Matrices, Jordan Normal Forms, and Spectral Radius Theory*

René Thiemann and Akihisa Yamada

March 19, 2025

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) ⟷ finite A› if ‹inj-on f A›
  using that by (fact finite-image-iff)
```

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

```
lemma finite-bounded-functions:
  assumes fS: finite S
  shows finite T ⟹ finite {f. (∀ i ∈ T. f i ∈ S) ∧ (∀ i. i ∉ T ⟶ f i = i)}
proof (induct T rule: finite-induct)
  case empty
  have th: {f. ∀ i. f i = i} = {id}
    by auto
  show ?case
    by (auto simp add: th)
next
  case (insert a T)
  let ?f = λ(y,g) i. if i = a then y else g i
  let ?S = ?f ‘ (S × {f. (∀ i∈T. f i ∈ S) ∧ (∀ i. i ∉ T ⟶ f i = i)})
  have ?S = {f. (∀ i∈ insert a T. f i ∈ S) ∧ (∀ i. i ∉ insert a T ⟶ f i = i)}
    apply (auto simp add: image-iff)
    apply (rule-tac x=x a in bexI)
    apply (rule-tac x = λi. if i = a then i else x i in exI)
    apply (insert insert, auto)
    done
  with finite-imageI[OF finite-cartesian-product[OF fS insert.hyps(3)], of ?f]
  show ?case
    by metis
qed

lemma finite-bounded-functions':
  assumes fS: finite S
  shows finite T ⟹ finite {f. (∀ i ∈ T. f i ∈ S) ∧ (∀ i. i ∉ T ⟶ f i = j)}
proof (induct T rule: finite-induct)
  case empty
  have th: {f. ∀ i. f i = j} = {(λ x. j)}
    by auto
  show ?case
    by (auto simp add: th)
next
  case (insert a T)
  let ?f = λ(y,g) i. if i = a then y else g i
```

```
  let ?S = ?f ‘ (S × {f. (∀ i∈T. f i ∈ S) ∧ (∀ i. i ∉ T ⟶ f i = j)})
  have ?S = {f. (∀ i∈ insert a T. f i ∈ S) ∧ (∀ i. i ∉ insert a T ⟶ f i = j)}
    apply (auto simp add: image-iff)
    apply (rule-tac x=x a in bexI)
    apply (rule-tac x = λi. if i = a then j else x i in exI)
    apply (insert insert, auto)
    done
  with finite-imageI[OF finite-cartesian-product[OF fS insert.hyps(3)], of ?f]
  show ?case
    by metis
qed

lemma permutes-less [simp]:
  assumes p: p permutes {0..<(n :: nat)}
  shows
    i < n ⟹ p i < n
    i < n ⟹ inv p i < n
    p (inv p i) = i
    inv p (p i) = i
  using assms
  by (simp-all add: permutes-inverses permutes-nat-less permutes-nat-inv-less)

lemma permutes-prod:
  assumes p: p permutes S
  shows (∏ s∈S. f (p s) s) = (∏ s∈S. f s (inv p s))
    (is ?l = ?r)
  using assms by (fact prod.permutes-inv)

lemma permutes-sum:
  assumes p: p permutes S
  shows (∑ s∈S. f (p s) s) = (∑ s∈S. f s (inv p s))
    (is ?l = ?r)
  using assms by (fact sum.permutes-inv)

context
  fixes A :: 'a set
    and B :: 'b set
    and a-to-b :: 'a ⇒ 'b
    and b-to-a :: 'b ⇒ '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 $a \in A$ *a-to-b* $a \in B$ *ip* (*a-to-b* a) $\in B$ p (*a-to-b* a) $\in B$
b-to-a (p (*a-to-b* a)) $\in A$ *b-to-a* (*ip* (*a-to-b* a)) $\in A$
proof −
let *?b* = *a-to-b a*
from *p* **have** *ip*: *ip permutes B* **unfolding** *ip-def* **by** (*rule permutes-inv*)
note *in-ip* = *permutes-in-image*[*OF ip*]
note *in-p* = *permutes-in-image*[*OF p*]
show *a*: $a \in A$ **by** *fact*
show *b*: *?b* $\in B$ **by** (*rule ab*[*OF a*])
show *pb*: *p ?b* $\in B$ **unfolding** *in-p* **by** (*rule b*)
show *ipb*: *ip ?b* $\in B$ **unfolding** *in-ip* **by** (*rule b*)
show *b-to-a* (*p ?b*) $\in A$ **by** (*rule ba*[*OF pb*])
show *b-to-a* (*ip ?b*) $\in A$ **by** (*rule ba*[*OF ipb*])
qed

lemma *permutes-bij-main*:
{*p* . *p permutes A*} $\supseteq$ (λ *p a*. *if* $a \in A$ *then* *b-to-a* (*p* (*a-to-b a*)) *else a*) ‘ {*p* . *p permutes B*}
(**is** *?A* $\supseteq$ *?f* ‘ *?B*)
proof
note *d* = *permutes-def*
let *?g* = λ *q b*. *if* $b \in B$ *then* *a-to-b* (*q* (*b-to-a b*)) *else b*
let *?inv* = *Hilbert-Choice.inv*
fix *p*
assume *p*: $p \in$ *?f* ‘ *?B*
then obtain *q* **where** *q*: *q permutes B* **and** *p*: *p* = *?f q* **by** *auto*
let *?iq* = *?inv q*
from *q* **have** *iq*: *?iq permutes B* **by** (*rule permutes-inv*)
note *in-iq* = *permutes-in-image*[*OF iq*]
note *in-q* = *permutes-in-image*[*OF q*]
have *qiB*: $\bigwedge$ *b*. $b \in B \Longrightarrow$ *q* (*?iq b*) = *b* **using** *q* **by** (*rule permutes-inverses*)
have *iqB*: $\bigwedge$ *b*. $b \in B \Longrightarrow$ *?iq* (*q b*) = *b* **using** *q* **by** (*rule permutes-inverses*)
from *q*[*unfolded d*]
have *q1*: $\bigwedge$ *b*. $b \notin B \Longrightarrow$ *q b* = *b*
and *q2*: $\bigwedge$ *b*. $\exists!b'$. *q* b' = *b* **by** *auto*
note *memb* = *permutes-memb*[*OF q*]
show $p \in$ *?A* **unfolding** *p d*
proof (*rule*, *intro conjI impI allI*, *force*)
fix *a*
show $\exists!a'$. *?f q* a' = *a*
proof (*cases* $a \in A$)
case *True*
note *a* = *memb*[*OF True*]
let *?a* = *b-to-a* (*?iq* (*a-to-b a*))
show *?thesis*
proof
show *?f q ?a* = *a* **using** *a* **by** (*simp add*: *ba-ab qiB ab-ba*)
next
fix a'

```
      assume id: ?f q a′ = a
      show a′ = ?a
      proof (cases a′ ∈ A)
        case False
        thus ?thesis using id a by auto
      next
        case True
        note a′ = memb[OF this]
        from id True have b-to-a (q (a-to-b a′)) = a by simp
        from arg-cong[OF this, of a-to-b] a′ a
        have q (a-to-b a′) = a-to-b a by (simp add: ba-ab)
        from arg-cong[OF this, of ?iq]
        have a-to-b a′ = ?iq (a-to-b a) unfolding iqB[OF a′(2)] .
        from arg-cong[OF this, of b-to-a] show ?thesis unfolding ab-ba[OF True]
.
      qed
    qed
  next
    case False note a = this
    show ?thesis
    proof
      show ?f q a = a using a by simp
    next
      fix a′
      assume id: ?f q a′ = a
      show a′ = a
      proof (cases a′ ∈ A)
        case False
        with id show ?thesis by simp
      next
        case True
        note a′ = memb[OF True]
        with id False show ?thesis by auto
      qed
    qed
  qed
 qed
qed

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 −
```

```
  note one-dir = ab ba ab-ba ba-ab
  note other-dir = ba ab ba-ab ab-ba
  let ?g = (λ p b. if b ∈ B then a-to-b (p (b-to-a b)) else b)
  define PA where PA = ?A
  define f where f = ?f
  define g where g = ?g
  {
    fix p
    assume p ∈ PA
    hence p: p permutes A unfolding PA-def by simp
    from p[unfolded permutes-def] have pnA: ⋀ a. a ∉ A ⟹ p a = a by auto
    have ?f (?g p) = p
    proof (rule ext)
      fix a
      show ?f (?g p) a = p a
      proof (cases a ∈ A)
        case False
        thus ?thesis by (simp add: pnA)
      next
        case True note a = this
        hence p a ∈ A unfolding permutes-in-image[OF p] .
        thus ?thesis using a by (simp add: ab-ba ba-ab ab)
      qed
    qed
    hence f (g p) = p unfolding f-def g-def .
  }
  hence f ‘ g ‘ PA = PA by force
  hence id: ?f ‘ ?g ‘ ?A = ?A unfolding PA-def f-def g-def .
  have ?f ‘ ?B ⊆ ?A by (rule permutes-bij-main[OF one-dir])
  moreover have ?g ‘ ?A ⊆ ?B by (rule permutes-bij-main[OF ba ab ba-ab ab-ba])
  hence ?f ‘ ?g ‘ ?A ⊆ ?f ‘ ?B by auto
  hence ?A ⊆ ?f ‘ ?B unfolding id .
  ultimately show ?thesis by blast
qed

lemma permutes-others:
  assumes p: p permutes S and x: x ∉ S shows p x = x
  using p x by (rule permutes-not-in)

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
  unfolding permutes-def
proof (intro conjI allI impI, rule f)
  fix y
  from endo-inj-surj[OF fin - i] fS have fs: f ‘ S = S by auto
  show ∃!x. f x = y
```

```
  proof (cases y ∈ S)
    case False
    thus ?thesis by (intro ex1I[of - y], insert fS f, auto)
  next
    case True
    with fs obtain x where x: x ∈ S and fx: f x = y by force
    show ?thesis
    proof (rule ex1I, rule fx)
      fix x′
      assume fx′: f x′ = y
      with True f[of x′] have x′ ∈ S by metis
      from inj-onD[OF i fx[folded fx′] x this]
      show x′ = x by simp
    qed
  qed
qed

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

lemma signof-id:
  signof id = 1
  signof (λx. x) = 1
  by simp-all

lemma signof-inv: finite S ⟹ p permutes S ⟹ signof (inv p) = signof p
  by (simp add: permutes-imp-permutation sign-inverse)

lemma signof-pm-one: signof p ∈ {1, − 1}
  by (simp add: sign-def)

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 −
  from assms have pp: permutation p permutation q
    by (auto simp: permutation-permutes)
  then show signof (p o q) = signof p ∗ signof q
    by (simp add: sign-compose)
qed

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 ..

end
```

partially ordered variant

```
class ordered-semiring-strict = semiring + comm-monoid-add + ordered-cancel-ab-semigroup-add
+
  assumes mult-strict-left-mono: a < b ⟹ 0 < c ⟹ c ∗ a < c ∗ b
  assumes mult-strict-right-mono: a < b ⟹ 0 < c ⟹ a ∗ c < b ∗ c
begin

subclass semiring-0-cancel ..

subclass ordered-semiring
proof
  fix a b c :: 'a
  assume A: a ≤ b 0 ≤ c
  from A show c ∗ a ≤ c ∗ b
    unfolding le-less
    using mult-strict-left-mono by (cases c = 0) auto
  from A show a ∗ c ≤ b ∗ c
    unfolding le-less
    using mult-strict-right-mono by (cases c = 0) auto
qed

lemma mult-pos-pos[simp]: 0 < a ⟹ 0 < b ⟹ 0 < a ∗ b
using mult-strict-left-mono [of 0 b a] by simp

lemma mult-pos-neg: 0 < a ⟹ b < 0 ⟹ a ∗ b < 0
using mult-strict-left-mono [of b 0 a] by simp

lemma mult-neg-pos: a < 0 ⟹ 0 < b ⟹ a ∗ b < 0
using mult-strict-right-mono [of a 0 b] by simp
```

Legacy - use *mult-neg-pos*

```
lemma mult-pos-neg2: 0 < a ⟹ b < 0 ⟹ b ∗ a < 0
by (drule mult-strict-right-mono [of b 0], auto)
```

Strict monotonicity in both arguments

```
lemma mult-strict-mono:
```

assumes $a < b$ **and** $c < d$ **and** $0 < b$ **and** $0 \le c$
shows $a * c < b * d$
using *assms* **apply** (*cases* $c=0$)
apply (*simp*)
apply (*erule mult-strict-right-mono* [*THEN less-trans*])
apply (*force simp add*: *le-less*)
apply (*erule mult-strict-left-mono*, *assumption*)
done

This weaker variant has more natural premises

lemma *mult-strict-mono'*:
assumes $a < b$ **and** $c < d$ **and** $0 \le a$ **and** $0 \le c$
shows $a * c < b * d$
by (*rule mult-strict-mono*) (*insert assms*, *auto*)

lemma *mult-less-le-imp-less*:
assumes $a < b$ **and** $c \le d$ **and** $0 \le a$ **and** $0 < c$
shows $a * c < b * d$
using *assms* **apply** (*subgoal-tac* $a * c < b * c$)
apply (*erule less-le-trans*)
apply (*erule mult-left-mono*)
apply *simp*
apply (*erule mult-strict-right-mono*)
apply *assumption*
done

lemma *mult-le-less-imp-less*:
assumes $a \le b$ **and** $c < d$ **and** $0 < a$ **and** $0 \le c$
shows $a * c < b * d$
using *assms* **apply** (*subgoal-tac* $a * c \le b * c$)
apply (*erule le-less-trans*)
apply (*erule mult-strict-left-mono*)
apply *simp*
apply (*erule mult-right-mono*)
apply *simp*
done

end

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

subclass *semiring-1* **..**
subclass *comm-ring-1* **..**
subclass *ordered-ring* **..**
subclass *ordered-comm-semiring* **by**(*unfold-locales*, *fact mult-left-mono*)
subclass *ordered-ab-semigroup-add* **..**

lemma *of-nat-ge-0*[*simp*]: *of-nat* $x \ge 0$

proof (*induct x*)
  case *0* **thus** *?case* **by** *auto*
  next case (*Suc x*)
    hence $0 \leq$ *of-nat x* **by** *auto*
    also have *of-nat x* $<$ *of-nat* (*Suc x*) **by** *auto*
    finally show *?case* **by** *auto*
qed

lemma *of-nat-eq-0*[*simp*]: *of-nat x* $= 0 \longleftrightarrow x = 0$
proof(*induct x*,*simp*)
  case (*Suc x*)
    have *of-nat* (*Suc x*) > 0 **apply**(*rule le-less-trans*[*of - of-nat x*]) **by** *auto*
    thus *?case* **by** *auto*
qed

lemma *inj-of-nat*: *inj* (*of-nat* :: *nat* $\Rightarrow$ *'a*)
proof(*rule injI*)
  fix *x y* **show** *of-nat x* $=$ *of-nat y* $\Longrightarrow x = y$
  proof (*induct x arbitrary*: *y*)
    case *0* **thus** *?case*
      proof (*induct y*)
        case *0* **thus** *?case* **by** *auto*
        next case (*Suc y*)
          hence *of-nat* (*Suc y*) $= 0$ **by** *auto*
          hence *Suc y* $= 0$ **unfolding** *of-nat-eq-0* **by** *auto*
          hence *False* **by** *auto*
          thus *?case* **by** *auto*
      qed
    next case (*Suc x*)
      thus *?case*
      proof (*induct y*)
        case *0*
          hence *of-nat* (*Suc x*) $= 0$ **by** *auto*
          hence *Suc x* $= 0$ **unfolding** *of-nat-eq-0* **by** *auto*
          hence *False* **by** *auto*
          thus *?case* **by** *auto*
        next case (*Suc y*) **thus** *?case* **by** *auto*
      qed
  qed
qed

subclass *ring-char-0* **by**(*unfold-locales*, *fact inj-of-nat*)

end

context *comm-monoid*
begin

lemma *finprod-reindex-bij-betw*: *bij-betw h S T*
$\Longrightarrow g \in h$ ‘ $S \rightarrow$ *carrier G*
$\Longrightarrow$ *finprod G* $(\lambda x.\ g\ (h\ x))\ S =$ *finprod G g T*
using *finprod-reindex*[*of g h S*] **unfolding** *bij-betw-def* **by** *auto*

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*: $g \in S \rightarrow$ *carrier G*
and *h*: $h \in j$ ‘ $S \rightarrow$ *carrier G*
shows *finprod G g S* = *finprod G h T*
proof −
have *b*: *bij-betw j S T*
using *bij-betw-byWitness*[**where** $A{=}S$ **and** $f{=}j$ **and** $f'{=}i$ **and** $A'{=}T$] *witness*
by *auto*
have *fp*: *finprod G g S* = *finprod G* $(\lambda x.\ h\ (j\ x))\ S$
by (*rule finprod-cong*, *insert eq g*, *auto*)
show *?thesis*
using *finprod-reindex-bij-betw*[*OF b h*] **unfolding** *fp* **.**
qed
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* **..**
end

lemma (**in** *comm-monoid*) *finprod-one'*:
$(\bigwedge a.\ a \in A \Longrightarrow f\ a = \mathbf{1}) \Longrightarrow$ *finprod G f A* $= \mathbf{1}$
by (*induct A rule*: *infinite-finite-induct*, *auto*)

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*
$(A - \{a\})$
by (*rule trans*[*OF trans*[*OF - finprod-Un-disjoint*[*of* $\{a\}$ $A - \{a\}$ *f*]]], *auto*,
rule arg-cong[*of - - finprod G f*], *auto*)

lemma (**in** *comm-monoid*) *finprod-finprod*:
finite A $\Longrightarrow$ *finite B* $\Longrightarrow (\bigwedge a\ b.\ a \in A \Longrightarrow b \in B \Longrightarrow g\ a\ b \in$ *carrier G*$) \Longrightarrow$

finprod G (λ a. finprod G (g a) B) A = finprod G (λ (a,b). g a b) (A × B)
proof (*induct A rule*: *finite-induct*)
case (*insert a′ A*)
note *IH = this*
let *?l = (⨂ a∈insert a′ A. finprod G (g a) B)*
let *?r = (⨂ a∈insert a′ A × B. case a of (a, b) ⇒ g a b)*
have *?l = finprod G (g a′) B ⊗ (⨂ a∈A. finprod G (g a) B)*
using *IH* **by** *simp*
also have *(⨂ a∈A. finprod G (g a) B) = finprod G (λ (a,b). g a b) (A × B)*
by (*rule IH(3)*, *insert IH*, *auto*)
finally have *idl*: *?l = finprod G (g a′) B ⊗ finprod G (λ (a,b). g a b) (A × B)* **.**
from *IH(2)* **have** *insert a′ A × B = {a′} × B ∪ A × B* **by** *auto*
hence *?r = (⨂ a∈{a′} × B ∪ A × B. case a of (a, b) ⇒ g a b)* **by** *simp*
also have … *= (⨂ a∈{a′} × B. case a of (a, b) ⇒ g a b) ⊗ (⨂ a∈ A × B. case a of (a, b) ⇒ g a b)*
by (*rule finprod-Un-disjoint*, *insert IH*, *auto*)
also have *(⨂ a∈{a′} × B. case a of (a, b) ⇒ g a b) = finprod G (g a′) B*
using *IH(4) IH(5)*
proof (*induct B rule*: *finite-induct*)
case (*insert b′ B*)
note *IH = this*
have *id*: *(⨂ a∈{a′} × B. case a of (a, b) ⇒ g a b) = finprod G (g a′) B*
by (*rule IH(3)[OF IH(4)]*, *auto*)
have *id2*: *⋀ x F. {a′} × insert x F = insert (a′,x) ({a′} × F)* **by** *auto*
have *id3*: *(⨂ a∈insert (a′, b′) ({a′} × B). case a of (a, b) ⇒ g a b)*
= g a′ b′ ⊗ (⨂ a∈({a′} × B). case a of (a, b) ⇒ g a b)
by (*rule trans[OF finprod-insert]*, *insert IH*, *auto*)
show *?case* **unfolding** *id2 id3 id*
by (*rule sym*, *rule finprod-insert*, *insert IH*, *auto*)
qed *simp*
finally have *idr*: *?r = finprod G (g a′) B ⊗ (⨂ a∈A × B. case a of (a, b) ⇒ g a b)* **.**
show *?case* **unfolding** *idl idr* **..**
qed *simp*

lemma (**in** *comm-monoid*) *finprod-swap*:
assumes *finite A finite B ⋀ a b. a ∈ A ⟹ b ∈ B ⟹ g a b ∈ carrier G*
shows *finprod G (λ (b,a). g a b) (B × A) = finprod G (λ (a,b). g a b) (A × B)*
proof −
have [*simp*]: *(λ(a, b). (b, a)) ‘ (A × B) = B × A* **by** *auto*
have [*simp*]: *(λ x. case case x of (a, b) ⇒ (b, a) of (a, b) ⇒ g b a) = (λ (a,b). g a b)*
by (*intro ext*, *auto*)
show *?thesis*
by (*rule trans[OF trans[OF - finprod-reindex[of λ (a,b). g b a λ (a,b). (b,a)]]]*, *insert assms*, *auto simp*: *inj-on-def*)
qed

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

finite A $\Longrightarrow$ *finite B* $\Longrightarrow$ ($\bigwedge$ *a b*. *a* $\in$ *A* $\Longrightarrow$ *b* $\in$ *B* $\Longrightarrow$ *g a b* $\in$ *carrier G*) $\Longrightarrow$
finprod G (λ *a*. *finprod G* (*g a*) *B*) *A* = *finprod G* (λ *b*. *finprod G* (λ *a*. *g a b*) *A*) *B*
using *finprod-finprod*[*of A B*] *finprod-finprod*[*of B A*] *finprod-swap*[*of A B*]
by *simp*

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*:
finite A $\Longrightarrow$ *f* $\in$ *A* $\to$ *carrier R* $\Longrightarrow$ $\exists$ *a*$\in$*A*. *f a* = **0**
$\Longrightarrow$ *finprod R f A* = **0**
proof (*induct A rule*: *finite-induct*)
case (*insert a A*)
from *finprod-insert*[*OF insert*(*1*−*2*), *of f*] *insert*(*4*)
have *ins*: *finprod R f* (*insert a A*) = *f a* $\otimes$ *finprod R f A* **by** *simp*
have *fA*: *finprod R f A* $\in$ *carrier R*
by (*rule finprod-closed*, *insert insert*, *auto*)
show *?case*
proof (*cases f a* = **0**)
case *True*
with *fA* **show** *?thesis* **unfolding** *ins* **by** *simp*
next
case *False*
with *insert*(*5*) **have** $\exists$ *a* $\in$ *A*. *f a* = **0** **by** *auto*
from *insert*(*3*)[*OF - this*] *insert* **have** *finprod R f A* = **0** **by** *auto*
with *insert* **show** *?thesis* **unfolding** *ins* **by** *auto*
qed
qed *simp*

lemma (**in** *semiring*) *finsum-product*:
assumes *A*: *finite A* **and** *B*: *finite B*
and *f*: *f* $\in$ *A* $\to$ *carrier R* **and** *g*: *g* $\in$ *B* $\to$ *carrier R*
shows *finsum R f A* $\otimes$ *finsum R g B* = ($\bigoplus$ *i*$\in$*A*. $\bigoplus$ *j*$\in$*B*. *f i* $\otimes$ *g j*)
unfolding *finsum-ldistr*[*OF A finsum-closed*[*OF g*] *f*]
proof (*rule finsum-cong'*[*OF refl*])
fix *a*
assume *a*: *a* $\in$ *A*
show *f a* $\otimes$ *finsum R g B* = ($\bigoplus$ *j*$\in$*B*. *f a* $\otimes$ *g j*)
by (*rule finsum-rdistr*[*OF B - g*], *insert a f*, *auto*)
qed (*insert f g B*, *auto intro*: *finsum-closed*)

lemma (**in** *semiring*) *Units-one-side-I*:
a $\in$ *carrier R* $\Longrightarrow$ *p* $\in$ *Units R* $\Longrightarrow$ *p* $\otimes$ *a* = **1** $\Longrightarrow$ *a* $\in$ *Units R*
a $\in$ *carrier R* $\Longrightarrow$ *p* $\in$ *Units R* $\Longrightarrow$ *a* $\otimes$ *p* = **1** $\Longrightarrow$ *a* $\in$ *Units R*

by (*metis Units-closed Units-inv-Units Units-l-inv inv-unique*)+

lemma *permutes-funcset*: *p permutes A* $\Longrightarrow$ (*p* ‘ *A* $\rightarrow$ *B*) = (*A* $\rightarrow$ *B*)
  by (*simp add*: *permutes-image*)

context *comm-monoid*
begin
lemma *finprod-permute*:
  assumes *p*: *p permutes S*
  and *f*: *f* $\in$ *S* $\rightarrow$ *carrier G*
  shows *finprod G f S* = *finprod G* (*f* $\circ$ *p*) *S*
proof –
  from ‹*p permutes S*› **have** *inj p*
    by (*rule permutes-inj*)
  then have *inj-on p S*
    by (*auto intro*: *subset-inj-on*)
  from *finprod-reindex*[*OF - this*, *unfolded permutes-image*[*OF p*], *OF f*]
  show *?thesis* **unfolding** *o-def* **.**
qed

lemma *finprod-singleton-set*[*simp*]: **assumes** *f a* $\in$ *carrier G*
  shows *finprod G f* {*a*} = *f a*
proof –
  have *finprod G f* {*a*} = *f a* $\otimes$ *finprod G f* {}
    by (*rule finprod-insert*, *insert assms*, *auto*)
  also have ... = *f a* **using** *assms* **by** *auto*
  finally show *?thesis* **.**
qed
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* $\in$ {*p*. *p permutes S*} $\rightarrow$ *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 –
  let *?inv* = *Hilbert-Choice.inv*
  let *?S* = {*p* . *p permutes S*}
  have *th0*: *inj-on ?inv ?S*
  proof (*auto simp add*: *inj-on-def*)
    fix *q r*
    assume *q*: *q permutes S*

```
      and r: r permutes S
      and qr: ?inv q = ?inv r
    then have ?inv (?inv q) = ?inv (?inv r)
      by simp
    with permutes-inv-inv[OF q] permutes-inv-inv[OF r] show q = r
      by metis
  qed
  have th1: ?inv ‘ ?S = ?S
    using image-inverse-permutations by blast
  have th2: ?rhs = finsum R (f ∘ ?inv) ?S
    by (simp add: o-def)
  from finsum-reindex[OF - th0, of f] show ?thesis unfolding th1 th2 using f .
qed

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 (λp. f(p ∘ q)) {p. p permutes
S}
  (is ?lhs = ?rhs)
proof −
  let ?S = {p. p permutes S}
  let ?inv = Hilbert-Choice.inv
  have th0: ?rhs = finsum R (f ∘ (λp. p ∘ q)) ?S
    by (simp add: o-def)
  have th1: inj-on (λp. p ∘ q) ?S
  proof (auto simp add: inj-on-def)
    fix p r
    assume p permutes S
      and r: r permutes S
      and rp: p ∘ q = r ∘ q
    then have p ∘ (q ∘ ?inv q) = r ∘ (q ∘ ?inv q)
      by (simp add: o-assoc)
    with permutes-surj[OF q, unfolded surj-iff] show p = r
      by simp
  qed
  have th3: (λp. p ∘ q) ‘ ?S = ?S
    using image-compose-permutations-right[OF q] by auto
  from finsum-reindex[OF - th1, of f]
  show ?thesis unfolding th0 th1 th3 using ∗ .
qed

end

end

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

class *conjugate* =
  fixes *conjugate* :: $'a \Rightarrow 'a$
  assumes *conjugate-id*[*simp*]: *conjugate* (*conjugate a*) = *a*
    and *conjugate-cancel-iff*[*simp*]: *conjugate a* = *conjugate b* $\longleftrightarrow$ *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* $\longleftrightarrow$ *a* = *0*
   using *conjugate-cancel-iff*[*of - 0*, *unfolded conjugate-zero*]**.**
end

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

lemma *sum-conjugate*:
  fixes *f* :: $'b \Rightarrow 'a$ :: *conjugatable-ring*
  assumes *finX*: *finite X*
  shows *conjugate* (*sum f X*) = *sum* (λ*x*. *conjugate* (*f x*)) *X*
  using *finX* **by** (*induct set*:*finite*, *auto simp*: *conjugate-dist-add*)

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*..
end

lemma *conjugate-square-0*:
  fixes *a* :: $'a$ :: {*conjugatable-ordered-ring*, *semiring-no-zero-divisors*}
  shows *a* ∗ *conjugate a* = *0* $\Longrightarrow$ *a* = *0* **by** *auto*

3.1 Instantiations

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

instance
  by *intro-classes* (*auto simp*: *less-eq-complex-def*)

end

instantiation *real* :: *conjugatable-ordered-field*
begin

definition [*simp*]: *conjugate* (*x*::*real*) ≡ *x*
instance by (*intro-classes*, *auto*)
end

instantiation *rat* :: *conjugatable-ordered-field*
begin
definition [*simp*]: *conjugate* (*x*::*rat*) ≡ *x*
instance by (*intro-classes*, *auto*)
end

instantiation *int* :: *conjugatable-ordered-ring*
begin
definition [*simp*]: *conjugate* (*x*::*int*) ≡ *x*
instance by (*intro-classes*, *auto*)
end

lemma *conjugate-square-eq-0* [*simp*]:
fixes *x* :: *'a* :: {*conjugatable-ring*,*semiring-no-zero-divisors*}
shows *x* ∗ *conjugate x* = *0* ⟷ *x* = *0 conjugate x* ∗ *x* = *0* ⟷ *x* = *0*
by *auto*

lemma *conjugate-square-greater-0* [*simp*]:
fixes *x* :: *'a* :: {*conjugatable-ordered-ring*,*ring-no-zero-divisors*}
shows *x* ∗ *conjugate x* > *0* ⟷ *x* ≠ *0*
using *conjugate-square-positive*[*of x*]
by (*auto simp*: *le-less*)

lemma *conjugate-square-smaller-0* [*simp*]:
fixes *x* :: *'a* :: {*conjugatable-ordered-ring*,*ring-no-zero-divisors*}
shows ¬ *x* ∗ *conjugate x* < *0*
using *conjugate-square-positive*[*of x*] **by** *auto*

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 *undef-vec* :: *nat* ⇒ *'a* **where**
undef-vec i ≡ [] ! *i*

definition *mk-vec* :: *nat* ⇒ (*nat* ⇒ *'a*) ⇒ (*nat* ⇒ *'a*) **where**
mk-vec n f ≡ λ *i*. *if i* < *n then f i else undef-vec* (*i* − *n*)

typedef *'a vec* = {(*n*, *mk-vec n f*) | *n f* :: *nat* ⇒ *'a*. *True*}
by *auto*

setup-lifting *type-definition-vec*

lift-definition *dim-vec* :: *'a vec* ⇒ *nat* **is** *fst* **.**
lift-definition *vec-index* :: *'a vec* ⇒ (*nat* ⇒ *'a*) (**infixl** ‹$› *100*) **is** *snd* **.**
lift-definition *vec* :: *nat* ⇒ (*nat* ⇒ *'a*) ⇒ *'a vec*
is λ *n f*. (*n*, *mk-vec n f*) **by** *auto*

lift-definition *vec-of-list* :: *'a list* ⇒ *'a vec* **is**
λ *v*. (*length v*, *mk-vec* (*length v*) (*nth v*)) **by** *auto*

lift-definition *list-of-vec* :: *'a vec* ⇒ *'a list* **is**
λ (*n*,*v*). *map v* [*0* ..< *n*] **.**

definition *carrier-vec* :: *nat* ⇒ *'a vec set* **where**
carrier-vec n = { *v* . *dim-vec v* = *n*}

lemma *carrier-vec-dim-vec*[*simp*]: *v* ∈ *carrier-vec* (*dim-vec v*) **unfolding** *carrier-vec-def*
by *auto*

lemma *dim-vec*[*simp*]: *dim-vec* (*vec n f*) = *n* **by** *transfer simp*
lemma *vec-carrier*[*simp*]: *vec n f* ∈ *carrier-vec n* **unfolding** *carrier-vec-def* **by** *auto*
lemma *index-vec*[*simp*]: *i* < *n* ⟹ *vec n f* $ *i* = *f i* **by** *transfer* (*simp add*: *mk-vec-def*)
lemma *eq-vecI*[*intro*]: (⋀ *i*. *i* < *dim-vec w* ⟹ *v* $ *i* = *w* $ *i*) ⟹ *dim-vec v* = *dim-vec w*
⟹ *v* = *w*
by (*transfer*, *auto simp*: *mk-vec-def*)

lemma *carrier-dim-vec*: $v \in \textit{carrier-vec}\ n \longleftrightarrow \textit{dim-vec}\ v = n$
  unfolding *carrier-vec-def* **by** *auto*

lemma *carrier-vecD*[*simp*]: $v \in \textit{carrier-vec}\ n \Longrightarrow \textit{dim-vec}\ v = n$ **using** *carrier-dim-vec*
by *auto*

lemma *carrier-vecI*: $\textit{dim-vec}\ v = n \Longrightarrow v \in \textit{carrier-vec}\ n$ **using** *carrier-dim-vec*
by *auto*

instantiation *vec* :: (*plus*) *plus*
begin
definition *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 ..
end

instantiation *vec* :: (*minus*) *minus*
begin
definition *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 ..
end

definition
  $\textit{zero-vec} :: \textit{nat} \Rightarrow 'a :: \textit{zero}\ \textit{vec}\ (\langle 0_v \rangle)$
  where $0_v\ n \equiv \textit{vec}\ n\ (\lambda\ i.\ 0)$

lemma *zero-carrier-vec*[*simp*]: $0_v\ n \in \textit{carrier-vec}\ n$
  unfolding *zero-vec-def carrier-vec-def* **by** *auto*

lemma *index-zero-vec*[*simp*]: $i < n \Longrightarrow 0_v\ n\ \$\ i = 0$ $\textit{dim-vec}\ (0_v\ n) = n$
  unfolding *zero-vec-def* **by** *auto*

lemma *vec-of-dim-0*[*simp*]: $\textit{dim-vec}\ v = 0 \longleftrightarrow v = 0_v\ 0$ **by** *auto*

definition
  $\textit{unit-vec} :: \textit{nat} \Rightarrow \textit{nat} \Rightarrow ('a :: \textit{zero-neq-one})\ \textit{vec}$
  where $\textit{unit-vec}\ n\ i = \textit{vec}\ n\ (\lambda\ j.\ \textit{if}\ j = i\ \textit{then}\ 1\ \textit{else}\ 0)$

lemma *index-unit-vec*[*simp*]:
  $i < n \Longrightarrow j < n \Longrightarrow \textit{unit-vec}\ n\ i\ \$\ j = (\textit{if}\ j = i\ \textit{then}\ 1\ \textit{else}\ 0)$
  $i < n \Longrightarrow \textit{unit-vec}\ n\ i\ \$\ i = 1$
  $\textit{dim-vec}\ (\textit{unit-vec}\ n\ i) = n$
  unfolding *unit-vec-def* **by** *auto*

lemma *unit-vec-eq*[*simp*]:
  assumes *i*: $i < n$
  shows $(\textit{unit-vec}\ n\ i = \textit{unit-vec}\ n\ j) = (i = j)$
proof −

```
  have i ≠ j ⟹ unit-vec n i $ i ≠ unit-vec n j $ i
    unfolding unit-vec-def using i by simp
  then show ?thesis by metis
qed

lemma unit-vec-nonzero[simp]:
  assumes i-n: i < n shows unit-vec n i ≠ zero-vec n (is ?l ≠ ?r)
proof –
  have ?l $ i = 1 ?r $ i = 0 using i-n by auto
  thus ?l ≠ ?r by auto
qed

lemma unit-vec-carrier[simp]: unit-vec n i ∈ carrier-vec n
  unfolding unit-vec-def carrier-vec-def by auto

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
  unfolding unit-vecs-def set-map set-upt
proof –
  {fix m
    have m ≤ n ⟹ map (unit-vec n) [0..<m] = unit-vecs-first n m
    proof (induct m)
      case (Suc m) then have mn:m≤n by auto
        show ?case unfolding upt-Suc using Suc(1)[OF mn] by auto
    qed auto
  }
  thus map (unit-vec n) [0..<n] = unit-vecs-first n n by auto
qed
```

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) ⊆ carrier-vec n
  by (induct i;auto)

lemma unit-vecs-last[code]: unit-vecs n = unit-vecs-last n n
proof –
  { fix m assume m = n
    have m ≤ n ⟹ map (unit-vec n) [n–m..<n] = unit-vecs-last n m
      proof (induction m)
```

case (*Suc m*)
  then have *nm*:$n - Suc\ m < n$ **by** *auto*
  have *ins*: $[n - Suc\ m\ ..< n] = (n - Suc\ m)\ \#\ [n - m\ ..< n]$
    unfolding *upt-conv-Cons*[*OF nm*]
    by (*auto simp*: *Suc.prems Suc-diff-Suc Suc-le-lessD*)
  show *?case*
    unfolding *ins*
    unfolding *unit-vecs-last.simps*
    unfolding *list.map*
    using *Suc*
    unfolding *Suc* **by** *auto*
qed *simp*
}
thus *unit-vecs n* = *unit-vecs-last n n*
  unfolding *unit-vecs-def* **by** *auto*
qed

lemma *unit-vecs-carrier*: *set* (*unit-vecs n*) ⊆ *carrier-vec n*
proof
  fix *u* :: *'a vec* **assume** *u*: $u \in$ *set* (*unit-vecs n*)
  then obtain *i* **where** *u* = *unit-vec n i* **unfolding** *unit-vecs-def* **by** *auto*
  then show $u \in$ *carrier-vec n*
    using *unit-vec-carrier* **by** *auto*
qed

lemma *unit-vecs-last-distinct*:
  $j \le n \Longrightarrow i < n - j \Longrightarrow$ *unit-vec n i* ∉ *set* (*unit-vecs-last n j*)
  by (*induction j arbitrary*:*i*, *auto*)

lemma *unit-vecs-first-distinct*:
  $i \le j \Longrightarrow j < n \Longrightarrow$ *unit-vec n j* ∉ *set* (*unit-vecs-first n i*)
  by (*induction i arbitrary*:*j*, *auto*)

definition *map-vec* **where** *map-vec f v* ≡ *vec* (*dim-vec v*) (λ*i*. *f* (*v* \$ *i*))

instantiation *vec* :: (*uminus*) *uminus*
begin
definition *uminus-vec* :: *'a* :: *uminus vec* ⇒ *'a vec* **where**
  − *v* ≡ *vec* (*dim-vec v*) (λ *i*. − (*v* \$ *i*))
instance ..
end

definition *smult-vec* :: *'a* :: *times* ⇒ *'a vec* ⇒ *'a vec* (**infixl** ‹$\cdot_v$› *70*)
  where $a \cdot_v v$ ≡ *vec* (*dim-vec v*) (λ *i*. *a* ∗ *v* \$ *i*)

definition *scalar-prod* :: *'a vec* ⇒ *'a vec* ⇒ *'a* :: *semiring-0* (**infix** ‹·› *70*)
  where *v* · *w* ≡ ∑ *i* ∈ {*0* ..< *dim-vec w*}. *v* \$ *i* ∗ *w* \$ *i*

definition *monoid-vec* :: *'a itself* ⇒ *nat* ⇒ (*'a* :: *monoid-add vec*) *monoid* **where**

$monoid\text{-}vec\ ty\ n \equiv (\!|$
  $carrier = carrier\text{-}vec\ n,$
  $mult = (+),$
  $one = 0_v\ n|\!)$

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

lemma *monoid-vec-simps*:
$mult\ (monoid\text{-}vec\ ty\ n) = (+)$
$carrier\ (monoid\text{-}vec\ ty\ n) = carrier\text{-}vec\ n$
$one\ (monoid\text{-}vec\ ty\ n) = 0_v\ n$
unfolding *monoid-vec-def* **by** *auto*

lemma *module-vec-simps*:
$add\ (module\text{-}vec\ ty\ n) = (+)$
$zero\ (module\text{-}vec\ ty\ n) = 0_v\ n$
$carrier\ (module\text{-}vec\ ty\ n) = carrier\text{-}vec\ n$
$smult\ (module\text{-}vec\ ty\ n) = (\cdot_v)$
unfolding *module-vec-def* **by** *auto*

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

lemma *index-add-vec*[*simp*]:
$i < dim\text{-}vec\ v_2 \Longrightarrow (v_1 + v_2)\ \$\ i = v_1\ \$\ i + v_2\ \$\ i$ $dim\text{-}vec\ (v_1 + v_2) = dim\text{-}vec\ v_2$
unfolding *plus-vec-def* **by** *auto*

lemma *index-minus-vec*[*simp*]:
$i < dim\text{-}vec\ v_2 \Longrightarrow (v_1 - v_2)\ \$\ i = v_1\ \$\ i - v_2\ \$\ i$ $dim\text{-}vec\ (v_1 - v_2) = dim\text{-}vec\ v_2$
unfolding *minus-vec-def* **by** *auto*

lemma *index-map-vec*[*simp*]:
$i < dim\text{-}vec\ v \Longrightarrow map\text{-}vec\ f\ v\ \$\ i = f\ (v\ \$\ i)$
$dim\text{-}vec\ (map\text{-}vec\ f\ v) = dim\text{-}vec\ v$
unfolding *map-vec-def* **by** *auto*

lemma *map-carrier-vec*[*simp*]: $map\text{-}vec\ h\ v \in carrier\text{-}vec\ n = (v \in carrier\text{-}vec\ n)$
unfolding *map-vec-def carrier-vec-def* **by** *auto*

lemma *index-uminus-vec*[*simp*]:
$i < \textit{dim-vec}\ v \Longrightarrow (-\ v)\ \$\ i = -\ (v\ \$\ i)$
$\textit{dim-vec}\ (-\ v) = \textit{dim-vec}\ v$
unfolding *uminus-vec-def* **by** *auto*

lemma *index-smult-vec*[*simp*]:
$i < \textit{dim-vec}\ v \Longrightarrow (a \cdot_v v)\ \$\ i = a * v\ \$\ i\ \textit{dim-vec}\ (a \cdot_v v) = \textit{dim-vec}\ v$
unfolding *smult-vec-def* **by** *auto*

lemma *add-carrier-vec*[*simp*]:
$v_1 \in \textit{carrier-vec}\ n \Longrightarrow v_2 \in \textit{carrier-vec}\ n \Longrightarrow v_1 + v_2 \in \textit{carrier-vec}\ n$
unfolding *carrier-vec-def* **by** *auto*

lemma *minus-carrier-vec*[*simp*]:
$v_1 \in \textit{carrier-vec}\ n \Longrightarrow v_2 \in \textit{carrier-vec}\ n \Longrightarrow v_1 - v_2 \in \textit{carrier-vec}\ n$
unfolding *carrier-vec-def* **by** *auto*

lemma *comm-add-vec*[*ac-simps*]:
$(v_1 :: 'a :: \textit{ab-semigroup-add vec}) \in \textit{carrier-vec}\ n \Longrightarrow v_2 \in \textit{carrier-vec}\ n \Longrightarrow v_1 + v_2 = v_2 + v_1$
by (*intro eq-vecI*, *auto simp*: *ac-simps*)

lemma *assoc-add-vec*[*simp*]:
$(v_1 :: 'a :: \textit{semigroup-add vec}) \in \textit{carrier-vec}\ n \Longrightarrow v_2 \in \textit{carrier-vec}\ n \Longrightarrow v_3 \in \textit{carrier-vec}\ n$
$\Longrightarrow (v_1 + v_2) + v_3 = v_1 + (v_2 + v_3)$
by (*intro eq-vecI*, *auto simp*: *ac-simps*)

lemma *zero-minus-vec*[*simp*]: $(v :: 'a :: \textit{group-add vec}) \in \textit{carrier-vec}\ n \Longrightarrow \mathit{0}_v\ n - v = -\ v$
by (*intro eq-vecI*, *auto*)

lemma *minus-zero-vec*[*simp*]: $(v :: 'a :: \textit{group-add vec}) \in \textit{carrier-vec}\ n \Longrightarrow v - \mathit{0}_v\ n = v$
by (*intro eq-vecI*, *auto*)

lemma *minus-cancel-vec*[*simp*]: $(v :: 'a :: \textit{group-add vec}) \in \textit{carrier-vec}\ n \Longrightarrow v - v = \mathit{0}_v\ n$
by (*intro eq-vecI*, *auto*)

lemma *minus-add-uminus-vec*: $(v :: 'a :: \textit{group-add vec}) \in \textit{carrier-vec}\ n \Longrightarrow$
$w \in \textit{carrier-vec}\ n \Longrightarrow v - w = v + (-\ w)$
by (*intro eq-vecI*, *auto*)

lemma *comm-monoid-vec*: *comm-monoid* (*monoid-vec TYPE* ($'a$ :: *comm-monoid-add*) n)
by (*unfold-locales*, *auto simp*: *monoid-vec-def ac-simps*)

lemma *left-zero-vec*[*simp*]: $(v :: 'a :: monoid\text{-}add\ vec) \in carrier\text{-}vec\ n \implies 0_v\ n + v = v$ **by** *auto*

lemma *right-zero-vec*[*simp*]: $(v :: 'a :: monoid\text{-}add\ vec) \in carrier\text{-}vec\ n \implies v + 0_v\ n = v$ **by** *auto*

lemma *uminus-carrier-vec*[*simp*]:
$(-\ v \in carrier\text{-}vec\ n) = (v \in carrier\text{-}vec\ n)$
unfolding *carrier-vec-def* **by** *auto*

lemma *uminus-r-inv-vec*[*simp*]:
$(v :: 'a :: group\text{-}add\ vec) \in carrier\text{-}vec\ n \implies (v + -\ v) = 0_v\ n$
by (*intro eq-vecI*, *auto*)

lemma *uminus-l-inv-vec*[*simp*]:
$(v :: 'a :: group\text{-}add\ vec) \in carrier\text{-}vec\ n \implies (-\ v + v) = 0_v\ n$
by (*intro eq-vecI*, *auto*)

lemma *add-inv-exists-vec*:
$(v :: 'a :: group\text{-}add\ vec) \in carrier\text{-}vec\ n \implies \exists\ w \in carrier\text{-}vec\ n.\ w + v = 0_v\ n \wedge v + w = 0_v\ n$
by (*intro bexI*[*of - $-\ v$*], *auto*)

lemma *comm-group-vec*: *comm-group* (*monoid-vec TYPE* ($'a$ :: *ab-group-add*) *n*)
by (*unfold-locales*, *insert add-inv-exists-vec*, *auto simp*: *monoid-vec-def ac-simps Units-def*)

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*]: $(a \cdot_v v \in carrier\text{-}vec\ n) = (v \in carrier\text{-}vec\ n)$
unfolding *carrier-vec-def* **by** *auto*

lemma *scalar-prod-left-zero*[*simp*]: $v \in carrier\text{-}vec\ n \implies 0_v\ n \bullet v = 0$
unfolding *scalar-prod-def*
by (*rule sum.neutral*, *auto*)

lemma *scalar-prod-right-zero*[*simp*]: $v \in carrier\text{-}vec\ n \implies v \bullet 0_v\ n = 0$
unfolding *scalar-prod-def*

by (*rule sum.neutral*, *auto*)

lemma *scalar-prod-left-unit*[*simp*]: **assumes** *v*: (*v* :: *'a* :: *semiring-1 vec*) $\in$ *carrier-vec n* **and** *i*: $i < n$
shows *unit-vec n i* $\cdot$ *v* = *v* \$ *i*
proof −
let *?f* = λ *k*. *unit-vec n i* \$ *k* $*$ *v* \$ *k*
have *id*: ($\sum$ *k*$\in${*0*..<*n*}. *?f k*) = *unit-vec n i* \$ *i* $*$ *v* \$ *i* + ($\sum$ *k*$\in${*0*..<*n*} − {*i*}. *?f k*)
by (*rule sum.remove*, *insert i*, *auto*)
also have ($\sum$ *k*$\in${*0*..<*n*} − {*i*}. *?f k*) = *0*
by (*rule sum.neutral*, *insert i*, *auto*)
finally
show *?thesis* **unfolding** *scalar-prod-def* **using** *i v* **by** *simp*
qed

lemma *scalar-prod-right-unit*[*simp*]: **assumes** *i*: $i < n$
shows (*v* :: *'a* :: *semiring-1 vec*) $\cdot$ *unit-vec n i* = *v* \$ *i*
proof −
let *?f* = λ *k*. *v* \$ *k* $*$ *unit-vec n i* \$ *k*
have *id*: ($\sum$ *k*$\in${*0*..<*n*}. *?f k*) = *v* \$ *i* $*$ *unit-vec n i* \$ *i* + ($\sum$ *k*$\in${*0*..<*n*} − {*i*}. *?f k*)
by (*rule sum.remove*, *insert i*, *auto*)
also have ($\sum$ *k*$\in${*0*..<*n*} − {*i*}. *?f k*) = *0*
by (*rule sum.neutral*, *insert i*, *auto*)
finally
show *?thesis* **unfolding** *scalar-prod-def* **using** *i* **by** *simp*
qed

lemma *add-scalar-prod-distrib*: **assumes** *v*: $v_1 \in$ *carrier-vec n* $v_2 \in$ *carrier-vec n* $v_3 \in$ *carrier-vec n*
shows $(v_1 + v_2) \cdot v_3 = v_1 \cdot v_3 + v_2 \cdot v_3$
proof −
have ($\sum$ *i*$\in${*0*..<*dim-vec* v_3}. $(v_1 + v_2)$ \$ *i* $*$ v_3 \$ *i*) = ($\sum$ *i*$\in${*0*..<*dim-vec* v_3}. v_1 \$ *i* $*$ v_3 \$ *i* + v_2 \$ *i* $*$ v_3 \$ *i*)
by (*rule sum.cong*, *insert v*, *auto simp*: *algebra-simps*)
thus *?thesis* **unfolding** *scalar-prod-def* **using** *v* **by** (*auto simp*: *sum.distrib*)
qed

lemma *scalar-prod-add-distrib*: **assumes** *v*: $v_1 \in$ *carrier-vec n* $v_2 \in$ *carrier-vec n* $v_3 \in$ *carrier-vec n*
shows $v_1 \cdot (v_2 + v_3) = v_1 \cdot v_2 + v_1 \cdot v_3$
proof −
have ($\sum$ *i*$\in${*0*..<*dim-vec* v_3}. v_1 \$ *i* $*$ $(v_2 + v_3)$ \$ *i*) = ($\sum$ *i*$\in${*0*..<*dim-vec* v_3}. v_1 \$ *i* $*$ v_2 \$ *i* + v_1 \$ *i* $*$ v_3 \$ *i*)
by (*rule sum.cong*, *insert v*, *auto simp*: *algebra-simps*)
thus *?thesis* **unfolding** *scalar-prod-def* **using** *v* **by** (*auto intro*: *sum.distrib*)
qed

lemma *smult-scalar-prod-distrib*[*simp*]: **assumes** *v*: $v_1 \in$ *carrier-vec n* $v_2 \in$ *carrier-vec n*
shows $(a \cdot_v v_1) \cdot v_2 = a * (v_1 \cdot v_2)$
unfolding *scalar-prod-def sum-distrib-left*
by (*rule sum.cong*, *insert v*, *auto simp*: *ac-simps*)

lemma *scalar-prod-smult-distrib*[*simp*]: **assumes** *v*: $v_1 \in$ *carrier-vec n* $v_2 \in$ *carrier-vec n*
shows $v_1 \cdot (a \cdot_v v_2) = (a :: 'a :: comm\text{-}ring) * (v_1 \cdot v_2)$
unfolding *scalar-prod-def sum-distrib-left*
by (*rule sum.cong*, *insert v*, *auto simp*: *ac-simps*)

lemma *comm-scalar-prod*: **assumes** $(v_1 :: 'a :: comm\text{-}semiring\text{-}0\ vec) \in$ *carrier-vec n* $v_2 \in$ *carrier-vec n*
shows $v_1 \cdot v_2 = v_2 \cdot v_1$
unfolding *scalar-prod-def*
by (*rule sum.cong*, *insert assms*, *auto simp*: *ac-simps*)

lemma *add-smult-distrib-vec*:
$((a::'a::ring) + b) \cdot_v v = a \cdot_v v + b \cdot_v v$
unfolding *smult-vec-def plus-vec-def*
by (*rule eq-vecI*, *auto simp*: *distrib-right*)

lemma *smult-add-distrib-vec*:
assumes $v \in$ *carrier-vec n* $w \in$ *carrier-vec n*
shows $(a::'a::ring) \cdot_v (v + w) = a \cdot_v v + a \cdot_v w$
apply (*rule eq-vecI*)
unfolding *smult-vec-def plus-vec-def*
using *assms distrib-left* **by** *auto*

lemma *smult-smult-assoc*:
$a \cdot_v (b \cdot_v v) = (a * b::'a::ring) \cdot_v v$
apply (*rule sym*, *rule eq-vecI*)
unfolding *smult-vec-def plus-vec-def* **using** *mult.assoc* **by** *auto*

lemma *one-smult-vec* [*simp*]:
$(1::'a::ring\text{-}1) \cdot_v v = v$ **unfolding** *smult-vec-def*
by (*rule eq-vecI*,*auto*)

lemma *uminus-zero-vec*[*simp*]: $- (0_v\ n) = (0_v\ n :: 'a :: group\text{-}add\ vec)$
by (*intro eq-vecI*, *auto*)

lemma *index-finsum-vec*: **assumes** *finite F* **and** *i*: $i < n$
and *vs*: $vs \in F \to$ *carrier-vec n*
shows *finsum-vec TYPE*(*'a* :: *comm-monoid-add*) *n vs F* \$ *i* = *sum* (λ *f*. *vs f* \$ *i*) *F*
using ‹*finite F*› *vs*
proof (*induct F*)
case (*insert f F*)

hence *IH*: *finsum-vec TYPE('a) n vs F \$ i* = ($\sum f \in F$. *vs f \$ i*)
  and *vs*: *vs* ∈ *F* → *carrier-vec n vs f* ∈ *carrier-vec n* **by** *auto*
show *?case* **unfolding** *finsum-vec-insert*[*OF insert*(*1−2*) *vs*]
  unfolding *sum.insert*[*OF insert*(*1−2*)]
  unfolding *IH*[*symmetric*]
  by (*rule index-add-vec*, *insert i*, *insert finsum-vec-closed*[*OF vs*(*1*)], *auto*)
qed (*insert i*, *auto simp*: *finsum-vec-empty*)

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 ..
end

instantiation *vec* :: (*preorder*) *preorder*
begin
instance
  by (*standard*, *auto simp*: *less-vec-def less-eq-vec-def order-trans*)
end

instantiation *vec* :: (*order*) *order*
begin
instance
  by (*standard*, *intro eq-vecI*, *auto simp*: *less-eq-vec-def order.antisym*)
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 (*cases x*)
  case (*Pair i j*)
  have *nth-map-ge*: ⋀ *i xs*. ¬ *i* < *length xs* ⟹ *xs* ! *i* = [] ! (*i* − *length xs*)
    by (*metis append-Nil2 nth-append*)
  note [*simp*] = *Pair undef-mat-def nth-map-ge*[*of i*] *nth-map-ge*[*of j*]
  show *?thesis*

by (*cases* $i < nr$, *simp*, *cases* $j < nc$, *insert assms*, *auto*)
qed

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

lemma *cong-mk-mat*: **assumes** ⋀ *i j*. $i < nr \Longrightarrow j < nc \Longrightarrow f\ (i,j) = f'\ (i,j)$
shows *mk-mat nr nc f* = *mk-mat nr nc f'*
using *undef-cong-mat*[*of nr nc f f'*, *OF assms*]
using *assms* **unfolding** *mk-mat-def*
by *auto*

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

setup-lifting *type-definition-mat*

lift-definition *dim-row* :: *'a mat* ⇒ *nat* **is** *fst* **.**
lift-definition *dim-col* :: *'a mat* ⇒ *nat* **is** *fst o snd* **.**
lift-definition *index-mat* :: *'a mat* ⇒ (*nat* × *nat* ⇒ *'a*) (**infixl** ‹$$› *100*) **is** *snd o snd* **.**
lift-definition *mat* :: *nat* ⇒ *nat* ⇒ (*nat* × *nat* ⇒ *'a*) ⇒ *'a mat*
is λ *nr nc f*. (*nr*, *nc*, *mk-mat nr nc f*) **by** *auto*
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*)) **by** *auto*

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*
unfolding *upper-triangular-def* **by** *auto*

lemma *upper-triangularI*[*intro*] :
(⋀*i j*. *j* < *i* ⟹ *i* < *dim-row A* ⟹ *A* $$ (*i*,*j*) = *0*) ⟹ *upper-triangular A*
unfolding *upper-triangular-def* **by** *auto*

lemma *dim-row-mat*[*simp*]: *dim-row* (*mat nr nc f*) = *nr dim-row* (mat_r *nr nc g*) = *nr*
by (*transfer*, *simp*)+

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

nc
by (*transfer*, *simp*)+

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*)
unfolding *carrier-mat-def* **by** *auto*

lemma *mat-carrier*[*simp*]: *mat nr nc f* ∈ *carrier-mat nr nc*
unfolding *carrier-mat-def* **by** *auto*

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*)
unfolding *elements-mat-def* **by** *force*

lemma *elements-matI* [*intro*]:
A ∈ *carrier-mat nr nc* ⟹ *i* < *nr* ⟹ *j* < *nc* ⟹ *a* = *A* $$ (*i*,*j*) ⟹ *a* ∈ *elements-mat A*
unfolding *elements-mat-def carrier-mat-def* **by** *force*

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*
by (*transfer'*, *simp add*: *mk-mat-def*)+

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*
by (*transfer*, *auto intro*!: *cong-mk-mat*, *auto simp*: *mk-mat-def*)

lemma *carrier-matI*[*intro*]:
assumes *dim-row A* = *nr dim-col A* = *nc* **shows** *A* ∈ *carrier-mat nr nc*
using *assms* **unfolding** *carrier-mat-def* **by** *auto*

lemma *carrier-matD*[*dest*,*simp*]: **assumes** *A* ∈ *carrier-mat nr nc*
shows *dim-row A* = *nr dim-col A* = *nc* **using** *assms*
unfolding *carrier-mat-def* **by** *auto*

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'*
by (*rule eq-matI*, *insert assms*, *auto*)

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*) **unfolding** *row-def* **by** *auto*

lemma *rows-carrier*[*simp*]: *set* (*rows A*) ⊆ *carrier-vec* (*dim-col A*) **unfolding** *rows-def* **by** *auto*

lemma *length-rows*[*simp*]: *length* (*rows A*) = *dim-row A* **unfolding** *rows-def* **by** *auto*

lemma *nth-rows*[*simp*]: *i* < *dim-row A* ⟹ *rows A* ! *i* = *row A i*
unfolding *rows-def* **by** *auto*

lemma *row-mat-of-row-fun*[*simp*]: *i* < *nr* ⟹ *dim-vec* (*f i*) = *nc* ⟹ *row* (mat_r *nr nc f*) *i* = *f i*
by (*rule eq-vecI*, *auto simp*: *row-def*)

lemma *set-rows-carrier*:
assumes *A* ∈ *carrier-mat m n* **and** *v* ∈ *set* (*rows A*) **shows** *v* ∈ *carrier-vec n*
using *assms* **by** (*auto simp*: *rows-def row-def*)

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*
unfolding *mat-of-rows-def* **by** *auto*

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*
unfolding *mat-of-rows-def row-def* **using** *n i* **by** *auto*

lemma *rows-mat-of-rows*[*simp*]:
assumes *set vs* ⊆ *carrier-vec n* **shows** *rows* (*mat-of-rows n vs*) = *vs*
unfolding *rows-def* **apply** (*rule nth-equalityI*)
using *assms* **unfolding** *subset-code*(*1*) **by** *auto*

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

unfolding *mat-of-rows-def* **by** (*rule*, *auto simp*: *row-def*)

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*) **unfolding** *col-def* **by** *auto*

lemma *dim-col*[*simp*]: *dim-vec* (*col A i*) = *dim-row A* **by** *auto*

lemma *cols-dim*[*simp*]: *set* (*cols A*) ⊆ *carrier-vec* (*dim-row A*) **unfolding** *cols-def* **by** *auto*

lemma *cols-length*[*simp*]: *length* (*cols A*) = *dim-col A* **unfolding** *cols-def* **by** *auto*

lemma *cols-nth*[*simp*]: *i* < *dim-col A* ⟹ *cols A* ! *i* = *col A i*
unfolding *cols-def* **by** *auto*

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*
unfolding *mat-of-cols-def* **by** *auto*

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*
unfolding *mat-of-cols-def col-def* **using** *j n* **by** *auto*

lemma *cols-mat-of-cols*[*simp*]:
assumes *set vs* ⊆ *carrier-vec n* **shows** *cols* (*mat-of-cols n vs*) = *vs*
unfolding *cols-def* **apply**(*rule nth-equalityI*)
using *assms* **unfolding** *subset-code*(*1*) **by** *auto*

lemma *mat-of-cols-cols*[*simp*]:
mat-of-cols (*dim-row A*) (*cols A*) = *A*
unfolding *mat-of-cols-def* **by** (*rule*, *auto simp*: *col-def*)

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 ..
end

instantiation *mat* :: (*preorder*) *preorder*
begin
instance
proof (*standard*, *auto simp*: *less-mat-def less-eq-mat-def*, *goal-cases*)
case (*1 A B C i j*)
thus *?case* **using** *order-trans*[*of A* $$ (*i*,*j*) *B* $$ (*i*,*j*) *C* $$ (*i*,*j*)] **by** *auto*
qed
end

instantiation *mat* :: (*order*) *order*
begin
instance
by (*standard*, *intro eq-matI*, *auto simp*: *less-eq-mat-def order.antisym*)
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 ..
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*}
unfolding *elements-mat-def zero-mat-def* **by** *auto*

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 ..
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 ..
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 ..
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 = (+),

$\ldots = b$⦈

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*
unfolding *ring-mat-def* **by** *auto*

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*) = $(\cdot_m)$
unfolding *module-mat-def* **by** *auto*

lemma *index-zero-mat*[*simp*]: $i < nr \Longrightarrow j < nc \Longrightarrow 0_m$ *nr nc* \$\$ $(i,j) = 0$
dim-row (0_m *nr nc*) = *nr* *dim-col* (0_m *nr nc*) = *nc*
unfolding *zero-mat-def* **by** *auto*

lemma *index-one-mat*[*simp*]: $i < n \Longrightarrow j < n \Longrightarrow 1_m$ *n* \$\$ (i,j) = (*if* $i = j$ *then* 1 *else* 0)
dim-row (1_m *n*) = *n* *dim-col* (1_m *n*) = *n*
unfolding *one-mat-def* **by** *auto*

lemma *index-add-mat*[*simp*]:
$i <$ *dim-row* $B \Longrightarrow j <$ *dim-col* $B \Longrightarrow (A + B)$ \$\$ $(i,j) = A$ \$\$ $(i,j) + B$ \$\$ (i,j)
dim-row $(A + B)$ = *dim-row* B *dim-col* $(A + B)$ = *dim-col* B
unfolding *plus-mat-def* **by** *auto*

lemma *index-minus-mat*[*simp*]:
$i <$ *dim-row* $B \Longrightarrow j <$ *dim-col* $B \Longrightarrow (A - B)$ \$\$ $(i,j) = A$ \$\$ $(i,j) - B$ \$\$ (i,j)
dim-row $(A - B)$ = *dim-row* B *dim-col* $(A - B)$ = *dim-col* B
unfolding *minus-mat-def* **by** *auto*

lemma *index-map-mat*[*simp*]:

$i < \textit{dim-row}\ A \Longrightarrow j < \textit{dim-col}\ A \Longrightarrow \textit{map-mat}\ f\ A\ \$\$\ (i,j) = f\ (A\ \$\$\ (i,j))$
$\textit{dim-row}\ (\textit{map-mat}\ f\ A) = \textit{dim-row}\ A\ \ \textit{dim-col}\ (\textit{map-mat}\ f\ A) = \textit{dim-col}\ A$
unfolding *map-mat-def* **by** *auto*

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

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

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

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

lemma *dim-mult-mat-vec*[*simp*]: $\textit{dim-vec}\ (A *_v v) = \textit{dim-row}\ A$
by (*auto simp*: *mult-mat-vec-def*)

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

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

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

lemma *upper-triangular-one*[*simp*]: $\textit{upper-triangular}\ (1_m\ n)$
by (*rule*, *auto*)

lemma *upper-triangular-zero*[*simp*]: $\textit{upper-triangular}\ (0_m\ n\ n)$
by (*rule*, *auto*)

lemma *mat-row-carrierI*[*intro*,*simp*]: $\textit{mat}_r\ \textit{nr}\ \textit{nc}\ r \in \textit{carrier-mat}\ \textit{nr}\ \textit{nc}$
by (*unfold carrier-mat-def carrier-vec-def*, *auto*)

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 (*rule eq-matI*[*OF - dims*])
fix i j
assume i: $i <$ *dim-row* B **and** j: $j <$ *dim-col* B
from *rows*[*OF i*] **have** *id*: *row* A i \$ j = *row* B i \$ j **by** *simp*
show A \$\$ $(i, j) = B$ \$\$ (i, j)
using *index-row*(*1*)[*OF i j*, *folded id*] *index-row*(*1*)[*of i A j*] *i j dims*
by *auto*
qed

lemma *elements-mat-map*[*simp*]: *elements-mat* (*map-mat f A*) = *f* ' *elements-mat* *A*
by *fastforce*

lemma *row-mat*[*simp*]: $i < nr \Longrightarrow$ *row* (*mat nr nc f*) i = *vec nc* (λ j. f (i,j))
by *auto*

lemma *col-mat*[*simp*]: $j < nc \Longrightarrow$ *col* (*mat nr nc f*) j = *vec nr* (λ i. f (i,j))
by *auto*

lemma *zero-carrier-mat*[*simp*]: 0_m *nr nc* $\in$ *carrier-mat nr nc*
unfolding *carrier-mat-def* **by** *auto*

lemma *smult-carrier-mat*[*simp*]:
$A \in$ *carrier-mat nr nc* $\Longrightarrow k \cdot_m A \in$ *carrier-mat nr nc*
unfolding *carrier-mat-def* **by** *auto*

lemma *add-carrier-mat*[*simp*]:
$B \in$ *carrier-mat nr nc* $\Longrightarrow A + B \in$ *carrier-mat nr nc*
unfolding *carrier-mat-def* **by** *force*

lemma *one-carrier-mat*[*simp*]: 1_m $n \in$ *carrier-mat n n*
unfolding *carrier-mat-def* **by** *auto*

lemma *uminus-carrier-mat*:
$A \in$ *carrier-mat nr nc* $\Longrightarrow$ ($- A \in$ *carrier-mat nr nc*)
unfolding *carrier-mat-def* **by** *auto*

lemma *uminus-carrier-iff-mat*[*simp*]:
($- A \in$ *carrier-mat nr nc*) = ($A \in$ *carrier-mat nr nc*)
unfolding *carrier-mat-def* **by** *auto*

lemma *minus-carrier-mat*:
$B \in$ *carrier-mat nr nc* $\Longrightarrow$ ($A - B \in$ *carrier-mat nr nc*)
unfolding *carrier-mat-def* **by** *auto*

lemma *transpose-carrier-mat*[*simp*]: (*transpose-mat* $A \in$ *carrier-mat nc nr*) = (A $\in$ *carrier-mat nr nc*)

unfolding *carrier-mat-def* **by** *auto*

lemma *row-carrier-vec*[*simp*]: $i < nr \Longrightarrow A \in carrier\text{-}mat\ nr\ nc \Longrightarrow row\ A\ i \in carrier\text{-}vec\ nc$
unfolding *carrier-vec-def* **by** *auto*

lemma *col-carrier-vec*[*simp*]: $j < nc \Longrightarrow A \in carrier\text{-}mat\ nr\ nc \Longrightarrow col\ A\ j \in carrier\text{-}vec\ nr$
unfolding *carrier-vec-def* **by** *auto*

lemma *mult-carrier-mat*[*simp*]:
$A \in carrier\text{-}mat\ nr\ n \Longrightarrow B \in carrier\text{-}mat\ n\ nc \Longrightarrow A * B \in carrier\text{-}mat\ nr\ nc$
unfolding *carrier-mat-def* **by** *auto*

lemma *mult-mat-vec-carrier*[*simp*]:
$A \in carrier\text{-}mat\ nr\ n \Longrightarrow v \in carrier\text{-}vec\ n \Longrightarrow A *_v v \in carrier\text{-}vec\ nr$
unfolding *carrier-mat-def carrier-vec-def* **by** *auto*

lemma *comm-add-mat*[*ac-simps*]:
$(A :: 'a :: comm\text{-}monoid\text{-}add\ mat) \in carrier\text{-}mat\ nr\ nc \Longrightarrow B \in carrier\text{-}mat\ nr\ nc \Longrightarrow A + B = B + A$
by (*intro eq-matI*, *auto simp*: *ac-simps*)

lemma *minus-r-inv-mat*[*simp*]:
$(A :: 'a :: group\text{-}add\ mat) \in carrier\text{-}mat\ nr\ nc \Longrightarrow (A - A) = 0_m\ nr\ nc$
by (*intro eq-matI*, *auto*)

lemma *uminus-l-inv-mat*[*simp*]:
$(A :: 'a :: group\text{-}add\ mat) \in carrier\text{-}mat\ nr\ nc \Longrightarrow (-\ A + A) = 0_m\ nr\ nc$
by (*intro eq-matI*, *auto*)

lemma *add-inv-exists-mat*:
$(A :: 'a :: group\text{-}add\ mat) \in carrier\text{-}mat\ nr\ nc \Longrightarrow \exists\ B \in carrier\text{-}mat\ nr\ nc.\ B + A = 0_m\ nr\ nc \wedge A + B = 0_m\ nr\ nc$
by (*intro bexI*[*of - − A*], *auto*)

lemma *assoc-add-mat*[*simp*]:
$(A :: 'a :: monoid\text{-}add\ mat) \in carrier\text{-}mat\ nr\ nc \Longrightarrow B \in carrier\text{-}mat\ nr\ nc \Longrightarrow C \in carrier\text{-}mat\ nr\ nc$
$\Longrightarrow (A + B) + C = A + (B + C)$
by (*intro eq-matI*, *auto simp*: *ac-simps*)

lemma *uminus-add-mat*: **fixes** $A :: 'a :: group\text{-}add\ mat$
assumes $A \in carrier\text{-}mat\ nr\ nc$
and $B \in carrier\text{-}mat\ nr\ nc$
shows $-\ (A + B) = -\ B + -\ A$
by (*intro eq-matI*, *insert assms*, *auto simp*: *minus-add*)

lemma *transpose-transpose*[*simp*]:
  transpose-mat (*transpose-mat* A) = A
  by (*intro eq-matI*, *auto*)

lemma *transpose-one*[*simp*]: *transpose-mat* (1_m n) = (1_m n)
  by *auto*

lemma *row-transpose*[*simp*]:
  j < *dim-col* A $\Longrightarrow$ *row* (*transpose-mat* A) j = *col* A j
  unfolding *row-def col-def*
  by (*intro eq-vecI*, *auto*)

lemma *col-transpose*[*simp*]:
  i < *dim-row* A $\Longrightarrow$ *col* (*transpose-mat* A) i = *row* A i
  unfolding *row-def col-def*
  by (*intro eq-vecI*, *auto*)

lemma *row-zero*[*simp*]:
  i < *nr* $\Longrightarrow$ *row* (0_m *nr nc*) i = 0_v *nc*
   by (*intro eq-vecI*, *auto*)

lemma *col-zero*[*simp*]:
  j < *nc* $\Longrightarrow$ *col* (0_m *nr nc*) j = 0_v *nr*
   by (*intro eq-vecI*, *auto*)

lemma *row-one*[*simp*]:
  i < n $\Longrightarrow$ *row* (1_m n) i = *unit-vec* n i
  by (*intro eq-vecI*, *auto*)

lemma *col-one*[*simp*]:
  j < n $\Longrightarrow$ *col* (1_m n) j = *unit-vec* n j
  by (*intro eq-vecI*, *auto*)

lemma *transpose-add*: $A \in$ *carrier-mat nr nc* $\Longrightarrow$ $B \in$ *carrier-mat nr nc*
  $\Longrightarrow$ *transpose-mat* ($A + B$) = *transpose-mat* A + *transpose-mat* B
  by (*intro eq-matI*, *auto*)

lemma *transpose-minus*: $A \in$ *carrier-mat nr nc* $\Longrightarrow$ $B \in$ *carrier-mat nr nc*
  $\Longrightarrow$ *transpose-mat* ($A - B$) = *transpose-mat* A − *transpose-mat* B
  by (*intro eq-matI*, *auto*)

lemma *transpose-uminus*: *transpose-mat* (− A) = − (*transpose-mat* A)
  by (*intro eq-matI*, *auto*)

