u\}))$ (**is** $?l = ?r$)
⟨*proof*⟩

lemma *not-lindepD*:
assumes $\sim$ *lin-dep S*
and *finite A* $A \subseteq S\ f : A \to carrier\ K\ lincomb\ f\ A = zero\ V$
shows $f : A \to \{zero\ K\}$
⟨*proof*⟩

lemma *span-mem*:
assumes *E*: $E \subseteq carrier\ V$ **and** *uE*: $u : E$ **shows** $u : span\ E$
⟨*proof*⟩

lemma *lincomb-distrib*:
assumes *U*: $U \subseteq carrier\ V$
and *a*: $a : U \to carrier\ K$
and *c*: $c : carrier\ K$
shows $c \odot_V lincomb\ a\ U = lincomb\ (\lambda u.\ c \otimes_K a\ u)\ U$
(**is** $- = lincomb\ ?b\ U$)
⟨*proof*⟩

lemma *span-swap*:
assumes *finE*[*simp*]: *finite E*
and *E*[*simp*]: $E \subseteq carrier\ V$
and *u*[*simp*]: $u : carrier\ V$
and *usE*: $u \notin span\ E$
and *v*[*simp*]: $v : carrier\ V$
and *usEv*: $u : span\ (insert\ v\ E)$
shows $span\ (insert\ u\ E) \subseteq span\ (insert\ v\ E)$ (**is** $?L \subseteq ?R$)
⟨*proof*⟩

lemma *basis-swap*:
assumes *finE*[*simp*]: *finite E*
and *u*[*simp*]: $u : carrier\ V$
and *uE*[*simp*]: $u \notin E$
and *b*: $basis\ (insert\ u\ E)$
and *v*[*simp*]: $v : carrier\ V$
and *uEv*: $u : span\ (insert\ v\ E)$
shows $basis\ (insert\ v\ E)$

⟨*proof*⟩

lemma *span-empty*: *span* {} = {*zero V*}
⟨*proof*⟩

lemma *span-self*: **assumes** [*simp*]: *v* : *carrier V* **shows** *v* : *span* {*v*}
⟨*proof*⟩

lemma *span-zero*: *zero V* : *span U* ⟨*proof*⟩

definition *emb* **where** *emb f D x* = (*if x* : *D then f x else zero K*)

lemma *emb-carrier*[*simp*]: *f* : *D* → *R* ⟹ *emb f D* : *D* → *R*
⟨*proof*⟩

lemma *emb-restrict*: *restrict* (*emb f D*) *D* = *restrict f D*
⟨*proof*⟩

lemma *emb-zero*: *emb f D* : *X* − *D* → { *zero K* }
⟨*proof*⟩

lemma *lincomb-clean*:
assumes *A*: *A* ⊆ *carrier V*
and *Z*: *Z* ⊆ *carrier V*
and *finA*: *finite A*
and *finZ*: *finite Z*
and *aA*: *a* : *A* → *carrier K*
and *aZ*: *a* : *Z* → { *zero K* }
shows *lincomb a* (*A* ∪ *Z*) = *lincomb a A*
⟨*proof*⟩

lemma *span-add1*:
assumes *U*: *U* ⊆ *carrier V* **and** *v*: *v* : *span U* **and** *w*: *w* : *span U*
shows $v \oplus_V w$: *span U*
⟨*proof*⟩

lemma *span-neg*:
assumes *U*: *U* ⊆ *carrier V* **and** *vU*: *v* : *span U*
shows $\ominus_V v$: *span U*
⟨*proof*⟩

lemma *span-closed*[*simp*]: *U* ⊆ *carrier V* ⟹ *v* : *span U* ⟹ *v* : *carrier V*
⟨*proof*⟩

lemma *span-add*:
assumes *U*: *U* ⊆ *carrier V* **and** *vU*: *v* : *span U* **and** *w*[*simp*]: *w* : *carrier V*
shows *w* : *span U* ⟷ $v \oplus_V w$: *span U* (**is** *?L* ⟷ *?R*)
⟨*proof*⟩

lemma *lincomb-union*:
assumes *U*: $U \subseteq$ *carrier V*
and *U′*[*simp*]: $U' \subseteq$ *carrier V*
and *disj*: $U \cap U' = \{\}$
and *finU*: *finite U*
and *finU′*: *finite U′*
and *a*: $a : U \cup U' \rightarrow$ *carrier K*
shows *lincomb a* $(U \cup U')$ = *lincomb a U* $\oplus_V$ *lincomb a U′*
⟨*proof*⟩

lemma *span-union1*:
assumes *U*: $U \subseteq$ *carrier V* **and** *U′*: $U' \subseteq$ *carrier V* **and** *UU′*: *span U* = *span U′*
and *W*: $W \subseteq$ *carrier V* **and** *W′*: $W' \subseteq$ *carrier V* **and** *WW′*: *span W* = *span W′*
shows *span* $(U \cup W) \subseteq$ *span* $(U' \cup W')$ (**is** *?L* $\subseteq$ *?R*)
⟨*proof*⟩

lemma *span-Un*:
assumes *U*: $U \subseteq$ *carrier V* **and** *U′*: $U' \subseteq$ *carrier V* **and** *UU′*: *span U* = *span U′*
and *W*: $W \subseteq$ *carrier V* **and** *W′*: $W' \subseteq$ *carrier V* **and** *WW′*: *span W* = *span W′*
shows *span* $(U \cup W)$ = *span* $(U' \cup W')$ (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *lincomb-zero*:
assumes *U*: $U \subseteq$ *carrier V* **and** *a*: $a : U \rightarrow \{$*zero K*$\}$
shows *lincomb a U* = *zero V*
⟨*proof*⟩

end

context *module*
begin

lemma *lincomb-empty*[*simp*]: *lincomb a* $\{\} = \mathbf{0}_M$
⟨*proof*⟩

end

context *linear-map*
begin

interpretation *Ker*: *vectorspace K* (*V.vs kerT*)
⟨*proof*⟩

interpretation *im*: *vectorspace K* (*W.vs imT*)

⟨*proof*⟩

lemma *inj-imp-Ker0*:
assumes *inj-on T* (*carrier V*)
shows *carrier* (*V.vs kerT*) = $\{\mathbf{0}_V\}$
⟨*proof*⟩

lemma *Ke0-imp-inj*:
assumes *c*: *carrier* (*V.vs kerT*) = $\{\mathbf{0}_V\}$
shows *inj-on T* (*carrier V*)
⟨*proof*⟩

corollary *Ke0-iff-inj*: *inj-on T* (*carrier V*) = (*carrier* (*V.vs kerT*) = $\{\mathbf{0}_V\}$)
⟨*proof*⟩

lemma *inj-imp-dim-ker0*:
assumes *inj-on T* (*carrier V*)
shows *vectorspace.dim K* (*V.vs kerT*) = *0*
⟨*proof*⟩

lemma *surj-imp-imT-carrier*:
assumes *surj*: *T‘* (*carrier V*) = *carrier W*
shows (*imT*) = *carrier W*
⟨*proof*⟩

lemma *dim-eq*:
assumes *fin-dim-V*: *V.fin-dim*
and *i*: *inj-on T* (*carrier V*) **and** *surj*: *T‘* (*carrier V*) = *carrier W*
shows *V.dim* = *W.dim*
⟨*proof*⟩

lemma *lincomb-linear-image*:
assumes *inj-T*: *inj-on T* (*carrier V*)
assumes *A-in-V*: *A* ⊆ *carrier V* **and** *a*: *a* ∈ (*T‘A*) → *carrier K*
assumes *f*: *finite A*
shows *W.module.lincomb a* (*T‘A*) = *T* (*V.module.lincomb* (*a* ∘ *T*) *A*)
⟨*proof*⟩

lemma *surj-fin-dim*:
assumes *fd*: *V.fin-dim* **and** *surj*: *T‘* (*carrier V*) = *carrier W*
shows *image-fin-dim*: *W.fin-dim*
⟨*proof*⟩

lemma *linear-inj-image-is-basis*:
assumes *inj-T*: *inj-on T* (*carrier V*) **and** *surj*: *T‘* (*carrier V*) = *carrier W*

and *basis-B*: *V.basis B*
and *fin-dim-V*: *V.fin-dim*
shows *W.basis* (*T‘B*)
⟨*proof*⟩

end

lemma (**in** *vectorspace*) *dim1I*:
assumes *gen-set* $\{v\}$
assumes $v \neq \mathbf{0}_V$ $v \in$ *carrier V*
shows *dim* = *1*
⟨*proof*⟩

lemma (**in** *vectorspace*) *dim0I*:
assumes *gen-set* $\{\mathbf{0}_V\}$
shows *dim* = *0*
⟨*proof*⟩

lemma (**in** *vectorspace*) *dim-le1I*:
assumes *gen-set* $\{v\}$
assumes $v \in$ *carrier V*
shows *dim* ≤ *1*
⟨*proof*⟩

definition *find-indices* **where** *find-indices x xs* ≡ [*i* ← [*0..<length xs*]. *xs*!*i* = *x*]

lemma *find-indices-Nil* [*simp*]:
  find-indices x [] = []
  ⟨*proof*⟩

lemma *find-indices-Cons*:
  find-indices x (*y*#*ys*) = (*if x* = *y then Cons 0 else id*) (*map Suc* (*find-indices x ys*))
⟨*proof*⟩

lemma *find-indices-snoc* [*simp*]:
  find-indices x (*ys*@[*y*]) = *find-indices x ys* @ (*if x* = *y then* [*length ys*] *else* [])
  ⟨*proof*⟩

lemma *mem-set-find-indices* [*simp*]: *i* ∈ *set* (*find-indices x xs*) ⟷ *i* < *length xs* ∧ *xs*!*i* = *x*
  ⟨*proof*⟩

lemma *distinct-find-indices*: *distinct* (*find-indices x xs*)
  ⟨*proof*⟩

context *abelian-monoid* **begin**

definition *sumlist*

where *sumlist xs* ≡ *foldr* (⊕) *xs* **0**

lemma [*simp*]:
shows *sumlist-Cons*: *sumlist* (*x*#*xs*) = *x* ⊕ *sumlist xs*
and *sumlist-Nil*: *sumlist* [] = **0**
⟨*proof*⟩

lemma *sumlist-carrier* [*simp*]:
assumes *set xs* ⊆ *carrier G* **shows** *sumlist xs* ∈ *carrier G*
⟨*proof*⟩

lemma *sumlist-neutral*:
assumes *set xs* ⊆ {**0**} **shows** *sumlist xs* = **0**
⟨*proof*⟩

lemma *sumlist-append*:
assumes *set xs* ⊆ *carrier G* **and** *set ys* ⊆ *carrier G*
shows *sumlist* (*xs* @ *ys*) = *sumlist xs* ⊕ *sumlist ys*
⟨*proof*⟩

lemma *sumlist-snoc*:
assumes *set xs* ⊆ *carrier G* **and** *x* ∈ *carrier G*
shows *sumlist* (*xs* @ [*x*]) = *sumlist xs* ⊕ *x*
⟨*proof*⟩

lemma *sumlist-as-finsum*:
assumes *set xs* ⊆ *carrier G* **and** *distinct xs* **shows** *sumlist xs* = ($\bigoplus$*x*∈*set xs*. *x*)
⟨*proof*⟩

lemma *sumlist-map-as-finsum*:
assumes *f* : *set xs* → *carrier G* **and** *distinct xs*
shows *sumlist* (*map f xs*) = ($\bigoplus$*x* ∈ *set xs*. *f x*)
⟨*proof*⟩

definition *summset* **where** *summset M* ≡ *fold-mset* (⊕) **0** *M*

lemma *summset-empty* [*simp*]: *summset* {#} = **0** ⟨*proof*⟩

lemma *fold-mset-add-carrier*: *a* ∈ *carrier G* ⟹ *set-mset M* ⊆ *carrier G* ⟹ *fold-mset* (⊕) *a M* ∈ *carrier G*
⟨*proof*⟩

lemma *summset-carrier*[*intro*]: *set-mset M* ⊆ *carrier G* ⟹ *summset M* ∈ *carrier G*
⟨*proof*⟩

lemma *summset-add-mset*[*simp*]:
assumes *a*: *a* ∈ *carrier G* **and** *MG*: *set-mset M* ⊆ *carrier G*

shows *summset* (*add-mset a M*) = *a* ⊕ *summset M*
⟨*proof*⟩

lemma *sumlist-as-summset*:
assumes *set xs* ⊆ *carrier G* **shows** *sumlist xs* = *summset* (*mset xs*)
⟨*proof*⟩

lemma *sumlist-rev*:
assumes *set xs* ⊆ *carrier G*
shows *sumlist* (*rev xs*) = *sumlist xs*
⟨*proof*⟩

lemma *sumlist-as-fold*:
assumes *set xs* ⊆ *carrier G*
shows *sumlist xs* = *fold* (⊕) *xs* **0**
⟨*proof*⟩

end

context *Module.module* **begin**

definition *lincomb-list*
where *lincomb-list c vs* = *sumlist* (*map* (λ*i*. *c i* $\odot_M$ *vs* ! *i*) [*0*..<*length vs*])

lemma *lincomb-list-carrier*:
assumes *set vs* ⊆ *carrier M* **and** *c* : {*0*..<*length vs*} → *carrier R*
shows *lincomb-list c vs* ∈ *carrier M*
⟨*proof*⟩

lemma *lincomb-list-Nil* [*simp*]: *lincomb-list c* [] = $\mathbf{0}_M$
⟨*proof*⟩

lemma *lincomb-list-Cons* [*simp*]:
lincomb-list c (*v*#*vs*) = *c 0* $\odot_M$ *v* $\oplus_M$ *lincomb-list* (*c o Suc*) *vs*
⟨*proof*⟩

lemma *lincomb-list-eq-0*:
assumes ⋀*i*. *i* < *length vs* ⟹ *c i* $\odot_M$ *vs* ! *i* = $\mathbf{0}_M$
shows *lincomb-list c vs* = $\mathbf{0}_M$
⟨*proof*⟩

definition *mk-coeff* **where** *mk-coeff vs c v* ≡ *R.sumlist* (*map c* (*find-indices v vs*))

lemma *mk-coeff-carrier*:
assumes *c* : {*0*..<*length vs*} → *carrier R* **shows** *mk-coeff vs c w* ∈ *carrier R*
⟨*proof*⟩

lemma *mk-coeff-Cons*:
assumes *c* : {*0*..<*length* (*v*#*vs*)} → *carrier R*

shows *mk-coeff* (*v#vs*) *c* = (*λw.* (*if w* = *v then c 0 else* **0**) ⊕ *mk-coeff vs* (*c o Suc*) *w*)
⟨*proof*⟩

lemma *mk-coeff-0*[*simp*]:
assumes *v* ∉ *set vs*
shows *mk-coeff vs c v* = **0**
⟨*proof*⟩

lemma *lincomb-list-as-lincomb*:
assumes *vs-M*: *set vs* ⊆ *carrier M* **and** *c*: *c* : {*0..<length vs*} → *carrier R*
shows *lincomb-list c vs* = *lincomb* (*mk-coeff vs c*) (*set vs*)
⟨*proof*⟩

definition *span-list vs* ≡ {*lincomb-list c vs* | *c. c* : {*0..<length vs*} → *carrier R*}

lemma *in-span-listI*:
assumes *c* : {*0..<length vs*} → *carrier R* **and** *v* = *lincomb-list c vs*
shows *v* ∈ *span-list vs*
⟨*proof*⟩

lemma *in-span-listE*:
assumes *v* ∈ *span-list vs*
and ⋀*c. c* : {*0..<length vs*} → *carrier R* ⟹ *v* = *lincomb-list c vs* ⟹ *thesis*
shows *thesis*
⟨*proof*⟩

lemmas *lincomb-insert2* = *lincomb-insert*[*unfolded insert-union*[*symmetric*]]

lemma *lincomb-zero*:
assumes *U*: *U* ⊆ *carrier M* **and** *a*: *a* : *U* → {*zero R*}
shows *lincomb a U* = *zero M*
⟨*proof*⟩

end

hide-const (**open**) *Multiset.mult*
end

15 Matrices as Vector Spaces

This theory connects the Matrix theory with the VectorSpace theory of Holden Lee. As a consequence notions like span, basis, linear dependence, etc. are available for vectors and matrices of the Matrix-theory.

theory *VS-Connect*
imports
Matrix
Missing-VectorSpace

Determinant
begin

hide-const (**open**) *Multiset.mult*
hide-const (**open**) *Polynomial.smult*
hide-const (**open**) *Modules.module*
hide-const (**open**) *subspace*
hide-fact (**open**) *subspace-def*

named-theorems *class-ring-simps*

abbreviation *class-ring* :: *'a* :: {*times*,*plus*,*one*,*zero*} *ring* **where**
class-ring ≡ ⦇ *carrier* = *UNIV*, *mult* = (∗), *one* = *1*, *zero* = *0*, *add* = (+) ⦈

interpretation *class-semiring*: *semiring class-ring* :: *'a* :: *semiring-1 ring*
rewrites [*class-ring-simps*]: *carrier class-ring* = *UNIV*
and [*class-ring-simps*]: *mult class-ring* = (∗)
and [*class-ring-simps*]: *add class-ring* = (+)
and [*class-ring-simps*]: *one class-ring* = *1*
and [*class-ring-simps*]: *zero class-ring* = *0*
and [*class-ring-simps*]: *pow* (*class-ring* :: *'a ring*) = (^)
and [*class-ring-simps*]: *finsum* (*class-ring* :: *'a ring*) = *sum*
⟨*proof*⟩

interpretation *class-ring*: *ring class-ring* :: *'a* :: *ring-1 ring*
rewrites *carrier class-ring* = *UNIV*
and *mult class-ring* = (∗)
and *add class-ring* = (+)
and *one class-ring* = *1*
and *zero class-ring* = *0*
and [*class-ring-simps*]: *a-inv* (*class-ring* :: *'a ring*) = *uminus*
and [*class-ring-simps*]: *a-minus* (*class-ring* :: *'a ring*) = *minus*
and *pow* (*class-ring* :: *'a ring*) = (^)
and *finsum* (*class-ring* :: *'a ring*) = *sum*
⟨*proof*⟩

interpretation *class-cring*: *cring class-ring* :: *'a* :: *comm-ring-1 ring*
rewrites *carrier class-ring* = *UNIV*
and *mult class-ring* = (∗)
and *add class-ring* = (+)
and *one class-ring* = *1*
and *zero class-ring* = *0*
and *a-inv* (*class-ring* :: *'a ring*) = *uminus*
and *a-minus* (*class-ring* :: *'a ring*) = *minus*
and *pow* (*class-ring* :: *'a ring*) = (^)
and *finsum* (*class-ring* :: *'a ring*) = *sum*
and [*class-ring-simps*]: *finprod class-ring* = *prod*
⟨*proof*⟩

definition *div0* :: *′a* :: {*one*,*plus*,*times*,*zero*} **where**
div0 ≡ *m-inv* (*class-ring* :: *′a ring*) *0*

lemma *class-field*: *field* (*class-ring* :: *′a* :: *field ring*) (**is** *field ?r*)
⟨*proof*⟩

interpretation *class-field*: *field class-ring* :: *′a* :: *field ring*
rewrites *carrier class-ring* = *UNIV*
and *mult class-ring* = (∗)
and *add class-ring* = (+)
and *one class-ring* = *1*
and *zero class-ring* = *0*
and *a-inv class-ring* = *uminus*
and *a-minus class-ring* = *minus*
and *pow class-ring* = (ˆ)
and *finsum class-ring* = *sum*
and *finprod class-ring* = *prod*
and [*class-ring-simps*]: *m-inv* (*class-ring* :: *′a ring*) *x* =
(*if x* = *0 then div0 else inverse x*)

⟨*proof*⟩

lemmas *matrix-vs-simps* = *module-mat-simps class-ring-simps*

definition *class-field* :: *′a* :: *field ring*
where [*class-ring-simps*]: *class-field* ≡ *class-ring*

locale *matrix-ring* =
fixes *n* :: *nat*
and *field-type* :: *′a* :: *field itself*
begin
abbreviation *R* **where** *R* ≡ *ring-mat TYPE*(*′a*) *n n*
sublocale *ring R*
rewrites *carrier R* = *carrier-mat n n*
and *add R* = (+)
and *mult R* = (∗)
and *one R* = 1_m *n*
and *zero R* = 0_m *n n*
⟨*proof*⟩

end

lemma *matrix-vs*: *vectorspace* (*class-ring* :: *′a* :: *field ring*) (*module-mat TYPE*(*′a*) *nr nc*)
⟨*proof*⟩

```
locale vec-module =
  fixes f-ty::'a::comm-ring-1 itself
  and n::nat
begin

abbreviation V where V ≡ module-vec TYPE('a) n

sublocale Module.module class-ring :: 'a ring V
  rewrites carrier V = carrier-vec n
    and add V = (+)
    and zero V = 0_v n
    and module.smult V = (·_v)
    and carrier class-ring = UNIV
    and monoid.mult class-ring = (∗)
    and add class-ring = (+)
    and one class-ring = 1
    and zero class-ring = 0
    and a-inv (class-ring :: 'a ring) = uminus
    and a-minus (class-ring :: 'a ring) = (−)
    and pow (class-ring :: 'a ring) = (^)
    and finsum (class-ring :: 'a ring) = sum
    and finprod (class-ring :: 'a ring) = prod
    and ⋀X. X ⊆ UNIV = True
    and ⋀x. x ∈ UNIV = True
    and ⋀a A. a ∈ A → UNIV ≡ True
    and ⋀P. P ∧ True ≡ P
    and ⋀P. (True ⟹ P) ≡ Trueprop P
  ⟨proof⟩

end

locale matrix-vs =
  fixes nr :: nat
    and nc :: nat
    and field-type :: 'a :: field itself
begin

abbreviation V where V ≡ module-mat TYPE('a) nr nc
sublocale
  vectorspace class-ring V
  rewrites carrier V = carrier-mat nr nc
    and add V = (+)
    and mult V = (∗)
    and one V = 1_m nr
    and zero V = 0_m nr nc
    and smult V = (·_m)
    and carrier class-ring = UNIV
    and mult class-ring = (∗)
```

and *add class-ring* = (+)
and *one class-ring* = *1*
and *zero class-ring* = *0*
and *a-inv* (*class-ring* :: *'a ring*) = *uminus*
and *a-minus* (*class-ring* :: *'a ring*) = *minus*
and *pow* (*class-ring* :: *'a ring*) = (^)
and *finsum* (*class-ring* :: *'a ring*) = *sum*
and *finprod* (*class-ring* :: *'a ring*) = *prod*
and *m-inv* (*class-ring* :: *'a ring*) *x* =
(*if x* = *0 then div0 else inverse x*)
⟨*proof*⟩
end

lemma *vec-module*: *module* (*class-ring* :: *'a* :: *field ring*) (*module-vec TYPE*(*'a*) *n*)
⟨*proof*⟩

lemma *vec-vs*: *vectorspace* (*class-ring* :: *'a* :: *field ring*) (*module-vec TYPE*(*'a*) *n*)
⟨*proof*⟩

locale *vec-space* =
fixes *f-ty*::*'a*::*field itself*
and *n*::*nat*
begin

sublocale *vec-module f-ty n*⟨*proof*⟩

sublocale *vectorspace class-ring V*
rewrites *cV*[*simp*]: *carrier V* = *carrier-vec n*
and [*simp*]: *add V* = (+)
and [*simp*]: *zero V* = 0_v *n*
and [*simp*]: *smult V* = $(\cdot_v)$
and *carrier class-ring* = *UNIV*
and *mult class-ring* = (∗)
and *add class-ring* = (+)
and *one class-ring* = *1*
and *zero class-ring* = *0*
and *a-inv* (*class-ring* :: *'a ring*) = *uminus*
and *a-minus* (*class-ring* :: *'a ring*) = *minus*
and *pow* (*class-ring* :: *'a ring*) = (^)
and *finsum* (*class-ring* :: *'a ring*) = *sum*
and *finprod* (*class-ring* :: *'a ring*) = *prod*
and *m-inv* (*class-ring* :: *'a ring*) *x* = (*if x* = *0 then div0 else inverse x*)
⟨*proof*⟩

lemma *finsum-vec*[*simp*]: *finsum-vec TYPE*(*'a*) *n* = *finsum V*
⟨*proof*⟩

lemma *finsum-scalar-prod-sum*:
assumes *f*: *f* : *U* → *carrier-vec n*

and *w*: *w*: *carrier-vec n*
shows *finsum V f U* $\cdot$ *w* = *sum* (λ*u*. *f u* $\cdot$ *w*) *U*
⟨*proof*⟩

lemma *vec-neg*[*simp*]: **assumes** *x* : *carrier-vec n* **shows** $\ominus_V$ *x* = − *x*
⟨*proof*⟩

lemma *finsum-dim*:
finite A $\Longrightarrow$ *f* $\in$ *A* $\rightarrow$ *carrier-vec n* $\Longrightarrow$ *dim-vec* (*finsum V f A*) = *n*
⟨*proof*⟩

lemma *lincomb-dim*:
assumes *fin*: *finite X*
and *X*: *X* $\subseteq$ *carrier-vec n*
shows *dim-vec* (*lincomb a X*) = *n*
⟨*proof*⟩

lemma *finsum-index*:
assumes *i*: *i* < *n*
and *f*: *f* $\in$ *X* $\rightarrow$ *carrier-vec n*
and *X*: *X* $\subseteq$ *carrier-vec n*
shows *finsum V f X* \$ *i* = *sum* (λ*x*. *f x* \$ *i*) *X*
⟨*proof*⟩

lemma *lincomb-index*:
assumes *i*: *i* < *n*
and *X*: *X* $\subseteq$ *carrier-vec n*
shows *lincomb a X* \$ *i* = *sum* (λ*x*. *a x* $*$ *x* \$ *i*) *X*
⟨*proof*⟩

lemma *append-insert*: *set* (*xs* @ [*x*]) = *insert x* (*set xs*) ⟨*proof*⟩

lemma *lincomb-units*:
assumes *i*: *i* < *n*
shows *lincomb a* (*set* (*unit-vecs n*)) \$ *i* = *a* (*unit-vec n i*)
⟨*proof*⟩

lemma *lincomb-coordinates*:
assumes *v*: *v* : *carrier-vec n*
defines *a* ≡ (λ*u*. *v* \$ (*THE i*. *u* = *unit-vec n i*))
shows *lincomb a* (*set* (*unit-vecs n*)) = *v*
⟨*proof*⟩

lemma *span-unit-vecs-is-carrier*: *span* (*set* (*unit-vecs n*)) = *carrier-vec n* (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *fin-dim*[*simp*]: *fin-dim*