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

$(A + B)$ i = *row* A i + *row* B i
  by (*rule eq-vecI*, *auto*)

lemma *col-add*[*simp*]:
  $A \in$ *carrier-mat nr nc* $\Longrightarrow B \in$ *carrier-mat nr nc* $\Longrightarrow j <$ *nc*
  $\Longrightarrow$ *col* $(A + B)$ j = *col* A j + *col* B j
  by (*rule eq-vecI*, *auto*)

lemma *row-mult*[*simp*]: **assumes** *m*: $A \in$ *carrier-mat nr n* $B \in$ *carrier-mat n nc*
  and *i*: $i <$ *nr*
  shows *row* $(A * B)$ i = *vec nc* (λ *j*. *row* A i $\cdot$ *col* B j)
  by (*rule eq-vecI*, *insert m i*, *auto*)

lemma *col-mult*[*simp*]: **assumes** *m*: $A \in$ *carrier-mat nr n* $B \in$ *carrier-mat n nc*
  and *j*: $j <$ *nc*
  shows *col* $(A * B)$ j = *vec nr* (λ *i*. *row* A i $\cdot$ *col* B j)
  by (*rule eq-vecI*, *insert m j*, *auto*)

lemma *transpose-mult*:
  (A :: '*a* :: *comm-semiring-0 mat*) $\in$ *carrier-mat nr n* $\Longrightarrow B \in$ *carrier-mat n nc*
  $\Longrightarrow$ *transpose-mat* $(A * B)$ = *transpose-mat* B $*$ *transpose-mat* A
  by (*intro eq-matI*, *auto simp*: *comm-scalar-prod*[*of - n*])

lemma *left-add-zero-mat*[*simp*]:
  (A :: '*a* :: *monoid-add mat*) $\in$ *carrier-mat nr nc* $\Longrightarrow$ 0_m *nr nc* + A = A
  by (*intro eq-matI*, *auto*)

lemma *add-uminus-minus-mat*: $A \in$ *carrier-mat nr nc* $\Longrightarrow B \in$ *carrier-mat nr nc*
$\Longrightarrow$
  $A + (- B) = A -$ (B :: '*a* :: *group-add mat*)
  by (*intro eq-matI*, *auto*)

lemma *right-add-zero-mat*[*simp*]: $A \in$ *carrier-mat nr nc* $\Longrightarrow$
  A + 0_m *nr nc* = (A :: '*a* :: *monoid-add mat*)
  by (*intro eq-matI*, *auto*)

lemma *left-mult-zero-mat*:
  $A \in$ *carrier-mat n nc* $\Longrightarrow$ 0_m *nr n* $*$ A = 0_m *nr nc*
  by (*intro eq-matI*, *auto*)

lemma *left-mult-zero-mat'*[*simp*]: *dim-row* A = n $\Longrightarrow$ 0_m *nr n* $*$ A = 0_m *nr*
(*dim-col* A)
  by (*rule left-mult-zero-mat*, *unfold carrier-mat-def*, *simp*)

lemma *right-mult-zero-mat*:
  $A \in$ *carrier-mat nr n* $\Longrightarrow$ A $*$ 0_m *n nc* = 0_m *nr nc*
  by (*intro eq-matI*, *auto*)

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

A) *nc*
by (*rule right-mult-zero-mat*, *unfold carrier-mat-def*, *simp*)

lemma *left-mult-one-mat*:
$(A :: 'a :: semiring\text{-}1\ mat) \in carrier\text{-}mat\ nr\ nc \Longrightarrow 1_m\ nr * A = A$
by (*intro eq-matI*, *auto*)

lemma *left-mult-one-mat'*[*simp*]: $dim\text{-}row\ (A :: 'a :: semiring\text{-}1\ mat) = n \Longrightarrow 1_m\ n * A = A$
by (*rule left-mult-one-mat*, *unfold carrier-mat-def*, *simp*)

lemma *right-mult-one-mat*:
$(A :: 'a :: semiring\text{-}1\ mat) \in carrier\text{-}mat\ nr\ nc \Longrightarrow A * 1_m\ nc = A$
by (*intro eq-matI*, *auto*)

lemma *right-mult-one-mat'*[*simp*]: $dim\text{-}col\ (A :: 'a :: semiring\text{-}1\ mat) = n \Longrightarrow A * 1_m\ n = A$
by (*rule right-mult-one-mat*, *unfold carrier-mat-def*, *simp*)

lemma *one-mult-mat-vec*[*simp*]:
$(v :: 'a :: semiring\text{-}1\ vec) \in carrier\text{-}vec\ n \Longrightarrow 1_m\ n *_v v = v$
by (*intro eq-vecI*, *auto*)

lemma *minus-add-uminus-mat*: **fixes** $A :: 'a :: group\text{-}add\ mat$
shows $A \in carrier\text{-}mat\ nr\ nc \Longrightarrow B \in carrier\text{-}mat\ nr\ nc \Longrightarrow$
$A - B = A + (- B)$
by (*intro eq-matI*, *auto*)

lemma *add-mult-distrib-mat*[*algebra-simps*]: **assumes** *m*: $A \in carrier\text{-}mat\ nr\ n$
$B \in carrier\text{-}mat\ nr\ n$ $C \in carrier\text{-}mat\ n\ nc$
shows $(A + B) * C = A * C + B * C$
using *m* **by** (*intro eq-matI*, *auto simp*: *add-scalar-prod-distrib*[*of - n*])

lemma *mult-add-distrib-mat*[*algebra-simps*]: **assumes** *m*: $A \in carrier\text{-}mat\ nr\ n$
$B \in carrier\text{-}mat\ n\ nc$ $C \in carrier\text{-}mat\ n\ nc$
shows $A * (B + C) = A * B + A * C$
using *m* **by** (*intro eq-matI*, *auto simp*: *scalar-prod-add-distrib*[*of - n*])

lemma *add-mult-distrib-mat-vec*[*algebra-simps*]: **assumes** *m*: $A \in carrier\text{-}mat\ nr\ nc$
$B \in carrier\text{-}mat\ nr\ nc$ $v \in carrier\text{-}vec\ nc$
shows $(A + B) *_v v = A *_v v + B *_v v$
using *m* **by** (*intro eq-vecI*, *auto intro*!: *add-scalar-prod-distrib*)

lemma *mult-add-distrib-mat-vec*[*algebra-simps*]: **assumes** *m*: $A \in carrier\text{-}mat\ nr\ nc$
$v_1 \in carrier\text{-}vec\ nc$ $v_2 \in carrier\text{-}vec\ nc$
shows $A *_v (v_1 + v_2) = A *_v v_1 + A *_v v_2$
using *m* **by** (*intro eq-vecI*, *auto simp*: *scalar-prod-add-distrib*[*of - nc*])

lemma *mult-mat-vec*:
  assumes *m*: $(A::'a::field\ mat) \in carrier\text{-}mat\ nr\ nc$ **and** *v*: $v \in carrier\text{-}vec\ nc$
  shows $A *_v (k \cdot_v v) = k \cdot_v (A *_v v)$ (**is** $?l = ?r$)
proof
  have *nr*: $dim\text{-}vec\ ?l = nr$ **using** *m v* **by** *auto*
  also have ... $= dim\text{-}vec\ ?r$ **using** *m v* **by** *auto*
  finally show $dim\text{-}vec\ ?l = dim\text{-}vec\ ?r$**.**

  show $\bigwedge i.\ i < dim\text{-}vec\ ?r \Longrightarrow ?l\ \$\ i = ?r\ \$\ i$
  proof −
    fix *i* **assume** $i < dim\text{-}vec\ ?r$
    hence *i*: $i < dim\text{-}row\ A$ **using** *nr m* **by** *auto*
    hence *i2*: $i < dim\text{-}vec\ (A *_v v)$ **using** *m* **by** *auto*
    show $?l\ \$\ i = ?r\ \$\ i$
    apply (*subst* (*1*) *mult-mat-vec-def*)
    apply (*subst* (*2*) *smult-vec-def*)
    unfolding *index-vec*[*OF i*] *index-vec*[*OF i2*]
    unfolding *mult-mat-vec-def smult-vec-def*
    unfolding *scalar-prod-def index-vec*[*OF i*]
    by (*simp add*: *mult.left-commute sum-distrib-left*)
  qed
qed

lemma *assoc-scalar-prod*: **assumes** $*$: $v_1 \in carrier\text{-}vec\ nr\ A \in carrier\text{-}mat\ nr\ nc$
$v_2 \in carrier\text{-}vec\ nc$
  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 −
  have $vec\ nc\ (\lambda j.\ v_1 \cdot col\ A\ j) \cdot v_2 = (\sum i\in\{0..<nc\}.\ vec\ nc\ (\lambda j.\ \sum k\in\{0..<nr\}.$
$v_1\ \$\ k * col\ A\ j\ \$\ k)\ \$\ i * v_2\ \$\ i)$
    unfolding *scalar-prod-def* **using** $*$ **by** *auto*
  also have $\ldots = (\sum i\in\{0..<nc\}.\ (\sum k\in\{0..<nr\}.\ v_1\ \$\ k * col\ A\ i\ \$\ k) * v_2\ \$\ i)$
    by (*rule sum.cong*, *auto*)
  also have $\ldots = (\sum i\in\{0..<nc\}.\ (\sum k\in\{0..<nr\}.\ v_1\ \$\ k * col\ A\ i\ \$\ k * v_2\ \$\ i))$
    unfolding *sum-distrib-right* **..**
  also have $\ldots = (\sum k\in\{0..<nr\}.\ (\sum i\in\{0..<nc\}.\ v_1\ \$\ k * col\ A\ i\ \$\ k * v_2\ \$\ i))$
    by (*rule sum.swap*)
  also have $\ldots = (\sum k\in\{0..<nr\}.\ (\sum i\in\{0..<nc\}.\ v_1\ \$\ k * (col\ A\ i\ \$\ k * v_2\ \$$
$i)))$
    by (*simp add*: *ac-simps*)
  also have $\ldots = (\sum k\in\{0..<nr\}.\ v_1\ \$\ k * (\sum i\in\{0..<nc\}.\ col\ A\ i\ \$\ k * v_2\ \$\ i))$
    unfolding *sum-distrib-left* **..**
  also have $\ldots = (\sum k\in\{0..<nr\}.\ v_1\ \$\ k * vec\ nr\ (\lambda k.\ \sum i\in\{0..<nc\}.\ row\ A\ k\ \$$
$i * v_2\ \$\ i)\ \$\ k)$
    using $*$ **by** *auto*
  also have $\ldots = v_1 \cdot vec\ nr\ (\lambda i.\ row\ A\ i \cdot v_2)$ **unfolding** *scalar-prod-def* **using**
$*$ **by** *simp*
  finally show *?thesis* **.**
qed

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 −
  have (*transpose-mat* $A *_v y$) = *vec nc* ($\lambda i.$ *col* A $i \cdot y$)
    unfolding *mult-mat-vec-def* **using** A **by** *auto*
  also have ... = *vec nc* ($\lambda i.$ $y \cdot$ *col* A i)
    by (*intro eq-vecI*, *simp*, *rule comm-scalar-prod*[*OF* - *y*], *insert A*, *auto*)
  also have ... $\cdot$ x = $y \cdot$ *vec nr* ($\lambda i.$ *row* A $i \cdot x$)
    by (*rule assoc-scalar-prod*[*OF y A x*])
  also have *vec nr* ($\lambda i.$ *row* A $i \cdot x$) = $A *_v x$
    unfolding *mult-mat-vec-def* **using** A **by** *auto*
  finally show *?thesis* **.**
qed

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
  $\Longrightarrow (A * B) * C = A * (B * C)$
  by (*intro eq-matI*, *auto simp*: *assoc-scalar-prod*)

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
  $\Longrightarrow (A * B) *_v v = A *_v (B *_v v)$
  by (*intro eq-vecI*, *auto simp add*: *mult-mat-vec-def assoc-scalar-prod*)

lemma *comm-monoid-mat*: *comm-monoid* (*monoid-mat TYPE*($'a$ :: *comm-monoid-add*) *nr nc*)
  by (*unfold-locales*, *auto simp*: *monoid-mat-def ac-simps*)

lemma *comm-group-mat*: *comm-group* (*monoid-mat TYPE*($'a$ :: *ab-group-add*) *nr nc*)
  by (*unfold-locales*, *insert add-inv-exists-mat*, *auto simp*: *monoid-mat-def ac-simps Units-def*)

lemma *semiring-mat*: *semiring* (*ring-mat TYPE*($'a$ :: *semiring-1*) *n b*)
  by (*unfold-locales*, *auto simp*: *ring-mat-def algebra-simps*)

lemma *ring-mat*: *ring* (*ring-mat TYPE*($'a$ :: *comm-ring-1*) *n b*)
  by (*unfold-locales*, *insert add-inv-exists-mat*, *auto simp*: *ring-mat-def algebra-simps Units-def*)

lemma *abelian-group-mat*: *abelian-group* (*module-mat TYPE*($'a$ :: *comm-ring-1*) *nr nc*)
  by (*unfold-locales*, *insert add-inv-exists-mat*, *auto simp*: *module-mat-def Units-def*)

lemma *row-smult*[*simp*]: **assumes** *i*: $i < dim\text{-}row\ A$
shows *row* $(k \cdot_m A)\ i = k \cdot_v (row\ A\ i)$
by (*rule eq-vecI*, *insert i*, *auto*)

lemma *col-smult*[*simp*]: **assumes** *i*: $i < dim\text{-}col\ A$
shows *col* $(k \cdot_m A)\ i = k \cdot_v (col\ A\ i)$
by (*rule eq-vecI*, *insert i*, *auto*)

lemma *row-uminus*[*simp*]: **assumes** *i*: $i < dim\text{-}row\ A$
shows *row* $(- A)\ i = - (row\ A\ i)$
by (*rule eq-vecI*, *insert i*, *auto*)

lemma *scalar-prod-uminus-left*[*simp*]: **assumes** *dim*: $dim\text{-}vec\ v = dim\text{-}vec\ (w :: 'a :: ring\ vec)$
shows $- v \cdot w = - (v \cdot w)$
unfolding *scalar-prod-def dim*[*symmetric*]
by (*subst sum-negf*[*symmetric*], *rule sum.cong*, *auto*)

lemma *col-uminus*[*simp*]: **assumes** *i*: $i < dim\text{-}col\ A$
shows *col* $(- A)\ i = - (col\ A\ i)$
by (*rule eq-vecI*, *insert i*, *auto*)

lemma *scalar-prod-uminus-right*[*simp*]: **assumes** *dim*: $dim\text{-}vec\ v = dim\text{-}vec\ (w :: 'a :: ring\ vec)$
shows $v \cdot - w = - (v \cdot w)$
unfolding *scalar-prod-def dim*
by (*subst sum-negf*[*symmetric*], *rule sum.cong*, *auto*)

context fixes $A\ B :: 'a :: ring\ mat$
assumes *dim*: $dim\text{-}col\ A = dim\text{-}row\ B$
begin
lemma *uminus-mult-left-mat*[*simp*]: $(- A * B) = - (A * B)$
by (*intro eq-matI*, *insert dim*, *auto*)

lemma *uminus-mult-right-mat*[*simp*]: $(A * - B) = - (A * B)$
by (*intro eq-matI*, *insert dim*, *auto*)
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 $(A - B) * C = A * C - B * C$
unfolding *minus-add-uminus-mat*[*OF m*(*1*,*2*)]
add-mult-distrib-mat[*OF m*(*1*) *uminus-carrier-mat*[*OF m*(*2*)] *m*(*3*)]
by (*subst uminus-mult-left-mat*, *insert m*, *auto*)

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

shows $(A - B) *_v v = A *_v v - B *_v v$
  unfolding *minus-add-uminus-mat*[*OF A B*]
  by (*subst add-mult-distrib-mat-vec*[*OF A - v*], *insert A B v*, *auto*)

lemma *mult-minus-distrib-mat-vec*[*algebra-simps*]: **assumes** *A*: (*A* :: *'a* :: *ring mat*) $\in$ *carrier-mat nr nc*
  and *v*: $v \in$ *carrier-vec nc*
  and *w*: $w \in$ *carrier-vec nc*
shows $A *_v (v - w) = A *_v v - A *_v w$
  unfolding *minus-add-uminus-vec*[*OF v w*]
  by (*subst mult-add-distrib-mat-vec*[*OF A*], *insert A v w*, *auto*)

lemma *mult-minus-distrib-mat*[*algebra-simps*]: **fixes** *A* :: *'a* :: *ring mat*
  assumes *m*: $A \in$ *carrier-mat nr n* $B \in$ *carrier-mat n nc* $C \in$ *carrier-mat n nc*
  shows $A * (B - C) = A * B - A * C$
  unfolding *minus-add-uminus-mat*[*OF m*(*2*,*3*)]
    mult-add-distrib-mat[*OF m*(*1*) *m*(*2*) *uminus-carrier-mat*[*OF m*(*3*)]]
  by (*subst uminus-mult-right-mat*, *insert m*, *auto*)

lemma *uminus-mult-mat-vec*[*simp*]: **assumes** *v*: *dim-vec v* = *dim-col* (*A* :: *'a* :: *ring mat*)
  shows $- A *_v v = - (A *_v v)$
  using *v* **by** (*intro eq-vecI*, *auto*)

lemma *uminus-zero-vec-eq*: **assumes** *v*: (*v* :: *'a* :: *group-add vec*) $\in$ *carrier-vec n*
  shows $(- v = 0_v\ n) = (v = 0_v\ n)$
proof
  assume *z*: $- v = 0_v\ n$
  {
    fix *i*
    assume *i*: $i < n$
    have $v \$ i = - (- (v \$ i))$ **by** *simp*
    also have $- (v \$ i) = 0$ **using** *arg-cong*[*OF z*, *of* λ *v*. *v* \$ *i*] *i v* **by** *auto*
    also have $- 0 = (0 :: 'a)$ **by** *simp*
    finally have $v \$ i = 0$ **.**
  }
  thus $v = 0_v\ n$ **using** *v*
    by (*intro eq-vecI*, *auto*)
qed *auto*

lemma *map-carrier-mat*[*simp*]:
  (*map-mat f A* $\in$ *carrier-mat nr nc*) = (*A* $\in$ *carrier-mat nr nc*)
  unfolding *carrier-mat-def* **by** *auto*

lemma *col-map-mat*[*simp*]:
  assumes $j <$ *dim-col A* **shows** *col* (*map-mat f A*) *j* = *map-vec f* (*col A j*)
  unfolding *map-mat-def map-vec-def* **using** *assms* **by** *auto*

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

by (*rule eq-vecI*, *auto*)

lemma *scalar-prod-smult-right*[*simp*]:
dim-vec w = *dim-vec v* $\Longrightarrow$ *w* $\cdot$ (*k* $\cdot_v$ *v*) = (*k* :: *'a* :: *comm-semiring-0*) $*$ (*w* $\cdot$ *v*)
unfolding *scalar-prod-def sum-distrib-left*
by (*auto intro*: *sum.cong simp*: *ac-simps*)

lemma *scalar-prod-smult-left*[*simp*]:
dim-vec w = *dim-vec v* $\Longrightarrow$ (*k* $\cdot_v$ *w*) $\cdot$ *v* = (*k* :: *'a* :: *comm-semiring-0*) $*$ (*w* $\cdot$ *v*)
unfolding *scalar-prod-def sum-distrib-left*
by (*auto intro*: *sum.cong simp*: *ac-simps*)

lemma *mult-smult-distrib*: **assumes** *A*: *A* $\in$ *carrier-mat nr n* **and** *B*: *B* $\in$ *carrier-mat n nc*
shows *A* $*$ (*k* $\cdot_m$ *B*) = (*k* :: *'a* :: *comm-semiring-0*) $\cdot_m$ (*A* $*$ *B*)
by (*rule eq-matI*, *insert A B*, *auto*)

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* :: *semiring*) $\cdot_m$ *A* + *k* $\cdot_m$ *B*
by (*rule eq-matI*, *insert assms*, *auto simp*: *field-simps*)

lemma *add-smult-distrib-right-mat*: **assumes** *A* $\in$ *carrier-mat nr nc*
shows (*k* + *l*) $\cdot_m$ *A* = (*k* :: *'a* :: *semiring*) $\cdot_m$ *A* + *l* $\cdot_m$ *A*
by (*rule eq-matI*, *insert assms*, *auto simp*: *field-simps*)

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* :: *comm-semiring-0*) $\cdot_m$ (*A* $*$ *B*)
by (*rule eq-matI*, *insert A B*, *auto*)

definition *similar-mat-wit* :: *'a* :: *semiring-1 mat* $\Rightarrow$ *'a mat* $\Rightarrow$ *'a mat* $\Rightarrow$ *'a mat* $\Rightarrow$ *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* $\Rightarrow$ *'a mat* $\Rightarrow$ *bool* **where**
similar-mat A B = ($\exists$ *P Q*. *similar-mat-wit A B P Q*)

lemma *similar-matD*: **assumes** *similar-mat A B*
shows $\exists$ *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*
using *assms* **unfolding** *similar-mat-def similar-mat-wit-def*[*abs-def*] *Let-def* **by** *blast*

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* **unfolding** *similar-mat-def*

by (*rule exI*[*of* - *P*], *rule exI*[*of* - *Q*], *unfold similar-mat-wit-def Let-def*, *insert assms*, *auto*)

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*)
by (*induct k*, *auto*)

lemma *pow-mat-dim-square*[*simp*]:
A ∈ *carrier-mat n n* ⟹ *dim-row* ($A \widehat{\ }_m k$) = *n*
A ∈ *carrier-mat n n* ⟹ *dim-col* ($A \widehat{\ }_m k$) = *n*
by *auto*

lemma *pow-carrier-mat*[*simp*]: *A* ∈ *carrier-mat n n* ⟹ $A \widehat{\ }_m k$ ∈ *carrier-mat n n*
unfolding *carrier-mat-def* **by** *auto*

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*) = (∏ *i* = *0* ..< *dim-row A*. *A* $$ (*i*,*i*))
unfolding *diag-mat-def*
by (*subst prod.distinct-set-conv-list*[*symmetric*], *auto*)

lemma *diag-mat-transpose*[*simp*]: *dim-row A* = *dim-col A* ⟹
diag-mat (*transpose-mat A*) = *diag-mat A* **unfolding** *diag-mat-def* **by** *auto*

lemma *diag-mat-zero*[*simp*]: *diag-mat* (0_m *n n*) = *replicate n 0*
unfolding *diag-mat-def*
by (*rule nth-equalityI*, *auto*)

lemma *diag-mat-one*[*simp*]: *diag-mat* (1_m *n*) = *replicate n 1*
unfolding *diag-mat-def*
by (*rule nth-equalityI*, *auto*)

lemma *pow-mat-ring-pow*: **assumes** *A*: (*A* :: (*'a* :: *semiring-1*)*mat*) ∈ *carrier-mat n n*
shows $A \widehat{\ }_m k = A \ [\urcorner]_{ring\text{-}mat\ TYPE('a)\ n\ b}\ ^k$
(**is** - = $A\ [\urcorner]_{?C}\ k$)
proof −
interpret *semiring ?C* **by** (*rule semiring-mat*)
show *?thesis*
by (*induct k*, *insert A*, *auto simp*: *ring-mat-def nat-pow-def*)
qed

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*)
unfolding *sum-mat-def*
by (*rule sum.neutral*, *auto*)

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 −
from *A B* **have** *id*: *dim-row A* = *nr dim-row B* = *nr dim-col A* = *nc dim-col B* = *nc*
by *auto*
show *?thesis* **unfolding** *sum-mat-def id*
by (*subst sum.distrib*[*symmetric*], *rule sum.cong*, *insert A B*, *auto*)
qed

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*]:
dim-vec (*v* $|_v$ *i* ↦ *a*) = *dim-vec v* **unfolding** *update-vec-def* **by** *simp*

lemma *index-update-vec1*[*simp*]:
assumes *i* < *dim-vec v* **shows** (*v* $|_v$ *i* ↦ *a*) $ *i* = *a*
unfolding *update-vec-def* **using** *assms* **by** *simp*

lemma *index-update-vec2*[*simp*]:
assumes *i'* ≠ *i* **shows** (*v* $|_v$ *i* ↦ *a*) $ *i'* = *v* $ *i'*
unfolding *update-vec-def*
using *assms* **apply** *transfer* **unfolding** *mk-vec-def* **by** *auto*

lemma *dim-update-mat*[*simp*]:
dim-row (*A* $|_m$ *ij* ↦ *a*) = *dim-row A*
dim-col (*A* $|_m$ *ij* ↦ *a*) = *dim-col A* **unfolding** *update-mat-def* **by** *simp+*

lemma *index-update-mat1*[*simp*]:

assumes $i < \mathit{dim\text{-}row}\ A\ j < \mathit{dim\text{-}col}\ A$ **shows** $(A\ |_m\ (i,j) \mapsto a)\ \$\$\ (i,j) = a$
unfolding *update-mat-def* **using** *assms* **by** *simp*

lemma *index-update-mat2*[*simp*]:
assumes i': $i' < \mathit{dim\text{-}row}\ A$ **and** j': $j' < \mathit{dim\text{-}col}\ A$ **and** *neq*: $(i',j') \neq ij$
shows $(A\ |_m\ ij \mapsto a)\ \$\$\ (i',j') = A\ \$\$\ (i',j')$
unfolding *update-mat-def* **using** *assms* **by** *auto*

4.4 Block Vectors and Matrices

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

lemma *index-append-vec*[*simp*]: $i < \mathit{dim\text{-}vec}\ v + \mathit{dim\text{-}vec}\ w$
$\Longrightarrow (v\ @_v\ w)\ \$\ i = (\mathit{if}\ i < \mathit{dim\text{-}vec}\ v\ \mathit{then}\ v\ \$\ i\ \mathit{else}\ w\ \$\ (i - \mathit{dim\text{-}vec}\ v))$
$\mathit{dim\text{-}vec}\ (v\ @_v\ w) = \mathit{dim\text{-}vec}\ v + \mathit{dim\text{-}vec}\ w$
unfolding *append-vec-def Let-def* **by** *auto*

lemma *append-carrier-vec*[*simp,intro*]:
$v \in \mathit{carrier\text{-}vec}\ n1 \Longrightarrow w \in \mathit{carrier\text{-}vec}\ n2 \Longrightarrow v\ @_v\ w \in \mathit{carrier\text{-}vec}\ (n1 + n2)$
unfolding *carrier-vec-def* **by** *auto*

lemma *scalar-prod-append*: **assumes** $v1 \in \mathit{carrier\text{-}vec}\ n1\ v2 \in \mathit{carrier\text{-}vec}\ n2$
$w1 \in \mathit{carrier\text{-}vec}\ n1\ w2 \in \mathit{carrier\text{-}vec}\ n2$
shows $(v1\ @_v\ v2) \cdot (w1\ @_v\ w2) = v1 \cdot w1 + v2 \cdot w2$
proof −
from *assms* **have** *dim*: $\mathit{dim\text{-}vec}\ v1 = n1\ \mathit{dim\text{-}vec}\ v2 = n2\ \mathit{dim\text{-}vec}\ w1 = n1$
$\mathit{dim\text{-}vec}\ w2 = n2$ **by** *auto*
have *id*: $\{0\ ..< n1 + n2\} = \{0\ ..< n1\} \cup \{n1\ ..< n1 + n2\}$ **by** *auto*
have *id2*: $\{n1\ ..< n1 + n2\} = (\mathit{plus}\ n1)\ `\ \{0\ ..< n2\}$
by (*simp add*: *ac-simps*)
have $(v1\ @_v\ v2) \cdot (w1\ @_v\ w2) = (\sum i = 0..<n1.\ v1\ \$\ i * w1\ \$\ i) +$
$(\sum i = n1..<n1 + n2.\ v2\ \$\ (i - n1) * w2\ \$\ (i - n1))$
unfolding *scalar-prod-def*
by (*auto simp*: *dim id*, *subst sum.union-disjoint*, *insert assms*, *force+*)
also have $(\sum i = n1..<n1 + n2.\ v2\ \$\ (i - n1) * w2\ \$\ (i - n1))$
$= (\sum i = 0..< n2.\ v2\ \$\ i * w2\ \$\ i)$
by (*rule sum.reindex-cong* [*OF - id2*]) *simp-all*
finally show *?thesis* **by** (*simp*, *insert assms*, *auto simp*: *scalar-prod-def*)
qed

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

lemma *dim-vec-first*[*simp*]: $\mathit{dim\text{-}vec}\ (\mathit{vec\text{-}first}\ v\ n) = n$ **unfolding** *vec-first-def* **by** *auto*
lemma *dim-vec-last*[*simp*]: $\mathit{dim\text{-}vec}\ (\mathit{vec\text{-}last}\ v\ n) = n$ **unfolding** *vec-last-def* **by** *auto*

lemma *vec-first-carrier*[*simp*]: *vec-first v n* $\in$ *carrier-vec n* **by** (*rule carrier-vecI*, *auto*)
lemma *vec-last-carrier*[*simp*]: *vec-last v n* $\in$ *carrier-vec n* **by** (*rule carrier-vecI*, *auto*)

lemma *vec-first-last-append*[*simp*]:
  assumes $v \in$ *carrier-vec* $(n+m)$ **shows** *vec-first v n* $@_v$ *vec-last v m* $= v$
  apply(*rule*) **unfolding** *vec-first-def vec-last-def* **using** *assms* **by** *auto*

lemma *append-vec-le*: **assumes** $v \in$ *carrier-vec n* **and** w: $w \in$ *carrier-vec n*
  shows $v\ @_v\ v' \le w\ @_v\ w' \longleftrightarrow v \le w \wedge v' \le w'$
proof –
  {
    fix i
    assume $*$: $\forall i.\ (\neg\ i < n \longrightarrow i < n +$ *dim-vec* $w' \longrightarrow v'\ \$\ (i - n) \le w'\ \$\ (i - n))$
      and i: $i <$ *dim-vec* w'
    have $v'\ \$\ i \le w'\ \$\ i$ **using** $*$[*rule-format*, *of* $n + i$] i **by** *auto*
  }
  thus *?thesis* **using** *assms* **unfolding** *less-eq-vec-def* **by** *auto*
qed

lemma *all-vec-append*: $(\forall\ x \in$ *carrier-vec* $(n + m).\ P\ x) \longleftrightarrow (\forall\ x1 \in$ *carrier-vec* $n.\ \forall\ x2 \in$ *carrier-vec* $m.\ P\ (x1\ @_v\ x2))$
proof (*standard*, *force*, *intro ballI*, *goal-cases*)
  case (*1 x*)
  have $x =$ *vec n* $(\lambda\ i.\ x\ \$\ i)\ @_v$ *vec m* $(\lambda\ i.\ x\ \$\ (n + i))$
    by (*rule eq-vecI*, *insert 1*(*2*), *auto*)
  hence $P\ x = P$ (*vec n* $(\lambda\ i.\ x\ \$\ i)\ @_v$ *vec m* $(\lambda\ i.\ x\ \$\ (n + i)))$ **by** *simp*
  also have ... **using** *1* **by** *auto*
  finally show *?case* **.**
qed

definition *four-block-mat* :: *'a mat* $\Rightarrow$ *'a mat* $\Rightarrow$ *'a mat* $\Rightarrow$ *'a mat* $\Rightarrow$ *'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* $\Longrightarrow j <$ *dim-col A* + *dim-col D* $\Longrightarrow$ *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
unfolding *four-block-mat-def Let-def* **by** *auto*

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)
unfolding *carrier-mat-def* **by** *auto*

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 **by** *auto*

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*)
by (*rule eq-matI*, *auto*)

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*)
by (*rule eq-matI*, *auto*)

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 −
assume *i*: *i < nr1*
show *?AB* **by** (*rule eq-vecI*, *insert i c*, *auto*)
next
assume *i*: *¬ i < nr1 i < nr1 + nr2*
show *?CD* **by** (*rule eq-vecI*, *insert i c*, *auto*)
qed

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** - $\Longrightarrow$ - $\Longrightarrow$ *?BD*)
proof −
  assume *j*: *j* < *nc1*
  show *?AC* **by** (*rule eq-vecI*, *insert j c*, *auto*)
next
  assume *j*: ¬ *j* < *nc1* *j* < *nc1* + *nc2*
  show *?BD* **by** (*rule eq-vecI*, *insert j c*, *auto*)
qed

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 −
  note *row* = *row-four-block-mat*[*OF c1*]
  note *col* = *col-four-block-mat*[*OF c2*]
  {
    fix *i j*
    assume *i*: *i* < *nr1* **and** *j*: *j* < *nc1*
    have *row ?M1 i* · *col ?M2 j* = *row A1 i* · *col A2 j* + *row B1 i* · *col C2 j*
      unfolding *row*(*1*)[*OF i*] *col*(*1*)[*OF j*]
      by (*rule scalar-prod-append*[*of - n1 - n2*], *insert c1 c2 i j*, *auto*)
  }
  moreover
  {
    fix *i j*
    assume *i*: ¬ *i* < *nr1* *i* < *nr1* + *nr2* **and** *j*: *j* < *nc1*
    hence *i'*: *i* − *nr1* < *nr2* **by** *auto*
    have *row ?M1 i* · *col ?M2 j* = *row C1* (*i* − *nr1*) · *col A2 j* + *row D1* (*i* − *nr1*) · *col C2 j*
      unfolding *row*(*2*)[*OF i*] *col*(*1*)[*OF j*]
      by (*rule scalar-prod-append*[*of - n1 - n2*], *insert c1 c2 i i' j*, *auto*)
  }
  moreover
  {
    fix *i j*
    assume *i*: *i* < *nr1* **and** *j*: ¬ *j* < *nc1* *j* < *nc1* + *nc2*
    hence *j'*: *j* − *nc1* < *nc2* **by** *auto*
    have *row ?M1 i* · *col ?M2 j* = *row A1 i* · *col B2* (*j* − *nc1*) + *row B1 i* · *col D2* (*j* − *nc1*)
      unfolding *row*(*1*)[*OF i*] *col*(*2*)[*OF j*]
      by (*rule scalar-prod-append*[*of - n1 - n2*], *insert c1 c2 i j' j*, *auto*)
  }
  moreover
  {

fix *i j*
assume *i*: ¬ *i* < *nr1* *i* < *nr1* + *nr2* **and** *j*: ¬ *j* < *nc1* *j* < *nc1* + *nc2*
hence *i'*: *i* − *nr1* < *nr2* **and** *j'*: *j* − *nc1* < *nc2* **by** *auto*
have *row ?M1 i* · *col ?M2 j* = *row C1* (*i* − *nr1*) · *col B2* (*j* − *nc1*) + *row D1* (*i* − *nr1*) · *col D2* (*j* − *nc1*)
unfolding *row*(*2*)[*OF i*] *col*(*2*)[*OF j*]
by (*rule scalar-prod-append*[*of - n1 - n2*], *insert c1 c2 i i' j' j*, *auto*)
}
ultimately show *?thesis*
by (*intro eq-matI*, *insert c1 c2*, *auto*)
qed

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*
unfolding *append-rows-def* **by** *auto*

lemma *col-mult2*[*simp*]:
assumes *A*: *A* : *carrier-mat nr n*
and *B*: *B* : *carrier-mat n nc*
and *j*: *j* < *nc*
shows *col* (*A* ∗ *B*) *j* = *A* $*_v$ *col B j*
proof
have *AB*: *A* ∗ *B* : *carrier-mat nr nc* **using** *A B* **by** *auto*
fix *i* **assume** *i*: *i* < *dim-vec* (*A* $*_v$ *col B j*)
show *col* (*A* ∗ *B*) *j* $ *i* = (*A* $*_v$ *col B j*) $ *i*
using *A B AB j i* **by** *simp*
qed *auto*

lemma *mat-vec-as-mat-mat-mult*: **assumes** *A*: *A* ∈ *carrier-mat nr nc*
and *v*: *v* ∈ *carrier-vec nc*
shows *A* $*_v$ *v* = *col* (*A* ∗ *mat-of-cols nc* [*v*]) *0*
by (*subst col-mult2*[*OF A*], *insert v*, *auto*)

lemma *mat-mult-append*: **assumes** *A*: *A* ∈ *carrier-mat nr1 nc*
and *B*: *B* ∈ *carrier-mat nr2 nc*
and *v*: *v* ∈ *carrier-vec nc*
shows (*A* $@_r$ *B*) $*_v$ *v* = (*A* $*_v$ *v*) $@_v$ (*B* $*_v$ *v*)
proof −
let *?Fb1* = *four-block-mat A* (0_m *nr1 0*) *B* (0_m *nr2 0*)
let *?Fb2* = *four-block-mat* (*mat-of-cols nc* [*v*]) (0_m *nc 0*) (0_m *0 1*) (0_m *0 0*)
have *id*: *?Fb2* = *mat-of-cols nc* [*v*]
using *v* **by** *auto*
have (*A* $@_r$ *B*) $*_v$ *v* = *col* (*?Fb1* ∗ *?Fb2*) *0* **unfolding** *id*
by (*subst mat-vec-as-mat-mat-mult*[*OF - v*], *insert A B*, *auto simp*: *append-rows-def*)
also have *?Fb1* ∗ *?Fb2* = *four-block-mat* (*A* ∗ *mat-of-cols nc* [*v*] + 0_m *nr1 0* ∗

0_m 0 1) ($A * 0_m$ nc 0 + 0_m $nr1$ $0 * 0_m$ 0 0)
    (B * *mat-of-cols nc* [v] + 0_m $nr2$ $0 * 0_m$ 0 1) ($B * 0_m$ nc 0 + 0_m $nr2$ 0 * 0_m 0 0)
  by (*rule mult-four-block-mat*[*OF A - B*], *auto*)
 also have (A * *mat-of-cols nc* [v] + 0_m $nr1$ $0 * 0_m$ 0 1) = A * *mat-of-cols nc* [v]
  using A v **by** *auto*
 also have (B * *mat-of-cols nc* [v] + 0_m $nr2$ $0 * 0_m$ 0 1) = B * *mat-of-cols nc* [v]
  using B v **by** *auto*
 also have ($A * 0_m$ nc 0 + 0_m $nr1$ $0 * 0_m$ 0 0) = 0_m $nr1$ 0 **using** A **by** *auto*
 also have ($B * 0_m$ nc 0 + 0_m $nr2$ $0 * 0_m$ 0 0) = 0_m $nr2$ 0 **using** B **by** *auto*
 finally have (A $@_r$ B) $*_v$ v = *col* (*four-block-mat* (A * *mat-of-cols nc* [v]) (0_m $nr1$ 0) (B * *mat-of-cols nc* [v]) (0_m $nr2$ 0)) 0 **.**
 also have ... = *col* (A * *mat-of-cols nc* [v]) 0 $@_v$ *col* (B * *mat-of-cols nc* [v]) 0
  by (*rule col-four-block-mat*, *insert A B v*, *auto*)
 also have *col* (A * *mat-of-cols nc* [v]) 0 = A $*_v$ v
  by (*rule mat-vec-as-mat-mat-mult*[*symmetric*, *OF A v*])
 also have *col* (B * *mat-of-cols nc* [v]) 0 = B $*_v$ v
  by (*rule mat-vec-as-mat-mat-mult*[*symmetric*, *OF B v*])
 finally show *?thesis* **.**
qed

lemma *append-rows-le*: **assumes** A: $A \in$ *carrier-mat nr1 nc*
 and B: $B \in$ *carrier-mat nr2 nc*
 and a: $a \in$ *carrier-vec nr1*
 and v: $v \in$ *carrier-vec nc*
shows (A $@_r$ B) $*_v$ $v \leq$ (a $@_v$ b) $\longleftrightarrow$ A $*_v$ $v \leq a \wedge B$ $*_v$ $v \leq b$
 unfolding *mat-mult-append*[*OF A B v*]
 by (*rule append-vec-le*[*OF - a*], *insert A v*, *auto*)

lemma *elements-four-block-mat*:
 assumes *c*: $A \in$ *carrier-mat nr1 nc1* $B \in$ *carrier-mat nr1 nc2*
 $C \in$ *carrier-mat nr2 nc1* $D \in$ *carrier-mat nr2 nc2*
 shows
 elements-mat (*four-block-mat A B C D*) $\subseteq$
  elements-mat A $\cup$ *elements-mat* B $\cup$ *elements-mat* C $\cup$ *elements-mat* D
  (**is** *elements-mat ?four* $\subseteq$ -)
proof *rule*
 fix a **assume** $a \in$ *elements-mat ?four*
 then obtain i j
  where *i4*: $i <$ *dim-row ?four* **and** *j4*: $j <$ *dim-col ?four* **and** a: a = *?four* $$ (i, j)
  by *auto*
 show $a \in$ *elements-mat* A $\cup$ *elements-mat* B $\cup$ *elements-mat* C $\cup$ *elements-mat* D
 proof (*cases* $i <$ *nr1*)
  case *True* **note** *i1* = *this*

show *?thesis*
proof (*cases* $j < nc1$)
case *True*
then have $a = A$ $$ (i,j) **using** *c i1 a* **by** *simp*
thus *?thesis* **using** *c i1 True* **by** *auto* **next**
case *False*
then have $a = B$ $$ $(i,j-nc1)$ **using** *c i1 a j4* **by** *simp*
moreover have $j - nc1 < nc2$ **using** *c j4 False* **by** *auto*
ultimately show *?thesis* **using** *c i1* **by** *auto*
qed next
case *False* **note** $i1 = this$
have *i2*: $i - nr1 < nr2$ **using** *c i1 i4* **by** *auto*
show *?thesis*
proof (*cases* $j < nc1$)
case *True*
then have $a = C$ $$ $(i-nr1,j)$ **using** *c i2 a i1* **by** *simp*
thus *?thesis* **using** *c i2 True* **by** *auto* **next**
case *False*
then have $a = D$ $$ $(i-nr1,j-nc1)$ **using** *c i2 a i1 j4* **by** *simp*
moreover have $j - nc1 < nc2$ **using** *c j4 False* **by** *auto*
ultimately show *?thesis* **using** *c i2* **by** *auto*
qed
qed
qed

lemma *assoc-four-block-mat*: **fixes** *FB* :: *'a mat* ⇒ *'a mat* ⇒ *'a* :: *zero mat*
defines *FB*: $FB \equiv \lambda$ *Bb Cc. four-block-mat Bb* (0_m (*dim-row Bb*) (*dim-col Cc*)) (0_m (*dim-row Cc*) (*dim-col Bb*)) *Cc*
shows *FB A* (*FB B C*) = *FB* (*FB A B*) *C* (**is** *?L* = *?R*)
proof –
let *?ar* = *dim-row A* **let** *?ac* = *dim-col A*
let *?br* = *dim-row B* **let** *?bc* = *dim-col B*
let *?cr* = *dim-row C* **let** *?cc* = *dim-col C*
let *?r* = *?ar* + *?br* + *?cr* **let** *?c* = *?ac* + *?bc* + *?cc*
let *?BC* = *FB B C* **let** *?AB* = *FB A B*
have *dL*: *dim-row ?L* = *?r dim-col ?L* = *?c* **unfolding** *FB* **by** *auto*
have *dR*: *dim-row ?R* = *?ar* + *?br* + *?cr dim-col ?R* = *?ac* + *?bc* + *?cc* **unfolding** *FB* **by** *auto*
have *dBC*: *dim-row ?BC* = *?br* + *?cr dim-col ?BC* = *?bc* + *?cc* **unfolding** *FB* **by** *auto*
have *dAB*: *dim-row ?AB* = *?ar* + *?br dim-col ?AB* = *?ac* + *?bc* **unfolding** *FB* **by** *auto*
show *?thesis*
proof (*intro eq-matI*[*of ?R ?L*, *unfolded dL dR*, *OF - refl refl*])
fix *i j*
assume *i*: *i* < *?r* **and** *j*: *j* < *?c*
show *?L* $$ (*i*,*j*) = *?R* $$ (*i*,*j*)
proof (*cases i* < *?ar*)
case *True* **note** *i* = *this*

```
      thus ?thesis using j
        by (cases j < ?ac, auto simp: FB)
    next
      case False note ii = this
      show ?thesis
      proof (cases j < ?ac)
        case True
        with i ii show ?thesis unfolding FB by auto
      next
        case False note jj = this
        from j jj i ii have L: ?L $$ (i,j) = ?BC $$ (i − ?ar, j − ?ac) unfolding
FB by auto
        have R: ?R $$ (i,j) = ?BC $$ (i − ?ar, j − ?ac) using ii jj i j
          by (cases i < ?ar + ?br; cases j < ?ac + ?bc, auto simp: FB)
        show ?thesis unfolding L R ..
      qed
    qed
  qed
qed

definition split-block :: 'a mat ⇒ nat ⇒ nat ⇒ ('a mat × 'a mat × 'a mat × 'a
mat)
  where split-block A sr sc = (let
    nr = dim-row A; nc = dim-col A;
    nr2 = nr − sr; nc2 = nc − sc;
    A1 = mat sr sc (λ ij. A $$ ij);
    A2 = mat sr nc2 (λ (i,j). A $$ (i,j+sc));
    A3 = mat nr2 sc (λ (i,j). A $$ (i+sr,j));
    A4 = mat nr2 nc2 (λ (i,j). A $$ (i+sr,j+sc))
  in (A1,A2,A3,A4))

lemma split-block: assumes res: split-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
  using res unfolding split-block-def Let-def
  by (auto simp: dims)
```

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

```
fun diag-block-mat :: 'a :: zero mat list ⇒ 'a mat where
  diag-block-mat [] = 0m 0 0
| diag-block-mat (A # As) = (let
     B = diag-block-mat As
     in four-block-mat A (0m (dim-row A) (dim-col B)) (0m (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 −
  have *?row* ∧ *?col*
    by (*induct As*, *auto simp*: *Let-def*)
  thus *?row* **and** *?col* **by** *auto*
qed

lemma *diag-block-mat-singleton*[*simp*]: *diag-block-mat* [*A*] = *A*
  by *auto*

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)
  unfolding *Let-def*
proof (*induct As*)
  case (*Cons A As*)
  show *?case*
    unfolding *append.simps*
    unfolding *diag-block-mat.simps Let-def*
    unfolding *Cons*
    by (*rule assoc-four-block-mat*)
qed *auto*

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)
  unfolding *diag-block-mat-append diag-block-mat-singleton* **by** *auto*

lemma *diag-block-mat-square*:
  Ball (*set As*) *square-mat* ⟹ *square-mat* (*diag-block-mat As*)
by (*induct As*, *auto simp*:*Let-def*)

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)))
  by (*induct As*, *auto simp*: *Let-def*)

lemma *elements-diag-block-mat*:
  elements-mat (*diag-block-mat As*) ⊆ {*0*} ∪ ⋃ (*set* (*map elements-mat As*))
proof (*induct As*)
  case *Nil* **then show** *?case* **using** *dim-diag-block-mat*[*of Nil*] **by** *auto* **next**
  case (*Cons A As*)
    let *?D* = *diag-block-mat As*
    let *?B* = 0_m (*dim-row A*) (*dim-col ?D*)
    let *?C* = 0_m (*dim-row ?D*) (*dim-col A*)
    have *A*: *A* ∈ *carrier-mat* (*dim-row A*) (*dim-col A*) **by** *auto*

```
    have B: ?B ∈ carrier-mat (dim-row A) (dim-col ?D) by auto
    have C: ?C ∈ carrier-mat (dim-row ?D) (dim-col A) by auto
    have D: ?D ∈ carrier-mat (dim-row ?D) (dim-col ?D) by auto
    have
      elements-mat (diag-block-mat (A#As)) ⊆
       elements-mat A ∪ elements-mat ?B ∪ elements-mat ?C ∪ elements-mat ?D
      unfolding diag-block-mat.simps Let-def
      using elements-four-block-mat[OF A B C D] elements-0-mat
      by auto
    also have ... ⊆ {0} ∪ elements-mat A ∪ elements-mat ?D
      using elements-0-mat by auto
    finally show ?case using Cons by auto
qed