⟨*proof*⟩

lemma *unit-vecs-basis*: *basis* (*set* (*unit-vecs n*)) ⟨*proof*⟩

lemma *unit-vecs-length*[*simp*]: *length* (*unit-vecs n*) = *n*
⟨*proof*⟩

lemma *unit-vecs-distinct*: *distinct* (*unit-vecs n*)
⟨*proof*⟩

lemma *dim-is-n*: *dim* = *n*
⟨*proof*⟩

end

locale *mat-space* =
vec-space f-ty nc **for** *f-ty*::*'a*::*field itself* **and** *nc*::*nat* +
fixes *nr* :: *nat*
begin
abbreviation *M* **where** *M* ≡ *ring-mat TYPE*(*'a*) *nc nr*
end

context *vec-space*
begin
lemma *fin-dim-span*:
assumes *finite A A* ⊆ *carrier V*
shows *vectorspace.fin-dim class-ring* (*vs* (*span A*))
⟨*proof*⟩

lemma *fin-dim-span-cols*:
assumes *A* ∈ *carrier-mat n nc*
shows *vectorspace.fin-dim class-ring* (*vs* (*span* (*set* (*cols A*))))
⟨*proof*⟩
end

context *vec-module*
begin

lemma *lincomb-list-as-mat-mult*:
assumes ∀ *w* ∈ *set ws*. *dim-vec w* = *n*
shows *lincomb-list c ws* = *mat-of-cols n ws* $*_v$ *vec* (*length ws*) *c* (**is** *?l ws c* = *?r ws c*)
⟨*proof*⟩

lemma *lincomb-vec-diff-add*:
assumes *A*: *A* ⊆ *carrier-vec n*
and *BA*: *B* ⊆ *A* **and** *fin-A*: *finite A*
and *f*: *f* ∈ *A* → *UNIV* **shows** *lincomb f A* = *lincomb f* (*A*−*B*) + *lincomb f B*
⟨*proof*⟩

```
lemma dim-sumlist:
  assumes ∀ x∈set xs. dim-vec x = n
  shows dim-vec (M.sumlist xs) = n ⟨proof⟩

lemma sumlist-nth:
  assumes ∀ x∈set xs. dim-vec x = n and i<n
  shows (M.sumlist xs) $ i= sum (λj. (xs ! j) $ i) {0..<length xs}
  ⟨proof⟩

lemma lincomb-as-lincomb-list-distinct:
  assumes s: set ws ⊆ carrier-vec n and d: distinct ws
  shows lincomb f (set ws) = lincomb-list (λi. f (ws ! i)) ws
⟨proof⟩

end

locale idom-vec = vec-module f-ty for f-ty :: 'a :: idom itself
begin

lemma lin-dep-cols-imp-det-0':
  fixes ws
  defines A ≡ mat-of-cols n ws
  assumes dimv-ws: ∀ w∈set ws. dim-vec w = n
  assumes A: A ∈ carrier-mat n n and ld-cols: lin-dep (set (cols A))
  shows  det A = 0
⟨proof⟩

lemma lin-dep-cols-imp-det-0:
  assumes A: A ∈ carrier-mat n n and ld: lin-dep (set (cols A))
  shows det A = 0
⟨proof⟩

corollary lin-dep-rows-imp-det-0:
  assumes A: A ∈ carrier-mat n n and ld: lin-dep (set (rows A))
  shows det A = 0
  ⟨proof⟩

lemma det-not-0-imp-lin-indpt-rows:
  assumes A: A ∈ carrier-mat n n and det: det A ≠ 0
  shows lin-indpt (set (rows A))
    ⟨proof⟩

lemma upper-triangular-imp-lin-indpt-rows:
  assumes A: A ∈ carrier-mat n n
    and tri: upper-triangular A
    and diag: 0 ∉ set (diag-mat A)
  shows lin-indpt (set (rows A))
  ⟨proof⟩
```

lemma *lincomb-as-lincomb-list*:
  fixes *ws f*
  assumes *s*: *set ws* ⊆ *carrier-vec n*
  shows *lincomb f* (*set ws*) = *lincomb-list* (λ*i*. *if* ∃*j*<*i*. *ws*!*i* = *ws*!*j then 0 else f* (*ws* ! *i*)) *ws*
  ⟨*proof*⟩

lemma *span-list-as-span*:
  assumes *set vs* ⊆ *carrier-vec n*
  shows *span-list vs* = *span* (*set vs*)
  ⟨*proof*⟩

lemma *in-spanI*[*intro*]:
  assumes *v* = *lincomb a A finite A A* ⊆ *W*
  shows *v* ∈ *span W*
⟨*proof*⟩
lemma *in-spanE*:
  assumes *v* ∈ *span W*
  shows ∃ *a A*. *v* = *lincomb a A* ∧ *finite A* ∧ *A* ⊆ *W*
⟨*proof*⟩

declare *in-own-span*[*intro*]

lemma *smult-in-span*:
  assumes *W* ⊆ *carrier-vec n* **and** *insp*: *x* ∈ *span W*
  shows $c \cdot_v x$ ∈ *span W*
⟨*proof*⟩

lemma *span-subsetI*: **assumes** *ws*: *ws* ⊆ *carrier-vec n*
  us ⊆ *span ws*
shows *span us* ⊆ *span ws*
  ⟨*proof*⟩

end

context *vec-space* **begin**
sublocale *idom-vec*⟨*proof*⟩

lemma *sumlist-in-span*: **assumes** *W*: *W* ⊆ *carrier-vec n*
  shows (⋀*x*. *x* ∈ *set xs* ⟹ *x* ∈ *span W*) ⟹ *sumlist xs* ∈ *span W*
⟨*proof*⟩

lemma *span-span*[*simp*]:
  assumes *W* ⊆ *carrier-vec n*
  shows *span* (*span W*) = *span W*
⟨*proof*⟩

lemma *upper-triangular-imp-basis*:
  assumes *A*: $A \in$ *carrier-mat n n*
    and *tri*: *upper-triangular A*
    and *diag*: $0 \notin$ *set* (*diag-mat A*)
  shows *basis* (*set* (*rows A*))
  ⟨*proof*⟩

lemma *fin-dim-span-rows*:
assumes *A*: $A \in$ *carrier-mat nr n*
shows *vectorspace.fin-dim class-ring* (*vs* (*span* (*set* (*rows A*))))
⟨*proof*⟩

definition *row-space* $B =$ *span* (*set* (*rows B*))
definition *col-space* $B =$ *span* (*set* (*cols B*))

lemma *row-space-eq-col-space-transpose*:
  shows *row-space* $A =$ *col-space* A^T
  ⟨*proof*⟩

lemma *col-space-eq-row-space-transpose*:
  shows *col-space* $A =$ *row-space* A^T
  ⟨*proof*⟩

lemma *col-space-eq*:
  assumes *A*: $A \in$ *carrier-mat n nc*
  shows *col-space* $A = \{y \in$ *carrier-vec* (*dim-row A*). $\exists x \in$ *carrier-vec* (*dim-col A*). $A *_v x = y\}$
⟨*proof*⟩

lemma *vector-space-row-space*:
  assumes *A*: $A \in$ *carrier-mat nr n*
  shows *vectorspace class-ring* (*vs* (*row-space A*))
⟨*proof*⟩

lemma *row-space-eq*:
  assumes *A*: $A \in$ *carrier-mat nr n*
  shows *row-space* $A = \{w \in$ *carrier-vec* (*dim-col A*). $\exists y \in$ *carrier-vec* (*dim-row A*). $A^T *_v y = w\}$
  ⟨*proof*⟩

lemma *row-space-is-preserved*:
  assumes *inv-P*: *invertible-mat P* **and** *P*: $P \in$ *carrier-mat m m* **and** *A*: $A \in$ *carrier-mat m n*
  shows *row-space* $(P*A) =$ *row-space A*
⟨*proof*⟩

end

context *vec-module* **begin**

lemma *R-sumlist*[*simp*]: *R.sumlist* = *sum-list*
⟨*proof*⟩

lemma *sumlist-dim*: **assumes** ⋀ *x*. *x* ∈ *set xs* ⟹ *x* ∈ *carrier-vec n*
  shows *dim-vec* (*sumlist xs*) = *n*
  ⟨*proof*⟩

lemma *sumlist-vec-index*: **assumes** ⋀ *x*. *x* ∈ *set xs* ⟹ *x* ∈ *carrier-vec n*
  and *i* < *n*
shows *sumlist xs* \$ *i* = *sum-list* (*map* (λ *x*. *x* \$ *i*) *xs*)
  ⟨*proof*⟩

lemma *scalar-prod-left-sum-distrib*:
  assumes *vs*: ⋀ *v*. *v* ∈ *set vvs* ⟹ *v* ∈ *carrier-vec n* **and** *w*: *w* ∈ *carrier-vec n*
  shows *sumlist vvs* • *w* = *sum-list* (*map* (λ *v*. *v* • *w*) *vvs*)
  ⟨*proof*⟩

lemma *scalar-prod-right-sum-distrib*:
  assumes *vs*: ⋀ *v*. *v* ∈ *set vvs* ⟹ *v* ∈ *carrier-vec n* **and** *w*: *w* ∈ *carrier-vec n*
  shows *w* • *sumlist vvs* = *sum-list* (*map* (λ *v*. *w* • *v*) *vvs*)
  ⟨*proof*⟩

lemma *lincomb-list-add-vec-2*: **assumes** *us*: *set us* ⊆ *carrier-vec n*
  and *x*: *x* = *lincomb-list lc* (*us* [*i* := *us* ! *i* + *c* $\cdot_v$ *us* ! *j*])
  and *i*: *j* < *length us i* < *length us i* ≠ *j*
shows *x* = *lincomb-list* (*lc* (*j* := *lc j* + *lc i* ∗ *c*)) *us* (**is** - = *?x*)
⟨*proof*⟩

lemma *lincomb-list-add-vec-1*: **assumes** *us*: *set us* ⊆ *carrier-vec n*
  and *x*: *x* = *lincomb-list lc us*
  and *i*: *j* < *length us i* < *length us i* ≠ *j*
shows *x* = *lincomb-list* (*lc* (*j* := *lc j* − *lc i* ∗ *c*)) (*us* [*i* := *us* ! *i* + *c* $\cdot_v$ *us* ! *j*]) (**is** - = *?x*)
⟨*proof*⟩

end

context *vec-space*
begin
lemma *add-vec-span*: **assumes** *us*: *set us* ⊆ *carrier-vec n*
  and *i*: *j* < *length us i* < *length us i* ≠ *j*
shows *span* (*set us*) = *span* (*set* (*us* [*i* := *us* ! *i* + *c* $\cdot_v$ *us* ! *j*])) (**is** - = *span* (*set ?ws*))
⟨*proof*⟩

lemma *prod-in-span*[*intro*!]:
assumes *b* ∈ *carrier-vec n S* ⊆ *carrier-vec n a* = *0* ∨ *b* ∈ *span S*
shows *a* $\cdot_v$ *b* ∈ *span S*
⟨*proof*⟩

lemma *det-nonzero-congruence*:
assumes *eq*:*A* ∗ *M* = *B* ∗ *M* **and** *det*:*det* (*M*::′*a mat*) ≠ *0*
and *M*: *M* ∈ *carrier-mat n n* **and** *carr*:*A* ∈ *carrier-mat n n B* ∈ *carrier-mat n n*
shows *A* = *B*
⟨*proof*⟩

lemma *mat-of-rows-mult-as-finsum*:
assumes *v* ∈ *carrier-vec* (*length lst*) ⋀ *i*. *i* < *length lst* ⟹ *lst* ! *i* ∈ *carrier-vec n*
defines *f l* ≡ *sum* (λ *i*. *if l* = *lst* ! *i then v* $ *i else 0*) {*0*..<*length lst*}
shows *mat-of-cols-mult-as-finsum*:*mat-of-cols n lst* $*_v$ *v* = *lincomb f* (*set lst*)
⟨*proof*⟩

end

end

16 Gram-Schmidt Orthogonalization

This theory provides the Gram-Schmidt orthogonalization algorithm, that takes the conjugate operation into account. It works over fields like the rational, real, or complex numbers.

theory *Gram-Schmidt*
imports
VS-Connect
Missing-VectorSpace
Conjugate
begin

16.1 Orthogonality with Conjugates

definition *corthogonal vs* ≡
∀ *i* < *length vs*. ∀ *j* < *length vs*. *vs* ! *i* ∙*c vs* ! *j* = *0* ⟷ *i* ≠ *j*

lemma *corthogonalD*[*elim*]:
corthogonal vs ⟹ *i* < *length vs* ⟹ *j* < *length vs* ⟹
vs ! *i* ∙*c vs* ! *j* = *0* ⟷ *i* ≠ *j*
⟨*proof*⟩

lemma *corthogonalI*[*intro*]:
(⋀*i j*. *i* < *length vs* ⟹ *j* < *length vs* ⟹ *vs* ! *i* ∙*c vs* ! *j* = *0* ⟷ *i* ≠ *j*) ⟹
corthogonal vs

⟨*proof*⟩

lemma *corthogonal-distinct*: *corthogonal us* ⟹ *distinct us*
⟨*proof*⟩

lemma *corthogonal-sort*:
  assumes *dist′*: *distinct us′*
    and *mem*: *set us* = *set us′*
  shows *corthogonal us* ⟹ *corthogonal us′*
⟨*proof*⟩

16.2 The Algorithm

fun *adjuster* :: *nat* ⇒ *′a* :: *conjugatable-field vec* ⇒ *′a vec list* ⇒ *′a vec*
  where *adjuster n w* [] = 0_v *n*
  | *adjuster n w* (*u*#*us*) = −(*w* •*c* *u*)/(*u* •*c* *u*) $\cdot_v$ *u* + *adjuster n w us*

The following formulation is easier to analyze, but outputs of the subroutine should be properly reversed.

fun *gram-schmidt-sub*
  where *gram-schmidt-sub n us* [] = *us*
  | *gram-schmidt-sub n us* (*w* # *ws*) =
    gram-schmidt-sub n ((*adjuster n w us* + *w*) # *us*) *ws*

definition *gram-schmidt* :: *nat* ⇒ *′a* :: *conjugatable-field vec list* ⇒ *′a vec list*
  where *gram-schmidt n ws* = *rev* (*gram-schmidt-sub n* [] *ws*)

The following formulation requires no reversal.

fun *gram-schmidt-sub2*
  where *gram-schmidt-sub2 n us* [] = []
  | *gram-schmidt-sub2 n us* (*w* # *ws*) =
    (*let u* = *adjuster n w us* + *w in*
     u # *gram-schmidt-sub2 n* (*u* # *us*) *ws*)

lemma *gram-schmidt-sub-eq*:
  rev (*gram-schmidt-sub n us ws*) = *rev us* @ *gram-schmidt-sub2 n us ws*
  ⟨*proof*⟩

lemma *gram-schmidt-code*[*code*]:
  gram-schmidt n ws = *gram-schmidt-sub2 n* [] *ws*
  ⟨*proof*⟩

16.3 Properties of the Algorithms

locale *cof-vec-space* = *vec-space f-ty* **for**
  f-ty :: *′a* :: *conjugatable-ordered-field itself*
begin

lemma *adjuster-finsum*:

assumes *U*: *set us* ⊆ *carrier-vec n*
and *dist*: *distinct* (*us* :: *'a vec list*)
shows *adjuster n w us* = *finsum V* (*λu.* −(*w* •*c u*)/(*u* •*c u*) $\cdot_v$ *u*) (*set us*)
⟨*proof*⟩

lemma *adjuster-lincomb*:
assumes *w*: (*w* :: *'a vec*) : *carrier-vec n*
and *us*: *set* (*us* :: *'a vec list*) ⊆ *carrier-vec n*
and *dist*: *distinct us*
shows *adjuster n w us* = *lincomb* (*λu.* −(*w* •*c u*)/(*u* •*c u*)) (*set us*)
(**is** - = *lincomb ?a* -)
⟨*proof*⟩

lemma *adjuster-in-span*:
assumes *w*: (*w* :: *'a vec*) : *carrier-vec n*
and *us*: *set* (*us* :: *'a vec list*) ⊆ *carrier-vec n*
and *dist*: *distinct us*
shows *adjuster n w us* : *span* (*set us*)
⟨*proof*⟩

lemma *adjuster-carrier*[*simp*]:
assumes *w*: (*w* :: *'a vec*) : *carrier-vec n*
and *us*: *set* (*us* :: *'a vec list*) ⊆ *carrier-vec n*
and *dist*: *distinct us*
shows *adjuster n w us* : *carrier-vec n*
⟨*proof*⟩

lemma *adjust-not-in-span*:
assumes *w*[*simp*]: (*w* :: *'a vec*) : *carrier-vec n*
and *us*: *set* (*us* :: *'a vec list*) ⊆ *carrier-vec n*
and *dist*: *distinct us*
and *ind*: *w* ∉ *span* (*set us*)
shows *adjuster n w us* + *w* ∉ *span* (*set us*)
⟨*proof*⟩

lemma *adjust-not-mem*:
assumes *w*[*simp*]: (*w* :: *'a vec*) : *carrier-vec n*
and *us*: *set* (*us* :: *'a vec list*) ⊆ *carrier-vec n*
and *dist*: *distinct us*
and *ind*: *w* ∉ *span* (*set us*)
shows *adjuster n w us* + *w* ∉ *set us*
⟨*proof*⟩

lemma *adjust-in-span*:
assumes *w*[*simp*]: (*w* :: *'a vec*) : *carrier-vec n*
and *us*: *set* (*us* :: *'a vec list*) ⊆ *carrier-vec n*
and *dist*: *distinct us*
shows *adjuster n w us* + *w* : *span* (*insert w* (*set us*)) (**is** *?v* + - : *span ?U*)
⟨*proof*⟩

lemma *adjust-not-lindep*:
assumes *w*[*simp*]: (*w* :: *'a vec*) : *carrier-vec n*
and *us*: *set* (*us* :: *'a vec list*) ⊆ *carrier-vec n*
and *dist*: *distinct us*
and *wus*: *w* ∉ *span* (*set us*)
and *ind*: ~ *lin-dep* (*set us*)
shows ~ *lin-dep* (*insert* (*adjuster n w us* + *w*) (*set us*))
(**is** ~ - (*insert ?v* -))
⟨*proof*⟩

lemma *adjust-preserves-span*:
assumes *w*[*simp*]: (*w* :: *'a vec*) : *carrier-vec n*
and *us*: *set* (*us* :: *'a vec list*) ⊆ *carrier-vec n*
and *dist*: *distinct us*
shows *w* : *span* (*set us*) ⟷ *adjuster n w us* + *w* : *span* (*set us*)
(**is** - ⟷ *?v* + - : -)
⟨*proof*⟩

lemma *in-span-adjust*:
assumes *w*[*simp*]: (*w* :: *'a vec*) : *carrier-vec n*
and *us*: *set* (*us* :: *'a vec list*) ⊆ *carrier-vec n*
and *dist*: *distinct us*
shows *w* : *span* (*insert* (*adjuster n w us* + *w*) (*set us*))
(**is** - : *span* (*insert ?v* -))
⟨*proof*⟩

lemma *adjust-zero*:
assumes *U*: *set* (*us* :: *'a vec list*) ⊆ *carrier-vec n*
and *orth*: *corthogonal us*
and *w*[*simp*]: *w* : *carrier-vec n*
and *i*: *i* < *length us*
shows (*adjuster n w us* + *w*) ·*c us*!*i* = *0*
⟨*proof*⟩

lemma *adjust-nonzero*:
assumes *U*: *set* (*us* :: *'a vec list*) ⊆ *carrier-vec n*
and *dist*: *distinct us*
and *w*[*simp*]: *w* : *carrier-vec n*
and *wsU*: *w* ∉ *span* (*set us*)
shows *adjuster n w us* + *w* ≠ 0_v *n* (**is** *?a* + - ≠ -)
⟨*proof*⟩

lemma *adjust-orthogonal*:
assumes *U*: *set* (*us* :: *'a vec list*) ⊆ *carrier-vec n*
and *orth*: *corthogonal us*
and *w*[*simp*]: *w* : *carrier-vec n*
and *wsU*: *w* ∉ *span* (*set us*)
shows *corthogonal* ((*adjuster n w us* + *w*) # *us*)

(**is** *corthogonal* (*?aw* # -))
⟨*proof*⟩

lemma *gram-schmidt-sub-span*:
assumes *w*[*simp*]: *w* : *carrier-vec n*
and *us*: *set us* ⊆ *carrier-vec n*
and *dist*: *distinct us*
shows *span* (*set* ((*adjuster n w us* + *w*) # *us*)) = *span* (*set* (*w* # *us*))
(**is** *span* (*set* (*?v* # -)) = *span ?wU*)
⟨*proof*⟩

lemma *gram-schmidt-sub-result*:
assumes *gram-schmidt-sub n us ws* = *us'*
and *set ws* ⊆ *carrier-vec n*
and *set us* ⊆ *carrier-vec n*
and *distinct* (*us* @ *ws*)
and ~ *lin-dep* (*set* (*us* @ *ws*))
and *corthogonal us*
shows *set us'* ⊆ *carrier-vec n* ∧
distinct us' ∧
corthogonal us' ∧
span (*set* (*us* @ *ws*)) = *span* (*set us'*) ∧ *length us'* = *length us* + *length ws*
⟨*proof*⟩

lemma *gram-schmidt-hd* [*simp*]:
assumes [*simp*]: *w* : *carrier-vec n* **shows** *hd* (*gram-schmidt n* (*w*#*ws*)) = *w*
⟨*proof*⟩

theorem *gram-schmidt-result*:
assumes *ws*: *set ws* ⊆ *carrier-vec n*
and *dist*: *distinct ws*
and *ind*: ~ *lin-dep* (*set ws*)
and *us*: *us* = *gram-schmidt n ws*
shows *span* (*set ws*) = *span* (*set us*)
and *corthogonal us*
and *set us* ⊆ *carrier-vec n*
and *length us* = *length ws*
and *distinct us*
⟨*proof*⟩
end

end

17 Schur Decomposition

We implement Schur decomposition as an algorithm which, given a square matrix A and a list eigenvalues, computes B, P, and Q such that $A = PBQ$, B is upper-triangular and $PQ = 1$. The algorithm works is generic in the kind of field and can be applied on the rationals, the reals, and the complex numbers. The algorithm relies on the method of Gram-Schmidt to create an orthogonal basis, and on the Gauss-Jordan algorithm to find eigenvectors to a given eigenvalue.

The algorithm is a key ingredient to show that every matrix with a linear factorizable characteristic polynomial has a Jordan normal form.

A further consequence of the algorithm is that the characteristic polynomial of a block diagonal matrix is the product of the characteristic polynomials of the blocks.

theory *Schur-Decomposition*
imports
  Polynomial-Interpolation.Missing-Polynomial
  Gram-Schmidt
  Char-Poly
begin

definition *vec-inv* :: *'a::conjugatable-field vec ⇒ 'a vec*
  where *vec-inv v = 1 / (v •c v)* $\cdot_v$ *conjugate v*

lemma *vec-inv-closed*[*simp*]: *v ∈ carrier-vec n ⟹ vec-inv v ∈ carrier-vec n*
  ⟨*proof*⟩

lemma *vec-inv-dim*[*simp*]: *dim-vec (vec-inv v) = dim-vec v*
  ⟨*proof*⟩

lemma *vec-inv*[*simp*]:
  assumes *v*: *v : carrier-vec n*
    and *v0*: (*v::'a::conjugatable-ordered-field vec*) ≠ 0_v *n*
  shows *vec-inv v · v = 1*
⟨*proof*⟩

lemma *corthogonal-inv*:
  assumes *orth*: *corthogonal (vs ::'a::conjugatable-field vec list)*
    and *V*: *set vs ⊆ carrier-vec n*
  shows *inverts-mat (mat-of-rows n (map vec-inv vs)) (mat-of-cols n vs)*
   (**is** *inverts-mat ?W ?V*)
⟨*proof*⟩

definition *corthogonal-inv* :: *'a::conjugatable-field mat ⇒ 'a mat*
  where *corthogonal-inv A = mat-of-rows (dim-row A) (map vec-inv (cols A))*

definition *mat-adjoint* :: *'a :: conjugatable-field mat ⇒ 'a mat*
  where *mat-adjoint A ≡ mat-of-rows (dim-row A) (map conjugate (cols A))*

definition *corthogonal-mat* :: *'a::conjugatable-field mat ⇒ bool*
  where *corthogonal-mat A ≡*
   let B = mat-adjoint A ∗ A in
   diagonal-mat B ∧ (∀ i<dim-col A. B $$ (i,i) ≠ 0)

lemma *corthogonal-matD*[*elim*]:
assumes *orth*: *corthogonal-mat A*
and *i*: *i* < *dim-col A*
and *j*: *j* < *dim-col A*
shows (*col A i* ·*c col A j* = *0*) = (*i* ≠ *j*)
⟨*proof*⟩

lemma *corthogonal-matI*[*intro*]:
assumes (⋀*i j*. *i* < *dim-col A* ⟹ *j* < *dim-col A* ⟹ (*col A i* ·*c col A j* = *0*) = (*i* ≠ *j*))
shows *corthogonal-mat A*
⟨*proof*⟩

lemma *corthogonal-inv-result*:
assumes *o*: *corthogonal-mat* (*A*::′*a*::*conjugatable-field mat*)
shows *inverts-mat* (*corthogonal-inv A*) *A*
⟨*proof*⟩

extends a vector to a basis

definition *basis-completion* :: ′*a*::*ring-1 vec* ⇒ ′*a vec list* **where**
basis-completion v ≡ *let*
n = *dim-vec v*;
drop-index = *hd* ([*i* . *i* <− [*0*..<*n*], *v* $ *i* ≠ *0*]);
vs = [*unit-vec n i*. *i* <− [*0*..<*n*], *i* ≠ *drop-index*]
in v # *vs*

lemma (**in** *vec-space*) *basis-completion*: **fixes** *v* :: ′*a* :: *field vec*
assumes *v*: *v* ∈ *carrier-vec n*
and *v0*: *v* ≠ 0_v *n*
shows
basis (*set* (*basis-completion v*))
set (*basis-completion v*) ⊆ *carrier-vec n*
span (*set* (*basis-completion v*)) = *carrier-vec n*
distinct (*basis-completion v*)
¬ *lin-dep* (*set* (*basis-completion v*))
length (*basis-completion v*) = *n*
hd (*basis-completion v*) = *v*
⟨*proof*⟩

lemma *orthogonal-mat-of-cols*:
assumes *W*: *set ws* ⊆ *carrier-vec n*
and *orth*: *corthogonal ws*
and *len*: *length ws* = *n*
shows *corthogonal-mat* (*mat-of-cols n ws*) (**is** *corthogonal-mat ?W*)
⟨*proof*⟩

lemma *corthogonal-col-ev-0*: **fixes** *A* :: ′*a* :: *conjugatable-ordered-field mat*
assumes *A*: *A* ∈ *carrier-mat n n*

and *v*: $v \in$ *carrier-vec n*
and *v0*: $v \neq 0_v\ n$
and *eigen*[*simp*]: $A *_v v = e \cdot_v v$
and *n*: $n \neq 0$
and *hdws*: *hd ws* = *v*
and *ws*: *set ws* ⊆ *carrier-vec n corthogonal ws length ws* = *n*
defines *W* == *mat-of-cols n ws*
defines *W'* == *corthogonal-inv W*
defines *A'* == *W'* ∗ *A* ∗ *W*
shows *col A' 0* = *vec n* (λ *i*. *if i* = *0 then e else 0*)
⟨*proof*⟩