lemma diag-block-pow-mat: assumes sq: Ball (set As) square-mat
  shows diag-block-mat As ^m n = diag-block-mat (map (λ A. A ^m n) As) (is
?As ^m - = -)
proof (induct n)
  case 0
  have ?As ^m 0 = 1m (dim-row ?As) by simp
  also have dim-row ?As = sum-list (map dim-row As)
    using diag-block-mat-square[OF sq] unfolding dim-diag-block-mat by auto
  also have 1m ... = diag-block-mat (map (λA. 1m (dim-row A)) As) by simp
  also have ... = diag-block-mat (map (λ A. A ^m 0) As) by simp
  finally show ?case .
next
  case (Suc n)
  let ?An = λ As. diag-block-mat (map (λA. A ^m n) As)
  let ?Asn = λ As. diag-block-mat (map (λA. A ^m n * A) As)
  from Suc have ?case = (?An As * diag-block-mat As = ?Asn As) by simp
  also have ... using sq
  proof (induct As)
    case (Cons A As)
    hence IH: ?An As * diag-block-mat As = ?Asn As
      and sq: Ball (set As) square-mat and A: dim-col A = dim-row A by auto
    have sq2: Ball (set (List.map (λA. A ^m n) As)) square-mat
      and sq3: Ball (set (List.map (λA. A ^m n * A) As)) square-mat
      using sq by auto
    define n1 where n1 = dim-row A
    define n2 where n2 = sum-list (map dim-row As)
    from A have A: A ∈ carrier-mat n1 n1 unfolding n1-def carrier-mat-def by
simp
    have [simp]: dim-col (?An As) = n2 dim-row (?An As) = n2
      unfolding n2-def
      using diag-block-mat-square[OF sq2,unfolded square-mat.simps]
      unfolding dim-diag-block-mat map-map by (auto simp:o-def)
    have [simp]: dim-col (?Asn As) = n2 dim-row (?Asn As) = n2
      unfolding n2-def
      using diag-block-mat-square[OF sq3,unfolded square-mat.simps]
```

unfolding *dim-diag-block-mat map-map* **by** (*auto simp*:*o-def*)
have [*simp*]:
dim-row (*diag-block-mat As*) = *n2*
dim-col (*diag-block-mat As*) = *n2*
unfolding *n2-def*
using *diag-block-mat-square*[*OF sq*,*unfolded square-mat.simps*]
unfolding *dim-diag-block-mat* **by** *auto*

have [*simp*]: *diag-block-mat As* ∈ *carrier-mat n2 n2* **unfolding** *carrier-mat-def* **by** *simp*
have [*simp*]: *?An As* ∈ *carrier-mat n2 n2* **unfolding** *carrier-mat-def* **by** *simp*
show *?case* **unfolding** *diag-block-mat.simps Let-def list.simps*
by (*subst mult-four-block-mat*[*of - n1 n1 - n2 - n2 - - n1 - n2*],
insert A, *auto simp*: *IH*)
qed *auto*
finally show *?case* **by** *simp*
qed

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*
using *assms*
proof (*induct As arbitrary*: *i j*)
case (*Cons A As i j*)
let *?n1* = *dim-row A*
let *?n2* = *sum-list* (*map dim-row As*)
from *Cons* **have** [*simp*]: *dim-col A* = *?n1* **by** *simp*
from *Cons* **have** *Ball* (*set As*) *square-mat* **by** *auto*
note [*simp*] = *diag-block-mat-square*[*OF this*,*unfolded square-mat.simps*]
note [*simp*] = *dim-diag-block-mat*(*1*)
from *Cons*(*5*) **have** *i*: *i* < *?n1* + *?n2* **by** *simp*
show *?case*
proof (*cases i* < *?n1*)
case *True*
with *Cons*(*4*) **have** *j*: *j* < *?n1* **by** *auto*
with *True Cons*(*2*)[*of A*, *OF - Cons*(*4*)] **show** *?thesis*
by (*simp add*: *Let-def*)
next
case *False* **note** *iAs* = *this*
show *?thesis*
proof (*cases j* < *?n1*)
case *True*
with *i iAs* **show** *?thesis* **by** (*simp add*: *Let-def*)
next
case *False* **note** *jAs* = *this*
from *Cons*(*4*) *i* **have** *j*: *j* < *?n1* + *?n2* **by** *auto*
show *?thesis* **using** *iAs jAs i j*

by (*simp add*: *Let-def*, *subst Cons*(*1*), *insert Cons*(*2*−*4*), *auto*)
qed
qed
qed *simp*

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*)
by (*rule eq-matI*, *insert c*, *auto*)

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*)
by (*rule eq-matI*, *insert c*, *auto*)

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*)
by (*rule eq-matI*, *insert assms*, *auto*)

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*
by (*rule nth-equalityI*, *insert c*, *auto simp*: *diag-mat-def nth-append*)

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*
unfolding *mk-diagonal-def* **by**(*induct as*, *auto simp*: *Let-def*)

lemma *mk-diagonal-diagonal*: *diagonal-mat* (*mk-diagonal as*)
unfolding *mk-diagonal-def*
proof (*induct as*)
case *Nil* **show** *?case* **unfolding** *mk-diagonal-def diagonal-mat-def* **by** *simp* **next**
case (*Cons a as*)
let *?n* = *length* (*a*#*as*)
let *?A* = *mat* (*Suc 0*) (*Suc 0*) (λ-. *a*)

let *?f* = *map* (*λa. mat* (*Suc 0*) (*Suc 0*) (*λ-. a*))
let *?AS* = *diag-block-mat* (*?f as*)
let *?AAS* = *diag-block-mat* (*?f* (*a#as*))
show *?case*
unfolding *diagonal-mat-def*
proof(*intro allI impI*)
fix *i j* **assume** *ir*: *i* < *dim-row ?AAS* **and** *jc*: *j* < *dim-col ?AAS* **and** *ij*: *i* ≠ *j*
hence *ir2*: *i* < *1* + *dim-row ?AS* **and** *jc2*: *j* < *1* + *dim-col ?AS*
unfolding *dim-row-mat list.map diag-block-mat.simps Let-def*
by *auto*
show *?AAS* $$ (*i,j*) = *0*
proof (*cases i* = *0*)
case *True*
then show *?thesis* **using** *jc ij* **by** (*auto simp*: *Let-def*) **next**
case *False* **note** *i0* = *this*
show *?thesis*
proof (*cases j* = *0*)
case *True*
then show *?thesis* **using** *ir ij* **by** (*auto simp*: *Let-def*) **next**
case *False*
have *ir3*: *i−1* < *dim-row ?AS* **and** *jc3*: *j−1* < *dim-col ?AS*
using *ir2 jc2 i0 False* **by** *auto*
have *IH*: ⋀*i j. i* < *dim-row ?AS* ⟹ *j* < *dim-col ?AS* ⟹ *i* ≠ *j* ⟹ *?AS* $$ (*i,j*) = *0*
using *Cons* **unfolding** *diagonal-mat-def* **by** *auto*
have *?AS* $$ (*i−1,j−1*) = *0*
using *IH*[*OF ir3 jc3*] *i0 False ij* **by** *auto*
thus *?thesis* **using** *ir jc ij* **by** (*simp add*: *Let-def*)
qed
qed
qed
qed

definition *orthogonal-mat* :: *'a::semiring-0 mat* ⇒ *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*)
unfolding *orthogonal-mat-def diagonal-mat-def* **by** *auto*

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
unfolding *orthogonal-mat-def diagonal-mat-def* **by** *auto*

definition *orthogonal* :: *'a::semiring-0 vec list* ⇒ *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*)
  unfolding *orthogonal-def* **by** *auto*

lemma *orthogonalI*[*intro*]:
  (⋀*i j*. *i* < *length vs* ⟹ *j* < *length vs* ⟹ (*nth vs i* · *nth vs j* = *0*) = (*i* ≠ *j*))
⟹
   orthogonal vs
  unfolding *orthogonal-def* **by** *auto*

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*)
  by (*rule eq-matI*, *insert* ∗, *auto*)

lemma *zero-transpose-mat*[*simp*]: *transpose-mat* (0_m *n m*) = (0_m *m n*)
  by (*rule eq-matI*, *auto*)

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 −
  let *?C* = *four-block-mat A B* (0_m *m n*) *D*
  from *AD* **have** *dim*: *dim-row ?C* = *n* + *m dim-col ?C* = *n* + *m dim-row A* = *n* **by** *auto*
  show *?thesis*
  proof (*rule upper-triangularI*, *unfold dim*)
    fix *i j*
    assume ∗: *j* < *i i* < *n* + *m*
    show *?C* $$ (*i*,*j*) = *0*
    proof (*cases i* < *n*)
      case *True*
      with *upper-triangularD*[*OF ut*(*1*) ∗(*1*)] ∗ *AD* **show** *?thesis* **by** *auto*
    next
      case *False* **note** *i* = *this*
       show *?thesis* **by** (*cases j* < *n*, *insert upper-triangularD*[*OF ut*(*2*)] ∗ *i AD*, *auto*)

    qed
  qed
qed

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 (*induct k*)
  case (*Suc k*)
  let *?FB* = λ *A B. four-block-mat A* (0_m *n m*) (0_m *m n*) *B*
  let *?A* = *?FB A B*
  let *?B* = *?FB* (A $\widehat{\ }_m$ k) (B $\widehat{\ }_m$ k)
  from *A B* **have** *Ak*: A $\widehat{\ }_m$ $k \in$ *carrier-mat n n* **and** *Bk*: B $\widehat{\ }_m$ $k \in$ *carrier-mat m m* **by** *auto*
  have *?A* $\widehat{\ }_m$ *Suc k* = *?A* $\widehat{\ }_m$ k $*$ *?A* **by** *simp*
  also have *?A* $\widehat{\ }_m$ k = *?B* **by** (*rule Suc*)
  also have *?B* $*$ *?A* = *?FB* (A $\widehat{\ }_m$ *Suc k*) (B $\widehat{\ }_m$ *Suc k*)
    by (*subst mult-four-block-mat*[*OF Ak - - Bk A - - B*], *insert A B*, *auto*)
  finally show *?case* **.**
qed (*insert A B*, *auto*)

lemma *uminus-scalar-prod*:
  assumes [*simp*]: *v* : *carrier-vec n w* : *carrier-vec n*
  shows $-$ ((*v*::*'a*::*field vec*) $\cdot$ *w*) = ($-$ *v*) $\cdot$ *w*
  unfolding *scalar-prod-def uminus-vec-def*
  apply (*subst sum-negf*[*symmetric*])
proof (*rule sum.cong*[*OF refl*])
  fix *i* **assume** *i*: *i* : {*0* ..<*dim-vec w*}
  have [*simp*]: *dim-vec v* = *n dim-vec w* = *n* **by** *auto*
  show $-$ (*v* \$ *i* $*$ *w* \$ *i*) = *vec* (*dim-vec v*) (λ*i*. $-$ *v* \$ *i*) \$ *i* $*$ *w* \$ *i*
    unfolding *minus-mult-left* **using** *i* **by** *auto*
qed

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* $\longleftrightarrow$ *?R*)
proof
  have [*simp*]: *dim-vec v* = *n dim-vec v'* = *n* **by** *auto*
  { assume *L*: *?L*
    have *vv'*: $v = v'$
    proof
      fix *i* **assume** *i*: $i <$ *dim-vec v'*
      have (v @$_v$ w) \$ *i* = (v' @$_v$ w') \$ *i* **using** *L* **by** *auto*
      thus *v* \$ *i* = *v'* \$ *i* **using** *i* **by** *auto*
    qed *auto*
    moreover have $w = w'$

proof
  show *dim-vec w* = *dim-vec w′* **using** *vv′ L*
    by (*metis add-diff-cancel-left′ index-append-vec*(*2*))
  moreover fix *i* **assume** *i*: *i* < *dim-vec w′*
  have (v @$_v$ w) \$ (n + i) = (v' @$_v$ w') \$ (n + i) **using** *L* **by** *auto*
  ultimately show *w* \$ *i* = *w′* \$ *i* **using** *i* **by** *simp*
qed
ultimately show *?R* **by** *simp*
}
qed *auto*

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
  have [*simp*]: *dim-vec v* = *n dim-vec v′* = *n* **by** *auto*
  have [*simp*]: *dim-vec w* = *m dim-vec w′* = *m* **by** *auto*
  fix *i* **assume** *i*: *i* < *dim-vec ?R*
  thus *?L* \$ *i* = *?R* \$ *i* **by** (*cases i* < *n*,*auto*)
qed *auto*

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
 have *ABCD*: *?ABCD* : *carrier-mat* (*nr1*+*nr2*) (*nc1*+*nc2*) **using** *four-block-carrier-mat*[*OF
A D*]**.**
  fix *i* **assume** *i*: *i* < *dim-vec ?r*
  show (*?ABCD* $*_v$ (a @$_v$ d)) \$ *i* = *?r* \$ *i* (**is** *?li* = -)
  proof (*cases i* < *nr1*)
   case *True*
    have *?li* = (*row A i* @$_v$ *row B i*) · (a @$_v$ d)
     using *A row-four-block-mat*[*OF A B C D*] *True* **by** *simp*
    also have ... = *row A i* · *a* + *row B i* · *d*
     apply (*rule scalar-prod-append*) **using** *A B D a d True* **by** *auto*
    finally show *?thesis* **using** *A B True* **by** *auto*
   next case *False*
    let *?i* = *i* − *nr1*
    have *?li* = (*row C ?i* @$_v$ *row D ?i*) · (a @$_v$ d)
     using *i row-four-block-mat*[*OF A B C D*] *False A B C D* **by** *simp*
    also have ... = *row C ?i* · *a* + *row D ?i* · *d*

apply (*rule scalar-prod-append*) **using** $A\ B\ C\ D\ a\ d\ False$ **by** *auto*
finally show *?thesis* **using** $A\ B\ C\ D\ False\ i$ **by** *auto*
qed
qed (*insert* $A\ B$, *auto*)

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 $(0_m\ n\ m)$ $(0_m\ m\ n)$ $D *_v (a\ @_v\ d) = A *_v a\ @_v\ D *_v d$
by (*subst four-block-mat-mult-vec*[*OF A - - D a d*], *insert* $A\ D\ a\ d$, *auto*)

lemma *similar-mat-witI*: **assumes** $P * Q = 1_m\ n$ $Q * P = 1_m\ n$ $A = P * B * Q$
$A \in$ *carrier-mat* $n\ n$ $B \in$ *carrier-mat* $n\ n$ $P \in$ *carrier-mat* $n\ n$ $Q \in$ *carrier-mat* $n\ n$
shows *similar-mat-wit* $A\ B\ P\ Q$ **using** *assms* **unfolding** *similar-mat-wit-def Let-def* **by** *auto*

lemma *similar-mat-witD*: **assumes** n = *dim-row* A *similar-mat-wit* $A\ B\ P\ Q$
shows $P * Q = 1_m\ n$ $Q * P = 1_m\ n$ $A = P * B * Q$
$A \in$ *carrier-mat* $n\ n$ $B \in$ *carrier-mat* $n\ n$ $P \in$ *carrier-mat* $n\ n$ $Q \in$ *carrier-mat* $n\ n$
using *assms*(*2*) **unfolding** *similar-mat-wit-def Let-def assms*(*1*)[*symmetric*] **by** *auto*

lemma *similar-mat-witD2*: **assumes** $A \in$ *carrier-mat* $n\ m$ *similar-mat-wit* $A\ B\ P\ Q$
shows $P * Q = 1_m\ n$ $Q * P = 1_m\ n$ $A = P * B * Q$
$A \in$ *carrier-mat* $n\ n$ $B \in$ *carrier-mat* $n\ n$ $P \in$ *carrier-mat* $n\ n$ $Q \in$ *carrier-mat* $n\ n$
using *similar-mat-witD*[*OF - assms*(*2*), *of n*] *assms*(*1*)[*unfolded carrier-mat-def*] **by** *auto*

lemma *similar-mat-wit-sym*: **assumes** *sim*: *similar-mat-wit* $A\ B\ P\ Q$
shows *similar-mat-wit* $B\ A\ Q\ P$
proof −
from *similar-mat-witD*[*OF refl sim*] **obtain** n **where**
AB: $\{A, B, P, Q\} \subseteq$ *carrier-mat* $n\ n$ $P * Q = 1_m\ n$ $Q * P = 1_m\ n$ **and** A: $A = P * B * Q$ **by** *blast*
hence $*$: $\{B, A, Q, P\} \subseteq$ *carrier-mat* $n\ n$ $Q * P = 1_m\ n$ $P * Q = 1_m\ n$ **by** *auto*
let $?c = \lambda\ A.\ A \in$ *carrier-mat* $n\ n$
from $*$ **have** *Carr*: $?c\ B\ ?c\ P\ ?c\ Q$ **by** *auto*
note [*simp*] = *assoc-mult-mat*[*of - n n - n - n*]
show *?thesis*
proof (*rule similar-mat-witI*[*of - - n*])

have $Q * A * P = (Q * P) * B * (Q * P)$
  using *Carr* **unfolding** A **by** *simp*
also have ... $= B$ **using** *Carr* **unfolding** AB **by** *simp*
finally show $B = Q * A * P$ **by** *simp*
qed (*insert* $*$ AB, *auto*)
qed

lemma *similar-mat-wit-refl*: **assumes** A: $A \in$ *carrier-mat n n*
shows *similar-mat-wit A A* (1_m n) (1_m n)
by (*rule similar-mat-witI*[*OF - - - A*], *insert A*, *auto*)

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 −
from *similar-mat-witD*[*OF refl AB*] **obtain** n **where**
AB: $\{A, B, P, Q\} \subseteq$ *carrier-mat n n* $P * Q = 1_m$ n $Q * P = 1_m$ n $A = P * B * Q$ **by** *blast*
hence B: $B \in$ *carrier-mat n n* **by** *auto*
from *similar-mat-witD2*[*OF B BC*] **have**
BC: $\{C, P', Q'\} \subseteq$ *carrier-mat n n* $P' * Q' = 1_m$ n $Q' * P' = 1_m$ n $B = P' * C * Q'$ **by** *auto*
let $?c = \lambda$ A. $A \in$ *carrier-mat n n*
let $?P = P * P'$
let $?Q = Q' * Q$
from AB BC **have** *carr*: $?c$ A $?c$ B $?c$ C $?c$ P $?c$ P' $?c$ Q $?c$ Q'
and *Carr*: $\{A, C, ?P, ?Q\} \subseteq$ *carrier-mat n n* **by** *auto*
note [*simp*] = *assoc-mult-mat*[*of - n n - n - n*]
have *id*: $A = ?P * C * ?Q$ **unfolding** $AB(4)$[*unfolded* $BC(4)$] **using** *carr*
by *simp*
have $?P * ?Q = P * (P' * Q') * Q$ **using** *carr* **by** *simp*
also have ... $= 1_m$ n **unfolding** BC **using** *carr AB* **by** *simp*
finally have PQ: $?P * ?Q = 1_m$ n **.**
have $?Q * ?P = Q' * (Q * P) * P'$ **using** *carr* **by** *simp*
also have ... $= 1_m$ n **unfolding** AB **using** *carr BC* **by** *simp*
finally have QP: $?Q * ?P = 1_m$ n **.**
show *?thesis*
by (*rule similar-mat-witI*[*OF PQ QP id*], *insert Carr*, *auto*)
qed

lemma *similar-mat-refl*: $A \in$ *carrier-mat n n* $\Longrightarrow$ *similar-mat A A*
using *similar-mat-wit-refl* **unfolding** *similar-mat-def* **by** *blast*

lemma *similar-mat-trans*: *similar-mat A B* $\Longrightarrow$ *similar-mat B C* $\Longrightarrow$ *similar-mat A C*
using *similar-mat-wit-trans* **unfolding** *similar-mat-def* **by** *blast*

lemma *similar-mat-sym*: *similar-mat A B* $\Longrightarrow$ *similar-mat B A*
using *similar-mat-wit-sym* **unfolding** *similar-mat-def* **by** *blast*

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*: $A1 \in$ *carrier-mat n n*
and *A2*: $A2 \in$ *carrier-mat m m*
and *LL*: $LL \in$ *carrier-mat m n*
and *UR*: $UR \in$ *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 −
let $?n = n + m$
let $?O1 = 1_m\ n$ **let** $?O2 = 1_m\ m$ **let** $?O = 1_m\ ?n$
from *similar-mat-witD2*[*OF A1 1*] **have** *11*: $P1 * Q1 = ?O1$ $Q1 * P1 = ?O1$
and *P1*: $P1 \in$ *carrier-mat n n* **and** *Q1*: $Q1 \in$ *carrier-mat n n*
and *B1*: $B1 \in$ *carrier-mat n n* **and** *1*: $A1 = P1 * B1 * Q1$ **by** *auto*
from *similar-mat-witD2*[*OF A2 2*] **have** *21*: $P2 * Q2 = ?O2$ $Q2 * P2 = ?O2$
and *P2*: $P2 \in$ *carrier-mat m m* **and** *Q2*: $Q2 \in$ *carrier-mat m m*
and *B2*: $B2 \in$ *carrier-mat m m* **and** *2*: $A2 = P2 * B2 * Q2$ **by** *auto*
have *PQ1*: $?P * ?Q = ?O$
by (*subst mult-four-block-mat*[*OF P1 - - P2 Q1 - - Q2*], *unfold 11 21*, *insert P1 P2 Q1 Q2*,
auto intro!: *eq-matI*)
have *QP1*: $?Q * ?P = ?O$
by (*subst mult-four-block-mat*[*OF Q1 - - Q2 P1 - - P2*], *unfold 11 21*, *insert P1 P2 Q1 Q2*,
auto intro!: *eq-matI*)
let $?PB = ?P * ?B$
have *P*: $?P \in$ *carrier-mat ?n ?n* **using** *P1 P2* **by** *auto*
have *Q*: $?Q \in$ *carrier-mat ?n ?n* **using** *Q1 Q2* **by** *auto*
have *B*: $?B \in$ *carrier-mat ?n ?n* **using** *B1 UR LL B2* **by** *auto*
have *PB*: $?PB \in$ *carrier-mat ?n ?n* **using** *P B* **by** *auto*
have *PB1*: $P1 * B1 \in$ *carrier-mat n n* **using** *P1 B1* **by** *auto*
have *PB2*: $P2 * B2 \in$ *carrier-mat m m* **using** *P2 B2* **by** *auto*
have *P1UR*: $P1 * UR \in$ *carrier-mat n m* **using** *P1 UR* **by** *auto*
have *P2LL*: $P2 * LL \in$ *carrier-mat m n* **using** *P2 LL* **by** *auto*
have *id*: $?PB =$ *four-block-mat* $(P1 * B1)$ $(P1 * UR)$ $(P2 * LL)$ $(P2 * B2)$
by (*subst mult-four-block-mat*[*OF P1 - - P2 B1 UR LL B2*], *insert P1 P2 B1 B2 LL UR*, *auto*)
have *id*: $?PB * ?Q =$ *four-block-mat* $(P1 * B1 * Q1)$ $(P1 * UR * Q2)$
$(P2 * LL * Q1)$ $(P2 * B2 * Q2)$ **unfolding** *id*
by (*subst mult-four-block-mat*[*OF PB1 P1UR P2LL PB2 Q1 - - Q2*],
insert P1 P2 B1 B2 Q1 Q2 UR LL, *auto*)
have *id*: $?A = ?P * ?B * ?Q$ **unfolding** *id 1 2 URA LLA* **..**

show *?thesis*
by (*rule similar-mat-witI*[*OF PQ1 QP1 id*], *insert A1 A2 B1 B2 Q1 Q2 P1 P2*, *auto*)
qed

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 −
from *1*[*unfolded similar-mat-def*] **obtain** *P1 Q1* **where** *1*: *similar-mat-wit A1 B1 P1 Q1* **by** *auto*
note *w1* = *similar-mat-witD2*[*OF A1 1*]
from *2*[*unfolded similar-mat-def*] **obtain** *P2 Q2* **where** *2*: *similar-mat-wit A2 B2 P2 Q2* **by** *auto*
note *w2* = *similar-mat-witD2*[*OF A2 2*]
from *w1 w2* **have** *C*: *B1* ∈ *carrier-mat n n B2* ∈ *carrier-mat m m* **by** *auto*
from *w1 w2* **have** *id*: 0_m *m n* = *Q2* ∗ 0_m *m n* ∗ *P1* **by** *simp*
let *?wit* = *Q1* ∗ *A0* ∗ *P2*
from *w1 w2 A0* **have** *wit*: *?wit* ∈ *carrier-mat n m* **by** *auto*
from *similar-mat-wit-sym*[*OF similar-mat-wit-four-block*[*OF similar-mat-wit-sym*[*OF 1*] *similar-mat-wit-sym*[*OF 2*]
refl id C zero-carrier-mat A0]]
have *similar-mat* (*four-block-mat A1 A0* (0_m *m n*) *A2*) (*four-block-mat B1* (*Q1* ∗ *A0* ∗ *P2*) (0_m *m n*) *B2*)
unfolding *similar-mat-def* **by** *auto*
thus *?thesis* **using** *wit* **by** *auto*
qed

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 −
from *1*[*unfolded similar-mat-def*] **obtain** *P1 Q1* **where** *1*: *similar-mat-wit A1 B1 P1 Q1* **by** *auto*
note *w1* = *similar-mat-witD2*[*OF A1 1*]
from *2*[*unfolded similar-mat-def*] **obtain** *P2 Q2* **where** *2*: *similar-mat-wit A2 B2 P2 Q2* **by** *auto*
note *w2* = *similar-mat-witD2*[*OF A2 2*]

from *w1 w2* have *C*: *B1* ∈ *carrier-mat n n B2* ∈ *carrier-mat m m* by *auto*
from *w1 w2* have *id*: 0_m *m n* = *Q2* ∗ 0_m *m n* ∗ *P1* by *simp*
from *w1 w2* have *id2*: 0_m *n m* = *Q1* ∗ 0_m *n m* ∗ *P2* by *simp*
from *similar-mat-wit-sym*[*OF similar-mat-wit-four-block*[*OF similar-mat-wit-sym*[*OF 1*] *similar-mat-wit-sym*[*OF 2*]
id2 id C zero-carrier-mat zero-carrier-mat]]
show *?thesis* unfolding *similar-mat-def* by *blast*
qed

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*))
using *assms*
proof (*induct Ms*)
case *Nil*
show *?case* **by** (*auto intro*!: *similar-mat-refl*[*of - 0*])
next
case (*Cons AB Ms*)
obtain *A B* **where** *AB*: *AB* = (*A*,*B*) **by** *force*
from *Cons*(*2*)[*of A B*] **have** *simAB*: *similar-mat A B* **unfolding** *AB* **by** *auto*
from *similar-matD*[*OF this*] **obtain** *n* **where** *A*: *A* ∈ *carrier-mat n n* **and** *B*: *B* ∈ *carrier-mat n n* **by** *auto*
hence [*simp*]: *dim-row A* = *n dim-col A* = *n dim-row B* = *n dim-col B* = *n* **by** *auto*
let *?C* = *diag-block-mat* (*map fst Ms*) **let** *?D* = *diag-block-mat* (*map snd Ms*)
from *Cons*(*1*)[*OF Cons*(*2*)] **have** *simRec*: *similar-mat ?C ?D* **by** *auto*
from *similar-matD*[*OF this*] **obtain** *m* **where** *C*: *?C* ∈ *carrier-mat m m* **and** *D*: *?D* ∈ *carrier-mat m m* **by** *auto*
hence [*simp*]: *dim-row ?C* = *m dim-col ?C* = *m dim-row ?D* = *m dim-col ?D* = *m* **by** *auto*
have *similar-mat* (*diag-block-mat* (*map fst* (*AB* # *Ms*))) (*diag-block-mat* (*map snd* (*AB* # *Ms*)))
= *similar-mat* (*four-block-mat A* (0_m *n m*) (0_m *m n*) *?C*) (*four-block-mat B* (0_m *n m*) (0_m *m n*) *?D*)
unfolding *AB* **by** (*simp add*: *Let-def*)
also have ...
by (*rule similar-mat-four-block-0-0*[*OF simAB simRec A C*])
finally show *?case* **.**
qed

lemma *similar-mat-wit-pow*: **assumes** *wit*: *similar-mat-wit A B P Q*
shows *similar-mat-wit* (*A* $\widehat{\ }_m$ *k*) (*B* $\widehat{\ }_m$ *k*) *P Q*
proof −
define *n* **where** *n* = *dim-row A*
let *?C* = *carrier-mat n n*
from *similar-mat-witD*[*OF refl wit*, *folded n-def*] **have**
A: *A* ∈ *?C* **and** *B*: *B* ∈ *?C* **and** *P*: *P* ∈ *?C* **and** *Q*: *Q* ∈ *?C*
and *PQ*: *P* ∗ *Q* = 1_m *n* **and** *QP*: *Q* ∗ *P* = 1_m *n*
and *AB*: *A* = *P* ∗ *B* ∗ *Q*

by *auto*
from *A B* **have** ∗: $(A \,\widehat{}_m\, k) \in$ *carrier-mat n n B* $\widehat{}_m$ *k* ∈ *carrier-mat n n* **by** *auto*
note *carr* = *A B P Q*
have *id*: $A \,\widehat{}_m\, k = P * B \,\widehat{}_m\, k * Q$ **unfolding** *AB*
proof (*induct k*)
case *0*
thus *?case* **using** *carr* **by** (*simp add*: *PQ*)
next
case (*Suc k*)
define *Bk* **where** $Bk = B \,\widehat{}_m\, k$
have *Bk*: *Bk* ∈ *carrier-mat n n* **unfolding** *Bk-def* **using** *carr* **by** *simp*
have $(P * B * Q) \,\widehat{}_m\, Suc\ k = (P * Bk * Q) * (P * B * Q)$ **by** (*simp add*: *Suc Bk-def*)
also have $\ldots = P * (Bk * (Q * P) * B) * Q$
using *carr Bk* **by** (*simp add*: *assoc-mult-mat*[*of - n n - n - n*])
also have $Bk * (Q * P) = Bk$ **unfolding** *QP* **using** *Bk* **by** *simp*
finally show *?case* **unfolding** *Bk-def* **by** *simp*
qed
show *?thesis*
by (*rule similar-mat-witI*[*OF PQ QP id* ∗ *P Q*])
qed

lemma *similar-mat-wit-pow-id*: *similar-mat-wit A B P Q* $\Longrightarrow A \,\widehat{}_m\, k = P * B \,\widehat{}_m\, k * Q$
using *similar-mat-wit-pow*[*of A B P Q k*] **unfolding** *similar-mat-wit-def Let-def* **by** *blast*

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*: $vec_h\ (0_v\ n) = 0_v\ n$
by (*rule eq-vecI*, *auto*)

lemma *mat-hom-one*: $mat_h\ (1_m\ n) = 1_m\ n$
by (*rule eq-matI*, *auto*)

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 −
let $?L = mat_h\ (A * B)$

let *?R* = mat_h *A* ∗ mat_h *B*
let *?A* = mat_h *A*
let *?B* = mat_h *B*
from *A B* **have** *id*:
dim-row ?L = *nr dim-row ?R* = *nr*
dim-col ?L = *nc dim-col ?R* = *nc* **by** *auto*
show *?thesis*
proof (*rule eq-matI*, *unfold id*)
fix *i j*
assume ∗: *i* < *nr j* < *nc*
define *I* **where** *I* = {*0* ..< *n*}
have *id*: {*0* ..< *dim-vec* (*col ?B j*)} = *I* {*0* ..< *dim-vec* (*col B j*)} = *I*
unfolding *I-def* **using** ∗ *B* **by** *auto*
have *finite*: *finite I* **unfolding** *I-def* **by** *auto*
have *I*: *I* ⊆ {*0* ..< *n*} **unfolding** *I-def* **by** *auto*
have *?L* $$ (*i*,*j*) = *hom* (*row A i* · *col B j*) **using** *A B* ∗ **by** *auto*
also have . . . = *row ?A i* · *col ?B j* **unfolding** *scalar-prod-def id* **using** *finite I*
proof (*induct I*)
case (*insert k I*)
show *?case* **unfolding** *sum.insert*[*OF insert*(*1*−*2*)] *hom-add hom-mult*
using *insert*(*3*−) ∗ *A B* **by** *auto*
qed *simp*
also have . . . = *?R* $$ (*i*,*j*) **using** *A B* ∗ **by** *auto*
finally
show *?L* $$ (*i*, *j*) = *?R* $$ (*i*, *j*) **.**
qed *auto*
qed

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 −
let *?L* = vec_h (*A* $*_v$ *v*)
let *?R* = mat_h *A* $*_v$ vec_h *v*
let *?A* = mat_h *A*
let *?v* = vec_h *v*
from *A v* **have** *id*:
dim-vec ?L = *nr dim-vec ?R* = *nr*
by *auto*
show *?thesis*
proof (*rule eq-vecI*, *unfold id*)
fix *i*
assume ∗: *i* < *nr*
define *I* **where** *I* = {*0* ..< *n*}
have *id*: {*0* ..< *dim-vec v*} = *I* {*0* ..< *dim-vec* (vec_h *v*)} = *I*
unfolding *I-def* **using** ∗ *v* **by** *auto*
have *finite*: *finite I* **unfolding** *I-def* **by** *auto*
have *I*: *I* ⊆ {*0* ..< *n*} **unfolding** *I-def* **by** *auto*

have *?L* $ *i* = *hom* (*row A i* · *v*) **using** *A v* ∗ **by** *auto*
also have . . . = *row ?A i* · *?v* **unfolding** *scalar-prod-def id* **using** *finite I*
proof (*induct I*)
case (*insert k I*)
show *?case* **unfolding** *sum.insert*[*OF insert*(*1*−*2*)] *hom-add hom-mult*
using *insert*(*3*−) ∗ *A v* **by** *auto*
qed *simp*
also have . . . = *?R* $ *i* **using** *A v* ∗ **by** *auto*
finally
show *?L* $ *i* = *?R* $ *i* .
qed *auto*
qed
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
assume *?r*
show *?l*
by (*rule eq-vecI*, *insert* ‹*?r*›, *auto*)
qed *simp*

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
assume *?r*
show *?l*
by (*rule eq-matI*, *insert* ‹*?r*›, *auto*)
qed *simp*

lemma (**in** *inj-semiring-hom*) *vec-hom-zero-iff*[*simp*]: (vec_h *x* = 0_v *n*) = (*x* = 0_v *n*)
proof −
{
fix *i*
assume *i*: *i* < *n dim-vec x* = *n*
hence vec_h *x* $ *i* = *0* ⟷ *x* $ *i* = *0*
using *index-map-vec*(*1*)[*of i x*] **by** *simp*
} **note** *main* = *this*
show *?thesis* **unfolding** *vec-eq-iff* **by** (*simp*, *insert main*, *auto*)
qed

lemma (**in** *inj-semiring-hom*) *mat-hom-inj*: mat_h *A* = mat_h *B* ⟹ *A* = *B*
unfolding *mat-eq-iff* **by** *auto*

lemma (**in** *inj-semiring-hom*) *vec-hom-inj*: vec_h *v* = vec_h *w* ⟹ *v* = *w*
unfolding *vec-eq-iff* **by** *auto*

lemma (**in** *semiring-hom*) *mat-hom-pow*: **assumes** *A*: *A* ∈ *carrier-mat n n*

shows mat_h $(A \mathbin{\widehat{}_m} k) = (mat_h\ A) \mathbin{\widehat{}_m} k$
proof (*induct k*)
case (*Suc k*)
thus *?case* **using** *mat-hom-mult*[*OF pow-carrier-mat*[*OF A*, *of k*] *A*] **by** *simp*
qed (*simp add*: *mat-hom-one*)

lemma (**in** *semiring-hom*) *hom-sum-mat*: *hom* (*sum-mat A*) = *sum-mat* (mat_h *A*)
proof −
obtain *B* **where** *id*: *?thesis* = (*hom* (*sum* (($$) *A*) *B*) = *sum* (($$) (mat_h *A*)) *B*)
and *B*: $B \subseteq \{0..<dim\text{-}row\ A\} \times \{0..<dim\text{-}col\ A\}$
unfolding *sum-mat-def* **by** *auto*
from *B* **have** *finite B*
using *finite-subset* **by** *blast*
thus *?thesis* **unfolding** *id* **using** *B*
proof (*induct B*)
case (*insert x F*)
show *?case* **unfolding** *sum.insert*[*OF insert*(*1*−*2*)] *hom-add*
using *insert*(*3*−) **by** *auto*
qed *simp*
qed

lemma (**in** *semiring-hom*) *vec-hom-smult*: vec_h $(ev \cdot_v v)$ = *hom ev* $\cdot_v$ vec_h *v*
by (*rule eq-vecI*, *auto simp*: *hom-distribs*)

lemma *minus-scalar-prod-distrib*: **fixes** v_1 :: *'a* :: *ring vec*
assumes *v*: $v_1 \in carrier\text{-}vec\ n$ $v_2 \in carrier\text{-}vec\ n$ $v_3 \in carrier\text{-}vec\ n$
shows $(v_1 - v_2) \cdot v_3 = v_1 \cdot v_3 - v_2 \cdot v_3$
unfolding *minus-add-uminus-vec*[*OF v*(*1*−*2*)]
by (*subst add-scalar-prod-distrib*[*OF v*(*1*)], *insert v*, *auto*)

lemma *scalar-prod-minus-distrib*: **fixes** v_1 :: *'a* :: *ring vec*
assumes *v*: $v_1 \in carrier\text{-}vec\ n$ $v_2 \in carrier\text{-}vec\ n$ $v_3 \in carrier\text{-}vec\ n$
shows $v_1 \cdot (v_2 - v_3) = v_1 \cdot v_2 - v_1 \cdot v_3$
unfolding *minus-add-uminus-vec*[*OF v*(*2*−*3*)]
by (*subst scalar-prod-add-distrib*[*OF v*(*1*)], *insert v*, *auto*)

lemma *uminus-add-minus-vec*:
assumes $l \in carrier\text{-}vec\ n$ $r \in carrier\text{-}vec\ n$
shows − ((*l*::*'a* :: *ab-group-add vec*) + *r*) = (− *l* − *r*)
using *assms* **by** *auto*

lemma *minus-add-minus-vec*: **fixes** *u* :: *'a* :: *ab-group-add vec*
assumes $u \in carrier\text{-}vec\ n$ $v \in carrier\text{-}vec\ n$ $w \in carrier\text{-}vec\ n$
shows $u - (v + w) = u - v - w$
using *assms* **by** *auto*

lemma *uminus-add-minus-mat*:
assumes $l \in carrier\text{-}mat\ nr\ nc$ $r \in carrier\text{-}mat\ nr\ nc$

shows $- ((l::'a :: ab\text{-}group\text{-}add\ mat) + r) = (- l - r)$
using *assms* **by** *auto*

lemma *minus-add-minus-mat*: **fixes** $u :: 'a :: ab\text{-}group\text{-}add\ mat$
assumes $u \in carrier\text{-}mat\ nr\ nc\ v \in carrier\text{-}mat\ nr\ nc\ w \in carrier\text{-}mat\ nr\ nc$
shows $u - (v + w) = u - v - w$
using *assms* **by** *auto*

lemma *uminus-uminus-vec*[*simp*]: $- (- (v::'a:: group\text{-}add\ vec)) = v$
by *auto*

lemma *uminus-eq-vec*[*simp*]: $- (v::'a:: group\text{-}add\ vec) = - w \longleftrightarrow v = w$
by (*metis uminus-uminus-vec*)

lemma *uminus-uminus-mat*[*simp*]: $- (- (A::'a:: group\text{-}add\ mat)) = A$
by *auto*

lemma *uminus-eq-mat*[*simp*]: $- (A::'a:: group\text{-}add\ mat) = - B \longleftrightarrow A = B$
by (*metis uminus-uminus-mat*)

lemma *smult-zero-mat*[*simp*]: $(k :: 'a :: mult\text{-}zero) \cdot_m 0_m\ nr\ nc = 0_m\ nr\ nc$
by (*intro eq-matI*, *auto*)

lemma *similar-mat-wit-smult*: **fixes** $A :: 'a :: comm\text{-}ring\text{-}1\ mat$
assumes *similar-mat-wit A B P Q*
shows *similar-mat-wit* $(k \cdot_m A)$ $(k \cdot_m B)$ *P Q*
proof −
define *n* **where** $n = dim\text{-}row\ A$
note *main* = *similar-mat-witD*[*OF n-def assms*]
show *?thesis*
by (*rule similar-mat-witI*[*OF main*(*1*−*2*) - - - *main*(*6*−*7*)], *insert main*(*3*−),
auto
simp: *mult-smult-distrib mult-smult-assoc-mat*[*of - n n - n*])
qed

lemma *similar-mat-smult*: **fixes** $A :: 'a :: comm\text{-}ring\text{-}1\ mat$
assumes *similar-mat A B*
shows *similar-mat* $(k \cdot_m A)$ $(k \cdot_m B)$
using *similar-mat-wit-smult assms* **unfolding** *similar-mat-def* **by** *blast*

definition $mat\text{-}diag :: nat \Rightarrow (nat \Rightarrow 'a :: zero) \Rightarrow 'a\ mat$ **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* $\in$ *carrier-mat n n*
unfolding *mat-diag-def* **by** *auto*

lemma *mat-diag-mult-left*: **assumes** *A*: $A \in carrier\text{-}mat\ n\ nr$
shows *mat-diag n f* $*$ *A* = *Matrix.mat n nr* (λ (*i*,*j*). *f i* $*$ *A* $$ (*i*,*j*))
proof (*rule eq-matI*, *insert A*, *auto simp*: *mat-diag-def scalar-prod-def*, *goal-cases*)

case (*1 i j*)
thus *?case* **by** (*subst sum.remove*[*of - i*], *auto*)
qed

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 (*rule eq-matI*, *insert A*, *auto simp*: *mat-diag-def scalar-prod-def*, *goal-cases*)
case (*1 i j*)
thus *?case* **by** (*subst sum.remove*[*of - j*], *auto*)
qed

lemma *mat-diag-diag*[*simp*]: *mat-diag n f* ∗ *mat-diag n g* = *mat-diag n* (λ *i*. *f i* ∗ *g i*)
by (*subst mat-diag-mult-left*[*of - n n*], *auto simp*: *mat-diag-def*)

lemma *mat-diag-one*[*simp*]: *mat-diag n* (λ *x*. *1*) = 1_m *n* **unfolding** *mat-diag-def* **by** *auto*

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*
unfolding *mat-of-row-def* **by** *auto*

lemma *mat-of-row-dim*[*simp*]: *dim-row* (*mat-of-row y*) = *1*
dim-col (*mat-of-row y*) = *dim-vec y*
unfolding *mat-of-row-def* **by** *auto*

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

unfolding *mat-of-row-def* **by** *auto*

lemma *row-mat-of-row*[*simp*]: *row* (*mat-of-row y*) *0* = *y*
by *auto*

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* $@_v$ *y2*) ∗ (*A1* $@_r$ *A2*) =
mat-of-row y1 ∗ *A1* + *mat-of-row y2* ∗ *A2*
proof −
from *A1 A2* **have** *dim*: *dim-row A1* = *nr1 dim-row A2* = *nr2* **by** *auto*
let *?M1* = *mat-of-row y1*
have *M1*: *?M1* ∈ *carrier-mat 1 nr1* **using** *y1* **by** *auto*
let *?M2* = *mat-of-row y2*
have *M2*: *?M2* ∈ *carrier-mat 1 nr2* **using** *y2* **by** *auto*

let *?M3* = 0_m *0 nr1*
let *?M4* = 0_m *0 nr2*
note *z* = *zero-carrier-mat*
have *id*: *mat-of-row* (*y1* $@_v$ *y2*) = *four-block-mat*
?M1 ?M2 ?M3 ?M4 **using** *y1 y2*
by (*intro eq-matI*, *auto simp*: *mat-of-rows-def*)
show *?thesis*
unfolding *id append-rows-def dim*
by (*subst mult-four-block-mat*[*OF M1 M2 z z A1 z A2 z*], *insert A1 A2*, *auto*)
qed

lemma *mat-of-row-uminus*: *mat-of-row* (− *v*) = − *mat-of-row v*
by *auto*

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*)) (λ*i*. *case i of 0* ⇒ *a* | *Suc i* ⇒ *v* $ *i*)

lemma *vec-index-vCons-0* [*simp*]: *vCons a v* $ *0* = *a*
by (*simp add*: *vCons-def*)

lemma *vec-index-vCons-Suc* [*simp*]:
fixes *v* :: *'a vec*
shows *vCons a v* $ *Suc n* = *v* $ *n*
proof−
have *1*: *vec* (*Suc d*) *f* $ *Suc n* = *vec d* (*f* ∘ *Suc*) $ *n* **for** *d* **and** *f* :: *nat* ⇒ *'a*
by (*transfer*, *auto simp*: *mk-vec-def*)
show *?thesis*
apply (*auto simp*: *1 vCons-def o-def*) **apply** *transfer* **apply** (*auto simp*: *mk-vec-def*)
done
qed

lemma *vec-index-vCons*: *vCons a v* $ *n* = (*if n* = *0 then a else v* $ (*n* − *1*))
by (*cases n*, *auto*)

lemma *dim-vec-vCons* [*simp*]: *dim-vec* (*vCons a v*) = *Suc* (*dim-vec v*)
by (*simp add*: *vCons-def*)

lemma *vCons-carrier-vec*[*simp*]: *vCons a v* ∈ *carrier-vec* (*Suc n*) ⟷ *v* ∈ *carrier-vec n*
by (*auto dest*!: *carrier-vecD intro*: *carrier-vecI*)

lemma *vec-Suc*: *vec* (*Suc n*) *f* = *vCons* (*f 0*) (*vec n* (*f* ∘ *Suc*)) (**is** *?l* = *?r*)
proof (*unfold vec-eq-iff*, *intro conjI allI impI*)
fix *i* **assume** *i* < *dim-vec ?r*
then show *?l* $ *i* = *?r* $ *i* **by** (*cases i*, *auto*)
qed *simp*

```
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 (cases dim-vec v)
  case 0 then show thesis by (intro assms(1), auto)
next
  case (Suc n)
  show thesis
  proof (rule assms(2))
    show v: v = vCons (v $ 0) (vec n (λi. v $ Suc i)) (is v = ?r)
    proof (rule eq-vecI, unfold dim-vec-vCons dim-vec Suc)
      fix i
      assume i < Suc n
      then show v $ i = ?r $ i by (cases i, auto simp: vCons-def)
    qed simp
  qed
qed

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 (induct dim-vec v arbitrary:v)
  case 0 then show ?case by (cases v, auto intro: assms(1))
next
  case (Suc n) then show ?case by (cases v, auto intro: assms(2))
qed

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 (insert v, induct n arbitrary: v)
  case 0 then have v = vec 0 ((!) []) by auto
  with 1 show ?case by auto
next
  case (Suc n) then show ?case by (cases v, auto dest!: carrier-vecD intro:2)
qed

lemma vec-of-list-Cons[simp]: vec-of-list (a#as) = vCons a (vec-of-list as)
  by (unfold vCons-def, transfer, auto simp:mk-vec-def split:nat.split)

lemma vec-of-list-Nil[simp]: vec-of-list [] = vNil
  by (transfer', auto)

lemma scalar-prod-vCons[simp]:
```

vCons a v · *vCons b w* = *a* ∗ *b* + *v* · *w*
apply (*unfold scalar-prod-def atLeast0-lessThan-Suc-eq-insert-0 dim-vec-vCons*)
apply (*subst sum.insert*) **apply** (*simp*,*simp*)
apply (*subst sum.reindex*) **apply** *force*
apply *simp*
done

lemma *zero-vec-Suc*: 0_v (*Suc n*) = *vCons 0* (0_v *n*)
by (*auto simp*: *zero-vec-def vec-Suc o-def*)

lemma *zero-vec-zero*[*simp*]: 0_v *0* = *vNil* **by** *auto*

lemma *vCons-eq-vCons*[*simp*]: *vCons a v* = *vCons b w* ⟷ *a* = *b* ∧ *v* = *w* (**is** *?l* ⟷ *?r*)
proof
assume *?l*
note *arg-cong*[*OF this*]
from *this*[*of dim-vec*] *this*[*of* λ*x*. *x*$*0*] *this*[*of* λ*x*. *x*$*Suc* -]
show *?r* **by** (*auto simp*: *vec-eq-iff*)
qed *simp*

lemma *vec-carrier-vec*[*simp*]: *vec n f* ∈ *carrier-vec m* ⟷ *n* = *m*
unfolding *carrier-vec-def* **by** *auto*

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

lemma *map-mat-transpose*: (*map-mat f A*)^T = *map-mat f A*^{T} **by** *auto*

lemma *cols-transpose*[*simp*]: *cols A*^{T} = *rows A* **unfolding** *cols-def rows-def* **by** *auto*
lemma *rows-transpose*[*simp*]: *rows A*^{T} = *cols A* **unfolding** *cols-def rows-def* **by** *auto*
lemma *list-of-vec-vec* [*simp*]: *list-of-vec* (*vec n f*) = *map f* [*0*..<*n*]
by (*transfer*, *auto simp*: *mk-vec-def*)

lemma *list-of-vec-0* [*simp*]: *list-of-vec* (0_v *n*) = *replicate n 0*
by (*simp add*: *zero-vec-def map-replicate-trivial*)

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 −
have *dim-eq*: *dim-row M* = *dim-col M* **using** *M-carrier* **by** *auto*
have *m*: *map-mat f M* $$ (*i*, *i*) = *f* (*M* $$ (*i*, *i*)) **if** *i*: *i* < *dim-row M* **for** *i*
using *dim-eq i* **by** *auto*
show *?thesis*
by (*rule nth-equalityI*, *insert m*, *auto simp add*: *diag-mat-def M-carrier*)
qed

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−
  have ∀ *x*∈*set vs*. *dim-vec x* = *n* **using** *x* **by** *auto*
  then show *?thesis* **by** (*auto simp add*: *mat-eq-iff map-vec-def mat-of-rows-def*)
qed

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−
  have ∀ *x*∈*set vs*. *dim-vec x* = *n* **using** *x* **by** *auto*
  then show *?thesis* **by** (*auto simp add*: *mat-eq-iff map-vec-def mat-of-cols-def*)
qed

lemma *vec-of-list-map* [*simp*]: *vec-of-list* (*map f xs*) = *map-vec f* (*vec-of-list xs*)
  unfolding *map-vec-def* **by** (*transfer*, *auto simp add*: *mk-vec-def*)

lemma *map-vec*: *map-vec f* (*vec n g*) = *vec n* (*f o g*) **by** *auto*

lemma *mat-of-cols-Cons-index-0*: *i* < *n* ⟹ *mat-of-cols n* (*w* # *ws*) $$ (*i*, *0*) = *w* $ *i*
  by (*unfold mat-of-cols-def*, *transfer′*, *auto simp*: *mk-mat-def*)

lemma *nth-map-out-of-bound*: *i* ≥ *length xs* ⟹ *map f xs* ! *i* = [] ! (*i* − *length xs*)
  by (*induct xs arbitrary*:*i*, *auto*)

lemma *mat-of-cols-Cons-index-Suc*:
  i < *n* ⟹ *mat-of-cols n* (*w* # *ws*) $$ (*i*, *Suc j*) = *mat-of-cols n ws* $$ (*i*,*j*)
  by (*unfold mat-of-cols-def*, *transfer*, *auto simp*: *mk-mat-def undef-mat-def nth-append nth-map-out-of-bound*)

lemma *mat-of-cols-index*: *i* < *n* ⟹ *j* < *length ws* ⟹ *mat-of-cols n ws* $$ (*i*,*j*) = *ws* ! *j* $ *i*
  by (*unfold mat-of-cols-def*, *auto*)

lemma *mat-of-rows-index*: *i* < *length rs* ⟹ *j* < *n* ⟹ *mat-of-rows n rs* $$ (*i*,*j*) = *rs* ! *i* $ *j*
  by (*unfold mat-of-rows-def*, *auto*)

lemma *transpose-mat-of-rows*: (*mat-of-rows n vs*)^T = *mat-of-cols n vs*
  by (*auto intro*!: *eq-matI simp*: *mat-of-rows-index mat-of-cols-index*)

lemma *transpose-mat-of-cols*: (*mat-of-cols n vs*)^T = *mat-of-rows n vs*
  by (*auto intro*!: *eq-matI simp*: *mat-of-rows-index mat-of-cols-index*)

lemma *nth-list-of-vec* [*simp*]:
  assumes *i* < *dim-vec v* **shows** *list-of-vec v* ! *i* = *v* $ *i*

using *assms* **by** (*transfer*, *auto*)

lemma *length-list-of-vec* [*simp*]:
length (*list-of-vec* *v*) = *dim-vec* *v* **by** (*transfer*, *auto*)

lemma *vec-eq-0-iff*:
$v = 0_v\ n \longleftrightarrow n = \textit{dim-vec}\ v \wedge (n = 0 \vee \textit{set}\ (\textit{list-of-vec}\ v) = \{0\})$ (**is** *?l* ⟷ *?r*)
proof
show *?l* ⟹ *?r* **by** *auto*
show *?r* ⟹ *?l* **by** (*intro iffI eq-vecI*, *force simp*: *set-conv-nth*, *force*)
qed

lemma *list-of-vec-vCons*[*simp*]: *list-of-vec* (*vCons a v*) = *a* # *list-of-vec v* (**is** *?l* = *?r*)
proof (*intro nth-equalityI*)
fix *i*
assume *i* < *length ?l*
then show *?l* ! *i* = *?r* ! *i* **by** (*cases i*, *auto*)
qed *simp*

lemma *append-vec-vCons*[*simp*]: $\textit{vCons}\ a\ v\ @_v\ w = \textit{vCons}\ a\ (v\ @_v\ w)$ (**is** *?l* = *?r*)
proof (*unfold vec-eq-iff*, *intro conjI allI impI*)
fix *i* **assume** *i* < *dim-vec ?r*
then show *?l* \$ *i* = *?r* \$ *i* **by** (*cases i*; *subst index-append-vec*, *auto*)
qed *simp*

lemma *append-vec-vNil*[*simp*]: $\textit{vNil}\ @_v\ v = v$
by (*unfold vec-eq-iff*, *auto*)

lemma *list-of-vec-append*[*simp*]: *list-of-vec* $(v\ @_v\ w)$ = *list-of-vec v* @ *list-of-vec w*
by (*induct v*, *auto*)

lemma *transpose-mat-eq*[*simp*]: $A^T = B^T \longleftrightarrow A = B$
using *transpose-transpose* **by** *metis*

lemma *mat-col-eqI*: **assumes** *cols*: $\bigwedge i.\ i < \textit{dim-col}\ B \Longrightarrow \textit{col}\ A\ i = \textit{col}\ B\ i$
and *dims*: *dim-row A* = *dim-row B* *dim-col A* = *dim-col B*
shows *A* = *B*
by(*subst transpose-mat-eq*[*symmetric*], *rule eq-rowI*,*insert assms*,*auto*)

lemma *upper-triangular-imp-distinct*:
assumes *A*: *A* ∈ *carrier-mat n n*
and *tri*: *upper-triangular A*
and *diag*: *0* ∉ *set* (*diag-mat A*)
shows *distinct* (*rows A*)
proof−
{ fix *i* **and** *j*

assume *eq*: *rows A* ! *i* = *rows A* ! *j* **and** *ij*: *i* < *j* **and** *jn*: *j* < *n*
from *tri A ij jn* **have** *rows A* ! *j* $ *i* = *0* **by** (*auto dest*!:*upper-triangularD*)
with *eq* **have** *rows A* ! *i* $ *i* = *0* **by** *auto*
with *diag ij jn A* **have** *False* **by** (*auto simp*: *diag-mat-def*)
}
with *A* **show** *?thesis* **by** (*force simp*: *distinct-conv-nth nat-neq-iff*)
qed

lemma *dim-vec-of-list*[*simp*] :*dim-vec* (*vec-of-list as*) = *length as* **by** *transfer auto*

lemma *list-vec*: *list-of-vec* (*vec-of-list xs*) = *xs*
by (*transfer*, *metis* (*mono-tags*, *lifting*) *atLeastLessThan-iff map-eq-conv map-nth mk-vec-def old.prod.case set-upt*)

lemma *vec-list*: *vec-of-list* (*list-of-vec v*) = *v*
apply *transfer* **unfolding** *mk-vec-def* **by** *auto*

lemma *index-vec-of-list*: *i*<*length xs* ⟹ (*vec-of-list xs*) $ *i* = *xs* ! *i*
by (*metis vec.abs-eq index-vec vec-of-list.abs-eq*)

lemma *vec-of-list-index*: *vec-of-list xs* $ *j* = *xs* ! *j*
apply *transfer* **unfolding** *mk-vec-def* **unfolding** *undef-vec-def*
by (*simp*, *metis append-Nil2 nth-append*)

lemma *list-of-vec-index*: *list-of-vec v* ! *j* = *v* $ *j*
by (*metis vec-list vec-of-list-index*)

lemma *list-of-vec-map*: *list-of-vec xs* = *map* (($) *xs*) [*0*..<*dim-vec xs*] **by** *transfer auto*

definition *component-mult v w* = *vec* (*min* (*dim-vec v*) (*dim-vec w*)) (λ*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*
unfolding *vec-set-def* **by** *auto*

lemma *index-component-mult*:
assumes *i* < *dim-vec v i* < *dim-vec w*
shows *component-mult v w* $ *i* = *v* $ *i* ∗ *w* $ *i*
unfolding *component-mult-def* **using** *assms index-vec* **by** *auto*

lemma *dim-component-mult*:
dim-vec (*component-mult v w*) = *min* (*dim-vec v*) (*dim-vec w*)
unfolding *component-mult-def* **using** *index-vec* **by** *auto*

lemma *vec-setE*:

assumes $a \in set_v\ v$
obtains i **where** $v\$i = a\ i{<}dim\text{-}vec\ v$ **using** *assms* **unfolding** *vec-set-def* **by** *blast*

lemma *vec-setI*:
assumes $v\$i = a\ i{<}dim\text{-}vec\ v$
shows $a \in set_v\ v$ **using** *assms* **unfolding** *vec-set-def* **using** *image-eqI lessThan-iff* **by** *blast*

lemma *set-list-of-vec*: $set\ (list\text{-}of\text{-}vec\ v) = set_v\ v$ **unfolding** *vec-set-def* **by** *transfer auto*

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

definition *conjugate-vec* :: $'a$ :: *conjugate vec* $\Rightarrow$ $'a$ *vec*
  where $conjugate\ v = vec\ (dim\text{-}vec\ v)\ (\lambda i.\ conjugate\ (v\ \$\ i))$

lemma *conjugate-vCons* [*simp*]:
  $conjugate\ (vCons\ a\ v) = vCons\ (conjugate\ a)\ (conjugate\ v)$
  by (*auto simp*: *vec-Suc conjugate-vec-def*)

lemma *dim-vec-conjugate*[*simp*]: $dim\text{-}vec\ (conjugate\ v) = dim\text{-}vec\ v$
  unfolding *conjugate-vec-def* **by** *auto*

lemma *carrier-vec-conjugate*[*simp*]: $v \in carrier\text{-}vec\ n \Longrightarrow conjugate\ v \in carrier\text{-}vec\ n$
  by (*auto intro*!: *carrier-vecI*)

lemma *vec-index-conjugate*[*simp*]:
  shows $i < dim\text{-}vec\ v \Longrightarrow conjugate\ v\ \$\ i = conjugate\ (v\ \$\ i)$
  unfolding *conjugate-vec-def* **by** *auto*

instance
proof
  fix $v\ w$ :: $'a$ *vec*
  show $conjugate\ (conjugate\ v) = v$ **by** (*induct v*, *auto simp*: *conjugate-vec-def*)
  let $?v = conjugate\ v$
  let $?w = conjugate\ w$
  show $conjugate\ v = conjugate\ w \longleftrightarrow v = w$
  proof(*rule iffI*)
    assume *cvw*: $?v = ?w$ **show** $v = w$
    proof(*rule*)
      have $dim\text{-}vec\ ?v = dim\text{-}vec\ ?w$ **using** *cvw* **by** *auto*
      then show *dim*: $dim\text{-}vec\ v = dim\text{-}vec\ w$ **by** *simp*
      fix i **assume** i: $i < dim\text{-}vec\ w$
      then have $conjugate\ v\ \$\ i = conjugate\ w\ \$\ i$ **using** *cvw* **by** *auto*
      then have $conjugate\ (v\$i) = conjugate\ (w\ \$\ i)$ **using** *i dim* **by** *auto*

```
    then show v $ i = w $ i by auto
  qed
 qed auto
qed

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
  by (rule, insert dim, auto simp: conjugate-dist-add)

lemma uminus-conjugate-vec:
  fixes v w :: 'a :: conjugatable-ring vec
  shows − (conjugate v) = conjugate (− v)
  by (rule, auto simp:conjugate-neg)

lemma conjugate-zero-vec[simp]:
  conjugate (0v n :: 'a :: conjugatable-ring vec) = 0v n by auto

lemma conjugate-vec-0[simp]:
  conjugate (vec 0 f) = vec 0 f by auto

lemma sprod-vec-0[simp]: v · vec 0 f = 0
  by(auto simp: scalar-prod-def)

lemma conjugate-zero-iff-vec[simp]:
  fixes v :: 'a :: conjugatable-ring vec
  shows conjugate v = 0v n ⟷ v = 0v n
  using conjugate-cancel-iff[of - 0v n :: 'a vec] by auto

lemma conjugate-smult-vec:
  fixes k :: 'a :: conjugatable-ring
  shows conjugate (k ·v v) = conjugate k ·v conjugate v
  using conjugate-dist-mul by (intro eq-vecI, auto)

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 (insert w v, induct w arbitrary: v rule:carrier-vec-induct)
  case 0 then show ?case by (cases v, auto)
next
  case (Suc n b w) then show ?case
   by (cases v, auto dest: carrier-vecD simp:conjugate-dist-add conjugate-dist-mul)
qed

abbreviation cscalar-prod :: 'a vec ⇒ 'a vec ⇒ 'a :: conjugatable-ring (infix ‹·c›
```

70)
  where $(\cdot c) \equiv \lambda v\ w.\ v \cdot \mathit{conjugate}\ w$

lemma *conjugate-conjugate-sprod*[*simp*]:
  assumes *v*[*simp*]: $v : \mathit{carrier\text{-}vec}\ n$ **and** *w*[*simp*]: $w : \mathit{carrier\text{-}vec}\ n$
  shows $\mathit{conjugate}\ (\mathit{conjugate}\ v \cdot w) = v \cdot c\ w$
  apply (*subst conjugate-sprod-vec*[*of* - *n*]) **by** *auto*

lemma *conjugate-vec-sprod-comm*:
  fixes $v\ w :: 'a :: \{\mathit{conjugatable\text{-}ring},\ \mathit{comm\text{-}ring}\}\ \mathit{vec}$
  assumes $v : \mathit{carrier\text{-}vec}\ n$ **and** $w : \mathit{carrier\text{-}vec}\ n$
  shows $v \cdot c\ w = (\mathit{conjugate}\ w \cdot v)$
  unfolding *scalar-prod-def* **using** *assms* **by**(*subst sum.ivl-cong*, *auto simp*: *ac-simps*)

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 (*induct v*)
  case *vNil*
  then show *?case* **by** *auto*
next
  case (*vCons a v*)
  then show *?case* **using** *conjugate-square-positive*[*of a*] **by** *auto*
qed

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 (*insert assms*, *induct rule*: *carrier-vec-induct*)
  case *0*
  then show *?case* **by** *auto*
next
  case (*Suc n a v*)
  then show *?case*
    using *conjugate-square-positive*[*of a*] *conjugate-square-ge-0-vec*[*of v*]
    by (*auto simp*: *le-less add-nonneg-eq-0-iff zero-vec-Suc*)
qed

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 $v \cdot c\ v > 0 \longleftrightarrow v \neq 0_v\ n$
  using *assms* **by** (*auto simp*: *less-le*)

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

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 $'a$ *vec-impl* $= \{(n,v :: 'a\ iarray).\ IArray.length\ v = n\}$ **by** *auto*
typedef $'a$ *mat-impl* $= \{(nr,nc,m :: 'a\ iarray\ iarray).$
  $IArray.length\ m = nr \wedge IArray.all\ (\lambda\ r.\ IArray.length\ r = nc)\ m\}$
  by (*rule exI*[*of - (0,0,IArray* [])], *auto*)

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

lift-definition *vec-impl* :: $'a$ *vec-impl* $\Rightarrow$ $'a$ *vec* **is**
  $\lambda\ (n,v).\ (n,mk\text{-}vec\ n\ (IArray.sub\ v))$ **by** *auto*

lift-definition *vec-add-impl* :: $'a::plus$ *vec-impl* $\Rightarrow$ $'a$ *vec-impl* $\Rightarrow$ $'a$ *vec-impl* **is**
  $\lambda\ (n,v)\ (m,w).$
  $(n,\ IArray.of\text{-}fun\ (\lambda i.\ IArray.sub\ v\ i + IArray.sub\ w\ i)\ n)$
by *auto*

lift-definition *mat-impl* :: $'a$ *mat-impl* $\Rightarrow$ $'a$ *mat* **is**
  $\lambda\ (nr,nc,m).\ (nr,nc,mk\text{-}mat\ nr\ nc\ (\lambda\ (i,j).\ IArray.sub\ (IArray.sub\ m\ i)\ j))$ **by** *auto*

lift-definition *vec-of-list-impl* :: $'a$ *list* $\Rightarrow$ $'a$ *vec-impl* **is**
  $\lambda\ v.\ (length\ v,\ IArray\ v)$ **by** *auto*

lift-definition *list-of-vec-impl* :: $'a$ *vec-impl* $\Rightarrow$ $'a$ *list* **is**
  $\lambda\ (n,v).\ IArray.list\text{-}of\ v$ **.**

lift-definition *vec-of-fun* :: $nat \Rightarrow (nat \Rightarrow 'a) \Rightarrow 'a$ *vec-impl* **is**
  $\lambda\ n\ f.\ (n,\ IArray.of\text{-}fun\ f\ n)$ **by** *auto*

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*) **by** *auto*

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

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* **.**

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

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* **.**

lift-definition *dim-vec-impl* :: *'a vec-impl* ⇒ *nat* **is** *fst* **.**

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

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

lemma *vec-code*[*code*]: *vec n f* = *vec-impl* (*vec-of-fun n f*)
by (*transfer*, *auto simp*: *mk-vec-def*)

lemma *mat-code*[*code*]: *mat nr nc f* = *mat-impl* (*mat-of-fun nr nc f*)
by (*transfer*, *auto simp*: *mk-mat-def*, *intro ext*, *clarsimp*,
auto intro: *undef-cong-mat*)

lemma *vec-of-list*[*code*]: *vec-of-list v* = *vec-impl* (*vec-of-list-impl v*)
by (*transfer*, *auto simp*: *mk-vec-def*)

lemma *list-of-vec-code*[*code*]: *list-of-vec* (*vec-impl v*) = *list-of-vec-impl v*
by (*transfer*, *auto simp*: *mk-vec-def*, *case-tac b*, *auto intro*: *nth-equalityI*)

lemma *empty-nth*: ¬ *i* < *length x* ⟹ *x* ! *i* = [] ! (*i* − *length x*)
by (*metis append-Nil2 nth-append*)

lemma *undef-vec*: ¬ *i* < *length x* ⟹ *undef-vec* (*i* − *length x*) = *x* ! *i*
unfolding *undef-vec-def* **by** (*rule empty-nth*[*symmetric*])

lemma *vec-index-code*[*code*]: (*vec-impl v*) $ *i* = *vec-index-impl v i*
by (*transfer*, *auto simp*: *mk-vec-def*, *case-tac b*, *auto simp*: *undef-vec*)

lemma *index-mat-code*[*code*]: (*mat-impl m*) $$ *ij* = (*index-mat-impl m ij* :: *'a*)
proof (*transfer*, *unfold o-def*, *clarify*)

fix *m* :: *'a iarray iarray* **and** *i j nc*
assume *all*: *IArray.all* (λ*r*. *IArray.length r* = *nc*) *m*
obtain *mm* **where** *m*: *m* = *IArray mm* **by** (*cases m*)
with *all* **have** *all*: $\bigwedge$ *v*. *v* ∈ *set mm* ⟹ *IArray.length v* = *nc* **by** *auto*
show *snd* (*snd* (*IArray.length m*, *nc*, *mk-mat* (*IArray.length m*) *nc* (λ(*i*, *y*). *m* !! *i* !! *y*))) (*i*, *j*) =
(*if i* < *IArray.length m then m* !! *i* !! *j*
else IArray ([] ! (*i* − *IArray.length m*)) !! *j*) (**is** *?l* = *?r*)
proof (*cases i* < *length mm*)
case *False*
hence $\bigwedge$ *f*. ¬ *i* < *length* (*map f* [*0*..<*length mm*]) **by** *simp*
note [*simp*] = *empty-nth*[*OF this*]
have *?l* = [] ! (*i* − *length mm*) ! *j* **using** *False* **unfolding** *m mk-mat-def undef-mat-def* **by** *simp*
also have ... = *?r* **unfolding** *m* **by** (*simp add*: *False empty-nth*[*OF False*])
finally show *?thesis* **.**
next
case *True*
obtain *v* **where** *mm*: *mm* ! *i* = *IArray v* **by** (*cases mm* ! *i*)
with *True all*[*of mm* ! *i*] **have** *len*: *length v* = *nc* **unfolding** *set-conv-nth* **by** *force*
from *mm True* **have** *?l* = *map* ((!) *v*) [*0*..<*nc*] ! *j* (**is** - = *?m*) **unfolding** *m mk-mat-def undef-mat-def* **by** *simp*
also have *?m* = *m* !! *i* !! *j*
proof (*cases j* < *length v*)
case *True*
thus *?thesis* **unfolding** *m* **using** *mm len* **by** *auto*
next
case *False*
hence *j*: ¬ *j* < *length* (*map* ((!) *v*) [*0*..<*length v*]) **by** *simp*
show *?thesis* **unfolding** *m* **using** *mm len* **by** (*auto simp*: *empty-nth*[*OF j*] *empty-nth*[*OF False*])
qed
also have ... = *?r* **using** *True m* **by** *simp*
finally show *?thesis* **.**
qed
qed

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
by (*auto split*: *if-splits simp*: *list-all-iff*)

lemma *mat-of-rows-list-impl*: *mat-of-rows-list-impl n rs* = *Some A* ⟹ *mat-impl A* = *mat-of-rows-list n rs*
unfolding *mat-of-rows-list-def*
by (*transfer*, *auto split*: *if-splits simp*: *list-all-iff intro*!: *cong-mk-mat*)

```
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 (cases mat-of-rows-list-impl nc vs)
  case (Some A)
  from mat-of-rows-list-impl[OF this] show ?thesis unfolding Some by simp
next
  case None
  show ?thesis unfolding None unfolding mat-of-rows-list-def mat-of-rows-def
    by (intro eq-matI, auto)
qed

lemma dim-vec-code[code]: dim-vec (vec-impl v) = dim-vec-impl v
  by (transfer, auto)

lemma dim-row-code[code]: dim-row (mat-impl m) = dim-row-impl m
  by (transfer, auto)

lemma dim-col-code[code]: dim-col (mat-impl m) = dim-col-impl m
  by (transfer, auto)

instantiation vec :: (type)equal
begin
  definition (equal-vec :: ('a vec ⇒ 'a vec ⇒ bool)) = (=)
instance
  by (intro-classes, auto simp: equal-vec-def)
end

instantiation mat :: (type)equal
begin
  definition (equal-mat :: ('a mat ⇒ 'a mat ⇒ bool)) = (=)
instance
  by (intro-classes, auto simp: equal-mat-def)
end

lemma veq-equal-code[code]: HOL.equal (vec-impl (v1 :: 'a vec-impl)) (vec-impl
v2) = vec-equal-impl v1 v2
proof –
  {
    fix x1 x2 :: 'a list
    assume len: length x1 = length x2
      and index: (λi. if i < length x2 then IArray x1 !! i else undef-vec (i − length
(IArray.list-of (IArray x1)))) =
              (λi. if i < length x2 then IArray x2 !! i else undef-vec (i − length
(IArray.list-of (IArray x2))))
    have x1 = x2
    proof (intro nth-equalityI[OF len])
      fix i
```

```
      assume i < length x1
      with fun-cong[OF index, of i] len show x1 ! i = x2 ! i by simp
    qed
  } note * = this
  show ?thesis unfolding equal-vec-def
    by (transfer, insert *, auto simp: mk-vec-def, case-tac b, case-tac ba, auto)
qed

lemma mat-equal-code[code]: HOL.equal (mat-impl (m1 :: 'a mat-impl)) (mat-impl
m2) = mat-equal-impl m1 m2
proof −
  show ?thesis unfolding equal-mat-def
  proof (transfer, auto, case-tac b, case-tac ba, auto)
    fix x1 x2 :: 'a iarray list and nc
    assume len: ∀ r∈set x1. length (IArray.list-of r) = nc
      ∀ r∈set x2. length (IArray.list-of r) = nc
      length x1 = length x2
     and index: mk-mat (length x2) nc (λ(i, j). x1 ! i !! j) = mk-mat (length x2)
nc (λ(i, j). x2 ! i !! j)
    show x1 = x2
    proof (rule nth-equalityI[OF len(3)])
      fix i
      assume i: i < length x1
      obtain ia1 where 1: x1 ! i = IArray ia1 by (cases x1 ! i)
      obtain ia2 where 2: x2 ! i = IArray ia2 by (cases x2 ! i)
      from i 1 len(1) have l1: length ia1 = nc using nth-mem by fastforce
      from i 2 len(2−3) have l2: length ia2 = nc using nth-mem by fastforce
      from l1 l2 have l: length ia1 = length ia2 by simp
      show x1 ! i = x2 ! i unfolding 1 2
      proof (simp, rule nth-equalityI[OF l])
        fix j
        assume j: j < length ia1
        with fun-cong[OF index, of (i,j)] i len(3)
        have x1 ! i !! j = x2 ! i !! j
          by (simp add: mk-mat-def l1)
        thus ia1 ! j = ia2 ! j unfolding 1 2 by simp
      qed
    qed
  qed
qed

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
```

$= nr \wedge$ *dim-col* $A = nc$) **by** *auto*
lemma *carrier-vec-code*[*code*]: *carrier-vec* n = *Collect-set* (λ v. *dim-vec* $v = n$)
unfolding *carrier-vec-def* **by** *auto*

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* :: ($'a \Rightarrow 'a \Rightarrow 'a$) $\Rightarrow$ *nat* $\Rightarrow$ $'a \Rightarrow 'a$ *mat* $\Rightarrow$ $'a$ *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* $\Rightarrow$ $'a$ :: *semiring-1* $\Rightarrow$ $'a$ *mat* $\Rightarrow$ $'a$ *mat* (‹*multrow*›)
where
multrow $\equiv$ *mat-multrow-gen* (($*$))

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

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

definition *mat-swaprows* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ $'a$ *mat* $\Rightarrow$ $'a$ *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* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ $'a$ :: *semiring-1 mat* **where**
swaprows-mat n k l = *mat n n*
(λ (i,j). *if* $k = i \wedge l = j \vee k = j \wedge l = i \vee i = j \wedge i \neq k \wedge i \neq l$ *then 1 else 0*)

definition *mat-addrow-gen* :: ($'a \Rightarrow 'a \Rightarrow 'a$) $\Rightarrow$ ($'a \Rightarrow 'a \Rightarrow 'a$) $\Rightarrow$ $'a \Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ $'a$ *mat* $\Rightarrow$ $'a$ *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* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ $'a$ *mat* $\Rightarrow$ $'a$ *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*
unfolding *mat-multrow-def* **by** *auto*

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*
unfolding *multrow-mat-def* **by** *auto*

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*
unfolding *mat-swaprows-def* **by** *auto*

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*
unfolding *swaprows-mat-def* **by** *auto*

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*

unfolding *mat-addrow-def* **by** *auto*

lemma *index-mat-addrow-mat*[*simp*]:
$i < n \Longrightarrow j < n \Longrightarrow$ *addrow-mat n a k l* \$\$ (*i*,*j*) =
(*if* $k = i \wedge l = j$ *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*
unfolding *addrow-mat-def* **by** *auto*

lemma *multrow-carrier*[*simp*]: (*mat-multrow-gen mul k a A* $\in$ *carrier-mat n nc*) = (*A* $\in$ *carrier-mat n nc*)
unfolding *carrier-mat-def* **by** *fastforce*

lemma *multrow-mat-carrier*[*simp*]: *multrow-mat n k a* $\in$ *carrier-mat n n*
unfolding *carrier-mat-def* **by** *auto*

lemma *addrow-mat-carrier*[*simp*]: *addrow-mat n a k l* $\in$ *carrier-mat n n*
unfolding *carrier-mat-def* **by** *auto*

lemma *swaprows-mat-carrier*[*simp*]: *swaprows-mat n k l* $\in$ *carrier-mat n n*
unfolding *carrier-mat-def* **by** *auto*

lemma *swaprows-carrier*[*simp*]: (*swaprows k l A* $\in$ *carrier-mat n nc*) = (*A* $\in$ *carrier-mat n nc*)
unfolding *carrier-mat-def* **by** *fastforce*

lemma *addrow-carrier*[*simp*]: (*mat-addrow-gen ad mul a k l A* $\in$ *carrier-mat n nc*) = (*A* $\in$ *carrier-mat n nc*)
unfolding *carrier-mat-def* **by** *fastforce*

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*
by (*rule eq-vecI*, *auto*)

lemma *multrow-mat*: **assumes** *A*: *A* $\in$ *carrier-mat n nc*
shows *multrow k a A* = *multrow-mat n k a* $*$ *A*
by (*rule eq-matI*, *insert A*, *auto simp*: *row-multrow smult-scalar-prod-distrib*[*of - n*])

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*
by (*rule eq-vecI*, *auto*)

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*
by (*rule eq-matI*, *insert l A*, *auto simp*: *row-addrow*

add-scalar-prod-distrib[*of* - *n*] *smult-scalar-prod-distrib*[*of* - *n*])

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*
  by (*rule eq-vecI*, *auto*)

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*
  by (*rule eq-matI*, *insert A k*, *auto simp*: *row-swaprows*)

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 −
  have *swaprows-mat n k l* $*$ *swaprows-mat n k l* =
    swaprows-mat n k l $*$ (*swaprows-mat n k l* $*$ 1_m *n*)
    by (*simp add*: *right-mult-one-mat*[*of* - *n*])
  also have *swaprows-mat n k l* $*$ 1_m *n* = *swaprows k l* (1_m *n*)
    by (*rule swaprows-mat*[*symmetric*, *OF* - *k l*, *of* - *n*], *simp*)
  also have *swaprows-mat n k l* $*$ ... = *swaprows k l* ...
    by (*rule swaprows-mat*[*symmetric*, *of* - - *n*], *insert k l*, *auto*)
  also have ... = 1_m *n*
    by (*rule eq-matI*, *insert k l*, *auto*)
  finally show *?thesis* **.**
qed

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 −
  interpret *m*: *semiring ring-mat TYPE*(*'a*) *n b* **by** (*rule semiring-mat*)
  show *?thesis* **unfolding** *Units-def*
    by (*rule*, *rule conjI*[*OF* - *bexI*[*of* - *swaprows-mat n k l*]],
    auto simp: *ring-mat-def swaprows-mat-inv*[*OF k l*] *swaprows-mat-inv*[*OF l k*])
qed

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 −
  have *addrow-mat n a k l* $*$ *addrow-mat n* (− *a*) *k l* =
    addrow-mat n a k l $*$ (*addrow-mat n* (− *a*) *k l* $*$ 1_m *n*)
    by (*simp add*: *right-mult-one-mat*[*of* - *n*])
  also have *addrow-mat n* (− *a*) *k l* $*$ 1_m *n* = *addrow* (− *a*) *k l* (1_m *n*)
    by (*rule addrow-mat*[*symmetric*, *of* - - *n*], *insert k l*, *auto*)
  also have *addrow-mat n a k l* $*$ ... = *addrow a k l* ...
    by (*rule addrow-mat*[*symmetric*, *of* - - *n*], *insert k l*, *auto*)
  also have ... = 1_m *n*

by (*rule eq-matI*, *insert k l neq*, *auto*, *algebra*)
finally show *?thesis* **.**
qed

lemma *addrow-mat-Unit*: **assumes** *k*: $k < n$ **and** *l*: $l < n$ **and** *neq*: $k \neq l$
shows *addrow-mat n a k l* $\in$ *Units* (*ring-mat TYPE*('*a* :: *comm-ring-1*) *n b*)
proof −
interpret *m*: *semiring ring-mat TYPE*('*a*) *n b* **by** (*rule semiring-mat*)
show *?thesis* **unfolding** *Units-def*
by (*rule*, *rule conjI*[*OF* - *bexI*[*of* - *addrow-mat n* (− *a*) *k l*]], *insert neq*,
auto simp: *ring-mat-def addrow-mat-inv*[*OF k l neq*],
rule trans[*OF* - *addrow-mat-inv*[*OF k l neq*, *of* − *a*]], *auto*)
qed

lemma *multrow-mat-inv*: **assumes** *k*: $k < n$ **and** *a*: (*a* :: '*a* :: *division-ring*) $\neq$ *0*
shows *multrow-mat n k a* $*$ *multrow-mat n k* (*inverse a*) = 1_m *n*
proof −
have *multrow-mat n k a* $*$ *multrow-mat n k* (*inverse a*) =
multrow-mat n k a $*$ (*multrow-mat n k* (*inverse a*) $*$ 1_m *n*)
using *k* **by** (*simp add*: *right-mult-one-mat*[*of* - *n*])
also have *multrow-mat n k* (*inverse a*) $*$ 1_m *n* = *multrow k* (*inverse a*) (1_m *n*)
by (*rule multrow-mat*[*symmetric*, *of* - - *n*], *insert k*, *auto*)
also have *multrow-mat n k a* $*$ ... = *multrow k a* ...
by (*rule multrow-mat*[*symmetric*, *of* - - *n*], *insert k*, *auto*)
also have ... = 1_m *n*
by (*rule eq-matI*, *insert a k a*, *auto*)
finally show *?thesis* **.**
qed

lemma *multrow-mat-Unit*: **assumes** *k*: $k < n$ **and** *a*: (*a* :: '*a* :: *division-ring*) $\neq$ *0*
shows *multrow-mat n k a* $\in$ *Units* (*ring-mat TYPE*('*a*) *n b*)
proof −
from *a* **have** *ia*: *inverse a* $\neq$ *0* **by** *auto*
interpret *m*: *semiring ring-mat TYPE*('*a*) *n b* **by** (*rule semiring-mat*)
show *?thesis* **unfolding** *Units-def*
by (*rule*, *rule conjI*[*OF* - *bexI*[*of* - *multrow-mat n k* (*inverse a*)]], *insert a*,
auto simp: *ring-mat-def multrow-mat-inv*[*OF k*],
rule trans[*OF* - *multrow-mat-inv*[*OF k ia*]], *insert a*, *auto*)
qed

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* :: $'a \Rightarrow 'a \Rightarrow 'a$
and *times* :: $'a \Rightarrow 'a \Rightarrow 'a$
begin

definition *eliminate-entries-gen* :: $(nat \Rightarrow 'a) \Rightarrow 'a\ mat \Rightarrow nat \Rightarrow nat \Rightarrow 'a\ mat$
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*
unfolding *eliminate-entries-gen-def* **by** *auto*

lemma *dimc-eliminate-entries-rec*[*simp*]: *dim-col* (*eliminate-entries-rec B i as*) = *dim-col B*
by (*induct as arbitrary*: *B*, *auto simp*: *Let-def*)

lemma *dimr-eliminate-entries-rec*[*simp*]: *dim-row* (*eliminate-entries-rec B i as*) = *dim-row B*
by (*induct as arbitrary*: *B*, *auto simp*: *Let-def*)

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*
unfolding *carrier-mat-def* **by** *auto*
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 −
let *?ais* = λ *is*. *map* (λ *i*. (− *A* $$ (*i*, *J*), *i*)) (*filter* (λ *i*. *i* ≠ *I*) *is*)
define *one-go* **where** *one-go* = (λ *B is*. *mat* (*dim-row B*) (*dim-col B*) (λ (*i*, *j*).
if i ≠ *I* ∧ *i* ∈ *set is then B* $$ (*i*,*j*) − (*A* $$ (*i*,*J*)) ∗ *B* $$ (*I*,*j*) *else B* $$ (*i*,*j*)))
{
fix *is* :: *nat list*
assume *distinct is*
from ∗ *this* **have** *eliminate-entries-rec B I* (*?ais is*) = *one-go B is*
proof (*induct is arbitrary*: *B*)
case *Nil*
show *?case* **unfolding** *one-go-def*
by (*rule eq-matI*, *auto*)

next
case (*Cons i is*)
note I = *Cons*(*2*) **note** *dim* = *Cons*(*3*)
note *II* = *Cons*(*2*)[*folded dim*]
let *?B* = *addrow* (− *A* $$ (*i*, *J*)) *i I B*
from *Cons*(*4*) *I dim* **have** *I* < *dim-row A dim-row ?B* = *dim-row A* **and** *dist*: *distinct is* **by** *auto*
note *IH* = *Cons*(*1*)[*OF this*]
from *Cons*(*4*) **have** *i*: *i* ∉ *set is* **by** *auto*
show *?case*
proof (*cases i* = *I*)
case *False*
hence *id*: *?ais* (*i* # *is*) = (− *A* $$ (*i*, *J*), *i*) # *?ais is* **by** *simp*
show *?thesis* **unfolding** *id eliminate-entries-rec.simps IH*
unfolding *one-go-def index-mat-addrow*
proof (*rule eq-matI*, *goal-cases*)
case (*1 ii jj*)
hence *ii*: *ii* < *dim-row B* **and** *jj*: *jj* < *dim-col B* **and** *iiA*: *ii* < *dim-row A* **using** *dim* **by** *auto*
show *?case* **unfolding** *index-mat*[*OF ii jj*] *split*
index-mat-addrow(*1*)[*OF ii jj*] *index-mat-addrow*(*1*)[*OF II jj*]
using *i False* **by** *auto*
qed *auto*
next
case *True*
hence *id*: *?ais* (*i* # *is*) = *?ais is* **by** *simp*
show *?thesis* **unfolding** *id Cons*(*1*)[*OF I dim dist*]
unfolding *one-go-def True* **by** *auto*
qed
qed
} **note** *main* = *this*
show *?thesis*
by (*subst main*, *force*, *unfold one-go-def eliminate-entries-gen-def*, *rule eq-matI*,

insert ∗, *auto*)
qed

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 (*induct is*)
case *Nil*
thus *?case* **by** (*intro bexI*[*of* - 1_m *nr*], *auto simp*: *Units-def ring-mat-def*)
next
case (*Cons ai′ is*)
obtain *a i′* **where** *ai′*: *ai′* = (*a*,*i′*) **by** *force*
let *?U* = *Units* (*ring-mat TYPE*(′*a*) *nr b*)
interpret *m*: *ring ring-mat TYPE*(′*a*) *nr b* **by** (*rule ring-mat*)

from *Cons(1)[OF Cons(2−3)]*
obtain *P* **where** *P*: *P* ∈ *?U* **and** *id*: ⋀ *B nc* . *B* ∈ *carrier-mat nr nc* ⟹
eliminate-entries-rec B i is = *P* ∗ *B* **by** *force*
let *?Add* = *addrow-mat nr a i′ i*
have *Add*: *?Add* ∈ *?U*
by (*rule addrow-mat-Unit*, *insert Cons ai′*, *auto*)
from *m.Units-m-closed*[*OF P Add*] **have** *PI*: *P* ∗ *?Add* ∈ *?U* **unfolding** *ring-mat-def*
by *simp*
from *m.Units-closed*[*OF P*] **have** *P*: *P* ∈ *carrier-mat nr nr* **unfolding** *ring-mat-def*
by *simp*
show *?case*
proof (*rule bexI*[*OF - PI*], *intro allI impI*)
fix *B* :: *′a mat* **and** *nc*
assume *BB*: *B* ∈ *carrier-mat nr nc*
let *?B* = *addrow a i′ i B*
from *BB* **have** *B*: *?B* ∈ *carrier-mat nr nc* **by** *simp*
from *id*[*OF B*] **have** *id*: *eliminate-entries-rec ?B i is* = *P* ∗ *?B* **.**
have *id2*: *eliminate-entries-rec B i* (*ai′* # *is*) = *eliminate-entries-rec ?B i is*
unfolding *ai′* **by** *simp*
show *eliminate-entries-rec B i* (*ai′* # *is*) = *P* ∗ *?Add* ∗ *B*
unfolding *id2 id* **unfolding** *addrow-mat*[*OF BB Cons*(*2*)]
by (*rule assoc-mult-mat*[*symmetric*, *OF P - BB*], *auto*)
qed
qed

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*))
by *pat-completeness auto*

termination
proof −
let *?R* = *measures* [λ (*A* :: *′a* :: *field mat*,*B*,*i*,*j*). *dim-col A* − *j*,
λ (*A*,*B*,*i*,*j*). *if A* $$ (*i*,*j*) = *0 then 2 else if A* $$ (*i*,*j*) = *1 then 0 else 1*]
show *?thesis*
proof
show *wf ?R* **by** *auto*
next

fix *A B* :: *'a mat* **and** *i j nr nc a i' is*
assume $*$: *nr* = *dim-row A* *nc* = *dim-col A* *i* < *nr* ∧ *j* < *nc* *a* = *A* \$\$ (*i*, *j*) *a* = *0*
and *ne*: [*i'* . *i'* <− [*Suc i* ..< *nr*], *A* \$\$ (*i'*,*j*) ≠ *0*] = *i'* # *is*
from *ne* **have** *i'* ∈ *set* ([*i'* . *i'* <− [*Suc i* ..< *nr*], *A* \$\$ (*i'*,*j*) ≠ *0*]) **by** *auto*
with $*$
show ((*swaprows i i' A*, *swaprows i i' B*, *i*, *j*), *A*, *B*, *i*, *j*) ∈ *?R* **by** *auto*
qed *auto*
qed

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 −
let *?U* = *Units* (*ring-mat TYPE*(*'a*) *nr b*)
interpret *m*: *ring ring-mat TYPE*(*'a*) *nr b* **by** (*rule ring-mat*)
{
fix *i j* :: *nat*
assume *gauss-jordan-main A B i j* = (*A'*,*B'*)
with *A B*
have ∃ *P* ∈ *?U*. *A'* = *P* $*$ *A* ∧ *B'* = *P* $*$ *B*
proof (*induction A B i j rule*: *gauss-jordan-main.induct*)
case (*1 A B i j*)
note *A* = *1*(*5*)
hence *dim*: *dim-row A* = *nr* *dim-col A* = *nc* **by** *auto*
note *B* = *1*(*6*)
hence *dimB*: *dim-row B* = *nr* *dim-col B* = *nc'* **by** *auto*
note *IH* = *1*(*1*−*4*)[*OF dim*[*symmetric*]]
note *res* = *1*(*7*)
note *simp* = *gauss-jordan-main.simps*[*of A B i j*] *Let-def*
let *?g* = *gauss-jordan-main A B i j*
show *?case*
proof (*cases i* < *nr* ∧ *j* < *nc*)
case *False*
with *res* **have** *res*: *A'* = *A* *B'* = *B* **unfolding** *simp dim* **by** *auto*
show *?thesis* **unfolding** *res*
by (*rule bexI*[*of* - 1_m *nr*], *insert A B*, *auto simp*: *Units-def ring-mat-def*)
next
case *True* **note** *valid* = *this*
note *IH* = *IH*[*OF valid refl*]
show *?thesis*
proof (*cases A* \$\$ (*i*,*j*) = *0*)
case *False* **note** *nZ* = *this*
note *IH* = *IH*(*3*−*4*)[*OF nZ*]

show *?thesis*
proof (*cases A* $$ (*i*,*j*) = *1*)
case *False* **note** *nO* = *this*
let *?inv* = *inverse* (*A* $$ (*i*,*j*))
from *nO nZ valid res*
have *gauss-jordan-main* (*multrow i ?inv A*) (*multrow i ?inv B*) *i j* = (*A'*,*B'*)
unfolding *simp dim* **by** *simp*
note *IH* = *IH*(*2*)[*OF nO refl*, *unfolded multrow-carrier*, *OF A B this*]
from *IH* **obtain** *P* **where** *P*: *P* ∈ *?U* **and**
id: *A'* = *P* ∗ *multrow i ?inv A B'* = *P* ∗ *multrow i ?inv B* **by** *blast*
let *?Inv* = *multrow-mat nr i ?inv*
from *nZ valid* **have** *i* < *nr ?inv* ≠ *0* **by** *auto*
from *multrow-mat-Unit*[*OF this*]
have *Inv*: *?Inv* ∈ *?U* **.**
from *m.Units-m-closed*[*OF P Inv*] **have** *PI*: *P* ∗ *?Inv* ∈ *?U* **unfolding** *ring-mat-def* **by** *simp*
from *m.Units-closed*[*OF P*] **have** *P*: *P* ∈ *carrier-mat nr nr* **unfolding** *ring-mat-def* **by** *simp*
show *?thesis* **unfolding** *id* **unfolding** *multrow-mat*[*OF A*] *multrow-mat*[*OF B*]
by (*rule bexI*[*OF - PI*], *intro conjI*,
rule assoc-mult-mat[*symmetric*, *OF P - A*], *simp*,
rule assoc-mult-mat[*symmetric*, *OF P - B*], *simp*)
next
case *True* **note** *O* = *this*
let *?is* = *filter* (λ *i'*. *i'* ≠ *i*) [*0* ..< *nr*]
let *?ais* = *map* (λ *i'*. (−*A* $$ (*i'*,*j*), *i'*)) *?is*
let *?E* = λ *B*. *eliminate-entries* (λ *i*. *A* $$ (*i*,*j*)) *B i j*
let *?EE* = λ *B*. *eliminate-entries-rec B i ?ais*
let *?A* = *?E A*
let *?B* = *?E B*
let *?AA* = *?EE A*
let *?BB* = *?EE B*
from *O nZ valid res* **have** *gauss-jordan-main ?A ?B* (*Suc i*) (*Suc j*) = (*A'*,*B'*)
unfolding *simp dim* **by** *simp*
note *IH* = *IH*(*1*)[*OF O refl carrier-eliminate-entries*(*1*)[*OF A*] *carrier-eliminate-entries*(*1*)[*OF B*] *this*]
from *IH* **obtain** *P* **where** *P*: *P* ∈ *?U* **and** *id*: *A'* = *P* ∗ *?A B'* = *P* ∗ *?B* **by** *blast*
have ∗: *j* < *dim-col A i* < *dim-row A* **by** (*auto simp add*: *dim valid*)
have ∃ *P*∈*?U*. ∀ *B nc*. *B* ∈ *carrier-mat nr nc* ⟶ *?EE B* = *P* ∗ *B*
by (*rule Unit-prod-eliminate-entries*, *insert valid*, *auto*)
then obtain *Q* **where** *Q*: *Q* ∈ *?U* **and** *QQ*: ⋀ *B nc*. *B* ∈ *carrier-mat nr nc* ⟹ *?EE B* = *Q* ∗ *B* **by** *auto*
{
fix *B* :: *'a mat* **and** *nc*
assume *B*: *B* ∈ *carrier-mat nr nc*

with *dim* **have** *dim-row B* = *dim-row A* **by** *auto*
from *eliminate-entries-convert*[*OF* ∗ *this*]
have *?E B* = *?EE B* **using** *dim* **by** *simp*
also have . . . = *Q* ∗ *B* **using** *QQ*[*OF B*] **by** *simp*
finally have *?E B* = *Q* ∗ *B* **.**
} **note** *QQ* = *this*
have *id3*: *?A* = *Q* ∗ *A* **by** (*rule QQ*[*OF A*])
have *id4*: *?B* = *Q* ∗ *B* **by** (*rule QQ*[*OF B*])
from *m.Units-closed*[*OF P*] **have** *Pc*: *P* ∈ *carrier-mat nr nr* **unfolding** *ring-mat-def* **by** *simp*
from *m.Units-closed*[*OF Q*] **have** *Qc*: *Q* ∈ *carrier-mat nr nr* **unfolding** *ring-mat-def* **by** *simp*
from *m.Units-m-closed*[*OF P Q*] **have** *PQ*: *P* ∗ *Q* ∈ *?U* **unfolding** *ring-mat-def* **by** *simp*
show *?thesis* **unfolding** *id* **unfolding** *id3 id4*
by (*rule bexI*[*OF - PQ*], *rule conjI*,
rule assoc-mult-mat[*symmetric*, *OF Pc Qc A*],
rule assoc-mult-mat[*symmetric*, *OF Pc Qc B*])
qed
next
case *True* **note** *Z* = *this*
note *IH* = *IH*(*1*−*2*)[*OF Z*]
let *?is* = [*i′* . *i′* <− [*Suc i* ..< *nr*], *A* $$ (*i′*,*j*) ≠ *0*]
show *?thesis*
proof (*cases ?is*)
case *Nil*
from *Z valid res* **have** *id*: *gauss-jordan-main A B i* (*Suc j*) = (*A′*,*B′*) **unfolding** *simp dim Nil* **by** *simp*
from *IH*(*1*)[*OF Nil A B this*] **show** *?thesis* **unfolding** *id* **.**
next
case (*Cons i′ iis*)
from *Z valid res* **have** *gauss-jordan-main* (*swaprows i i′ A*) (*swaprows i i′ B*) *i j* = (*A′*,*B′*)
unfolding *simp dim Cons* **by** *simp*
from *IH*(*2*)[*OF Cons*, *unfolded swaprows-carrier*, *OF A B this*]
obtain *P* **where** *P*: *P* ∈ *?U* **and**
id: *A′* = *P* ∗ *swaprows i i′ A B′* = *P* ∗ *swaprows i i′ B* **by** *blast*
let *?Swap* = *swaprows-mat nr i i′*
from *Cons* **have** *i′* ∈ *set ?is* **by** *auto*
with *valid* **have** *i′*: *i* < *nr i′* < *nr* **by** *auto*
from *swaprows-mat-Unit*[*OF this*] **have** *Swap*: *?Swap* ∈ *?U* **.**
from *m.Units-m-closed*[*OF P Swap*] **have** *PI*: *P* ∗ *?Swap* ∈ *?U* **unfolding** *ring-mat-def* **by** *simp*
from *m.Units-closed*[*OF P*] **have** *P*: *P* ∈ *carrier-mat nr nr* **unfolding** *ring-mat-def* **by** *simp*
show *?thesis* **unfolding** *id swaprows-mat*[*OF A i′*] *swaprows-mat*[*OF B i′*]
by (*rule bexI*[*OF - PI*], *rule conjI*,
rule assoc-mult-mat[*symmetric*, *OF P - A*], *simp*,

rule assoc-mult-mat[*symmetric*, *OF P - B*], *simp*)
qed
qed
qed
qed
}
from *this*[*of 0 0*, *folded gauss-jordan-def*, *OF res*] **show** *?thesis* **.**
qed

lemma *gauss-jordan-carrier*: **assumes** *A*: $(A :: 'a :: field\ mat) \in carrier\text{-}mat\ nr\ nc$
and *B*: $B \in carrier\text{-}mat\ nr\ nc'$
and *res*: $gauss\text{-}jordan\ A\ B = (A',B')$
shows $A' \in carrier\text{-}mat\ nr\ nc\ B' \in carrier\text{-}mat\ nr\ nc'$
proof −
from *gauss-jordan-transform*[*OF A B res*, *of undefined*]
obtain *P* **where** *P*: $P \in Units\ (ring\text{-}mat\ TYPE('a)\ nr\ undefined)$
and *id*: $A' = P * A\ B' = P * B$ **by** *auto*
from *P* **have** *P*: $P \in carrier\text{-}mat\ nr\ nr$ **unfolding** *Units-def ring-mat-def* **by** *auto*
show $A' \in carrier\text{-}mat\ nr\ nc\ B' \in carrier\text{-}mat\ nr\ nc'$ **unfolding** *id*
using *P A B* **by** *auto*
qed

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

lemma *pivot-funI*: **assumes** *d*: $dim\text{-}row\ A = nr$
and $*$: $\bigwedge i.\ i < nr \Longrightarrow f\ i \leq nc$
$\bigwedge i\ j.\ i < nr \Longrightarrow j < f\ i \Longrightarrow A\ \$\$\ (i,j) = 0$
$\bigwedge i.\ i < nr \Longrightarrow Suc\ i < nr \Longrightarrow f\ (Suc\ i) > f\ i \vee f\ (Suc\ i) = nc$
$\bigwedge i.\ i < nr \Longrightarrow f\ i < nc \Longrightarrow A\ \$\$\ (i,\ f\ i) = 1$
$\bigwedge i\ i'.\ i < nr \Longrightarrow f\ i < nc \Longrightarrow i' < nr \Longrightarrow i' \neq i \Longrightarrow A\ \$\$\ (i',f\ i) = 0$
shows *pivot-fun A f nc*
unfolding *pivot-fun-def Let-def d* **using** $*$ **by** *blast*

lemma *pivot-funD*: **assumes** *d*: $dim\text{-}row\ A = nr$
and *p*: *pivot-fun A f nc*
shows $\bigwedge i.\ i < nr \Longrightarrow f\ i \leq nc$
$\bigwedge i\ j.\ i < nr \Longrightarrow j < f\ i \Longrightarrow A\ \$\$\ (i,j) = 0$
$\bigwedge i.\ i < nr \Longrightarrow Suc\ i < nr \Longrightarrow f\ (Suc\ i) > f\ i \vee f\ (Suc\ i) = nc$
$\bigwedge i.\ i < nr \Longrightarrow f\ i < nc \Longrightarrow A\ \$\$\ (i,\ f\ i) = 1$
$\bigwedge i\ i'.\ i < nr \Longrightarrow f\ i < nc \Longrightarrow i' < nr \Longrightarrow i' \neq i \Longrightarrow A\ \$\$\ (i',f\ i) = 0$
using *p* **unfolding** *pivot-fun-def Let-def d* **by** *blast+*

```
lemma pivot-fun-multrow: assumes p: pivot-fun A f jj
  and d: dim-row A = nr dim-col A = nc
  and fi: f i0 = jj
  and jj: jj ≤ nc
  shows pivot-fun (multrow i0 a A) f jj
proof −
  note p = pivot-funD[OF d(1) p]
  let ?A = multrow i0 a A
  have dim-row ?A = nr using d by simp
  thus ?thesis
  proof (rule pivot-funI)
    fix i
    assume i: i < nr
    note p = p[OF i]
    show f i ≤ jj by fact
    show Suc i < nr ⟹ f i < f (Suc i) ∨ f (Suc i) = jj by fact
    {
      fix i'
      assume ∗: f i < jj i' < nr i' ≠ i
      from p(5)[OF this]
      show ?A $$ (i', f i) = 0
        by (subst index-mat-multrow(1), insert ∗ d jj, auto)
    }
    {
      assume ∗: f i < jj
      from p(4)[OF this] have A: A $$ (i, f i) = 1 by auto
      show ?A $$ (i, f i) = 1
        by (subst index-mat-multrow(1), insert ∗ d i A jj fi, auto)
    }
    {
      fix j
      assume j: j < f i
      from p(2)[OF j]
      show ?A $$ (i, j) = 0
        by (subst index-mat-multrow(1), insert j d i p jj fi, auto)
    }
  qed
qed

lemma pivot-fun-swaprows: assumes p: pivot-fun A f jj
  and d: dim-row A = nr dim-col A = nc
  and flk: f l = jj f k = jj
  and nr: l < nr k < nr
  and jj: jj ≤ nc
  shows pivot-fun (swaprows l k A) f jj
proof −
  note pivot = pivot-funD[OF d(1) p]
  let ?A = swaprows l k A
```

```
  have dim-row ?A = nr using d by simp
  thus ?thesis
  proof (rule pivot-funI)
    fix i
    assume i: i < nr
    note p = pivot[OF i]
    show f i ≤ jj by fact
    show Suc i < nr ⟹ f i < f (Suc i) ∨ f (Suc i) = jj by fact
    {
      fix i'
      assume *: f i < jj i' < nr i' ≠ i
      from *(1) flk have diff: l ≠ i k ≠ i by auto
      from p(5)[OF *] p(5)[OF *(1) nr(1) diff(1)] p(5)[OF *(1) nr(2) diff(2)]
      show ?A $$ (i', f i) = 0
        by (subst index-mat-swaprows(1), insert * d jj, auto)
    }
    {
      assume *: f i < jj
      from p(4)[OF this] have A: A $$ (i, f i) = 1 by auto
      show ?A $$ (i, f i) = 1
        by (subst index-mat-swaprows(1), insert * d i A jj flk, auto)
    }
    {
      fix j
      assume j: j < f i
      with p(1) flk have le: j < f l j < f k by auto
      from p(2)[OF j] pivot(2)[OF nr(1) le(1)] pivot(2)[OF nr(2) le(2)]
      show ?A $$ (i, j) = 0
        by (subst index-mat-swaprows(1), insert j d i p jj, auto)
    }
  qed
qed

lemma pivot-fun-eliminate-entries: assumes p: pivot-fun A f jj
  and d: dim-row A = nr dim-col A = nc
  and fl: f l = jj
  and nr: l < nr
  and jj: jj ≤ nc
shows pivot-fun (eliminate-entries vs A l j) f jj
proof −
  note pD = pivot-funD[OF d(1) p]
  {
    fix i j
    assume *: i < nr j < f i
    from pD(1)[OF this(1)] this(2) jj have j: j < nc by auto
    from pD nr fl * j have A $$ (l, j) = 0 by (meson less-le-trans)
    note j this
  } note hint = this
  show ?thesis by (rule pivot-funI, insert fl nr jj pD, auto simp: eliminate-entries-gen-def
```

d hint)
qed

definition *row-echelon-form* :: $'a$ :: $\{zero,one\}$ *mat* $\Rightarrow$ *bool* **where**
row-echelon-form $A \equiv \exists\ f.$ *pivot-fun* $A\ f$ (*dim-col* A)

lemma *pivot-fun-init*: *pivot-fun* A (λ -. *0*) *0*
by (*rule pivot-funI*, *auto*)

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 −
fix *b*
interpret *m*: *ring ring-mat TYPE*($'a$) *nr b* **by** (*rule ring-mat*)
show *?thesis*
using *assms*
proof (*induct A B i j arbitrary*: *f rule*: *gauss-jordan-main.induct*)
case (*1 A B i j f*)
note $A = 1(5)$
hence *dim*: *dim-row* $A = nr$ *dim-col* $A = nc$ **by** *auto*
note $res = 1(6)$
note $pivot = 1(7)$
note $f = 1(8-9)$
note $ij = 1(10-11)$
note $IH = 1(1-4)$[*OF dim*[*symmetric*]]
note *simp* = *gauss-jordan-main.simps*[*of A B i j*] *Let-def*
let *?g* = *gauss-jordan-main A B i j*
show *?case*
proof (*cases* $i < nr \wedge j < nc$)
case *False* **note** *nij* = *this*
with *res* **have** *id*: $A' = A$ **unfolding** *simp dim* **by** *auto*
have *pivot-fun A f nc*
proof (*cases* $j \geq nc$)
case *True*
with *ij* **have** *j*: $j = nc$ **by** *auto*
with *pivot* **show** *pivot-fun A f nc* **by** *simp*
next
case *False*
hence *j*: $j < nc$ **by** *simp*
from *False nij ij* **have** *i*: $i = nr$ **by** *auto*
note $f = f$[*unfolded i*]
note p = *pivot-funD*[*OF dim*(*1*) *pivot*]
show *?thesis*

proof (*rule pivot-funI*[*OF dim*(*1*)])
fix *i*
assume *i*: $i < nr$
note $p = p[OF\ i]$
from *p*(*1*) *j* **show** $f\ i \leq nc$ **by** *simp*
from *f*(*1*)[*OF i*] **have** *fij*: $f\ i < j$ **.**
from *p*(*4*)[*OF fij*] **show** *A* $$ (*i*, *f i*) = *1* **.**
from *p*(*5*)[*OF fij*] **show** $\bigwedge i'.\ i' < nr \Longrightarrow i' \neq i \Longrightarrow$ *A* $$ (i', *f i*) = *0* **.**
show $\bigwedge j.\ j < f\ i \Longrightarrow$ *A* $$ (*i*, *j*) = *0* **by** (*rule p*(*2*))
assume *Suc i* < *nr*
with *p*(*3*)[*OF this*] *f*
show *f i* < *f* (*Suc i*) ∨ *f* (*Suc i*) = *nc* **by** *auto*
qed
qed
thus *?thesis* **using** *pivot* **unfolding** *id row-echelon-form-def dim* **by** *blast*
next
case *True* **note** *valid* = *this*
hence *sij*: *Suc i* ≤ *nr Suc j* ≤ *nc* **by** *auto*
note *IH* = *IH*[*OF valid refl*]
show *?thesis*
proof (*cases A* $$ (*i*,*j*) = *0*)
case *False* **note** *nZ* = *this*
note *IH* = *IH*(*3*−*4*)[*OF nZ*]
show *?thesis*
proof (*cases A* $$ (*i*,*j*) = *1*)
case *False* **note** *nO* = *this*
let *?inv* = *inverse* (*A* $$ (*i*,*j*))
let *?A* = *multrow i ?inv A*
from *nO nZ valid res* **have** *id*: *gauss-jordan-main* (*multrow i ?inv A*) (*multrow i ?inv B*) *i j* = (A', B')
unfolding *simp dim* **by** *simp*
have *pivot-fun ?A f j*
by (*rule pivot-fun-multrow*[*OF pivot dim f*(*2*) *ij*(*2*)], *auto*)
note *IH* = *IH*(*2*)[*OF nO refl*, *unfolded multrow-carrier*, *OF A id this f ij*]
show *?thesis* **unfolding** *id* **using** *IH* **.**
next
case *True* **note** *O* = *this*
let *?E* = λ *B*. *eliminate-entries* (λ *i*. *A* $$ (*i*,*j*)) *B i j*
let *?A* = *?E A*
let *?B* = *?E B*
define *E* **where** *E* = *?A*
let *?f* = λ i'. *if* $i' = i$ *then j else if* $i' > i$ *then Suc j else f* i'
have *pivot*: *pivot-fun E f j* **unfolding** *E-def*
by (*rule pivot-fun-eliminate-entries*[*OF pivot dim f*(*2*)], *insert valid*, *auto*)
{
fix i'
assume i': $i' < nr$
have *E* $$ (i', *j*) = (*if* $i' = i$ *then 1 else 0*)

unfolding *E-def eliminate-entries-gen-def* **using** *dim O i′ valid* **by** *auto*
} **note** *Ej = this*
have *E*: *E* ∈ *carrier-mat nr nc* **unfolding** *E-def* **by** (*rule carrier-eliminate-entries*[*OF A*])
hence *dimE*: *dim-row E = nr dim-col E = nc* **by** *auto*
note *pivot = pivot-funD*[*OF dimE*(*1*) *pivot*]
have *pivot-fun E ?f* (*Suc j*)
proof (*rule pivot-funI*[*OF dimE*(*1*)])
fix *ii*
assume *ii*: *ii < nr*
note *p = pivot*[*OF ii*]
show *?f ii ≤ Suc j* **using** *p*(*1*) **by** *simp*
{
fix *jj*
assume *jj*: *jj < ?f ii*
show *E* $$ (*ii*,*jj*) = *0*
proof (*cases ii < i*)
case *True*
with *jj* **have** *jj < f ii* **by** *auto*
from *p*(*2*)[*OF this*] **show** *?thesis* **.**
next
case *False* **note** *ge = this*
with *f* **have** *fiij*: *f ii = j* **by** *simp*
show *?thesis*
proof (*cases i < ii*)
case *True*
with *jj* **have** *jj*: *jj ≤ j* **by** *auto*
show *?thesis*
proof (*cases jj < j*)
case *True*
with *p*(*2*)[*of jj*] *fiij* **show** *?thesis* **by** *auto*
next
case *False*
with *jj* **have** *jj*: *jj = j* **by** *auto*
with *Ej*[*OF ii*] *True* **show** *?thesis* **by** *auto*
qed
next
case *False*
with *ge* **have** *ii*: *ii = i* **by** *simp*
with *jj* **have** *jj*: *jj < j* **by** *simp*
from *p*(*2*)[*of jj*] *ii jj fiij* **show** *?thesis* **by** *auto*
qed
qed
}
{
assume *Suc ii < nr*
from *p*(*3*)[*OF this*] *f*
show *?f* (*Suc ii*) > *?f ii* ∨ *?f* (*Suc ii*) = *Suc j* **by** *auto*

}
{
assume *fii*: *?f ii* < *Suc j*
show *E* $$ (*ii*, *?f ii*) = *1*
proof (*cases ii* = *i*)
case *True*
with *Ej*[*of i*] *valid* **show** *?thesis* **by** *auto*
next
case *False*
with *fii* **have** *ii*: *ii* < *i* **by** (*auto split*: *if-splits*)
from *f*(*1*)[*OF this*] **have** *f ii* < *j* **by** *auto*
from *p*(*4*)[*OF this*] *ii* **show** *?thesis* **by** *simp*
qed
}
{
fix *i′*
assume ∗: *?f ii* < *Suc j i′* < *nr i′* ≠ *ii*
show *E* $$ (*i′*, *?f ii*) = *0*
proof (*cases ii* = *i*)
case *False*
with ∗(*1*) **have** *iii*: *ii* < *i* **by** (*auto split*: *if-splits*)
from *f*(*1*)[*OF this*] **have** *f ii* < *j* **by** *auto*
from *p*(*5*)[*OF this* ∗(*2*−*3*)] **show** *?thesis* **using** *iii* **by** *simp*
next
case *True*
with ∗(*2*−*3*) *Ej*[*of i′*] **show** *?thesis* **by** *auto*
qed
}
qed
note *IH* = *IH*(*1*)[*OF O refl*, *folded E-def*, *OF E - this - - sij*]
from *O nZ valid res* **have** *gauss-jordan-main E ?B* (*Suc i*) (*Suc j*) = (*A′*,
B′)
unfolding *E-def simp dim* **by** *simp*
note *IH* = *IH*[*OF this*]
show *?thesis*
proof (*rule IH*)
fix *i′*
assume *i′* < *Suc i*
thus *?f i′* < *Suc j* **using** *f*[*of i′*] **by** (*cases i′* < *i*, *auto*)
qed *auto*
qed
next
case *True* **note** *Z* = *this*
note *IH* = *IH*(*1*−*2*)[*OF Z*]
let *?is* = [*i′* . *i′* <− [*Suc i* ..< *nr*], *A* $$ (*i′*,*j*) ≠ *0*]
show *?thesis*
proof (*cases ?is*)
case *Nil*
{

fix i'
assume $i \leq i'$ **and** $i' < nr$
hence $i' = i \vee i' \in \{Suc\ i\ ..< nr\}$ **by** *auto*
from *this arg-cong*[*OF Nil*, *of set*] *Z* **have** $A\ \$\$\ (i',j) = 0$ **by** *auto*
} **note** *zero* = *this*
let *?f* = $\lambda\ i'$. *if* $i' < i$ *then* $f\ i'$ *else* $Suc\ j$
note p = *pivot-funD*[*OF dim*(*1*) *pivot*]
have *pivot-fun A ?f* (*Suc j*)
proof (*rule pivot-funI*[*OF dim*(*1*)])
fix *ii*
assume *ii*: $ii < nr$
note $p = p$[*OF this*]
show *?f ii* $\leq$ *Suc j* **using** p(*1*) **by** *simp*
{
fix *jj*
assume *jj*: $jj <$ *?f ii*
show $A\ \$\$\ (ii,jj) = 0$
proof (*cases* $ii < i$)
case *True*
with *jj* **have** $jj < f\ ii$ **by** *auto*
from p(*2*)[*OF this*] **show** *?thesis* **.**
next
case *False*
with *jj* **have** *ii'*: $ii \geq i$ **and** *jjj*: $jj \leq j$ **by** *auto*
from *zero*[*OF ii' ii*] **have** *Az*: $A\ \$\$\ (ii,j) = 0$ **.**
show *?thesis*
proof (*cases* $jj < j$)
case *False*
with *jjj* **have** $jj = j$ **by** *auto*
with *Az* **show** *?thesis* **by** *simp*
next
case *True*
show *?thesis*
by (*rule* p(*2*), *insert True False f*, *auto*)
qed
qed
}
{
assume *sii*: *Suc ii* $< nr$
show *?f ii* $<$ *?f* (*Suc ii*) $\vee$ *?f* (*Suc ii*) $=$ *Suc j*
using p(*3*)[*OF sii*] f **by** *auto*
}
{
assume *fii*: *?f ii* $<$ *Suc j*
thus $A\ \$\$\ (ii,$ *?f ii*$) = 1$
using p(*4*) f **by** (*cases* $ii < i$, *auto*)
fix i'
assume $i' < nr\ i' \neq ii$
from p(*5*)[*OF - this*] *f fii*

show *A* $$ (*i′*, *?f ii*) = *0*
by (*cases ii* < *i*, *auto*)
}
qed
note *IH* = *IH*(*1*)[*OF Nil A* - *this* - - *ij*(*1*) *sij*(*2*)]
from *Z valid res* **have** *gauss-jordan-main A B i* (*Suc j*) = (*A′*,*B′*) **unfolding** *simp dim Nil* **by** *simp*
note *IH* = *IH*[*OF this*]
show *?thesis*
by (*rule IH*, *insert f*, *force*+)
next
case (*Cons i′ iis*)
from *arg-cong*[*OF this*, *of set*] **have** *i′*: *i′* ≥ *Suc i i′* < *nr* **by** *auto*
from *f*[*of i*] *f*[*of i′*] *i′* **have** *fij*: *f i* = *j f i′* = *j* **by** *auto*
let *?A* = *swaprows i i′ A*
let *?B* = *swaprows i i′ B*
have *pivot-fun ?A f j*
by (*rule pivot-fun-swaprows*[*OF pivot dim fij*], *insert i′ ij*, *auto*)
note *IH* = *IH*(*2*)[*OF Cons*, *unfolded swaprows-carrier*, *OF A* - *this f ij*]
from *Z valid res* **have** *id*: *gauss-jordan-main ?A ?B i j* = (*A′*, *B′*) **unfolding** *simp dim Cons* **by** *simp*
note *IH* = *IH*[*OF this*]
show *?thesis* **using** *IH* **.**
qed
qed
qed
qed
qed

lemma *gauss-jordan-row-echelon*:
assumes *A*: *A* ∈ *carrier-mat nr nc*
and *res*: *gauss-jordan A B* = (*A′*, *B′*)
shows *row-echelon-form A′*
by (*rule gauss-jordan-main-row-echelon*[*OF A res*[*unfolded gauss-jordan-def*] *pivot-fun-init*], *auto*)

lemma *pivot-bound*: **assumes** *dim*: *dim-row A* = *nr*
and *pivot*: *pivot-fun A f n*
shows $i + j < nr \Longrightarrow f\ (i + j) = n \vee f\ (i + j) \geq j + f\ i$
proof (*induct j*)
case (*Suc j*)
hence *IH*: $f\ (i + j) = n \vee j + f\ i \leq f\ (i + j)$
and *lt*: $i + j < nr$ *Suc* $(i + j) < nr$ **by** *auto*
note *p* = *pivot-funD*[*OF dim pivot*]
from *p*(*3*)[*OF lt*] *IH p*(*1*)[*OF lt*(*2*)] **show** *?case* **by** *auto*
qed *simp*

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 []) **by** *pat-completeness auto*

termination by (*relation measures* [(λ (*i*,*j*). *Suc nr* − *i*), (λ (*i*,*j*). *Suc nc* − *j*)], *auto*)

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

context
fixes *A* :: *'a* :: *semiring-1 mat*
and *nr nc* :: *nat*
begin

abbreviation *pivot-positions-main* ≡ *pivot-positions-main-gen* (*0* :: *'a*) *A nr nc*

lemma *pivot-positions-main*: **assumes** *A*: *A* ∈ *carrier-mat nr nc*
and *pivot*: *pivot-fun A f nc*
shows *j* ≤ *f i* ∨ *i* ≥ *nr* ⟹
set (*pivot-positions-main i j*) = {(*i'*, *f i'*) | *i'*. *i* ≤ *i'* ∧ *i'* < *nr*} − *UNIV* × {*nc*}
∧ *distinct* (*map snd* (*pivot-positions-main i j*))
∧ *distinct* (*map fst* (*pivot-positions-main i j*))
proof (*induct i j rule*: *pivot-positions-main-gen.induct*[*of nr nc A 0*])
case (*1 i j*)
let *?a* = *A* $$ (*i*,*j*)
let *?pivot* = λ *i j*. *pivot-positions-main i j*
let *?set* = λ *i*. {(*i'*,*f i'*) | *i'*. *i* ≤ *i'* ∧ *i'* < *nr*}
let *?s* = *?set i*
let *?set* = λ *i*. {(*i'*,*f i'*) | *i'*. *i* ≤ *i'* ∧ *i'* < *nr*}
let *?s* = *?set i*
let *?p* = *?pivot i j*
from *A* **have** *dA*: *dim-row A* = *nr* **by** *simp*
note [*simp*] = *pivot-positions-main-gen.simps*[*of 0 A nr nc i j*]
show *?case*
proof (*cases i* < *nr*)
case *True* **note** *i* = *this*
note *IH* = *1*(*1*−*2*)[*OF True*]
have *jfi*: *j* ≤ *f i* **using** *1*(*3*) *i* **by** *auto*

note *pivotB* = *pivot-bound*[*OF dA pivot*]
note *pivot′* = *pivot-funD*[*OF dA pivot*]
note *pivot* = *pivot′*[*OF True*]
have *id1*: [*i* ..< *nr*] = *i* # [*Suc i* ..< *nr*] **using** *i* **by** (*rule upt-conv-Cons*)
show *?thesis*
proof (*cases j* < *nc*)
case *True* **note** *j* = *this*
note *IH* = *IH*(*1*−*2*)[*OF True*]
show *?thesis*
proof (*cases ?a* = *0*)
case *True* **note** *a* = *this*
from *i j a* **have** *p*: *?p* = *?pivot i* (*Suc j*) **by** *simp*
{
assume *f i* = *j*
with *pivot*(*4*) *j* **have** *?a* = *1* **by** *simp*
with *a* **have** *False* **by** *simp*
}
with *jfi* **have** *Suc j* ≤ *f i* ∨ *i* ≥ *nr* **by** *fastforce*
note *IH* = *IH*(*1*)[*OF True this*]
thus *?thesis* **unfolding** *p* **.**
next
case *False* **note** *a* = *this*
from *i j a* **have** *p*: *?p* = (*i*,*j*) # *?pivot* (*Suc i*) (*Suc j*) **by** *simp*
from *pivot*(*2*)[*of j*] *jfi a* **have** *jfi*: *j* = *f i* **by** *force*
from *pivotB*[*of i Suc 0*] *jfi* **have** *Suc j* ≤ *f* (*Suc i*) ∨ *nr* ≤ *Suc i*
using *Suc-le-eq j leI* **by** *auto*
note *IH* = *IH*(*2*)[*OF False this*]
{
fix *i′*
assume ∗: *f i* = *f i′ Suc i* ≤ *i′ i′* < *nr*
hence *i* + (*i′* − *i*) = *i′* **by** *auto*
from *pivotB*[*of i i′* − *i*, *unfolded this*] ∗ *jfi j* **have** *False* **by** *auto*
} **note** *distinct* = *this*
have *id2*: *?s* = *insert* (*i*,*j*) (*?set* (*Suc i*)) **using** *i jfi not-less-eq-eq*
by *fastforce*
show *?thesis* **using** *IH j jfi i* **unfolding** *p id1 id2* **by** (*auto intro*: *distinct*)
qed
next
case *False* **note** *j* = *this*
from *pivot*(*1*) *j jfi* **have** ∗: *f i* = *nc nc* = *j* **by** *auto*
from *i j* **have** *p*: *?p* = [] **by** *simp*
from *pivotB*[*of i Suc 0*] ∗ **have** *j* ≤ *f* (*Suc i*) ∨ *nr* ≤ *Suc i* **by** *auto*
{
fix *i′*
assume ∗∗: *i* ≤ *i′ i′* < *nr*
hence *i* + (*i′* − *i*) = *i′* **by** *auto*
from *pivotB*[*of i i′* − *i*, *unfolded this*] ∗∗ ∗ **have** *nc* ≤ *f i′* **by** *auto*
with *pivot′*(*1*)[*OF* ‹*i′* < *nr*›] **have** *f i′* = *nc* **by** *auto*
}

thus *?thesis* **using** *IH* **unfolding** *p id1* **by** *auto*
qed
qed *auto*
qed
end

lemma *pivot-fun-zero-row-iff*: **assumes** *pivot*: *pivot-fun* (A :: $'a$ :: *semiring-1 mat*) *f nc*
and *A*: $A \in$ *carrier-mat nr nc*
and *i*: $i < nr$
shows $f\ i = nc \longleftrightarrow row\ A\ i = 0_v\ nc$
proof −
from *A* **have** *dim*: *dim-row* $A = nr$ **by** *auto*
note *pivot* = *pivot-funD*[*OF dim pivot i*]
{
assume $f\ i = nc$
from *pivot*(*2*)[*unfolded this*]
have $row\ A\ i = 0_v\ nc$
by (*intro eq-vecI*, *insert A*, *auto simp*: *row-def*)
}
moreover
{
assume *row*: $row\ A\ i = 0_v\ nc$
assume $f\ i \neq nc$
with *pivot*(*1*) **have** $f\ i < nc$ **by** *auto*
with *pivot*(*4*)[*OF this*] *i A arg-cong*[*OF row*, *of* λ *v*. *v* \$ *f i*] **have** *False* **by** *auto*
}
ultimately show *?thesis* **by** *auto*
qed

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

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

lemmas *pivot-positions-def* = *pivot-positions-gen-def*

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

let *?pp* = *pivot-positions A*
show *id*: *set ?pp* = {(*i*, *f i*) | *i*. *i* < *nr* ∧ *f i* ≠ *nc*}
and *dist*: *distinct* (*map fst ?pp*)
and *distinct* (*map snd ?pp*)
using *pivot-positions-main*[*OF A pivot*, *of 0 0*] *A*
unfolding *pivot-positions-def* **by** *auto*
have *length ?pp* = *length* (*map fst ?pp*) **by** *simp*
also have ... = *card* (*fst ‘ set ?pp*) **using** *distinct-card*[*OF dist*] **by** *simp*
also have *fst ‘ set ?pp* = { *i*. *i* < *nr* ∧ *f i* ≠ *nc*} **unfolding** *id* **by** *force*
also have ... = { *i*. *i* < *nr* ∧ *row A i* ≠ 0_v *nc*}
using *pivot-fun-zero-row-iff*[*OF pivot A*] **by** *auto*
finally show *length ?pp* = *card* {*i*. *i* < *nr* ∧ *row A i* ≠ 0_v *nc*} **.**
qed

context
fixes *uminus* :: ′*a* ⇒ ′*a*
and *zero* :: ′*a*
and *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* $\Rightarrow$ *nat*
assumes *ref*: *row-echelon-form A*
and A: $A \in$ *carrier-mat nr nc*
begin

lemma *non-pivot-base*:
defines *pp*: *pp* $\equiv$ *pivot-positions A*
assumes *qj*: $qj < nc$ $qj \notin$ *snd* ‘ *set pp*
shows *non-pivot-base A pp qj* $\in$ *carrier-vec nc*
non-pivot-base A pp qj \$ $qj = 1$
$A *_v$ *non-pivot-base A pp qj* $= 0_v$ *nr*
$\bigwedge$ qj'. $qj' < nc \Longrightarrow qj' \notin$ *snd* ‘ *set pp* $\Longrightarrow qj \neq qj' \Longrightarrow$ *non-pivot-base A pp qj* \$ $qj' = 0$
proof $-$
from A **have** *dim*: *dim-row A* = *nr* *dim-col A* = *nc* **by** *auto*
from *ref*[*unfolded row-echelon-form-def*] **obtain** *p*
where *pivot*: *pivot-fun A p nc* **using** *dim* **by** *auto*
note *pivot'* = *pivot-funD*[*OF dim*(*1*) *pivot*]
note *pp* = *pivot-positions*[*OF A pivot*, *folded pp*]
let *?p* = λ *i*. $i < nr \wedge p\ i = nc \vee i = nr$
let *?spp* = *map prod.swap pp*
let *?map* = *map-of ?spp*
define *I* **where** $I = (\lambda$ *i*. *case map-of* (*map prod.swap pp*) *i of Some j* $\Rightarrow -$ *A* \$\$ (*j*,*qj*) | *None* $\Rightarrow$ *0*)
have *d*: *non-pivot-base A pp qj* = *vec nc* (λ *i*. *if i* = *qj then 1 else I i*)
unfolding *non-pivot-base-def Let-def dim I-def* **..**
from *pp* **have** *dist*: *distinct* (*map fst ?spp*)
unfolding *map-map o-def prod.swap-def* **by** *auto*
let *?r* = *set* (*map snd pp*)
have *r*: *?r* = *p* ‘ {*0* ..< *nr*} $-$ {*nc*} **unfolding** *set-map pp* **by** *force*
let *?l* = *set* (*map fst pp*)
from *qj* **have** *qj'*: $qj \notin p$ ‘ {*0* ..< *nr*} **using** *r* **by** *auto*
let *?v* = *non-pivot-base A pp qj*
let *?P* = *p* ‘ {*0* ..< *nr*}
have *dimv*: *dim-vec ?v* = *nc* **unfolding** *d* **by** *simp*
thus *?v* $\in$ *carrier-vec nc* **unfolding** *carrier-vec-def* **by** *auto*
show *vqj*: *?v* \$ *qj* = *1* **unfolding** *d* **using** *qj* **by** *auto*
{
fix *qj'*
assume $*$: $qj' < nc$ $qj \neq qj'$ $qj' \notin$ *snd* ‘ *set pp*
hence *?map qj'* = *None* **unfolding** *map-of-eq-None-iff* **by** *force*
hence *I qj'* = *0* **unfolding** *I-def* **by** *simp*
with $*$ **show** *non-pivot-base A pp qj* \$ *qj'* = *0*
unfolding *d* **by** *simp*
}
{
fix *i*
assume *i*: $i < nr$

let ?I = {j. ?map j = Some i}
have row A i · ?v = 0
proof −
have id: ({0..<nc} ∩ ?P) ∪ ({0..<nc} − ?P) = {0..<nc} **by** auto
let ?e = λ j. row A i $ j ∗ ?v $ j
let ?e′ = λ j. (if ?map j = Some i then − A $$ (i, qj) else 0)
{
fix j
assume j: j < nc j ∈ ?P
then obtain ii **where** ii: ii < nr **and** jpi: j = p ii **and** pii: p ii < nc **by** auto
hence mem: (ii,j) ∈ set pp **and** (j,ii) ∈ set ?spp **by** (auto simp: pp)
from map-of-is-SomeI[OF dist this(2)]
have map: ?map j = Some ii **by** auto
from mem j qj **have** jqj: j ≠ qj **by** force
note p = pivot′(4−5)[OF ii pii]
define start **where** start = ?e j
have start = A $$ (i,j) ∗ ?v $ j **using** j i A **by** (auto simp: start-def)
also have A $$ (i,j) = A $$ (i, p ii) **unfolding** jpi **..**
also have . . . = (if i = ii then 1 else 0) **using** p(1) p(2)[OF i] **by** auto
also have . . . ∗ ?v $ j = (if i = ii then ?v $ j else 0) **by** simp
also have ?v $ j = I j **unfolding** d
using j jqj A **by** auto
also have I j = − A $$ (ii, qj) **unfolding** I-def map **by** simp
finally have ?e j = ?e′ j
unfolding start-def map **by** auto
} **note** piv = this
have row A i · ?v = (∑ j = 0..<nc. ?e j) **unfolding** row-def scalar-prod-def dimv **..**
also have . . . = sum ?e ({0..<nc} ∩ ?P) + sum ?e ({0..<nc} − ?P)
by (subst sum.union-disjoint[symmetric], auto simp: id)
also have sum ?e ({0..<nc} − ?P) = ?e qj + sum ?e ({0 ..<nc} − ?P − {qj})
by (rule sum.remove, insert qj qj′, auto)
also have ?e qj = row A i $ qj **unfolding** vqj **by** simp
also have row A i $ qj = A $$ (i, qj) **using** i A qj **by** auto
also have sum ?e ({0 ..<nc} − ?P − {qj}) = 0
proof (rule sum.neutral, intro ballI)
fix j
assume j ∈ {0 ..<nc} − ?P − {qj}
hence j: j < nc j ∉ ?P j ≠ qj j ∉ ?r **unfolding** r **by** auto
hence id: map-of ?spp j = None **unfolding** map-of-eq-None-iff **by** force
have ?v $ j = I j **unfolding** d **using** j **by** simp
also have . . . = 0 **unfolding** I-def id **by** simp
finally show row A i $ j ∗ ?v $ j = 0 **by** simp
qed
also have A $$ (i, qj) + 0 = A $$ (i, qj) **by** simp
also have sum ?e ({0..<nc} ∩ ?P) = sum ?e′ ({0..<nc} ∩ ?P)
by (rule sum.cong, insert piv, auto)

```
      also have {0..<nc} ∩ ?P = {0..<nc} ∩ ?I ∩ ?P ∪ ({0..<nc} − ?I) ∩ ?P
by auto
      also have sum ?e' ({0..<nc} ∩ ?I ∩ ?P ∪ ({0..<nc} − ?I) ∩ ?P)
        = sum ?e' ({0..<nc} ∩ ?I ∩ ?P) + sum ?e' (({0..<nc} − ?I) ∩ ?P)
        by (rule sum.union-disjoint, auto)
      also have sum ?e' (({0..<nc} − ?I) ∩ ?P) = 0
        by (rule sum.neutral, auto)
      also have sum ?e' ({0..<nc} ∩ ?I ∩ ?P) =
        sum (λ -. − A $$ (i, qj)) ({0..<nc} ∩ ?I ∩ ?P)
        by (rule sum.cong, auto)
      also have ... + 0 = ... by simp
      also have sum (λ -. − A $$ (i, qj)) ({0..<nc} ∩ ?I ∩ ?P) + A $$ (i, qj) =
0
      proof (cases i ∈ ?l)
        case False
        with pp(1) i have p i = nc by force
        from pivot'(2)[OF i, unfolded this, OF qj(1)] have z: A $$ (i, qj) = 0 .
        show ?thesis
          by (subst sum.neutral, auto simp: z)
      next
        case True
        then obtain j where mem: (i,j) ∈ set pp and id: (j,i) ∈ set ?spp by auto
        from map-of-is-SomeI[OF dist this(2)] have map: ?map j = Some i .
        from pivot'(1)[OF i] have pi: p i ≤ nc .
        with mem[unfolded pp] have j: j = p i j < nc by auto
        {
          fix j'
          assume j' ∈ ?I
          hence ?map j' = Some i by auto
          from map-of-SomeD[OF this] have (i, j') ∈ set pp by auto
          with mem pp(2) have j' = j using map-of-is-SomeI by fastforce
        }
        with map have II: ?I = {j} by blast
        have II: {0..<nc} ∩ ?I ∩ ?P = {j} unfolding II using mem[unfolded pp]
i j by auto
        show ?thesis unfolding II by simp
      qed
      finally show row A i · ?v = 0 .
    qed
  } note main = this
  show A *v ?v = 0v nr
    by (rule eq-vecI, auto simp: dim main)
qed

lemma find-base-vector: assumes snd ‘ set (pivot-positions A) ≠ {0 ..< nc}
  shows
    find-base-vector A ∈ carrier-vec nc
    find-base-vector A ≠ 0v nc
    A *v find-base-vector A = 0v nr
```

proof −
  define *cands* **where** $cands = filter\ (\lambda\ j.\ j \notin snd\ `\ set\ (pivot\text{-}positions\ A))\ [0\ ..< nc]$
  from *A* **have** *dim*: $dim\text{-}row\ A = nr$ $dim\text{-}col\ A = nc$ **by** *auto*
  from *ref*[*unfolded row-echelon-form-def*] **obtain** *p*
  where *pivot*: *pivot-fun A p nc* **using** *dim* **by** *auto*
  note $piv = pivot\text{-}funD[OF\ dim(1)\ pivot]$
  have $set\ cands \neq \{\}$ **using** *assms piv* **unfolding** *cands-def pivot-positions*[*OF A pivot*]
    by (*auto simp*: *le-neq-implies-less*)
  then obtain *c cs* **where** *cands*: $cands = c \# cs$ **by** (*cases cands*, *auto*)
  hence *res*: $find\text{-}base\text{-}vector\ A = non\text{-}pivot\text{-}base\ A\ (pivot\text{-}positions\ A)\ c$
    unfolding *find-base-vector-def Let-def cands-def dim* **by** *auto*
  from *cands* **have** $c \in set\ cands$ **by** *auto*
  hence *c*: $c < nc$ $c \notin snd\ `\ set\ (pivot\text{-}positions\ A)$
    unfolding *cands-def* **by** *auto*
  from *non-pivot-base*[*OF this*, *folded res*] *c* **show**
    $find\text{-}base\text{-}vector\ A \in carrier\text{-}vec\ nc$
    $find\text{-}base\text{-}vector\ A \neq 0_v\ nc$
    $A *_v find\text{-}base\text{-}vector\ A = 0_v\ nr$
  by *auto*
qed
end

lemma *row-echelon-form-imp-1-or-0-row*: **assumes** *A*: $A \in carrier\text{-}mat\ n\ n$
  and *row*: *row-echelon-form A*
  shows $A = 1_m\ n \vee (n > 0 \wedge row\ A\ (n - 1) = 0_v\ n)$
proof −
  from *A* **have** *dim*: $dim\text{-}row\ A = n$ $dim\text{-}col\ A = n$ **by** *auto*
  from *row*[*unfolded row-echelon-form-def*] *A*
  obtain *f* **where** *pivot*: *pivot-fun A f n* **by** *auto*
  note $p = pivot\text{-}funD[OF\ dim(1)\ this]$
  show *?thesis*
  proof (*cases* $\exists\ i < n.\ f\ i \neq i$)
    case *True*
    then obtain *i* **where** *i*: $i < n$ **and** *fi*: $f\ i \neq i$ **by** *auto*
    note $pb = pivot\text{-}bound[OF\ dim(1)\ pivot]$
    from *pb*[*of 0 i*] *i* **have** $f\ i = n \vee i \leq f\ i$ **by** *auto*
    with *fi* **have** *fi*: $f\ i = n \vee i < f\ i$ **by** *auto*
    from *i* **have** *n*: $n - 1 = i + (n - i - 1)$ **by** *auto*
    from *pb*[*of i* $n - i - 1$, *folded n*] *fi i* *p*(*1*)[*of* $n - 1$]
    have *fn*: $f\ (n - 1) = n$ **by** *auto*
    from *i* **have** *n0*: $n > 0$ **and** *n1*: $n - 1 < n$ **by** *auto*
    from *p*(*2*)[*OF n1*, *unfolded fn*] **have** *zero*: $\bigwedge\ j.\ j < n \Longrightarrow A\ \$\$\ (n - 1, j) = 0$ **by** *auto*
    show *?thesis*
      by (*rule disjI2*[*OF conjI*[*OF n0*]], *rule eq-vecI*, *insert zero A*, *auto*)
  next
    case *False*

{
fix j
assume j: $j < n$
with *False* **have** id: $f\ j = j$ **by** *auto*
note $pj = p[OF\ j,\ unfolded\ id]$
from $pj(5)[OF\ j]\ pj(4)[OF\ j]$
have $\bigwedge\ i.\ i < n \Longrightarrow A\ \$\$\ (i,j) = (if\ i = j\ then\ 1\ else\ 0)$ **by** *auto*
} **note** $id = this$
show *?thesis*
by (*rule disjI1*, *rule eq-matI*, *subst id*, *insert A*, *auto*)
qed
qed

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

lemma *find-base-vector-not-1-pivot-positions*: $snd\ `\ set\ (pivot\text{-}positions\ A) \neq \{0\ ..< n\}$
proof
let $?pp = pivot\text{-}positions\ A$
assume id: $snd\ `\ set\ ?pp = \{0\ ..< n\}$
from A **have** dim: $dim\text{-}row\ A = n\ dim\text{-}col\ A = n$ **by** *auto*
let $?n = n - 1$
from $row\text{-}echelon\text{-}form\text{-}imp\text{-}1\text{-}or\text{-}0\text{-}row[OF\ A\ ref]\ 1$
have $*$: $0 < n$ **and** *row*: $row\ A\ ?n = 0_v\ n$ **by** *auto*
from $ref[unfolded\ row\text{-}echelon\text{-}form\text{-}def]$ **obtain** p
where *pivot*: $pivot\text{-}fun\ A\ p\ n$ **using** *dim* **by** *auto*
note $pp = pivot\text{-}positions[OF\ A\ pivot]$
note $piv = pivot\text{-}funD[OF\ dim(1)\ pivot]$
from $*$ **have** n: $?n < n$ **by** *auto*
{

assume $p\ ?n < n$
with $piv(4)[OF\ n\ this]\ row\ n\ A$ **have** *False*
by (*metis dim index-row(1) index-zero-vec(1) zero-neq-one*)
}
with $piv(1)[OF\ n]$ **have** pn: $p\ ?n = n$ **by** *fastforce*
hence $?n \notin fst\ `\ set\ ?pp$ **unfolding** *pp* **by** *auto*
hence $fst\ `\ set\ ?pp \subseteq \{0\ ..< n\} - \{?n\}$ **unfolding** *pp* **by** *force*
also have $\ldots \subseteq \{0\ ..< n - 1\}$ **by** *auto*
finally have $card\ (fst\ `\ set\ ?pp) \leq card\ \{0\ ..< n - 1\}$ **using** *card-mono* **by** *blast*
also have $\ldots = n - 1$ **by** *auto*
also have $card\ (fst\ `\ set\ ?pp) = card\ (snd\ `\ set\ ?pp)$
unfolding $set\text{-}map[symmetric]\ distinct\text{-}card[OF\ pp(2)]\ distinct\text{-}card[OF\ pp(3)]$
by *simp*

also have ... = n **unfolding** *id* **by** *simp*
finally show *False* **using** n **by** *simp*
qed

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* $A = 0_v\ n$
using *find-base-vector*[*OF ref A find-base-vector-not-1-pivot-positions*] **.**
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 −
from *gauss-jordan-transform*[*OF A B gauss*, *unfolded ring-mat-def Units-def*, *simplified*]
obtain $P\ Q$ **where** P: $P \in$ *carrier-mat nr nr* **and** Q: $Q \in$ *carrier-mat nr nr*
and *inv*: $Q * P = 1_m\ nr$
and *CPA*: $C = P * A$
and *DPB*: $D = P * B$ **by** *auto*
from *CPA P A* **show** C: $C \in$ *carrier-mat nr nc* **by** *auto*
from *DPB P B* **show** D: $D \in$ *carrier-mat nr nc2* **by** *auto*
have $A = 1_m\ nr * A$ **using** A **by** *simp*
also have ... = $Q * C$ **unfolding** *inv*[*symmetric*] *CPA* **using** $Q\ P\ A$ **by** *simp*
finally have *AQC*: $A = Q * C$ **.**
have $B = 1_m\ nr * B$ **using** B **by** *simp*
also have ... = $Q * D$ **unfolding** *inv*[*symmetric*] *DPB* **using** $Q\ P\ B$ **by** *simp*
finally have *BQD*: $B = Q * D$ **.**
{
assume x: $x \in$ *carrier-vec nc*
{
assume ?l
from *arg-cong*[*OF this*, *of* λ *v*. $P *_v v$] $P\ A\ x$ **have** ?r **unfolding** *CPA* **by** *auto*
}
moreover
{
assume ?r
from *arg-cong*[*OF this*, *of* λ *v*. $Q *_v v$] $Q\ C\ x$ **have** ?l **unfolding** *AQC* **by** *auto*
}
ultimately show ?l = ?r **by** *auto*

}
{
assume *X*: $X \in$ *carrier-mat nc nc2*
{
assume *?l2*
from *arg-cong*[*OF this*, *of* λ *X*. $P * X$] *P A X* **have** *?r2* **unfolding** *CPA DPB* **by** *simp*
}
moreover
{
assume *?r2*
from *arg-cong*[*OF this*, *of* λ *X*. $Q * X$] *Q C X* **have** *?l2* **unfolding** *AQC BQD* **by** *simp*
}
ultimately show *?l2* = *?r2* **by** *auto*
}
qed

definition *gauss-jordan-single* :: *'a* :: *field mat* $\Rightarrow$ *'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
$\exists$ *P Q*. $C = P * A \wedge P \in$ *carrier-mat nr nr* $\wedge Q \in$ *carrier-mat nr nr* $\wedge P * Q = 1_m$ *nr* $\wedge Q * P = 1_m$ *nr* (**is** *?ex*)
proof −
from *A gauss*[*unfolded gauss-jordan-single-def*] **obtain** *D* **where** *gauss*: *gauss-jordan A* (0_m *nr 0*) = (*C*,*D*)
by (*cases gauss-jordan A* (0_m *nr 0*), *auto*)
from *gauss-jordan*[*OF A zero-carrier-mat gauss*] *gauss-jordan-row-echelon*[*OF A gauss*]
gauss-jordan-transform[*OF A zero-carrier-mat gauss*, *of* ()]
show $x \in$ *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 ?ex* **unfolding** *Units-def ring-mat-def* **by** *auto*
qed

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 −
let *?R* = *ring-mat TYPE*(*'a*) *n b*

let *?U* = *Units ?R*
interpret *m*: *ring ?R* **by** (*rule ring-mat*)
from *gauss-jordan-transform*[*OF A B res*, *of b*]
obtain *P* **where** *P*: $P \in$ *?U* **and** *id*: $P * A = 1_m\ n$ **and** B': $B' = P * B$ **by** *auto*
from *P* **have** *Pc*: $P \in$ *carrier-mat n n* **unfolding** *Units-def ring-mat-def* **by** *auto*
from *m.Units-one-side-I*(*1*)[*of A P*] *A P id* **show** *Au*: $A \in$ *?U* **unfolding** *ring-mat-def* **by** *auto*
assume *B*: $B = 1_m\ n$
from B'[*unfolded this*] *Pc* **have** B': $B' = P$ **by** *auto*
show $A * B' = 1_m\ n \wedge B' * A = 1_m\ n$ **unfolding** B'
using *m.Units-inv-comm*[*OF - P Au*] *id* **by** (*auto simp*: *ring-mat-def*)
qed

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 −
let *?R* = *ring-mat TYPE*($'a$) *n b*
let *?U* = *Units ?R*
interpret *m*: *ring ?R* **by** (*rule ring-mat*)
from *AU* **have** *A*: $A \in$ *carrier-mat n n* **unfolding** *Units-def ring-mat-def* **by** *auto*
obtain A' B' **where** *res*: *gauss-jordan A B* = (A',B') **by** *force*
from *gauss-jordan-transform*[*OF A B res*, *of b*]
obtain *P* **where** *P*: $P \in$ *?U* **and** *id*: $A' = P * A$ **by** *auto*
from *m.Units-m-closed*[*OF P AU*] **have** A': $A' \in$ *?U* **unfolding** *id ring-mat-def* **by** *auto*
hence $A'c$: $A' \in$ *carrier-mat n n* **unfolding** *Units-def ring-mat-def* **by** *auto*
from A'[*unfolded Units-def ring-mat-def*] **obtain** IA' **where** IA': $IA' \in$ *carrier-mat n n*
and *IA*: $A' * IA' = 1_m\ n$ **by** *auto*
from *row-echelon-form-imp-1-or-0-row*[*OF gauss-jordan-carrier*(*1*)[*OF A B res*] *gauss-jordan-row-echelon*[*OF A res*]]
have *choice*: $A' = 1_m\ n \vee 0 < n \wedge$ *row* A' $(n - 1) = 0_v\ n$ **.**
hence $A' = 1_m\ n$
proof
let $?n = n - 1$
assume $0 < n \wedge$ *row* A' $?n = 0_v\ n$
hence *n*: $?n < n$ **and** *row*: *row* A' $?n = 0_v\ n$ **by** *auto*
have $1 = 1_m\ n$ $$ $(?n,?n)$ **using** *n* **by** *simp*
also have $1_m\ n = A' * IA'$ **unfolding** *IA* **..**
also have $(A' * IA')$ $$ $(?n, ?n)$ = *row* A' $?n \cdot$ *col* IA' $?n$
using *n* IA' $A'c$ **by** *simp*
also have *row* A' $?n = 0_v\ n$ **unfolding** *row* **..**
also have $0_v\ n \cdot$ *col* IA' $?n = 0$ **using** IA' *n* **by** *simp*
finally have $1 = (0 :: 'a)$ **by** *simp*

thus *?thesis* **by** *simp*
qed
with *res* **show** *?thesis* **by** *simp*
qed

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 −
from *gauss-jordan-inverse-one-direction*(*2*)[*OF A - res refl*, *of n*]
show $A * B' = 1_m\ n$ $B' * A = 1_m\ n$ **by** *auto*
from *gauss-jordan-carrier*(*2*)[*OF A - res*, *of n*] **show** $B' \in$ *carrier-mat n n* **by** *auto*
qed

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
assume *?l*
show *?r*
by (*rule gauss-jordan-inverse-other-direction*[*OF* ‹*?l*› *B*])
next
let *?g* = *gauss-jordan A B*
assume *?r*
then obtain B' **where** *?g* = $(1_m\ n,\ B')$ **by** (*cases ?g*, *auto*)
from *gauss-jordan-inverse-one-direction*(*1*)[*OF A B this*]
show *?l* **.**
qed

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 −
let *?one* = $1_m\ n$
obtain *BB C* **where** *res*: *gauss-jordan A ?one* = (*BB*,*C*) **by** *force*
{
assume *mat-inverse A* = *None*
with *res* **have** *BB* ≠ *?one* **unfolding** *mat-inverse-def* **using** *A* **by** *auto*

thus *A* ∉ *Units* (*ring-mat TYPE*('*a* :: *field*) *n b*)
  using *gauss-jordan-check-invertable*[*OF A*, *of ?one n*] *res* **by** *force*
}
{
  assume *mat-inverse A* = *Some B*
  with *res A* **have** *BB* = *?one C* = *B* **unfolding** *mat-inverse-def*
    by (*auto split*: *if-splits option.splits*)
  from *gauss-jordan-compute-inverse*[*OF A res*[*unfolded this*]]
  show $A * B = 1_m\ n \wedge B * A = 1_m\ n \wedge B \in$ *carrier-mat n n* **by** *auto*
}
qed
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*)
  by (*auto split*: *if-splits*)

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 (*cases l* < *dim-row-impl A* ∧ *k* < *dim-row-impl A*)
  case *True*
  hence *id*: *?r* = *mat-impl* (*mat-swaprows-impl k l A*) **by** *simp*
  show *?thesis* **unfolding** *id* **unfolding** *mat-swaprows-def*
  proof (*rule eq-matI*, *goal-cases*)
    case (*1 i j*)
    thus *?case* **using** *True*
    proof (*transfer*, *goal-cases*)

case (*1 i k l A j*)
obtain *nr nc rows* **where** *A*: *A* = (*nr*,*nc*,*rows*) **by** (*cases A*, *auto*)
from *1*[*unfolded A*]
have *nr*: *length* (*IArray.list-of rows*) = *nr*
and *nc*: *IArray.all* (*λr*. *length* (*IArray.list-of r*) = *nc*) *rows*
and *ij*: *i* < *nr j* < *nc* **and** *ij′*: (*i* < *nr* ∧ *j* < *nc*) = *True*
and *l*: *l* < *nr k* < *nr* **by** *auto*
show *?case* **unfolding** *A prod.simps fst-conv o-def snd-conv Let-def mk-mat-def ij′ if-True*
using *ij nr nc l*
by (*cases k* = *i*; *cases l* = *i*, *auto*)
qed
qed ((*transfer*, *auto*)+)
qed *auto*

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 (*auto*, *goal-cases*)
case (*1 mul k a nc b row*)
show *?case*
proof (*cases b*)
case (*IArray rows*)
with *1* **have** *row* ∈ *set rows* ∨ *k* < *length rows* ∧ *row* = *IArray* (*map* (*mul a*) (*IArray.list-of* (*rows* ! *k*)))
by (*cases k* < *length rows*, *auto simp*: *set-list-update dest*: *in-set-takeD in-set-dropD*)
with *1 IArray* **show** *?thesis* **by** (*cases*, *auto*)
qed
qed

lemma [*code*]: *mat-multrow-gen mul k a* (*mat-impl A*) = *mat-impl* (*mat-multrow-gen-impl mul k a A*)
unfolding *mat-multrow-gen-def*
proof (*rule eq-matI*, *goal-cases*)
case (*1 i j*)
thus *?case*
proof (*transfer*, *goal-cases*)
case (*1 i mul k a A j*)
obtain *nr nc rows* **where** *A*: *A* = (*nr*,*nc*,*rows*) **by** (*cases A*, *auto*)
from *1*[*unfolded A*]
have *nr*: *length* (*IArray.list-of rows*) = *nr*
and *nc*: *IArray.all* (*λr*. *length* (*IArray.list-of r*) = *nc*) *rows*
and *ij*: *i* < *nr j* < *nc* **and** *ij′*: (*i* < *nr* ∧ *j* < *nc*) = *True* **by** *auto*
have *len*: *j* < *length* (*IArray.list-of* (*IArray.list-of rows* ! *i*))

using *ij nc nr* **by** (*cases rows*, *auto*)
show *?case* **unfolding** *A prod.simps fst-conv o-def snd-conv Let-def mk-mat-def ij′ if-True*
using *ij nr nc*
by (*cases* $k = i$, *auto simp*: *len*)
qed
qed ((*transfer*, *auto*)+)

lift-definition *mat-addrow-gen-impl*
:: $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a \Rightarrow nat \Rightarrow nat \Rightarrow 'a\ mat\text{-}impl \Rightarrow 'a\ mat\text{-}impl$ **is**
λ *ad mul a k l* (*nr*,*nc*,*A*). *if* $l < nr$ *then let* $Ak = IArray.sub\ A\ k$; $Al = IArray.sub\ A\ l$;
$Ak' = IArray.of\text{-}fun\ (\lambda\ i.\ ad\ (mul\ a\ (Al\ !!\ i))\ (Ak\ !!\ i))\ (min\ (IArray.length\ Ak)\ (IArray.length\ Al))$;
$A' = IArray.of\text{-}fun\ (\lambda\ i.\ if\ i = k\ then\ Ak'\ else\ A\ !!\ i)\ (IArray.length\ A)$
in (*nr*,*nc*,A') *else* (*nr*,*nc*,*A*)
proof (*goal-cases*)
case (*1 ad mul a k l pp*)
obtain *nr nc A* **where** *pp*: *pp* = (*nr*,*nc*,*A*) **by** (*cases pp*)
obtain *rows* **where** *A*: *A* = *IArray rows* **by** (*cases A*)
from *1*[*unfolded pp A*, *simplified*]
have *nr*: *length rows* = *nr* **and** *nc*: $\bigwedge$ *r*. $r \in set\ rows \Longrightarrow length\ (IArray.list\text{-}of\ r) = nc$ **by** *auto*
show *?case*
proof (*cases* $l < nr$)
case *False*
thus *?thesis* **unfolding** *pp A prod.simps* **using** *nr nc* **by** *auto*
next
case *True*
thus *?thesis* **unfolding** *pp A prod.simps Let-def* **using** *nr nc*
by (*auto simp*: *set-list-update dest*: *in-set-takeD in-set-dropD*)
qed
qed

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 (*cases* $l <$ *dim-row-impl A*)
case *True*
hence *id*: *?r* = *mat-impl* (*mat-addrow-gen-impl ad mul a k l A*) **by** *simp*
show *?thesis* **unfolding** *id* **unfolding** *mat-addrow-gen-def*
proof (*rule eq-matI*, *goal-cases*)
case (*1 i j*)
thus *?case* **using** *True*
proof (*transfer*, *goal-cases*)
case (*1 i ad mul a k l A j*)

obtain *nr nc rows* **where** *A*: $A = (nr,nc,rows)$ **by** (*cases A*, *auto*)
from *1*[*unfolded A Let-def*]
have *nr*: *length* (*IArray.list-of rows*) = *nr*
and *nc*: *IArray.all* (λr. *length* (*IArray.list-of r*) = *nc*) *rows*
and *ij*: $i < nr$ $j < nc$ **and** *ij'*: $(i < nr \wedge j < nc) = True$
and *l*: $l < nr$ **by** *auto*
have *len*: $j <$ *length* (*IArray.list-of* (*IArray.list-of rows* ! *i*))
$j <$ *length* (*IArray.list-of* (*IArray.list-of rows* ! *l*))
using *ij nc nr l* **by** (*cases rows*, *auto*)+
show *?case* **unfolding** *A prod.simps fst-conv o-def snd-conv Let-def mk-mat-def ij' if-True*
using *ij nr nc l*
by (*cases* $k = i$, *auto simp*: *len*)
qed next
qed ((*transfer*, *auto simp*:*Let-def*)+)
qed *simp*

lemma *gauss-jordan-main-code*[*code*]:
gauss-jordan-main A B i j = (*let nr* = *dim-row A*; *nc* = *dim-col A in*
if $i < nr \wedge j < nc$ *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 −
note *simps* = *gauss-jordan-main.simps*[*of A B i j*] *Let-def*
let *?nr* = *dim-row A*
let *?nc* = *dim-col A*
let *?A'* = *multrow i* (*inverse* (*A* \$\$ (*i*,*j*))) *A*
let *?B'* = *multrow i* (*inverse* (*A* \$\$ (*i*,*j*))) *B*
show *?thesis*
proof (*cases* $i <$ *?nr* $\wedge$ $j <$ *?nc* $\wedge$ *A* \$\$ (*i*,*j*) ≠ *0* ∧ *A* \$\$ (*i*,*j*) ≠ *1*)
case *False*
thus *?thesis* **unfolding** *simps* **by** (*auto split*: *if-splits*)
next
case *True*
from *True* **have** *id*: *?A'* \$\$ (*i*,*j*) = *1* **by** *auto*
from *True* **have** *?l* = *gauss-jordan-main ?A' ?B' i j* **unfolding** *simps* **by** (*simp add*: *Let-def*)
also have . . . = *?r* **unfolding** *Let-def gauss-jordan-main.simps*[*of ?A' ?B' i j*] *id*
using *True* **by** *simp*
finally show *?thesis* **.**

qed
qed

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*
unfolding *mat-multcol-def* **by** *auto*

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*
unfolding *mat-swapcols-def* **by** *auto*

lemma *index-mat-addcol*[*simp*]:
i < *dim-row A* $\Longrightarrow$ j < *dim-col A* $\Longrightarrow$ *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* $\Longrightarrow$ j < *dim-col A* $\Longrightarrow$ *addcol a j l A* \$\$ (*i*,*j*) = *a* $*$ *A* \$\$(*i*,*l*) + *A*\$\$(*i*,*j*)
i < *dim-row A* $\Longrightarrow$ j < *dim-col A* $\Longrightarrow$ $k \neq j$ $\Longrightarrow$ *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*
unfolding *mat-addcol-def* **by** *auto*

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

lemma *col-addrow*:
$l \neq i \Longrightarrow i < n \Longrightarrow$ *col* (*addrow-mat n a k l*) *i* = *unit-vec n i*
$k < n \Longrightarrow l < n \Longrightarrow$ *col* (*addrow-mat n a k l*) *l* = *a* $\cdot_v$ *unit-vec n k* + *unit-vec n l*
by (*rule eq-vecI*, *auto*)

lemma *col-addcol*[*simp*]:
k < *dim-col A* $\Longrightarrow$ l < *dim-col A* $\Longrightarrow$ *col* (*addcol a k l A*) *k* = *a* $\cdot_v$ *col A l* + *col A k*
by (*rule eq-vecI*;*simp*)

lemma *addcol-mat*: **assumes** *A*: *A* $\in$ *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*
by (*rule eq-matI*, *insert A k*, *auto simp*: *col-addrow*
scalar-prod-add-distrib[*of - n*] *scalar-prod-smult-distrib*[*of - n*])

lemma *col-multrow*: $k \neq i \Longrightarrow i < n \Longrightarrow$ *col* (*multrow-mat n k a*) *i* = *unit-vec n i*
$k < n \Longrightarrow$ *col* (*multrow-mat n k a*) *k* = *a* $\cdot_v$ *unit-vec n k*
by (*rule eq-vecI*, *auto*)

lemma *multcol-mat*: **assumes** *A*: (*A* :: *'a* :: *comm-ring-1 mat*) $\in$ *carrier-mat nr n*
shows *multcol k a A* = *A* $*$ *multrow-mat n k a*
by (*rule eq-matI*, *insert A*, *auto simp*: *col-multrow smult-scalar-prod-distrib*[*of - n*])

lemma *col-swaprows*:
$l < n \Longrightarrow$ *col* (*swaprows-mat n l l*) *l* = *unit-vec n l*
$i \neq k \Longrightarrow i \neq l \Longrightarrow i < n \Longrightarrow$ *col* (*swaprows-mat n k l*) *i* = *unit-vec n i*
$k < n \Longrightarrow l < n \Longrightarrow$ *col* (*swaprows-mat n k l*) *l* = *unit-vec n k*
$k < n \Longrightarrow l < n \Longrightarrow$ *col* (*swaprows-mat n k l*) *k* = *unit-vec n l*
by (*rule eq-vecI*, *auto*)

lemma *swapcols-mat*: **assumes** *A*: $A \in$ *carrier-mat nr n* **and** *k*: $k < n$ $l < n$
shows *swapcols k l A* = *A* ∗ *swaprows-mat n k l*
by (*rule eq-matI*, *insert A k*, *auto simp*: *col-swaprows*)

Combining row and column-operations yields similarity transformations.

definition *add-col-sub-row* :: *'a* :: *ring-1* ⇒ *nat* ⇒ *nat* ⇒ *'a mat* ⇒ *'a mat* **where**
add-col-sub-row a k l A = *addrow* (− *a*) *k l* (*addcol a l k A*)

definition *mult-col-div-row* :: *'a* :: *field* ⇒ *nat* ⇒ *'a mat* ⇒ *'a mat* **where**
mult-col-div-row a k A = *multrow k* (*inverse a*) (*multcol k a A*)

definition *swap-cols-rows* :: *nat* ⇒ *nat* ⇒ *'a mat* ⇒ *'a mat* **where**
swap-cols-rows k l A = *swaprows k l* (*swapcols k l A*)

lemma *add-col-sub-row-carrier*[*simp*]:
dim-row (*add-col-sub-row a k l A*) = *dim-row A*
dim-col (*add-col-sub-row a k l A*) = *dim-col A*
A ∈ *carrier-mat n n* ⟹ *add-col-sub-row a k l A* ∈ *carrier-mat n n*
unfolding *add-col-sub-row-def carrier-mat-def* **by** *auto*

lemma *add-col-sub-index-row*[*simp*]:
i < *dim-row A* ⟹ *i* < *dim-col A* ⟹ *j* < *dim-row A* ⟹ *j* < *dim-col A* ⟹ *l* < *dim-row A*
⟹ *add-col-sub-row a k l A* $$ (*i*,*j*) = (*if*
i = *k* ∧ *j* = *l* *then A* $$ (*i*, *j*) + *a* ∗ *A* $$ (*i*, *i*) − *a* ∗ *a* ∗ *A* $$ (*j*, *i*) − *a* ∗ *A* $$ (*j*, *j*) *else if*
i = *k* ∧ *j* ≠ *l* *then A* $$ (*i*, *j*) − *a* ∗ *A* $$ (*l*, *j*) *else if*
i ≠ *k* ∧ *j* = *l* *then A* $$ (*i*, *j*) + *a* ∗ *A* $$ (*i*, *k*) *else A* $$ (*i*,*j*))
unfolding *add-col-sub-row-def* **by** (*auto simp*: *field-simps*)

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*))
unfolding *mult-col-div-row-def* **by** *auto*

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*
unfolding *mult-col-div-row-def carrier-mat-def* **by** *auto*

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*

unfolding *swap-cols-rows-def carrier-mat-def* **by** *auto*

lemma *swap-cols-rows-index*[*simp*]:
$i < \textit{dim-row}\ A \Longrightarrow i < \textit{dim-col}\ A \Longrightarrow j < \textit{dim-row}\ A \Longrightarrow j < \textit{dim-col}\ A \Longrightarrow a < \textit{dim-row}\ A \Longrightarrow b < \textit{dim-row}\ A$
$\Longrightarrow \textit{swap-cols-rows}\ a\ b\ A\ \$\$\ (i,j) = A\ \$\$\ (\textit{if}\ i = a\ \textit{then}\ b\ \textit{else if}\ i = b\ \textit{then}\ a\ \textit{else}\ i,$
$\textit{if}\ j = a\ \textit{then}\ b\ \textit{else if}\ j = b\ \textit{then}\ a\ \textit{else}\ j)$
unfolding *swap-cols-rows-def*
by *auto*

lemma *add-col-sub-row-similar*: **assumes** *A*: $A \in \textit{carrier-mat}\ n\ n$ **and** *kl*: $k < n\ l < n\ k \neq l$
shows *similar-mat* (*add-col-sub-row a k l A*) (A :: $'a$ :: *comm-ring-1 mat*)
proof (*rule similar-matI*)
let $?P = \textit{addrow-mat}\ n\ (-a)\ k\ l$
let $?Q = \textit{addrow-mat}\ n\ a\ k\ l$
let $?B = \textit{add-col-sub-row}\ a\ k\ l\ A$
show *carr*: $\{?B, A, ?P, ?Q\} \subseteq \textit{carrier-mat}\ n\ n$ **using** *A* **by** *auto*
show $?Q * ?P = 1_m\ n$ **by** (*rule addrow-mat-inv*[*OF kl*])
show $?P * ?Q = 1_m\ n$ **using** *addrow-mat-inv*[*OF kl*, *of* $-a$] **by** *simp*
have *col*: $\textit{addcol}\ a\ l\ k\ A = A * ?Q$
by (*rule addcol-mat*[*OF A kl*(*1*)])
have $?B = ?P * (A * ?Q)$ **unfolding** *add-col-sub-row-def col*
by (*rule addrow-mat*[*OF - kl*(*2*), *of - n*], *insert A*, *simp*)
thus $?B = ?P * A * ?Q$ **using** *carr* **by** (*simp add*: *assoc-mult-mat*[*of - n n - n - n*])
qed

lemma *mult-col-div-row-similar*: **assumes** *A*: $A \in \textit{carrier-mat}\ n\ n$ **and** *ak*: $k < n\ a \neq 0$
shows *similar-mat* (*mult-col-div-row a k A*) *A*
proof (*rule similar-matI*)
let $?P = \textit{multrow-mat}\ n\ k\ (\textit{inverse}\ a)$
let $?Q = \textit{multrow-mat}\ n\ k\ a$
let $?B = \textit{mult-col-div-row}\ a\ k\ A$
show *carr*: $\{?B, A, ?P, ?Q\} \subseteq \textit{carrier-mat}\ n\ n$ **using** *A* **by** *auto*
show $?Q * ?P = 1_m\ n$ **by** (*rule multrow-mat-inv*[*OF ak*])
show $?P * ?Q = 1_m\ n$ **using** *multrow-mat-inv*[*OF ak*(*1*), *of inverse a*] *ak*(*2*) **by** *simp*
have *col*: $\textit{multcol}\ k\ a\ A = A * ?Q$
by (*rule multcol-mat*[*OF A*])
have $?B = ?P * (A * ?Q)$ **unfolding** *mult-col-div-row-def col*
by (*rule multrow-mat*[*of - n n*], *insert A*, *simp*)
thus $?B = ?P * A * ?Q$ **using** *carr* **by** (*simp add*: *assoc-mult-mat*[*of - n n - n - n*])
qed

lemma *swap-cols-rows-similar*: **assumes** *A*: $A \in \textit{carrier-mat}\ n\ n$ **and** *kl*: $k < n\ l$

< *n*
  shows *similar-mat* (*swap-cols-rows k l A*) *A*
proof (*rule similar-matI*)
  let *?P* = *swaprows-mat n k l*
  let *?B* = *swap-cols-rows k l A*
  show *carr*: {*?B*, *A*, *?P*, *?P*} ⊆ *carrier-mat n n* **using** *A* **by** *auto*
  show *?P* ∗ *?P* = 1_m *n* **by** (*rule swaprows-mat-inv*[*OF kl*])
  show *?P* ∗ *?P* = 1_m *n* **by** *fact*
  have *col*: *swapcols k l A* = *A* ∗ *?P*
    by (*rule swapcols-mat*[*OF A kl*])
  have *?B* = *?P* ∗ (*A* ∗ *?P*) **unfolding** *swap-cols-rows-def col*
    by (*rule swaprows-mat*[*of - n n*], *insert A kl*, *auto*)
  thus *?B* = *?P* ∗ *A* ∗ *?P* **using** *carr* **by** (*simp add*: *assoc-mult-mat*[*of - n n - n - n*])
qed

lemma *swapcols-carrier*[*simp*]: (*swapcols l k A* ∈ *carrier-mat n m*) = (*A* ∈ *carrier-mat n m*)
  unfolding *mat-swapcols-def carrier-mat-def* **by** *auto*

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* ⇒ *nat* ⇒ *'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* ∈ *carrier-mat n m* ⟹ *I* < *n* ⟹ *swap-row-to-front A I* =
  mat n m (λ (*i*,*j*). *if i* = *0 then A* $$ (*I*,*j*)
  else if i ≤ *I then A* $$ (*i* − *1*, *j*) *else A* $$ (*i*,*j*))
proof (*induct I arbitrary*: *A*)
  case *0*
  thus *?case*
    by (*intro eq-matI*, *auto*)
next
  case (*Suc I A*)
  from *Suc*(*3*) **have** *I*: *I* < *n* **by** *auto*
  let *?I* = *Suc I*
  let *?A* = *swaprows I ?I A*
  have *AA*: *?A* ∈ *carrier-mat n m* **using** *Suc*(*2*) **by** *simp*
  have *swap-row-to-front A* (*Suc I*) = *swap-row-to-front ?A I* **by** *simp*
  also have ... = *mat n m*
   (λ(*i*, *j*). *if i* = *0 then ?A* $$ (*I*, *j*)
      else if i ≤ *I then ?A* $$ (*i* − *1*, *j*) *else ?A* $$ (*i*, *j*))
    using *Suc*(*1*)[*OF AA I*] **by** *simp*

also have ... = *mat n m*
($\lambda(i, j)$. *if* $i = 0$ *then* A \$\$ ($?I$, j)
else if $i \leq ?I$ *then* A \$\$ ($i - 1$, j) *else* A \$\$ (i, j))
by (*rule eq-matI*, *insert I Suc*(*2*), *auto*)
finally show *?case* **.**
qed

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 (*induct J arbitrary*: *A*)
case *0*
thus *?case*
by (*intro eq-matI*, *auto*)
next
case (*Suc J A*)
from *Suc*(*3*) **have** *J*: $J < m$ **by** *auto*
let $?J = Suc\ J$
let $?A = swapcols\ J\ ?J\ A$
have *AA*: $?A \in$ *carrier-mat n m* **using** *Suc*(*2*) **by** *simp*
have *swap-col-to-front A* (*Suc J*) = *swap-col-to-front ?A J* **by** *simp*
also have ... = *mat n m*
($\lambda(i, j)$. *if* $j = 0$ *then* $?A$ \$\$ (i, J)
else if $j \leq J$ *then* $?A$ \$\$ (i, $j - 1$) *else* $?A$ \$\$ (i, j))
using *Suc*(*1*)[*OF AA J*] **by** *simp*
also have ... = *mat n m*
($\lambda(i, j)$. *if* $j = 0$ *then* A \$\$ (i, $?J$)
else if $j \leq ?J$ *then* A \$\$ (i, $j - 1$) *else* A \$\$ (i, j))
by (*rule eq-matI*, *insert J Suc*(*2*), *auto*)
finally show *?case* **.**
qed

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*))
using *assms* **by** (*intro eq-matI*, *auto*)

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 ⇒ '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
  by (simp add: hom-uminus sign-def)

lemma(in comm-ring-hom) hom-det[simp]: det (map-mat hom A) = hom (det A)
  unfolding det-def by (auto simp: hom-distribs)

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))) unfolding det-def by auto

lemma det-smult[simp]: det (a ·m A) = a ^ dim-col A ∗ det A
proof −
 have [simp]: (∏ i = 0..<dim-col A. a) = a ^ dim-col A by(subst prod-constant;simp)
  show ?thesis
  unfolding det-def
  unfolding index-smult-mat
  by (auto intro: sum.cong simp: sum-distrib-left prod.distrib)
qed

lemma det-transpose: assumes A: A ∈ carrier-mat n n
  shows det (transpose-mat A) = det A
proof −
  let ?di = λA i j. A $$ (i,j)
  let ?U = {0 ..< n}
  have fU: finite ?U by simp
  let ?inv = Hilbert-Choice.inv
  {
    fix p
    assume p: p ∈ {p. p permutes ?U}
    from p have pU: p permutes ?U
      by blast
    have sth: signof (?inv p) = signof p
```