Schur decomposition

fun *schur-decomposition* :: *'a::conjugatable-field mat* ⇒ *'a list* ⇒ *'a mat* × *'a mat* × *'a mat* **where**
schur-decomposition A [] = (*A*, 1_m (*dim-row A*), 1_m (*dim-row A*))
| *schur-decomposition A* (*e* # *es*) = (*let*
n = *dim-row A*;
n1 = *n* − *1*;
v = *find-eigenvector A e*;
ws = *gram-schmidt n* (*basis-completion v*);
W = *mat-of-cols n ws*;
W' = *corthogonal-inv W*;
A' = *W'* ∗ *A* ∗ *W*;
(*A1*,*A2*,*A0*,*A3*) = *split-block A' 1 1*;
(*B*,*P*,*Q*) = *schur-decomposition A3 es*;
z-row = (0_m *1 n1*);
z-col = (0_m *n1 1*);
one-1 = 1_m *1*
in (*four-block-mat A1* (*A2* ∗ *P*) *A0 B*,
W ∗ *four-block-mat one-1 z-row z-col P*,
four-block-mat one-1 z-row z-col Q ∗ *W'*))

theorem *schur-decomposition*:
assumes *A*: (*A::'a::conjugatable-ordered-field mat*) ∈ *carrier-mat n n*
and *c*: *char-poly A* = (∏ (*e* :: *'a*) ← *es*. [:− *e*, *1*:])
and *B*: *schur-decomposition A es* = (*B*,*P*,*Q*)
shows *similar-mat-wit A B P Q* ∧ *upper-triangular B* ∧ *diag-mat B* = *es*
⟨*proof*⟩

definition *schur-upper-triangular* :: *'a::conjugatable-field mat* ⇒ *'a list* ⇒ *'a mat* **where**
schur-upper-triangular A es = (*case schur-decomposition A es of* (*B*,-,-) ⇒ *B*)

lemma *schur-upper-triangular*:
assumes *A*: (*A* :: *'a* :: *conjugatable-ordered-field mat*) ∈ *carrier-mat n n*
and *linear*: *char-poly A* = (∏ *a* ← *es*. [:− *a*, *1*:])

defines *B*: *B* ≡ *schur-upper-triangular A es*
shows *B* ∈ *carrier-mat n n upper-triangular B similar-mat A B*
⟨*proof*⟩

lemma *schur-decomposition-exists*: **assumes** *A*: *A* ∈ *carrier-mat n n*
and *linear*: *char-poly A* = (∏ (*a* :: *'a* :: *conjugatable-ordered-field*) ← *es*. [:− *a*, *1*:])
shows ∃ *B* ∈ *carrier-mat n n*. *upper-triangular B* ∧ *similar-mat A B*
⟨*proof*⟩

lemma *char-poly-0-block*: **fixes** *A* :: *'a* :: *conjugatable-ordered-field mat*
assumes *A*: *A* = *four-block-mat B C* (0_m *m n*) *D*
and *linearB*: ∃ *es*. *char-poly B* = (∏ *a* ← *es*. [:− *a*, *1*:])
and *linearD*: ∃ *es*. *char-poly D* = (∏ *a* ← *es*. [:− *a*, *1*:])
and *B*: *B* ∈ *carrier-mat n n*
and *C*: *C* ∈ *carrier-mat n m*
and *D*: *D* ∈ *carrier-mat m m*
shows *char-poly A* = *char-poly B* ∗ *char-poly D*
⟨*proof*⟩

lemma *char-poly-0-block'*: **fixes** *A* :: *'a* :: *conjugatable-ordered-field mat*
assumes *A*: *A* = *four-block-mat B* (0_m *n m*) *C D*
and *linearB*: ∃ *es*. *char-poly B* = (∏ *a* ← *es*. [:− *a*, *1*:])
and *linearD*: ∃ *es*. *char-poly D* = (∏ *a* ← *es*. [:− *a*, *1*:])
and *B*: *B* ∈ *carrier-mat n n*
and *C*: *C* ∈ *carrier-mat m n*
and *D*: *D* ∈ *carrier-mat m m*
shows *char-poly A* = *char-poly B* ∗ *char-poly D*
⟨*proof*⟩

end

18 Computing Jordan Normal Forms

theory *Jordan-Normal-Form-Existence*
imports
Jordan-Normal-Form
Column-Operations
Schur-Decomposition
begin

hide-const (**open**) *Coset.order*

We prove existence of Jordan normal forms by means of first applying Schur's algorithm to convert a matrix into upper-triangular form, and then applying the following algorithm to convert a upper-triangular matrix into a Jordan normal form. It only consists of basic row- and column-operations.

18.1 Pseudo Code Algorithm

The following algorithm is used to compute JNFs from upper-triangular matrices. It was generalized from [5, Sect. 11.1.4] where this algorithm was not explicitly specified but only applied on an example. We further introduced step 2 which does not occur in the textbook description.

1. Eliminate entries within blocks besides EV a and above EV b for $a \neq b$: for A_{ij} with EV a left of it, and EV b below of it, perform *add-col-sub-row* $(A_{ij} \ / \ (b - a))\ i\ j$. The iteration should be by first increasing j and the inner loop by decreasing i.
2. Move rows of same EV together, can only be done after 1., otherwise triangular-property is lost. Say both rows i and j ($i < j$) contain EV a, but all rows between i and j have different EV. Then perform *swap-cols-rows* $(i + 1)\ j$, *swap-cols-rows* $(i + 2)\ j$, … *swap-cols-rows* $(j - 1)\ j$. Afterwards row j will be at row $i+1$, and rows $i+1, \ldots, j-1$ will be moved to $i+2, \ldots, j$. The global iteration works by increasing j.
3. Transform each EV-block into JNF, do this for increasing upper $n \times k$ matrices, where each new column k will be treated as follows.
    a) Eliminate entries A_{ik} in rows of form $0 \ldots 0\ ev\ 1\ 0 \ldots 0\ A_{ik}$: *add-col-sub-row* $(- A_{ik})\ (i + 1)\ k$. Perform elimination by increasing i.
    b) Figure out largest JB (of $n - 1 \times n - 1$ sub-matrix) with lowest row of form $0 \ldots 0\ ev\ 0 \ldots 0\ A_{lk}$ where $A_{lk} \neq 0$, and set $x := A_{lk}$.
    c) If such a JB does not exist, continue with next column. Otherwise, eliminate all other non-zero-entries $y := A_{ik}$ via row l: *add-col-sub-row* $(y \ / \ x)\ i\ l$, *add-col-sub-row* $(y \ / \ x)\ (i - 1)\ (l - 1)$, *add-col-sub-row* $(y \ / \ x)\ (i - 2)\ (l - 2)$, … where the number of steps is determined by the size of the JB left-above of A_{ik}. Perform an iteration over i.
    d) Normalize value in row l to 1: *mult-col-div-row* $((1::'a) \ / \ x)\ k$.
    e) Move the 1 down from row l to row $k-1$: *swap-cols-rows* $(l + 1)\ k$, *swap-cols-rows* $(l + 2)\ k$, …, *swap-cols-rows* $(k - 1)\ k$.

18.2 Real Algorithm

```
fun lookup-ev :: 'a ⇒ nat ⇒ 'a mat ⇒ nat option where
  lookup-ev ev 0 A = None
| lookup-ev ev (Suc i) A = (if A $$ (i,i) = ev then Some i else lookup-ev ev i A)
```

```
function swap-cols-rows-block :: nat ⇒ nat ⇒ 'a mat ⇒ 'a mat where
  swap-cols-rows-block i j A = (if i < j then
    swap-cols-rows-block (Suc i) j (swap-cols-rows i j A) else A)
  ⟨proof⟩
termination ⟨proof⟩
```

```
fun identify-block :: 'a :: one mat ⇒ nat ⇒ nat where
  identify-block A 0 = 0
| identify-block A (Suc i) = (if A $$ (i,Suc i) = 1 then
    identify-block A i else (Suc i))
```

```
function identify-blocks-main :: 'a :: ring-1 mat ⇒ nat ⇒ (nat × nat) list ⇒ (nat
× nat) list where
  identify-blocks-main A 0 list = list
| identify-blocks-main A (Suc i-end) list = (
    let i-begin = identify-block A i-end
    in identify-blocks-main A i-begin ((i-begin, i-end) # list)
    )
  ⟨proof⟩
```

```
definition identify-blocks :: 'a :: ring-1 mat ⇒ nat ⇒ (nat × nat)list where
  identify-blocks A i = identify-blocks-main A i []
```

```
fun find-largest-block :: nat × nat ⇒ (nat × nat)list ⇒ nat × nat where
  find-largest-block block [] = block
| find-largest-block (m-start,m-end) ((i-start,i-end) # blocks) =
    (if i-end − i-start ≥ m-end − m-start then
      find-largest-block (i-start,i-end) blocks else
      find-largest-block (m-start,m-end) blocks)
```

```
fun lookup-other-ev :: 'a ⇒ nat ⇒ 'a mat ⇒ nat option where
  lookup-other-ev ev 0 A = None
| lookup-other-ev ev (Suc i) A = (if A $$ (i,i) ≠ ev then Some i else lookup-other-ev
ev i A)
```

```
partial-function (tailrec) partition-ev-blocks :: 'a mat ⇒ 'a mat list ⇒ 'a mat list
where
  [code]: partition-ev-blocks A bs = (let n = dim-row A in
    if n = 0 then bs
    else (case lookup-other-ev (A $$ (n−1, n−1)) (n−1) A of
      None ⇒ A # bs
   | Some i ⇒ case split-block A (Suc i) (Suc i) of (UL,-,-,LR) ⇒ partition-ev-blocks
UL (LR # bs)))
```

```
context
  fixes n :: nat
  and ty :: 'a :: field itself
begin
```

```
function step-1-main :: nat ⇒ nat ⇒ 'a mat ⇒ 'a mat where
  step-1-main i j A = (if j ≥ n then A else if i = 0 then step-1-main (j+1) (j+1)
A
    else let
      i' = i − 1;
      ev-left = A $$ (i',i');
      ev-below = A $$ (j,j);
      aij = A $$ (i',j);
       B = if (ev-left ≠ ev-below ∧ aij ≠ 0) then add-col-sub-row (aij / (ev-below
− ev-left)) i' j A else A
     in step-1-main i' j B)
  ⟨proof⟩
termination ⟨proof⟩

function step-2-main :: nat ⇒ 'a mat ⇒ 'a mat where
  step-2-main j A = (if j ≥ n then A
    else
      let ev = A $$ (j,j);
        B = (case lookup-ev ev j A of
          None ⇒ A
        | Some i ⇒ swap-cols-rows-block (Suc i) j A
          )
     in step-2-main (Suc j) B)
  ⟨proof⟩
termination ⟨proof⟩

fun step-3-a :: nat ⇒ nat ⇒ 'a mat ⇒ 'a mat where
  step-3-a 0 j A = A
| step-3-a (Suc i) j A = (let
    aij = A $$ (i,j);
    B = (if A $$ (i,i+1) = 1 ∧ aij ≠ 0
    then add-col-sub-row (− aij) (Suc i) j A else A)
    in step-3-a i j B)

fun step-3-c-inner-loop :: 'a ⇒ nat ⇒ nat ⇒ nat ⇒ 'a mat ⇒ 'a mat where
  step-3-c-inner-loop val l i 0 A = A
| step-3-c-inner-loop val l i (Suc k) A = step-3-c-inner-loop val (l − 1) (i − 1) k
     (add-col-sub-row val i l A)

fun step-3-c :: 'a ⇒ nat ⇒ nat ⇒ (nat × nat)list ⇒ 'a mat ⇒ 'a mat where
  step-3-c x l k [] A = A
| step-3-c x l k ((i-begin,i-end) # blocks) A = (
    let
      B = (if i-end = l then A else
        step-3-c-inner-loop (A $$ (i-end,k) / x) l i-end (Suc i-end − i-begin) A)
      in step-3-c x l k blocks B)

function step-3-main :: nat ⇒ 'a mat ⇒ 'a mat where
  step-3-main k A = (if k ≥ n then A
```

```
    else let
      B = step-3-a (k−1) k A; — 3-a
      all-blocks = identify-blocks B k;
      blocks = filter (λ block. B $$ (snd block,k) ≠ 0) all-blocks;
      F = (if blocks = [] — column k has only 0s
        then B
        else let
          (l-start,l) = find-largest-block (hd blocks) (tl blocks); — 3-b
          x = B $$ (l,k);
          C = step-3-c x l k blocks B; — 3-c
          D = mult-col-div-row (inverse x) k C; — 3-d
          E = swap-cols-rows-block (Suc l) k D — 3-e
        in E)
      in step-3-main (Suc k) F)
  ⟨proof⟩
termination ⟨proof⟩

end

definition step-1 :: 'a :: field mat ⇒ 'a mat where
  step-1 A = step-1-main (dim-row A) 0 0 A

definition step-2 :: 'a :: field mat ⇒ 'a mat where
  step-2 A = step-2-main (dim-row A) 0 A

definition step-3 :: 'a :: field mat ⇒ 'a mat where
  step-3 A = step-3-main (dim-row A) 1 A

declare swap-cols-rows-block.simps[simp del]
declare step-1-main.simps[simp del]
declare step-2-main.simps[simp del]
declare step-3-main.simps[simp del]

function jnf-vector-main :: nat ⇒ 'a :: one mat ⇒ (nat × 'a) list where
  jnf-vector-main 0 A = []
| jnf-vector-main (Suc i-end) A = (let
    i-start = identify-block A i-end
    in jnf-vector-main i-start A @ [(Suc i-end − i-start, A $$ (i-start,i-start))])
  ⟨proof⟩

definition jnf-vector :: 'a :: one mat ⇒ (nat × 'a) list where
  jnf-vector A = jnf-vector-main (dim-row A) A

definition triangular-to-jnf-vector :: 'a :: field mat ⇒ (nat × 'a) list where
  triangular-to-jnf-vector A ≡ let B = step-2 (step-1 A)
    in concat (map (jnf-vector o step-3) (partition-ev-blocks B []))
```

18.3 Preservation of Dimensions

lemma *swap-cols-rows-block-dims-main*:
dim-row (*swap-cols-rows-block i j A*) = *dim-row A* ∧ *dim-col* (*swap-cols-rows-block i j A*) = *dim-col A*
⟨*proof*⟩

lemma *swap-cols-rows-block-dims*[*simp*]:
dim-row (*swap-cols-rows-block i j A*) = *dim-row A*
dim-col (*swap-cols-rows-block i j A*) = *dim-col A*
A ∈ *carrier-mat n n* ⟹ *swap-cols-rows-block i j A* ∈ *carrier-mat n n*
⟨*proof*⟩

lemma *step-1-main-dims-main*:
dim-row (*step-1-main n i j A*) = *dim-row A* ∧ *dim-col* (*step-1-main n i j A*) = *dim-col A*
⟨*proof*⟩

lemma *step-1-main-dims*[*simp*]:
dim-row (*step-1-main n i j A*) = *dim-row A*
dim-col (*step-1-main n i j A*) = *dim-col A*
⟨*proof*⟩

lemma *step-2-main-dims-main*:
dim-row (*step-2-main n j A*) = *dim-row A* ∧ *dim-col* (*step-2-main n j A*) = *dim-col A*
⟨*proof*⟩

lemma *step-2-main-dims*[*simp*]:
dim-row (*step-2-main n j A*) = *dim-row A*
dim-col (*step-2-main n j A*) = *dim-col A*
⟨*proof*⟩

lemma *step-3-a-dims-main*:
dim-row (*step-3-a i j A*) = *dim-row A* ∧ *dim-col* (*step-3-a i j A*) = *dim-col A*
⟨*proof*⟩

lemma *step-3-a-dims*[*simp*]:
dim-row (*step-3-a i j A*) = *dim-row A*
dim-col (*step-3-a i j A*) = *dim-col A*
⟨*proof*⟩

lemma *step-3-c-inner-loop-dims-main*:
dim-row (*step-3-c-inner-loop val l i j A*) = *dim-row A* ∧ *dim-col* (*step-3-c-inner-loop val l i j A*) = *dim-col A*
⟨*proof*⟩

lemma *step-3-c-inner-loop-dims*[*simp*]:
dim-row (*step-3-c-inner-loop val l i j A*) = *dim-row A*
dim-col (*step-3-c-inner-loop val l i j A*) = *dim-col A*

⟨*proof*⟩

lemma *step-3-c-dims-main*:
dim-row (*step-3-c x l k i A*) = *dim-row A* ∧ *dim-col* (*step-3-c x l k i A*) = *dim-col A*
⟨*proof*⟩

lemma *step-3-c-dims*[*simp*]:
dim-row (*step-3-c x l k i A*) = *dim-row A*
dim-col (*step-3-c x l k i A*) = *dim-col A*
⟨*proof*⟩

lemma *step-3-main-dims-main*:
dim-row (*step-3-main n k A*) = *dim-row A* ∧ *dim-col* (*step-3-main n k A*) = *dim-col A*
⟨*proof*⟩

lemma *step-3-main-dims*[*simp*]:
dim-row (*step-3-main n j A*) = *dim-row A*
dim-col (*step-3-main n j A*) = *dim-col A*
⟨*proof*⟩

lemma *triangular-to-jnf-steps-dims*[*simp*]:
dim-row (*step-1 A*) = *dim-row A*
dim-col (*step-1 A*) = *dim-col A*
dim-row (*step-2 A*) = *dim-row A*
dim-col (*step-2 A*) = *dim-col A*
dim-row (*step-3 A*) = *dim-row A*
dim-col (*step-3 A*) = *dim-col A*
⟨*proof*⟩

18.4 Properties of Auxiliary Algorithms

lemma *lookup-ev-Some*:
lookup-ev ev j A = *Some i* ⟹
i < *j* ∧ *A* $$ (*i*,*i*) = *ev* ∧ (∀ *k*. *i* < *k* ⟶ *k* < *j* ⟶ *A* $$ (*k*,*k*) ≠ *ev*)
⟨*proof*⟩

lemma *lookup-ev-None*: *lookup-ev ev j A* = *None* ⟹ *i* < *j* ⟹ *A* $$ (*i*,*i*) ≠ *ev*
⟨*proof*⟩

lemma *swap-cols-rows-block-index*[*simp*]:
i < *dim-row A* ⟹ *i* < *dim-col A* ⟹ *j* < *dim-row A* ⟹ *j* < *dim-col A*
⟹ *low* ≤ *high* ⟹ *high* < *dim-row A* ⟹ *high* < *dim-col A*
⟹ *swap-cols-rows-block low high A* $$ (*i*,*j*) = *A* $$
(*if i* = *low then high else if i* > *low* ∧ *i* ≤ *high then i* − *1 else i*,
if j = *low then high else if j* > *low* ∧ *j* ≤ *high then j* − *1 else j*)
⟨*proof*⟩

lemma *find-largest-block-main*: **assumes** *find-largest-block block blocks* = (*m-b*, *m-e*)
shows (*m-b*, *m-e*) ∈ *insert block* (*set blocks*)
∧ (∀ *b* ∈ *insert block* (*set blocks*). *m-e* − *m-b* ≥ *snd b* − *fst b*)
⟨*proof*⟩

lemma *find-largest-block*: **assumes** *bl*: *blocks* ≠ []
and *find*: *find-largest-block* (*hd blocks*) (*tl blocks*) = (*m-begin*, *m-end*)
shows (*m-begin*,*m-end*) ∈ *set blocks*
⋀ *i-begin i-end*. (*i-begin*,*i-end*) ∈ *set blocks* ⟹ *m-end* − *m-begin* ≥ *i-end* − *i-begin*
⟨*proof*⟩

context
fixes *ev* :: *'a* :: *one*
and *A* :: *'a mat*
begin

lemma *identify-block-main*: **assumes** *identify-block A j* = *i*
shows *i* ≤ *j* ∧ (*i* = *0* ∨ *A* $$ (*i* − *1*, *i*) ≠ *1*) ∧ (∀ *k*. *i* ≤ *k* ⟶ *k* < *j* ⟶ *A* $$ (*k*, *Suc k*) = *1*)
(**is** *?P j*)
⟨*proof*⟩

lemma *identify-block-le*: *identify-block A i* ≤ *i*
⟨*proof*⟩
end

lemma *identify-block*: **assumes** *identify-block A j* = *i*
shows *i* ≤ *j*
i = *0* ∨ *A* $$ (*i* − *1*, *i*) ≠ *1*
i ≤ *k* ⟹ *k* < *j* ⟹ *A* $$ (*k*, *Suc k*) = *1*
⟨*proof*⟩

lemmas *identify-block-le'* = *identify-block*(*1*)

lemma *identify-block-le-rev*: *j* = *identify-block A i* ⟹ *j* ≤ *i*
⟨*proof*⟩

termination *identify-blocks-main* ⟨*proof*⟩

termination *jnf-vector-main* ⟨*proof*⟩

lemma *identify-blocks-main*: **assumes** (*i-start*,*i-end*) ∈ *set* (*identify-blocks-main A i list*)
and ⋀ *i-s i-e*. (*i-s*, *i-e*) ∈ *set list* ⟹ *i-s* ≤ *i-e* ∧ *i-e* < *k*
and *i* ≤ *k*

shows *i-start* ≤ *i-end* ∧ *i-end* < *k* ⟨*proof*⟩

lemma *identify-blocks*: **assumes** (*i-start*,*i-end*) ∈ *set* (*identify-blocks B k*)
shows *i-start* ≤ *i-end* ∧ *i-end* < *k*
⟨*proof*⟩

18.5 Proving Similarity

context
begin
private lemma *swap-cols-rows-block-similar*: **assumes** *A* ∈ *carrier-mat n n*
and *j* < *n* **and** *i* ≤ *j*
shows *similar-mat* (*swap-cols-rows-block i j A*) *A*
⟨*proof*⟩ **lemma** *step-1-main-similar*: *i* ≤ *j* ⟹ *A* ∈ *carrier-mat n n* ⟹ *similar-mat* (*step-1-main n i j A*) *A*
⟨*proof*⟩ **lemma** *step-2-main-similar*: *A* ∈ *carrier-mat n n* ⟹ *similar-mat* (*step-2-main n j A*) *A*
⟨*proof*⟩ **lemma** *step-3-a-similar*: *A* ∈ *carrier-mat n n* ⟹ *i* < *j* ⟹ *j* < *n* ⟹ *similar-mat* (*step-3-a i j A*) *A*
⟨*proof*⟩ **lemma** *step-3-c-inner-loop-similar*:
A ∈ *carrier-mat n n* ⟹ *l* ≠ *i* ⟹ *k* − *1* ≤ *l* ⟹ *k* − *1* ≤ *i* ⟹ *l* < *n* ⟹ *i* < *n* ⟹
similar-mat (*step-3-c-inner-loop val l i k A*) *A*
⟨*proof*⟩ **lemma** *step-3-c-similar*:
A ∈ *carrier-mat n n* ⟹ *l* < *k* ⟹ *k* < *n*
⟹ (⋀ *i-begin i-end*. (*i-begin*, *i-end*) ∈ *set blocks* ⟹ *i-end* ≤ *k* ∧ *i-end* − *i-begin* ≤ *l*)
⟹ *similar-mat* (*step-3-c x l k blocks A*) *A*
⟨*proof*⟩ **lemma** *step-3-main-similar*: *A* ∈ *carrier-mat n n* ⟹ *k* > *0* ⟹ *similar-mat* (*step-3-main n k A*) *A*
⟨*proof*⟩

lemma *step-1-similar*: *A* ∈ *carrier-mat n n* ⟹ *similar-mat* (*step-1 A*) *A*
⟨*proof*⟩

lemma *step-2-similar*: *A* ∈ *carrier-mat n n* ⟹ *similar-mat* (*step-2 A*) *A*
⟨*proof*⟩

lemma *step-3-similar*: *A* ∈ *carrier-mat n n* ⟹ *similar-mat* (*step-3 A*) *A*
⟨*proof*⟩

end

18.6 Invariants for Proving that Result is in JNF

context
fixes *n* :: *nat*
and *ty* :: *'a* :: *field itself*
begin

definition $\mathit{uppert} :: {'a}\ \mathit{mat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{bool}$ **where**
$\mathit{uppert}\ A\ i\ j \equiv j < i \longrightarrow A\ \$\$\ (i,j) = 0$

definition $\mathit{diff\text{-}ev} :: {'a}\ \mathit{mat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{bool}$ **where**
$\mathit{diff\text{-}ev}\ A\ i\ j \equiv i < j \longrightarrow A\ \$\$\ (i,i) \neq A\ \$\$\ (j,j) \longrightarrow A\ \$\$\ (i,j) = 0$

definition $\mathit{ev\text{-}blocks\text{-}part} :: \mathit{nat} \Rightarrow {'a}\ \mathit{mat} \Rightarrow \mathit{bool}$ **where**
$\mathit{ev\text{-}blocks\text{-}part}\ m\ A \equiv \forall\ i\ j\ k.\ i < j \longrightarrow j < k \longrightarrow k < m \longrightarrow A\ \$\$\ (k,k) = A\ \$\$\ (i,i) \longrightarrow A\ \$\$\ (j,j) = A\ \$\$\ (i,i)$

definition $\mathit{ev\text{-}blocks} :: {'a}\ \mathit{mat} \Rightarrow \mathit{bool}$ **where**
$\mathit{ev\text{-}blocks} \equiv \mathit{ev\text{-}blocks\text{-}part}\ n$

In step 3, there is a separation at which iteration we are. The columns left of k will be in JNF, the columns right of k or equal to k will satisfy *local.uppert*, *local.diff-ev*, and *local.ev-blocks*, and the column at k will have one of the following properties, which are ensured in the different phases of step 3.

private definition $\mathit{one\text{-}zero} :: {'a}\ \mathit{mat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{bool}$ **where**
$\mathit{one\text{-}zero}\ A\ i\ j \equiv$
$(\mathit{Suc}\ i < j \longrightarrow A\ \$\$\ (i,\mathit{Suc}\ i) = 1 \longrightarrow A\ \$\$\ (i,j) = 0) \wedge$
$(j < i \longrightarrow A\ \$\$\ (i,j) = 0) \wedge$
$(i < j \longrightarrow A\ \$\$\ (i,i) \neq A\ \$\$\ (j,j) \longrightarrow A\ \$\$\ (i,j) = 0)$

private definition $\mathit{single\text{-}non\text{-}zero} :: \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow {'a} \Rightarrow {'a}\ \mathit{mat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{bool}$ **where**
$\mathit{single\text{-}non\text{-}zero} \equiv \lambda\ l\ k\ x.\ (\lambda\ A\ i\ j.\ (i \notin \{k,l\} \longrightarrow A\ \$\$\ (i,k) = 0) \wedge A\ \$\$\ (l,k) = x)$

private definition $\mathit{single\text{-}one} :: \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow {'a}\ \mathit{mat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{bool}$ **where**
$\mathit{single\text{-}one} \equiv \lambda\ l\ k.\ (\lambda\ A\ i\ j.\ (i \notin \{k,l\} \longrightarrow A\ \$\$\ (i,k) = 0) \wedge A\ \$\$\ (l,k) = 1)$

private definition $\mathit{lower\text{-}one} :: \mathit{nat} \Rightarrow {'a}\ \mathit{mat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{bool}$ **where**
$\mathit{lower\text{-}one}\ k\ A\ i\ j = (j = k \longrightarrow$
$(A\ \$\$\ (i,j) = 0 \vee i = j \vee (A\ \$\$\ (i,j) = 1 \wedge j = \mathit{Suc}\ i \wedge A\ \$\$\ (i,i) = A\ \$\$\ (j,j))))$

definition $\mathit{jb} :: {'a}\ \mathit{mat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{bool}$ **where**
$\mathit{jb}\ A\ i\ j \equiv (\mathit{Suc}\ i = j \longrightarrow A\ \$\$\ (i,j) \in \{0,1\})$
$\wedge\ (i \neq j \longrightarrow (\mathit{Suc}\ i \neq j \vee A\ \$\$\ (i,i) \neq A\ \$\$\ (j,j)) \longrightarrow A\ \$\$\ (i,j) = 0)$