by (*rule signof-inv*[*OF* - *pU*], *simp*)
from *permutes-inj*[*OF pU*]
have *pi*: *inj-on p ?U*
by (*blast intro*: *subset-inj-on*)
let *?f* = λ*i*. *transpose-mat A* $$ (*i*, *?inv p i*)
note *pU-U* = *permutes-image*[*OF pU*]
note [*simp*] = *permutes-less*[*OF pU*]
have *prod ?f ?U* = *prod ?f* (*p* ' *?U*)
using *pU-U* **by** *simp*
also have ... = *prod* (*?f* $\circ$ *p*) *?U*
by (*rule prod.reindex*[*OF pi*])
also have ... = *prod* (λ*i*. *A* $$ (*i*, *p i*)) *?U*
by (*rule prod.cong*, *insert A*, *auto*)
finally have *signof* (*?inv p*) $*$ *prod ?f ?U* =
signof p $*$ *prod* (λ*i*. *A* $$ (*i*, *p i*)) *?U*
unfolding *sth* **by** *simp*
}
then show *?thesis*
unfolding *det-def* **using** *A*
by (*simp*, *subst sum-permutations-inverse*, *intro sum.cong*, *auto*)
qed

lemma *det-col*:
assumes *A*: *A* $\in$ *carrier-mat n n*
shows *det A* = ($\sum$ *p* | *p permutes* {*0* ..< *n*}. *signof p* $*$ ($\prod$*j*<*n*. *A* $$ (*p j*, *j*)))
(**is** - = (*sum* (λ*p*. - $*$ *?prod p*) *?P*))
proof −
let *?i* = *Hilbert-Choice.inv*
let *?N* = {*0* ..< *n*}
let *?f* = λ*p*. *signof p* $*$ *?prod p*
let *?prod'* = λ*p*. $\prod$*j*<*n*. *A* $$ (*j*, *?i p j*)
let *?prod''* = λ*p*. $\prod$*j*<*n*. *A* $$ (*j*, *p j*)
let *?f'* = λ*p*. *signof* (*?i p*) $*$ *?prod' p*
let *?f''* = λ*p*. *signof p* $*$ *?prod'' p*
let *?P'* = { *?i p* | *p*. *p permutes ?N* }
have [*simp*]: {*0*..<*n*} = {..<*n*} **by** *auto*
have *sum ?f ?P* = *sum ?f' ?P*
proof (*rule sum.cong*[*OF refl*],*unfold mem-Collect-eq*)
fix *p* **assume** *p*: *p permutes ?N*
have [*simp*]: *?prod p* = *?prod' p*
using *permutes-prod*[*OF p*, *of* λ*x y*. *A* $$ (*x*,*y*)] **by** *auto*
have [*simp*]: *signof p* = *signof* (*?i p*)
apply(*rule signof-inv*[*symmetric*]) **using** *p* **by** *auto*
show *?f p* = *?f' p* **by** *auto*
qed
also have ... = *sum ?f' ?P'*
by (*rule sum.cong*[*OF image-inverse-permutations*[*symmetric*]],*auto*)
also have ... = *sum ?f'' ?P*
unfolding *sum.reindex*[*OF inv-inj-on-permutes*,*unfolded image-Collect*]

unfolding *o-def*
apply (*rule sum.cong*[*OF refl*])
using *inv-inv-eq*[*OF permutes-bij*] **by** *force*
finally show *?thesis* **unfolding** *det-def'*[*OF A*] **by** *auto*
qed

lemma *mat-det-left-def*: **assumes** *A*: $A \in$ *carrier-mat n n*
shows *det A* = ($\sum$ *p*∈{*p. p permutes* {*0..<dim-row A*}}. *signof p* ∗ ($\prod$ *i* = *0* ..< *dim-row A. A* $$ (*p i*, *i*)))
proof −
have *cong*: $\bigwedge$ *a b c. b* = *c* ⟹ *a* ∗ *b* = *a* ∗ *c* **by** *simp*
show *?thesis*
unfolding *det-transpose*[*OF A*, *symmetric*]
unfolding *det-def index-transpose-mat* **using** *A* **by** *simp*
qed

lemma *det-upper-triangular*:
assumes *ut*: *upper-triangular A*
and *m*: $A \in$ *carrier-mat n n*
shows *det A* = *prod-list* (*diag-mat A*)
proof −
note *det-def* = *det-def'*[*OF m*]
let *?U* = {*0..<n*}
let *?PU* = {*p. p permutes ?U*}
let *?pp* = *λp. signof p* ∗ ($\prod$ *i* = *0* ..< *n. A* $$ (*i*, *p i*))
have *fU*: *finite ?U*
by *simp*
from *finite-permutations*[*OF fU*] **have** *fPU*: *finite ?PU* **.**
have *id0*: {*id*} ⊆ *?PU*
by (*auto simp add*: *permutes-id*)
{
fix *p*
assume *p*: *p* ∈ *?PU* − {*id*}
from *p* **have** *pU*: *p permutes ?U* **and** *pid*: *p* ≠ *id*
by *blast+*
from *permutes-natset-ge*[*OF pU*] *pid* **obtain** *i* **where** *i*: *p i* < *i* **and** *i* < *n*
by *fastforce*
from *upper-triangularD*[*OF ut i*] ‹*i* < *n*› *m*
have *ex*:∃ *i* ∈ *?U. A* $$ (*i*,*p i*) = *0* **by** *auto*
have ($\prod$ *i* = *0* ..< *n. A* $$ (*i*, *p i*)) = *0*
by (*rule prod-zero*[*OF fU ex*])
hence *?pp p* = *0* **by** *simp*
}
then have *p0*: $\bigwedge$ *p. p* ∈ *?PU* − {*id*} ⟹ *?pp p* = *0*
by *blast*
from *m* **have** *dim*: *dim-row A* = *n* **by** *simp*
have *det A* = ($\sum$ *p* ∈ *?PU. ?pp p*) **unfolding** *det-def* **by** *auto*
also have ... = *?pp id* + ($\sum$ *p* ∈ *?PU* − {*id*}. *?pp p*)
by (*rule sum.remove*, *insert id0 fPU m*, *auto simp*: *p0*)

also have ($\sum$ *p* ∈ *?PU* − {*id*}. *?pp p*) = *0*
by (*rule sum.neutral*, *insert fPU*, *auto simp*: *p0*)
finally show *?thesis* **using** *m* **by** (*auto simp*: *prod-list-diag-prod*)
qed

lemma *det-single*: **assumes** *A* ∈ *carrier-mat 1 1*
shows *det A* = *A* $$ (*0*,*0*)
by (*subst det-upper-triangular*[*of - 1*], *insert assms*, *auto simp*: *diag-mat-def*)

lemma *det-one*[*simp*]: *det* (1_m *n*) = *1*
proof −
have *det* (1_m *n*) = *prod-list* (*diag-mat* (1_m *n*))
by (*rule det-upper-triangular*[*of - n*], *auto*)
also have ... = *1* **by** (*induct n*, *auto*)
finally show *?thesis* **.**
qed

lemma *det-zero*[*simp*]: **assumes** *n* > *0* **shows** *det* (0_m *n n*) = *0*
proof −
have *det* (0_m *n n*) = *prod-list* (*diag-mat* (0_m *n n*))
by (*rule det-upper-triangular*[*of - n*], *auto*)
also have ... = *0* **using** ‹*n* > *0*› **by** (*cases n*, *auto*)
finally show *?thesis* **.**
qed

lemma *det-dim-zero*[*simp*]: *A* ∈ *carrier-mat 0 0* ⟹ *det A* = *1*
unfolding *det-def carrier-mat-def sign-def* **by** *auto*

lemma *det-lower-triangular*:
assumes *ld*: ⋀*i j*. *i* < *j* ⟹ *j* < *n* ⟹ *A* $$ (*i*,*j*) = *0*
and *m*: *A* ∈ *carrier-mat n n*
shows *det A* = *prod-list* (*diag-mat A*)
proof −
have *det A* = *det* (*transpose-mat A*) **using** *det-transpose*[*OF m*] **by** *simp*
also have ... = *prod-list* (*diag-mat* (*transpose-mat A*))
by (*rule det-upper-triangular*, *insert m ld*, *auto*)
finally show *?thesis* **using** *m* **by** *simp*
qed

lemma *det-permute-rows*: **assumes** *A*: *A* ∈ *carrier-mat n n*
and *p*: *p permutes* {*0* ..< (*n* :: *nat*)}
shows *det* (*mat n n* (λ (*i*,*j*). *A* $$ (*p i*, *j*))) = *signof p* ∗ *det A*
proof −
let *?U* = {*0* ..< (*n* :: *nat*)}
have *cong*: ⋀ *a b c*. *b* = *c* ⟹ *a* ∗ *b* = *a* ∗ *c* **by** *auto*
have *det* (*mat n n* (λ (*i*,*j*). *A* $$ (*p i*, *j*))) =
($\sum$ *q* ∈ {*q* . *q permutes ?U*}. *signof q* ∗ ($\prod$ *i* ∈ *?U*. *A* $$ (*p i*, *q i*)))
unfolding *det-def* **using** *A p* **by** *auto*
also have ... = ($\sum$ *q* ∈ {*q* . *q permutes ?U*}. *signof* (*q* ∘ *p*) ∗ ($\prod$ *i* ∈ *?U*. *A*

$$ (p i, (q ∘ p) i)))
    by (*rule sum-permutations-compose-right*[*OF p*])
  finally have *1*: *det* (*mat n n* (λ (*i*,*j*). *A* $$ (*p i*, *j*)))
    = (∑ *q* ∈ {*q* . *q permutes ?U*}. *signof* (*q* ∘ *p*) ∗ (∏ *i* ∈ *?U*. *A* $$ (*p i*, (*q* ∘ *p*) *i*))) **.**
  have *2*: *signof p* ∗ *det A* =
    (∑ *q*∈{*q*. *q permutes ?U*}. *signof p* ∗ *signof q* ∗ (∏ *i*∈ *?U*. *A* $$ (*i*, *q i*)))
    unfolding *det-def'*[*OF A*] *sum-distrib-left* **by** (*simp add*: *ac-simps*)
  show *?thesis* **unfolding** *1 2*
  proof (*rule sum.cong*, *insert p A*, *auto*)
    fix *q*
    assume *q*: *q permutes ?U*
    let *?inv* = *Hilbert-Choice.inv*
    from *permutes-inv*[*OF p*] **have** *ip*: *?inv p permutes ?U* **.**
    have *prod* (λ*i*. *A* $$ (*p i*, (*q* ∘ *p*) *i*)) *?U* =
      prod (λ*i*. *A* $$ ((*p* ∘ *?inv p*) *i*, (*q* ∘ (*p* ∘ *?inv p*)) *i*)) *?U* **unfolding** *o-def*
      by (*rule trans*[*OF prod.permute*[*OF ip*] *prod.cong*], *insert A p q*, *auto*)
    also have … = *prod* (λ*i*. *A*$$(*i*,*q i*)) *?U*
      by (*simp only*: *o-def permutes-inverses*[*OF p*])
    finally have *thp*: *prod* (λ*i*. *A* $$ (*p i*, (*q* ∘ *p*) *i*)) *?U* = *prod* (λ*i*. *A*$$(*i*,*q i*)) *?U* **.**
    show *signof* (*q* ∘ *p*) ∗ (∏ *i*∈{*0*..<*n*}. *A* $$ (*p i*, *q* (*p i*))) =
        signof p ∗ *signof q* ∗ (∏ *i*∈{*0*..<*n*}. *A* $$ (*i*, *q i*))
      unfolding *thp*[*symmetric*] *signof-compose*[*OF q p*]
      by (*simp add*: *ac-simps*)
  qed
qed

lemma *det-multrow-mat*: **assumes** *k*: *k* < *n*
  shows *det* (*multrow-mat n k a*) = *a*
proof (*rule trans*[*OF det-lower-triangular*[*of n*]], *unfold prod-list-diag-prod*)
  let *?f* = λ *i*. *multrow-mat n k a* $$ (*i*, *i*)
  have (∏ *i*∈{*0*..<*n*}. *?f i*) = *?f k* ∗ (∏ *i*∈{*0*..<*n*} − {*k*}. *?f i*)
    by (*rule prod.remove*, *insert k*, *auto*)
  also have (∏ *i*∈{*0*..<*n*} − {*k*}. *?f i*) = *1*
    by (*rule prod.neutral*, *auto*)
  finally show (∏ *i*∈{*0*..<*dim-row* (*multrow-mat n k a*)}. *?f i*) = *a* **using** *k* **by** *simp*
qed (*insert k*, *auto*)

lemma *swap-rows-mat-eq-permute*:
  k < *n* ⟹ *l* < *n* ⟹ *swaprows-mat n k l* = *mat n n* (λ(*i*, *j*). 1_m *n* $$ (*transpose k l i*, *j*))
  by (*rule eq-matI*) (*auto simp add*: *transpose-def*)

lemma *det-swaprows-mat*: **assumes** *k*: *k* < *n* **and** *l*: *l* < *n* **and** *kl*: *k* ≠ *l*
  shows *det* (*swaprows-mat n k l*) = − *1*
proof −
  let *?n* = {*0* ..< (*n* :: *nat*)}

let *?p* = *transpose k l*
have *p*: *?p permutes ?n*
by (*rule permutes-swap-id*, *insert k l*, *auto*)
show *?thesis*
by (*rule trans*[*OF trans*[*OF - det-permute-rows*[*OF one-carrier-mat*[*of n*] *p*]]], *subst swap-rows-mat-eq-permute*[*OF k l*], *auto simp*: *sign-swap-id kl*)
qed

lemma *det-addrow-mat*:
assumes *l*: $k \neq l$
shows *det* (*addrow-mat n a k l*) = *1*
proof −
have *det* (*addrow-mat n a k l*) = *prod-list* (*diag-mat* (*addrow-mat n a k l*))
proof (*cases* $k < l$)
case *True*
show *?thesis*
by (*rule det-upper-triangular*[*of - n*], *insert True*, *auto intro*!: *upper-triangularI*)
next
case *False*
show *?thesis*
by (*rule det-lower-triangular*[*of n*], *insert False*, *auto*)
qed
also have . . . = *1* **unfolding** *prod-list-diag-prod*
by (*rule prod.neutral*, *insert l*, *auto*)
finally show *?thesis* **.**
qed

The following proof is new, as it does not use $2 \neq 0$ as in Multivariate-Analysis.

lemma *det-identical-rows*:
assumes *A*: $A \in$ *carrier-mat n n*
and *ij*: $i \neq j$
and *i*: $i < n$ **and** *j*: $j < n$
and *r*: *row A i* = *row A j*
shows *det A* = *0*
proof−
let *?p* = *transpose i j*
let *?n* = {*0* ..< *n*}
have *sp*: *signof ?p* = − *1 sign ?p* = (− *1* :: *int*) **using** *ij*
by (*auto simp add*: *sign-swap-id*)
let *?f* = λ *p*. *signof p* ∗ ($\prod$ *i*∈*?n*. *A* $$ (*p i*, *i*))
let *?all* = {*p*. *p permutes ?n*}
let *?one* = {*p*. *p permutes ?n* ∧ *sign p* = (*1* :: *int*)}
let *?none* = {*p*. *p permutes ?n* ∧ *sign p* ≠ (*1* :: *int*)}
let *?pone* = (λ *p*. *?p o p*) ‘ *?one*
have *split*: *?one* ∪ *?none* = *?all* **by** *auto*
have *p*: *?p permutes ?n* **by** (*rule permutes-swap-id*, *insert i j*, *auto*)
from *permutes-inj*[*OF p*] **have** *injp*: *inj ?p* **by** *auto*
{

fix *q*
assume *q*: *q permutes ?n*
have $(\prod k\in ?n.\ A\ \$\$\ (?p\ (q\ k),\ k)) = (\prod k\in ?n.\ A\ \$\$\ (q\ k,\ k))$
proof (*rule prod.cong*)
fix *k*
assume *k*: $k \in ?n$
from *r* **have** *row*: *row A i \$ k = row A j \$ k* **by** *simp*
hence *A \$\$ (i,k) = A \$\$ (j,k)* **using** *k i j A* **by** *auto*
thus *A \$\$ (?p (q k), k) = A \$\$ (q k, k)*
by (*cases q k = i, auto, cases q k = j, auto*)
qed (*insert A q, auto*)
} **note** $*$ = *this*
have *pp*: $\bigwedge q.\ q$ *permutes ?n* $\Longrightarrow$ *permutation q* **unfolding** *permutation-permutes* **by** *auto*
have *det A* = $(\sum p\in ?one \cup ?none.\ ?f\ p)$
using *A* **unfolding** *mat-det-left-def*[*OF A*] *split* **by** *simp*
also have ... = $(\sum p\in ?one.\ ?f\ p) + (\sum p\in ?none.\ ?f\ p)$
by (*rule sum.union-disjoint*, *insert A*, *auto simp*: *finite-permutations*)
also have *?none = ?pone*
proof −
{
fix *q*
assume $q \in ?none$
hence *q*: *q permutes ?n* **and** *sq*: *sign q* = $(-\ 1 :: int)$ **unfolding** *sign-def* **by** *auto*
from *permutes-compose*[*OF q p*] *sign-compose*[*OF pp*[*OF p*] *pp*[*OF q*], *unfolded sp sq*]
have $?p\ o\ q \in ?one$ **by** *auto*
hence $?p\ o\ (?p\ o\ q) \in ?pone$ **by** *auto*
also have $?p\ o\ (?p\ o\ q) = q$
by (*auto simp*: *swap-id-eq*)
finally have $q \in ?pone$ **.**
}
moreover
{
fix *pq*
assume $pq \in ?pone$
then obtain *q* **where** *q*: $q \in ?one$ **and** *pq*: $pq = ?p\ o\ q$ **by** *auto*
from *q* **have** *q*: *q permutes ?n* **and** *sq*: *sign q* = $(1 :: int)$ **by** *auto*
from *sign-compose*[*OF pp*[*OF p*] *pp*[*OF q*], *unfolded sq sp*]
have *spq*: *sign pq* = $(-\ 1 :: int)$ **unfolding** *pq* **by** *auto*
from *permutes-compose*[*OF q p*] **have** *pq*: *pq permutes ?n* **unfolding** *pq* **by** *auto*
from *pq spq* **have** $pq \in ?none$ **by** *auto*
}
ultimately
show *?thesis* **by** *blast*
qed
also have $(\sum p\in ?pone.\ ?f\ p) = (\sum p\in ?one.\ ?f\ (?p\ o\ p))$

proof (*rule trans*[*OF sum.reindex*])
show *inj-on* ((∘) *?p*) *?one*
using *fun.inj-map*[*OF injp*] **unfolding** *inj-on-def* **by** *auto*
qed *simp*
also have ($\sum$ *p*∈ *?one*. *?f p*) + ($\sum$ *p*∈ *?one*. *?f* (*?p o p*))
= ($\sum$ *p*∈ *?one*. *?f p* + *?f* (*?p o p*))
by (*rule sum.distrib*[*symmetric*])
also have . . . = *0*
by (*rule sum.neutral*, *insert A*, *auto simp*:
sp sign-compose[*OF pp*[*OF p*] *pp*] *ij finite-permutations* ∗)
finally show *?thesis* **.**
qed

lemma *det-row-0*: **assumes** *k*: *k* < *n*
and *c*: *c* ∈ {*0* ..< *n*} → *carrier-vec n*
shows *det* (mat_r *n n* (λ*i*. *if i* = *k then* 0_v *n else c i*)) = *0*
proof −
{
fix *p*
assume *p*: *p permutes* {*0* ..< *n*}
have ($\prod$ *i*∈{*0*..<*n*}. mat_r *n n* (λ*i*. *if i* = *k then* 0_v *n else c i*) $$ (*i*, *p i*)) = *0*
by (*rule prod-zero*[*OF - bexI*[*of - k*]],
insert k p c[*unfolded carrier-vec-def*], *auto*)
}
thus *?thesis* **unfolding** *det-def* **by** *simp*
qed

lemma *det-row-add*:
assumes *abc*: *a k* ∈ *carrier-vec n b k* ∈ *carrier-vec n c* ∈ {*0*..<*n*} → *carrier-vec n*
and *k*: *k* < *n*
shows *det*(mat_r *n n* (λ *i*. *if i* = *k then a i* + *b i else c i*)) =
det(mat_r *n n* (λ *i*. *if i* = *k then a i else c i*)) +
det(mat_r *n n* (λ *i*. *if i* = *k then b i else c i*))
(**is** *?lhs* = *?rhs*)
proof −
let *?n* = {*0*..<*n*}
let *?m* = λ *a b p i*. mat_r *n n* (λ*i*. *if i* = *k then a i else b i*) $$ (*i*, *p i*)
let *?c* = λ *p i*. mat_r *n n c* $$ (*i*, *p i*)
let *?ab* = λ *i*. *a i* + *b i*
note *intros* = *add-carrier-vec*[*of - n*]
have *?rhs* = ($\sum$ *p*∈{*p*. *p permutes ?n*}.
signof p ∗ ($\prod$ *i*∈*?n*. *?m a c p i*)) + ($\sum$ *p*∈{*p*. *p permutes ?n*}. *signof p* ∗ ($\prod$ *i*∈*?n*. *?m b c p i*))
unfolding *det-def* **by** *simp*
also have . . . = ($\sum$ *p*∈{*p*. *p permutes ?n*}. *signof p* ∗ ($\prod$ *i*∈*?n*. *?m a c p i*) + *signof p* ∗ ($\prod$ *i*∈*?n*. *?m b c p i*))
by (*rule sum.distrib*[*symmetric*])
also have . . . = ($\sum$ *p*∈{*p*. *p permutes ?n*}. *signof p* ∗ ($\prod$ *i*∈*?n*. *?m ?ab c p i*))

proof (*rule sum.cong*, *force*)
fix *p*
assume $p \in \{p.\ p\ permutes\ ?n\}$
hence *p*: *p permutes ?n* **by** *simp*
show $signof\ p * (\prod i{\in}?n.\ ?m\ a\ c\ p\ i) + signof\ p * (\prod i{\in}?n.\ ?m\ b\ c\ p\ i) =$
$signof\ p * (\prod i{\in}?n.\ ?m\ ?ab\ c\ p\ i)$
unfolding *distrib-left*[*symmetric*]
proof (*rule arg-cong*[*of - -* λ *a. signof p* $*$ *a*])
from *k* **have** *f*: *finite ?n* **and** *k'*: $k \in ?n$ **by** *auto*
let $?nk = ?n - \{k\}$
note *split* = *prod.remove*[*OF f k'*]
have *id1*: $(\prod i{\in}?n.\ ?m\ a\ c\ p\ i) = ?m\ a\ c\ p\ k * (\prod i{\in}?nk.\ ?m\ a\ c\ p\ i)$
by (*rule split*)
have *id2*: $(\prod i{\in}?n.\ ?m\ b\ c\ p\ i) = ?m\ b\ c\ p\ k * (\prod i{\in}?nk.\ ?m\ b\ c\ p\ i)$
by (*rule split*)
have *id3*: $(\prod i{\in}?n.\ ?m\ ?ab\ c\ p\ i) = ?m\ ?ab\ c\ p\ k * (\prod i{\in}?nk.\ ?m\ ?ab\ c\ p\ i)$
by (*rule split*)
have *id*: $\bigwedge a.\ (\prod i{\in}?nk.\ ?m\ a\ c\ p\ i) = (\prod i{\in}?nk.\ ?c\ p\ i)$
by (*rule prod.cong*, *insert abc k p*, *auto intro*!: *intros*)
have *ab*: $?ab\ k \in carrier\text{-}vec\ n$ **using** *abc* **by** (*auto intro*: *intros*)
{
fix *f*
assume $f\ k \in (carrier\text{-}vec\ n :: 'a\ vec\ set)$
hence $mat_r\ n\ n\ (\lambda i.\ if\ i = k\ then\ f\ i\ else\ c\ i)\ \$\$\ (k,\ p\ k) = f\ k\ \$\ p\ k$
by (*insert p k abc*, *auto*)
} **note** *first* = *this*
note *id'* = *id1 id2 id3*
have *dist*: $(a\ k + b\ k)\ \$\ p\ k = a\ k\ \$\ p\ k + b\ k\ \$\ p\ k$
by (*rule index-add-vec*(*1*), *insert p k abc*, *force*)
show $(\prod i{\in}?n.\ ?m\ a\ c\ p\ i) + (\prod i{\in}?n.\ ?m\ b\ c\ p\ i) = (\prod i{\in}?n.\ ?m\ ?ab\ c\ p\ i)$
unfolding *id' id first*[*of a*, *OF abc*(*1*)] *first*[*of b*, *OF abc*(*2*)] *first*[*of ?ab*, *OF ab*] *dist*
by (*rule distrib-right*[*symmetric*])
qed
qed
also have ... = *?lhs* **unfolding** *det-def* **by** *simp*
finally show *?thesis* **by** *simp*
qed

lemma *det-linear-row-finsum*:
assumes *fS*: *finite S* **and** *c*: $c \in \{0..<n\} \to carrier\text{-}vec\ n$ **and** *k*: $k < n$
and *a*: $a\ k \in S \to carrier\text{-}vec\ n$
shows $det\ (mat_r\ n\ n\ (\lambda\ i.\ if\ i = k\ then\ finsum\text{-}vec\ TYPE('a :: comm\text{-}ring\text{-}1)\ n\ (a\ i)\ S\ else\ c\ i)) =$
$sum\ (\lambda j.\ det\ (mat_r\ n\ n\ (\lambda\ i.\ if\ i = k\ then\ a\ \ i\ j\ else\ c\ i)))\ S$
proof −
let *?sum* = *finsum-vec TYPE*('*a*) *n*
show *?thesis* **using** *a*

```
  proof (induct rule: finite-induct[OF fS])
    case 1
    show ?case
      by (simp, unfold finsum-vec-empty, rule det-row-0[OF k c])
  next
    case (2 x F)
    from 2(4) have ak: a k ∈ F → carrier-vec n and akx: a k x ∈ carrier-vec n
by auto
    {
      fix i
      note if-cong[OF refl finsum-vec-insert[OF 2(1−2)],
        of - a i n c i c i]
    } note ∗ = this
    show ?case
    proof (subst ∗)
      show det (mat_r n n (λi. if i = k then a i x + ?sum (a i) F else c i)) =
        (∑j∈insert x F. det (mat_r n n (λi. if i = k then a i j else c i)))
      proof (subst det-row-add)
        show det (mat_r n n (λi. if i = k then a i x else c i)) +
          det (mat_r n n (λi. if i = k then ?sum (a i) F else c i)) =
        (∑j∈insert x F. det (mat_r n n (λi. if i = k then a i j else c i)))
        unfolding 2(3)[OF ak] sum.insert[OF 2(1−2)] by simp
      qed (insert c k ak akx 2(1),
        auto intro!: finsum-vec-closed)
    qed (insert akx ak, force+)
  qed
qed


lemma det-linear-rows-finsum-lemma:
  assumes fS: finite S
    and fT: finite T and c: c ∈ {0..<n} → carrier-vec n
    and T: T ⊆ {0 ..< n}
    and a: a ∈ T → S → carrier-vec n
  shows det (mat_r n n (λ i. if i ∈ T then finsum-vec TYPE('a :: comm-ring-1) n
(a i) S else c i)) =
    sum (λf. det(mat_r n n (λ i. if i ∈ T then a i (f i) else c i)))
      {f. (∀ i ∈ T. f i ∈ S) ∧ (∀ i. i ∉ T ⟶ f i = i)}
proof −
  let ?sum = finsum-vec TYPE('a) n
  show ?thesis using fT c a T
  proof (induct T arbitrary: a c set: finite)
    case empty
    let ?f = (λ i. i) :: nat ⇒ nat
    have [simp]: {f. ∀ i. f i = i} = {?f} by auto
    show ?case by simp
  next
    case (insert z T a c)
    hence z: z < n and azS: a z ∈ S → carrier-vec n by auto
```

let $?F = \lambda T. \{f. (\forall i \in T. f\ i \in S) \wedge (\forall i. i \notin T \longrightarrow f\ i = i)\}$
let $?h = \lambda(y,g)\ i.$ *if* $i = z$ *then* y *else* $g\ i$
let $?k = \lambda h. (h(z),(\lambda i.$ *if* $i = z$ *then* i *else* $h\ i))$
let $?s = \lambda\ k\ a\ c\ f.\ det(mat_r\ n\ n\ (\lambda\ i.$ *if* $i \in T$ *then* $a\ i\ (f\ i)$ *else* $c\ i))$
let $?c = \lambda j\ i.$ *if* $i = z$ *then* $a\ i\ j$ *else* $c\ i$
have *thif*: $\bigwedge a\ b\ c\ d.$ (*if* $a \vee b$ *then* c *else* d) = (*if* a *then* c *else if* b *then* c *else* d)
by *simp*
have *thif2*: $\bigwedge a\ b\ c\ d\ e.$ (*if* a *then* b *else if* c *then* d *else* e) =
(*if* c *then* (*if* a *then* b *else* d) *else* (*if* a *then* b *else* e))
by *simp*
from ‹$z \notin T$› **have** *nz*: $\bigwedge i.\ i \in T \Longrightarrow i = z \longleftrightarrow False$
by *auto*
from *insert* **have** *c*: $\bigwedge\ i.\ i < n \Longrightarrow c\ i \in carrier\text{-}vec\ n$ **by** *auto*
have *fin*: *finite* $\{f. (\forall i{\in}T. f\ i \in S) \wedge (\forall i. i \notin T \longrightarrow f\ i = i)\}$
by (*rule finite-bounded-functions*[*OF fS insert*(*1*)])
have *det* ($mat_r\ n\ n$ ($\lambda\ i.$ *if* $i \in$ *insert* $z\ T$ *then* $?sum\ (a\ i)\ S$ *else* $c\ i$)) =
det ($mat_r\ n\ n$ ($\lambda\ i.$ *if* $i = z$ *then* $?sum\ (a\ i)\ S$ *else if* $i \in T$ *then* $?sum\ (a\ i)\ S$ *else* $c\ i$))
unfolding *insert-iff thif* **..**
also have ... = ($\sum j{\in}S.$ *det* ($mat_r\ n\ n$ ($\lambda\ i.$ *if* $i \in T$ *then* $?sum\ (a\ i)\ S$ *else if* $i = z$ *then* $a\ i\ j$ *else* $c\ i$)))
apply (*subst det-linear-row-finsum*[*OF fS - z*])
prefer *3*
apply (*subst thif2*)
using *nz*
apply (*simp cong del*: *if-weak-cong cong add*: *if-cong*)
apply (*insert azS c fS insert*(*5*), (*force intro*!: *finsum-vec-closed*)+)
done
also have ... = (*sum* (λ (j, f). *det* ($mat_r\ n\ n$ ($\lambda\ i.$ *if* $i \in T$ *then* $a\ i\ (f\ i)$
else if $i = z$ *then* $a\ i\ j$
else $c\ i$))) ($S \times\ ?F\ T$))
unfolding *sum.cartesian-product*[*symmetric*]
by (*rule sum.cong*[*OF refl*], *subst insert.hyps*(*3*),
insert azS c fin z insert(*5−6*), *auto*)
finally have *tha*:
det ($mat_r\ n\ n$ ($\lambda\ i.$ *if* $i \in$ *insert* $z\ T$ *then* $?sum\ (a\ i)\ S$ *else* $c\ i$)) =
(*sum* (λ (j, f). *det* ($mat_r\ n\ n$ ($\lambda\ i.$ *if* $i \in T$ *then* $a\ i\ (f\ i)$
else if $i = z$ *then* $a\ i\ j$
else $c\ i$))) ($S \times\ ?F\ T$)) **.**
show *?case* **unfolding** *tha*
by (*rule sum.reindex-bij-witness*[**where** $i{=}?k$ **and** $j{=}?h$], *insert* ‹$z \notin T$›
azS c fS insert(*5−6*) *z fin*,
auto intro!: *arg-cong*[*of - - det*])
qed
qed

lemma *det-linear-rows-sum*:
assumes *fS*: *finite S*

and *a*: $a \in \{0..<n\} \rightarrow S \rightarrow$ *carrier-vec n*
shows *det* (*mat*$_r$ *n n* (λ *i. finsum-vec TYPE*('*a* :: *comm-ring-1*) *n* (*a i*) *S*)) =
sum (λf. *det* (*mat*$_r$ *n n* (λ *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 −
let *?T* = $\{0..<n\}$
have *fT*: *finite ?T* **by** *auto*
have *th0*: $\bigwedge x\ y$. *mat*$_r$ *n n* (λ *i. if i* $\in$ *?T then x i else y i*) = *mat*$_r$ *n n* (λ *i. x i*)
by (*rule eq-rowI*, *auto*)
have *c*: (λ -. 0_v *n*) $\in$ *?T* $\rightarrow$ *carrier-vec n* **by** *auto*
show *?thesis*
by (*rule det-linear-rows-finsum-lemma*[*OF fS fT c subset-refl a*, *unfolded th0*])
qed

lemma *det-rows-mul*:
assumes *a*: $a \in \{0..<n\} \rightarrow$ *carrier-vec n*
shows *det*(*mat*$_r$ *n n* (λ *i. c i* $\cdot_v$ *a i*)) =
prod c $\{0..<n\}$ $*$ *det*(*mat*$_r$ *n n* (λ *i. a i*))
proof −
have *A*: *mat*$_r$ *n n* (λ *i. c i* $\cdot_v$ *a i*) $\in$ *carrier-mat n n*
and *A'*: *mat*$_r$ *n n* (λ *i. a i*) $\in$ *carrier-mat n n* **using** *a* **unfolding** *carrier-mat-def*
by *auto*
show *?thesis* **unfolding** *det-def'*[*OF A*] *det-def'*[*OF A'*]
proof (*rule trans*[*OF sum.cong sum-distrib-left*[*symmetric*]])
fix *p*
assume *p*: $p \in \{p.\ p$ *permutes* $\{0..<n\}\}$
have *id*: ($\prod$ *ia*$\in\{0..<n\}$. *mat*$_r$ *n n* (λi. *c i* $\cdot_v$ *a i*) $$ (*ia*, *p ia*))
= *prod c* $\{0..<n\}$ $*$ ($\prod$ *ia*$\in\{0..<n\}$. *mat*$_r$ *n n a* $$ (*ia*, *p ia*))
unfolding *prod.distrib*[*symmetric*]
by (*rule prod.cong*, *insert p a*, *force*+)
show *signof p* $*$ ($\prod$ *ia*$\in\{0..<n\}$. *mat*$_r$ *n n* (λi. *c i* $\cdot_v$ *a i*) $$ (*ia*, *p ia*)) =
prod c $\{0..<n\}$ $*$ (*signof p* $*$ ($\prod$ *ia*$\in\{0..<n\}$. *mat*$_r$ *n n a* $$ (*ia*, *p ia*)))
unfolding *id* **by** *auto*
qed *simp*
qed

lemma *mat-mul-finsum-alt*:
assumes *A*: $A \in$ *carrier-mat nr n* **and** *B*: $B \in$ *carrier-mat n nc*
shows $A * B$ = *mat*$_r$ *nr nc* (λ *i. finsum-vec TYPE*('*a* :: *semiring-0*) *nc* (λk. *A* $$ (*i*,*k*) $\cdot_v$ *row B k*) $\{0 ..< n\}$)
by (*rule eq-matI*, *insert A B*, *auto*, *subst index-finsum-vec*, *auto simp*: *scalar-prod-def intro*: *sum.cong*)

lemma *det-mult*:
assumes *A*: $A \in$ *carrier-mat n n* **and** *B*: $B \in$ *carrier-mat n n*
shows *det* ($A * B$) = *det A* $*$ *det* (*B* :: '*a* :: *comm-ring-1 mat*)
proof −

let *?U* = {*0* ..< *n*}
let *?F* = {*f*. (∀ *i*∈ *?U*. *f i* ∈ *?U*) ∧ (∀ *i*. *i* ∉ *?U* ⟶ *f i* = *i*)}
let *?PU* = {*p*. *p permutes ?U*}
have *fU*: *finite ?U*
  by *blast*
have *fF*: *finite ?F*
  by (*rule finite-bounded-functions*, *auto*)
{
  fix *p*
  assume *p*: *p permutes ?U*
  have *p* ∈ *?F* **unfolding** *mem-Collect-eq permutes-in-image*[*OF p*]
    using *p*[*unfolded permutes-def*] **by** *simp*
}
then have *PUF*: *?PU* ⊆ *?F* **by** *blast*
{
  fix *f*
  assume *fPU*: *f* ∈ *?F* − *?PU*
  have *fUU*: *f* ‘ *?U* ⊆ *?U*
    using *fPU* **by** *auto*
  from *fPU* **have** *f*: ∀ *i* ∈ *?U*. *f i* ∈ *?U* ∀ *i*. *i* ∉ *?U* ⟶ *f i* = *i* ¬(∀ *y*. ∃!*x*. *f x* = *y*)
    unfolding *permutes-def* **by** *auto*
  let *?A* = *mat*$_r$ *n n* (λ *i*. *A* $$ (*i*, *f i*) $\cdot_v$ *row B* (*f i*))
  let *?B* = *mat*$_r$ *n n* (λ *i*. *row B* (*f i*))
  have *B′*: *?B* ∈ *carrier-mat n n*
    by (*intro mat-row-carrierI*)
  {
    assume *fi*: *inj-on f ?U*
    from *inj-on-nat-permutes*[*OF fi*] *f*
    have *f permutes ?U* **by** *auto*
    with *fPU* **have** *False* **by** *simp*
  }
  hence *fni*: ¬ *inj-on f ?U* **by** *auto*
  then obtain *i j* **where** *ij*: *f i* = *f j i* ≠ *j i* < *n j* < *n*
    unfolding *inj-on-def* **by** *auto*
  from *ij*
  have *rth*: *row ?B i* = *row ?B j* **by** *auto*
  have *det ?A* = *0*
    by (*subst det-rows-mul*, *unfold det-identical-rows*[*OF B′ ij*(*2*−*4*) *rth*], *insert f A B*, *auto*)
}
then have *zth*: ⋀ *f*. *f* ∈ *?F* − *?PU* ⟹ *det* (*mat*$_r$ *n n* (λ *i*. *A* $$ (*i*, *f i*) $\cdot_v$ *row B* (*f i*))) = *0*
  by *simp*
{
  fix *p*
  assume *pU*: *p* ∈ *?PU*
  from *pU* **have** *p*: *p permutes ?U*
    by *blast*

let *?s* = λp. (*signof p*) :: ′*a*
let *?f* = λq. *?s p* ∗ (∏ *i*∈ *?U*. *A* $$ (*i*,*p i*)) ∗ (*?s q* ∗ (∏ *i*∈ *?U*. *B* $$ (*i*, *q i*)))
have (*sum* (λq. *?s q* ∗
(∏ *i*∈ *?U*. mat_r *n n* (λ *i*. *A* $$ (*i*, *p i*) $\cdot_v$ *row B* (*p i*)) $$ (*i*, *q i*))) *?PU*) =
(*sum* (λq. *?s p* ∗ (∏ *i*∈ *?U*. *A* $$ (*i*,*p i*)) ∗ (*?s q* ∗ (∏ *i*∈ *?U*. *B* $$ (*i*, *q i*)))) *?PU*)
unfolding *sum-permutations-compose-right*[*OF permutes-inv*[*OF p*], *of ?f*]
proof (*rule sum.cong*[*OF refl*])
fix *q*
assume *q* ∈ {*q*. *q permutes ?U*}
hence *q*: *q permutes ?U* **by** *simp*
from *p q* **have** *pp*: *permutation p* **and** *pq*: *permutation q*
unfolding *permutation-permutes* **by** *auto*
note *sign* = *signof-compose*[*OF q permutes-inv*[*OF p*], *unfolded signof-inv*[*OF fU p*]]
let *?inv* = *Hilbert-Choice.inv*
have *th001*: *prod* (λi. *B*$$ (*i*, *q* (*?inv p i*))) *?U* = *prod* ((λi. *B*$$ (*i*, *q* (*?inv p i*))) ∘ *p*) *?U*
by (*rule prod.permute*[*OF p*])
have *thp*: *prod* (λi. mat_r *n n* (λ *i*. *A*$$(*i*,*p i*) $\cdot_v$ *row B* (*p i*)) $$ (*i*, *q i*)) *?U* =
prod (λi. *A*$$(*i*,*p i*)) *?U* ∗ *prod* (λi. *B*$$ (*i*, *q* (*?inv p i*))) *?U*
unfolding *th001 o-def permutes-inverses*[*OF p*]
by (*subst prod.distrib*[*symmetric*], *insert A p q B*, *auto intro*: *prod.cong*)
define *AA* **where** *AA* = (∏ *i*∈*?U*. *A* $$ (*i*, *p i*))
define *BB* **where** *BB* = (∏ *ia*∈{*0*..<*n*}. *B* $$ (*ia*, *q* (*?inv p ia*)))
have *?s q* ∗ (∏ *ia*∈{*0*..<*n*}. mat_r *n n* (λi. *A* $$ (*i*, *p i*) $\cdot_v$ *row B* (*p i*)) $$ (*ia*, *q ia*)) =
?s p ∗ (∏ *i*∈{*0*..<*n*}. *A* $$ (*i*, *p i*)) ∗ (*?s* (*q* ∘ *?inv p*) ∗ (∏ *ia*∈{*0*..<*n*}. *B* $$ (*ia*, *q* (*?inv p ia*))))
unfolding *sign thp*
unfolding *AA-def*[*symmetric*] *BB-def*[*symmetric*]
by (*simp add*: *ac-simps flip*: *of-int-mult*)
thus *?s q* ∗ (∏ *i* = *0*..<*n*. mat_r *n n* (λi. *A* $$ (*i*, *p i*) $\cdot_v$ *row B* (*p i*)) $$ (*i*, *q i*)) =
?s p ∗ (∏ *i* = *0*..<*n*. *A* $$ (*i*, *p i*)) ∗
(*?s* (*q* ∘ *?inv p*) ∗ (∏ *i* = *0*..<*n*. *B* $$ (*i*, (*q* ∘ *?inv p*) *i*))) **by** *simp*
qed
} **note** ∗ = *this*
have *th2*: *sum* (λf. *det* (mat_r *n n* (λ *i*. *A*$$(*i*,*f i*) $\cdot_v$ *row B* (*f i*)))) *?PU* = *det A* ∗ *det B*
unfolding *det-def*′[*OF A*] *det-def*′[*OF B*] *det-def*′[*OF mat-row-carrierI*]
unfolding *sum-product dim-row-mat*
by (*rule sum.cong*, *insert A*, *force*, *subst* ∗, *insert A B*, *auto*)
let *?f* = λ *f*. *det* (mat_r *n n* (λ *i*. *A* $$ (*i*, *f i*) $\cdot_v$ *row B* (*f i*)))
have *det* (*A* ∗ *B*) = *sum ?f ?F*
unfolding *mat-mul-finsum-alt*[*OF A B*]
by (*rule det-linear-rows-sum*[*OF fU*], *insert A B*, *auto*)
also have ... = *sum ?f* ((*?F* − *?PU*) ∪ (*?F* ∩ *?PU*))

by (*rule arg-cong*[**where** $f = sum\ ?f$], *auto*)
also have ... $=$ *sum ?f* $(?F - ?PU) +$ *sum ?f* $(?F \cap ?PU)$
by (*rule sum.union-disjoint*, *insert A B finite-bounded-functions*[*OF fU fU*], *auto*)
also have *sum ?f* $(?F - ?PU) = 0$
by (*rule sum.neutral*, *insert zth*, *auto*)
also have $?F \cap ?PU = ?PU$ **unfolding** *permutes-def* **by** *fastforce*
also have *sum ?f ?PU* $=$ *det A* $*$ *det B*
unfolding *th2* **..**
finally show *?thesis* **by** *simp*
qed

lemma *unit-imp-det-non-zero*: **assumes** $A \in$ *Units* (*ring-mat TYPE*('*a* :: *comm-ring-1*) *n b*)
shows *det A* $\neq 0$
proof $-$
from *assms*[*unfolded Units-def ring-mat-def*]
obtain *B* **where** *A*: $A \in$ *carrier-mat n n* **and** *B*: $B \in$ *carrier-mat n n* **and** *BA*: $B * A = 1_m\ n$ **by** *auto*
from *arg-cong*[*OF BA*, *of det*, *unfolded det-mult*[*OF B A*] *det-one*]
show *?thesis* **by** *auto*
qed

The following proof is based on the Gauss-Jordan algorithm.

lemma *det-non-zero-imp-unit*: **assumes** *A*: $A \in$ *carrier-mat n n*
and *dA*: *det A* $\neq$ (*0* :: '*a* :: *field*)
shows $A \in$ *Units* (*ring-mat TYPE*('*a*) *n b*)
proof (*rule ccontr*)
let *?g* $=$ *gauss-jordan A* $(0_m\ n\ 0)$
let *?B* $=$ *fst ?g*
obtain *B C* **where** *B*: *?g* $= (B,C)$ **by** (*cases ?g*)
assume $\neg$ *?thesis*
from *this*[*unfolded gauss-jordan-check-invertable*[*OF A zero-carrier-mat*[*of n 0*]] *B*]
have $B \neq 1_m\ n$ **by** *auto*
with *row-echelon-form-imp-1-or-0-row*[*OF gauss-jordan-carrier*(*1*)[*OF A - B*] *gauss-jordan-row-echelon*[*OF A B*], *of 0*]
have *n*: $0 < n$ **and** *row*: *row B* $(n - 1) = 0_v\ n$ **by** *auto*
let $?n = n - 1$
from *n* **have** *n1*: $?n < n$ **by** *auto*
from *gauss-jordan-transform*[*OF A - B*, *of 0 b*] **obtain** *P*
where *P*: *P*$\in$*Units* (*ring-mat TYPE*('*a*) *n b*) **and** *PA*: $B = P * A$ **by** *auto*
from *unit-imp-det-non-zero*[*OF P*] **have** *dP*: *det P* $\neq 0$ **by** *auto*
from *P* **have** *P*: $P \in$ *carrier-mat n n* **unfolding** *Units-def ring-mat-def* **by** *auto*
from *det-mult*[*OF P A*] *dP dA* **have** *det B* $\neq 0$ **unfolding** *PA* **by** *simp*
also have *det B* $= 0$
proof $-$
from *gauss-jordan-carrier*[*OF A - B*, *of 0*] **have** *B*: $B \in$ *carrier-mat n n* **by** *auto*

{
fix j
assume j: $j < n$
from *index-row*(*1*)[*symmetric*, *of ?n B j*, *unfolded row*] B
have B $$ (*?n*, j) = *0* **using** B n j **by** *auto*
}
hence $B = mat_r\ n\ n\ (\lambda i.\ if\ i = ?n\ then\ 0_v\ n\ else\ row\ B\ i)$
by (*intro eq-matI*, *insert B*, *auto*)
also have *det* ... = *0*
by (*rule det-row-0*[*OF n1*], *insert B*, *auto*)
finally show *det B* = *0* **.**
qed
finally show *False* **by** *simp*
qed

lemma *mat-mult-left-right-inverse*: **assumes** A: (A :: $'a$ :: *field mat*) $\in$ *carrier-mat* n n
and B: $B \in$ *carrier-mat* n n **and** *AB*: $A * B = 1_m\ n$
shows $B * A = 1_m\ n$
proof $-$
let *?R* = *ring-mat TYPE*($'a$) *n undefined*
from *det-mult*[*OF A B*, *unfolded AB*] **have** *det A* $\neq$ *0* *det B* $\neq$ *0* **by** *auto*
from *det-non-zero-imp-unit*[*OF A this*(*1*)] *det-non-zero-imp-unit*[*OF B this*(*2*)]

have U: $A \in$ *Units ?R* $B \in$ *Units ?R* **.**
interpret *ring ?R* **by** (*rule ring-mat*)
from *Units-inv-comm*[*unfolded ring-mat-simps*, *OF AB U*] **show** *?thesis* **.**
qed

lemma *det-zero-imp-zero-row*: **assumes** A: (A :: $'a$ :: *field mat*) $\in$ *carrier-mat* n n
and *det*: *det A* = *0*
shows $\exists$ P. $P \in$ *Units* (*ring-mat TYPE*($'a$) n b) $\wedge$ *row* ($P * A$) ($n - 1$) = 0_v $n \wedge 0 < n$
$\wedge$ *row-echelon-form* ($P * A$)
proof $-$
let *?R* = *ring-mat TYPE*($'a$) n b
let *?U* = *Units ?R*
interpret m: *ring ?R* **by** (*rule ring-mat*)
let *?g* = *gauss-jordan A A*
obtain A' B' **where** g: *?g* = (A', B') **by** (*cases ?g*)
from *det unit-imp-det-non-zero*[*of A n b*] **have** *AU*: $A \notin$ *?U* **by** *auto*
with *gauss-jordan-inverse-one-direction*(*1*)[*OF A A*, *of - b*]
have $A'1$: $A' \neq 1_m\ n$ **using** g **by** *auto*
from *gauss-jordan-carrier*(*1*)[*OF A A g*] **have** A': $A' \in$ *carrier-mat n n* **by** *auto*
from *gauss-jordan-row-echelon*[*OF A g*] **have** *re*: *row-echelon-form* A' **.**
from *row-echelon-form-imp-1-or-0-row*[*OF* A' *this*] $A'1$
have n: $0 < n$ **and** *row*: *row* A' ($n - 1$) = 0_v n **by** *auto*
from *gauss-jordan-transform*[*OF A A g*, *of b*] **obtain** P

where P: $P \in$ *?U* **and** A': $A' = P * A$ **by** *auto*
thus *?thesis* **using** *n row re* **by** *auto*
qed

lemma *det-0-iff-vec-prod-zero-field*: **assumes** A: (A :: $'a$ :: *field mat*) $\in$ *carrier-mat* n n
shows *det* $A = 0 \longleftrightarrow (\exists\ v.\ v \in$ *carrier-vec* $n \wedge v \neq 0_v\ n \wedge A *_v v = 0_v\ n)$ (**is** *?l* = ($\exists$ v. *?P* v))
proof −
let *?R* = *ring-mat TYPE*($'a$) n ()
let *?U* = *Units ?R*
interpret m: *ring ?R* **by** (*rule ring-mat*)
show *?thesis*
proof (*cases det* $A = 0$)
case *False*
from *det-non-zero-imp-unit*[*OF A this*, *of* ()]
have $A \in$ *?U* **.**
then obtain B **where** *unit*: $B * A = 1_m\ n$ **and** B: $B \in$ *carrier-mat* n n
unfolding *Units-def ring-mat-def* **by** *auto*
{
fix v
assume *?P* v
hence v: $v \in$ *carrier-vec* n $v \neq 0_v\ n$ $A *_v v = 0_v\ n$ **by** *auto*
have $v = (B * A) *_v v$ **using** v B **unfolding** *unit* **by** *auto*
also have $\ldots = B *_v (A *_v v)$ **using** B A v **by** *simp*
also have $\ldots = B *_v 0_v\ n$ **unfolding** v **..**
also have $\ldots = 0_v\ n$ **using** B **by** *auto*
finally have *False* **using** v **by** *simp*
}
with *False* **show** *?thesis* **by** *blast*
next
case *True*
let *?n* = $n - 1$
from *det-zero-imp-zero-row*[*OF A True*, *of* ()]
obtain P **where** *PU*: $P \in$ *?U* **and** *row*: *row* ($P * A$) *?n* $= 0_v\ n$ **and** n: $0 <$ n *?n* $< n$
and *re*: *row-echelon-form* ($P * A$) **by** *auto*
define *PA* **where** *PA* = $P * A$
note *row* = *row*[*folded PA-def*]
note *re* = *re*[*folded PA-def*]
from *PU* **obtain** Q **where** P: $P \in$ *carrier-mat* n n **and** Q: $Q \in$ *carrier-mat* n n
and *unit*: $Q * P = 1_m\ n$ $P * Q = 1_m\ n$ **unfolding** *Units-def ring-mat-def* **by** *auto*
from P A **have** *PA*: *PA* $\in$ *carrier-mat* n n **and** *dimPA*: *dim-row PA* $= n$ **unfolding** *PA-def* **by** *auto*
from *re*[*unfolded row-echelon-form-def*] **obtain** p **where** p: *pivot-fun PA* p n **using** *PA* **by** *auto*
note *piv* = *pivot-positions*[*OF PA p*]

note *pivot* = *pivot-funD*[*OF dimPA p n*(*2*)]
{
assume *p ?n* < *n*
with *pivot*(*4*)[*OF this*] *n arg-cong*[*OF row*, *of* λ *v*. *v* $ *p ?n*] **have** *False* **using** *PA* **by** *auto*
}
with *pivot*(*1*) **have** *pn*: *p ?n* = *n* **by** *fastforce*
with *piv*(*1*) **have** *set* (*pivot-positions PA*) ⊆ {(*i*, *p i*) |*i*. *i* < *n* ∧ *p i* ≠ *n*} − {(*?n*,*p ?n*)} **by** *auto*
also have … ⊆ {(*i*, *p i*) | *i*. *i* < *?n*} **using** *n* **by** *force*
finally have *card* (*set* (*pivot-positions PA*)) ≤ *card* {(*i*, *p i*) | *i*. *i* < *?n*}
by (*intro card-mono*, *auto*)
also have {(*i*, *p i*) | *i*. *i* < *?n*} = (λ *i*. (*i*, *p i*)) ‘ {*0* ..< *?n*} **by** *auto*
also have *card* … = *card* {*0* ..< *?n*} **by** (*rule card-image*, *auto simp*: *inj-on-def*)
also have … < *n* **using** *n* **by** *simp*
finally have *card* (*set* (*pivot-positions PA*)) < *n* **.**
hence *card* (*snd* ‘ (*set* (*pivot-positions PA*))) < *n*
using *card-image-le*[*OF finite-set*, *of snd pivot-positions PA*] **by** *auto*
hence *neq*: *snd* ‘ (*set* (*pivot-positions PA*)) ≠ {*0* ..< *n*} **by** *auto*
from *find-base-vector*[*OF re PA neq*] **obtain** *v* **where** *v*: *v* ∈ *carrier-vec n*
and *v0*: *v* ≠ 0_v *n* **and** *pav*: *PA* $*_v$ *v* = 0_v *n* **by** *auto*
have *A* $*_v$ *v* = *Q* ∗ *P* $*_v$ (*A* $*_v$ *v*) **unfolding** *unit* **using** *A v* **by** *auto*
also have … = *Q* $*_v$ (*PA* $*_v$ *v*) **unfolding** *PA-def* **using** *Q P A v* **by** *auto*
also have *PA* $*_v$ *v* = 0_v *n* **unfolding** *pav* **..**
also have *Q* $*_v$ 0_v *n* = 0_v *n* **using** *Q* **by** *auto*
finally have *Av*: *A* $*_v$ *v* = 0_v *n* **by** *auto*
show *?thesis* **unfolding** *True* **using** *Av v0 v* **by** *auto*
qed
qed