The following properties are useful to easily ensure the above invariants just from invariants of other matrices. The properties are essential in showing that the blocks identified in step 3b are the same as one would identify for the matrices in the upcoming steps 3c and 3d.

definition *same-diag* :: $'a\ mat \Rightarrow\ 'a\ mat \Rightarrow bool$ **where**
same-diag $A\ B \equiv \forall\ i < n.\ A\ \$\$\ (i,i) = B\ \$\$\ (i,i)$

private definition *same-upto* :: $nat \Rightarrow\ 'a\ mat \Rightarrow\ 'a\ mat \Rightarrow bool$ **where**
same-upto $j\ A\ B \equiv \forall\ i'\ j'.\ i' < n \longrightarrow j' < j \longrightarrow A\ \$\$\ (i',j') = B\ \$\$\ (i',j')$

Definitions stating where the properties hold

definition *inv-all* :: $('a\ mat \Rightarrow nat \Rightarrow nat \Rightarrow bool) \Rightarrow\ 'a\ mat \Rightarrow bool$ **where**
inv-all $p\ A \equiv \forall\ i\ j.\ i < n \longrightarrow j < n \longrightarrow p\ A\ i\ j$

private definition *inv-part* :: $('a\ mat \Rightarrow nat \Rightarrow nat \Rightarrow bool) \Rightarrow\ 'a\ mat \Rightarrow nat \Rightarrow nat \Rightarrow bool$ **where**
inv-part $p\ A\ m\text{-}i\ m\text{-}j \equiv \forall\ i\ j.\ i < n \longrightarrow j < n \longrightarrow j < m\text{-}j \vee j = m\text{-}j \wedge i \geq m\text{-}i \longrightarrow p\ A\ i\ j$

private definition *inv-upto* :: $('a\ mat \Rightarrow nat \Rightarrow nat \Rightarrow bool) \Rightarrow\ 'a\ mat \Rightarrow nat \Rightarrow bool$ **where**
inv-upto $p\ A\ m \equiv \forall\ i\ j.\ i < n \longrightarrow j < n \longrightarrow j < m \longrightarrow p\ A\ i\ j$

private definition *inv-from* :: $('a\ mat \Rightarrow nat \Rightarrow nat \Rightarrow bool) \Rightarrow\ 'a\ mat \Rightarrow nat \Rightarrow bool$ **where**
inv-from $p\ A\ m \equiv \forall\ i\ j.\ i < n \longrightarrow j < n \longrightarrow j > m \longrightarrow p\ A\ i\ j$

private definition *inv-at* :: $('a\ mat \Rightarrow nat \Rightarrow nat \Rightarrow bool) \Rightarrow\ 'a\ mat \Rightarrow nat \Rightarrow bool$ **where**
inv-at $p\ A\ m \equiv \forall\ i.\ i < n \longrightarrow p\ A\ i\ m$

private definition *inv-from-bot* :: $('a\ mat \Rightarrow nat \Rightarrow bool) \Rightarrow\ 'a\ mat \Rightarrow nat \Rightarrow bool$ **where**
inv-from-bot $p\ A\ mi \equiv \forall\ i.\ i \geq mi \longrightarrow i < n \longrightarrow p\ A\ i$

Auxiliary Lemmas on Handling, Comparing, and Accessing Invariants

lemma *jb-imp-uppert*: $jb\ A\ i\ j \Longrightarrow uppert\ A\ i\ j$
⟨*proof*⟩ **lemma** *ev-blocks-partD*:
ev-blocks-part $m\ A \Longrightarrow i < j \Longrightarrow j < k \Longrightarrow k < m \Longrightarrow A\ \$\$\ (k,k) = A\ \$\$\ (i,i) \Longrightarrow A\ \$\$\ (j,j) = A\ \$\$\ (i,i)$
⟨*proof*⟩ **lemma** *ev-blocks-part-leD*:
assumes *ev-blocks-part* $m\ A$
$i \leq j\ j \leq k\ k < m\ A\ \$\$\ (k,k) = A\ \$\$\ (i,i)$
shows $A\ \$\$\ (j,j) = A\ \$\$\ (i,i)$
⟨*proof*⟩ **lemma** *ev-blocks-partI*:
assumes $\bigwedge i\ j\ k.\ i < j \Longrightarrow j < k \Longrightarrow k < m \Longrightarrow A\ \$\$\ (k,k) = A\ \$\$\ (i,i) \Longrightarrow A\ \$\$\ (j,j) = A\ \$\$\ (i,i)$
shows *ev-blocks-part* $m\ A$
⟨*proof*⟩ **lemma** *ev-blocksD*:

ev-blocks $A \Longrightarrow i < j \Longrightarrow j < k \Longrightarrow k < n \Longrightarrow A \ \$\$ \ (k,k) = A \ \$\$ \ (i,i) \Longrightarrow A \ \$\$ \ (j,j) = A \ \$\$ \ (i,i)$

⟨*proof*⟩ **lemma** *ev-blocks-leD*:

ev-blocks $A \Longrightarrow i \leq j \Longrightarrow j \leq k \Longrightarrow k < n \Longrightarrow A \ \$\$ \ (k,k) = A \ \$\$ \ (i,i) \Longrightarrow A \ \$\$ \ (j,j) = A \ \$\$ \ (i,i)$

⟨*proof*⟩

lemma *inv-allD*: *inv-all* $p \ A \Longrightarrow i < n \Longrightarrow j < n \Longrightarrow p \ A \ i \ j$

⟨*proof*⟩ **lemma** *inv-allI*: **assumes** $\bigwedge i \ j. \ i < n \Longrightarrow j < n \Longrightarrow p \ A \ i \ j$

shows *inv-all* $p \ A$

⟨*proof*⟩ **lemma** *inv-partI*: **assumes** $\bigwedge i \ j. \ i < n \Longrightarrow j < n \Longrightarrow j <$ *m-j* $\vee \ j =$ *m-j* $\wedge \ i \geq$ *m-i* $\Longrightarrow p \ A \ i \ j$

shows *inv-part* $p \ A$ *m-i m-j*

⟨*proof*⟩ **lemma** *inv-partD*: **assumes** *inv-part* $p \ A$ *m-i m-j* $i < n \ j < n$

shows $j <$ *m-j* $\Longrightarrow p \ A \ i \ j$

and $j =$ *m-j* $\Longrightarrow i \geq$ *m-i* $\Longrightarrow p \ A \ i \ j$

and $j <$ *m-j* $\vee \ j =$ *m-j* $\wedge \ i \geq$ *m-i* $\Longrightarrow p \ A \ i \ j$

⟨*proof*⟩ **lemma** *inv-uptoI*: **assumes** $\bigwedge i \ j. \ i < n \Longrightarrow j < n \Longrightarrow j < m \Longrightarrow p \ A \ i \ j$

shows *inv-upto* $p \ A \ m$

⟨*proof*⟩ **lemma** *inv-uptoD*: **assumes** *inv-upto* $p \ A \ m \ i < n \ j < n \ j < m$

shows $p \ A \ i \ j$

⟨*proof*⟩ **lemma** *inv-upto-Suc*: **assumes** *inv-upto* $p \ A \ m$

and $\bigwedge i. \ i < n \Longrightarrow p \ A \ i \ m$

shows *inv-upto* $p \ A \ (Suc \ m)$

⟨*proof*⟩ **lemma** *inv-upto-mono*: **assumes** $\bigwedge i \ j. \ i < n \Longrightarrow j < k \Longrightarrow p \ A \ i \ j \Longrightarrow q \ A \ i \ j$

shows *inv-upto* $p \ A \ k \Longrightarrow$ *inv-upto* $q \ A \ k$

⟨*proof*⟩ **lemma** *inv-fromI*: **assumes** $\bigwedge i \ j. \ i < n \Longrightarrow j < n \Longrightarrow j > m \Longrightarrow p \ A \ i \ j$

shows *inv-from* $p \ A \ m$

⟨*proof*⟩ **lemma** *inv-fromD*: **assumes** *inv-from* $p \ A \ m \ i < n \ j < n \ j > m$

shows $p \ A \ i \ j$

⟨*proof*⟩ **lemma** *inv-atI*[*intro*]: **assumes** $\bigwedge i. \ i < n \Longrightarrow p \ A \ i \ m$

shows *inv-at* $p \ A \ m$

⟨*proof*⟩ **lemma** *inv-atD*: **assumes** *inv-at* $p \ A \ m \ i < n$

shows $p \ A \ i \ m$

⟨*proof*⟩ **lemma** *inv-all-imp-inv-part*: $m \ i \leq n \Longrightarrow$ *m-j* $\leq n \Longrightarrow$ *inv-all* $p \ A \Longrightarrow$ *inv-part* $p \ A$ *m-i m-j*

⟨*proof*⟩ **lemma** *inv-all-eq-inv-part*: *inv-all* $p \ A =$ *inv-part* $p \ A \ n \ n$

⟨*proof*⟩ **lemma** *inv-part-0-Suc*: *m-j* $< n \Longrightarrow$ *inv-part* $p \ A \ 0$ *m-j* $=$ *inv-part* $p \ A \ n \ (Suc$ *m-j*$)$

⟨*proof*⟩ **lemma** *inv-all-uppertD*: *inv-all uppert* $A \Longrightarrow j < i \Longrightarrow i < n \Longrightarrow A \ \$\$ \ (i,j) = 0$

⟨*proof*⟩ **lemma** *inv-all-diff-evD*: *inv-all diff-ev* $A \Longrightarrow i < j \Longrightarrow j < n$
$\Longrightarrow A \ \$\$ \ (i, i) \neq A \ \$\$ \ (j, j) \Longrightarrow A \ \$\$ \ (i,j) = 0$

⟨*proof*⟩ **lemma** *inv-all-diff-ev-uppertD*: **assumes** *inv-all diff-ev* A
inv-all uppert A
$i < n \ j < n$

and *neg*: $A \$\$ (i, i) \neq A \$\$ (j, j)$
shows $A \$\$ (i,j) = 0$
⟨*proof*⟩ **lemma** *inv-from-bot-step*: $p\ A\ i \Longrightarrow \mathit{inv\text{-}from\text{-}bot}\ p\ A\ (\mathit{Suc}\ i) \Longrightarrow \mathit{inv\text{-}from\text{-}bot}\ p\ A\ i$
⟨*proof*⟩ **lemma** *same-diag-refl*[*simp*]: $\mathit{same\text{-}diag}\ A\ A$ ⟨*proof*⟩ **lemma** *same-diag-trans*: $\mathit{same\text{-}diag}\ A\ B \Longrightarrow \mathit{same\text{-}diag}\ B\ C \Longrightarrow \mathit{same\text{-}diag}\ A\ C$
⟨*proof*⟩ **lemma** *same-diag-ev-blocks*: $\mathit{same\text{-}diag}\ A\ B \Longrightarrow \mathit{ev\text{-}blocks}\ A \Longrightarrow \mathit{ev\text{-}blocks}\ B$
⟨*proof*⟩ **lemma** *same-uptoI*[*intro*]: **assumes** $\bigwedge i'\ j'.\ i' < n \Longrightarrow j' < j \Longrightarrow A \$\$ (i',j') = B \$\$ (i',j')$
shows $\mathit{same\text{-}upto}\ j\ A\ B$
⟨*proof*⟩ **lemma** *same-uptoD*[*dest*]: **assumes** $\mathit{same\text{-}upto}\ j\ A\ B\ i' < n\ j' < j$
shows $A \$\$ (i',j') = B \$\$ (i',j')$
⟨*proof*⟩ **lemma** *same-upto-refl*[*simp*]: $\mathit{same\text{-}upto}\ j\ A\ A$ ⟨*proof*⟩ **lemma** *same-upto-trans*: $\mathit{same\text{-}upto}\ j\ A\ B \Longrightarrow \mathit{same\text{-}upto}\ j\ B\ C \Longrightarrow \mathit{same\text{-}upto}\ j\ A\ C$
⟨*proof*⟩ **lemma** *same-upto-inv-upto-jb*: $\mathit{same\text{-}upto}\ j\ A\ B \Longrightarrow \mathit{inv\text{-}upto}\ \mathit{jb}\ A\ j \Longrightarrow \mathit{inv\text{-}upto}\ \mathit{jb}\ B\ j$
⟨*proof*⟩

lemma *jb-imp-diff-ev*: $\mathit{jb}\ A\ i\ j \Longrightarrow \mathit{diff\text{-}ev}\ A\ i\ j$
⟨*proof*⟩ **lemma** *ev-blocks-diag*:
$\mathit{same\text{-}diag}\ A\ B \Longrightarrow \mathit{ev\text{-}blocks}\ B \Longrightarrow \mathit{ev\text{-}blocks}\ A$
⟨*proof*⟩ **lemma** *inv-all-imp-inv-from*: $\mathit{inv\text{-}all}\ p\ A \Longrightarrow \mathit{inv\text{-}from}\ p\ A\ k$
⟨*proof*⟩ **lemma** *inv-all-imp-inv-at*: $\mathit{inv\text{-}all}\ p\ A \Longrightarrow k < n \Longrightarrow \mathit{inv\text{-}at}\ p\ A\ k$
⟨*proof*⟩ **lemma** *inv-from-upto-at-all*:
assumes $\mathit{inv\text{-}upto}\ \mathit{jb}\ A\ k\ \mathit{inv\text{-}from}\ \mathit{diff\text{-}ev}\ A\ k\ \mathit{inv\text{-}from}\ \mathit{uppert}\ A\ k\ \mathit{inv\text{-}at}\ p\ A\ k$
and $\bigwedge i.\ i < n \Longrightarrow p\ A\ i\ k \Longrightarrow \mathit{diff\text{-}ev}\ A\ i\ k \wedge \mathit{uppert}\ A\ i\ k$
shows $\mathit{inv\text{-}all}\ \mathit{diff\text{-}ev}\ A\ \mathit{inv\text{-}all}\ \mathit{uppert}\ A$
⟨*proof*⟩ **lemma** *lower-one-diff-uppert*:
$i < n \Longrightarrow \mathit{lower\text{-}one}\ k\ B\ i\ k \Longrightarrow \mathit{diff\text{-}ev}\ B\ i\ k \wedge \mathit{uppert}\ B\ i\ k$
⟨*proof*⟩

definition *ev-block* :: $\mathit{nat} \Rightarrow {'a}\ \mathit{mat} \Rightarrow \mathit{bool}$ **where**
$\bigwedge n.\ \mathit{ev\text{-}block}\ n\ A = (\forall\ i\ j.\ i < n \longrightarrow j < n \longrightarrow A \$\$ (i,i) = A \$\$ (j,j))$

lemma *ev-blockD*: $\bigwedge n.\ \mathit{ev\text{-}block}\ n\ A \Longrightarrow i < n \Longrightarrow j < n \Longrightarrow A \$\$ (i,i) = A \$\$ (j,j)$
⟨*proof*⟩

lemma *same-diag-ev-block*: $\mathit{same\text{-}diag}\ A\ B \Longrightarrow \mathit{ev\text{-}block}\ n\ A \Longrightarrow \mathit{ev\text{-}block}\ n\ B$
⟨*proof*⟩

18.7 Alternative Characterization of *identify-blocks* in Presence of *local.ev-block*

private lemma *identify-blocks-main-iff*: **assumes** $*$: $k \leq k'$
$k' \neq k \longrightarrow k > 0 \longrightarrow A \$\$ (k - 1, k) \neq 1$ **and** $k' < n$
shows $\mathit{set}\ (\mathit{identify\text{-}blocks\text{-}main}\ A\ k\ \mathit{list}) =$
$\mathit{set}\ \mathit{list} \cup \{(i,j) \mid i\ j.\ i \leq j \wedge j < k \wedge (\forall\ l.\ i \leq l \longrightarrow l < j \longrightarrow A \$\$ (l, \mathit{Suc}\ l)$

= *1*)
∧ (*Suc j* ≠ *k'* ⟶ *A* $$ (*j*, *Suc j*) ≠ *1*) ∧ (*i* > *0* ⟶ *A* $$ (*i* − *1*, *i*) ≠ *1*)} (**is**
- = - ∪ *?ss A k*)
⟨*proof*⟩ **lemma** *identify-blocks-iff*: **assumes** *k* < *n*
shows *set* (*identify-blocks A k*) =
{(*i*,*j*) | *i j*. *i* ≤ *j* ∧ *j* < *k* ∧ (∀ *l*. *i* ≤ *l* ⟶ *l* < *j* ⟶ *A* $$ (*l*, *Suc l*) = *1*)
∧ (*Suc j* ≠ *k* ⟶ *A* $$ (*j*, *Suc j*) ≠ *1*) ∧ (*i* > *0* ⟶ *A* $$ (*i* − *1*, *i*) ≠ *1*)}
⟨*proof*⟩ **lemma** *identify-blocksD*: **assumes** *k* < *n* **and** (*i*,*j*) ∈ *set* (*identify-blocks A k*)
shows *i* ≤ *j j* < *k*
⋀ *l*. *i* ≤ *l* ⟹ *l* < *j* ⟹ *A* $$ (*l*, *Suc l*) = *1*
Suc j ≠ *k* ⟹ *A* $$ (*j*, *Suc j*) ≠ *1*
i > *0* ⟹ *A* $$ (*i* − *1*, *i* − *1*) ≠ *A* $$ (*k*,*k*) ∨ *A* $$ (*i* − *1*, *i*) ≠ *1*
⟨*proof*⟩ **lemma** *identify-blocksI*: **assumes** *inv*: *k* < *n*
i ≤ *j j* < *k* ⋀ *l*. *i* ≤ *l* ⟹ *l* < *j* ⟹ *A* $$ (*l*, *Suc l*) = *1*
Suc j ≠ *k* ⟹ *A* $$ (*j*, *Suc j*) ≠ *1* *i* > *0* ⟹ *A* $$ (*i* − *1*, *i*) ≠ *1*
shows (*i*,*j*) ∈ *set* (*identify-blocks A k*)
⟨*proof*⟩ **lemma** *identify-blocks-rev*: **assumes** *A* $$ (*i*, *Suc i*) = *0* ∧ *Suc i* < *k* ∨ *Suc i* = *k*
and *inv*: *k* < *n*
shows (*identify-block A i*, *i*) ∈ *set* (*identify-blocks A k*)
⟨*proof*⟩

18.8 Proving the Invariants

private lemma *add-col-sub-row-diag*: **assumes** *A*: *A* ∈ *carrier-mat n n*
and *ut*: *inv-all uppert A*
and *ijk*: *i* < *j j* < *n k* < *n*
shows *add-col-sub-row a i j A* $$ (*k*,*k*) = *A* $$ (*k*,*k*)
⟨*proof*⟩ **lemma** *add-col-sub-row-diff-ev-part-old*: **assumes** *A*: *A* ∈ *carrier-mat n n*
and *ij*: *i* ≤ *j i* ≠ *0 i* < *n j* < *n i'* < *n j'* < *n*
and *choice*: *j'* < *j* ∨ *j'* = *j* ∧ *i'* ≥ *i*
and *old*: *inv-part diff-ev A i j*
and *ut*: *inv-all uppert A*
shows *diff-ev* (*add-col-sub-row a* (*i* − *1*) *j A*) *i' j'*
⟨*proof*⟩ **lemma** *add-col-sub-row-uppert*: **assumes** *A* ∈ *carrier-mat n n*
and *i* < *j*
and *j* < *n*
and *inv*: *inv-all uppert* (*A* :: *'a mat*)
shows *inv-all uppert* (*add-col-sub-row a i j A*)
⟨*proof*⟩ **lemma** *step-1-main-inv*: *i* ≤ *j*
⟹ *A* ∈ *carrier-mat n n*
⟹ *inv-all uppert A*
⟹ *inv-part diff-ev A i j*
⟹ *inv-all uppert* (*step-1-main n i j A*) ∧ *inv-all diff-ev* (*step-1-main n i j A*)
⟨*proof*⟩ **lemma** *step-2-main-inv*: *A* ∈ *carrier-mat n n*
⟹ *inv-all uppert A*
⟹ *inv-all diff-ev A*
⟹ *ev-blocks-part j A*

$\Longrightarrow$ *inv-all uppert* (*step-2-main n j A*) $\wedge$ *inv-all diff-ev* (*step-2-main n j A*)
$\wedge$ *ev-blocks* (*step-2-main n j A*)
⟨*proof*⟩ **lemma** *add-col-sub-row-same-upto*: **assumes** $i < j$ $j < n$ $A \in$ *carrier-mat n n inv-upto uppert A j*
shows *same-upto j A* (*add-col-sub-row v i j A*)
⟨*proof*⟩ **lemma** *add-col-sub-row-inv-from-uppert*: **assumes** $*$: *inv-from uppert A j*
and $**$: $A \in$ *carrier-mat n n* $i < n$ $i < j$ $j < n$
shows *inv-from uppert* (*add-col-sub-row v i j A*) *j*
⟨*proof*⟩ **lemma** *step-3-a-inv*: $A \in$ *carrier-mat n n*
$\Longrightarrow i < j \Longrightarrow j < n$
$\Longrightarrow$ *inv-upto jb A j*
$\Longrightarrow$ *inv-from uppert A j*
$\Longrightarrow$ *inv-from-bot* (λ *A i*. *one-zero A i j*) *A i*
$\Longrightarrow$ *ev-block n A*
$\Longrightarrow$ *inv-from uppert* (*step-3-a i j A*) *j*
$\wedge$ *inv-upto jb* (*step-3-a i j A*) *j*
$\wedge$ *inv-at one-zero* (*step-3-a i j A*) *j* $\wedge$ *same-diag A* (*step-3-a i j A*)
⟨*proof*⟩ **lemma** *identify-block-cong*: **assumes** *su*: *same-upto k A B* **and** *kn*: $k < n$
shows $i < k \Longrightarrow$ *identify-block A i* $=$ *identify-block B i*
⟨*proof*⟩ **lemma** *identify-blocks-main-cong*:
$k < n \Longrightarrow$ *same-upto k A B* $\Longrightarrow$ *identify-blocks-main A k xs* $=$ *identify-blocks-main B k xs*
⟨*proof*⟩ **lemma** *identify-blocks-cong*:
$k < n \Longrightarrow$ *same-diag A B* $\Longrightarrow$ *same-upto k A B* $\Longrightarrow$ *identify-blocks A k* $=$ *identify-blocks B k*
⟨*proof*⟩ **lemma** *inv-from-upto-at-all-ev-block*:
assumes *jb*: *inv-upto jb A k* **and** *ut*: *inv-from uppert A k* **and** *at*: *inv-at p A k* **and** *evb*: *ev-block n A*
and *p*: $\bigwedge$ *i*. $i < n \Longrightarrow$ *p A i k* $\Longrightarrow$ *uppert A i k*
and *k*: $k < n$
shows *inv-all uppert A*
⟨*proof*⟩

For step 3c, during the inner loop, the invariants are NOT preserved. However, at the end of the inner loop, the invariants are again preserved. Therefore, for the inner loop we prove how the resulting matrix looks like in each iteration.

private lemma *step-3-c-inner-result*: **assumes** *inv*:
inv-upto jb A k
inv-from uppert A k
inv-at one-zero A k
ev-block n A
and *k*: $k < n$
and *A*: $A \in$ *carrier-mat n n*
and *lbl*: (*lb*,*l*) $\in$ *set* (*identify-blocks A k*)
and *ib-block*: (*i-begin*,*i-end*) $\in$ *set* (*identify-blocks A k*)
and *il*: *i-end* $\neq$ *l*
and *large*: *l* $-$ *lb* $\geq$ *i-end* $-$ *i-begin*

and *Alk*: *A* $$ (*l*,*k*) ≠ *0*
shows *step-3-c-inner-loop* (*A* $$ (*i-end*, *k*) / *A* $$ (*l*,*k*)) *l i-end* (*Suc i-end* − *i-begin*) *A* =
mat n n
(λ(*i*, *j*). *if* (*i*, *j*) = (*i-end*, *k*) *then 0*
else if i-begin ≤ *i* ∧ *i* ≤ *i-end* ∧ *k* < *j then A* $$ (*i*, *j*) − *A* $$ (*i-end*, *k*) / *A* $$ (*l*,*k*) ∗ *A* $$ (*l* + *i* − *i-end*, *j*)
else A $$ (*i*, *j*)) (**is** *?L* = *?R*)
⟨*proof*⟩ **lemma** *step-3-c-inv*: *A* ∈ *carrier-mat n n*
⟹ *k* < *n*
⟹ (*lb*,*l*) ∈ *set* (*identify-blocks A k*)
⟹ *inv-upto jb A k*
⟹ *inv-from uppert A k*
⟹ *inv-at one-zero A k*
⟹ *ev-block n A*
⟹ *set bs* ⊆ *set* (*identify-blocks A k*)
⟹ (⋀ *be*. *be* ∉ *snd* ‘ *set bs* ⟹ *be* ∉ {*l*,*k*} ⟹ *be* < *n* ⟹ *A* $$ (*be*,*k*) = *0*)
⟹ (⋀ *bb be*. (*bb*,*be*) ∈ *set bs* ⟹ *be* − *bb* ≤ *l* − *lb*) — largest block
⟹ *x* = *A* $$ (*l*,*k*)
⟹ *x* ≠ *0*
⟹ *inv-all uppert* (*step-3-c x l k bs A*)
∧ *same-diag A* (*step-3-c x l k bs A*)
∧ *same-upto k A* (*step-3-c x l k bs A*)
∧ *inv-at* (*single-non-zero l k x*) (*step-3-c x l k bs A*) *k*
⟨*proof*⟩

lemma *step-3-main-inv*: *A* ∈ *carrier-mat n n*
⟹ *k* > *0*
⟹ *inv-all uppert A*
⟹ *ev-block n A*
⟹ *inv-upto jb A k*
⟹ *inv-all jb* (*step-3-main n k A*) ∧ *same-diag A* (*step-3-main n k A*)
⟨*proof*⟩

lemma *step-1-2-inv*:
assumes *A*: *A* ∈ *carrier-mat n n*
and *upper-t*: *upper-triangular A*
and *Bid*: *B* = *step-2* (*step-1 A*)
shows *inv-all uppert B inv-all diff-ev B ev-blocks B*
⟨*proof*⟩

definition *inv-all′* :: (*′a mat* ⇒ *nat* ⇒ *nat* ⇒ *bool*) ⇒ *′a mat* ⇒ *bool* **where**
inv-all′ p A ≡ ∀ *i j*. *i* < *dim-row A* ⟶ *j* < *dim-row A* ⟶ *p A i j*

private lemma *lookup-other-ev-None*: **assumes** *lookup-other-ev ev k A* = *None*
and *i* < *k*
shows *A* $$ (*i*,*i*) = *ev*
⟨*proof*⟩ **lemma** *lookup-other-ev-Some*: **assumes** *lookup-other-ev ev k A* = *Some i*

shows $i < k \land A \ \$\$ \ (i,i) \neq ev \land (\forall\ j.\ i < j \land j < k \longrightarrow A \ \$\$ \ (j,j) = ev)$
⟨*proof*⟩

lemma *partition-jb*: **assumes** *A*: (*A* :: *'a mat*) ∈ *carrier-mat n n*
and *inv*: *inv-all uppert A inv-all diff-ev A ev-blocks A*
and *part*: *partition-ev-blocks A* [] = *bs*
shows *A* = *diag-block-mat bs* $\bigwedge$ *B*. *B* ∈ *set bs* ⟹ *inv-all' uppert B* ∧ *ev-block* (*dim-col B*) *B* ∧ *dim-row B* = *dim-col B*
⟨*proof*⟩

lemma *uppert-to-jb*: **assumes** *ut*: *inv-all uppert A* **and** *A* ∈ *carrier-mat n n*
shows *inv-upto jb A 1*
⟨*proof*⟩

lemma *jnf-vector*: **assumes** *A*: *A* ∈ *carrier-mat n n*
and *jb*: $\bigwedge$ *i j*. $i < n \Longrightarrow j < n \Longrightarrow$ *jb A i j*
and *evb*: *ev-block n A*
shows *jordan-matrix* (*jnf-vector A*) = (*A* :: *'a mat*)
0 ∉ *fst* ‘ *set* (*jnf-vector A*)
⟨*proof*⟩

end

lemma *triangular-to-jnf-vector*:
assumes *A*: *A* ∈ *carrier-mat n n*
and *upper-t*: *upper-triangular A*
shows *jordan-nf A* (*triangular-to-jnf-vector A*)
⟨*proof*⟩

hide-const
lookup-ev
find-largest-block
swap-cols-rows-block
identify-block
identify-blocks-main
identify-blocks
inv-all inv-all' same-diag
jb uppert diff-ev ev-blocks ev-block
step-1-main step-1
step-2-main step-2
step-3-a step-3-c step-3-c-inner-loop step-3
jnf-vector-main