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* :: *idom mat*) ∈ *carrier-mat n n*
shows *det A* = *0* ⟷ (∃ *v*. *v* ∈ *carrier-vec n* ∧ *v* ≠ 0_v *n* ∧ *A* $*_v$ *v* = 0_v *n*)
proof −
let *?h* = *to-fract* :: ′*a* ⇒ ′*a fract*
let *?A* = *map-mat ?h A*
have *A*′: *?A* ∈ *carrier-mat n n* **using** *A* **by** *auto*
interpret *inj-comm-ring-hom ?h* **by** (*unfold-locales*, *auto*)
have (*det A* = *0*) = (*?h* (*det A*) = *?h 0*) **by** *auto*
also have … = (*det ?A* = *0*) **unfolding** *hom-zero hom-det* **..**
also have … = ((∃ *v*. *v* ∈ *carrier-vec n* ∧ *v* ≠ 0_v *n* ∧ *?A* $*_v$ *v* = 0_v *n*))
unfolding *det-0-iff-vec-prod-zero-field*[*OF A*′] **..**
also have … = ((∃ *v*. *v* ∈ *carrier-vec n* ∧ *v* ≠ 0_v *n* ∧ *A* $*_v$ *v* = 0_v *n*)) (**is** *?l* = *?r*)
proof
assume *?r*
then obtain *v* **where** *v*: *v* ∈ *carrier-vec n v* ≠ 0_v *n A* $*_v$ *v* = 0_v *n* **by** *auto*

show *?l*
by (*rule exI*[*of - map-vec ?h v*], *insert v*, *auto simp*: *mult-mat-vec-hom*[*symmetric*, *OF A v*(*1*)])
next
assume *?l*
then obtain v **where** v: $v \in$ *carrier-vec* n **and** *v0*: $v \neq 0_v\ n$ **and** *Av*: $?A *_v v = 0_v\ n$ **by** *auto*
have $\forall\ i.\ \exists\ a\ b.\ v\ \$\ i =$ *Fraction-Field.Fract* $a\ b \wedge b \neq 0$ **using** *Fract-cases*[*of* $v\ \$\ i$ **for** *i*] **by** *metis*
from *choice*[*OF this*] **obtain** a **where** $\forall\ i.\ \exists\ b.\ v\ \$\ i =$ *Fraction-Field.Fract* $(a\ i)\ b \wedge b \neq 0$ **by** *metis*
from *choice*[*OF this*] **obtain** b **where** *vi*: $\bigwedge i.\ v\ \$\ i =$ *Fraction-Field.Fract* $(a\ i)\ (b\ i)$ **and** *bi*: $\bigwedge i.\ b\ i \neq 0$ **by** *auto*
define m **where** $m =$ *prod-list* (*map* b [*0*..<*n*])
let $?m = ?h\ m$
have *m0*: $m \neq 0$ **unfolding** *m-def hom-0-iff prod-list-zero-iff* **using** *bi* **by** *auto*
from *v0*[*unfolded vec-eq-iff*] v **obtain** i **where** i: $i < n\ v\ \$\ i \neq 0$ **by** *auto*
{
fix *i*
assume $i < n$
hence $b\ i \in$ *set* (*map* b [*0* ..< *n*]) **by** *auto*
from *split-list*[*OF this*]
obtain *ys zs* **where** *map* b [*0*..<*n*] $=$ *ys* @ $b\ i$ # *zs* **by** *auto*
hence $b\ i$ *dvd* m **unfolding** *m-def* **by** *auto*
then obtain c **where** $m = b\ i * c$ **..**
hence $?m * v\ \$\ i = ?h\ (a\ i * c)$ **unfolding** *vi* **using** *bi*[*of i*]
by (*simp add*: *eq-fract to-fract-def*)
hence $\exists\ c.\ ?m * v\ \$\ i = ?h\ c$ **..**
}
hence $\forall\ i.\ \exists\ c.\ i < n \longrightarrow ?m * v\ \$\ i = ?h\ c$ **by** *auto*
from *choice*[*OF this*] **obtain** c **where** c: $\bigwedge i.\ i < n \Longrightarrow ?m * v\ \$\ i = ?h\ (c\ i)$ **by** *auto*
define w **where** $w =$ *vec* $n\ c$
have w: $w \in$ *carrier-vec* n **unfolding** *w-def* **by** *simp*
have *mvw*: $?m \cdot_v v =$ *map-vec* $?h\ w$ **unfolding** *w-def* **using** $c\ v$
by (*intro eq-vecI*, *auto*)
with *m0* i c[*OF* i(*1*)] **have** $w\ \$\ i \neq 0$ **unfolding** *w-def* **by** *auto*
with $i\ w$ **have** *w0*: $w \neq 0_v\ n$ **by** *auto*
from *arg-cong*[*OF Av*, *of* $\lambda\ v.\ ?m \cdot_v v$]
have $?m \cdot_v (?A *_v v) =$ *map-vec* $?h\ (0_v\ n)$ **by** *auto*
also have $?m \cdot_v (?A *_v v) = ?A *_v (?m \cdot_v v)$ **using** $A\ v$ **by** *auto*
also have $\ldots = ?A *_v$ (*map-vec* $?h\ w$) **unfolding** *mvw* **..**
also have $\ldots =$ *map-vec* $?h\ (A *_v w)$ **unfolding** *mult-mat-vec-hom*[*OF A w*] **..**
finally have $A *_v w = 0_v\ n$ **by** (*rule vec-hom-inj*)
with w *w0* **show** *?r* **by** *blast*
qed
finally show *?thesis* **.**
qed

lemma *det-0-negate*: **assumes** *A*: $(A :: 'a :: field\ mat) \in carrier\text{-}mat\ n\ n$
shows $(det\ (-\ A) = 0) = (det\ A = 0)$
proof −
from *A* **have** *mA*: $-\ A \in carrier\text{-}mat\ n\ n$ **by** *auto*
{
fix $v :: 'a\ vec$
assume *v*: $v \in carrier\text{-}vec\ n$
hence *Av*: $A *_v v \in carrier\text{-}vec\ n$ **using** *A* **by** *auto*
have *id*: $-\ A *_v v = -\ (A *_v v)$ **using** *v A* **by** *simp*
have $(-\ A *_v v = 0_v\ n) = (A *_v v = 0_v\ n)$ **unfolding** *id*
unfolding *uminus-zero-vec-eq*[*OF Av*] **..**
}
thus *?thesis* **unfolding** *det-0-iff-vec-prod-zero*[*OF A*] *det-0-iff-vec-prod-zero*[*OF mA*] **by** *auto*
qed

lemma *det-multrow*:
assumes *k*: $k < n$ **and** *A*: $A \in carrier\text{-}mat\ n\ n$
shows $det\ (multrow\ k\ a\ A) = a * det\ A$
proof −
have $multrow\ k\ a\ A = multrow\text{-}mat\ n\ k\ a * A$
by (*rule multrow-mat*[*OF A*])
also have $det\ (multrow\text{-}mat\ n\ k\ a * A) = det\ (multrow\text{-}mat\ n\ k\ a) * det\ A$
by (*rule det-mult*[*OF - A*], *auto*)
also have $det\ (multrow\text{-}mat\ n\ k\ a) = a$
by (*rule det-multrow-mat*[*OF k*])
finally show *?thesis* **.**
qed

lemma *det-multrow-div*:
assumes *k*: $k < n$ **and** *A*: $A \in carrier\text{-}mat\ n\ n$ **and** *a0*: $a \neq 0$
shows $det\ (multrow\ k\ a\ A :: 'a :: idom\text{-}divide\ mat)\ div\ a = det\ A$
proof −
have $det\ (multrow\ k\ a\ A)\ div\ a = a * det\ A\ div\ a$ **using** *k A*
by (*simp add*: *det-multrow*)
also have ... $= det\ A$ **using** *a0* **by** *auto*
finally show *?thesis***.**
qed

lemma *det-addrow*:
assumes *l*: $l < n$ **and** *k*: $k \neq l$ **and** *A*: $A \in carrier\text{-}mat\ n\ n$
shows $det\ (addrow\ a\ k\ l\ A) = det\ A$
proof −
have $addrow\ a\ k\ l\ A = addrow\text{-}mat\ n\ a\ k\ l * A$
by (*rule addrow-mat*[*OF A l*])
also have $det\ (addrow\text{-}mat\ n\ a\ k\ l * A) = det\ (addrow\text{-}mat\ n\ a\ k\ l) * det\ A$
by (*rule det-mult*[*OF - A*], *auto*)
also have $det\ (addrow\text{-}mat\ n\ a\ k\ l) = 1$

by (*rule det-addrow-mat*[*OF k*])
finally show *?thesis* **using** *A* **by** *simp*
qed

lemma *det-swaprows*:
assumes $*$: $k < n$ $l < n$ **and** *k*: $k \neq l$ **and** *A*: $A \in$ *carrier-mat n n*
shows *det* (*swaprows k l A*) $= -$ *det A*
proof $-$
have *swaprows k l A* = *swaprows-mat n k l* $*$ *A*
by (*rule swaprows-mat*[*OF A* $*$])
also have *det* (*swaprows-mat n k l* $*$ *A*) = *det* (*swaprows-mat n k l*) $*$ *det A*
by (*rule det-mult*[*OF - A*], *insert A*, *auto*)
also have *det* (*swaprows-mat n k l*) = $-$ *1*
by (*rule det-swaprows-mat*[*OF* $*$ *k*])
finally show *?thesis* **using** *A* **by** *simp*
qed

lemma *det-similar*: **assumes** *similar-mat A B*
shows *det A* = *det B*
proof $-$
from *similar-matD*[*OF assms*] **obtain** *n P Q* **where**
carr: {*A*, *B*, *P*, *Q*} $\subseteq$ *carrier-mat n n* (**is** - $\subseteq$ *?C*)
and *PQ*: *P* $*$ *Q* = 1_m *n*
and *AB*: *A* = *P* $*$ *B* $*$ *Q* **by** *blast*
hence *A*: $A \in$ *?C* **and** *B*: $B \in$ *?C* **and** *P*: $P \in$ *?C* **and** *Q*: $Q \in$ *?C* **by** *auto*
from *det-mult*[*OF P Q*, *unfolded PQ*] **have** *PQ*: *det P* $*$ *det Q* = *1* **by** *auto*
from *det-mult*[*OF - Q*, *of P* $*$ *B*, *unfolded det-mult*[*OF P B*] *AB*[*symmetric*]] *P B*
have *det A* = *det P* $*$ *det B* $*$ *det Q* **by** *auto*
also have ... = (*det P* $*$ *det Q*) $*$ *det B* **by** (*simp add*: *ac-simps*)
also have ... = *det B* **unfolding** *PQ* **by** *simp*
finally show *?thesis* **.**
qed

lemma *det-four-block-mat-upper-right-zero-col*: **assumes** *A1*: *A1* $\in$ *carrier-mat n n*
and *A20*: *A2* = (0_m *n 1*) **and** *A3*: *A3* $\in$ *carrier-mat 1 n*
and *A4*: *A4* $\in$ *carrier-mat 1 1*
shows *det* (*four-block-mat A1 A2 A3 A4*) = *det A1* $*$ *det A4* (**is** *det ?A* = -)
proof $-$
let *?A* = *four-block-mat A1 A2 A3 A4*
from *A20* **have** *A2*: *A2* $\in$ *carrier-mat n 1* **by** *auto*
define *A* **where** *A* = *?A*
from *four-block-carrier-mat*[*OF A1 A4*] *A1*
have *A*: $A \in$ *carrier-mat* (*Suc n*) (*Suc n*) **and** *dim*: *dim-row A1* = *n* **unfolding** *A-def* **by** *auto*
let *?Pn* = λ *p*. *p permutes* {*0* ..< *n*}
let *?Psn* = λ *p*. *p permutes* {*0* ..< *Suc n*}
let *?perm* = {*p*. *?Psn p*}

let *?permn* = {*p*. *?Pn p*}
let *?prod* = *λ p. signof p* ∗ ($\prod$ *i* = *0..<Suc n. A* $$ (*p i*, *i*))
let *?prod′* = *λ p. A* $$ (*p n*, *n*) ∗ *signof p* ∗ ($\prod$ *i* = *0..<n. A* $$ (*p i*, *i*))
let *?prod′′* = *λ p. signof p* ∗ ($\prod$ *i* = *0..< n. A* $$ (*p i*, *i*))
let *?prod′′′* = *λ p. signof p* ∗ ($\prod$ *i* = *0..< n. A1* $$ (*p i*, *i*))
let *?p0* = {*p. p 0* = *0*}
have [*simp*]: {*0..<Suc n*} − {*n*} = {*0..< n*} **by** *auto*
{
fix *p*
assume *?Psn p*
have *?prod p* = *signof p* ∗ (*A* $$ (*p n*, *n*) ∗ ($\prod$ *i* ∈ {*0..< n*}. *A* $$ (*p i*, *i*)))
by (*subst prod.remove*[*of - n*], *auto*)
also have . . . = *A* $$ (*p n*, *n*) ∗ *signof p* ∗ ($\prod$ *i* ∈ {*0..< n*}. *A* $$ (*p i*, *i*)) **by**
simp
finally have *?prod p* = *?prod′ p* **.**
} **note** *prod-id* = *this*
define *prod′* **where** *prod′* = *?prod′*
{
fix *i q*
assume *i*: *i* ∈ {*0..< n*} *q permutes* {*0 ..< n*}
hence *Fun.swap n i id* (*q n*) < *n*
unfolding *permutes-def* **by** *auto*
hence *A* $$ (*Fun.swap n i id* (*q n*), *n*) = *0*
unfolding *A-def* **using** *A1 A20 A3 A4* **by** *auto*
hence *prod′* (*Fun.swap n i id* ∘ *q*) = *0*
unfolding *prod′-def* **by** *simp*
} **note** *zero* = *this*
have *cong*: $\bigwedge$ *a b c. b* = *c* ⟹ *a* ∗ *b* = *a* ∗ *c* **by** *auto*
have *det ?A* = *sum ?prod ?perm*
unfolding *A-def*[*symmetric*] **using** *mat-det-left-def*[*OF A*] *A* **by** *simp*
also have . . . = *sum prod′ ?perm* **unfolding** *prod′-def*
by (*rule sum.cong*[*OF refl*], *insert prod-id*, *auto*)
also have {*0 ..< Suc n*} = *insert n* {*0 ..< n*} **by** *auto*
also have *sum prod′* {*p. p permutes* . . .} =
($\sum$ *i*∈*insert n* {*0..<n*}. $\sum$ *q*∈*?permn. prod′* (*Fun.swap n i id* ∘ *q*))
by (*subst sum-over-permutations-insert*, *auto*)
also have . . . = ($\sum$ *q*∈*?permn. prod′ q*) +
($\sum$ *i*∈{*0..<n*}. $\sum$ *q*∈*?permn. prod′* (*Fun.swap n i id* ∘ *q*))
by (*subst sum.insert*, *auto*)
also have ($\sum$ *i*∈{*0..<n*}. $\sum$ *q*∈*?permn. prod′* (*Fun.swap n i id* ∘ *q*)) = *0*
by (*rule sum.neutral*, *intro ballI*, *rule sum.neutral*, *intro ballI*, *rule zero*, *auto*)
also have ($\sum$ *q*∈ *?permn. prod′ q*) = *A* $$ (*n*,*n*) ∗ ($\sum$ *q*∈ *?permn. ?prod′′ q*)
unfolding *prod′-def*
by (*subst sum-distrib-left*, *rule sum.cong*[*OF refl*], *auto simp*: *permutes-def*
ac-simps)
also have *A* $$ (*n*,*n*) = *A4* $$ (*0*,*0*) **unfolding** *A-def* **using** *A1 A2 A3 A4* **by**
auto
also have ($\sum$ *q*∈ *?permn. ?prod′′ q*) = ($\sum$ *q*∈ *?permn. ?prod′′′ q*)
by (*rule sum.cong*[*OF refl*], *rule cong*, *rule prod.cong*,

insert A1 A2 A3 A4, *auto simp*: *permutes-def A-def*)
also have ... = *det A1*
unfolding *mat-det-left-def*[*OF A1*] *dim* **by** *auto*
also have *A4* $$ (*0*,*0*) = *det A4*
using *A4* **unfolding** *det-def*[*of A4*] **by** (*auto simp*: *sign-def*)
finally show *?thesis* **by** *simp*
qed

lemma *det-swap-initial-rows*: **assumes** *A*: $A \in$ *carrier-mat m m*
and *lt*: $k + n \leq m$
shows *det A* = (− *1*) ^ (*k* ∗ *n*) ∗
det (*mat m m* (λ(*i*, *j*). *A* $$ (*if i* < *n then i* + *k else if i* < *k* + *n then i* − *n else i*, *j*)))
proof −
define *sw* **where** *sw* = (λ (*A* :: ′*a mat*) *xs*. *fold* (λ (*i*,*j*). *swaprows i j*) *xs A*)
have *dim-sw*[*simp*]: *dim-row* (*sw A xs*) = *dim-row A dim-col* (*sw A xs*) = *dim-col A* **for** *xs A*
unfolding *sw-def* **by** (*induct xs arbitrary*: *A*, *auto*)
{
fix *xs* **and** *A* :: ′*a mat*
assume *dim-row A* = *dim-col A* ⋀ *i j*. (*i*,*j*) ∈ *set xs* ⟹ *i* < *dim-col A* ∧ *j* < *dim-col A* ∧ *i* ≠ *j*
hence *det* (*sw A xs*) = (−*1*)^(*length xs*) ∗ *det A*
unfolding *sw-def*
proof (*induct xs arbitrary*: *A*)
case (*Cons xy xs A*)
obtain *x y* **where** *xy*: *xy* = (*x*,*y*) **by** *force*
from *Cons*(*3*)[*unfolded xy*, *of x y*] *Cons*(*2*)
have [*simp*]: *det* (*swaprows x y A*) = − *det A*
by (*intro det-swaprows*, *auto*)
show *?case* **unfolding** *xy* **by** (*simp*, *insert Cons*(*2*−), (*subst Cons*(*1*), *auto*)+)
qed *simp*
} **note** *sw* = *this*
define *swb* **where** *swb* = (λ *A i n*. *sw A* (*map* (λ *j*. (*j*,*Suc j*)) [*i* ..< *i* + *n*]))
{
fix *k n* **and** *A* :: ′*a mat*
assume *k-n*: *k* + *n* < *dim-row A*
hence *swb A k n* = *mat* (*dim-row A*) (*dim-col A*) (λ (*i*,*j*). *let r* =
(*if i* < *k* ∨ *i* > *k* + *n then i else if i* = *k* + *n then k else Suc i*)
in A $$ (*r*,*j*))
proof (*induct n*)
case *0*
show *?case* **unfolding** *swb-def sw-def* **by** (*rule eq-matI*, *auto*)
next
case (*Suc n*)
hence *dim*: *k* + *n* < *dim-row A* **by** *auto*
have *id*: *swb A k* (*Suc n*) = *swaprows* (*k* + *n*) (*Suc k* + *n*) (*swb A k n*) **unfolding** *swb-def sw-def* **by** *simp*
show *?case* **unfolding** *id Suc*(*1*)[*OF dim*]

```
      by (rule eq-matI, insert Suc(2), auto)
    qed
  } note swb = this
  define swbl where swbl = (λ A k n. fold (λ i A. swb A i n) (rev [0 ..< k]) A)
  {
    fix k n and A :: 'a mat
    assume k-n: k + n ≤ dim-row A
    hence swbl A k n = mat (dim-row A) (dim-col A) (λ (i,j). let r =
      (if i < n then i + k else if i < k + n then i − n else i)
      in A $$ (r,j))
    proof (induct k arbitrary: A)
      case 0
      thus ?case unfolding swbl-def by (intro eq-matI, auto simp: swb)
    next
      case (Suc k)
      hence dim: k + n < dim-row A by auto
       have id: swbl A (Suc k) n = swbl (swb A k n) k n unfolding swbl-def by
simp
      show ?case unfolding id swb[OF dim]
       by (subst Suc(1), insert dim, force, intro eq-matI, auto simp: less-Suc-eq-le)

    qed
  } note swbl = this
  {
    fix k n and A :: 'a mat
    assume k-n: k + n ≤ dim-col A dim-row A = dim-col A
    hence det (swbl A k n) = (−1)^(k*n) * det A
    proof (induct k arbitrary: A)
      case 0
      thus ?case unfolding swbl-def by auto
    next
      case (Suc k)
      hence dim: k + n < dim-row A by auto
       have id: swbl A (Suc k) n = swbl (swb A k n) k n unfolding swbl-def by
simp
      have det: det (swb A k n) = (−1)^n * det A unfolding swb-def
        by (subst sw, insert Suc(2−), auto)
      show ?case unfolding id
       by (subst Suc(1), insert Suc(2−), auto simp: det, auto simp: swb power-add)
    qed
  } note det-swbl = this
  from assms have dim: dim-row A = dim-col A k + n ≤ dim-col A k + n ≤
dim-row A dim-col A = m by auto
  from arg-cong[OF det-swbl[OF dim(2,1), unfolded swbl[OF dim(3)], unfolded
Let-def dim],
      of λ x. (−1)^(k*n) * x]
  show ?thesis by simp
qed
```

lemma *det-swap-rows*: **assumes** *A*: $A \in$ *carrier-mat* $(k + n)$ $(k + n)$
shows *det* $A = (-1)$^$(k * n) *$ *det* (*mat* $(k + n)$ $(k + n)$ $(\lambda\ (i,j).$
A \$\$ ((*if* $i < k$ *then* $i + n$ *else* $i - k$),j)))
proof –
have *le*: $n + k \le k + n$ **by** *simp*
show *?thesis* **unfolding** *det-swap-initial-rows*[*OF A le*]
by (*intro arg-cong2*[*of - - - -* $\lambda\ x\ y.\ ((-1)$^$x *$ *det* y)], *force*, *intro eq-matI*, *auto*)
qed

lemma *det-swap-final-rows*: **assumes** *A*: $A \in$ *carrier-mat m m*
and *m*: $m = l + k + n$
shows *det* $A = (-\ 1)$ ^ $(k * n) *$
det (*mat m m* ($\lambda(i, j)$. A \$\$ (*if* $i < l$ *then* i *else if* $i < l + n$ *then* $i + k$ *else* $i - n$, j)))
(**is** - = - * *det ?M*)
proof –

have *m1*: $m = (k + n) + l$ **using** *m* **by** *simp*
have *m2*: $k + n \le m$ **using** *m* **by** *simp*
have *m3*: $m = l + (n + k)$ **using** *m* **by** *simp*
define *M* **where** *M = ?M*
let *?M1* = *mat m m* ($\lambda(i, j)$. A \$\$ (*if* $i < k + n$ *then* $i + l$ *else* $i - (k + n)$, j))
let *?M2* = *mat m m*
($\lambda(i, j)$. A \$\$ (*if* $i < n$ *then* $i + k + l$ *else if* $i < k + n$ *then* $i - n + l$ *else* $i - (k + n)$, j))
have *M2*: *?M2* $\in$ *carrier-mat m m* **by** *auto*
have *det* $A = (-\ 1)$ ^ $((k + n) * l) *$ *det ?M1*
unfolding *det-swap-rows*[*OF A*[*unfolded m1*]] *m1*[*symmetric*] **by** *simp*
also have *det ?M1* $= (-\ 1)$ ^ $(k * n) *$ *det ?M2*
by (*subst det-swap-initial-rows*[*OF - m2*], *force*, *rule arg-cong*[*of - -* λ *x*. - * *det x*], *rule eq-matI*, *auto simp*: *m*)
also have *det ?M2* $= (-\ 1)$ ^ $(l * (n + k)) *$ *det M*
unfolding *M-def det-swap-rows*[*OF M2*[*unfolded m3*], *folded m3*]
by (*rule arg-cong*[*of - -* λ *x*. - * *det x*], *rule eq-matI*, *auto simp*: *m*)
finally have *det* $A = (-1)$ ^ $((k + n) * l + (k * n) + l * (n + k)) *$ *det M* (**is** - = *?b* ^ - * -)
by (*simp add*: *power-add*)
also have $(k + n) * l + (k * n) + l * (n + k) = 2 * (l * (n + k)) + k * n$ **by** *simp*
also have *?b* ^ ... = *?b* ^ $(k * n)$ **by** (*simp add*: *power-add*)
finally show *?thesis* **unfolding** *M-def* **.**
qed

lemma *det-swap-final-cols*: **assumes** *A*: $A \in$ *carrier-mat m m*
and *m*: $m = l + k + n$
shows *det* $A = (-\ 1)$ ^ $(k * n) *$
det (*mat m m* ($\lambda(i, j)$. A \$\$ (i, *if* $j < l$ *then* j *else if* $j < l + n$ *then* $j + k$ *else*

$j - n$)))
proof −
  have *det A* = *det* (A^T) **unfolding** *det-transpose*[*OF A*] **..**
  also have ... = (− *1*) ^(*k* ∗ *n*) ∗
    det (*mat m m* (λ(*i*, *j*). A^T $$ (*if i* < *l then i else if i* < *l* + *n then i* + *k else i* − *n*, *j*)))
    (**is** - = - ∗ *det ?M*)
    by (*rule det-swap-final-rows*[*OF* - *m*], *insert A*, *auto*)
  also have *det ?M* = *det* (*?*M^T) **by** (*subst det-transpose*, *auto*)
  also have *?*M^T = *mat m m* (λ(*i*, *j*). *A* $$ (*i*, *if j* < *l then j else if j* < *l* + *n then j* + *k else j* − *n*))
    unfolding *transpose-mat-def* **using** *A m*
    by (*intro eq-matI*, *auto*)
  finally show *?thesis* **.**
qed

lemma *det-swap-initial-cols*: **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* $$ (*i*, *if j* < *n then j* + *k else if j* < *k* + *n then j* − *n else j*)))
proof −
  have *det A* = *det* (A^T) **unfolding** *det-transpose*[*OF A*] **..**
  also have ... = (− *1*) ^(*k* ∗ *n*) ∗
    det (*mat m m* (λ(*j*, *i*). A^T $$ (*if j* < *n then j* + *k else if j* < *k* + *n then j* − *n else j*,*i*)))
    (**is** - = - ∗ *det ?M*)
    by (*rule det-swap-initial-rows*[*OF* - *lt*], *insert A*, *auto*)
  also have *det ?M* = *det* (*?*M^T) **by** (*subst det-transpose*, *auto*)
  also have *?*M^T = *mat m m* (λ(*i*, *j*). *A* $$ (*i*, *if j* < *n then j* + *k else if j* < *k* + *n then j* − *n else j*))
    unfolding *transpose-mat-def* **using** *A lt*
    by (*intro eq-matI*, *auto*)
  finally show *?thesis* **.**
qed

lemma *det-swap-cols*: **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 $$ (*i*,(*if j* < *k then j* + *n else j* − *k*)))) (**is** - = - ∗ *det ?B*)
proof −
  have *le*: *n* + *k* ≤ *k* + *n* **by** *simp*
  show *?thesis* **unfolding** *det-swap-initial-cols*[*OF A le*]
    by (*intro arg-cong2*[*of* - - - - λ *x y*. ((−*1*)^*x* ∗ *det y*)], *force*, *intro eq-matI*, *auto*)
qed

lemma *det-four-block-mat-upper-right-zero*: **fixes** *A1* :: *'a* :: *idom mat*
  assumes *A1*: *A1* ∈ *carrier-mat n n*
  and *A20*: *A2* = (0_m *n m*) **and** *A3*: *A3* ∈ *carrier-mat m n*

and *A4*: *A4* ∈ *carrier-mat m m*
shows *det* (*four-block-mat A1 A2 A3 A4*) = *det A1* ∗ *det A4*
using *assms*(*2*−)
proof (*induct m arbitrary*: *A2 A3 A4*)
case (*0 A2 A3 A4*)
hence ∗: *four-block-mat A1 A2 A3 A4* = *A1* **using** *A1*
by (*intro eq-matI*, *auto*)
from *0* **have** *4*: *A4* = 1_m *0* **by** *auto*
show *?case* **unfolding** ∗ **unfolding** *4* **by** *simp*
next
case (*Suc m A2 A3 A4*)
let *?m* = *Suc m*
from *Suc* **have** *A2*: *A2* ∈ *carrier-mat n ?m* **by** *auto*
note *A20* = *Suc*(*2*)
note *A34* = *Suc*(*3*−*4*)
let *?A* = *four-block-mat A1 A2 A3 A4*
let *?P* = λ *B3 B4 v k*. *v* ≠ *0* ∧ *v* ∗ *det ?A* = *det* (*four-block-mat A1 A2 B3 B4*)
∧ *v* ∗ *det A4* = *det B4* ∧ *B3* ∈ *carrier-mat ?m n* ∧ *B4* ∈ *carrier-mat ?m ?m*
∧ (∀ *i* < *k*. *B4* $$ (*i*,*m*) = *0*)
have *k* ≤ *m* ⟹ ∃ *B3 B4 v*. *?P B3 B4 v k* **for** *k*
proof (*induct k*)
case *0*
have *?P A3 A4 1 0* **using** *A34* **by** *auto*
thus *?case* **by** *blast*
next
case (*Suc k*)
then obtain *B3 B4 v* **where** *v*: *v* ≠ *0* **and** *det*: *v* ∗ *det ?A* =
det (*four-block-mat A1 A2 B3 B4*) *v* ∗ *det A4* = *det B4*
and *B3*: *B3* ∈ *carrier-mat ?m n* **and** *B4*: *B4* ∈ *carrier-mat ?m ?m* **and** *0*:
∀ *i* < *k*. *B4* $$ (*i*,*m*) = *0* **by** *auto*
show *?case*
proof (*cases B4* $$ (*k*,*m*) = *0*)
case *True*
with *0* **have** *0*: ∀ *i* < *Suc k*. *B4* $$ (*i*,*m*) = *0* **using** *less-Suc-eq* **by** *auto*
with *v det B3 B4* **have** *?P B3 B4 v* (*Suc k*) **by** *auto*
thus *?thesis* **by** *blast*
next
case *Bk*: *False*
let *?k* = *Suc k*
from *Suc*(*2*) **have** *k*: *k* < *?m Suc k* < *?m k* ≠ *Suc k* **by** *auto*
show *?thesis*
proof (*cases B4* $$ (*?k*,*m*) = *0*)
case *True*
let *?B4* = *swaprows k* (*Suc k*) *B4*
let *?B3* = *swaprows k* (*Suc k*) *B3*
let *?B* = *four-block-mat A1 A2 ?B3 ?B4*
let *?v* = −*v*
from *det-swaprows*[*OF k B4*] *det* **have** *det1*: *?v* ∗ *det A4* = *det ?B4* **by**
simp

from *v* **have** *v*: *?v* ≠ *0* **by** *auto*
from *B3* **have** *B3'*: *?B3* ∈ *carrier-mat ?m n* **by** *auto*
from *B4* **have** *B4'*: *?B4* ∈ *carrier-mat ?m ?m* **by** *auto*
have *?v* ∗ *det ?A* = − *det* (*four-block-mat A1 A2 B3 B4*) **using** *det* **by** *simp*
also have . . . = *det* (*swaprows* (*n* + *k*) (*n* + *?k*) (*four-block-mat A1 A2 B3 B4*))
by (*rule sym*, *rule det-swaprows*[*of - n* + *?m*], *insert A1 A2 B3 B4 k*, *auto*)
also have *swaprows* (*n* + *k*) (*n* + *?k*) (*four-block-mat A1 A2 B3 B4*) = *?B*
proof (*rule eq-matI*, *unfold index-mat-four-block index-mat-swaprows*, *goal-cases*)
case (*1 i j*)
show *?case*
proof (*cases i* < *n*)
case *True*
thus *?thesis* **using** *1*(*2*) *A1 A2 B3 B4* **by** *simp*
next
case *False*
hence *i* = *n* + (*i* − *n*) **by** *simp*
then obtain *d* **where** *i* = *n* + *d* **by** *blast*
thus *?thesis* **using** *1 A1 A2 B3 B4 k*(*2*) **by** *simp*
qed
qed *auto*
finally have *det2*: *?v* ∗ *det ?A* = *det ?B* **.**
from *True 0 B4 k*(*2*) **have** ∀ *i* < *Suc k*. *?B4* $$ (*i*,*m*) = *0* **unfolding** *less-Suc-eq* **by** *auto*
with *det1 det2 B3' B4' v* **have** *?P ?B3 ?B4 ?v* (*Suc k*) **by** *auto*
thus *?thesis* **by** *blast*
next
case *False*
let *?bk* = *B4* $$ (*?k*,*m*)
let *?b* = *B4* $$ (*k*,*m*)
let *?v* = *v* ∗ *?bk*
let *?B3* = *addrow* (− *?b*) *k ?k* (*multrow k ?bk B3*)
let *?B4* = *addrow* (− *?b*) *k ?k* (*multrow k ?bk B4*)
have ∗: *det ?B4* = *?bk* ∗ *det B4*
by (*subst det-addrow*[*OF k*(*2*−*3*)], *force simp*: *B4*, *rule det-multrow*[*OF k*(*1*) *B4*])
with *det*(*2*)[*symmetric*] **have** *det2*: *?v* ∗ *det A4* = *det ?B4* **by** (*auto simp*: *ac-simps*)
from *0 k*(*2*) *B4* **have** *0*: ∀ *i* < *Suc k*. *?B4* $$ (*i*,*m*) = *0* **unfolding** *less-Suc-eq* **by** *auto*
from *False v* **have** *v*: *?v* ≠ *0* **by** *auto*
from *B3* **have** *B3'*: *?B3* ∈ *carrier-mat ?m n* **by** *auto*
from *B4* **have** *B4'*: *?B4* ∈ *carrier-mat ?m ?m* **by** *auto*
let *?B'* = *multrow* (*n* + *k*) *?bk* (*four-block-mat A1 A2 B3 B4*)
have *B'*: *?B'* ∈ *carrier-mat* (*n* + *?m*) (*n* + *?m*) **using** *A1 A2 B3 B4 k* **by** *auto*
let *?B* = *four-block-mat A1 A2 ?B3 ?B4*

have *?v* ∗ *det ?A* = *?bk* ∗ *det* (*four-block-mat A1 A2 B3 B4*) **using** *det* **by** *simp*
also have . . . = *det* (*addrow* (− *?b*) (*n* + *k*) (*n* + *?k*) *?B′*)
by (*subst det-addrow*[*OF* - - *B′*], *insert k*(*2*), *force*, *force*, *rule sym*, *rule det-multrow*[*of* - *n* + *?m*],
insert A1 A2 B3 B4 k, *auto*)
also have *addrow* (− *?b*) (*n* + *k*) (*n* + *?k*) *?B′* = *?B*
proof (*rule eq-matI*, *unfold index-mat-four-block index-mat-multrow index-mat-addrow*, *goal-cases*)
case (*1 i j*)
show *?case*
proof (*cases i* < *n*)
case *True*
thus *?thesis* **using** *1*(*2*) *A1 A2 B3 B4* **by** *simp*
next
case *False*
hence *i* = *n* + (*i* − *n*) **by** *simp*
then obtain *d* **where** *i* = *n* + *d* **by** *blast*
thus *?thesis* **using** *1 A1 A2 B3 B4 k*(*2*) **by** *simp*
qed
qed *auto*
finally have *det1*: *?v* ∗ *det ?A* = *det ?B* **.**
from *det1 det2 B3′ B4′ v 0* **have** *?P ?B3 ?B4 ?v* (*Suc k*) **by** *auto*
thus *?thesis* **by** *blast*
qed
qed
qed
from *this*[*OF le-refl*] **obtain** *B3 B4 v* **where** *P*: *?P B3 B4 v m* **by** *blast*
let *?B* = *four-block-mat A1 A2 B3 B4*
from *P* **have** *v*: *v* ≠ *0* **and** *det*: *v* ∗ *det ?A* = *det ?B v* ∗ *det A4* = *det B4*
and *B3*: *B3* ∈ *carrier-mat ?m n* **and** *B4*: *B4* ∈ *carrier-mat ?m ?m* **and** *0*: ⋀ *i*. *i* < *m* ⟹ *B4* $$ (*i*, *m*) = *0*
by *auto*
let *?A2* = 0_m *n m*
let *?A3* = *mat m n* (λ *ij*. *B3* $$ *ij*)
let *?A4* = *mat m m* (λ *ij*. *B4* $$ *ij*)
let *?B1* = *four-block-mat A1 ?A2 ?A3 ?A4*
let *?B2* = 0_m (*n* + *m*) *1*
let *?B3* = *mat 1* (*n* + *m*) (λ (*i*,*j*). *if j* < *n then B3* $$ (*m*,*j*) *else B4* $$ (*m*,*j* − *n*))
let *?B4* = *mat 1 1* (λ -. *B4* $$ (*m*,*m*))
have *B44*: *B4* = *four-block-mat ?A4* (0_m *m 1*) (*mat 1 m* (λ (*i*,*j*). *B4* $$ (*m*,*j*))) *?B4*
proof (*rule eq-matI*, *unfold index-mat-four-block dim-col-mat dim-row-mat*, *goal-cases*)
case (*1 i j*)
hence [*simp*]: ¬ *i* < *m* ⟹ *i* = *m* ¬ *j* < *m* ⟹ *j* = *m* **by** *auto*
from *1* **show** *?case* **using** *B4 0* **by** *auto*
qed (*insert B4*, *auto*)
have *?B* = *four-block-mat ?B1 ?B2 ?B3 ?B4*

proof (*rule eq-matI*, *unfold index-mat-four-block dim-col-mat dim-row-mat*, *goal-cases*)
case (*1 i j*)
then consider (*UL*) $i < n + m$ $j < n + m$ | (*UR*) $i < n + m$ $j = n + m$
| (*LL*) $i = n + m$ $j < n + m$ | (*LR*) $i = n + m$ $j = n + m$ **using** *A1* **by** *auto linarith*
thus *?case*
proof *cases*
case *UL*
hence [*simp*]: $\neg\ i < n \Longrightarrow i - n < m$
$\neg\ j < n \Longrightarrow j - n < m$ $\neg\ j < n \Longrightarrow j - n < Suc\ m$ **by** *auto*
from *UL* **show** *?thesis* **using** *A1 A20 B3 B4* **by** *simp*
next
case *LL*
hence [*simp*]: $\neg\ j < n \Longrightarrow j - n < m$ $\neg\ j < n \Longrightarrow j - n < Suc\ m$ **by** *auto*
from *LL* **show** *?thesis* **using** *A1 A2 B3 B4* **by** *simp*
next
case *LR*
thus *?thesis* **using** *A1 A2 B3 B4* **by** *simp*
next
case *UR*
hence [*simp*]: $\neg\ i < n \Longrightarrow i - n < m$ **by** *auto*
from *UR* **show** *?thesis* **using** *A1 A20 0 B3 B4* **by** *simp*
qed
qed (*insert B4*, *auto*)
hence *det ?B* = *det* (*four-block-mat ?B1 ?B2 ?B3 ?B4*) **by** *simp*
also have ... = *det ?B1* ∗ *det ?B4*
by (*rule det-four-block-mat-upper-right-zero-col*[*of - n + m*], *insert A1 A2 B3 B4*, *auto*)
also have *det ?B1* = *det A1* ∗ *det* (*mat m m* (($$) *B4*))
by (*rule Suc*(*1*), *insert B3 B4*, *auto*)
also have ... ∗ *det ?B4* = *det A1* ∗ (*det* (*mat m m* (($$) *B4*)) ∗ *det ?B4*) **by** *simp*
also have *det* (*mat m m* (($$) *B4*)) ∗ *det ?B4* = *det B4*
unfolding *arg-cong*[*OF B44*, *of det*]
by (*subst det-four-block-mat-upper-right-zero-col*[*OF - refl*], *auto*)
finally have *id*: *det ?B* = *det A1* ∗ *det B4* **.**
from *this*[*folded det*] **have** *v* ∗ *det ?A* = *v* ∗ (*det A1* ∗ *det A4*) **by** *simp*
with *v* **show** *det ?A* = *det A1* ∗ *det A4* **by** *simp*
qed

lemma *det-four-block-mat-lower-left-zero*: **fixes** *A1* :: *'a* :: *idom mat*
assumes *A1*: *A1* ∈ *carrier-mat n n*
and *A2*: *A2* ∈ *carrier-mat n m* **and** *A30*: *A3* = 0_m *m n*
and *A4*: *A4* ∈ *carrier-mat m m*
shows *det* (*four-block-mat A1 A2 A3 A4*) = *det A1* ∗ *det A4*
proof −
have *A3*: *A3* ∈ *carrier-mat m n* **using** *A30* **by** *auto*
show *?thesis*
apply (*subst det-transpose*[*OF four-block-carrier-mat*[*OF A1 A4*], *symmetric*])

apply (*subst transpose-four-block-mat*[*OF A1 A2 A3 A4*])
apply (*subst det-four-block-mat-upper-right-zero*[*of - n - m*],
insert A1 A2 A30 A4, *auto simp*: *det-transpose*)
done
qed

context
begin
private lemma *det-four-block-mat-preliminary*: **assumes** *A*: (A :: $'a$ :: *idom mat*) $\in$ *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*: *det* $D \neq 0$
shows *det* (*four-block-mat A B C D*) = *det* ($A * D - B * C$)
proof $-$
let *?m* = $n + n$
let *?M* = *four-block-mat A B C D*
let *?N* = *four-block-mat D* (0_m *n n*) ($-C$) (1_m *n*)
have *M*: *?M* $\in$ *carrier-mat ?m ?m* **using** *A B C D* **by** *auto*
have *N*: *?N* $\in$ *carrier-mat ?m ?m* **using** *A B C D* **by** *auto*
have *det ?M* $*$ *det ?N* = *det* (*?M* $*$ *?N*) **using** *det-mult*[*OF M N*] **..**
also have *?M* $*$ *?N* = *four-block-mat* ($A * D - B * C$) *B* (0_m *n n*) *D*
by (*subst mult-four-block-mat*[*OF A B C D D*], *insert A B C D*, *auto simp*: *commute*)
also have *det* ... = *det* ($A * D - B * C$) $*$ *det D*
by (*rule det-four-block-mat-lower-left-zero*, *insert A B C D*, *auto*)
also have *det ?N* = *det D*
by (*subst det-four-block-mat-upper-right-zero*[*OF D*], *insert C*, *auto*)
finally have *det ?M* $*$ *det D* = *det* ($A * D - B * C$) $*$ *det D* **.**
with *detD* **show** *?thesis* **by** *simp*
qed

lemma *det-four-block-mat*: **assumes** *A*: (A :: $'a$ :: *idom mat*) $\in$ *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*: $C * D = D * C$
shows *det* (*four-block-mat A B C D*) = *det* ($A * D - B * C$)
proof (*cases* $n = 0$)
case *True*
hence *four-block-mat A B C D* = $A * D - B * C$ **using** *four-block-carrier-mat*[*OF A D*, *of B C*]
A B C D **by** *auto*
thus *?thesis* **by** *simp*
next
case *n*: *False*
define *l* **where** *l* = *map-mat* (λ *x* :: $'a$. [: *x* :])

have *coeff*: *coeff* [: x :] n = (*if* n = *0 then* x *else 0*) **for** x :: $'a$ **and** n
by (*simp add*: *coeff-eq-0*)
have *l-mult*: l ($A * B$) = l A $*$ l B **if** $A \in$ *carrier-mat* n n $B \in$ *carrier-mat* n n **for** A B :: $'a$ *mat*
unfolding *l-def* **using** *that*
apply (*intro eq-matI*, *auto simp*: *scalar-prod-def coeff-sum intro*!: *poly-eqI*)
subgoal for *i j n*
by (*cases n*, *auto simp*: *coeff ac-simps*)
done
let *?p0* = *map-mat* (λp. *poly p* (*0* :: $'a$))
have *poly-det*: *poly* (*det A*) *0* = *det* (*?p A*) **if** $A \in$ *carrier-mat* n n **for** A n
apply (*subst* (*1 2*) *det-def'*, *use that* **in** *force*, *rule that*)
unfolding *poly-sum poly-mult poly-prod* **using** *that*
by (*intro sum.cong*[*OF refl*] *arg-cong2*[*of - - - -* ($*$)] *prod.cong*[*OF refl*], *auto simp*: *sign-def*)
let *?A* = *l A*
let *?B* = *l B*
let *?C* = *l C*
let *?D* = *l D*
let *?Dx* = *?D* + *monom 1 1* $\cdot_m$ 1_m n
from *A B C D*
have
lA: *?A* $\in$ *carrier-mat n n* **and**
lB: *?B* $\in$ *carrier-mat n n* **and**
lC: *?C* $\in$ *carrier-mat n n* **and**
lD: *?D* $\in$ *carrier-mat n n* **and**
lDx: *?Dx* $\in$ *carrier-mat n n*
unfolding *l-def* **by** *auto*
have *?C* $*$ *?Dx* = *?C* $*$ *?D* + *?C* $*$ (*monom 1 1* $\cdot_m$ 1_m *n*)
by (*subst mult-add-distrib-mat*[*OF lC lD*], *auto*)
also have *?C* $*$ *?D* = *l* (*C* $*$ *D*) **using** *l-mult*[*OF C D*] **by** *simp*
also have ... = *l* (*D* $*$ *C*) **using** *commute* **by** *auto*
also have ... = *?D* $*$ *?C* **using** *l-mult*[*OF D C*] **by** *simp*
also have *?C* $*$ (*monom 1 1* $\cdot_m$ 1_m *n*) = (*monom 1 1* $\cdot_m$ 1_m *n*) $*$ *?C* **using** *lC* **by** *auto*
also have *?D* $*$ *?C* + ... = *?Dx* $*$ *?C*
by (*subst add-mult-distrib-mat*[*OF lD - lC*], *auto*)
finally have *comm*: *?C* $*$ *?Dx* = *?Dx* $*$ *?C* **.**
have *det* (*four-block-mat ?A ?B ?C ?Dx*) =
det (*?A* $*$ *?Dx* $-$ *?B* $*$ *?C*)
proof (*rule det-four-block-mat-preliminary*[*OF lA lB lC lDx comm*])
define *f* **where** *f* = (λ *p*. *of-int* (*sign p*) $*$ ($\prod$ *i* = *0*..<*n*. (*l D* + *monom 1 1* $\cdot_m$ 1_m *n*) $$ (*i*, *p i*)))
have *deg*: *degree* ([:*D* $$ *ij*:] + *monom 1* (*Suc 0*)) = *1* **for** *ij*
by (*subst degree-add-eq-right*, *auto simp*: *degree-monom-eq*)
have *deg-id*: *degree* (*f id*) = *n* **unfolding** *f-def*
apply (*simp*, *subst degree-prod-eq-sum-degree*, *insert D*, *auto simp*: *l-def deg*)
subgoal for *i* **using** *deg*[*of* (*i*,*i*)] **by** *auto*
done

have *degree* (*det ?Dx*) = *degree* (*sum f* {*p. p permutes* {*0..<n*}})
by (*subst det-def′*, *use lD* **in** *force*, *auto simp*: *f-def*)
also have *sum f* {*p. p permutes* {*0..<n*}} = *f id* + *sum f* ({*p. p permutes* {*0..<n*}} − {*id*})
by (*rule sum.remove*, *auto intro*: *finite-permutations*)
also have *degree* ... = *degree* (*f id*)
proof (*rule degree-add-eq-left*, *unfold deg-id*,
rule le-less-trans[*of - n − 1*], *rule degree-sum-le*,
force intro: *finite-permutations*)
fix *p*
assume *p*: *p* ∈ {*p. p permutes* {*0..<n*}} − {*id*}
then obtain *i* **where** *i*: *i* ∈ {*0 ..< n*} **and** *pi*: *p i* ≠ *i*
by (*metis Diff-iff mem-Collect-eq permutes-empty permutes-superset singletonI*)
from *p i* **have** *pin*: *p i* < *n* **by** *auto*
define *g* **where** *g* = (λ *i*. (*l D* + *monom 1 1* $\cdot_m$ 1_m *n*) $$ (*i*, *p i*))
have *degree* (*f p*) ≤ *degree* (*of-int* (*sign p*) :: ′*a poly*) + *degree* (∏ *i* = *0..<n*. *g i*)
unfolding *f-def g-def* **by** (*rule degree-mult-le*)
also have *degree* (*of-int* (*sign p*) :: ′*a poly*) = *0* **unfolding** *sign-def* **by** *auto*
also have *0* + *degree* (∏ *i* = *0..<n*. *g i*) = *degree* (∏ *i* = *0..<n*. *g i*) **by** *simp*
also have (∏ *i* = *0..<n*. *g i*) = *g i* ∗ (*prod g* ({*0..<n*} − {*i*}))
by (*subst prod.remove*[*OF - i*], *auto*)
also have *degree* ... ≤ *degree* (*g i*) + *degree* (*prod g* ({*0..<n*} − {*i*}))
by (*rule degree-mult-le*)
also have ... = *degree* (*prod g* ({*0..<n*} − {*i*})) **unfolding** *g-def l-def* **using** *i D pi pin* **by** *auto*
also have ... ≤ *sum* (*degree* ∘ *g*) ({*0..<n*} − {*i*})
by (*rule degree-prod-sum-le*, *auto*)
also have ... ≤ *sum* (λ *i*. *1*) ({*0..<n*} − {*i*})
by (*rule sum-mono*, *insert p D*, *auto simp*: *g-def o-def l-def deg*)
also have ... = *n* − *1* **using** *i* **by** *simp*
finally show *degree* (*f p*) ≤ *n* − *1* **.**
qed (*insert n*, *auto*)
also have ... = *n* **by** *fact*
finally show *det ?Dx* ≠ *0* **using** *n* **by** *auto*
qed
hence *poly* (*det* (*four-block-mat ?A ?B ?C ?Dx*)) *0* = *poly* (*det* (*?A* ∗ *?Dx* − *?B* ∗ *?C*)) *0* **by** *simp*
also have ... = *det* (*?p0* (*?A* ∗ (*?D* + *monom 1 1* $\cdot_m$ 1_m *n*) − *?B* ∗ *?C*))
by (*rule poly-det*, *use lA lB lC lD* **in** *force*)
also have *?A* ∗ (*?D* + *monom 1 1* $\cdot_m$ 1_m *n*) = *?A* ∗ *?D* + *l A* ∗ (*monom 1 1* $\cdot_m$ 1_m *n*)
by (*rule mult-add-distrib-mat*[*OF lA lD*], *auto*)
also have *?A* ∗ (*monom 1 1* $\cdot_m$ 1_m *n*) = *monom 1 1* $\cdot_m$ *?A* **using** *lA* **by** *auto*
also have *?p0* (*?A* ∗ *?D* + ... − *?B* ∗ *?C*) = *?p0* (*?A* ∗ *?D*) − *?p0* (*?B* ∗ *?C*)
by (*intro eq-matI*, *insert lA lB lC lD*, *auto simp*: *poly-monom*)
also have *?A* ∗ *?D* = *l* (*A* ∗ *D*) **using** *l-mult*[*OF A D*] **by** *auto*
also have *?p0* ... = *A* ∗ *D* **unfolding** *l-def* **by** *fastforce*

also have *?B* ∗ *?C* = *l* (*B* ∗ *C*) **using** *l-mult*[*OF B C*] **by** *auto*
also have *?p0* ... = *B* ∗ *C* **unfolding** *l-def* **by** *fastforce*
also have *poly* (*det* (*four-block-mat ?A ?B ?C ?Dx*)) *0* = *det* (*?p0* (*four-block-mat ?A ?B ?C ?Dx*))
by (*rule poly-det*[*OF four-block-carrier-mat*[*OF lA*]], *insert lD*, *auto*)
also have *?p0* (*four-block-mat ?A ?B ?C ?Dx*) = *four-block-mat* (*?p0 ?A*) (*?p0 ?B*) (*?p0 ?C*) (*?p0 ?Dx*)
by (*rule map-four-block-mat*[*OF lA lB lC*], *insert lD*, *auto*)
also have *?p0 ?A* = *A* **unfolding** *l-def* **by** *fastforce*
also have *?p0 ?B* = *B* **unfolding** *l-def* **by** *fastforce*
also have *?p0 ?C* = *C* **unfolding** *l-def* **by** *fastforce*
also have *?p0 ?Dx* = *D* **unfolding** *l-def* **using** *D* **by** (*intro eq-matI*, *auto simp*: *poly-monom*)
finally show *?thesis* **.**
qed
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 −
let *?B* = *transpose-mat A*
let *?C* = *swaprows k l ?B*
let *?D* = *transpose-mat ?C*
have *C*: *?C* ∈ *carrier-mat n n* **and** *B*: *?B* ∈ *carrier-mat n n*
unfolding *transpose-carrier-mat swaprows-carrier* **using** *A* **by** *auto*
show *?thesis*
unfolding
swapcols-is-transp-swap-rows[*OF A* ∗(*1*−*2*)]
det-transpose[*OF C*] *det-swaprows*[*OF* ∗ *B*] *det-transpose*[*OF A*] **..**
qed

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 (*induct I arbitrary*: *A*)
case (*Suc I A*)
from *Suc*(*3*) **have** *I*: *I* < *n* **by** *auto*
let *?I* = *Suc I*
let *?A* = *swaprows I ?I A*
have *AA*: *?A* ∈ *carrier-mat n n* **using** *Suc*(*2*) **by** *simp*
have *det* (*swap-row-to-front A* (*Suc I*)) = *det* (*swap-row-to-front ?A I*) **by** *simp*
also have ... = (−*1*)^*I* ∗ *det ?A* **by** (*rule Suc*(*1*)[*OF AA I*])
also have *det ?A* = −*1* ∗ *det A* **using** *det-swaprows*[*OF I Suc*(*3*) - *Suc*(*2*)] **by** *simp*
finally show *?case* **by** *simp*
qed *simp*

lemma *swap-col-to-front-det*: $A \in$ *carrier-mat n n* $\Longrightarrow I < n \Longrightarrow$ *det* (*swap-col-to-front A I*)
  $= (-1)\hat{~}I * det\ A$
proof (*induct I arbitrary*: *A*)
  case (*Suc I A*)
  from *Suc*(*3*) **have** *I*: $I < n$ **by** *auto*
  let *?I* = *Suc I*
  let *?A* = *swapcols I ?I A*
  have *AA*: *?A* $\in$ *carrier-mat n n* **using** *Suc*(*2*) **by** *simp*
  have *det* (*swap-col-to-front A* (*Suc I*)) = *det* (*swap-col-to-front ?A I*) **by** *simp*
  also have ... $= (-1)\hat{~}I * det$ *?A* **by** (*rule Suc*(*1*)[*OF AA I*])
  also have *det ?A* $= -1 * det\ A$ **using** *det-swapcols*[*OF I Suc*(*3*) - *Suc*(*2*)] **by** *simp*
  finally show *?case* **by** *simp*
qed *simp*

lemma *swap-row-to-front-four-block*: **assumes** *A1*: *A1* $\in$ *carrier-mat n m1*
  and *A2*: *A2* $\in$ *carrier-mat n m2*
  and *A3*: *A3* $\in$ *carrier-mat 1 m1*
  and *A4*: *A4* $\in$ *carrier-mat 1 m2*
  shows *swap-row-to-front* (*four-block-mat A1 A2 A3 A4*) *n* = *four-block-mat A3 A4 A1 A2*
  by (*subst swap-row-to-front-result*[*OF four-block-carrier-mat*[*OF A1 A4*]], *force*, *rule eq-matI*, *insert A1 A2 A3 A4*, *auto*)

lemma *swap-col-to-front-four-block*: **assumes** *A1*: *A1* $\in$ *carrier-mat n1 m*
  and *A2*: *A2* $\in$ *carrier-mat n1 1*
  and *A3*: *A3* $\in$ *carrier-mat n2 m*
  and *A4*: *A4* $\in$ *carrier-mat n2 1*
  shows *swap-col-to-front* (*four-block-mat A1 A2 A3 A4*) *m* = *four-block-mat A2 A1 A4 A3*
  by (*subst swap-col-to-front-result*[*OF four-block-carrier-mat*[*OF A1 A4*]], *force*, *rule eq-matI*, *insert A1 A2 A3 A4*, *auto*)

lemma *det-four-block-mat-lower-right-zero-col*: **assumes** *A1*: *A1* $\in$ *carrier-mat 1 n*
  and *A2*: *A2* $\in$ *carrier-mat 1 1*
  and *A3*: *A3* $\in$ *carrier-mat n n*
  and *A40*: *A4* = (0_m *n 1*)
  shows *det* (*four-block-mat A1 A2 A3 A4*) $= (-1)\hat{~}n * det\ A2 * det\ A3$ (**is** *det ?A* = -)
proof −
  let *?B* = *four-block-mat A3 A4 A1 A2*
  from *four-block-carrier-mat*[*OF A3 A2*]
  have *B*: *?B* $\in$ *carrier-mat* (*Suc n*) (*Suc n*) **by** *simp*
  from *A40* **have** *A4*: *A4* $\in$ *carrier-mat n 1* **by** *auto*
  from *arg-cong*[*OF swap-row-to-front-four-block*[*OF A3 A4 A1 A2*], *of det*]

swap-row-to-front-det[*OF B*, *of n*]
have *det ?A* = (−*1*)^*n* ∗ *det ?B* **by** *auto*
also have *det ?B* = *det A3* ∗ *det A2*
by (*rule det-four-block-mat-upper-right-zero-col*[*OF A3 A40 A1 A2*])
finally show *?thesis* **by** *simp*
qed

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 −
from *A30* **have** *A3*: *A3* ∈ *carrier-mat n 1* **by** *auto*
let *?B* = *four-block-mat A2 A1 A4 A3*
from *four-block-carrier-mat*[*OF A2 A3*]
have *B*: *?B* ∈ *carrier-mat* (*Suc n*) (*Suc n*) **by** *simp*
from *arg-cong*[*OF swap-col-to-front-four-block*[*OF A2 A1 A4 A3*], *of det*]
swap-col-to-front-det[*OF B*, *of n*]
have *det ?A* = (−*1*)^*n* ∗ *det ?B* **by** *auto*
also have *det ?B* = (− *1*) ^ *n* ∗ *det A1* ∗ *det A4*
by (*rule det-four-block-mat-lower-right-zero-col*[*OF A2 A1 A4 A30*])
also have (−*1*)^*n* ∗ . . . = (−*1* ∗ −*1*)^*n* ∗ *det A1* ∗ *det A4*
unfolding *power-mult-distrib* **by** (*simp add*: *ac-simps*)
finally show *?thesis* **by** *simp*
qed

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 −
have *addcol a k l A* = *A* ∗ *addrow-mat n a l k*
using *addcol-mat*[*OF A l*]**.**
also have *det* (*A* ∗ *addrow-mat n a l k*) = *det A* ∗ *det* (*addrow-mat n a l k*)
by(*rule det-mult*[*OF A*], *auto*)
also have *det* (*addrow-mat n a l k*) = *1*
using *det-addrow-mat*[*OF k*[*symmetric*]]**.**
finally show *?thesis* **using** *A* **by** *simp*
qed

definition *insert-index i* ≡ *λi′. if i′* < *i then i′ else Suc i′*

definition *delete-index i* ≡ *λi′. 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′*
unfolding *insert-index-def* **by** *auto*

lemma *delete-insert-index*[*simp*]:
  delete-index i (*insert-index i i'*) = *i'*
  unfolding *insert-index-def delete-index-def* **by** *auto*

lemma *insert-delete-index*:
  assumes *i'i*: $i' \neq i$
  shows *insert-index i* (*delete-index i i'*) = *i'*
  unfolding *insert-index-def delete-index-def* **using** *i'i* **by** *auto*

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*
  unfolding *permutation-insert-expand* **by** *auto*

lemma *permutation-insert-base*:
  assumes *p*: *p permutes* {*0*..<*n*}
  shows *permutation-insert n n p* = *p*
proof (*rule ext*)
  fix *x* **show** *permutation-insert n n p x* = *p x*
    apply (*cases rule*: *linorder-cases*[*of x n*])
    unfolding *permutation-insert-expand*
    using *permutes-others*[*OF p*] *p* **by** *auto*
qed

lemma *permutation-insert-row-step*:
  ‹*permutation-insert* (*Suc i*) *j p* ∘ *transpose i* (*Suc i*) = *permutation-insert i j p*›
(**is** ‹*?l* = *?r*›)
proof (*rule ext*)
  fix *x*
  consider ‹*x* < *i*› | ‹*x* = *i*› | ‹*x* = *Suc i*› | ‹*Suc i* < *x*›

using *Suc-lessI* **by** (*cases rule*: *linorder-cases* [*of x i*]) *blast+*
then show ‹*?l x* = *?r x*›
by *cases* (*simp-all add*: *permutation-insert-expand*)
qed

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 (*rule ext*)
fix *x* **show** *?l x* = *?r x*
proof (*cases rule*: *linorder-cases*[*of x i*])
case *less* **note** *x* = *this*
show *?thesis*
apply (*cases rule*: *linorder-cases*[*of p x j*])
unfolding *permutation-insert-expand* **using** *x* **by** *simp+*
next case *equal* **thus** *?thesis* **by** *simp*
next case *greater* **note** *x* = *this*
show *?thesis*
apply (*cases rule*: *linorder-cases*[*of p* (*x*−*1*) *j*])
unfolding *permutation-insert-expand* **using** *x* **by** *simp+*
qed
qed

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(*unfold set-eq-iff*, *intro allI iffI*)
fix *j'*
{ **assume** *L*: *j'* ∈ *?L*
then obtain *i'* **where** *i'*: *i'* ∈ {*0..<n*} **and** *dj'*: *delete-dom f i i'* = *j'* **by** *auto*
show *j'* ∈ *?R*
proof(*cases i'* < *i*)
case *True*
show *?thesis*
unfolding *image-def*
unfolding *Diff-iff*
unfolding *mem-Collect-eq singleton-iff*
proof(*intro conjI bexI*)
show *j'* ≠ *f i*
proof
assume *j'*: *j'* = *f i*
hence *f i'* = *f i*
using *dj'*[*unfolded delete-dom-def insert-index-def*] **using** *True* **by** *simp*
thus *False* **using** *iff i True* **by** *auto*
qed
show *j'* = *f i'*

```
          using dj′ True unfolding delete-dom-def insert-index-def by simp
        qed (insert i′,simp)
      next case False
        show ?thesis
          unfolding image-def
          unfolding Diff-iff
          unfolding mem-Collect-eq singleton-iff
        proof(intro conjI bexI)
          show Si′: Suc i′ ∈ ?N using i′ by auto
          show j′ ≠ f i
          proof
            assume j′: j′ = f i
            hence f (Suc i′) = f i
              using dj′[unfolded delete-dom-def insert-index-def] j′ False by simp
            thus False using iff Si′ False by auto
          qed
          show j′ = f (Suc i′)
            using dj′ False unfolding delete-dom-def insert-index-def by simp
        qed
    qed
  }
  { assume R: j′ ∈ ?R
    then obtain i′
      where i′: i′ ∈ ?N and j′fi: j′ ≠ f i and j′fi′: j′ = f i′ by auto
    hence i′i: i′ ≠ i using iff by auto
    hence n: n > 0 using i i′ by auto
    show j′ ∈ ?L
    proof (cases i′ < i)
      case True show ?thesis
        proof
          show j′ = delete-dom f i i′
            unfolding delete-dom-def insert-index-def using True j′fi′ by simp
        qed (insert True i, simp)
      next case False show ?thesis
        proof
          show i′−1 ∈ {0..<n} using i′ n by auto
          show j′ = delete-dom f i (i′−1)
            unfolding delete-dom-def insert-index-def using False j′fi′ i′i by auto
        qed
    qed
  }
qed

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(unfold set-eq-iff , intro allI iffI)
  fix j′
```

```
{ assume L: j' ∈ ?L
  then obtain i where i: i ∈ {0..<n} and ij': delete-ran f j i = j' by auto
  have f i ∈ ?N − {j} using fimg i by blast
  thus j' ∈ ?R using ij' j unfolding delete-ran-def delete-index-def by auto
}
{ assume R: j' ∈ ?R show j' ∈ ?L
  proof (cases j' < j)
    case True
      hence j' ∈ ?N − {j} using R by auto
      then obtain i where fij': f i = j' and i: i ∈ {0..<n}
        unfolding fimg[symmetric] by auto
      have delete-ran f j i = j'
         unfolding delete-ran-def delete-index-def unfolding fij' using True by
simp
      thus ?thesis using i by auto
    next case False
      hence Suc j' ∈ ?N − {j} using R by auto
      then obtain i where fij': f i = Suc j' and i: i ∈ {0..<n}
        unfolding fimg[symmetric] by auto
      have delete-ran f j i = j'
         unfolding delete-ran-def delete-index-def unfolding fij' using False by
simp
      thus ?thesis using i by auto
  qed
}
qed

lemma delete-index-inj-on:
  assumes iS: i ∉ S
  shows inj-on (delete-index i) S
proof(intro inj-onI)
  fix x y
  assume eq: delete-index i x = delete-index i y and x: x ∈ S and y: y ∈ S
  have x ≠ i y ≠ i using x y iS by auto
  thus x = y
    using eq unfolding delete-index-def
    by(cases x < i; cases y < i;simp)
qed

lemma insert-index-inj-on:
  shows inj-on (insert-index i) S
proof(intro inj-onI)
  fix x y
  assume eq: insert-index i x = insert-index i y and x: x ∈ S and y: y ∈ S
  show x = y
    using eq unfolding insert-index-def
    by(cases x < i; cases y < i;simp)
qed
```

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 (*rule eq-card-imp-inj-on*)
  have *card ?N* = *card* (*f* ‘ *?N*) **using** *card-image*[*OF inj*]**..**
  hence *card* {*0*..<*n*} = *card* (*f* ‘ *?N* − {*f i*}) **using** *i* **by** *auto*
  also have ... = *card* (*delete-dom f i* ‘ {*0*..<*n*})
    apply(*subst delete-dom-image*[*symmetric*])
    using *i inj* **unfolding** *inj-on-def* **by** *auto*
  finally show *card* (*delete-dom f i* ‘ {*0*..<*n*}) = *card* {*0*..<*n*}**..**
qed *simp*

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*}
  apply (*rule eq-card-imp-inj-on*)
  unfolding *delete-ran-image*[*OF j img*] **by** *simp+*

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−
  have *inj*: *inj-on p ?N* **using** *bij-betw-imp-inj-on*[*OF bij*]**.**
  have *ran*: *p* ‘ *?N* = *?N* **using** *bij-betw-imp-surj-on*[*OF bij*]**.**
  hence *j*: *p i* ∈ *?N* **using** *i* **by** *auto*
  have ∀ *i'*∈*?N*. *p i'* = *p i* ⟶ *i'* = *i* **using** *inj i* **unfolding** *inj-on-def* **by** *auto*
  from *delete-dom-image*[*OF i this*]
  have *delete-dom p i* ‘ {*0*..<*n*} = *?N* − {*p i*} **unfolding** *ran***.**
  from *delete-ran-inj-on*[*OF j this*] *delete-ran-image*[*OF j this*]
  show *?thesis* **unfolding** *permutation-delete-def*
    using *bij-betw-imageI* **by** *blast*
qed

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 (*rule bij-imp-permutes*, *rule permutation-delete-bij-betw*)
  have *pi*: *p i* < *Suc n* **using** *p i* **by** *auto*
  show *bij-betw p ?N ?N* **using** *permutes-imp-bij*[*OF p*]**.**
  fix *x* **assume** *x* ∉ {*0*..<*n*} **hence** *x*: *x* ≥ *n* **by** *simp*
    show *?p x* = *x*
    proof(*cases x* < *i*)
      case *True* **thus** *?thesis*
        unfolding *permutation-delete-def* **using** *x i* **by** *simp*

```
    next case False
      hence p (Suc x) = Suc x using x permutes-others[OF p] by auto
      thus ?thesis
      unfolding permutation-delete-expand using False pi x by simp
  qed
qed (insert i,auto)

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
  fix i'
  show ?l i' = p i'
  proof (cases rule: linorder-cases[of i' i])
    case less note i'i = this
      show ?thesis
      proof (cases p i = p i')
        case True
          hence i = i' using permutes-inj[OF p] injD by metis
          hence False using i'i by auto
          thus ?thesis by auto
        next case False thus ?thesis
          unfolding permutation-insert-expand permutation-delete-expand
          using i'i by auto
      qed
    next case equal thus ?thesis unfolding permutation-insert-expand by simp
    next case greater hence i'i: i' > i by auto
      hence cond: ¬ i' − 1 < i using i'i by simp
      show ?thesis
      proof (cases rule: linorder-cases[of p i' p i])
        case less
          hence pd: permutation-delete p i (i'−1) = p i'
            unfolding permutation-delete-expand
            using i'i cond by auto
          show ?thesis
            unfolding permutation-insert-expand pd
            using i'i less by simp
        next case equal
          hence i = i' using permutes-inj[OF p] injD by metis
          hence False using i'i by auto
          thus ?thesis by auto
        next case greater
          hence pd: permutation-delete p i (i'−1) = p i' − 1
            unfolding permutation-delete-expand
            using i'i cond by simp
          show ?thesis
            unfolding permutation-insert-expand pd
```

using $i'i$ *greater* **by** *auto*
qed
qed
qed

lemma *insert-index-exclude*[*simp*]:
insert-index i $i' \neq i$ **unfolding** *insert-index-def* **by** *auto*

lemma *insert-index-image*:
assumes i: $i < Suc\ n$
shows *insert-index* i ‘ $\{0..<n\} = \{0..<Suc\ n\} - \{i\}$ (**is** $?L = ?R$)
proof(*unfold set-eq-iff*, *intro allI iffI*)
let $?N = \{0..<Suc\ n\}$
fix i'
{ assume L: $i' \in ?L$
then obtain i''
where *ins*: $i' =$ *insert-index* i i'' **and** i'': $i'' \in \{0..<n\}$ **by** *auto*
show $i' \in ?N - \{i\}$
proof(*rule DiffI*)
show $i' \in ?N$ **using** *ins* **unfolding** *insert-index-def* **using** i'' **by** *auto*
show $i' \notin \{i\}$
unfolding *singleton-iff*
unfolding *ins* **unfolding** *insert-index-def* **by** *auto*
qed
}
{ assume R: $i' \in ?R$
show $i' \in ?L$
proof(*cases rule*: *linorder-cases*[*of* i' i])
case *less*
hence i': $i' \in \{0..<n\}$ **using** i **by** *auto*
hence *insert-index* i $i' = i'$ **unfolding** *insert-index-def* **using** *less* **by** *auto*
thus *?thesis* **using** i' **by** *force*
next case *equal*
hence *False* **using** R **by** *auto*
thus *?thesis* **by** *auto*
next case *greater*
hence i'': $i'-1 \in \{0..<n\}$ **using** i R **by** *auto*
hence *insert-index* i $(i'-1) = i'$
unfolding *insert-index-def* **using** *greater* **by** *auto*
thus *?thesis* **using** i'' **by** *force*
qed
}
qed

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 −

have $?L = (\lambda i.\ \textit{insert-index}\ j\ (f\ i))$ ‘ $\{0..<n\}$ **unfolding** *insert-ran-def* **..**
also have ... $= (\textit{insert-index}\ j \circ f)$ ‘ $\{0..<n\}$ **by** *auto*
also have ... $= \textit{insert-index}\ j$ ‘ f ‘ $\{0..<n\}$ **by** *auto*
also have ... $= \textit{insert-index}\ j$ ‘ $\{0..<n\}$ **unfolding** *img* **by** *auto*
finally show *?thesis* **using** *insert-index-image*[*OF j*] **by** *auto*
qed

lemma *insert-dom-image*:
assumes i: $i < \textit{Suc}\ n$ **and** j: $j < \textit{Suc}\ n$
and *img*: f ‘ $\{0..<n\} = \{0..<\textit{Suc}\ n\} - \{j\}$ (**is** - = $?N$ − -)
shows *insert-dom f i j* ‘ $?N = ?N$ (**is** $?f$ ‘ - = -)
proof(*unfold set-eq-iff*,*intro allI iffI*)
fix j'
{ assume $j' \in ?f$ ‘ $?N$
then obtain i' **where** i': $i' \in ?N$ **and** j': $j' = ?f\ i'$ **by** *auto*
show $j' \in ?N$
proof (*cases rule*:*linorder-cases*[*of* i' i])
case *less*
hence $i' \in \{0..<n\}$ **using** i **by** *auto*
hence $f\ i' < \textit{Suc}\ n$ **using** *imageI*[*of* i' $\{0..<n\}$ f] *img* **by** *auto*
thus *?thesis*
unfolding j' **unfolding** *insert-dom-def* **using** *less* **by** *auto*
next case *equal*
thus *?thesis* **unfolding** j' *insert-dom-def* **using** j **by** *auto*
next case *greater*
hence $i'{-}1 \in \{0..<n\}$ **using** i' **by** *auto*
hence $f\ (i'{-}1) < \textit{Suc}\ n$ **using** *imageI*[*of* $i'{-}1$ $\{0..<n\}$ f] *img* **by** *auto*
thus *?thesis*
unfolding j' *insert-dom-def* **using** *greater* **by** *auto*
qed
}
{ assume j': $j' \in ?N$ **show** $j' \in ?f$ ‘ $?N$
proof (*cases* $j' = j$)
case *True*
hence $?f\ i = j'$ **unfolding** *insert-dom-def* **by** *auto*
thus *?thesis* **using** i **by** *auto*
next case *False*
hence j': $j' \in ?N - \{j\}$ **using** j j' **by** *auto*
then obtain i' **where** $j'\!fi$: $j' = f\ i'$ **and** i': $i' \in \{0..<n\}$
unfolding *img*[*symmetric*] **by** *auto*
show *?thesis*
proof(*cases* $i' < i$)
case *True* **thus** *?thesis* **unfolding** $j'\!fi$ *insert-dom-def* **using** i' **by** *auto*
next case *False*
hence $?f\ (\textit{Suc}\ i') = j'$ **unfolding** $j'\!fi$ *insert-dom-def* **using** i' **by** *auto*
thus *?thesis* **using** i' **by** *auto*
qed
qed
}

qed

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 (*rule inj-onI*)
  fix $i\ i'$
  assume *i*: $i \in \{0..<n\}$ **and** *i'*: $i' \in \{0..<n\}$ **and** *eq*: $?f\ i = ?f\ i'$
  note $eq2 = eq$[*unfolded insert-ran-def insert-index-def*]
  have $f\ i = f\ i'$
  proof (*cases* $f\ i < j$)
    case *True*
      moreover have $f\ i' < j$ **apply** (*rule ccontr*) **using** *eq2 True* **by** *auto*
      ultimately show *?thesis* **using** *eq2* **by** *auto*
    next case *False*
      moreover have $\neg\ f\ i' < j$ **apply** (*rule ccontr*) **using** *eq2 False* **by** *auto*
      ultimately show *?thesis* **using** *eq2* **by** *auto*
  qed
  from *inj-onD*[*OF inj this i i'*] **show** $i = i'$**.**
qed

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*
  apply(*rule eq-card-imp-inj-on*)
  unfolding *insert-dom-image*[*OF i j img*] **by** *simp+*

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 (*rule bij-betw-imageI*)
  have *img*: q ‘ $\{0..<n\} = \{0..<n\}$ **using** *permutes-image*[*OF q*]**.**
  show $?q$ ‘ $?N = ?N$
    unfolding *permutation-insert-def*
    using *insert-dom-image*[*OF i j insert-ran-image*[*OF j permutes-image*[*OF q*]]]**.**
  have *inj*: *inj-on q* $\{0..<n\}$
    apply(*rule subset-inj-on*) **using** *permutes-inj*[*OF q*] **by** *auto*
  show *inj-on ?q ?N*
    unfolding *permutation-insert-def*
    using *insert-dom-inj-on*[*OF - i j*]
    using *insert-ran-inj-on*[*OF inj j*] *insert-ran-image*[*OF j img*] **by** *auto*
qed

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*)
using *permutation-insert-bij-betw*[*OF q i j*]
proof (*rule bij-imp-permutes*)
fix i' **assume** $i' \notin ?N$
moreover hence $q\ (i'-1) = i'-1$ **using** *permutes-others*[*OF q*] **by** *auto*
ultimately show $?p\ i' = i'$
unfolding *permutation-insert-expand* **using** *i j* **by** *auto*
qed

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** $?L = ?R$)
unfolding *set-eq-iff*
proof(*intro allI iffI*)
let $?N = \{0..<Suc\ n\}$
fix *p*
{ assume $p \in ?L$
hence *p*: *p permutes ?N* **and** *pij*: $p\ i = j$ **by** *auto*
show $p \in ?R$
unfolding *mem-Collect-eq*
using *permutation-delete-permutes*[*OF p i*]
using *permutation-insert-delete*[*OF p i,symmetric*]
unfolding *pij* **by** *auto*
}
{ assume $p \in ?R$
then obtain *q* **where** *pq*: $p =$ *permutation-insert i j q* **and** *q*: *q permutes*
$\{0..<n\}$ **by** *auto*
hence $p\ i = j$ **unfolding** *permutation-insert-expand* **by** *simp*
thus $p \in ?L$
using *pq permutation-insert-permutes*[*OF q i j*] **by** *auto*
}
qed

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** $?L = ?R$)
unfolding *set-eq-iff*
proof(*intro allI iffI*)
fix *p*
{ assume $p \in ?L$
hence *p*: *p permutes S* **by** *auto*
obtain *i* **where** *i*: $i \in S$ **and** *pij*: $p\ i = j$ **using** *j permutes-image*[*OF p*] **by**
force
thus $p \in ?R$ **using** *p* **by** *auto*
}
{ assume $p \in ?R$

then obtain *i*
  where *p*: *p permutes S* **and** *i*: $i \in S$ **and** *pij*: $p\ i = j$
  by *auto*
show $p \in ?L$
  unfolding *mem-Collect-eq* **using** *p*.
}
qed

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** $?L \cap ?R = \{\}$)
proof −
  {
    fix *p* **assume** $p \in ?L \cap ?R$
    hence *p*: *p permutes S* **and** $p\ i = j$ **and** $p\ i' = j$ **by** *auto*
    hence $p\ i = p\ i'$ **by** *auto*
    note *injD*[*OF permutes-inj*[*OF p*] *this*]
    hence *False* **using** *ii'* **by** *auto*
  }
  thus *?thesis* **by** *auto*
qed

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** $?L \cap ?R = \{\}$)
proof −
  {
    fix *p* **assume** $p \in ?L \cap ?R$
    hence *p permutes S* **and** $p\ i = j$ **and** $p\ i = j'$ **by** *auto*
    hence *False* **using** *jj'* **by** *auto*
  }
  thus *?thesis* **by** *auto*
qed

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 (*rule inj-onI*)
  fix *q q'*
  assume $q \in ?S$ $q' \in ?S$
  hence *q*: *q permutes* $\{0..<n\}$ **and** *q'*: *q' permutes* $\{0..<n\}$ **by** *auto*
  assume $?f\ q = ?f\ q'$
  hence *eq*: *permutation-insert i j q* = *permutation-insert i j q'* **by** *auto*
  note *eq* = *cong*[*OF eq refl*, *unfolded permutation-insert-expand*]
  show *qq'*: $q = q'$

proof(*rule ext*)
  fix *x*
  have *foo*: *Suc x* − *1* = *x* **by** *auto*
  show *q x* = *q′ x*
  proof(*cases x* < *i*)
    case *True* **thus** *?thesis* **apply**(*cases q x* < *j*;*cases q′ x* < *j*) **using** *eq*[*of x*] **by** *auto*
  next case *False* **thus** *?thesis*
    apply(*cases q x* < *j*;*cases q′ x* < *j*) **using** *eq*[*of Suc x*] **by** *auto*
  qed
qed
qed

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 −
  { **fix** *j* **assume** *j* ≤ *n*
  hence *signof* (*permutation-insert n* (*n*−*j*) *p*) = (−*1*::′*a*)^(*n*+(*n*−*j*)) ∗ *signof p*
  proof(*induct j*)
    case *0* **show** *?case* **using** *permutation-insert-base*[*OF p*] **by** (*simp add*: *mult-2*[*symmetric*])
  next case (*Suc j*)
    hence *Sjn*: *Suc j* ≤ *n* **and** *j*: *j* < *n* **and** *Sj*: *n* − *Suc j* < *n* **by** *auto*
    hence *n0*: *n* > *0* **by** *auto*
    have *ease*: *Suc* (*n* − *Suc j*) = *n* − *j* **using** *j* **by** *auto*
    let *?swap* = *transpose* (*n* − *Suc j*) (*n* − *j*)
    let *?prev* = *permutation-insert n* (*n* − *j*) *p*
    have *signof* (*permutation-insert n* (*n* − *Suc j*) *p*) = *signof* (*?swap* ∘ *?prev*)
      unfolding *permutation-insert-column-step* [*OF p Sj*, *unfolded ease*] **..**
    also have ... = *signof ?swap* ∗ *signof ?prev*
      proof(*rule signof-compose*)
        show *?swap permutes* {*0*..<*Suc n*} **by** (*rule permutes-swap-id*,*auto*)
      show *?prev permutes* {*0*..<*Suc n*} **by** (*rule permutation-insert-permutes*[*OF p*],*auto*)
      qed
    also have *signof ?swap* = −*1*
      proof−
        have *n* − *Suc j* < *n* − *j* **using** *Sjn* **by** *simp*
        thus *?thesis* **unfolding** *sign-swap-id* **by** *simp*
      qed
    also have *signof ?prev* = (−*1*::′*a*) ^(*n* + (*n* − *j*)) ∗ *signof p* **using** *Suc*(*1*) *j* **by** *auto*
    also have (−*1*) ∗ ... = (−*1*) ^(*1* + *n* + (*n* − *j*)) ∗ *signof p* **by** *simp*
    also have *n* − *j* = *1* + (*n* − *Suc j*) **using** *j* **by** *simp*
    also have *1* + *n* + ... = *2* + (*n* + (*n* − *Suc j*)) **by** *simp*
    also have (−*1*::′*a*) ^... = (−*1*) ^*2* ∗ (−*1*) ^(*n* + (*n* − *Suc j*)) **by** *simp*
    also have ... = (−*1*) ^(*n* + (*n* − *Suc j*)) **by** *simp*
    finally show *?case***.**

```
    qed
  }
  note col = this
  have nj: n − j ≤ n using j by auto
  have row-base: signof (permutation-insert n j p) = (−1::'a)^(n+j) ∗ signof p
    using col[OF nj] using j by simp
  { fix i assume i ≤ n
    hence signof (permutation-insert (n−i) j p) = (−1::'a)^((n−i)+j) ∗ signof p
    proof (induct i)
      case 0 show ?case using row-base by auto
      next case (Suc i)
        hence Sin: Suc i ≤ n and i: i ≤ n and Si: n − Suc i < n by auto
        have ease: Suc (n − Suc i) = n − i using Sin by auto
        let ?prev = permutation-insert (n−i) j p
        let ?swap = transpose (n − Suc i) (n−i)
        have signof (permutation-insert (n − Suc i) j p) = signof (?prev ∘ ?swap)
          using permutation-insert-row-step[of n − Suc i] unfolding ease by auto
        also have ... = signof ?prev ∗ signof ?swap
          proof(rule signof-compose)
            show ?swap permutes {0..<Suc n} by (rule permutes-swap-id,auto)
            show ?prev permutes {0..<Suc n}
              apply(rule permutation-insert-permutes[OF p]) using j by auto
          qed
        also have signof ?swap = (−1)
          proof−
            have n − Suc i < n − i using Sin by simp
            thus ?thesis unfolding sign-swap-id by simp
          qed
        also have signof ?prev = (−1::'a) ^(n − i + j) ∗ signof p
          using Suc(1)[OF i].
        also have ... ∗ (−1) = (−1) ^Suc (n − i + j) ∗ signof p
          by auto
        also have Suc (n − i + j) = Suc (Suc (n − Suc i + j))
          using Sin by auto
        also have (−1::int) ^... = (−1) ^(n − Suc i + j) by auto
        ultimately show ?case by auto
    qed
  }
  note row = this
  have ni: n − i ≤ n using i by auto
  show ?thesis using row[OF ni] using i by simp
qed

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)
  unfolding set-eq-iff
```

proof(*intro allI iffI*)
  fix *ij*
  { assume *ij* ∈ *?L*
    then obtain *i′*
      where *ij*: *ij* = (*i′*, *permutation-insert i j q i′*) **and** *i′*: *i′* < *Suc n* **and** *i′i*: *i′* ≠ *i*
      by *auto*
    show *ij* ∈ *?R* **unfolding** *mem-Collect-eq*
    proof(*intro exI conjI*)
      show *ij* = (*insert-index i* (*delete-index i i′*), *insert-index j* (*q* (*delete-index i i′*)))
        using *ij* **unfolding** *insert-delete-index*[*OF i′i*] **using** *i′i*
         unfolding *permutation-insert-expand insert-index-def delete-index-def* **by** *auto*
      show *delete-index i i′* < *n* **using** *i′ i i′i* **unfolding** *delete-index-def* **by** *auto*
    qed
  }
  { assume *ij* ∈ *?R*
    then obtain *i″*
      where *ij*: *ij* = (*insert-index i i″*, *insert-index j* (*q i″*)) **and** *i″*: *i″* < *n*
      by *auto*
    show *ij* ∈ *?L* **unfolding** *mem-Collect-eq*
    proof(*intro exI conjI*)
      show *insert-index i i″* ∈ {*0..<Suc n*} − {*i*}
        unfolding *insert-index-image*[*OF i*,*symmetric*] **using** *i″* **by** *auto*
      have *insert-index j* (*q i″*) = *permutation-insert i j q* (*insert-index i i″*)
        unfolding *permutation-insert-expand insert-index-def* **by** *auto*
      thus *ij* = (*insert-index i i″*, *permutation-insert i j q* (*insert-index i i″*))
        unfolding *ij* **by** *auto*
    qed
  }
qed

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*
  unfolding *mat-delete-def* **by** *auto*

lemma *mat-delete-carrier*:
  assumes *A*: *A* ∈ *carrier-mat m n*
  shows *mat-delete A i j* ∈ *carrier-mat* (*m*−*1*) (*n*−*1*) **unfolding** *mat-delete-def* **using** *A* **by** *auto*

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 $A\ \$\$\ (insert\text{-}index\ i\ i',\ insert\text{-}index\ j\ j') = mat\text{-}delete\ A\ i\ j\ \$\$\ (i',\ j')$
unfolding *mat-delete-def*
unfolding *permutation-insert-expand*
unfolding *insert-index-def*
using $A\ i\ j\ i'\ j'$ **by** *auto*

definition $cofactor\ A\ i\ j = (-1)\widehat{\ }(i+j) * det\ (mat\text{-}delete\ A\ i\ j)$

lemma *laplace-expansion-column*:
assumes A: $(A :: 'a :: comm\text{-}ring\text{-}1\ mat) \in carrier\text{-}mat\ n\ n$
and j: $j < n$
shows $det\ A = (\sum i<n.\ A\ \$\$\ (i,j) * cofactor\ A\ i\ j)$
proof $-$
define l **where** $l = n-1$
have A: $A \in carrier\text{-}mat\ (Suc\ l)\ (Suc\ l)$
and jl: $j < Suc\ l$ **using** $A\ j$ **unfolding** *l-def* **by** *auto*
let $?N = \{0\ ..<\ Suc\ l\}$
define f **where** $f = (\lambda p\ i.\ A\ \$\$\ (i,\ p\ i))$
define g **where** $g = (\lambda p.\ prod\ (f\ p)\ ?N)$
define h **where** $h = (\lambda p.\ signof\ p * g\ p)$
define Q **where** $Q = \{\ q\ .\ q\ permutes\ \{0..<l\}\ \}$
have jN: $j \in\ ?N$ **using** jl **by** *auto*
have $disj$: $\forall\, i \in\ ?N.\ \forall\, i' \in\ ?N.\ i \neq i' \longrightarrow$
$\{p.\ p\ permutes\ ?N \wedge p\ i = j\} \cap \{p.\ p\ permutes\ ?N \wedge p\ i' = j\} = \{\}$
using *permutation-disjoint-dom*[*OF - - jN*] **by** *auto*
have fin: $\forall\, i{\in}?N.\ finite\ \{p.\ p\ permutes\ ?N \wedge p\ i = j\}$
using *finite-permutations*[*of ?N*] **by** *auto*

have $det\ A = sum\ h\ \{\ p.\ p\ permutes\ ?N\ \}$
using *det-def'*[*OF A*] **unfolding** *h-def g-def f-def* **using** *atLeast0LessThan* **by** *auto*
also have ... $= sum\ h\ (\bigcup i{\in}?N.\ \{p.\ p\ permutes\ ?N \wedge p\ i = j\})$
unfolding *permutation-split-ran*[*OF jN*]**..**
also have ... $= (\sum i{\in}?N.\ sum\ h\ \{\ p\ |\ p.\ p\ permutes\ ?N \wedge p\ i = j\})$
using *sum.UNION-disjoint*[*OF - fin disj*] **by** *auto*
also {
fix i **assume** $i \in\ ?N$
hence i: $i < Suc\ l$ **by** *auto*
have $sum\ h\ \{\ p\ |\ p.\ p\ permutes\ ?N \wedge p\ i = j\} = sum\ h\ (permutation\text{-}insert\ i\ j\ `\ Q)$
using *permutation-fix*[*OF i jl*] **unfolding** *Q-def* **by** *auto*
also have ... $= sum\ (h \circ permutation\text{-}insert\ i\ j)\ Q$
unfolding *Q-def* **using** *sum.reindex*[*OF permutation-insert-inj-on*[*OF i jl*]]**.**
also have ... $= (\sum\ q \in Q.$
$signof\ (permutation\text{-}insert\ i\ j\ q) * prod\ (f\ (permutation\text{-}insert\ i\ j\ q))\ ?N)$
unfolding *h-def g-def Q-def* **by** *simp*

also {
fix *q* **assume** $q \in Q$
hence *q*: *q permutes* $\{0..<l\}$ **unfolding** *Q-def* **by** *auto*
let *?p* = *permutation-insert i j q*
have *fin*: *finite* (*?N* − {*i*}) **by** *auto*
have *notin*: $i \notin$ *?N* − {*i*} **by** *auto*
have *close*: *insert i* (*?N* − {*i*}) = *?N* **using** *notin i* **by** *auto*
have *prod* (*f ?p*) *?N* = *f ?p i* ∗ *prod* (*f ?p*) (*?N*−{*i*})
unfolding *prod.insert*[*OF fin notin*, *unfolded close*] **by** *auto*
also have ... = *A* $$ (*i*, *j*) ∗ *prod* (*f ?p*) (*?N*−{*i*})
unfolding *f-def Q-def* **using** *permutation-insert-inserted* **by** *simp*
also have *prod* (*f ?p*) (*?N*−{*i*}) = *prod* (λ*i'*. *A* $$ (*i'*, *permutation-insert i j q i'*)) (*?N*−{*i*})
unfolding *f-def*..
also have ... = *prod* (λ*ij*. *A* $$ *ij*) ((λ*i'*. (*i'*, *permutation-insert i j q i'*)) ' (*?N*−{*i*}))
(**is** - = *prod* - *?part*)
unfolding *prod.reindex*[*OF inj-on-convol-ident*] *o-def*..
also have *?part* = {(*i'*, *permutation-insert i j q i'*) | *i'*. *i'* ∈ *?N*−{*i*} }
unfolding *image-def* **by** *metis*
also have ... = {(*insert-index i i''*, *insert-index j* (*q i''*)) | *i''*. *i''* < *l*}
unfolding *foo*[*OF i jl q*]..
also have ... = ((λ*i''*. (*insert-index i i''*, *insert-index j* (*q i''*))) ' {*0*..<*l*})
unfolding *image-def* **by** *auto*
also have *prod* (λ*ij*. *A* $$ *ij*)... = *prod* ((λ*ij*. *A* $$ *ij*) ∘ (λ*i''*. (*insert-index i i''*, *insert-index j* (*q i''*)))) {*0*..<*l*}
proof(*subst prod.reindex*[*symmetric*])
have *1*: *inj* (λ*i''*. (*i''*, *insert-index j* (*q i''*))) **using** *inj-on-convol-ident*.
have *2*: *inj* (λ(*i''*,*j*). (*insert-index i i''*, *j*))
apply (*intro injI*) **using** *injD*[*OF insert-index-inj-on*[*of - UNIV*]] **by** *auto*
have *inj* (λ*i''*. (*insert-index i i''*, *insert-index j* (*q i''*)))
using *inj-compose*[*OF 2 1*] **unfolding** *o-def* **by** *auto*
thus *inj-on* (λ*i''*. (*insert-index i i''*, *insert-index j* (*q i''*))) {*0*..<*l*}
using *subset-inj-on* **by** *auto*
qed *auto*
also have ... = *prod* (λ*i''*. *A* $$ (*insert-index i i''*, *insert-index j* (*q i''*))) {*0*..<*l*}
by *auto*
also have ... = *prod* (λ*i''*. *mat-delete A i j* $$ (*i''*, *q i''*)) {*0*..<*l*}
proof (*rule prod.cong*[*OF refl*], *unfold atLeastLessThan-iff*, *elim conjE*)
fix *x* **assume** *x*: *x* < *l*
show *A* $$ (*insert-index i x*, *insert-index j* (*q x*)) = *mat-delete A i j* $$ (*x*, *q x*)
apply(*rule mat-delete-index*[*OF A i jl*]) **using** *q x* **by** *auto*
qed
finally have *prod* (*f ?p*) *?N* =
A $$ (*i*, *j*) ∗ (∏ *i''* = *0*..< *l*. *mat-delete A i j* $$ (*i''*, *q i''*))
by *auto*

```
      hence signof ?p * prod (f ?p) ?N  = (−1::'a)^(i+j) * signof q * ...
        unfolding signof-permutation-insert[OF q i jl] by auto
    }
    hence ... = (∑  q ∈ Q. (−1)^(i+j) * signof q *
      A $$ (i, j) * (∏ i'' = 0 ..< l. mat-delete A i j $$ (i'', q i'')))
      by(intro sum.cong[OF refl],auto)
    also have ... = ( ∑  q ∈ Q. A $$ (i, j) * (−1)^(i+j) *
       ( signof q * (∏ i'' = 0..< l. mat-delete A i j $$ (i'', q i'')) ) )
      by (intro sum.cong[OF refl],auto)
    also have ... = A $$ (i, j) * (−1)^(i+j) *
      ( ∑  q ∈ Q. signof q * (∏ i''= 0 ..< l. mat-delete A i j $$ (i'', q i'')) )
      unfolding sum-distrib-left by auto
    also have ... = (A $$ (i, j) * (−1)^(i+j) * det (mat-delete A i j))
      unfolding det-def'[OF mat-delete-carrier[OF A]]
      unfolding Q-def by auto
    finally have sum h {p | p. p permutes ?N ∧ p i = j} = A $$ (i, j) * cofactor
A i j
      unfolding cofactor-def by auto
  }
  hence ... = (∑ i∈?N. A $$ (i,j) * cofactor A i j) by auto
  finally show ?thesis unfolding atLeast0LessThan using A j unfolding l-def
by auto
qed

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 $$ (i,j) * cofactor A i j)
proof −
  have det A = det (A^T) using det-transpose[OF A] by simp
  also have . . . = (∑j<n. A^T $$ (j, i) * cofactor A^T j i)
    by (rule laplace-expansion-column[OF - i], insert A, auto)
  also have . . . = (∑j<n. A $$ (i,j) * cofactor A i j) unfolding cofactor-def
  proof (rule sum.cong[OF refl], rule arg-cong2[of - - - - λ x y. x * y], goal-cases)
    case (1 j)
    thus ?case using A i by auto
  next
    case (2 j)
    have det (mat-delete A^T j i) = det ((mat-delete A^T j i)^T)
      by (subst det-transpose, insert A, auto simp: mat-delete-def)
    also have (mat-delete A^T j i)^T = mat-delete A i j
      unfolding mat-delete-def using A by auto
    finally show ?case by (simp add: ac-simps)
  qed
  finally show ?thesis .
qed


lemma degree-det-le: assumes ⋀ i j. i < n ⟹ j < n ⟹ degree (A $$ (i,j)) ≤ k
```

```
  and A: A ∈ carrier-mat n n
shows degree (det A) ≤ k * n
proof −
  {
    fix p
    assume p: p permutes {0..<n}
    have (∑x = 0..<n. degree (A $$ (x, p x))) ≤ (∑x = 0..<n. k)
      by (rule sum-mono[OF assms(1)], insert p, auto)
    also have ... = k * n unfolding sum-constant by simp
    also note calculation
  } note * = this
  show ?thesis unfolding det-def'[OF A]
    apply (rule degree-sum-le)
     apply (simp-all add: finite-permutations)
    apply (drule *)
    apply (rule order.trans [OF degree-mult-le])
    apply simp
    apply (rule order.trans [OF degree-prod-sum-le])
     apply simp-all
    done
qed

lemma upper-triangular-imp-det-eq-0-iff:
  fixes A :: 'a :: idom mat
  assumes A ∈ carrier-mat n n and upper-triangular A
  shows det A = 0 ⟷ 0 ∈ set (diag-mat A)
  using assms by (auto simp: det-upper-triangular)

lemma det-identical-columns:
  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−
  have det A = det A^T using det-transpose[OF A] ..
  also have ... = 0
  proof (rule det-identical-rows[of - n i j])
     show row (transpose-mat A) i = row (transpose-mat A) j
       using A i j r by auto
  qed (auto simp add: assms)
  finally show ?thesis .
qed

definition adj-mat :: 'a :: comm-ring-1 mat ⇒ 'a mat where
  adj-mat A = mat (dim-row A) (dim-col A) (λ (i,j). cofactor A j i)

lemma adj-mat: assumes A: A ∈ carrier-mat n n
  shows adj-mat A ∈ carrier-mat n n
```

```
  A ∗ adj-mat A = det A ·m 1m n
  adj-mat A ∗ A = det A ·m 1m n
proof −
  from A have dims: dim-row A = n dim-col A = n by auto
  show aA: adj-mat A ∈ carrier-mat n n unfolding adj-mat-def dims by simp
  {
    fix i j
    assume ij: i < n j < n
    define B where B = mat n n (λ (i′,j′). if i′ = j then A $$ (i,j′) else A $$
(i′,j′))
    have (A ∗ adj-mat A) $$ (i,j) = (∑ k < n. A $$ (i,k) ∗ cofactor A j k)
      unfolding times-mat-def scalar-prod-def adj-mat-def using ij A by (auto
intro: sum.cong)
    also have . . . = (∑ k < n. A $$ (i,k) ∗ (−1)^(j + k) ∗ det (mat-delete A j
k))
      unfolding cofactor-def by (auto intro: sum.cong)
    also have . . . = (∑ k < n. B $$ (j,k) ∗ (−1)^(j + k) ∗ det (mat-delete B j
k))
      by (rule sum.cong[OF refl], intro arg-cong2[of - - - - λ x y. y ∗ - ∗ det x],
insert A ij,
        auto simp: B-def mat-delete-def)
    also have . . . = (∑ k < n. B $$ (j,k) ∗ cofactor B j k)
      unfolding cofactor-def by (simp add: ac-simps)
    also have . . . = det B
      by (rule laplace-expansion-row[symmetric], insert ij, auto simp: B-def)
    also have . . . = (if i = j then det A else 0)
    proof (cases i = j)
      case True
      hence B = A using A by (auto simp add: B-def)
      with True show ?thesis by simp
    next
      case False
      have det B = 0
        by (rule Determinant.det-identical-rows[OF - False ij], insert A ij, auto
simp: B-def)
      with False show ?thesis by simp
    qed
    also have . . . = (det A ·m 1m n) $$ (i,j) using ij by auto
    finally have (A ∗ adj-mat A) $$ (i, j) = (det A ·m 1m n) $$ (i, j) .
  } note main = this
  show A ∗ adj-mat A = det A ·m 1m n
    by (rule eq-matI[OF main], insert A aA, auto)

  {
    fix i j
    assume ij: i < n j < n
    define B where B = mat n n (λ (i′,j′). if j′ = i then A $$ (i′,j) else A $$
(i′,j′))
    have (adj-mat A ∗ A) $$ (i,j) = (∑ k < n. A $$ (k,j) ∗ cofactor A k i)
```

unfolding *times-mat-def scalar-prod-def adj-mat-def* **using** *ij A* **by** (*auto intro*: *sum.cong*)
also have ... = ($\sum$ *k* < *n*. *A* $$ (*k*,*j*) ∗ (−*1*)^(*k* + *i*) ∗ *det* (*mat-delete A k i*))
unfolding *cofactor-def* **by** (*auto intro*: *sum.cong*)
also have ... = ($\sum$ *k* < *n*. *B* $$ (*k*,*i*) ∗ (−*1*)^(*k* + *i*) ∗ *det* (*mat-delete B k i*))
by (*rule sum.cong*[*OF refl*], *intro arg-cong2*[*of - - - - λ x y*. *y* ∗ - ∗ *det x*], *insert A ij*,
auto simp: *B-def mat-delete-def*)
also have ... = ($\sum$ *k* < *n*. *B* $$ (*k*,*i*) ∗ *cofactor B k i*)
unfolding *cofactor-def* **by** (*simp add*: *ac-simps*)
also have ... = *det B*
by (*rule laplace-expansion-column*[*symmetric*], *insert ij*, *auto simp*: *B-def*)
also have ... = (*if i* = *j then det A else 0*)
proof (*cases i* = *j*)
case *True*
hence *B* = *A* **using** *A* **by** (*auto simp add*: *B-def*)
with *True* **show** *?thesis* **by** *simp*
next
case *False*
have *det B* = *0*
by (*rule Determinant.det-identical-columns*[*OF - False ij*], *insert A ij*, *auto simp*: *B-def*)
with *False* **show** *?thesis* **by** *simp*
qed
also have ... = (*det A* $\cdot_m$ *1*$_m$ *n*) $$ (*i*,*j*) **using** *ij* **by** *auto*
finally have (*adj-mat A* ∗ *A*) $$ (*i*, *j*) = (*det A* $\cdot_m$ *1*$_m$ *n*) $$ (*i*, *j*) **.**
} **note** *main* = *this*
show *adj-mat A* ∗ *A* = *det A* $\cdot_m$ *1*$_m$ *n*
by (*rule eq-matI*[*OF main*], *insert A aA*, *auto*)
qed

definition *replace-col A b k* = *mat* (*dim-row A*) (*dim-col A*) (λ (*i*,*j*). *if j* = *k then b* $ *i else A* $$ (*i*,*j*))

lemma *cramer-lemma-mat*:
assumes *A*: *A* ∈ *carrier-mat n n*
and *x*: *x* ∈ *carrier-vec n*
and *k*: *k* < *n*
shows *det* (*replace-col A* (*A* $*_v$ *x*) *k*) = *x* $ *k* ∗ *det A*
proof −
define *b* **where** *b* = *A* $*_v$ *x*
have *b*: *b* ∈ *carrier-vec n* **using** *A x* **unfolding** *b-def* **by** *auto*
let *?Ab* = *replace-col A b k*
have *Ab*: *?Ab* ∈ *carrier-mat n n* **using** *A* **by** (*auto simp*: *replace-col-def*)
have *x* $ *k* ∗ *det A* = (*det A* $\cdot_v$ *x*) $ *k* **using** *A k x* **by** *auto*
also have *det A* $\cdot_v$ *x* = *det A* $\cdot_v$ (*1*$_m$ *n* $*_v$ *x*) **using** *x* **by** *auto*
also have ... = (*det A* $\cdot_m$ *1*$_m$ *n*) $*_v$ *x* **using** *A x* **by** *auto*

also have ... = (*adj-mat* $A * A$) $*_v$ x **using** *adj-mat*[*OF A*] **by** *simp*
also have ... = *adj-mat* $A *_v b$ **using** *adj-mat*[*OF A*] A x **unfolding** *b-def*
by (*metis assoc-mult-mat-vec*)
also have ... \$ k = *row* (*adj-mat A*) $k \cdot b$ **using** *adj-mat*[*OF A*] b k **by** *auto*
also have ... = *det* (*replace-col A b k*) **unfolding** *scalar-prod-def* **using** b k A
by (*subst laplace-expansion-column*[*OF Ab k*], *auto intro*!: *sum.cong arg-cong*[*of* - - *det*]
arg-cong[*of* - - λ *x*. - $*$ *x*] *eq-matI*
simp: *replace-col-def adj-mat-def Matrix.row-def cofactor-def mat-delete-def ac-simps*)
finally show *?thesis* **unfolding** *b-def* **by** *simp*
qed

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*
unfolding *det-selection-fun-def* **by** *auto*

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*
  unfolding *triangular-column-def* **by** *auto*

lemma *triangular-columnI* [*intro*]:
  (⋀*i*. *j* < *i* ⟹ *i* < *dim-row A* ⟹ *A* $$ (*i*,*j*) = *0*) ⟹ *triangular-column j A*
  unfolding *triangular-column-def* **by** *auto*

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*
  unfolding *triangular-to-def triangular-column-def upper-triangular-def*
  by *auto*

lemma *triangular-toD* [*dest*]:
  triangular-to k A ⟹ *j* < *k* ⟹ *j* < *i* ⟹ *i* < *dim-row A* ⟹ *A* $$ (*i*,*j*) = *0*
  unfolding *triangular-to-def triangular-column-def* **by** *auto*

lemma *triangular-toI* [*intro*]:
  (⋀*i j*. *j* < *k* ⟹ *j* < *i* ⟹ *i* < *dim-row A* ⟹ *A* $$ (*i*,*j*) = *0*) ⟹ *triangular-to k A*
  unfolding *triangular-to-def triangular-column-def* **by** *auto*

lemma *triangle-growth*:
  assumes *tri*:*triangular-to k A*
  and *col*:*triangular-column k A*
  shows *triangular-to* (*Suc k*) *A*
  unfolding *triangular-to-def*
proof (*intro allI impI*)
  fix *i* **assume** *iSk*:*i* < *Suc k*
  show *triangular-column i A*
  proof (*cases i* = *k*)
    case *True*
      then show *?thesis* **using** *col* **by** *auto* **next**
    case *False*

then have $i < k$ **using** *iSk* **by** *auto*
thus *?thesis* **using** *tri* **unfolding** *triangular-to-def* **by** *auto*
qed
qed

lemma *triangle-trans*: $\textit{triangular-to } k\ A \Longrightarrow k > k' \Longrightarrow \textit{triangular-to } k'\ A$
by (*intro triangular-toI*, *elim triangular-toD*, *auto*)

10.2 Algorithms for Triangulization

context
fixes $mf :: 'a \Rightarrow 'a \Rightarrow 'a \times 'a \times 'a$
begin

private fun $mute :: 'a \Rightarrow nat \Rightarrow nat \Rightarrow 'a \times 'a\ mat \Rightarrow 'a \times 'a\ mat$ **where**
$\textit{mute } A\text{-}ll\ k\ l\ (r,A) = (\textit{let } p = A\ \$\$\ (k,l)\ \textit{in if } p = 0\ \textit{then } (r,A)\ \textit{else}$
$\textit{case mf } A\text{-}ll\ p\ \textit{of } (q',p',g) \Rightarrow$
$(r * q',\ \textit{addrow } (-p')\ k\ l\ (\textit{multrow } k\ q'\ A)))$

lemma *mute-preserves-dimensions*:
assumes $\textit{mute } q\ k\ l\ (r,A) = (r',A')$
shows [*simp*]: $\textit{dim-row } A' = \textit{dim-row } A$ **and** [*simp*]: $\textit{dim-col } A' = \textit{dim-col } A$
using *assms* **by** (*auto simp*: *Let-def split*: *if-splits prod.splits*)

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 $\textit{mute } (A\ \$\$\ (l,l))\ k\ l\ (r,A) = (r',A')$
$l < \textit{dim-row } A$
$l < \textit{dim-col } A$
$k < \textit{dim-row } A$
$k \neq l$
shows $A'\ \$\$\ (k,l) = 0$
proof $-$
define a **where** $a = A\ \$\$\ (l,\ l)$
define b **where** $b = A\ \$\$\ (k,\ l)$
let $?mf = mf\ (A\ \$\$\ (l,\ l))\ (A\ \$\$\ (k,\ l))$
obtain $q'\ p'\ g$ **where** *id*: $?mf = (q',p',g)$ **by** (*cases ?mf*, *auto*)
note *mf* = *mute-fun*[*unfolded mute-fun-def*, *rule-format*, *OF id*]
from *assms* **show** *?thesis*
unfolding *mat-addrow-def* **using** *mf id* **by** (*auto simp*: *ac-simps Let-def split*:
if-splits)
qed

It will not touch unexpected rows.

lemma *mute-preserves*:
$\textit{mute } q\ k\ l\ (r,A) = (r',A') \Longrightarrow$
$i < \textit{dim-row } A \Longrightarrow$
$j < \textit{dim-col } A \Longrightarrow$
$l < \textit{dim-row } A \Longrightarrow$

$k < dim\text{-}row\ A \Longrightarrow$
$i \neq k \Longrightarrow$
A' $$ $(i,j) = A$ $$ (i,j)
by (*auto simp*: *Let-def split*: *if-splits prod.splits*)

It preserves 0s in the touched row.

lemma *mute-preserves-0*:
mute q k l $(r,A) = (r',A') \Longrightarrow$
$i < dim\text{-}row\ A \Longrightarrow$
$j < dim\text{-}col\ A \Longrightarrow$
$l < dim\text{-}row\ A \Longrightarrow$
$k < dim\text{-}row\ A \Longrightarrow$
A $$ $(i,j) = 0 \Longrightarrow$
A $$ $(l,j) = 0 \Longrightarrow$
A' $$ $(i,j) = 0$
by (*auto simp*: *Let-def split*: *if-splits prod.splits*)

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*: $k < dim\text{-}row\ A$
and *lr*: $l < dim\text{-}row\ A$
and *lc*: $l < dim\text{-}col\ A$
shows *triangular-to l A'*
proof (*rule triangular-toI*)
fix *i j*
assume *jl*: $j < l$ **and** *ji*: $j < i$ **and** *ir'*: $i < dim\text{-}row\ A'$
then have *A0*: A $$ $(i,j) = 0$ **using** *triA rA'* **by** *auto*
moreover have A $$ $(l,j) = 0$ **using** *triA jl jl lr* **by** *auto*
moreover have *jc*:$j < dim\text{-}col\ A$ **using** *jl lc* **by** *auto*
moreover have *ir*: $i < dim\text{-}row\ A$ **using** *ir' rA'* **by** *auto*
ultimately show A' $$ $(i,j) = 0$
using *mute-preserves-0*[*OF rA'*] *lr kr* **by** *auto*
qed

Recursive application of *mute*

private fun *sub1* :: $'a \Rightarrow nat \Rightarrow nat \Rightarrow 'a \times 'a\ mat \Rightarrow 'a \times 'a\ mat$
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 $(r,A) = (r',A') \Longrightarrow dim\text{-}row\ A' = dim\text{-}row\ A$
sub1 q k l $(r,A) = (r',A') \Longrightarrow dim\text{-}col\ A' = dim\text{-}col\ A$
proof (*induction k arbitrary*: *r' A'*)
case (*Suc k*)
moreover obtain *r' A'* **where** *rA'*: *sub1 q k l* $(r, A) = (r', A')$ **by** *force*
moreover fix *r'' A''* **assume** *sub1 q* (*Suc k*) *l* $(r, A) = (r'', A'')$

ultimately show *dim-row* A'' = *dim-row* A *dim-col* A'' = *dim-col* A **by** *auto*
qed *auto*

lemma *sub1-closed* [*simp*]:
sub1 q k l $(r,A) = (r',A') \Longrightarrow A \in$ *carrier-mat* m $n \Longrightarrow A' \in$ *carrier-mat* m n
unfolding *carrier-mat-def* **by** *auto*

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)
using *assms*
proof −
show $k + l <$ *dim-row* $A \Longrightarrow$ *sub1* q k l $(r,A) = (r',A') \Longrightarrow$
A' \$\$ $(l,l) = A$ \$\$ (l,l)
proof (*induction* k *arbitrary*: r' A')
case (*Suc* k)
obtain r'' A'' **where** rA''[*simp*]: *sub1* q k l $(r,A) = (r'',A'')$ **by** *force*
have [*simp*]:*dim-row* A'' = *dim-row* A **and** [*simp*]:*dim-col* A'' = *dim-col* A
using *snd-conv sub1-preserves-dimensions*[*OF* rA''] **by** *auto*
have A'' \$\$ $(l,l) = A$ \$\$ (l,l) **using** *assms Suc* **by** *auto*
have rA': *mute* q $(l +$ *Suc* $k)$ l $(r'', A'') = (r',A')$
using *Suc* **by** *auto*
show *?case* **using** *subst mute-preserves*[*OF* rA'] *Suc assms* **by** *auto*
qed *auto*
qed

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'
using *assms*
proof −
show *sub1* q k l $(r,A) = (r',A') \Longrightarrow l + k <$ *dim-row* $A \Longrightarrow$
triangular-to l A'
proof (*induction* k *arbitrary*: r' A')
case (*Suc* k)
then have *sub1* q (*Suc* k) l $(r,A) = (r',A')$ **by** *auto*
moreover obtain r'' A''
where rA'': *sub1* q k l $(r, A) = (r'',A'')$ **by** *force*
ultimately
have rA': *mute* q (*Suc* $(l + k)$) l $(r'',A'') = (r',A')$ **by** *auto*
have *triangular-to* l A'' **using** rA'' *Suc* **by** *auto*
thus *?case*

```
    using Suc assms mute-preserves-triangle[OF rA'] rA'' by auto
  qed (insert assms,auto)
qed

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
using assms
proof −
  show sub1 (A $$ (l,l)) k l (r,A) = (r',A') ⟹ i ≤ k + l ⟹ k + l < dim-row
A ⟹
    A' $$ (i,l) = 0
  using lr lc li
  proof (induction k arbitrary: r' A')
  case (Suc k)
    obtain r' A' where rA': sub1 (A $$ (l,l)) k l (r, A) = (r',A') by force
    fix r'' A''
   from sub1-preserves-diagnal[OF rA'] have AA': A $$ (l, l) = A' $$ (l, l) using
Suc(2−) by auto
    assume sub1 (A $$ (l,l)) (Suc k) l (r, A) = (r'',A'')
    then have rA'': mute (A $$ (l,l)) (Suc (l + k)) l (r', A') = (r'', A'')
      using rA' by simp
    have ir: i < dim-row A using Suc by auto
    have il: i ≠ l using li by auto
    have lr': l < dim-row A' using lr rA' by auto
    have lc': l < dim-col A' using lc rA' by auto
    have Slkr': Suc (l+k) < dim-row A' using Suc rA' by auto
    show A'' $$ (i,l) = 0
    proof (cases Suc(l + k) = i)
      case True {
        have l: Suc (l + k) ≠ l by auto
        show ?thesis
          using mute-makes-0[OF mf rA''[unfolded AA'] lr' lc' Slkr' l] ir il rA'
          by (simp add:True)
      } next
      case False {
        then have ikl: i ≤ k+l using Suc by auto
        have ir': i < dim-row A' using ir rA' by auto
        have lc': l < dim-col A' using lc rA' by auto
        have IH: A' $$ (i,l) = 0 using rA' Suc False by auto
        thus ?thesis using mute-preserves[OF rA'' ir' lc'] rA' False Suc
```

```
      by simp
    }
  qed
 qed auto
qed

lemma sub1-triangulizes-column:
 assumes rA': sub1 (A $$ (l,l)) (dim-row A − Suc l) l (r,A) = (r',A')
 and tri:triangular-to l A
 and r: dim-row A > 0
 and lr: l < dim-row A
 and lc: l < dim-col A
 shows triangular-column l A'
proof (intro triangular-columnI)
 fix i
 assume li: l < i
 assume ir: i < dim-row A'
  also have ... = dim-row A using sub1-preserves-dimensions[OF rA'] by auto
  also have ... = dim-row A − l + l using lr li by auto
  finally have ir2: i ≤ dim-row A − l + l by auto
 show A' $$ (i,l) = 0
  apply (subst sub1-makes-0s[OF rA' lr lc])
  using li ir assms
  by auto
qed
```

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 −
 have triangular-to l A'
  using sub1-preserves-triangle[OF rA'] assms by auto
 moreover have triangular-column l A'
  using sub1-triangulizes-column[OF rA'] assms by auto
 ultimately show ?thesis by (rule triangle-growth)
qed
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* $$ (*m*,*l*) ≠ *0* *l* < *m* *m* < *dim-row A*
proof −
let *?xs* = *filter* (λ (*i*,*Ail*). *Ail* ≠ *0*) (*map* (λ *i*. (*i*, *A* $$ (*i*,*l*))) [*Suc l*..<*dim-row A*])
from *res*[*unfolded find-non0-def Let-def*]
have *xs*: *?xs* ≠ [] **and** *m*: *m* = *sel-fun ?xs*
by (*cases ?xs*, *auto*)+
from *det-selection-funD*[*OF sel-fun xs*, *folded m*] **show** *A* $$ (*m*, *l*) ≠ *0* *l* < *m* *m* < *dim-row A* **by** *auto*
qed

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 (*intro triangular-columnI*)
fix *i*
let *?xs* = *filter* (λ (*i*,*Ail*). *Ail* ≠ *0*) (*map* (λ *i*. (*i*, *A* $$ (*i*,*l*))) [*Suc l*..<*dim-row A*])
assume *none*: *find-non0 l A* = *None* **and** *li*: *l* < *i* *i* < *dim-row A*
from *none* **have** *xs*: *?xs* = []
unfolding *find-non0-def Let-def* **by** (*cases ?xs*, *auto*)
from *li* **have** (*i*, *A* $$ (*i*,*l*)) ∈ *set* (*map* (λ *i*. (*i*, *A* $$ (*i*,*l*))) [*Suc l*..<*dim-row A*]) **by** *auto*
with *xs* **show** *A* $$ (*i*,*l*) = *0*
by (*metis* (*mono-tags*) *xs case-prodI filter-empty-conv*)
qed

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 (*cases find-non0 l A*)
case *None* **then show** *?thesis* **using** *rA'* **by** *auto* **next**
case (*Some m*) **then show** *?thesis* **using** *rA'* **by** (*cases m* = *l*, *auto simp*: *Let-def*)
qed

lemma *sub2-closed* [*simp*]:

```
  sub2 d l (r,A) = (r',A') ⟹ A ∈ carrier-mat m n ⟹ A' ∈ carrier-mat m n
  unfolding carrier-mat-def by auto

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 -
  have lr: l < dim-row A using ld dr by auto
  show ?thesis
  proof (cases find-non0 l A)
    case None then show ?thesis using rA' tri by simp next
    case (Some m) {
      have lm : l < m and mr : m < dim-row A
        using find-non0[OF sel-fun Some] by auto
      let ?A1 = swaprows m l A

      have tri'': triangular-to l ?A1
      proof (intro triangular-toI)
        fix i j
        assume jl:j < l and ji:j < i and ir1: i < dim-row ?A1
        have jm: j < m using jl lm by auto
        have ir: i < dim-row A using ir1 by auto
        have jc: j < dim-col A using jl lc by auto
        show ?A1 $$ (i, j) = 0
        proof (cases m=i)
          case True {
            then have li: l ≠ i using lm by auto
            hence ?A1 $$ (i,j) = A $$ (l,j) using ir jc ‹m=i› by auto
            also have ... = 0 using tri jl lr by auto
            finally show ?thesis.
          } next
          case False show ?thesis
            proof (cases l=i)
              case True {
                then have ?A1 $$ (i,j) = A $$ (m,j)
                  using ir jc ‹m≠i› by auto
                thus ?A1 $$ (i,j) = 0 using tri jl jm mr by auto
              } next
              case False {
                then have ?A1 $$ (i,j) = A $$ (i,j)
                  using ir jc ‹m≠i› by auto
```

```
            thus ?A1 $$ (i,j) = 0 using tri jl ji ir by auto
          }
        qed
      qed
    qed

    let ?rA3 = sub1 (?A1 $$ (l,l)) (d − Suc l) l (−r,?A1)
    have [simp]: dim-row ?A1 = dim-row A ∧ dim-col ?A1 = dim-col A by auto
    have rA'2: ?rA3 = (r',A') using rA' Some by (simp add: Let-def)
    have l + (d − Suc l) < dim-row A using ld dr by auto
    thus ?thesis
      using sub1-preserves-triangle[OF rA'2 tri''] lr lc rA' by auto
  }
  qed
qed

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 (rule triangle-growth)
  show triangular-to l A'
    using sub2-preserves-triangle[OF rA' tri lc lr] by auto
    next
  have r0: 0 < dim-row A using lr by auto
  show triangular-column l A'
    proof (cases find-non0 l A)
      case None {
        then have A' = A using rA' by simp
        moreover have triangular-column l A  using find-non0-all0[OF None].
        ultimately show ?thesis by auto
      } next
      case (Some m) {
        have lm: l < m and mr: m < dim-row A
          using find-non0[OF sel-fun Some] by auto
        let ?A = swaprows m l A
        have tri2: triangular-to l ?A
          proof
            fix i j assume jl: j < l and ji:j < i and ir: i < dim-row ?A
            show ?A $$ (i,j) = 0
            proof (cases i = m)
              case True {
                then have ?A $$ (i,j) = A $$ (l,j)
                  using jl lc ir by simp
                also have ... = 0
                  using triangular-toD[OF tri jl jl] lr by auto
```

```
          finally show ?thesis by auto
        } next
        case False show ?thesis
          proof (cases i = l)
            case True {
              then have ?A $$ (i,j) = A $$ (m,j)
                using jl lc ir by auto
              also have ... = 0
                using triangular-toD[OF tri jl] jl lm mr by auto
              finally show ?thesis by auto
            } next
            case False {
              then have ?A $$ (i,j) = A $$ (i,j)
                using ‹i ≠ m› jl lc ir by auto
              thus ?thesis using tri jl ji ir by auto
            }
          qed
        qed
      qed
    have rA'2: sub1 (?A $$ (l,l)) (dim-row ?A − Suc l) l (−r, ?A) = (r',A')
      using lm Some rA' by (simp add: Let-def)
    show ?thesis
      using sub1-triangulizes-column[OF mf rA'2 tri2] r0 lr lc by auto
  }
  qed
qed
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 (induction l arbitrary: r' A')
  case (Suc l)
    obtain r' A' where rA': sub3 d l (r, A) = (r', A') by force
    fix r'' A'' assume rA'': sub3 d (Suc l) (r, A) = (r'', A'')
    then show dim-row A'' = dim-row A dim-col A'' = dim-col A
    using Suc rA' by auto
qed auto

lemma sub3-closed[simp]:
  sub3 k l (r,A) = (r',A') ⟹ A ∈ carrier-mat m n ⟹ A' ∈ carrier-mat m n
  unfolding carrier-mat-def by auto
```

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'*
  using *assms*
proof −
  show *sub3* (*dim-row A*) *l* (*r*,*A*) = (*r'*,*A'*) ⟹ *l* ≤ *dim-row A* ⟹ *l* ≤ *dim-col A*
⟹
    triangular-to l A'
  proof (*induction l arbitrary*:*r' A'*)
    case (*Suc l*)
      then have *Slr*: *Suc l* ≤ *dim-row A* **and** *Slc*: *Suc l* ≤ *dim-col A* **by** *auto*
      hence *lr*: *l* < *dim-row A* **and** *lc*: *l* < *dim-col A* **by** *auto*
      moreover obtain *r'' A''*
        where *rA''*: *sub3* (*dim-row A*) *l* (*r*,*A*) = (*r''*,*A''*) **by** *force*
      ultimately have *IH*: *triangular-to l A''* **using** *Suc* **by** *auto*
      have [*simp*]:*dim-row A''* = *dim-row A* **and** [*simp*]:*dim-col A''* = *dim-col A*
        using *Slr Slc rA''* **by** *auto*
      fix *r' A'*
      assume *sub3* (*dim-row A*) (*Suc l*) (*r*, *A*) = (*r'*, *A'*)
      then have *rA'*: *sub2* (*dim-row A''*) *l* (*r''*,*A''*) = (*r'*,*A'*)
        using *rA''* **by** *auto*
      show *triangular-to* (*Suc l*) *A'*
        using *sub2-grows-triangle*[*OF sel-fun mf rA'*] *lr lc rA'' IH* **by** *auto*
  qed *auto*
qed

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*
  unfolding *triangulize-def* **by** *auto*

lemma *triangulize-closed*[*simp*]:
  triangulize A = (*r'*,*A'*) ⟹ *A* ∈ *carrier-mat m n* ⟹ *A'* ∈ *carrier-mat m n*
  unfolding *carrier-mat-def* **by** *auto*

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 (*cases* $0<n$)
  case *True*
    have rA': *sub3* (*dim-row A*) (*dim-row A*) $(1,A) = (r',A')$
      using *assms* **unfolding** *triangulize-def* **by** *auto*
    have *nr*:$n =$ *dim-row A* **and** *nc*:$n =$ *dim-col A* **and** nr':$n =$ *dim-row* A'
    using *assms* **by** *auto*
    thus *?thesis*
      unfolding *triangular-to-triangular*
      using *sub3-makes-triangle*[*OF mf sel-fun* rA'] *True* **by** *auto*
    next
  case *False*
    then have nr':*dim-row* $A' = 0$ **using** *assms* **by** *auto*
    thus *?thesis* **unfolding** *upper-triangular-def* **by** *auto*
qed

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 $r' \neq 0$
using *assms*
proof −
  show *sub1 q k l* $(r,A) = (r',A') \Longrightarrow k + l <$ *dim-row A* $\Longrightarrow r' \neq 0$
  proof (*induction k arbitrary*: r' A')
    case (*Suc k*)
      obtain r'' A'' **where** rA'': *sub1 q k l* $(r,\ A) = (r'',\ A'')$ **by** *force*
      then have *IH*: $r'' \neq 0$ **using** *Suc* **by** *auto*
     obtain q' l' g **where** *mf-id*: *mf q* (A'' $$ (*Suc* $(l + k)$, l)) $= (q',l',g)$ **by** (*rule prod-cases3*)
      define *fact* **where** *fact* = (*if* A'' $$ (*Suc* $(l{+}k)$,l) $= 0$ *then 1 else* q')
      note *mf* = *mf*[*unfolded mute-fun-def*, *rule-format*, *OF mf-id*]
      have *All*: $q \neq 0$
        using *sub1-preserves-diagnal*[*OF* rA'' *lc*] *All Suc* **by** *auto*
      moreover have *fact* $\neq 0$ **unfolding** *fact-def* **using** *All mf* **by** *auto*
      moreover assume *sub1 q* (*Suc k*) l $(r,A) = (r',A')$
        then have *mute q* (*Suc* $(l + k)$) l $(r'',A'') = (r',A')$
          using rA'' **by** *auto*
        hence $r'' *$ *fact* $= r'$

unfolding *mute.simps fact-def Let-def mf-id* **using** *IH* **by** (*auto split*: *if-splits*)
ultimately show $r' \neq 0$ **using** *IH* **by** *auto*
qed (*insert r0*, *simp*)
qed

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* ≤ *dim-row A*
and *lc*: *l* < *dim-col A*
and *r0*: *r* ≠ *0*
shows *r′* ≠ *0*
using *assms*
proof (*cases find-non0 l A*) **{**
case (*Some m*)
then have *Aml0*: *A* $$ (*m*,*l*) ≠ *0* **using** *find-non0*[*OF sel-fun*] **by** *auto*
have *md*: *m* < *dim-row A* **using** *find-non0*[*OF sel-fun Some*] *lk kr* **by** *auto*
let *?A′′* = *swaprows m l A*
have *rA′2*: *sub1* (*?A′′* $$ (*l*,*l*)) (*k* − *Suc l*) *l* (−*r*, *?A′′*) = (*r′*,*A′*)
using *rA′ Some* **by** (*simp add*: *Let-def*)
have *All0*: *?A′′* $$ (*l*,*l*) ≠ *0* **using** *Aml0 md lk kr lc* **by** *auto*
show *?thesis* **using** *sub1-divisor*[*OF rA′2 - All0*] *r0 lk kr lc* **by** *simp*
} **qed** *auto*

lemma *sub3-divisor* [*simp*]:
assumes *sub3 d l* (*r*,*A*) = (*r′′*,*A′′*)
and *l* ≤ *d*
and *d* ≤ *dim-row A*
and *l* ≤ *dim-col A*
and *r0*: *r* ≠ *0*
shows *r′′* ≠ *0*
using *assms*
proof −
show
sub3 d l (*r*,*A*) = (*r′′*,*A′′*) ⟹
l ≤ *d* ⟹ *d* ≤ *dim-row A* ⟹ *l* ≤ *dim-col A* ⟹ *?thesis*
proof (*induction l arbitrary*: *r′′ A′′*)
case *0*
then show *?case* **using** *r0* **by** *simp*
next
case (*Suc l*)
obtain *r′ A′* **where** *rA′*: *sub3 d l* (*r*,*A*) = (*r′*,*A′*) **by** *force*
then have [*simp*]:*dim-row A′* = *dim-row A* **and** [*simp*]:*dim-col A′* = *dim-col A*
by *auto*
from *rA′* **have** *r′* ≠ *0* **using** *Suc r0* **by** *auto*
moreover have *sub2 d l* (*r′*,*A′*) = (*r′′*,*A′′*)

using rA' *Suc* **by** *simp*
ultimately show *?case* **using** *sub2-divisor* **using** *Suc* **by** *simp*
qed
qed

theorem *triangulize-divisor*:
assumes *A*: $A \in$ *carrier-mat d d*
shows *triangulize* $A = (r',A') \Longrightarrow r' \neq 0$
unfolding *triangulize-def*
proof $-$
assume rA': *sub3* (*dim-row A*) (*dim-row A*) $(1, A) = (r', A')$
show *?thesis* **using** *sub3-divisor*[*OF* rA'] *A* **by** *auto*
qed

10.8 Determinant Preservation Results

For each sub-algorithm f, we show $f(r, A) = (r', A')$ implies $r * det\ A' = r' * det\ A$.

lemma *mute-det*:
assumes $A \in$ *carrier-mat n n*
and rA': *mute q k l* $(r,A) = (r',A')$
and $k < n$
and $l < n$
and $k \neq l$
shows $r * det\ A' = r' * det\ A$
proof (*cases A* $$ $(k,l) = 0$)
case *True*
thus *?thesis* **using** *assms* **by** *auto*
next
case *False*
obtain p' q' *g* **where** *mf-id*: *mf q* (*A* $$ $(k,l)) = (q',p',g)$ **by** (*rule prod-cases3*)
let *?All* $= q'$
let *?Akl* $= -\ p'$
let *?B* = *multrow k ?All A*
let *?C* = *addrow ?Akl k l ?B*
have $r * det\ A' = r * det$ *?C* **using** *assms* **by** (*simp add*: *Let-def mf-id False*)
also have *det ?C* = *det ?B* **using** *assms* **by** (*auto simp*: *det-addrow*)
also have ... = *?All* $*$ *det A* **using** *assms det-multrow* **by** *auto*
also have $r *$ ... = ($r *$ *?All*) $*$ *det A* **by** *simp*
also have *r*: $r *$ *?All* $= r'$ **using** *assms* **by** (*simp add*: *Let-def mf-id False*)
finally show *?thesis*.
qed

lemma *sub1-det*:
assumes *A*: $A \in$ *carrier-mat n n*
and *sub1*: *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 ∗ det A′′ = r′′ ∗ det A
  using sub1 l
proof (induction k arbitrary: A′′ r′′)
  case 0
  then show ?case by auto
next
  case (Suc k)
  let ?rA′ = sub1 q k l (r,A)
  obtain r′ A′ where rA′:?rA′ = (r′,A′) by force
  have A′:A′ ∈ carrier-mat n n using sub1-closed[OF rA′] A by auto
  have IH: r ∗ det A′ = r′ ∗ det A using Suc assms rA′ by auto
  assume sub1 q (Suc k) l (r,A) = (r′′,A′′)
  then have rA′′:mute q (Suc (l+k)) l (r′,A′) = (r′′,A′′) using rA′ by auto
  hence lem: r′ ∗ det A′′ = r′′ ∗ det A′
    using assms Suc A′ mute-det[OF A′ rA′′] by auto
  hence r ∗ r′ ∗ det A′′ = r ∗ r′′ ∗ det A′ by auto
  also from IH have ... = r′′ ∗ r′ ∗ det A by auto
  finally have ∗: r ∗ r′ ∗ det A′′ = r′′ ∗ r′ ∗ det A .
  then have r ∗ r′ ∗ det A′′ div r′ = r′′ ∗ r′ ∗ det A div r′ by presburger
  moreover have r′ ≠ 0
    using r0 sub1-divisor[OF rA′] All0 Suc A by auto
  ultimately show ?case using ∗ by auto
qed

lemma sub2-det:
  assumes A: A ∈ carrier-mat d d
  and rA′: sub2 d l (r,A) = (r′,A′)
  and r0: r ≠ 0
  and ld: l < d
  shows r ∗ det A′ = r′ ∗ det A
proof (cases find-non0 l A)
  case None then show ?thesis using assms by auto next
  case (Some m) {
    then have lm: l < m and md: m < d
      using A find-non0[OF sel-fun Some] ld by auto
    hence m ≠ l by auto
    let ?A′′ = swaprows m l A
    have rA′2: sub1 (?A′′ $$ (l,l)) (d − Suc l) l (−r, ?A′′) = (r′,A′)
      using rA′ Some by (simp add: Let-def)
    have A′′: ?A′′ ∈ carrier-mat d d using A by auto
    hence A′′ll0: ?A′′ $$ (l,l) ≠ 0
      using find-non0[OF sel-fun Some] ld by auto
    hence −r ∗ det A′ = r′ ∗ det ?A′′
      using sub1-det[OF A′′ rA′2] ld A r0 by auto
    also have r ∗ ... = −r ∗ r′ ∗ det A
      using det-swaprows[OF md ld ‹m≠l› A] by auto
    finally have r ∗ −r ∗ det A′ = −r ∗ r′ ∗ det A by auto
    thus ?thesis using r0 by auto
  }
```