18.9 Combination with Schur-decomposition

definition *jordan-nf-via-factored-charpoly* :: *'a* :: *conjugatable-ordered-field mat* ⇒ *'a list* ⇒ (*nat* × *'a*)*list*
  where *jordan-nf-via-factored-charpoly A es* =
    triangular-to-jnf-vector (*schur-upper-triangular A es*)

lemma *jordan-nf-via-factored-charpoly*: **assumes** *A*: *A* ∈ *carrier-mat n n*
  and *linear*: *char-poly A* = (∏ *a* ← *es*. [:− *a*, *1*:])
  shows *jordan-nf A* (*jordan-nf-via-factored-charpoly A es*)
⟨*proof*⟩

lemma *jordan-nf-exists*: **assumes** *A*: *A* ∈ *carrier-mat n n*
  and *linear*: *char-poly A* = (∏ (*a* :: *'a* :: *conjugatable-ordered-field*) ← *as*. [:− *a*, *1*:])
  shows ∃ *n-as*. *jordan-nf A n-as*
  ⟨*proof*⟩

lemma *jordan-nf-iff-linear-factorization*: **fixes** *A* :: *'a* :: *conjugatable-ordered-field mat*
  assumes *A*: *A* ∈ *carrier-mat n n*
  shows (∃ *n-as*. *jordan-nf A n-as*) = (∃ *as*. *char-poly A* = (∏ *a* ← *as*. [:− *a*, *1*:]))
    (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *similar-iff-same-jordan-nf*: **fixes** *A* :: *complex mat*
  assumes *A*: *A* ∈ *carrier-mat n n* **and** *B*: *B* ∈ *carrier-mat n n*
  shows *similar-mat A B* = (*jordan-nf A* = *jordan-nf B*)
⟨*proof*⟩

lemma *order-char-poly-smult*: **fixes** *A* :: *complex mat*
  assumes *A*: *A* ∈ *carrier-mat n n*
  and *k*: *k* ≠ *0*
shows *order x* (*char-poly* ($k \cdot_m A$)) = *order* (*x* / *k*) (*char-poly A*)
⟨*proof*⟩

18.10 Application for Complexity

We can estimate the complexity via the multiplicity of the eigenvalues with norm 1.

lemma *factored-char-poly-norm-bound-cof*: **assumes** *A*: *A* ∈ *carrier-mat n n*
  and *linear-factors*: *char-poly A* = (∏ (*a* :: *'a* :: {*conjugatable-ordered-field*, *real-normed-field*}) ← *as*. [:− *a*, *1*:])
  and *le-1*: ⋀ *a*. *a* ∈ *set as* ⟹ *norm a* ≤ *1*
  and *le-N*: ⋀ *a*. *a* ∈ *set as* ⟹ *norm a* = *1* ⟹ *length* (*filter* ((=) *a*) *as*) ≤ *N*
  shows ∃ *c1 c2*. ∀ *k*. *norm-bound* ($A \mathbin{\hat{}}_m k$) (*c1* + *c2* ∗ *of-nat k* ^ (*N* − *1*))
  ⟨*proof*⟩

If we have an upper triangular matrix, then EVs are exactly the entries on the diagonal. So then we don't need to explicitly compute the characteristic polynomial.

lemma *counting-ones-complexity*:
  fixes *A* :: *'a* :: *real-normed-field mat*
  assumes *A*: *A* ∈ *carrier-mat n n*
  and *upper-t*: *upper-triangular A*
  and *le-1*: ⋀ *a*. *a* ∈ *set* (*diag-mat A*) ⟹ *norm a* ≤ *1*
  and *le-N*: ⋀ *a*. *a* ∈ *set* (*diag-mat A*) ⟹ *norm a* = *1* ⟹ *length* (*filter* ((=) *a*) (*diag-mat A*)) ≤ *N*
  shows ∃ *c1 c2*. ∀ *k*. *norm-bound* (*A* $\widehat{}_m$ *k*) (*c1* + *c2* ∗ *of-nat k* ^ (*N* − *1*))
⟨*proof*⟩

If we have an upper triangular matrix A then we can compute a JNF-vector of it. If this vector does not contain entries (n, ev) with ev being larger 1, then the growth rate of A^k can be restricted by $\mathcal{O}(k^{N-1})$ where N is the maximal value for n, where $(n, |ev| = 1)$ occurs in the vector, i.e., the size of the largest Jordan Block with Eigenvalue of norm 1. This method gives a precise complexity bound.

lemma *compute-jnf-complexity*:
  assumes *A*: *A* ∈ *carrier-mat n n*
  and *upper-t*: *upper-triangular* (*A* :: *'a* :: *real-normed-field mat*)
  and *le-1*: ⋀ *n a*. (*n*,*a*) ∈ *set* (*triangular-to-jnf-vector A*) ⟹ *norm a* ≤ *1*
  and *le-N*: ⋀ *n a*. (*n*,*a*) ∈ *set* (*triangular-to-jnf-vector A*) ⟹ *norm a* = *1* ⟹ *n* ≤ *N*
  shows ∃ *c1 c2*. ∀ *k*. *norm-bound* (*A* $\widehat{}_m$ *k*) (*c1* + *c2* ∗ *of-nat k* ^ (*N* − *1*))
⟨*proof*⟩

end

19 Code Equations for All Algorithms

In this theory we load all executable algorithms, i.e., Gauss-Jordan, determinants, Jordan normal form computation, etc., and perform some basic tests.

theory *Matrix-Impl*
imports
  Matrix-IArray-Impl
  Gauss-Jordan-IArray-Impl
  Determinant-Impl
  Show-Matrix
  Shows-Literal-Matrix
  Jordan-Normal-Form-Existence
  Show.Show-Instances
begin

For determinants we require class *idom-divide*, so integers, rationals, etc.

can be used.

```
value[code] det (mat-of-rows-list 4 [[1 :: int, 4, 9, −1], [−3, −1, 5, 4], [4, 2,
0,2], [8,−9, 5,7]])
value[code] det (mat-of-rows-list 4 [[1 :: rat, 4, 9, −1], [−3, −1, 5, 4], [4, 2,
0,2], [8,−9, 5,7]])
```

Since polynomials require *field* elements to be in class *idom-divide*, the implementation of characteristic polynomials is not applicable for integer matrices, but it is for rational and real matrices.

```
value[code] char-poly (mat-of-rows-list 4 [[1 :: real, 4, 9, −1], [−3, −1, 5, 4], [4,
2, 0,2], [8,−9, 5,7]])
```

Also Jordan normal form computation requires matrices over *field* entries.

```
value[code] triangular-to-jnf-vector (mat-of-rows-list 6 [
  [3,4,1,4,7,18],
  [0,3,0,8,9,4],
  [0,0,3,2,0,4],
  [0,0,0,5,17,7],
  [0,0,0,0,5,3],
  [0,0,0,0,0,3 :: rat]])
```

Export to strings or string literals

```
value[code] show  (mat-of-rows-list 3 [[1, 4, 5], [3, 6, 8]] ∗ mat 3 4 (λ (i,j). i +
2 ∗ j))
value[code] showl (mat-of-rows-list 3 [[1, 4, 5], [3, 6, 8]] ∗ mat 3 4 (λ (i,j). i +
2 ∗ j))
```

Inverses can only be computed for matrices over fields.

```
value[code] show (mat-inverse (mat-of-rows-list 4 [[1 :: rat, 4, 9, −1], [−3, −1,
5, 4], [4, 2, 0,2], [8,−9, 5,7]]))

value[code] show (mat-inverse (mat-of-rows-list 4 [[1 :: rat, 4, 9, −1], [−3, −1,
5, 4], [−2, 3,14,3], [8,−9, 5,7]]))

end
```

20 Strassen's algorithm for matrix multiplication.

We define the algorithm for arbitrary matrices over rings, where an alignment of the dimensions to even numbers will be performed throughout the algorithm.

```
theory Strassen-Algorithm
imports
  Matrix
begin
```

With *four-block-mat* and *split-block* we can define Strassen's multiplication algorithm.

We start with a simple heuristic on when to switch to the basic algorithm.

```
definition strassen-constant :: nat where
  [code-unfold]: strassen-constant = 20
```

```
definition strassen-too-small A B ≡
  dim-row A < strassen-constant ∨
  dim-col A < strassen-constant ∨
  dim-col B < strassen-constant
```

We have to make a case analysis on whether all dimensions are even.

```
definition strassen-even A B ≡ even (dim-row A) ∧ even (dim-col A) ∧ even
(dim-col B)
```

And then we can define the algorithm.

```
function strassen-mat-mult :: 'a :: ring mat ⇒ 'a mat ⇒ 'a mat where
  strassen-mat-mult A B = (let nr = dim-row A; n = dim-col A; nc = dim-col B
in
    if strassen-too-small A B then A * B else
    if strassen-even A B then let
      nr2 = nr div 2;
      n2 = n div 2;
      nc2 = nc div 2;
      (A1,A2,A3,A4) = split-block A nr2 n2;
      (B1,B2,B3,B4) = split-block B n2 nc2;
      M1 = strassen-mat-mult (A1 + A4) (B1 + B4);
      M2 = strassen-mat-mult (A3 + A4) B1;
      M3 = strassen-mat-mult A1 (B2 − B4);
      M4 = strassen-mat-mult A4 (B3 − B1);
      M5 = strassen-mat-mult (A1 + A2) B4;
      M6 = strassen-mat-mult (A3 − A1) (B1 + B2);
      M7 = strassen-mat-mult (A2 − A4) (B3 + B4);
      C1 = M1 + M4 − M5 + M7;
      C2 = M3 + M5;
      C3 = M2 + M4;
      C4 = M1 − M2 + M3 + M6
    in four-block-mat C1 C2 C3 C4 else
    let
     nr′ = (nr div 2) * 2;
     n′ = (n div 2) * 2;
     nc′ = (nc div 2) * 2;
     (A1,A2,A3,A4) = split-block A nr′ n′;
     (B1,B2,B3,B4) = split-block B n′ nc′;
     C1 = strassen-mat-mult A1 B1 + A2 * B3;
     C2 = A1 * B2 + A2 * B4;
     C3 = A3 * B1 + A4 * B3;
     C4 = A3 * B2 + A4 * B4
```

in four-block-mat C1 C2 C3 C4)
⟨*proof*⟩

For termination, we use the following measure.

definition *strassen-measure* ≡ λ (*A*,*B*). (*dim-row A* + *dim-col A* + *dim-col B*)
+ (*dim-row A* + *dim-col A* + *dim-col B*) + (*if strassen-even A B then 0 else 1*)

lemma *strassen-measure-add*[*simp*]:
strassen-measure (*A* + *B*, *C*) = *strassen-measure* (*B*,*C*)
strassen-measure (*A*, *B* + *C*) = *strassen-measure* (*A*,*C*)
strassen-measure (*A* − *B*, *C*) = *strassen-measure* (*B*,*C*)
strassen-measure (*A*, *B* − *C*) = *strassen-measure* (*A*,*C*)
strassen-measure (− *A*, *B*) = *strassen-measure* (*A*,*B*)
strassen-measure (*A*, − *B*) = *strassen-measure* (*A*,*B*)
⟨*proof*⟩

lemma *strassen-measure-div-2*: **assumes** (*A1*, *A2*, *A3*, *A4*) = *split-block A* (*dim-row A div 2*) (*dim-col A div 2*)
(*B1*, *B2*, *B3*, *B4*) = *split-block B* (*dim-col A div 2*) (*dim-col B div 2*)
and *large*: ¬ *strassen-too-small A B*
shows
strassen-measure (*A1*,*B4*) < *strassen-measure* (*A*,*B*)
strassen-measure (*A1*,*B2*) < *strassen-measure* (*A*,*B*)
strassen-measure (*A2*,*B4*) < *strassen-measure* (*A*,*B*)
strassen-measure (*A3*,*B2*) < *strassen-measure* (*A*,*B*)
strassen-measure (*A4*,*B1*) < *strassen-measure* (*A*,*B*)
strassen-measure (*A4*,*B3*) < *strassen-measure* (*A*,*B*)
strassen-measure (*A4*,*B4*) < *strassen-measure* (*A*,*B*)
⟨*proof*⟩

lemma *strassen-measure-odd*: **assumes** (*A1*, *A2*, *A3*, *A4*) = *split-block A* ((*dim-row A div 2*) ∗ *2*) ((*dim-col A div 2*) ∗ *2*)
and (*B1*, *B2*, *B3*, *B4*) = *split-block B* ((*dim-col A div 2*) ∗ *2*) ((*dim-col B div 2*) ∗ *2*)
and *odd*: ¬ *strassen-even A B*
shows *strassen-measure* (*A1*,*B1*) < *strassen-measure* (*A*,*B*)
⟨*proof*⟩

termination ⟨*proof*⟩

lemma *strassen-mat-mult*:
dim-col A = *dim-row B* ⟹ *strassen-mat-mult A B* = *A* ∗ *B*
⟨*proof*⟩

end

21 Strassen's Algorithm as Code Equation

We replace the code-equations for matrix-multiplication by Strassen's algorithm. Note that this will strengthen the class-constraint for matrix multiplication from semirings to rings!

theory *Strassen-Algorithm-Code*
imports
  Strassen-Algorithm
begin

The aim is to replace the implementation of *?A* $*$ *?B* $\equiv$ *mat* (*dim-row* *?A*) (*dim-col* *?B*) (λ(*i*, *j*). *row* *?A* *i* $\cdot$ *col* *?B* *j*) by *strassen-mat-mult.*

We first need a copy of standard matrix multiplication to execute the base case.

definition *basic-mat-mult* = ($*$)
lemma *basic-mat-mult-code*[*code*]: *basic-mat-mult A B* = *mat* (*dim-row A*) (*dim-col B*) (λ (*i*,*j*). *row A i* $\cdot$ *col B j*)
  ⟨*proof*⟩

Next use this new matrix multiplication code within Strassen's algorithm.

lemmas *strassen-mat-mult-code*[*code*] = *strassen-mat-mult.simps*[*folded basic-mat-mult-def*]

And finally use Strassen's algorithm for implementing matrix-multiplication.

lemma *mat-mult-code*[*code*]: *A* $*$ *B* = (*if dim-col A* = *dim-row B then strassen-mat-mult A B else basic-mat-mult A B*)
  ⟨*proof*⟩

end

22 Comparison of Matrices

We use matrices over ordered semirings to again define ordered semirings. There are two instances, one for ordinary semirings (where addition is monotone w.r.t. the strict ordering in a single argument); and one for semirings like the arctic one, where addition is interpreted as maximum, and therefore monotonicity of the strict ordering in a single argument is no longer provided.

Both ordered semirings are used for checking termination proofs, where at the moment only the ordinary semirings is supported for checking complexity proofs.

theory *Matrix-Comparison*
imports
  Matrix
  Matrix.Ordered-Semiring

begin

context *ord*
begin
definition *mat-ge* :: $'a\ mat \Rightarrow 'a\ mat \Rightarrow bool$ (**infix** $\geq_m$ *50*) **where**
$A \geq_m B = (\forall\ i < dim\text{-}row\ A.\ \forall\ j < dim\text{-}col\ A.\ A\ \$\$\ (i,j) \geq B\ \$\$\ (i,j))$

lemma *mat-geI*[*intro*]: **assumes** $A \in carrier\text{-}mat\ nr\ nc$
$\bigwedge i\ j.\ i < nr \Longrightarrow j < nc \Longrightarrow A\ \$\$\ (i,j) \geq B\ \$\$\ (i,j)$
shows $A \geq_m B$
⟨*proof*⟩

lemma *mat-geD*[*dest*]: **assumes** $A \geq_m B$ **and** $i < dim\text{-}row\ A\ j < dim\text{-}col\ A$
shows $A\ \$\$\ (i,j) \geq B\ \$\$\ (i,j)$
⟨*proof*⟩

definition *mat-gt* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow nat \Rightarrow 'a\ mat \Rightarrow 'a\ mat \Rightarrow bool$ **where**
$mat\text{-}gt\ gt\ sd\ A\ B = (A \geq_m B \wedge (\exists\ i < sd.\ \exists\ j < sd.\ gt\ (A\ \$\$\ (i,j))\ (B\ \$\$\ (i,j))))$

lemma *mat-gtI*[*intro*]: **assumes** $A \geq_m B$
and $i < sd\ j < sd\ gt\ (A\ \$\$\ (i,j))\ (B\ \$\$\ (i,j))$
shows $mat\text{-}gt\ gt\ sd\ A\ B$
⟨*proof*⟩

lemma *mat-gtD*[*dest*]: **assumes** $mat\text{-}gt\ gt\ sd\ A\ B$
shows $A \geq_m B\ \exists\ i < sd.\ \exists\ j < sd.\ gt\ (A\ \$\$\ (i,j))\ (B\ \$\$\ (i,j))$
⟨*proof*⟩

definition *mat-max* :: $'a\ mat \Rightarrow 'a\ mat \Rightarrow 'a\ mat$ (max_m) **where**
$max_m\ A\ B = mat\ (dim\text{-}row\ A)\ (dim\text{-}col\ A)\ (\lambda\ ij.\ max\ (A\ \$\$\ ij)\ (B\ \$\$\ ij))$

lemma *mat-max-carrier*[*simp*]:
$max_m\ A\ B \in carrier\text{-}mat\ (dim\text{-}row\ A)\ (dim\text{-}col\ A)$
⟨*proof*⟩

lemma *mat-max-closed*[*intro*]:
$A \in carrier\text{-}mat\ nr\ nc \Longrightarrow B \in carrier\text{-}mat\ nr\ nc \Longrightarrow max_m\ A\ B \in carrier\text{-}mat\ nr\ nc$
⟨*proof*⟩

lemma *mat-max-index*:
assumes $i < dim\text{-}row\ A\ j < dim\text{-}col\ A$
shows $(mat\text{-}max\ A\ B)\ \$\$\ (i,j) = max\ (A\ \$\$\ (i,j))\ (B\ \$\$\ (i,j))$
⟨*proof*⟩

definition (**in** *zero*) *mat-default* :: $'a \Rightarrow nat \Rightarrow 'a\ mat$ ($default_m$) **where**
$default_m\ d\ n = mat\ n\ n\ (\lambda\ (i,j).\ if\ i = j\ then\ d\ else\ 0)$

lemma *mat-default-carrier*[*simp*]: $default_m\ d\ n \in carrier\text{-}mat\ n\ n$

⟨*proof*⟩
end

definition *mat-mono* :: $('a \Rightarrow bool) \Rightarrow nat \Rightarrow 'a\ mat \Rightarrow bool$
where *mat-mono* $P\ sd\ A = (\forall\ j < sd.\ \exists\ i < sd.\ P\ (A\ \$\$\ (i,j)))$

context *non-strict-order*
begin
lemma *mat-ge-trans*: **assumes** $A \geq_m B$ $B \geq_m C$
and $A \in carrier\text{-}mat\ nr\ nc$ $B \in carrier\text{-}mat\ nr\ nc$
shows $A \geq_m C$
⟨*proof*⟩

lemma *mat-ge-refl*: $A \geq_m A$
⟨*proof*⟩

lemma *mat-max-comm*: $A \in carrier\text{-}mat\ nr\ nc \Longrightarrow B \in carrier\text{-}mat\ nr\ nc \Longrightarrow max_m\ A\ B = max_m\ B\ A$
⟨*proof*⟩

lemma *mat-max-ge*: $max_m\ A\ B \geq_m A$
⟨*proof*⟩

lemma *mat-max-ge-0*: $A \in carrier\text{-}mat\ nr\ nc \Longrightarrow B \in carrier\text{-}mat\ nr\ nc \Longrightarrow A \geq_m B \Longrightarrow max_m\ A\ B = A$
⟨*proof*⟩

lemma *mat-max-mono*: $A \geq_m B \Longrightarrow$
$A \in carrier\text{-}mat\ nr\ nc \Longrightarrow B \in carrier\text{-}mat\ nr\ nc \Longrightarrow C \in carrier\text{-}mat\ nr\ nc \Longrightarrow$
$max_m\ C\ A \geq_m max_m\ C\ B$
⟨*proof*⟩
end

lemma *mat-plus-left-mono*: $A \geq_m (B :: 'a :: ordered\text{-}ab\text{-}semigroup\ mat)$
$\Longrightarrow A \in carrier\text{-}mat\ nr\ nc \Longrightarrow B \in carrier\text{-}mat\ nr\ nc \Longrightarrow C \in carrier\text{-}mat\ nr\ nc$
$\Longrightarrow A + C \geq_m B + C$
⟨*proof*⟩

lemma *mat-plus-right-mono*: $B \geq_m (C :: 'a :: ordered\text{-}ab\text{-}semigroup\ mat)$
$\Longrightarrow A \in carrier\text{-}mat\ nr\ nc \Longrightarrow B \in carrier\text{-}mat\ nr\ nc \Longrightarrow C \in carrier\text{-}mat\ nr\ nc$
$\Longrightarrow A + B \geq_m A + C$
⟨*proof*⟩

lemma *plus-mono*: $x_1 \geq (x_2 :: 'a :: ordered\text{-}ab\text{-}semigroup) \Longrightarrow$
$y_1 \geq y_2 \Longrightarrow x_1 + y_1 \geq x_2 + y_2$

⟨*proof*⟩

Since one cannot use $(\bigwedge i.\ i \in ?K \Longrightarrow ?f\ i \le ?g\ i) \Longrightarrow sum\ ?f\ ?K \le sum\ ?g\ ?K$ (it requires other class constraints like *order*), we make our own copy of this fact.

lemma *sum-mono-ge*:
assumes *ge*: $\bigwedge i.\ i{\in}K \Longrightarrow f\ (i{::}'a) \ge ((g\ i){::}('b{::}ordered\text{-}semiring\text{-}0))$
shows $(\sum i{\in}K.\ f\ i) \ge (\sum i{\in}K.\ g\ i)$
⟨*proof*⟩

lemma (**in** *one-mono-ordered-semiring-1*) *sum-mono-gt*:
assumes *le*: $\bigwedge i.\ i{\in}K \Longrightarrow f\ (i{::}'b) \ge ((g\ i){::}'a)$
and *i*: $i \in K$
and *gt*: $f\ i \succ g\ i$
and *K*: $finite\ K$
shows $(\sum i{\in}K.\ f\ i) \succ (\sum i{\in}K.\ g\ i)$
⟨*proof*⟩

lemma *scalar-left-mono*: **assumes**
$u \in carrier\text{-}vec\ n\ v \in carrier\text{-}vec\ n\ w \in carrier\text{-}vec\ n$
and $\bigwedge\ i.\ i < n \Longrightarrow u\ \$\ i \ge v\ \$\ i$
and $\bigwedge\ i.\ i < n \Longrightarrow w\ \$\ i \ge (0\ ::\ 'a\ ::\ ordered\text{-}semiring\text{-}0)$
shows $u \cdot w \ge v \cdot w$ ⟨*proof*⟩

lemma *scalar-right-mono*: **assumes**
$u \in carrier\text{-}vec\ n\ v \in carrier\text{-}vec\ n\ w \in carrier\text{-}vec\ n$
and $\bigwedge\ i.\ i < n \Longrightarrow v\ \$\ i \ge w\ \$\ i$
and $\bigwedge\ i.\ i < n \Longrightarrow u\ \$\ i \ge (0\ ::\ 'a\ ::\ ordered\text{-}semiring\text{-}0)$
shows $u \cdot v \ge u \cdot w$
⟨*proof*⟩

lemma *mat-mult-left-mono*: **assumes** *C0*: $C \ge_m 0_m\ n\ n$
and *AB*: $A \ge_m (B\ ::\ 'a\ ::\ ordered\text{-}semiring\text{-}0\ mat)$
and *carr*: $A \in carrier\text{-}mat\ n\ n\ B \in carrier\text{-}mat\ n\ n\ C \in carrier\text{-}mat\ n\ n$
shows $A * C \ge_m B * C$
⟨*proof*⟩

lemma *mat-mult-right-mono*: **assumes** *A0*: $A \ge_m 0_m\ n\ n$
and *BC*: $B \ge_m (C\ ::\ 'a\ ::\ ordered\text{-}semiring\text{-}0\ mat)$
and *carr*: $A \in carrier\text{-}mat\ n\ n\ B \in carrier\text{-}mat\ n\ n\ C \in carrier\text{-}mat\ n\ n$
shows $A * B \ge_m A * C$
⟨*proof*⟩

lemma *one-mat-ge-zero*: $(1_m\ n\ ::\ 'a\ ::\ ordered\text{-}semiring\text{-}1\ mat) \ge_m 0_m\ n\ n$
⟨*proof*⟩

context *order-pair*
begin
lemma *mat-ge-gt-trans*: **assumes** *sd*: $sd \le n$ **and** *AB*: $A \ge_m B$ **and** *BC*: *mat-gt*

gt sd B C
and *A*: *A* $\in$ *carrier-mat n n* **and** *B*: *B* $\in$ *carrier-mat n n*
shows *mat-gt gt sd A C*
⟨*proof*⟩

lemma *mat-gt-ge-trans*: **assumes** *sd*: *sd* $\leq$ *n* **and** *AB*: *mat-gt gt sd A B* **and** *BC*: *B* $\geq_m$ *C*
and *A*: *A* $\in$ *carrier-mat n n* **and** *B*: *B* $\in$ *carrier-mat n n*
shows *mat-gt gt sd A C*
⟨*proof*⟩

lemma *mat-gt-imp-mat-ge*: *mat-gt gt sd A B* $\Longrightarrow$ *A* $\geq_m$ *B*
⟨*proof*⟩

lemma *mat-gt-trans*: **assumes** *sd*: *sd* $\leq$ *n* **and** *AB*: *mat-gt gt sd A B* **and** *BC*: *mat-gt gt sd B C*
and *A*: *A* $\in$ *carrier-mat n n* **and** *B*: *B* $\in$ *carrier-mat n n*
shows *mat-gt gt sd A C*
⟨*proof*⟩

lemma *mat-default-ge-0*: $default_m$ *default n* $\geq_m$ 0_m *n n*
⟨*proof*⟩
end

definition *mat-ordered-semiring* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ (*'a* :: *ordered-semiring-1* $\Rightarrow$ *'a* $\Rightarrow$ *bool*) $\Rightarrow$ *'b* $\Rightarrow$ (*'a mat,'b*) *ordered-semiring-scheme* **where**
mat-ordered-semiring n sd gt b $\equiv$ *ring-mat TYPE('a) n* ⦇
ordered-semiring.geq = ($\geq_m$),
gt = *mat-gt gt sd*,
max = max_m,
... = *b*⦈

lemma (**in** *one-mono-ordered-semiring-1*) *mat-ordered-semiring*: **assumes** *sd-n*: *sd* $\leq$ *n*
shows *ordered-semiring*
(*mat-ordered-semiring n sd* ($\succ$) *b* :: (*'a mat,'b*) *ordered-semiring-scheme*)
(**is** *ordered-semiring ?R*)
⟨*proof*⟩

context *weak-SN-strict-mono-ordered-semiring-1*
begin

lemma *weak-mat-gt-mono*: **assumes** *sd-n*: *sd* $\leq$ *n* **and**
orient: $\bigwedge$ *A B*. *A* $\in$ *carrier-mat n n* $\Longrightarrow$ *B* $\in$ *carrier-mat n n* $\Longrightarrow$ (*A*,*B*) $\in$ *set ABs* $\Longrightarrow$ *mat-gt weak-gt sd A B*
shows $\exists$ *gt*. *SN-strict-mono-ordered-semiring-1 default gt mono* $\wedge$
($\forall$ *A B*. *A* $\in$ *carrier-mat n n* $\longrightarrow$ *B* $\in$ *carrier-mat n n* $\longrightarrow$ (*A*, *B*) $\in$ *set ABs* $\longrightarrow$ *mat-gt gt sd A B*)
⟨*proof*⟩

end

lemma *sum-mat-mono*:
 assumes *A*: $A \in$ *carrier-mat nr nc* **and** *B*: $B \in$ *carrier-mat nr nc*
 and *AB*: $A \geq_m$ (*B* :: *'a* :: *ordered-semiring-0 mat*)
 shows *sum-mat A* $\geq$ *sum-mat B*
⟨*proof*⟩

context *one-mono-ordered-semiring-1*
begin
lemma *sum-mat-mono-gt*:
 assumes *sd* $\leq$ *n*
 and *A*: $A \in$ *carrier-mat n n* **and** *B*: $B \in$ *carrier-mat n n*
 and *AB*: *mat-gt* ($\succ$) *sd A* (*B* :: *'a mat*)
 shows *sum-mat A* $\succ$ *sum-mat B*
⟨*proof*⟩

lemma *mat-plus-gt-left-mono*: **assumes** *sd-n*: *sd* $\leq$ *n* **and** *gt*: *mat-gt* ($\succ$) *sd A B*

 and *A*: $A \in$ *carrier-mat n n* **and** *B*: $B \in$ *carrier-mat n n* **and** *C*: $C \in$ *carrier-mat n n*
 shows *mat-gt* ($\succ$) *sd* (*A* + *C*) (*B* + *C*)
⟨*proof*⟩

lemma *mat-gt-ge-mono*: *sd* $\leq$ *n* $\Longrightarrow$ *mat-gt gt sd A B* $\Longrightarrow$
 mat-gt gt sd C D $\Longrightarrow$
 $A \in$ *carrier-mat n n* $\Longrightarrow$
 $B \in$ *carrier-mat n n* $\Longrightarrow$
 $C \in$ *carrier-mat n n* $\Longrightarrow$
 $D \in$ *carrier-mat n n* $\Longrightarrow$
 mat-gt gt sd (*A* + *C*) (*B* + *D*)
 ⟨*proof*⟩

lemma *mat-default-gt-mat0*: **assumes** *sd-pos*: *sd* > *0* **and** *sd-n*: *sd* $\leq$ *n*
 shows *mat-gt* ($\succ$) *sd* ($default_m$ *default n*) (0_m *n n*)
⟨*proof*⟩
end

context *SN-one-mono-ordered-semiring-1*
begin

abbreviation *mat-s* :: *'a mat* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *'a mat* $\Rightarrow$ *bool* ((- $\succ_m$ - - -) [*51*,*51*,*51*,*51*] *50*)
 where $A \succ_m$ *n sd B* $\equiv$ ($A \in$ *carrier-mat n n* $\wedge$ $B \in$ *carrier-mat n n* $\wedge$ $B \geq_m 0_m$ *n n* $\wedge$ *mat-gt* ($\succ$) *sd A B*)

lemma *mat-gt-SN*: **assumes** *sd-n*: *sd* $\leq$ *n* **shows** *SN* {(*m1*,*m2*) . *m1* $\succ_m$ *n sd m2*}

⟨*proof*⟩
end

context *SN-strict-mono-ordered-semiring-1*
begin

lemma *mat-mono*: **assumes** *sd-n*: $sd \leq n$ **and** *A*: $A \in$ *carrier-mat n n* **and** *B*: $B \in$ *carrier-mat n n* **and** *C*: $C \in$ *carrier-mat n n*
and *gt*: *mat-gt* (≻) *sd B C* **and** *gez*: $A \geq_m 0_m$ *n n* **and** *mmono*: *mat-mono mono sd A*
shows *mat-gt* (≻) *sd* $(A * B)$ $(A * C)$ (**is** *mat-gt - - ?AB ?AC*)
⟨*proof*⟩
end

definition *mat-comp-all* :: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ '*a mat* ⇒ '*a mat* ⇒ *bool*
where *mat-comp-all r A B* =
(∀ $i <$ *dim-row A*. ∀ $j <$ *dim-col A*. *r* (*A* $$ (*i*,*j*)) (*B* $$ (*i*,*j*)))

lemma *mat-comp-allI*:
assumes $A \in$ *carrier-mat nr nc* $B \in$ *carrier-mat nr nc*
and ⋀ *i j*. $i < nr \Longrightarrow j < nc \Longrightarrow$ *r* (*A* $$(*i*,*j*)) (*B* $$ (*i*,*j*))
shows *mat-comp-all r A B*
⟨*proof*⟩

lemma *mat-comp-allE*:
assumes *mat-comp-all r A B*
and $A \in$ *carrier-mat nr nc* $B \in$ *carrier-mat nr nc*
shows ⋀ *i j*. $i < nr \Longrightarrow j < nc \Longrightarrow$ *r* (*A* $$ (*i*,*j*)) (*B* $$(*i*,*j*))
⟨*proof*⟩

context *weak-SN-both-mono-ordered-semiring-1*
begin

abbreviation *weak-mat-gt-arc* :: '*a mat* ⇒ '*a mat* ⇒ *bool*
where *weak-mat-gt-arc* ≡ *mat-comp-all weak-gt*

lemma *weak-mat-gt-both-mono*:
assumes *ABs*: *set ABs* ⊆ *carrier-mat n n* × *carrier-mat n n*
and *orient*: ∀ (*A*,*B*) ∈ *set ABs*. *weak-mat-gt-arc A B*
shows ∃ *gt*. *SN-both-mono-ordered-semiring-1 default gt arc-pos* ∧
(∀ (*A*,*B*) ∈ *set ABs*. *mat-comp-all gt A B*)
⟨*proof*⟩
end

definition *mat-both-ordered-semiring* :: *nat* ⇒ ('*a* :: *ordered-semiring-1* ⇒ '*a* ⇒ *bool*) ⇒ '*b* ⇒ ('*a mat*,'*b*) *ordered-semiring-scheme* **where**
mat-both-ordered-semiring n gt b ≡ *ring-mat TYPE*('*a*) *n* ⦇
ordered-semiring.geq = *mat-ge*,
gt = *mat-comp-all gt*,

max = *mat-max*,
. . . = *b*⦈

definition *mat-arc-posI* :: (′*a* ⇒ *bool*) ⇒ ′*a mat* ⇒ *bool*
where *mat-arc-posI ap A* ≡ *ap* (*A* $$ (*0*,*0*))

context *both-mono-ordered-semiring-1*
begin

abbreviation *mat-gt-arc* :: ′*a mat* ⇒ ′*a mat* ⇒ *bool*
where *mat-gt-arc* ≡ *mat-comp-all gt*

abbreviation *mat-arc-pos* :: ′*a mat* ⇒ *bool*
where *mat-arc-pos* ≡ *mat-arc-posI arc-pos*

lemma *mat-max-id*: **fixes** *A* :: ′*a mat*
  assumes *ge*: *mat-ge A B*
  and *A*: *A* ∈ *carrier-mat nr nc*
  and *B*: *B* ∈ *carrier-mat nr nc*
  shows *mat-max A B* = *A*
  ⟨*proof*⟩

lemma *mat-gt-arc-trans*:
  assumes *A-B*: *mat-gt-arc A B*
  and *B-C*: *mat-gt-arc B C*
  and *A*: *A* ∈ *carrier-mat nr nc*
  and *B*: *B* ∈ *carrier-mat nr nc*
  and *C*: *C* ∈ *carrier-mat nr nc*
  shows *mat-gt-arc A C*
⟨*proof*⟩

lemma *mat-gt-arc-compat*:
  assumes *ge*: *mat-ge A B*
  and *gt*: *mat-gt-arc B C*
  and *A*: *A* ∈ *carrier-mat nr nc*
  and *B*: *B* ∈ *carrier-mat nr nc*
  and *C*: *C* ∈ *carrier-mat nr nc*
  shows *mat-gt-arc A C*
⟨*proof*⟩

lemma *mat-gt-arc-compat2*:
  assumes *gt*: *mat-gt-arc A B*
  and *ge*: *mat-ge B C*
  and *A*: *A* ∈ *carrier-mat nr nc*
  and *B*: *B* ∈ *carrier-mat nr nc*
  and *C*: *C* ∈ *carrier-mat nr nc*
  shows *mat-gt-arc A C*
⟨*proof*⟩

lemma *mat-gt-arc-imp-mat-ge*:
  assumes *gt*: *mat-gt-arc A B*
  and *A*: $A \in$ *carrier-mat nr nc*
  and *B*: $B \in$ *carrier-mat nr nc*
  shows *mat-ge A B*
  ⟨*proof*⟩

lemma (**in** *both-mono-ordered-semiring-1*) *mat-both-ordered-semiring*: **assumes** *n*: $n > 0$
  shows *ordered-semiring*
    (*mat-both-ordered-semiring n* ($\succ$) *b* :: (*'a mat,'b*) *ordered-semiring-scheme*)
  (**is** *ordered-semiring ?R*)
⟨*proof*⟩

lemma *mat0-leastI*:
  assumes *A*: $A \in$ *carrier-mat nr nc*
  shows *mat-gt-arc A* (0_m *nr nc*)
⟨*proof*⟩

lemma *mat0-leastII*:
  assumes *gt*: *mat-gt-arc* (0_m *nr nc*) *A*
  and *A*: $A \in$ *carrier-mat nr nc*
  shows $A = 0_m$ *nr nc*
  ⟨*proof*⟩

lemma *mat0-leastIII*:
  assumes *A*: $A \in$ *carrier-mat nr nc*
  shows *mat-ge A* ((0_m *nr nc*) :: *'a mat*)
⟨*proof*⟩

lemma *mat-max-0-id*: **fixes** *A* :: *'a mat*
  assumes *A*: $A \in$ *carrier-mat nr nc*
  shows *mat-max* (0_m *nr nc*) $A = A$
  ⟨*proof*⟩

lemma *mat-arc-pos-one*:
  assumes *n0*: $n > 0$
  shows *mat-arc-posI arc-pos* (1_m *n*)
  ⟨*proof*⟩

lemma *mat-arc-pos-zero*:
  assumes *n0*: $n > 0$
  shows $\neg$ *mat-arc-posI arc-pos* (0_m *n n*)
  ⟨*proof*⟩

lemma *mat-gt-arc-plus-mono*:
  assumes *gt1*: *mat-gt-arc A B*

and *gt2*: *mat-gt-arc C D*
and *A*: (*A*::*'a mat*) ∈ *carrier-mat nr nc*
and *B*: (*B*::*'a mat*) ∈ *carrier-mat nr nc*
and *C*: (*C*::*'a mat*) ∈ *carrier-mat nr nc*
and *D*: (*D*::*'a mat*) ∈ *carrier-mat nr nc*
shows *mat-gt-arc* (*A* + *C*) (*B* + *D*) (**is** *mat-gt-arc ?AC ?BD*)
⟨*proof*⟩

definition *vec-comp-all* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *'a vec* ⇒ *'a vec* ⇒ *bool*
where *vec-comp-all r v w* ≡ ∀ *i* < *dim-vec v*. *r* (*v* $ *i*) (*w* $ *i*)

lemma *vec-comp-allI*:
assumes ⋀*i*. *i* < *dim-vec v* ⟹ *r* (*v* $ *i*) (*w* $ *i*)
shows *vec-comp-all r v w*
⟨*proof*⟩

lemma *vec-comp-allE*:
vec-comp-all r v w ⟹ *i* < *dim-vec v* ⟹ *r* (*v* $ *i*) (*w* $ *i*)
⟨*proof*⟩

lemma *scalar-prod-left-mono*:
assumes *u*: *u* ∈ *carrier-vec n*
and *v*: *v* ∈ *carrier-vec n*
and *w*: *w* ∈ *carrier-vec n*
and *uv*: *vec-comp-all gt u v*
shows *scalar-prod u w* ≻ *scalar-prod v w*
⟨*proof*⟩

lemma *scalar-prod-right-mono*:
assumes *u*: *u* ∈ *carrier-vec n*
and *v*: *v* ∈ *carrier-vec n*
and *w*: *w* ∈ *carrier-vec n*
and *vw*: *vec-comp-all gt v w*
shows *scalar-prod u v* ≻ *scalar-prod u w*
⟨*proof*⟩

lemma *mat-gt-arc-mult-left-mono*:
assumes *gt1*: *mat-gt-arc A B*
and *A*: (*A*::*'a mat*) ∈ *carrier-mat nr n*
and *B*: (*B*::*'a mat*) ∈ *carrier-mat nr n*
and *C*: (*C*::*'a mat*) ∈ *carrier-mat n nc*
shows *mat-gt-arc* (*A* ∗ *C*) (*B* ∗ *C*) (**is** *mat-gt-arc ?AC ?BC*)
⟨*proof*⟩

lemma *mat-gt-arc-mult-right-mono*:
assumes *gt1*: *mat-gt-arc B C*
and *A*: (*A*::*'a mat*) ∈ *carrier-mat nr n*
and *B*: (*B*::*'a mat*) ∈ *carrier-mat n nc*
and *C*: (*C*::*'a mat*) ∈ *carrier-mat n nc*

shows *mat-gt-arc* $(A * B)$ $(A * C)$ (**is** *mat-gt-arc ?AB ?AC*)
⟨*proof*⟩

lemma *mat-arc-pos-plus*:
assumes *n0*: $n > 0$
and *A*: $A \in$ *carrier-mat n n*
and *B*: $B \in$ *carrier-mat n n*
and *arc-pos*: *mat-arc-pos A*
shows *mat-arc-pos* $(A + B)$
⟨*proof*⟩

lemma *scalar-prod-split-head*: **assumes**
$A \in$ *carrier-mat n n* $B \in$ *carrier-mat n n* $n > 0$
shows *row A 0* · *col B 0* = *A* \$\$ *(0,0)* ∗ *B* \$\$ *(0,0)* + (∑ *i* = *1..<n*. *A* \$\$ *(0, i)* ∗ *B* \$\$ *(i, 0)*)
⟨*proof*⟩

lemma *mat-arc-pos-mult*:
assumes *n0*: $n > 0$
and *A*: $A \in$ *carrier-mat n n*
and *B*: $B \in$ *carrier-mat n n*
and *apA*: *mat-arc-pos A*
and *apB*: *mat-arc-pos B*
shows *mat-arc-pos* $(A * B)$
⟨*proof*⟩

lemma *mat-arc-pos-mat-default*:
assumes *n0*: $n > 0$ **shows** *mat-arc-pos* (*mat-default default n*)
⟨*proof*⟩

lemma *mat-not-all-ge*:
assumes *n-pos*: $n > 0$
and *A*: $A \in$ *carrier-mat n n*
and *B*: $B \in$ *carrier-mat n n*
and *apB*: *mat-arc-pos B*
shows ∃ *C*. *C* ∈ *carrier-mat n n* ∧ *mat-ge C* (0_m *n n*) ∧ *mat-arc-pos C* ∧ ¬ *mat-ge A* $(B * C)$
⟨*proof*⟩

end

context *SN-both-mono-ordered-semiring-1*
begin

lemma *mat-gt-arc-SN*:
assumes *n-pos*: $n > 0$
shows *SN* {(*A*,*B*) ∈ *carrier-mat n n* × *carrier-mat n n*. *mat-arc-pos B* ∧ *mat-gt-arc A B*}

(**is** *SN ?rel*)
⟨*proof*⟩

end

end

23 Matrix Conversions

Essentially, the idea is to use the JNF results to estimate the growth rates of matrices. Since the results in JNF are only applicable for real normed fields, we cannot directly use them for matrices over the integers or the rational numbers. To this end, we define a homomorphism which allows us to first convert all numbers to real numbers, and then do the analysis.

theory *Ring-Hom-Matrix*
imports
Matrix
Polynomial-Interpolation.Ring-Hom
begin

locale *ord-ring-hom* = *idom-hom hom* **for**
hom :: $'a$ :: *linordered-idom* ⇒ $'b$ :: *floor-ceiling* +
assumes *hom-le*: *hom* $x \leq z \Longrightarrow x \leq$ *of-int* $\lceil z \rceil$

Now a class based variant especially for homomorphisms into the reals.

class *real-embedding* = *linordered-idom* +
fixes *real-of* :: $'a$ ⇒ *real*
assumes
real-add: *real-of* $((x :: 'a) + y)$ = *real-of* x + *real-of* y **and**
real-mult: *real-of* $(x * y)$ = *real-of* x * *real-of* y **and**
real-zero: *real-of* $0 = 0$ **and**
real-one: *real-of* $1 = 1$ **and**
real-le: *real-of* $x \leq z \Longrightarrow x \leq$ *of-int* $\lceil z \rceil$

interpretation *real-embedding*: *ord-ring-hom* (*real-of* :: $'a$ :: *real-embedding* ⇒ *real*)
⟨*proof*⟩

instantiation *real* :: *real-embedding*
begin
definition *real-of-real* :: *real* ⇒ *real* **where**
real-of-real $x = x$

instance
⟨*proof*⟩
end

instantiation *int* :: *real-embedding*
begin

definition *real-of-int* :: *int* ⇒ *real* **where**
  real-of-int x = *x*

instance
  ⟨*proof*⟩
end

lemma *real-of-rat-ineq*: **assumes** *real-of-rat x* ≤ *z*
  shows *x* ≤ *of-int* ⌈*z*⌉
⟨*proof*⟩

instantiation *rat* :: *real-embedding*
begin
definition *real-of-rat* :: *rat* ⇒ *real* **where**
  real-of-rat x = *of-rat x*

instance
  ⟨*proof*⟩
end

abbreviation *mat-real* ($mat_{\mathbb{R}}$) **where** $mat_{\mathbb{R}}$ ≡ *map-mat* (*real-of* :: *'a* :: *real-embedding* ⇒ *real*)

end

24 Derivation Bounds

Starting from this point onwards we apply the results on matrices to derive complexity bounds in IsaFoR. So, here begins the connection to the definitions and prerequisites that have originally been defined within IsaFoR.

This theory contains the notion of a derivation bound.

theory *Derivation-Bound*
imports
  Abstract−Rewriting.Abstract-Rewriting
begin

definition *deriv-bound* :: *'a rel* ⇒ *'a* ⇒ *nat* ⇒ *bool*
where
  deriv-bound r a n ⟷ ¬ (∃ *b*. (*a*, *b*) ∈ *r* ^^ *Suc n*)

lemma *deriv-boundI* [*intro?*]:
  (⋀*b m*. *n* < *m* ⟹ (*a*, *b*) ∈ *r* ^^ *m* ⟹ *False*) ⟹ *deriv-bound r a n*
  ⟨*proof*⟩

lemma *deriv-boundE*:

assumes *deriv-bound r a n*
and $(\bigwedge b\ m.\ n < m \Longrightarrow (a,\ b) \in r\ \widehat{\ }\widehat{\ }\ m \Longrightarrow \mathit{False}) \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *deriv-bound-iff*:
$\mathit{deriv\text{-}bound}\ r\ a\ n \longleftrightarrow (\forall\ b\ m.\ n < m \longrightarrow (a,\ b) \notin r\ \widehat{\ }\widehat{\ }\ m)$
⟨*proof*⟩

lemma *deriv-bound-empty* [*simp*]:
$\mathit{deriv\text{-}bound}\ \{\}\ a\ n$
⟨*proof*⟩

lemma *deriv-bound-mono*:
assumes $m \le n$ **and** *deriv-bound r a m*
shows *deriv-bound r a n*
⟨*proof*⟩

lemma *deriv-bound-image*:
assumes *b*: $\mathit{deriv\text{-}bound}\ r'\ (f\ a)\ n$
and *step*: $\bigwedge\ a\ b.\ (a,\ b) \in r \Longrightarrow (f\ a,\ f\ b) \in r'^{+}$
shows *deriv-bound r a n*
⟨*proof*⟩

lemma *deriv-bound-subset*:
assumes $r \subseteq r'^{+}$
and *b*: $\mathit{deriv\text{-}bound}\ r'\ a\ n$
shows *deriv-bound r a n*
⟨*proof*⟩

lemma *deriv-bound-SN-on*:
assumes *deriv-bound r a n*
shows $\mathit{SN\text{-}on}\ r\ \{a\}$
⟨*proof*⟩

lemma *deriv-bound-steps*:
assumes $(a,\ b) \in r\ \widehat{\ }\widehat{\ }\ n$
and *deriv-bound r a m*
shows $n \le m$
⟨*proof*⟩
end

25 Complexity Carrier

We define which properties a carrier of matrices must exhibit, so that it can be used for checking complexity proofs.

theory *Complexity-Carrier*
imports

Abstract−Rewriting.SN-Order-Carrier
Ring-Hom-Matrix
Derivation-Bound
HOL.Real
begin

class *large-real-ordered-semiring-1* = *large-ordered-semiring-1* + *real-embedding*

instance *real* :: *large-real-ordered-semiring-1* ⟨*proof*⟩
instance *int* :: *large-real-ordered-semiring-1* ⟨*proof*⟩
instance *rat* :: *large-real-ordered-semiring-1* ⟨*proof*⟩

For complexity analysis, we need a bounding function which tells us how often one can strictly decrease a value. To this end, δ-orderings are usually applied when working with the reals or rational numbers.

locale *complexity-one-mono-ordered-semiring-1* = *one-mono-ordered-semiring-1 default gt*
for *gt* :: *'a* :: *large-ordered-semiring-1* ⇒ *'a* ⇒ *bool* (**infix** ≻ *50*) **and** *default* :: *'a* +
fixes *bound* :: *'a* ⇒ *nat*
assumes *bound-mono*: ⋀ *a b*. *a* ≥ *b* ⟹ *bound a* ≥ *bound b*
and *bound-plus*: ⋀ *a b*. *bound* (*a* + *b*) ≤ *bound a* + *bound b*
and *bound-plus-of-nat*: ⋀ *a n*. *a* ≥ *0* ⟹ *bound* (*a* + *of-nat n*) = *bound a* + *bound* (*of-nat n*)
and *bound-zero*[*simp*]: *bound 0* = *0*
and *bound-one*: *bound 1* ≥ *1*
and *bound*: ⋀ *a*. *deriv-bound* {(*a*,*b*). *b* ≥ *0* ∧ *a* ≻ *b*} *a* (*bound a*)
begin

lemma *bound-linear*: ∃ *c*. ∀ *n*. *bound* (*of-nat n*) ≤ *c* ∗ *n*
⟨*proof*⟩

lemma *bound-of-nat-times*: *bound* (*of-nat n* ∗ *v*) ≤ *n* ∗ *bound v*
⟨*proof*⟩

lemma *bound-mult-of-nat*: *bound* (*a* ∗ *of-nat n*) ≤ *bound a* ∗ *bound* (*of-nat n*)
⟨*proof*⟩

lemma *bound-pow-of-nat*: *bound* (*a* ∗ *of-nat n* ^ *deg*) ≤ *bound a* ∗ *of-nat n* ^ *deg*
⟨*proof*⟩
end

end

26 Converting Arctic Numbers to Strings

We just instantiate arctic numbers in the show-class.

```
theory Show-Arctic
imports
  Abstract−Rewriting.SN-Order-Carrier
  Show.Show-Instances
begin

instantiation arctic :: show
begin

fun shows-arctic :: arctic ⇒ shows
where
  shows-arctic (Num-arc i) = shows i |
  shows-arctic (MinInfty) = shows ″−inf″

definition shows-prec (p :: nat) ai = shows-arctic ai

lemma shows-prec-artic-append [show-law-simps]:
  shows-prec p (a :: arctic) (r @ s) = shows-prec p a r @ s
  ⟨proof⟩

definition shows-list (as :: arctic list) = showsp-list shows-prec 0 as

instance
  ⟨proof⟩

end

instantiation arctic-delta :: (show) show
begin

fun shows-arctic-delta :: ′a arctic-delta ⇒ shows
where
  shows-arctic-delta (Num-arc-delta i) = shows i |
  shows-arctic-delta (MinInfty-delta)  = shows ″−inf″

definition shows-prec (d :: nat) ari = shows-arctic-delta ari

lemma shows-prec-arctic-delta-append [show-law-simps]:
  shows-prec d (a :: ′a arctic-delta) (r @ s) = shows-prec d a r @ s
  ⟨proof⟩

definition shows-list (ps :: ′a arctic-delta list) = showsp-list shows-prec 0 ps

instance
  ⟨proof⟩

end

end
```

27 Application: Complexity of Matrix Orderings

In this theory we provide various carriers which can be used for matrix interpretations.

theory *Matrix-Complexity*
imports
Matrix-Comparison
Complexity-Carrier
Show-Arctic
begin

27.1 Locales for Carriers of Matrix Interpretations and Polynomial Orders

locale *matrix-carrier* = *SN-one-mono-ordered-semiring-1 d gt*
for *gt* :: *'a* :: {*show*,*ordered-semiring-1*} ⇒ *'a* ⇒ *bool* (**infix** ≻ *50*) **and** *d* :: *'a*

locale *mono-matrix-carrier* = *complexity-one-mono-ordered-semiring-1 gt d bound*
for *gt* :: *'a* :: {*show*,*large-real-ordered-semiring-1*} ⇒ *'a* ⇒ *bool* (**infix** ≻ *50*) **and** *d* :: *'a*
and *bound* :: *'a* ⇒ *nat*
\+ **fixes** *mono* :: *'a* ⇒ *bool*
assumes *mono*: ⋀ *x y z*. *mono x* ⟹ *y* ≻ *z* ⟹ *x* ≥ *0* ⟹ *x* ∗ *y* ≻ *x* ∗ *z*

The weak version make comparison with > and then synthesize a suitable δ-ordering by choosing the least difference in the finite set of comparisons.

locale *weak-complexity-linear-poly-order-carrier* =
fixes *weak-gt* :: *'a* :: {*large-real-ordered-semiring-1*,*show*} ⇒ *'a* ⇒ *bool*
and *default* :: *'a*
and *mono* :: *'a* ⇒ *bool*
assumes *weak-gt-mono*: ∀ *x y*. (*x*,*y*) ∈ *set xys* ⟶ *weak-gt x y*
⟹ ∃ *gt bound*. *mono-matrix-carrier gt default bound mono* ∧ (∀ *x y*. (*x*,*y*) ∈ *set xys* ⟶ *gt x y*)
begin

abbreviation *weak-mat-gt* :: *nat* ⇒ *'a mat* ⇒ *'a mat* ⇒ *bool*
where *weak-mat-gt* ≡ *mat-gt weak-gt*

lemma *weak-mat-gt-mono*: **assumes** *sd-n*: *sd* ≤ *n* **and**
orient: ⋀ *A B*. *A* ∈ *carrier-mat n n* ⟹ *B* ∈ *carrier-mat n n* ⟹ (*A*,*B*) ∈ *set ABs* ⟹ *weak-mat-gt sd A B*
shows ∃ *gt bound*. *mono-matrix-carrier gt default bound mono*
∧ (∀ *A B*. *A* ∈ *carrier-mat n n* ⟶ *B* ∈ *carrier-mat n n* ⟶ (*A*, *B*) ∈ *set ABs* ⟶ *mat-gt gt sd A B*)
⟨*proof*⟩
end

sublocale *mono-matrix-carrier* ⊆ *SN-strict-mono-ordered-semiring-1 d gt mono*
⟨*proof*⟩

sublocale *mono-matrix-carrier* ⊆ *matrix-carrier* ⟨*proof*⟩

27.2 The Integers as Carrier

lemma *int-complexity*:
mono-matrix-carrier ((>) :: *int* ⇒ *int* ⇒ *bool*) *1 nat int-mono*
⟨*proof*⟩

lemma *int-weak-complexity*:
weak-complexity-linear-poly-order-carrier (>) *1 int-mono*
⟨*proof*⟩

27.3 The Rational and Real Numbers as Carrier

definition *delta-bound* :: *'a* :: *floor-ceiling* ⇒ *'a* ⇒ *nat*
where
delta-bound d x = *nat* (*ceiling* (*x* ∗ *of-int* (*ceiling* (*1* / *d*))))

lemma *delta-complexity*:
assumes *d0*: *d* > *0* **and** *d1*: *d* ≤ *def*
shows *mono-matrix-carrier* (*delta-gt d*) *def* (*delta-bound d*) *delta-mono*
⟨*proof*⟩

lemma *delta-weak-complexity-carrier*:
assumes *d0*: *def* > *0*
shows *weak-complexity-linear-poly-order-carrier* (>) *def delta-mono*
⟨*proof*⟩

27.4 The Arctic Numbers as Carrier

lemma *arctic-delta-weak-carrier*:
weak-SN-both-mono-ordered-semiring-1 weak-gt-arctic-delta 1 pos-arctic-delta ⟨*proof*⟩

lemma *arctic-weak-carrier*:
weak-SN-both-mono-ordered-semiring-1 (>) *1 pos-arctic*
⟨*proof*⟩

end

28 Matrix Kernel

We define the kernel of a matrix A and prove the following properties.

- The kernel stays invariant when multiplying A with an invertible matrix from the left.
- The dimension of the kernel stays invariant when multiplying A with an invertible matrix from the right.

- The function find-base-vectors returns a basis of the kernel if A is in row-echelon form.
- The dimension of the kernel of a block-diagonal matrix is the sum of the dimensions of the kernels of the blocks.
- There is an executable algorithm which computes the dimension of the kernel of a matrix (which just invokes Gauss-Jordan and then counts the number of pivot elements).

theory *Matrix-Kernel*
imports
  VS-Connect
  Missing-VectorSpace
  Determinant
begin

hide-const *real-vector.span*
hide-const (**open**) *Real-Vector-Spaces.span*
hide-const *real-vector.dim*
hide-const (**open**) *Real-Vector-Spaces.dim*

definition *mat-kernel* :: *'a* :: *comm-ring-1 mat* ⇒ *'a vec set* **where**
  mat-kernel A = { *v* . *v* ∈ *carrier-vec* (*dim-col A*) ∧ *A* $*_v$ *v* = 0_v (*dim-row A*)}

lemma *mat-kernelI*: **assumes** *A* ∈ *carrier-mat nr nc v* ∈ *carrier-vec nc A* $*_v$ *v* = 0_v *nr*
  shows *v* ∈ *mat-kernel A*
  ⟨*proof*⟩

lemma *mat-kernelD*: **assumes** *A* ∈ *carrier-mat nr nc v* ∈ *mat-kernel A*
  shows *v* ∈ *carrier-vec nc A* $*_v$ *v* = 0_v *nr*
  ⟨*proof*⟩

lemma *mat-kernel*: **assumes** *A* ∈ *carrier-mat nr nc*
  shows *mat-kernel A* = {*v*. *v* ∈ *carrier-vec nc* ∧ *A* $*_v$ *v* = 0_v *nr*}
  ⟨*proof*⟩

lemma *mat-kernel-carrier*:
  assumes *A* ∈ *carrier-mat nr nc* **shows** *mat-kernel A* ⊆ *carrier-vec nc*
  ⟨*proof*⟩

lemma *mat-kernel-mult-subset*: **assumes** *A*: *A* ∈ *carrier-mat nr nc*
  and *B*: *B* ∈ *carrier-mat n nr*
  shows *mat-kernel A* ⊆ *mat-kernel* (*B* ∗ *A*)
⟨*proof*⟩

lemma *mat-kernel-smult*: **assumes** *A*: *A* ∈ *carrier-mat nr nc*
  and *v*: *v* ∈ *mat-kernel A*

shows $a \cdot_v v \in$ *mat-kernel A*
⟨*proof*⟩

lemma *mat-kernel-mult-eq*: **assumes** *A*: $A \in$ *carrier-mat nr nc*
and *B*: $B \in$ *carrier-mat nr nr*
and *C*: $C \in$ *carrier-mat nr nr*
and *inv*: $C * B = 1_m$ *nr*
shows *mat-kernel* $(B * A) =$ *mat-kernel A*
⟨*proof*⟩

locale *kernel* =
fixes *nr* :: *nat*
and *nc* :: *nat*
and *A* :: *'a* :: *field mat*
assumes *A*: $A \in$ *carrier-mat nr nc*
begin

sublocale *NC*: *vec-space TYPE('a) nc* ⟨*proof*⟩

abbreviation *VK* ≡ *NC.V*⦇*carrier* := *mat-kernel A*⦈

sublocale *Ker*: *vectorspace class-ring VK*
rewrites *carrier VK* = *mat-kernel A*
and [*simp*]: *add VK* = (+)
and [*simp*]: *zero VK* = 0_v *nc*
and [*simp*]: *module.smult VK* = $(\cdot_v)$
and *carrier class-ring* = *UNIV*
and *monoid.mult class-ring* = (∗)
and *add class-ring* = (+)
and *one class-ring* = *1*
and *zero class-ring* = *0*
and *a-inv* (*class-ring* :: *'a ring*) = *uminus*
and *a-minus* (*class-ring* :: *'a ring*) = *minus*
and *pow* (*class-ring* :: *'a ring*) = (^)
and *finsum* (*class-ring* :: *'a ring*) = *sum*
and *finprod* (*class-ring* :: *'a ring*) = *prod*
and *m-inv* (*class-ring* :: *'a ring*) *x* = (*if x = 0 then div0 else inverse x*)
⟨*proof*⟩

abbreviation *basis* ≡ *Ker.basis*
abbreviation *span* ≡ *Ker.span*
abbreviation *lincomb* ≡ *Ker.lincomb*
abbreviation *dim* ≡ *Ker.dim*
abbreviation *lin-dep* ≡ *Ker.lin-dep*
abbreviation *lin-indpt* ≡ *Ker.lin-indpt*
abbreviation *gen-set* ≡ *Ker.gen-set*

lemma *finsum-same*:
assumes *f* : *S* → *mat-kernel A*

shows *finsum VK f S* = *finsum NC.V f S*
⟨*proof*⟩

lemma *lincomb-same*:
assumes *S-kernel*: $S \subseteq$ *mat-kernel A*
shows *lincomb a S* = *NC.lincomb a S*
⟨*proof*⟩

lemma *span-same*:
assumes *S-kernel*: $S \subseteq$ *mat-kernel A*
shows *span S* = *NC.span S*
⟨*proof*⟩

lemma *lindep-same*:
assumes *S-kernel*: $S \subseteq$ *mat-kernel A*
shows *Ker.lin-dep S* = *NC.lin-dep S*
⟨*proof*⟩

lemma *lincomb-index*:
assumes *i*: $i < nc$
and *Xk*: $X \subseteq$ *mat-kernel A*
shows *lincomb a X* \$ *i* = *sum* ($\lambda x.$ *a x* $*$ *x* \$ *i*) *X*
⟨*proof*⟩

end

lemma *find-base-vectors*: **assumes** *ref*: *row-echelon-form A*
and *A*: $A \in$ *carrier-mat nr nc* **shows**
set (*find-base-vectors A*) $\subseteq$ *mat-kernel A*
0_v *nc* $\notin$ *set* (*find-base-vectors A*)
kernel.basis nc A (*set* (*find-base-vectors A*))
card (*set* (*find-base-vectors A*)) = *nc* − *card* { *i*. $i < nr \wedge$ *row A i* $\neq 0_v$ *nc*}
length (*pivot-positions A*) = *card* { *i*. $i < nr \wedge$ *row A i* $\neq 0_v$ *nc*}
kernel.dim nc A = *nc* − *card* { *i*. $i < nr \wedge$ *row A i* $\neq 0_v$ *nc*}
⟨*proof*⟩

definition *kernel-dim* :: *'a* :: *field mat* ⇒ *nat* **where**
[*code del*]: *kernel-dim A* = *kernel.dim* (*dim-col A*) *A*

lemma (**in** *kernel*) *kernel-dim* [*simp*]: *kernel-dim A* = *dim* ⟨*proof*⟩

lemma *kernel-dim-code*[*code*]:
kernel-dim A = *dim-col A* − *length* (*pivot-positions* (*gauss-jordan-single A*))
⟨*proof*⟩

lemma *kernel-one-mat*: **fixes** *A* :: *'a* :: *field mat* **and** *n* :: *nat*
defines *A*: $A \equiv 1_m$ *n*

shows
kernel.dim n A = *0*
kernel.basis n A {}
⟨*proof*⟩

lemma *kernel-upper-triangular*: **assumes** *A*: *A* ∈ *carrier-mat n n*
and *ut*: *upper-triangular A* **and** *0*: *0* ∉ *set* (*diag-mat A*)
shows *kernel.dim n A* = *0 kernel.basis n A* {}
⟨*proof*⟩

lemma *kernel-basis-exists*: **assumes** *A*: *A* ∈ *carrier-mat nr nc*
shows ∃ *B*. *finite B* ∧ *kernel.basis nc A B*
⟨*proof*⟩

lemma *mat-kernel-mult-right-gen-set*: **assumes** *A*: *A* ∈ *carrier-mat nr nc*
and *B*: *B* ∈ *carrier-mat nc nc*
and *C*: *C* ∈ *carrier-mat nc nc*
and *inv*: $B * C = 1_m$ *nc*
and *gen-set*: *kernel.gen-set nc* (*A* ∗ *B*) *gen* **and** *gen*: *gen* ⊆ *mat-kernel* (*A* ∗ *B*)
shows *kernel.gen-set nc A* ((($*_v$) *B*) ‘ *gen*) ($*_v$) *B* ‘ *gen* ⊆ *mat-kernel A card* ((($*_v$) *B*) ‘ *gen*) = *card gen*
⟨*proof*⟩

lemma *mat-kernel-mult-right-basis*: **assumes** *A*: *A* ∈ *carrier-mat nr nc*
and *B*: *B* ∈ *carrier-mat nc nc*
and *C*: *C* ∈ *carrier-mat nc nc*
and *inv*: $B * C = 1_m$ *nc*
and *fin*: *finite gen*
and *basis*: *kernel.basis nc* (*A* ∗ *B*) *gen*
shows *kernel.basis nc A* ((($*_v$) *B*) ‘ *gen*)
card ((($*_v$) *B*) ‘ *gen*) = *card gen*
⟨*proof*⟩

lemma *mat-kernel-dim-mult-eq-right*: **assumes** *A*: *A* ∈ *carrier-mat nr nc*
and *B*: *B* ∈ *carrier-mat nc nc*
and *C*: *C* ∈ *carrier-mat nc nc*
and *BC*: $B * C = 1_m$ *nc*
shows *kernel.dim nc* (*A* ∗ *B*) = *kernel.dim nc A*
⟨*proof*⟩

locale *vardim* =
fixes *f-ty* :: *'a* :: *field itself*
begin

abbreviation *M* == *λk*. *module-vec TYPE*(*'a*) *k*

abbreviation *span* == *λk. LinearCombinations.module.span class-ring* (*M k*)
abbreviation *lincomb* == *λk. module.lincomb* (*M k*)
abbreviation *lin-dep* == *λk. module.lin-dep class-ring* (*M k*)
abbreviation *padr m v* == *v* $@_v$ 0_v *m*
definition *unpadr m v* == *vec* (*dim-vec v* − *m*) (*λi. v* $ *i*)
abbreviation *padl m v* == 0_v *m* $@_v$ *v*
definition *unpadl m v* == *vec* (*dim-vec v* − *m*) (*λi. v* $ (*m*+*i*))

lemma *unpadr-padr*[*simp*]: *unpadr m* (*padr m v*) = *v* ⟨*proof*⟩
lemma *unpadl-padl*[*simp*]: *unpadl m* (*padl m v*) = *v* ⟨*proof*⟩

lemma *padr-unpadr*[*simp*]: *v* : *padr m* ‘ *U* ⟹ *padr m* (*unpadr m v*) = *v* ⟨*proof*⟩
lemma *padl-unpadl*[*simp*]: *v* : *padl m* ‘ *U* ⟹ *padl m* (*unpadl m v*) = *v* ⟨*proof*⟩

lemma *padr-image*:
  assumes *U* ⊆ *carrier-vec n* **shows** *padr m* ‘ *U* ⊆ *carrier-vec* (*n* + *m*)
⟨*proof*⟩
lemma *padl-image*:
  assumes *U* ⊆ *carrier-vec n* **shows** *padl m* ‘ *U* ⊆ *carrier-vec* (*m* + *n*)
⟨*proof*⟩

lemma *padr-inj*:
  shows *inj-on* (*padr m*) (*carrier-vec n* :: *'a vec set*)
  ⟨*proof*⟩

lemma *padl-inj*:
  shows *inj-on* (*padl m*) (*carrier-vec n* :: *'a vec set*)
  ⟨*proof*⟩

lemma *lincomb-pad*:
  fixes *m n a*
  assumes *U*: (*U* :: *'a vec set*) ⊆ *carrier-vec n*
    and *finU*: *finite U*
  defines *goal pad unpad W* == *pad m* (*lincomb n a W*) = *lincomb* (*n*+*m*) (*a o unpad m*) (*pad m* ‘ *W*)
  shows *goal padr unpadr U* (**is** *?R*) **and** *goal padl unpadl U* (**is** *?L*)
⟨*proof*⟩

lemma *span-pad*:
  assumes *U*: (*U*::*'a vec set*) ⊆ *carrier-vec n*
  defines *goal pad m* == *pad m* ‘ *span n U* = *span* (*n*+*m*) (*pad m* ‘ *U*)
  shows *goal padr m goal padl m*
⟨*proof*⟩

lemma *kernel-padr*:
  assumes *aA*: *a* : *mat-kernel* (*A* :: *'a* :: *field mat*)
    and *A*: *A* : *carrier-mat nr1 nc1*
    and *B*: *B* : *carrier-mat nr1 nc2*

and *D*: *D* : *carrier-mat nr2 nc2*
shows *padr nc2 a* : *mat-kernel* (*four-block-mat A B* (0_m *nr2 nc1*) *D*) (**is** - : *mat-kernel ?ABCD*)
⟨*proof*⟩

lemma *kernel-padl*:
assumes *dD*: *d* ∈ *mat-kernel* (*D* :: *'a* :: *field mat*)
and *A*: *A* ∈ *carrier-mat nr1 nc1*
and *C*: *C* ∈ *carrier-mat nr2 nc1*
and *D*: *D* ∈ *carrier-mat nr2 nc2*
shows *padl nc1 d* ∈ *mat-kernel* (*four-block-mat A* (0_m *nr1 nc2*) *C D*) (**is** - ∈ *mat-kernel ?ABCD*)
⟨*proof*⟩

lemma *mat-kernel-split*:
assumes *A*: *A* ∈ *carrier-mat n n*
and *D*: *D* ∈ *carrier-mat m m*
and *kAD*: *k* ∈ *mat-kernel* (*four-block-mat A* (0_m *n m*) (0_m *m n*) *D*)
(**is** - ∈ *mat-kernel ?A00D*)
shows *vec-first k n* ∈ *mat-kernel A* (**is** *?a* ∈ -)
and *vec-last k m* ∈ *mat-kernel D* (**is** *?d* ∈ -)
⟨*proof*⟩

lemma *padr-padl-eq*:
assumes *v*: *v* : *carrier-vec n*
shows *padr m v* = *padl n u* ⟷ *v* = 0_v *n* ∧ *u* = 0_v *m*
⟨*proof*⟩

lemma *pad-disjoint*:
assumes *A*: *A* ⊆ *carrier-vec n* **and** *A0*: 0_v *n* ∉ *A* **and** *B*: *B* ⊆ *carrier-vec m*
shows *padr m* ' *A* ∩ *padl n* ' *B* = {} (**is** *?A* ∩ *?B* = -)
⟨*proof*⟩

lemma *padr-padl-lindep*:
assumes *A*: *A* ⊆ *carrier-vec n* **and** *liA*: ~ *lin-dep n A*
and *B*: *B* ⊆ *carrier-vec m* **and** *liB*: ~ *lin-dep m B*
shows ~ *lin-dep* (*n*+*m*) (*padr m* ' *A* ∪ *padl n* ' *B*) (**is** ~ *lin-dep* - (*?A* ∪ *?B*))
⟨*proof*⟩

end

lemma *kernel-four-block-0-mat*:
assumes *Adef*: (*A* :: *'a*::*field mat*) = *four-block-mat B* (0_m *n m*) (0_m *m n*) *D*
and *B*: *B* ∈ *carrier-mat n n*
and *D*: *D* ∈ *carrier-mat m m*
shows *kernel.dim* (*n* + *m*) *A* = *kernel.dim n B* + *kernel.dim m D*
⟨*proof*⟩

lemma *similar-mat-wit-kernel-dim*: **assumes** *A*: *A* ∈ *carrier-mat n n*
  and *wit*: *similar-mat-wit A B P Q*
  shows *kernel.dim n A* = *kernel.dim n B*
⟨*proof*⟩

end

29 Jordan Normal Form – Uniqueness

We prove that the Jordan normal form of a matrix is unique up to permutations of the blocks. We do this via generalized eigenspaces, and an algorithm which computes for each potential jordan block (ev,n), how often it occurs in any Jordan normal form.

theory *Jordan-Normal-Form-Uniqueness*
imports
  Jordan-Normal-Form
  Matrix-Kernel
begin

lemma *similar-mat-wit-char-matrix*: **assumes** *wit*: *similar-mat-wit A B P Q*
  shows *similar-mat-wit* (*char-matrix A ev*) (*char-matrix B ev*) *P Q*
⟨*proof*⟩

context fixes *ty* :: *'a* :: *field itself*
begin

lemma *dim-kernel-non-zero-jordan-block-pow*: **assumes** *a*: *a* ≠ *0*
  shows *kernel.dim n* (*jordan-block n* (*a* :: *'a*) $\widehat{}_{m}$ *k*) = *0*
  ⟨*proof*⟩

lemma *dim-kernel-zero-jordan-block-pow*:
  kernel.dim n ((*jordan-block n* (*0* :: *'a*)) $\widehat{}_{m}$ *k*) = *min k n* (**is** *kernel.dim - ?A* = *?c*)
⟨*proof*⟩

definition *dim-gen-eigenspace* :: *'a mat* ⇒ *'a* ⇒ *nat* ⇒ *nat* **where**
  dim-gen-eigenspace A ev k = *kernel-dim* ((*char-matrix A ev*) $\widehat{}_{m}$ *k*)

lemma *dim-gen-eigenspace-jordan-matrix*:
  dim-gen-eigenspace (*jordan-matrix n-as*) *ev k*
    = (∑ *n* ← *map fst* [(*n*, *e*)←*n-as* . *e* = *ev*]. *min k n*)
⟨*proof*⟩

lemma *dim-gen-eigenspace-similar*: **assumes** *sim*: *similar-mat A B*
  shows *dim-gen-eigenspace A* = *dim-gen-eigenspace B*
⟨*proof*⟩

lemma *dim-gen-eigenspace*: **assumes** *jordan-nf A n-as*
  shows *dim-gen-eigenspace A ev k*
    = (∑ n ← map fst [(n, e)←n-as . e = ev]. min k n)
⟨*proof*⟩

definition *compute-nr-of-jordan-blocks* :: *'a mat ⇒ 'a ⇒ nat ⇒ nat* **where**
  compute-nr-of-jordan-blocks A ev k = 2 ∗ dim-gen-eigenspace A ev k −
    dim-gen-eigenspace A ev (k − 1) − dim-gen-eigenspace A ev (Suc k)

This lemma finally shows uniqueness of JNFs. Take an arbitrary JNF of a matrix A, (encoded by the list of Jordan-blocks *n-as*), then then number of occurrences of each Jordan-Block in *n-as* is uniquely determined, namely by *local.compute-nr-of-jordan-blocks*. The condition $k \neq 0$ is to ensure that we do not count blocks of dimension 0.

lemma *compute-nr-of-jordan-blocks*: **assumes** *jnf*: *jordan-nf A n-as*
  and *no-0*: *k ≠ 0*
  shows *compute-nr-of-jordan-blocks A ev k = length (filter ((=) (k,ev)) n-as)*
⟨*proof*⟩

definition *compute-set-of-jordan-blocks* :: *'a mat ⇒ 'a ⇒ (nat × 'a)list* **where**
  compute-set-of-jordan-blocks A ev ≡ let
    k = Polynomial.order ev (char-poly A);
    as = map (dim-gen-eigenspace A ev) [0 ..< Suc (Suc k)];
    cards = map (λ k. (k, 2 ∗ as ! k − as ! (k − 1) − as ! Suc k)) [1 ..< Suc k]
    in map (λ (k,c). (k,ev)) (filter (λ (k,c). c ≠ 0) cards)

lemma *compute-set-of-jordan-blocks*: **assumes** *jnf*: *jordan-nf A n-as*
  shows *set (compute-set-of-jordan-blocks A ev) = set n-as ∩ UNIV × {ev}* (**is** *?C = ?N'*)
⟨*proof*⟩

lemma *jordan-nf-unique*: **assumes** *jordan-nf (A :: 'a mat) n-as* **and** *jordan-nf A m-bs*
shows *set n-as = set m-bs*
⟨*proof*⟩

One might get more fine-grained and prove the uniqueness lemma for multisets, so one takes multiplicities into account. For the moment we don't require this for complexity analysis, so it remains as future work.

end

end

30 Spectral Radius Theory

The following results show that the spectral radius characterize polynomial growth of matrix powers.

theory *Spectral-Radius*
imports
  Jordan-Normal-Form-Existence
begin

definition *spectrum A* = *Collect* (*eigenvalue A*)

lemma *spectrum-root-char-poly*: **assumes** *A*: (*A* :: *'a* :: *field mat*) ∈ *carrier-mat n n*
  shows *spectrum A* = {*k*. *poly* (*char-poly A*) *k* = *0*}
  ⟨*proof*⟩

lemma *card-finite-spectrum*: **assumes** *A*: (*A* :: *'a* :: *field mat*) ∈ *carrier-mat n n*
  shows *finite* (*spectrum A*) *card* (*spectrum A*) ≤ *n*
⟨*proof*⟩

lemma *spectrum-non-empty*: **assumes** *A*: (*A* :: *complex mat*) ∈ *carrier-mat n n*
  and *n*: *n* > *0*
  shows *spectrum A* ≠ {}
⟨*proof*⟩

definition *spectral-radius* :: *complex mat* ⇒ *real* **where**
  spectral-radius A = *Max* (*norm* ‘ *spectrum A*)

lemma *spectral-radius-mem-max*: **assumes** *A*: *A* ∈ *carrier-mat n n*
  and *n*: *n* > *0*
  shows *spectral-radius A* ∈ *norm* ‘ *spectrum A* (**is** *?one*)
  a ∈ *norm* ‘ *spectrum A* ⟹ *a* ≤ *spectral-radius A*
⟨*proof*⟩

If spectral radius is at most 1, and JNF exists, then we have polynomial growth.

lemma *spectral-radius-jnf-norm-bound-le-1*: **assumes** *A*: *A* ∈ *carrier-mat n n*
  and *sr-1*: *spectral-radius A* ≤ *1*
  and *jnf-exists*: ∃ *n-as*. *jordan-nf A n-as*
  shows ∃ *c1 c2*. ∀ *k*. *norm-bound* (*A* $\widehat{\ }_m$ *k*) (*c1* + *c2* ∗ *of-nat k* ^ (*n* − *1*))
⟨*proof*⟩

If spectral radius is smaller than 1, and JNF exists, then we have a constant bound.

lemma *spectral-radius-jnf-norm-bound-less-1*: **assumes** *A*: *A* ∈ *carrier-mat n n*
  and *sr-1*: *spectral-radius A* < *1*
  and *jnf-exists*: ∃ *n-as*. *jordan-nf A n-as*
  shows ∃ *c*. ∀ *k*. *norm-bound* (*A* $\widehat{\ }_m$ *k*) *c*
⟨*proof*⟩

If spectral radius is larger than 1, than we have exponential growth.

lemma *spectral-radius-gt-1*: **assumes** *A*: *A* ∈ *carrier-mat n n*
  and *n*: *n* > *0*

and *sr-1*: *spectral-radius A* $>$ *1*
shows ∃ *v c. v* ∈ *carrier-vec n* ∧ *norm c* $>$ *1* ∧ *v* ≠ 0_v *n* ∧ *A* $\hat{}_m$ *k* $*_v$ *v* = *c*^*k* $\cdot_v$ *v*
⟨*proof*⟩

If spectral radius is at most 1 for a complex matrix, then we have polynomial growth.

lemma *spectral-radius-jnf-norm-bound-le-1-upper-triangular*: **assumes** *A*: (*A* :: *complex mat*) ∈ *carrier-mat n n*
and *sr-1*: *spectral-radius A* ≤ *1*
shows ∃ *c1 c2*. ∀ *k. norm-bound* (*A* $\hat{}_m$ *k*) (*c1* + *c2* ∗ *of-nat k* ^ (*n* − *1*))
⟨*proof*⟩

If spectral radius is less than 1 for a complex matrix, then we have a constant bound.

lemma *spectral-radius-jnf-norm-bound-less-1-upper-triangular*: **assumes** *A*: (*A* :: *complex mat*) ∈ *carrier-mat n n*
and *sr-1*: *spectral-radius A* $<$ *1*
shows ∃ *c*. ∀ *k. norm-bound* (*A* $\hat{}_m$ *k*) *c*
⟨*proof*⟩

And we can also get a quantative approximation via the multiplicity of the eigenvalues.

lemma *spectral-radius-poly-bound*: **fixes** *A* :: *complex mat*
assumes *A*: *A* ∈ *carrier-mat n n*
and *sr-1*: *spectral-radius A* ≤ *1*
and *eq-1*: ⋀ *ev k. poly* (*char-poly A*) *ev* = *0* ⟹ *norm ev* = *1* ⟹ *Polynomial.order ev* (*char-poly A*) ≤ *d*
shows ∃ *c1 c2*. ∀ *k. norm-bound* (*A* $\hat{}_m$ *k*) (*c1* + *c2* ∗ *of-nat k* ^ (*d* − *1*))
⟨*proof*⟩

end

31 Missing Lemmas of List

theory *DL-Missing-List*
imports *Main*
begin

lemma *nth-map-zip*:
assumes *i* $<$ *length xs*
assumes *i* $<$ *length ys*
shows *map f* (*zip xs ys*) ! *i* = *f* (*xs* ! *i*, *ys* ! *i*)
⟨*proof*⟩

lemma *nth-map-zip2*:
assumes *i* $<$ *length* (*map f* (*zip xs ys*))
shows *map f* (*zip xs ys*) ! *i* = *f* (*xs* ! *i*, *ys* ! *i*)

⟨*proof*⟩

fun *find-first* **where**
find-first a [] = *undefined* |
find-first a (*x* # *xs*) = (*if x* = *a then 0 else Suc* (*find-first a xs*))

lemma *find-first-le*:
assumes *a* ∈ *set xs*
shows *find-first a xs* < *length xs*
⟨*proof*⟩

lemma *nth-find-first*:
assumes *a* ∈ *set xs*
shows *xs* ! (*find-first a xs*) = *a*
⟨*proof*⟩

lemma *find-first-unique*:
assumes *distinct xs*
and *i* < *length xs*
shows *find-first* (*xs* ! *i*) *xs* = *i*
⟨*proof*⟩

end

32 Matrix Rank

theory *DL-Rank*
imports *VS-Connect DL-Missing-List*
Determinant
Missing-VectorSpace
begin

lemma (**in** *vectorspace*) *full-dim-span*:
assumes *S* ⊆ *carrier V*
and *finite S*
and *vectorspace.dim K* (*span-vs S*) = *card S*
shows *lin-indpt S*
⟨*proof*⟩

lemma (**in** *vectorspace*) *dim-span*:
assumes *S* ⊆ *carrier V*
and *finite S*
and *maximal U* (λ*T*. *T* ⊆ *S* ∧ *lin-indpt T*)
shows *vectorspace.dim K* (*span-vs S*) = *card U*
⟨*proof*⟩

definition (**in** *vec-space*) *rank* ::′*a mat* ⇒ *nat*
where *rank A* = *vectorspace.dim class-ring* (*span-vs* (*set* (*cols A*)))

lemma (**in** *vec-space*) *rank-card-indpt*:
assumes $A \in$ *carrier-mat n nc*
assumes *maximal S* (λT. $T \subseteq$ *set* (*cols A*) $\wedge$ *lin-indpt T*)
shows *rank A* = *card S*
⟨*proof*⟩

lemma *maximal-exists-superset*:
assumes *finite S*
assumes *maxc*: $\bigwedge A$. $P\ A \Longrightarrow A \subseteq S$ **and** $P\ B$
shows $\exists A$. *finite A* $\wedge$ *maximal A P* $\wedge$ $B \subseteq A$
⟨*proof*⟩

lemma (**in** *vec-space*) *rank-ge-card-indpt*:
assumes $A \in$ *carrier-mat n nc*
assumes $U \subseteq$ *set* (*cols A*)
assumes *lin-indpt U*
shows *rank A* $\geq$ *card U*
⟨*proof*⟩

lemma (**in** *vec-space*) *lin-indpt-full-rank*:
assumes $A \in$ *carrier-mat n nc*
assumes *distinct* (*cols A*)
assumes *lin-indpt* (*set* (*cols A*))
shows *rank A* = *nc*
⟨*proof*⟩

lemma (**in** *vec-space*) *rank-le-nc*:
assumes $A \in$ *carrier-mat n nc*
shows *rank A* $\leq$ *nc*
⟨*proof*⟩

lemma (**in** *vec-space*) *full-rank-lin-indpt*:
assumes $A \in$ *carrier-mat n nc*
assumes *rank A* = *nc*
assumes *distinct* (*cols A*)
shows *lin-indpt* (*set* (*cols A*))
⟨*proof*⟩

lemma (**in** *vec-space*) *mat-mult-eq-lincomb*:
assumes $A \in$ *carrier-mat n nc*
assumes *distinct* (*cols A*)
shows $A *_v$ (*vec nc* (λi. *a* (*col A i*))) = *lincomb a* (*set* (*cols A*))
⟨*proof*⟩

lemma (**in** *vec-space*) *lincomb-eq-mat-mult*:
assumes $A \in$ *carrier-mat n nc*
assumes $v \in$ *carrier-vec nc*

assumes *distinct* (*cols A*)
shows *lincomb* (*λa. v* $ *find-first a* (*cols A*)) (*set* (*cols A*)) = (*A* $*_v$ *v*)
⟨*proof*⟩

lemma (**in** *vec-space*) *lin-depI*:
assumes *A* ∈ *carrier-mat n nc*
assumes *v* ∈ *carrier-vec nc v* ≠ 0_v *nc A* $*_v$ *v* = 0_v *n*
assumes *distinct* (*cols A*)
shows *lin-dep* (*set* (*cols A*))
⟨*proof*⟩

lemma (**in** *vec-space*) *lin-depE*:
assumes *A* ∈ *carrier-mat n nc*
assumes *lin-dep* (*set* (*cols A*))
assumes *distinct* (*cols A*)
obtains *v* **where** *v* ∈ *carrier-vec nc v* ≠ 0_v *nc A* $*_v$ *v* = 0_v *n*
⟨*proof*⟩

lemma (**in** *vec-space*) *non-distinct-low-rank*:
assumes *A* ∈ *carrier-mat n n*
and ¬ *distinct* (*cols A*)
shows *rank A* < *n*
⟨*proof*⟩

The theorem "det non-zero ⟷ full rank" is practically proven in det_0_iff_vec_prod_zero_field, but without an actual definition of the rank.

lemma (**in** *vec-space*) *det-zero-low-rank*:
assumes *A* ∈ *carrier-mat n n*
and *det A* = *0*
shows *rank A* < *n*
⟨*proof*⟩

lemma *det-identical-cols*:
  assumes *A*: *A* ∈ *carrier-mat n n*
    and *ij*: *i* ≠ *j*
    and *i*: *i* < *n* **and** *j*: *j* < *n*
    and *r*: *col A i* = *col A j*
  shows *det A* = *0*
  ⟨*proof*⟩

lemma (**in** *vec-space*) *low-rank-det-zero*:
assumes *A* ∈ *carrier-mat n n*
and *det A* ≠ *0*
shows *rank A* = *n*
⟨*proof*⟩

lemma (**in** *vec-space*) *det-rank-iff*:
assumes *A* ∈ *carrier-mat n n*
shows *det A* ≠ *0* ⟷ *rank A* = *n*

⟨*proof*⟩

33 Subadditivity of rank

Subadditivity is the property of rank, that rank (A + B) <= rank A + rank B.

lemma (**in** *Module.module*) *lincomb-add*:
assumes *finite* (*b1* ∪ *b2*)
assumes *b1* ∪ *b2* ⊆ *carrier M*
assumes *x1* = *lincomb a1 b1 a1*∈ (*b1*→*carrier R*)
assumes *x2* = *lincomb a2 b2 a2*∈ (*b2*→*carrier R*)
assumes $x = x1 \oplus_M x2$
shows *lincomb* (*λv.* (*λv. if v* ∈ *b1 then a1 v else* **0**) *v* ⊕ (*λv. if v* ∈ *b2 then a2 v else* **0**) *v*) (*b1* ∪ *b2*) = *x*
⟨*proof*⟩

lemma (**in** *vectorspace*) *dim-subadditive*:
assumes *subspace K W1 V*
and *vectorspace.fin-dim K* (*vs W1*)
assumes *subspace K W2 V*
and *vectorspace.fin-dim K* (*vs W2*)
shows *vectorspace.dim K* (*vs* (*subspace-sum W1 W2*)) ≤ *vectorspace.dim K* (*vs W1*) + *vectorspace.dim K* (*vs W2*)
⟨*proof*⟩

lemma (**in** *Module.module*) *nested-submodules*:
assumes *submodule R W M*
assumes *submodule R X M*
assumes *X* ⊆ *W*
shows *submodule R X* (*md W*)
⟨*proof*⟩

lemma (**in** *vectorspace*) *nested-subspaces*:
assumes *subspace K W V*
assumes *subspace K X V*
assumes *X* ⊆ *W*
shows *subspace K X* (*vs W*)
⟨*proof*⟩

lemma (**in** *vectorspace*) *subspace-dim*:
assumes *subspace K X V fin-dim vectorspace.fin-dim K* (*vs X*)
shows *vectorspace.dim K* (*vs X*) ≤ *dim*
⟨*proof*⟩

lemma (**in** *vectorspace*) *fin-dim-subspace-sum*:
assumes *subspace K W1 V*
assumes *subspace K W2 V*
assumes *vectorspace.fin-dim K* (*vs W1*) *vectorspace.fin-dim K* (*vs W2*)

shows *vectorspace.fin-dim K* (*vs* (*subspace-sum W1 W2*))
⟨*proof*⟩

lemma (**in** *vec-space*) *rank-subadditive*:
assumes $A \in$ *carrier-mat n nc*
assumes $B \in$ *carrier-mat n nc*
shows *rank* $(A + B) \leq$ *rank A* + *rank B*
⟨*proof*⟩

lemma (**in** *vec-space*) *span-zero*: *span* {*zero V*} = {*zero V*}
⟨*proof*⟩

lemma (**in** *vec-space*) *dim-zero-vs*: *vectorspace.dim class-ring* (*span-vs* {}) = *0*
⟨*proof*⟩

lemma (**in** *vec-space*) *rank-0I*: *rank* (0_m *n nc*) = *0*
⟨*proof*⟩

lemma (**in** *vec-space*) *rank-le-1-product-entries*:
fixes *f g*::*nat* ⇒ *'a*
assumes $A \in$ *carrier-mat n nc*
assumes ⋀*r c*. *r*<*dim-row A* ⟹ *c*<*dim-col A* ⟹ *A* $$ (*r*,*c*) = *f r* ∗ *g c*
shows *rank A* ≤ *1*
⟨*proof*⟩

end

34 Missing Lemmas of Sublist

theory *DL-Missing-Sublist*
imports *Main*
begin

lemma *nths-only-one*:
assumes {*i*. *i* < *length xs* ∧ *i*∈*I*} = {*j*}
shows *nths xs I* = [*xs*!*j*]
⟨*proof*⟩

lemma *nths-replicate*:
nths (*replicate n x*) *A* = (*replicate* (*card* {*i*. *i* < *n* ∧ *i* ∈ *A*}) *x*)
⟨*proof*⟩

lemma *length-nths-even*:
assumes *even* (*length xs*)
shows *length* (*nths xs* (*Collect even*)) = *length* (*nths xs* (*Collect odd*))
⟨*proof*⟩

lemma *nths-map*:

$\mathit{nths}\ (\mathit{map}\ f\ \mathit{xs})\ A = \mathit{map}\ f\ (\mathit{nths}\ \mathit{xs}\ A)$
$\langle \mathit{proof} \rangle$

35 Pick

fun $\mathit{pick} :: \mathit{nat}\ \mathit{set} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat}$ **where**
$\mathit{pick}\ S\ 0 = (\mathit{LEAST}\ a.\ a {\in} S)\ |$
$\mathit{pick}\ S\ (\mathit{Suc}\ n) = (\mathit{LEAST}\ a.\ a {\in} S \wedge a > \mathit{pick}\ S\ n)$

lemma *pick-in-set-inf*:
assumes *infinite S*
shows $\mathit{pick}\ S\ n \in S$
$\langle \mathit{proof} \rangle$

lemma *pick-mono-inf*:
assumes *infinite S*
shows $m < n \Longrightarrow \mathit{pick}\ S\ m < \mathit{pick}\ S\ n$
$\langle \mathit{proof} \rangle$

lemma *pick-eq-iff-inf*:
assumes *infinite S*
shows $x = y \longleftrightarrow \mathit{pick}\ S\ x = \mathit{pick}\ S\ y$
$\langle \mathit{proof} \rangle$

lemma *card-le-pick-inf*:
assumes *infinite S*
and $\mathit{pick}\ S\ n \geq i$
shows $\mathit{card}\ \{a {\in} S.\ a < i\} \leq n$
$\langle \mathit{proof} \rangle$

lemma *card-pick-inf*:
assumes *infinite S*
shows $\mathit{card}\ \{a {\in} S.\ a < \mathit{pick}\ S\ n\} = n$
$\langle \mathit{proof} \rangle$

lemma
assumes $n < \mathit{card}\ S$
shows
pick-in-set-le:$\mathit{pick}\ S\ n \in S$ **and**
card-pick-le: $\mathit{card}\ \{a {\in} S.\ a < \mathit{pick}\ S\ n\} = n$ **and**
pick-mono-le: $m < n \Longrightarrow \mathit{pick}\ S\ m < \mathit{pick}\ S\ n$
$\langle \mathit{proof} \rangle$

lemma *card-le-pick-le*:
assumes $n < \mathit{card}\ S$
and $\mathit{pick}\ S\ n \geq i$
shows $\mathit{card}\ \{a {\in} S.\ a < i\} \leq n$
$\langle \mathit{proof} \rangle$

lemma
assumes $n < \mathit{card}\ S \lor \mathit{infinite}\ S$
shows
  pick-in-set:$\mathit{pick}\ S\ n \in S$ **and**
  card-le-pick: $i \leq \mathit{pick}\ S\ n \texttt{==>} \mathit{card}\ \{a{\in}S.\ a < i\} \leq n$ **and**
  card-pick: $\mathit{card}\ \{a{\in}S.\ a < \mathit{pick}\ S\ n\} = n$ **and**
  pick-mono: $m < n \Longrightarrow \mathit{pick}\ S\ m < \mathit{pick}\ S\ n$
    ⟨*proof*⟩

lemma *pick-card*:
$\mathit{pick}\ I\ (\mathit{card}\ \{a{\in}I.\ a < i\}) = (\mathit{LEAST}\ a.\ a{\in}I \land a \geq i)$
⟨*proof*⟩

lemma *pick-card-in-set*: $i{\in}I \Longrightarrow \mathit{pick}\ I\ (\mathit{card}\ \{a{\in}I.\ a < i\}) = i$
  ⟨*proof*⟩

36 Sublist

lemma *nth-nths-card*:
assumes $j{<}\mathit{length}\ \mathit{xs}$
and $j{\in}J$
shows $\mathit{nths}\ \mathit{xs}\ J\ !\ \mathit{card}\ \{j0.\ j0 < j \land j0 \in J\} = \mathit{xs}!j$
⟨*proof*⟩

lemma *pick-reduce-set*:
assumes $i{<}\mathit{card}\ \{a.\ a{<}m \land a{\in}I\}$
shows $\mathit{pick}\ I\ i = \mathit{pick}\ \{a.\ a < m \land a \in I\}\ i$
⟨*proof*⟩

lemma *nth-nths*:
assumes $i{<}\mathit{card}\ \{i.\ i{<}\mathit{length}\ \mathit{xs} \land i{\in}I\}$
shows $\mathit{nths}\ \mathit{xs}\ I\ !\ i = \mathit{xs}\ !\ \mathit{pick}\ I\ i$
⟨*proof*⟩

lemma *pick-UNIV*: $\mathit{pick}\ \mathit{UNIV}\ j = j$
⟨*proof*⟩

lemma *pick-le*:
assumes $n < \mathit{card}\ \{a.\ a < i \land a \in S\}$
shows $\mathit{pick}\ S\ n < i$
⟨*proof*⟩

lemma *prod-list-complementary-nthss*:
fixes $f :: 'a \Rightarrow 'b{::}\mathit{comm\text{-}monoid\text{-}mult}$
shows $\mathit{prod\text{-}list}\ (\mathit{map}\ f\ \mathit{xs}) = \mathit{prod\text{-}list}\ (\mathit{map}\ f\ (\mathit{nths}\ \mathit{xs}\ A)) * \mathit{prod\text{-}list}\ (\mathit{map}\ f\ (\mathit{nths}\ \mathit{xs}\ ({-}A)))$
⟨*proof*⟩

lemma *nths-zip*: $\mathit{nths}\ (\mathit{zip}\ \mathit{xs}\ \mathit{ys})\ I = \mathit{zip}\ (\mathit{nths}\ \mathit{xs}\ I)\ (\mathit{nths}\ \mathit{ys}\ I)$

⟨*proof*⟩

37 weave

definition *weave* :: *nat set* ⇒ *'a list* ⇒ *'a list* ⇒ *'a list* **where**
weave A xs ys = *map* (*λi. if i∈A then xs*!(*card* {*a∈A. a<i*}) *else ys*!(*card* {*a∈−A. a<i*})) [*0..<length xs* + *length ys*]

lemma *length-weave*:
shows *length* (*weave A xs ys*) = *length xs* + *length ys*
⟨*proof*⟩

lemma *nth-weave*:
assumes *i* < *length* (*weave A xs ys*)
shows *weave A xs ys* ! *i* = (*if i∈A then xs*!(*card* {*a∈A. a<i*}) *else ys*!(*card* {*a∈−A. a<i*}))
⟨*proof*⟩

lemma *weave-append1*:
assumes *length xs* + *length ys* ∈ *A*
assumes *length xs* = *card* {*a∈A. a* < *length xs* + *length ys*}
shows *weave A* (*xs* @ [*x*]) *ys* = *weave A xs ys* @ [*x*]
⟨*proof*⟩

lemma *weave-append2*:
assumes *length xs* + *length ys* ∉ *A*
assumes *length ys* = *card* {*a∈−A. a* < *length xs* + *length ys*}
shows *weave A xs* (*ys* @ [*y*]) = *weave A xs ys* @ [*y*]
⟨*proof*⟩

lemma *nths-nth*:
assumes *n∈A n<length xs*
shows *nths xs A* ! (*card* {*i. i<n* ∧ *i∈A*}) = *xs* ! *n*
⟨*proof*⟩

lemma *list-all2-nths*:
assumes *list-all2 P* (*nths xs A*) (*nths ys A*)
and *list-all2 P* (*nths xs* (−*A*)) (*nths ys* (−*A*))
shows *list-all2 P xs ys*
⟨*proof*⟩

lemma *nths-weave*:
assumes *length xs* = *card* {*a∈A. a* < *length xs* + *length ys*}
assumes *length ys* = *card* {*a∈*(−*A*). *a* < *length xs* + *length ys*}
shows *nths* (*weave A xs ys*) *A* = *xs* ∧ *nths* (*weave A xs ys*) (−*A*) = *ys*
⟨*proof*⟩

lemma *set-weave*:
assumes *length xs* = *card* {*a∈A. a* < *length xs* + *length ys*}

assumes *length ys* = *card* {*a*∈−*A*. *a* < *length xs* + *length ys*}
shows *set* (*weave A xs ys*) = *set xs* ∪ *set ys*
⟨*proof*⟩

lemma *weave-complementary-nthss*[*simp*]:
weave A (*nths xs A*) (*nths xs* (−*A*)) = *xs*
⟨*proof*⟩

lemma *length-nths′*: *length* (*nths xs I*) = *card* {*i*∈*I*. *i* < *length xs*}
⟨*proof*⟩

end

38 Submatrices

theory *DL-Submatrix*
imports *Matrix DL-Missing-Sublist*
begin

39 Submatrix

definition *submatrix* :: *′a mat* ⇒ *nat set* ⇒ *nat set* ⇒ *′a mat* **where**
submatrix A I J = *mat* (*card* {*i*. *i*<*dim-row A* ∧ *i*∈*I*}) (*card* {*j*. *j*<*dim-col A* ∧ *j*∈*J*}) (λ(*i*,*j*). *A* $$ (*pick I i*, *pick J j*))

lemma *dim-submatrix*: *dim-row* (*submatrix A I J*) = *card* {*i*. *i*<*dim-row A* ∧ *i*∈*I*}
dim-col (*submatrix A I J*) = *card* {*j*. *j*<*dim-col A* ∧ *j*∈*J*}
⟨*proof*⟩

lemma *submatrix-index*:
assumes *i*<*card* {*i*. *i*<*dim-row A* ∧ *i*∈*I*}
assumes *j*<*card* {*j*. *j*<*dim-col A* ∧ *j*∈*J*}
shows *submatrix A I J* $$ (*i*,*j*) = *A* $$ (*pick I i*, *pick J j*)
⟨*proof*⟩

lemma *set-le-in*:{*a*. *a* < *n* ∧ *a* ∈ *I*} = {*a* ∈ *I*. *a* < *n*} ⟨*proof*⟩

lemma *submatrix-index-card*:
assumes *i*<*dim-row A j*<*dim-col A i*∈*I j*∈*J*
shows *submatrix A I J* $$ (*card* {*a*∈*I*. *a* < *i*}, *card* {*a*∈*J*. *a* < *j*}) = *A* $$ (*i*, *j*)
⟨*proof*⟩

lemma *submatrix-split*: *submatrix A I J* = *submatrix* (*submatrix A UNIV J*) *I UNIV*
⟨*proof*⟩

end

40 Rank and Submatrices

theory *DL-Rank-Submatrix*
imports *DL-Rank DL-Submatrix Matrix*
begin

lemma *row-submatrix-UNIV*:
assumes $\mathit{i} < \mathit{card}\ \{i.\ i < \mathit{dim\text{-}row}\ A \wedge i \in I\}$
shows $\mathit{row}\ (\mathit{submatrix}\ A\ I\ \mathit{UNIV})\ i = \mathit{row}\ A\ (\mathit{pick}\ I\ i)$
⟨*proof*⟩

lemma *distinct-cols-submatrix-UNIV*:
assumes $\mathit{distinct}\ (\mathit{cols}\ (\mathit{submatrix}\ A\ I\ \mathit{UNIV}))$
shows $\mathit{distinct}\ (\mathit{cols}\ A)$
⟨*proof*⟩

lemma *cols-submatrix-subset*: $\mathit{set}\ (\mathit{cols}\ (\mathit{submatrix}\ A\ \mathit{UNIV}\ J)) \subseteq \mathit{set}\ (\mathit{cols}\ A)$
⟨*proof*⟩

lemma (**in** *vec-space*) *lin-dep-submatrix-UNIV*:
assumes $A \in \mathit{carrier\text{-}mat}\ n\ \mathit{nc}$
assumes $\mathit{lin\text{-}dep}\ (\mathit{set}\ (\mathit{cols}\ A))$
assumes $\mathit{distinct}\ (\mathit{cols}\ (\mathit{submatrix}\ A\ I\ \mathit{UNIV}))$
shows $\mathit{LinearCombinations.module.lin\text{-}dep}\ \mathit{class\text{-}ring}\ (\mathit{module\text{-}vec}\ \mathit{TYPE}('a)\ (\mathit{card}\ \{i.\ i < n \wedge i \in I\}))\ (\mathit{set}\ (\mathit{cols}\ (\mathit{submatrix}\ A\ I\ \mathit{UNIV})))$
(**is** $\mathit{LinearCombinations.module.lin\text{-}dep}\ \mathit{class\text{-}ring}\ ?M\ (\mathit{set}\ ?S')$)
⟨*proof*⟩

lemma (**in** *vec-space*) *rank-gt-minor*:
assumes $A \in \mathit{carrier\text{-}mat}\ n\ \mathit{nc}$
assumes $\mathit{det}\ (\mathit{submatrix}\ A\ I\ J) \neq 0$
shows $\mathit{card}\ \{j.\ j < \mathit{nc} \wedge j \in J\} \leq \mathit{rank}\ A$
⟨*proof*⟩

end

References

[1] M. Avanzini, C. Sternagel, and R. Thiemann. Certification of complexity proofs using CeTA. In *Proc. RTA 2015*, LIPIcs 36, pages 23–39, 2015.

[2] J. Divasón and J. Aransay. Gauss-jordan algorithm and its applications. *Archive of Formal Proofs*, Sept. 2014. http://isa-afp.org/entries/Gauss_Jordan.shtml, Formal proof development.

[3] J. Endrullis, J. Waldmann, and H. Zantema. Matrix Interpretations for Proving Termination of Term Rewriting. *Journal of Automated Reasoning*, 40(2-3):195–220, 2008.

[4] A. Lochbihler. Light-weight containers. *Archive of Formal Proofs*, Apr. 2013. http://isa-afp.org/entries/Containers.shtml, Formal proof development.

[5] R. Piziak and P. L. Odell. *Matrix theory: from generalized inverses to Jordan form.* CRC Press, 2007.

[6] C. Sternagel and R. Thiemann. Executable matrix operations on matrices of arbitrary dimensions. *Archive of Formal Proofs*, June 2010. http://isa-afp.org/entries/Matrix.shtml, Formal proof development.

[7] R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *Proc. TPHOLs'09*, LNCS 5674, pages 452–468, 2009.