```
qed

lemma sub3-det:
  assumes A:A ∈ carrier-mat d d
  and sub3 d l (r,A) = (r'',A'')
  and r0: r ≠ 0
  and l ≤ d
  shows r ∗ det A'' = r'' ∗ det A
  using assms
proof −
  have d: d = dim-row A using A by auto
  show sub3 d l (r,A) = (r'',A'') ⟹ l ≤ d ⟹ r ∗ det A'' = r'' ∗ det A
  proof (induction l arbitrary: r'' A'')
    case (Suc l)
      let ?rA' = sub3 d l (r,A)
      obtain r' A' where rA':?rA' = (r',A') by force
      then have rA'': sub2 d l (r',A') = (r'',A'')
        using Suc by auto
      have A': A' ∈ carrier-mat d d using A rA' rA'' by auto
      have r'0: r' ≠ 0 using r0 sub3-divisor[OF rA'] A Suc by auto
      have r' ∗ det A'' = r'' ∗ det A'
        using Suc r'0 A by(subst sub2-det[OF A' rA''],auto)
      also have r ∗ ... = r'' ∗ (r ∗ det A') by auto
      also have r ∗ det A' = r' ∗ det A using Suc rA' by auto
      also have r'' ∗ ... div r' = r'' ∗ r' ∗ det A div r' by (simp add: ac-simps)
      finally show r ∗ det A'' = r'' ∗ det A using r'0
        by (metis ‹r ∗ det A' = r' ∗ det A› ‹r' ∗ det A'' = r'' ∗ det A'›
          mult.left-commute mult-cancel-left)
  qed simp
qed

theorem triangulize-det:
  assumes A: A ∈ carrier-mat d d
  and rA': triangulize A = (r',A')
  shows det A ∗ r' = det A'
proof −
  have rA'2: sub3 d d (1,A) = (r',A')
    using A rA' unfolding triangulize-def by auto
  show ?thesis
  proof (cases d = 0)
    case True
      then have A': A' ∈ carrier-mat 0 0 using A rA'2 by auto
      have rA'3: (r',A') = (1,A) using True rA'2 by simp
      thus ?thesis by auto
      next
    case False
      then show ?thesis using sub3-det[OF A rA'2] assms by auto
  qed
qed
```

end

10.9 Determinant Computation

definition *det-code* :: *'a mat* ⇒ *'a* **where**
det-code A = (*if dim-row A* = *dim-col A 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*
using *det-code-def*[*simp*]
proof (*cases dim-row A* = *dim-col A*)
case *True*
then have *A*: *A* ∈ *carrier-mat* (*dim-row A*) (*dim-row A*) **unfolding** *carrier-mat-def* **by** *auto*
obtain *r' A'* **where** *rA'*: *triangulize A* = (*r'*,*A'*) **by** *force*
from *triangulize-divisor*[*OF mf sel-fun A*] *rA'* **have** *r'0*: *r'* ≠ *0* **by** *auto*
from *triangulize-det*[*OF mf sel-fun A rA'*] **have** *det'*: *det A* ∗ *r'* = *det A'* **by** *auto*
from *triangulized*[*OF mf sel-fun A*, *unfolded rA'*] **have** *tri'*: *upper-triangular A'* **by** *simp*
have *A'*: *A'* ∈ *carrier-mat* (*dim-row A'*) (*dim-row A'*)
using *triangulize-closed*[*OF rA' A*] **by** *auto*
from *tri'* **have** *tr*: *triangular-to* (*dim-row A'*) *A'* **by** *auto*
have *det-code A* = *prod-list* (*diag-mat A'*) *div r'* **using** *rA' True* **by** *simp*
also have *prod-list* (*diag-mat A'*) = *det A'*
unfolding *det-upper-triangular*[*OF tri' A'*] **..**
also have ... = *det A* ∗ *r'* **by** (*simp add*: *det'*)
also have ... *div r'* = *det A* **using** *r'0* **by** *auto*
finally show *?thesis* **.**
qed (*simp add*: *det-def*)

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*
unfolding *mute-fun-def trivial-mute-fun-def* **by** *auto*

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*
unfolding *det-selection-fun-def fst-sel-fun-def* **by** *auto*

context
fixes *measure* :: $'a \Rightarrow nat$
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*
unfolding *det-selection-fun-def*
proof (*intro allI impI*)
fix *xs* :: (*nat* × *'a*)*list*
assume *xs* ≠ []
then obtain *i p ys* **where** *xs*: *xs* = ((*i*,*p*) # *ys*) **by** (*cases xs*, *auto*)
then obtain *m* **where** *id*: *select-min xs* = *select-min-main m i ys* **unfolding** *select-min-def* **by** *auto*
have *i* ∈ *fst* ‘ *set xs set ys* ⊆ *set xs* **unfolding** *xs* **by** *auto*
thus *select-min xs* ∈ *fst* ‘ *set xs* **unfolding** *id*
proof (*induct ys arbitrary*: *m i*)
case (*Cons jp ys m i*)
obtain *j p* **where** *jp*: *jp* = (*j*,*p*) **by** *force*
obtain *k n* **where** *res*: *select-min-main m i* (*jp* # *ys*) = *select-min-main n k ys*
and *k*: *k* ∈ *fst* ‘ *set xs*
using *Cons*(*2*−) **unfolding** *jp* **by** (*cases measure p* < *m*; *force simp*: *Let-def*)
from *Cons*(*1*)[*OF k*, *of n*] *Cons*(*3*)
show *?case* **unfolding** *res* **by** *auto*
qed *simp*
qed
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*
by *simp*

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*
unfolding *mute-fun-def field-mute-fun-def* **by** *auto*

definition *det-field* :: $'a$ :: *field mat* $\Rightarrow$ $'a$ **where**
det-field A = *det-code fst-sel-fun field-mute-fun A*

lemma *det-field*[*simp*]: *det-field* = *det*
by (*intro ext*, *auto simp*: *det-field-def*)

definition *gcd-mute-fun* :: $'a$ :: *ring-gcd* $\Rightarrow$ $'a \Rightarrow$ $'a \times$ $'a \times$ $'a$ **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*
unfolding *mute-fun-def gcd-mute-fun-def* **by** (*auto simp*: *Let-def div-mult-swap mult.commute*)

definition *det-int* :: *int mat* $\Rightarrow$ *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*
by (*intro ext*, *auto simp*: *det-int-def*)

definition *det-field-poly* :: $'a$ :: {*field*,*field-gcd*} *poly mat* $\Rightarrow$ $'a$ *poly* **where**
det-field-poly A = *det-code* (*select-min degree*) *gcd-mute-fun A*

lemma *det-field-poly*[*simp*]: *det-field-poly* = *det*
by (*intro ext*, *auto simp*: *det-field-poly-def*)

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* $\Rightarrow$ *shows* **where**
shows-vec v $\equiv$ *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
by *standard* (*simp-all add*: *shows-vec-def show-law-simps shows-prec-vec-def shows-list-vec-def*)
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
by *standard* (*simp-all add*: *shows-mat-def show-law-simps shows-prec-mat-def shows-list-mat-def*)
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 ..
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 ..
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 ·_v v)

lemma eigenvector-pow: assumes A: A ∈ carrier-mat n n
  and ev: eigenvector A v (k :: 'a :: comm-ring-1)
  shows A ^_m i *_v v = k^i ·_v v
proof −
  let ?G = monoid-vec TYPE ('a) n
  from A have dim: dim-row A = n by auto
  from ev[unfolded eigenvector-def dim]
  have v: v ∈ carrier-vec n and Av: A *_v v = k ·_v v by auto
  interpret v: comm-group ?G by (rule comm-group-vec)
  show ?thesis
  proof (induct i)
    case 0
    show ?case using v dim by simp
  next
    case (Suc i)
    define P where P = A ^_m i
    have P: P ∈ carrier-mat n n using A unfolding P-def by simp
    have A ^_m Suc i = P * A unfolding P-def by simp
    also have ... *_v v = P *_v (A *_v v) using P A v by simp
    also have A *_v v = k ·_v v by (rule Av)
    also have P *_v (k ·_v v) = k ·_v (P *_v v)
      by (rule eq-vecI, insert v P, auto)
    also have (P *_v v) = (k ^ i) ·_v v unfolding P-def by (rule Suc)
    also have k ·_v ((k ^ i) ·_v v) = (k * k ^ i) ·_v v
      by (rule eq-vecI, insert v, auto)
    also have k * k ^ i = k ^(Suc i) by auto
    finally show ?case .
  qed
```

qed

definition *eigenvalue* :: $'a$ :: *comm-ring-1 mat* $\Rightarrow$ $'a \Rightarrow$ *bool* **where**
eigenvalue A k = ($\exists$ *v*. *eigenvector A v k*)

definition *char-matrix* :: $'a$ :: *field mat* $\Rightarrow$ $'a \Rightarrow$ $'a$ *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* $\Longrightarrow$ *char-matrix A e* ∈ *carrier-mat n n*
unfolding *char-matrix-def* **by** *auto*

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 −
from *A* **have** *dim*: *dim-row A* = *n* *dim-col A* = *n* **by** *auto*
{
assume *v*: *v* ∈ *carrier-vec n*
hence *dimv*: *dim-vec v* = *n* **by** *auto*
have (*A* $*_v$ *v* = *e* $\cdot_v$ *v*) = (*A* $*_v$ *v* + (−*e*) $\cdot_v$ *v* = 0_v *n*) (**is** *?id1* = *?id2*)
proof
assume *?id1*
from *arg-cong*[*OF this*, *of* λ *w*. *w* + (−*e*) $\cdot_v$ *v*]
show *?id2* **using** *A v* **by** *auto*
next
assume *?id2*
have *A* $*_v$ *v* + − *e* $\cdot_v$ *v* + *e* $\cdot_v$ *v* = *A* $*_v$ *v* **using** *A v* **by** *auto*
from *arg-cong*[*OF* ‹*?id2*›, *of* λ *w*. *w* + *e* $\cdot_v$ *v*, *unfolded this*]
show *?id1* **using** *A v* **by** *simp*
qed
also have (*A* $*_v$ *v* + (−*e*) $\cdot_v$ *v*) = *char-matrix A e* $*_v$ *v* **unfolding** *char-matrix-def*
by (*rule eq-vecI*, *insert v A dim*, *auto simp*: *add-scalar-prod-distrib*[*of* - *n*])
finally have (*A* $*_v$ *v* = *e* $\cdot_v$ *v*) = (*char-matrix A e* $*_v$ *v* = 0_v *n*) **.**
}
thus *?thesis* **unfolding** *eigenvector-def dim* **by** *blast*
qed

lemma *eigenvalue-char-matrix*: **assumes** *A*: (*A* :: $'a$ :: *field mat*) ∈ *carrier-mat n n*
shows *eigenvalue A e* = ($\exists$ *v*. *v* ∈ *carrier-vec n* ∧ *v* ≠ 0_v *n* ∧ *char-matrix A e* $*_v$ *v* = 0_v *n*)
unfolding *eigenvalue-def eigenvector-char-matrix*[*OF A*] **..**

definition *find-eigenvector* :: $'a$::*field mat* $\Rightarrow$ $'a \Rightarrow$ $'a$ *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*: $A \in$ *carrier-mat n n*
  and *ev*: *eigenvalue A e*
  shows *eigenvector A* (*find-eigenvector A e*) *e*
proof −
  define *B* **where** *B* = *char-matrix A e*
  from *ev*[*unfolded eigenvalue-char-matrix*[*OF A*]] **obtain** *v* **where**
    v: $v \in$ *carrier-vec n v* $\neq 0_v$ *n* **and** *Bv*: $B *_v v = 0_v$ *n* **unfolding** *B-def* **by** *auto*
  have *B*: $B \in$ *carrier-mat n n* **using** *A* **unfolding** *B-def* **by** *simp*
  let *?z* = 0_m (*dim-row A*) *0*
  obtain *C D* **where** *gauss*: *gauss-jordan B ?z* = (*C*,*D*) **by** *force*
  define *w* **where** *w* = *find-base-vector C*
  have *res*: *find-eigenvector A e* = *w* **unfolding** *w-def find-eigenvector-def Let-def gauss B-def*[*symmetric*]
    by *simp*
  have *?z* $\in$ *carrier-mat n 0* **using** *A* **by** *auto*
  note *gauss-0* = *gauss-jordan*[*OF B this gauss*]
  hence *C*: $C \in$ *carrier-mat n n* **by** *auto*
  from *gauss-0*(*1*)[*OF v*(*1*)] *Bv* **have** *Cv*: $C *_v v = 0_v$ *n* **by** *auto*
  {
    assume *C*: *C* = 1_m *n*
    have *False* **using** *id Cv v* **unfolding** *C* **by** *auto*
  }
  hence *C1*: $C \neq 1_m$ *n* **by** *auto*
  from *find-base-vector-not-1*[*OF gauss-jordan-row-echelon*[*OF B gauss*] *C C1*]
  have *w*: $w \in$ *carrier-vec n w* $\neq 0_v$ *n* **and** *id*: $C *_v w = 0_v$ *n* **unfolding** *w-def* **by** *auto*
  from *gauss-0*(*1*)[*OF w*(*1*)] *id* **have** *Bw*: $B *_v w = 0_v$ *n* **by** *simp*
  from *w Bw* **have** *eigenvector A w e*
    unfolding *eigenvector-char-matrix*[*OF A*] *B-def* **by** *auto*
  thus *?thesis* **unfolding** *res* **.**
qed

lemma *eigenvalue-imp-nonzero-dim*: **assumes** $A \in$ *carrier-mat n n*
  and *eigenvalue A ev*
  shows $n > 0$
proof (*cases n*)
  case *0*
  from *assms* **obtain** *v* **where** *eigenvector A v ev* **unfolding** *eigenvalue-def* **by** *auto*
  from *this*[*unfolded eigenvector-def*] *assms 0*
  have $v \in$ *carrier-vec 0 v* $\neq 0_v$ *0* **by** *auto*
  hence *False* **by** *auto*
  thus *?thesis* **by** *auto*
qed *simp*

lemma *eigenvalue-det*: **assumes** *A*: (*A* :: *'a* :: *field mat*) ∈ *carrier-mat n n* **shows**
eigenvalue A e = (*det* (*char-matrix A e*) = *0*)
proof −
from *A* **have** *cA*: *char-matrix A e* ∈ *carrier-mat n n* **by** *auto*
show *?thesis*
unfolding *eigenvalue-char-matrix*[*OF A*]
unfolding *id det-0-negate*[*OF cA*] *det-0-iff-vec-prod-zero*[*OF cA*]
eigenvalue-def **by** *auto*
qed

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*
unfolding *char-poly-matrix-def* **by** *auto*

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*)
unfolding *char-poly-defs*
by (*rule eq-matI*, *insert A*, *auto simp*: *smult-mat-def hom-distribs*)

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 −
interpret *map-poly-hom*: *map-poly-comm-ring-hom hom*..
show *?thesis*
unfolding *char-poly-def map-poly-hom.hom-det*[*symmetric*] *char-poly-matrix-hom*[*OF A*] ..
qed

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 −
let *?A* = mat_h *A*
let *?v* = vec_h *v*
let *?ev* = *hom ev*

from *ev*[*unfolded eigenvector-def*] *A*
have *v*: $v \in$ *carrier-vec n v* $\neq 0_v$ *n A* $*_v$ *v* = *ev* $\cdot_v$ *v* **by** *auto*
from *v*(*1*) **have** *v1*: *?v* $\in$ *carrier-vec n* **by** *simp*
from *v*(*1*−*2*) **obtain** *i* **where** *i* < *n* **and** *v* \$ *i* $\neq$ *0* **by** *force*
with *v*(*1*) **have** *?v* \$ *i* $\neq$ *0* **by** *auto*
hence *v2*: *?v* $\neq 0_v$ *n* **using** ‹*i* < *n*› *v*(*1*) **by** *force*
from *arg-cong*[*OF v*(*3*), *of* vec_h, *unfolded mult-mat-vec-hom*[*OF A v*(*1*)] *vec-hom-smult*]
have *v3*: *?A* $*_v$ *?v* = *?ev* $\cdot_v$ *?v* **.**
from *v1 v2 v3*
show *?thesis* **unfolding** *eigenvector-def* **using** *A* **by** *auto*
qed

lemma *eigenvalue-hom*: **assumes** *A*: *A* $\in$ *carrier-mat n n*
and *ev*: *eigenvalue A ev*
shows *eigenvalue* (mat_h *A*) (*hom ev*)
using *eigenvector-hom*[*OF A*, *of - ev*] *ev*
unfolding *eigenvalue-def* **by** *auto*

lemma *eigenvector-hom-rev*: **assumes** *A*: *A* $\in$ *carrier-mat n n*
and *ev*: *eigenvector* (mat_h *A*) (vec_h *v*) (*hom ev*)
shows *eigenvector A v ev*
proof −
let *?A* = mat_h *A*
let *?v* = vec_h *v*
let *?ev* = *hom ev*
from *ev*[*unfolded eigenvector-def*] *A*
have *v*: $v \in$ *carrier-vec n ?v* $\neq 0_v$ *n ?A* $*_v$ *?v* = *?ev* $\cdot_v$ *?v* **by** *auto*
from *v*(*1*−*2*) **obtain** *i* **where** *i* < *n* **and** *v* \$ *i* $\neq$ *0* **by** *force*
with *v*(*1*) **have** *v* \$ *i* $\neq$ *0* **by** *auto*
hence *v2*: *v* $\neq 0_v$ *n* **using** ‹*i* < *n*› *v*(*1*) **by** *force*
from *vec-hom-inj*[*OF v*(*3*)[*folded mult-mat-vec-hom*[*OF A v*(*1*)] *vec-hom-smult*]]
have *v3*: *A* $*_v$ *v* = *ev* $\cdot_v$ *v* **.**
from *v*(*1*) *v2 v3*
show *?thesis* **unfolding** *eigenvector-def* **using** *A* **by** *auto*
qed

end

lemma *poly-det-cong*: **assumes** *A*: *A* $\in$ *carrier-mat n n*
and *B*: *B* $\in$ *carrier-mat n n*
and *poly*: $\bigwedge$ *i j*. *i* < *n* $\Longrightarrow$ *j* < *n* $\Longrightarrow$ *poly* (*B* \$\$ (*i*,*j*)) *k* = *A* \$\$ (*i*,*j*)
shows *poly* (*det B*) *k* = *det A*
proof −
show *?thesis*
unfolding *det-def* ′[*OF A*] *det-def* ′[*OF B*] *poly-sum poly-mult poly-prod*
proof (*rule sum.cong*[*OF refl*])
fix *x*
assume *x*: *x* $\in$ {*p*. *p permutes* {*0*..<*n*}}

let *?l* = $\prod$ *ka* = *0..<n. poly* (*B* $$ (*ka*, *x ka*)) *k*
let *?r* = $\prod$ *i* = *0..<n. A* $$ (*i*, *x i*)
have *id*: *?l* = *?r*
by (*rule prod.cong*[*OF refl poly*], *insert x*, *auto*)
show *poly* (*signof x*) *k* ∗ *?l* = *signof x* ∗ *?r*
by (*cases x rule*: *sign-cases*) (*simp-all add*: *id*)
qed
qed

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*)) **unfolding** *char-poly-def*
by (*rule poly-det-cong*[*of - n*], *insert A*, *auto simp*: *char-poly-matrix-def char-matrix-def*)

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*
unfolding *eigenvalue-det*[*OF A*] *char-poly-matrix*[*OF A*]
by (*subst det-0-negate*[*of - n*], *insert A*, *auto*)

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*)
using *A ut* **unfolding** *upper-triangular-def char-poly-matrix-def* **by** *auto*

lemma *char-poly-upper-triangular*:
char-poly A = ($\prod$ *a* ← *diag-mat A*. [:− *a*, *1*:])
proof −
from *A* **have** *cA*: *char-poly-matrix A* ∈ *carrier-mat n n* **by** *simp*
show *?thesis*
unfolding *char-poly-def det-upper-triangular* [*OF char-poly-matrix-upper-triangular cA*]
by (*rule arg-cong*[**where** *f* = *prod-list*], *unfold list-eq-iff-nth-eq*, *insert cA A*, *auto simp*: *diag-mat-def*
char-poly-matrix-def)
qed
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 −
from *A B* **have** *dim*: *dim-row A* = *nr dim-col A* = *n dim-row B* = *n dim-col B* = *nc* **by** *auto*
{
fix *i j*
have $i < nr \Longrightarrow j < nc \Longrightarrow$
row (*map-mat* ($\lambda a.$ [:*a*:]) *A*) *i* · *col* (*map-mat* ($\lambda a.$ [:*a*:]) *B*) *j* = [:(*row A i* · *col B j*):]
unfolding *scalar-prod-def*
by (*auto simp*: *dim ac-simps*, *induct n*, *auto*)
} **note** *id* = *this*
{
fix *i j*
have $i < nr \Longrightarrow j < nc \Longrightarrow$
[:(*row A i* · *col B j*):] ∗ *p* = *row* (*map-mat* (λ *a.* [: *a* :] ∗ *p*) *A*) *i* · *col* (*map-mat* ($\lambda a.$ [:*a*:]) *B*) *j*
unfolding *scalar-prod-def*
by (*auto simp*: *dim ac-simps smult-sum*)
} **note** *left* = *this*
{
fix *i j*
have $i < nr \Longrightarrow j < nc \Longrightarrow$
[:(*row A i* · *col B j*):] ∗ *p* = *row* (*map-mat* (λ *a.* [: *a* :]) *A*) *i* · *col* (*map-mat* ($\lambda a.$ [:*a*:] ∗ *p*) *B*) *j*
unfolding *scalar-prod-def*
by (*auto simp*: *dim ac-simps smult-sum*)
} **note** *right* = *this*
show *?id*
by (*rule eq-matI*, *insert id*, *auto simp*: *dim*)
show *?left*
by (*rule eq-matI*, *insert left*, *auto simp*: *dim*)
show *?right*
by (*rule eq-matI*, *insert right*, *auto simp*: *dim*)
qed

lemma *char-poly-similar*: **assumes** *similar-mat A* (*B* :: *'a* :: *comm-ring-1 mat*)
shows *char-poly A* = *char-poly B*
proof −
from *similar-matD*[*OF assms*] **obtain** *n P Q* **where**
carr: {*A*, *B*, *P*, *Q*} ⊆ *carrier-mat n n* (**is** - ⊆ *?C*)
and *PQ*: *P* ∗ *Q* = 1_m *n*
and *AB*: *A* = *P* ∗ *B* ∗ *Q* **by** *auto*
hence *A*: *A* ∈ *?C* **and** *B*: *B* ∈ *?C* **and** *P*: *P* ∈ *?C* **and** *Q*: *Q* ∈ *?C* **by** *auto*
let *?m* = λ *a.* [: −*a* :]
let *?P* = *map-mat* (λ *a.* [: *a* :]) *P*
let *?Q* = *map-mat* (λ *a.* [: *a* :]) *Q*
let *?B* = *map-mat ?m B*
let *?I* = *map-mat* (λ *a.* [: *a* :]) (1_m *n*)
let *?XI* = [:*0*, *1*:] $\cdot_m$ 1_m *n*

from A B **have** *dim*: *dim-row* $A = n$ *dim-row* $B = n$ **by** *auto*
have *cong*: $\bigwedge x\ y\ z.\ x = y \Longrightarrow x * z = y * z$ **by** *auto*
have *id*: $?m = (\lambda\ a :: 'a.\ [:\ a\ :] * [:\ -1\ :])$ **by** (*intro ext*, *auto*)
have *char-poly* $A = det\ (?XI + map\text{-}mat\ (\lambda a.\ [:-\ a:])\ (P * B * Q))$ **unfolding** *char-poly-defs dim*
by (*simp add*: *AB*)
also have $?XI = ?P * ?XI * ?Q$ (**is** - = *?left*)
proof −
have $?P * ?XI = [:0,\ 1:] \cdot_m (?P * 1_m\ n)$
by (*rule mult-smult-distrib*[*of* - *n n* - *n*], *insert P*, *auto*)
also have $?P * 1_m\ n = ?P$ **using** P **by** *simp*
also have $([:\ 0,\ 1:] \cdot_m ?P) * ?Q = [:\ 0,\ 1:] \cdot_m (?P * ?Q)$
by (*rule mult-smult-assoc-mat*, *insert P Q*, *auto*)
also have $?P * ?Q = ?I$ **unfolding** *PQ*[*symmetric*]
by (*rule map-poly-mult*[*symmetric*, *OF P Q*])
also have $[:\ 0,\ 1:] \cdot_m ?I = ?XI$
by *rule auto*
finally show *?thesis* **..**
qed
also have *map-mat* $?m\ (P * B * Q) = ?P * ?B * ?Q$ (**is** - = *?right*)
unfolding *id*
by (*subst map-poly-mult*[*OF mult-carrier-mat*[*OF P B*] *Q*],
subst map-poly-mult(*3*)[*OF P B*], *simp*)
also have $?left + ?right = (?P * ?XI + ?P * ?B) * ?Q$
by (*rule add-mult-distrib-mat*[*symmetric*, *of* - *n n*], *insert B P Q*, *auto*)
also have $?P * ?XI + ?P * ?B = ?P * (?XI + ?B)$
by (*rule mult-add-distrib-mat*[*symmetric*, *of* - *n n*], *insert B P Q*, *auto*)
also have $det\ (?P * (?XI + ?B) * ?Q) = det\ ?P * det\ (?XI + ?B) * det\ ?Q$
by (*rule trans*[*OF det-mult*[*of* - *n*] *cong*[*OF det-mult*]], *insert P Q B*, *auto*)
also have $\ldots = (det\ ?P * det\ ?Q) * det\ (?XI + ?B)$ **by** (*simp add*: *ac-simps*)
also have $det\ (?XI + ?B) =$ *char-poly B* **unfolding** *char-poly-defs dim* **by** *simp*
also have $det\ ?P * det\ ?Q = det\ (?P * ?Q)$
by (*rule det-mult*[*symmetric*], *insert P Q*, *auto*)
also have $?P * ?Q = ?I$ **unfolding** *PQ*[*symmetric*]
by (*rule map-poly-mult*[*symmetric*, *OF P Q*])
also have $det\ \ldots =$ *prod-list* (*diag-mat* $?I$)
by (*rule det-upper-triangular*[*of* - *n*], *auto*)
also have $\ldots = 1$ **unfolding** *prod-list-diag-prod*
by (*rule prod.neutral*) *simp*
finally show *?thesis* **by** *simp*
qed

lemma *degree-signof-mult*[*simp*]: *degree* (*signof* $p * q$) = *degree* q
by (*cases p rule*: *sign-cases*) *simp-all*

lemma *degree-monic-char-poly*: **assumes** A: $A \in$ *carrier-mat n n*
shows *degree* (*char-poly* A) $= n \wedge$ *coeff* (*char-poly* A) $n = 1$
proof −
from A **have** A': $[:0,\ 1:] \cdot_m 1_m\ (dim\text{-}row\ A) + map\text{-}mat\ (\lambda a.\ [:-\ a:])\ A \in$

carrier-mat n n **by** *auto*
  from *A* **have** *dA*: *dim-row A = n* **by** *simp*
  show *?thesis*
    unfolding *char-poly-defs det-def'*[*OF A'*]
  proof (*rule degree-lcoeff-sum*[*of - id*], *auto simp*: *finite-permutations permutes-id dA*)
    have *both*: *degree* ($\prod$ *i = 0..<n.* ([:*0*, *1*:] $\cdot_m$ 1_m *n* + *map-mat* (λ*a.* [:− *a*:]) *A*) $$ (*i*, *i*)) = *n* ∧
      coeff ($\prod$ *i = 0..<n.* ([:*0*, *1*:] $\cdot_m$ 1_m *n* + *map-mat* (λ*a.* [:− *a*:]) *A*) $$ (*i*, *i*)) *n = 1*
      by (*rule degree-prod-monic*, *insert A*, *auto*)
    from *both* **show** *degree* ($\prod$ *i = 0..<n.* ([:*0*, *1*:] $\cdot_m$ 1_m *n* + *map-mat* (λ*a.* [:− *a*:]) *A*) $$ (*i*, *i*)) = *n* **..**
    from *both* **show** *coeff* ($\prod$ *i = 0..<n.* ([:*0*, *1*:] $\cdot_m$ 1_m *n* + *map-mat* (λ*a.* [:− *a*:]) *A*) $$ (*i*, *i*)) *n = 1* **..**
  next
    fix *p*
    assume *p*: *p permutes* {*0..<n*}
      and *p* ≠ *id*
    then obtain *i* **where** *i*: *i < n* **and** *pi*: *p i* ≠ *i*
      by (*metis atLeastLessThan-iff order-refl permutes-natset-le*)
    show *degree* ($\prod$ *i = 0..<n.* ([:*0*, *1*:] $\cdot_m$ 1_m *n* + *map-mat* (λ*a.* [:− *a*:]) *A*) $$ (*i*, *p i*)) < *n*
      by (*rule degree-prod-sum-lt-n*[*OF - i*], *insert p i pi A*, *auto*)
  qed
qed

lemma *char-poly-factorized*: **fixes** *A* :: *complex mat*
  assumes *A*: *A* ∈ *carrier-mat n n*
  shows ∃ *as. char-poly A* = ($\prod$ *a* ← *as.* [:− *a*, *1*:]) ∧ *length as = n*
proof −
  let *?p = char-poly A*
  from *fundamental-theorem-algebra-factorized*[*of ?p*] **obtain** *as*
   where *Polynomial.smult* (*coeff ?p* (*degree ?p*)) ($\prod$ *a*←*as.* [:− *a*, *1*:]) = *?p* **by** *blast*
  also have *coeff ?p* (*degree ?p*) = *1* **using** *degree-monic-char-poly*[*OF A*] **by** *simp*
  finally have *cA*: *?p* = ($\prod$ *a*←*as.* [:− *a*, *1*:]) **by** *simp*
  from *degree-monic-char-poly*[*OF A*] **have** *degree ?p = n* **..**
  with *degree-linear-factors*[*of uminus as*, *folded cA*] **have** *length as = n* **by** *auto*
  with *cA* **show** *?thesis* **by** *blast*
qed

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 −
  let *?cm* = λ *A.* [:*0*, *1*:] $\cdot_m$ 1_m (*dim-row A*) +

map-mat ($\lambda a.$ [:− *a*:]) *A*
let *?B2* = *map-mat* ($\lambda a.$ [:− *a*:]) *A2*
have *char-poly ?A* = *det* (*?cm ?A*)
unfolding *char-poly-defs* **using** *A1 A3* **by** *simp*
also have *?cm ?A* = *four-block-mat* (*?cm A1*) *?B2* (0_m *n 1*) (*?cm A3*)
by (*rule eq-matI*, *insert A1 A2 A3*, *auto simp*: *one-poly-def*)
also have *det* ... = *det* (*?cm A1*) ∗ *det* (*?cm A3*)
by (*rule det-four-block-mat-lower-left-zero-col*[*OF* - - *refl*], *insert A1 A2 A3*, *auto*)
also have ... = *?cp1* ∗ *?cp3* **unfolding** *char-poly-defs* **..**
finally show *?thesis* **.**
qed

lemma *char-poly-transpose-mat*[*simp*]: **assumes** *A*: *A* ∈ *carrier-mat n n*
shows *char-poly* (*transpose-mat A*) = *char-poly A*
proof −
let *?A* = [:*0*, *1*:] $\cdot_m$ 1_m (*dim-row A*) + *map-mat* ($\lambda a.$ [:− *a*:]) *A*
have *A'*: *?A* ∈ *carrier-mat n n* **using** *A* **by** *auto*
show *?thesis* **unfolding** *char-poly-defs*
by (*subst det-transpose*[*symmetric*, *OF A'*], *rule arg-cong*[*of* - - *det*],
insert A, *auto*)
qed

lemma *pderiv-char-poly*: **fixes** *A* :: *'a* :: *idom mat*
assumes *A*: *A* ∈ *carrier-mat n n*
shows *pderiv* (*char-poly A*) = ($\sum i < n.$ *char-poly* (*mat-delete A i i*))
proof −
let *?det* = *Determinant.det*
let *?m* = *map-mat* ($\lambda a.$ [:− *a*:])
let *?lam* = [:*0*, *1*:] $\cdot_m$ 1_m *n* :: *'a poly mat*
from *A* **have** *id*: *dim-row A* = *n* **by** *auto*

define *mA* **where** *mA* = *?m A*
define *lam* **where** *lam* = *?lam*
let *?sum* = *lam* + *mA*
define *Sum* **where** *Sum* = *?sum*
have *mA*: *mA* ∈ *carrier-mat n n* **and**
lam: *lam* ∈ *carrier-mat n n* **and**
Sum: *Sum* ∈ *carrier-mat n n*
using *A* **unfolding** *mA-def Sum-def lam-def* **by** *auto*
let *?P* = {*p*. *p permutes* {*0*..<*n*}}
let *?e* = λ *p*. ($\prod i = 0..<n.$ *Sum* $$ (*i*, *p i*))
let *?f* = λ *p a*. *signof p* ∗ ($\prod i \in \{0..<n\} - \{a\}.$ *Sum* $$ (*i*, *p i*)) ∗ *pderiv* (*Sum* $$ (*a*, *p a*))
let *?g* = λ *p a*. *signof p* ∗ ($\prod i \in \{0..<n\} - \{a\}.$ *Sum* $$ (*i*, *p i*))
define *f* **where** *f* = *?f*
define *g* **where** *g* = *?g*
{
fix *p*

assume *p*: *p permutes* $\{0\ ..< n\}$
have *pderiv* (*signof p* :: *'a poly*) = *0*
by (*cases p rule*: *sign-cases*) (*simp-all add*: *pderiv-minus*)
hence *pderiv* (*signof p* ∗ *?e p*) = *signof p* ∗ *pderiv* ($\prod i = 0..<n.$ *Sum* $$ (*i*, *p i*))
unfolding *pderiv-mult* **by** *auto*
also have *signof p* ∗ *pderiv* ($\prod i = 0..<n.$ *Sum* $$ (*i*, *p i*)) =
($\sum a = 0..<n.$ *f p a*)
unfolding *pderiv-prod sum-distrib-left f-def* **by** (*simp add*: *ac-simps*)
also note *calculation*
} **note** *to-f* = *this*
{
fix *a*
assume *a*: *a* < *n*
have *Psplit*: *?P* = {*p*. *p permutes* $\{0..<n\}$ ∧ *p a* = *a*} ∪ (*?P* − {*p*. *p a* = *a*}) (**is** - = *?Pa* ∪ *?Pz*) **by** *auto*
{
fix *p*
assume *p*: *p permutes* $\{0\ ..< n\}$ *p a* ≠ *a*
hence *pderiv* (*Sum* $$ (*a*, *p a*)) = *0* **unfolding** *Sum-def lam-def mA-def* **using** *a p A* **by** *auto*
hence *f p a* = *0* **unfolding** *f-def* **by** *auto*
} **note** *0* = *this*
{
fix *p*
assume *p*: *p permutes* $\{0\ ..< n\}$ *p a* = *a*
hence *pderiv* (*Sum* $$ (*a*, *p a*)) = *1* **unfolding** *Sum-def lam-def mA-def* **using** *a p A*
by (*auto simp*: *pderiv-pCons*)
hence *f p a* = *g p a* **unfolding** *f-def g-def* **by** *auto*
} **note** *fg* = *this*
let *?n* = *n* − *1*
from *a* **have** *n*: *Suc ?n* = *n* **by** *simp*
let *?B* = [:*0*, *1*:] $\cdot_m$ 1_m *?n* + *?m* (*mat-delete A a a*)
have *B*: *?B* ∈ *carrier-mat ?n ?n* **using** *A* **by** *auto*
have *sum* (λ *p*. *f p a*) *?P* = *sum* (λ *p*. *f p a*) *?Pa* + *sum* (λ *p*. *f p a*) *?Pz*
by (*subst sum.union-disjoint*[*symmetric*], *auto simp*: *finite-permutations Psplit*[*symmetric*])
also have ... = *sum* (λ *p*. *f p a*) *?Pa*
by (*subst* (*2*) *sum.neutral*, *insert 0*, *auto*)
also have ... = *sum* (λ *p*. *g p a*) *?Pa*
by (*rule sum.cong*, *auto simp*: *fg*)
also have ... = *?det ?B*
unfolding *det-def'*[*OF B*]
unfolding *permutation-fix*[*of a ?n a*, *unfolded n*, *OF a a*]
unfolding *sum.reindex*[*OF permutation-insert-inj-on*[*of a ?n a*, *unfolded n*, *OF a a*]] *o-def*
proof (*rule sum.cong*[*OF refl*])
fix *p*
let *?Q* = {*p*. *p permutes* $\{0..<?n\}$}

assume $p \in$ *?Q*
hence *p*: *p permutes* $\{0 ..< ?n\}$ **by** *auto*
let *?p* = *permutation-insert a a p*
let *?i* = *insert-index a*
have *sign*: *signof ?p* = *signof p*
unfolding *signof-permutation-insert*[*OF p*, *unfolded n*, *OF a a*] **by** *simp*
show *g* (*permutation-insert a a p*) *a* = *signof p* $*$ ($\prod$ *i* = *0..<?n. ?B* $$ (*i*, *p i*))
unfolding *g-def sign*
proof (*rule arg-cong*[*of - -* ($*$) (*signof p*)])
have ($\prod$ $i \in \{0..<n\} - \{a\}$. *Sum* $$ (*i*, *?p i*)) =
prod (($$) *Sum*) (($\lambda x.$ (*x*, *?p x*)) ‘ ($\{0..<n\} - \{a\}$))
unfolding *prod.reindex*[*OF inj-on-convol-ident*, *of - ?p*] *o-def* **..**
also have ... = ($\prod$ *ii* $\in \{(i', ?p\ i') \mid i'.\ i' \in \{0..<n\} - \{a\}\}$. *Sum* $$ *ii*)
by (*rule prod.cong*, *auto*)
also have ... = *prod* (($$) *Sum*) ((λ *i*. (*?i i*, *?i* (*p i*))) ‘ $\{0 ..< ?n\}$)
unfolding *Determinant.foo*[*of a ?n a*, *unfolded n*, *OF a a p*]
by (*rule arg-cong*[*of - - prod -*], *auto*)
also have ... = *prod* (λ *i*. *Sum* $$ (*?i i*, *?i* (*p i*))) $\{0 ..< ?n\}$
proof (*subst prod.reindex*, *unfold o-def*)
show *inj-on* ($\lambda i.$ (*?i i*, *?i* (*p i*))) $\{0..<?n\}$ **using** *insert-index-inj-on*[*of a*]
by (*auto simp*: *inj-on-def*)
qed *simp*
also have ... = ($\prod$ *i* = *0..<?n. ?B* $$ (*i*, *p i*))
proof (*rule prod.cong*[*OF refl*], *rename-tac i*)
fix *j*
assume $j \in \{0 ..< ?n\}$
hence *j*: *j* < *?n* **by** *auto*
with *p* **have** *pj*: *p j* < *?n* **by** *auto*
from *j pj* **have** *jj*: *?i j* < *n* *?i* (*p j*) < *n* **by** (*auto simp*: *insert-index-def*)
let *?jj* = (*?i j*, *?i* (*p j*))
note *index-adj* = *mat-delete-index*[*of - ?n*, *unfolded n*, *OF - a a j pj*]
have *Sum* $$ *?jj* = *lam* $$ *?jj* + *mA* $$ *?jj* **unfolding** *Sum-def* **using** *jj A lam mA* **by** *auto*
also have ... = *?B* $$ (*j*, *p j*)
unfolding *index-adj*[*OF mA*] *index-adj*[*OF lam*] **using** *j pj A*
by (*simp add*: *mA-def lam-def mat-delete-def*)
finally show *Sum* $$ *?jj* = *?B* $$ (*j*, *p j*) **.**
qed
finally
show ($\prod$ $i \in \{0..<n\} - \{a\}$. *Sum* $$ (*i*, *?p i*)) = ($\prod$ *i* = *0..<?n. ?B* $$ (*i*, *p i*)) **.**
qed
qed
also have ... = *char-poly* (*mat-delete A a a*) **unfolding** *char-poly-def char-poly-matrix-def*
using *A* **by** *simp*
also note *calculation*
} **note** *to-char-poly* = *this*
have *pderiv* (*char-poly A*) = *pderiv* (*?det Sum*)

unfolding *char-poly-def char-poly-matrix-def id lam-def mA-def Sum-def* **by** *auto*
also have ... = *sum* (λ *p*. *pderiv* (*signof p* ∗ *?e p*)) *?P* **unfolding** *det-def′*[*OF Sum*]
pderiv-sum **by** (*rule sum.cong*, *auto*)
also have ... = *sum* (λ *p*. ($\sum$ *a* = *0*..<*n*. *f p a*)) *?P*
by (*rule sum.cong*[*OF refl*], *subst to-f*, *auto*)
also have ... = ($\sum$ *a* = *0*..<*n*. *sum* (λ *p*. *f p a*) *?P*)
by (*rule sum.swap*)
also have ... = ($\sum$ *a* <*n*. *char-poly* (*mat-delete A a a*))
by (*rule sum.cong*, *auto simp*: *to-char-poly*)
finally show *?thesis* **.**
qed

lemma *char-poly-0-column*: **fixes** *A* :: *′a* :: *idom mat*
assumes *0*: $\bigwedge$ *j*. *j* < *n* ⟹ *A* $$ (*j*,*i*) = *0*
and *A*: *A* ∈ *carrier-mat n n*
and *i*: *i* < *n*
shows *char-poly A* = *monom 1 1* ∗ *char-poly* (*mat-delete A i i*)
proof −
let *?n* = *n* − *1*
let *?A* = *mat-delete A i i*
let *?sum* = [:*0*, *1*:] $\cdot_m$ 1_m *n* + *map-mat* (λ*a*. [:− *a*:]) *A*
define *Sum* **where** *Sum* = *?sum*
let *?f* = λ *j*. *Sum* $$ (*j*, *i*) ∗ *cofactor Sum j i*
have *Sum*: *Sum* ∈ *carrier-mat n n* **using** *A* **unfolding** *Sum-def* **by** *auto*
from *A* **have** *id*: *dim-row A* = *n* **by** *auto*
have *char-poly A* = ($\sum$ *j*<*n*. *?f j*)
unfolding *char-poly-def*[*of A*] *char-poly-matrix-def*
using *laplace-expansion-column*[*OF Sum i*] **unfolding** *Sum-def* **using** *A* **by** *simp*
also have ... = *?f i* + *sum ?f* ({..<*n*} − {*i*})
by (*rule sum.remove*[*of - i*], *insert i*, *auto*)
also have ... = *?f i*
proof (*subst sum.neutral*, *intro ballI*)
fix *j*
assume *j* ∈ {..<*n*} − {*i*}
hence *j*: *j* < *n* **and** *ji*: *j* ≠ *i* **by** *auto*
show *?f j* = *0* **unfolding** *Sum-def* **using** *ji j i A 0*[*OF j*] **by** *simp*
qed *simp*
also have *?f i* = [:*0*, *1*:] ∗ (*cofactor Sum i i*)
unfolding *Sum-def* **using** *i A 0*[*OF i*] **by** *simp*
also have *cofactor Sum i i* = *det* (*mat-delete Sum i i*)
unfolding *cofactor-def* **by** *simp*
also have ... = *char-poly ?A*
unfolding *char-poly-def char-poly-matrix-def Sum-def*
proof (*rule arg-cong*[*of - - det*])
show *mat-delete ?sum i i* = [:*0*, *1*:] $\cdot_m$ 1_m (*dim-row ?A*) + *map-mat* (λ*a*. [:− *a*:]) *?A*

using *i A* **by** (*auto simp*: *mat-delete-def*)
qed
also have [:*0*, *1*:] = (*monom 1 1* :: *'a poly*) **by** (*rule x-as-monom*)
finally show *?thesis* **.**
qed

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*
unfolding *mat-erase-def carrier-mat-def* **by** *simp*

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*) = (∑ *i* < *n*. *char-poly* (*mat-erase A i i*))
proof −
show *?thesis* **unfolding** *pderiv-char-poly*[*OF A*] *sum-distrib-left*
proof (*rule sum.cong*[*OF refl*])
fix *i*
assume *i* ∈ {..<*n*}
hence *i*: *i* < *n* **by** *simp*
have *mA*: *mat-erase A i i* ∈ *carrier-mat n n* **using** *A* **by** *simp*
show *monom 1 1* ∗ *char-poly* (*mat-delete A i i*) = *char-poly* (*mat-erase A i i*)
by (*subst char-poly-0-column*[*OF - mA i*], (*insert i A*, *force simp*: *mat-erase-def*),
rule arg-cong[*of - - λ x. f* ∗ *char-poly x* **for** *f*],
auto simp: *mat-delete-def mat-erase-def*)
qed
qed

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*
unfolding *jordan-block-def* **by** *auto*

lemma *jordan-block-carrier*[*simp*]: *jordan-block n k* ∈ *carrier-mat n n*
unfolding *carrier-mat-def* **by** *auto*

lemma *jordan-block-char-poly*: *char-poly* (*jordan-block n a*) = [: −*a*, *1*:]^*n*
unfolding *char-poly-defs* **by** (*subst det-upper-triangular*[*of - n*], *auto simp*: *prod-list-diag-prod*)

lemma *jordan-block-pow-carrier*[*simp*]:
jordan-block n a ^_m *r* ∈ *carrier-mat n n* **by** *auto*
lemma *jordan-block-pow-dim*[*simp*]:
dim-row (*jordan-block n a* ^_m *r*) = *n dim-col* (*jordan-block n a* ^_m *r*) = *n* **by** *auto*

lemma *jordan-block-pow*: (*jordan-block n* (*a* :: *'a* :: *comm-ring-1*)) ^_m *r* =
mat n n (λ (*i*,*j*). *if i* ≤ *j then of-nat* (*r choose* (*j* − *i*)) ∗ *a* ^ (*r* + *i* − *j*) *else 0*)
proof (*induct r*)
case *0*
{
fix *i j* :: *nat*
assume *i* ≠ *j i* ≤ *j*
hence *j* − *i* > *0* **by** *auto*
hence *0 choose* (*j* − *i*) = *0* **by** *simp*
} **note** [*simp*] = *this*
show *?case*
by (*simp*, *rule eq-matI*, *auto*)
next
case (*Suc r*)
let *?jb* = *jordan-block n a*
let *?rij* = λ *r i j*. *of-nat* (*r choose* (*j* − *i*)) ∗ *a* ^ (*r* + *i* − *j*)
let *?v* = λ *i j*. *if i* ≤ *j then of-nat* (*r choose* (*j* − *i*)) ∗ *a* ^ (*r* + *i* − *j*) *else 0*
have *?jb* ^_m *Suc r* = *mat n n* (λ (*i*,*j*). *if i* ≤ *j then ?rij r i j else 0*) ∗ *?jb* **by** (*simp add*: *Suc*)
also have . . . = *mat n n* (λ (*i*,*j*). *if i* ≤ *j then ?rij* (*Suc r*) *i j else 0*)
proof −
{
fix *j*
assume *j*: *j* < *n*
hence *col*: *col* (*jordan-block n a*) *j* = *vec n* (λ*i*. *if i* = *j then a else if Suc i* = *j then 1 else 0*)
unfolding *jordan-block-def col-mat*[*OF j*] **by** *simp*
fix *f*
have *vec n f* · *col* (*jordan-block n a*) *j* = (*f j* ∗ *a* + (*if j* = *0 then 0 else f* (*j*

```
− 1)))
    proof −
      define p where p = (λ i. vec n f $ i ∗ col (jordan-block n a) j $ i)
      have vec n f · col (jordan-block n a) j = (∑ i = 0 ..< n. p i)
        unfolding scalar-prod-def p-def by simp
      also have . . . = p j + sum p ({0 ..< n} − {j}) using j
        by (subst sum.remove[of - j], auto)
      also have p j = f j ∗ a unfolding p-def col using j by auto
      also have sum p ({0 ..< n} − {j}) = (if j = 0 then 0 else f (j − 1))
      proof (cases j)
        case 0
        have sum p ({0 ..< n} − {j}) = 0
          by (rule sum.neutral, auto simp: p-def col 0)
        thus ?thesis using 0 by simp
      next
        case (Suc jj)
        with j have jj: jj ∈ {0 ..< n} − {j} by auto
        have sum p ({0 ..< n} − {j}) = p jj + sum p ({0 ..< n} − {j} − {jj})
          by (subst sum.remove[OF - jj], auto)
        also have p jj = f (j − 1) unfolding p-def col using jj
          by (auto simp: Suc)
        also have sum p ({0 ..< n} − {j} − {jj}) = 0
          by (rule sum.neutral, auto simp: p-def col, auto simp: Suc)
        finally show ?thesis unfolding Suc by simp
      qed
      finally show ?thesis .
    qed
  } note scalar-to-sum = this
  {
    fix i j
    assume i: i < n and ij: i > j
    hence j: j < n by auto
    have vec n (?v i) · col (jordan-block n a) j = 0
      unfolding scalar-to-sum[OF j] using ij i j by auto
  } note easy-case = this
  {
    fix i j
    assume j: j < n and ij: i ≤ j
    hence i: i < n and id: ⋀ p q. (if i ≤ j then p else q) = p by auto
    have vec n (?v i) · col (jordan-block n a) j =
      (of-nat (r choose (j − i)) ∗ (a ^ (Suc (r + i − j)))) +
        (if j = 0 then 0
        else if i ≤ j − 1 then of-nat (r choose (j − 1 − i)) ∗ a ^ (r + i − (j −
1)) else 0)
    unfolding scalar-to-sum[OF j]
    using ij by simp
    also have . . . = of-nat (Suc r choose (j − i)) ∗ a ^ (Suc (r + i) − j)
    proof (cases j)
      case (Suc jj)
```

{
  assume $i \leq Suc\ jj$ **and** $\neg\ i \leq jj$
  hence $i = Suc\ jj$ **by** *auto*
  hence $a * a \ \widehat{}\ (r + i - Suc\ jj) = a \ \widehat{}\ (r + i - jj)$ **by** *simp*
}
moreover
{
  assume *ijj*: $i \leq jj$
  have *of-nat* $(r\ choose\ (Suc\ jj - i)) * (a * a \ \widehat{}\ (r + i - Suc\ jj))$
  $+$ *of-nat* $(r\ choose\ (jj - i)) * a \ \widehat{}\ (r + i - jj) =$
    of-nat $(Suc\ r\ choose\ (Suc\ jj - i)) * a \ \widehat{}\ (r + i - jj)$
  proof $(cases\ r + i < jj)$
    case *True*
    hence *gt*: $jj - i > r\ Suc\ jj - i > r\ Suc\ jj - i > Suc\ r$ **by** *auto*
    show *?thesis*
        unfolding *binomial-eq-0*[*OF gt*(*1*)] *binomial-eq-0*[*OF gt*(*2*)] *binomial-eq-0*[*OF gt*(*3*)]
      by *simp*
  next
    case *False*
    hence *ge*: $r + i \geq jj$ **by** *simp*
    show *?thesis*
    proof $(cases\ jj = r + i)$
      case *True*
      have *gt*: $r < Suc\ r$ **by** *simp*
      show *?thesis* **unfolding** *True* **by** (*simp add*: *binomial-eq-0*[*OF gt*])
    next
      case *False*
      with *ge* **have** *lt*: $jj < r + i$ **by** *auto*
      hence $r + i - jj = Suc\ (r + i - Suc\ jj)$ **by** *simp*
      hence *prod*: $a * a \ \widehat{}\ (r + i - Suc\ jj) = a \ \widehat{}\ (r + i - jj)$ **by** *simp*
      from *ijj* **have** *id*: $Suc\ jj - i = Suc\ (jj - i)$ **by** *simp*
      have *binom*: $(Suc\ r\ choose\ (Suc\ jj - i)) =$
        $(r\ choose\ (Suc\ jj - i)) + (r\ choose\ (jj - i))$
        unfolding *id* **by** (*subst binomial-Suc-Suc*, *simp*)
      show *?thesis* **unfolding** *prod binom*
        by (*simp add*: *field-simps*)
    qed
  qed
}
ultimately show *?thesis* **using** *ij* **unfolding** *Suc* **by** *auto*
qed *auto*
finally have *vec n* (*?v i*) $\cdot$ *col* (*jordan-block n a*) *j*
  $=$ *of-nat* $(Suc\ r\ choose\ (j - i)) * a \ \widehat{}\ (Suc\ (r + i) - j)$ **.**
} **note** *main-case* = *this*
show *?thesis*
  by (*rule eq-matI*, *insert easy-case main-case*, *auto*)
qed
finally show *?case* **by** *simp*

qed

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*)
unfolding *jordan-matrix-def*
by (*subst dim-diag-block-mat*, *auto*, (*induct n-as*, *auto simp*: *Let-def*)+)

lemma *jordan-matrix-carrier*[*simp*]:
jordan-matrix n-as ∈ *carrier-mat* (*sum-list* (*map fst n-as*)) (*sum-list* (*map fst n-as*))
unfolding *carrier-mat-def* **by** *auto*

lemma *jordan-matrix-upper-triangular*: *i* < *sum-list* (*map fst n-as*)
⟹ *j* < *i* ⟹ *jordan-matrix n-as* $$ (*i*,*j*) = *0*
unfolding *jordan-matrix-def*
by (*rule diag-block-upper-triangular*, *auto simp*: *jordan-matrix-def*[*symmetric*])

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*)
unfolding *jordan-matrix-def*
by (*subst diag-block-pow-mat*, *force*, *rule arg-cong*[*of - - diag-block-mat*], *auto*)

lemma *jordan-matrix-char-poly*:
char-poly (*jordan-matrix n-as*) = (∏(*n*, *a*)←*n-as*. [:− *a*, *1*:] ^ *n*)
proof −
let *?n* = *sum-list* (*map fst n-as*)
have *diag-mat*
([:*0*, *1*:] $\cdot_m$ 1_m (*sum-list* (*map fst n-as*)) + *map-mat* (λ*a*. [:− *a*:]) (*jordan-matrix n-as*)) =
concat (*map* (λ(*n*, *a*). *replicate n* [:− *a*, *1*:]) *n-as*) **unfolding** *jordan-matrix-def*
proof (*induct n-as*)
case (*Cons na n-as*)
obtain *n a* **where** *na*: *na* = (*n*,*a*) **by** *force*
let *?n2* = *sum-list* (*map fst n-as*)
note *fbo* = *four-block-one-mat*
note *mz* = *zero-carrier-mat*
note *mo* = *one-carrier-mat*
have *mA*: ⋀ *A*. *A* ∈ *carrier-mat* (*dim-row A*) (*dim-col A*) **unfolding** *carrier-mat-def* **by** *auto*
let *?Bs* = *map* (λ(*x*, *y*). *jordan-block x y*) *n-as*
let *?B* = *diag-block-mat ?Bs*
from *jordan-matrix-dim*[*of n-as*, *unfolded jordan-matrix-def*]
have *dimB*: *dim-row ?B* = *?n2 dim-col ?B* = *?n2* **by** *auto*
hence *B*: *?B* ∈ *carrier-mat ?n2 ?n2* **unfolding** *carrier-mat-def* **by** *simp*
show *?case* **unfolding** *na fbo*

```
    apply (simp add: Let-def fbo[symmetric] del: fbo)
    apply (subst smult-four-block-mat[OF mo mz mz mo])
    apply (subst map-four-block-mat[OF jordan-block-carrier mz mz mA])
    apply (subst add-four-block-mat[of - n n - ?n2 - ?n2], auto simp: dimB B)
    apply (subst diag-four-block-mat[of - n - ?n2], auto simp: dimB B)
    apply (subst Cons, auto simp: jordan-block-def diag-mat-def,
      intro nth-equalityI, auto)
    done
  qed (force simp: diag-mat-def)
  also have prod-list ... = (∏(n, a)←n-as. [:− a, 1:] ^ n)
    by (induct n-as, auto)
  finally
  show ?thesis unfolding char-poly-defs
   by (subst det-upper-triangular[of - ?n], auto simp: jordan-matrix-upper-triangular)
qed

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 ^m k = P * (jordan-matrix n-as)^m k * Q
proof −
  from A have dim: dim-row A = n by auto
  assume obt: ⋀P Q. P ∈ carrier-mat n n ⟹ Q ∈ carrier-mat n n ⟹
    char-poly A = (∏(na, a)←n-as. [:− a, 1:] ^ na) ⟹
    (⋀k. A ^m k = P * jordan-matrix n-as ^m k * Q) ⟹ thesis
  from jnf[unfolded jordan-nf-def] obtain P Q where
    simw: similar-mat-wit A (jordan-matrix n-as) P Q
     and sim: similar-mat A (jordan-matrix n-as) unfolding similar-mat-def by
blast
  show thesis
  proof (rule obt)
    show ⋀ k. A ^m k = P * jordan-matrix n-as ^m k * Q
      by (rule similar-mat-wit-pow-id[OF simw])
    show char-poly A = (∏(na, a)←n-as. [:− a, 1:] ^ na)
      unfolding char-poly-similar[OF sim] jordan-matrix-char-poly ..
  qed (insert simw[unfolded similar-mat-wit-def Let-def dim], auto)
qed

lemma choose-poly-bound: assumes i ≤ d
  shows r choose i ≤ max 1 (r^d)
proof (cases i ≤ r)
  case False
  hence r choose i = 0 by simp
  thus ?thesis by arith
next
```

case *True*
show *?thesis*
proof (*cases r*)
case (*Suc rr*)
from *binomial-le-pow*[*OF True*] **have** r *choose* $i \le r$ ^ i **by** *simp*
also have ... $\le r$^d **using** *power-increasing*[*OF* ‹$i \le d$›, *of r*] *Suc* **by** *auto*
finally show *?thesis* **by** *simp*
qed (*insert True*, *simp*)
qed

context
fixes b :: $'a$ :: *archimedean-field*
assumes b: $0 < b$ $b < 1$
begin

lemma *poly-exp-constant-bound*: $\exists\ p.\ \forall\ x.\ c * b$ ^ $x *$ *of-nat* x ^ *deg* $\le p$
proof (*cases* $c \le 0$)
case *True*
show *?thesis*
by (*rule exI*[*of - 0*], *intro allI*,
rule mult-nonpos-nonneg[*OF mult-nonpos-nonneg*[*OF True*]], *insert b*, *auto*)
next
case *False*
hence c: $c \ge 0$ **by** *simp*
from *poly-exp-bound*[*OF b*, *of deg*] **obtain** p **where** $\bigwedge x.\ b$ ^ $x *$ *of-nat* x ^ *deg* $\le p$ **by** *auto*
from *mult-left-mono*[*OF this c*]
show *?thesis* **by** (*intro exI*[*of - c * p*], *auto simp*: *ac-simps*)
qed

lemma *poly-exp-max-constant-bound*: $\exists\ p.\ \forall\ x.\ c * b$ ^ $x *$ *max 1* (*of-nat* x ^ *deg*) $\le p$
proof –
from *poly-exp-constant-bound*[*of c deg*] **obtain** p **where**
p: $\bigwedge x.\ c * b$ ^ $x *$ *of-nat* x ^ *deg* $\le p$ **by** *auto*
show *?thesis*
proof (*rule exI*[*of - max p c*], *intro allI*)
fix x
let *?exp* = *of-nat* x ^ *deg* :: $'a$
show $c * b$ ^ $x *$ *max 1 ?exp* $\le$ *max p c*
proof (*cases* $x = 0$)
case *False*
hence *?exp* $\ne$ *of-nat 0* **by** *simp*
hence *?exp* ≥ 1 **by** (*metis less-one not-less of-nat-1 of-nat-less-iff of-nat-power*)
hence *max 1 ?exp* = *?exp* **by** *simp*
thus *?thesis* **using** *p*[*of x*] **by** *simp*
qed (*cases deg*, *auto*)
qed
qed

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* $\widehat{\ }_m$ *k*) $$ (*i*,*j*))
    $\leq$ *norm a* $\widehat{\ }$(*k* + *i* − *j*) ∗ *max 1* (*of-nat k* $\widehat{\ }$(*n* − *1*))
    (**is** *?lhs* ≤ *?rhs*)
proof −
  have *id*: (*jordan-block n a* $\widehat{\ }_m$ *k*) $$ (*i*,*j*) = (*if i* ≤ *j then of-nat* (*k choose* (*j* − *i*)) ∗ *a* $\widehat{\ }$(*k* + *i* − *j*) *else 0*)
    unfolding *jordan-block-pow* **using** *i j* **by** *auto*
  from *i j* **have** *diff*: *j* − *i* ≤ *n* − *1* **by** *auto*
  show *?thesis*
  proof (*cases i* ≤ *j*)
    case *False*
    thus *?thesis* **unfolding** *id* **by** *simp*
  next
    case *True*
    hence *?lhs* = *norm* (*of-nat* (*k choose* (*j* − *i*)) ∗ *a* $\widehat{\ }$(*k* + *i* − *j*)) **unfolding** *id* **by** *simp*
    also have ... ≤ *norm* (*of-nat* (*k choose* (*j* − *i*)) :: *'a*) ∗ *norm* (*a* $\widehat{\ }$(*k* + *i* − *j*))
      by (*rule norm-mult-ineq*)
    also have ... ≤ (*max 1* (*of-nat k* $\widehat{\ }$(*n* − *1*))) ∗ *norm a* $\widehat{\ }$(*k* + *i* − *j*)
    proof (*rule mult-mono*[*OF - norm-power-ineq - norm-ge-zero*])
      have *k choose* (*j* − *i*) ≤ *max 1* (*k* $\widehat{\ }$(*n* − *1*))
        by (*rule choose-poly-bound*[*OF diff*])
      hence *norm* (*of-nat* (*k choose* (*j* − *i*)) :: *'a*) ≤ *of-nat* (*max 1* (*k* $\widehat{\ }$(*n* − *1*)))
        unfolding *norm-of-nat of-nat-le-iff* **.**
      also have ... = *max 1* (*of-nat k* $\widehat{\ }$(*n* − *1*)) **by** (*metis max-def of-nat-1 of-nat-le-iff of-nat-power*)
      finally show *norm* (*of-nat* (*k choose* (*j* − *i*)) :: *'a*) ≤ *max 1* (*real-of-nat k* $\widehat{\ }$(*n* − *1*)) **.**
    qed *simp*
    also have ... = *?rhs* **by** *simp*
    finally show *?thesis* **.**
  qed
qed

lemma *jordan-block-poly-bound*:
  assumes *i*: $i < n$ **and** *j*: $j < n$ **and** *a*: *norm a* = *1*
  shows *norm* ((*jordan-block n a* $\widehat{\ }_m$ *k*) $$ (*i*,*j*)) ≤ *max 1* (*of-nat k* $\widehat{\ }$(*n* − *1*))
    (**is** *?lhs* ≤ *?rhs*)
proof −
  from *jordan-block-bound*[*OF i j*, *of k*, *unfolded a*]
  show *?thesis* **by** *simp*

qed

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*) $$ (*i*,*j*)) ≤ *p*
proof (*cases a* = *0*)
case *True*
show *?thesis*
proof (*rule exI*[*of* - *1*], *intro allI impI*)
fix *i j k*
assume ∗: *i* < *n j* < *n*
{
assume *ij*: *i* ≤ *j*
have *norm* ((*of-nat* (*k choose* (*j* − *i*)) :: ′*a*) ∗ *0* ^(*k* + *i* − *j*)) ≤ *1* (**is** *norm ?lhs* ≤ *1*)
proof (*cases k* + *i* > *j*)
case *True*
hence *?lhs* = *0* **by** *simp*
also have *norm* (...) ≤ *1* **by** *simp*
finally show *?thesis* **.**
next
case *False*
hence *id*: *?lhs* = (*of-nat* (*k choose* (*j* − *i*)) :: ′*a*) **and** *j*: *j* − *i* ≥ *k* **by** *auto*
from *j* **have** *k choose* (*j* − *i*) = *0* ∨ *k choose* (*j* − *i*) = *1* **by** (*simp add*: *nat-less-le*)
thus *norm ?lhs* ≤ *1*
proof
assume *k*: *k choose* (*j* − *i*) = *0*
show *?thesis* **unfolding** *id k* **by** *simp*
next
assume *k*: *k choose* (*j* − *i*) = *1*
show *?thesis* **unfolding** *id* **unfolding** *k* **by** *simp*
qed
qed
}
thus *norm* ((*jordan-block n a* ^$_m$ *k*) $$ (*i*,*j*)) ≤ *1* **unfolding** *True*
unfolding *jordan-block-pow* **using** ∗ **by** *auto*
qed
next
case *False*
hence *na*: *norm a* > *0* **by** *auto*
define *c* **where** *c* = *inverse* (*norm a* ^ *n*)
define *deg* **where** *deg* = *n* − *1*
have *c*: *c* > *0* **unfolding** *c-def* **using** *na* **by** *auto*
define *b* **where** *b* = *norm a*
from *a na* **have** *0* < *b b* < *1* **unfolding** *b-def* **by** *auto*
from *poly-exp-max-constant-bound*[*OF this*, *of c deg*]
obtain *p* **where** ⋀ *k*. *c* ∗ *b* ^ *k* ∗ *max 1* (*of-nat k* ^ *deg*) ≤ *p* **by** *auto*

show *?thesis*
proof (*intro exI*[*of - p*], *intro allI impI*)
fix *i j k*
assume *ij*: $i < n$ $j < n$
from *jordan-block-bound*[*OF this*]
have *norm* ((*jordan-block n a* $\hat{}_m$ *k*) \$\$ (*i*, *j*))
$\leq$ *norm a* ^ ($k + i - j$) $*$ *max 1* (*real-of-nat k* ^ ($n - 1$)) **.**
also have ... $\leq c *$ *norm a* ^ *k* $*$ *max 1* (*real-of-nat k* ^ ($n - 1$))
proof (*rule mult-right-mono*)
from *ij* **have** $i - j \leq n$ **by** *auto*
show *norm a* ^ ($k + i - j$) $\leq c *$ *norm a* ^ *k*
proof (*rule mult-left-le-imp-le*)
show $0 <$ *norm a* ^ *n* **using** *na* **by** *auto*
let *?lhs* = *norm a* ^ *n* $*$ *norm a* ^ ($k + i - j$)
let *?rhs* = *norm a* ^ *n* $*$ ($c *$ *norm a* ^ *k*)
from *ij* **have** *ge*: $n + (k + i - j) \geq k$ **by** *arith*
have *?lhs* = *norm a* ^ ($n + (k + i - j)$) **by** (*simp add*: *power-add*)
also have ... $\leq$ *norm a* ^ *k* **using** *ge a na* **using** *less-imp-le power-decreasing*
by *blast*
also have ... = *?rhs* **unfolding** *c-def* **using** *na* **by** *simp*
finally show *?lhs* $\leq$ *?rhs* **.**
qed
qed *simp*
also have ... = $c * b$ ^ $k *$ *max 1* (*real-of-nat k* ^ *deg*) **unfolding** *b-def deg-def*
..
also have ... $\leq p$ **by** *fact*
finally show *norm* ((*jordan-block n a* $\hat{}_m$ *k*) \$\$ (*i*, *j*)) $\leq p$ **.**
qed
qed

definition *norm-bound* :: *'a mat* $\Rightarrow$ *real* $\Rightarrow$ *bool* **where**
norm-bound A b $\equiv \forall$ *i j*. $i <$ *dim-row A* $\longrightarrow j <$ *dim-col A* $\longrightarrow$ *norm* (*A* \$\$
(*i*,*j*)) $\leq b$

lemma *norm-boundI*[*intro*]:
assumes $\bigwedge$ *i j*. $i <$ *dim-row A* $\Longrightarrow j <$ *dim-col A* $\Longrightarrow$ *norm* (*A* \$\$ (*i*,*j*)) $\leq b$
shows *norm-bound A b*
unfolding *norm-bound-def* **using** *assms* **by** *blast*

lemma *jordan-block-constant-bound2*:
$\exists p$. *norm* (*a* :: *'a* :: *real-normed-field*) $< 1 \longrightarrow$
($\forall$ *i j k*. $i < n \longrightarrow j < n \longrightarrow$ *norm* ((*jordan-block n a* $\hat{}_m$ *k*) \$\$ (*i*, *j*)) $\leq p$)
using *jordan-block-constant-bound* **by** *auto*

lemma *jordan-matrix-poly-bound2*:
fixes *n-as* :: (*nat* $\times$ *'a*) *list*
assumes *n-as*: $\bigwedge$ *n a*. (*n*,*a*) $\in$ *set n-as* $\Longrightarrow n > 0 \Longrightarrow$ *norm a* ≤ 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*. $\forall$ *e* $\in$ *elements-mat* (*jordan-matrix n-as* $\hat{}_m$ *k*).

```
    norm e ≤ c1 + of-nat k ^ (N − 1)
proof −
  from jordan-matrix-carrier[of n-as] obtain d where
    jm: jordan-matrix n-as ∈ carrier-mat d d by blast
  define f where f = (λn (a::'a) i j k. norm ((jordan-block n a ^m k) $$ (i,j)))
  let ?g = λk c1. c1 + of-nat k ^ (N−1)
  let ?P = λn (a::'a) i j k c1. f n a i j k ≤ ?g k c1
  define Q where Q = (λn (a::'a) k c1. ∀ i j. i<n ⟶ j<n ⟶ ?P n a i j k c1)
  have ⋀ c c' k n a i j. c ≤ c' ⟹ ?P n a i j k c ⟹ ?P n a i j k c' by auto
  hence Q-mono: ⋀n a c c'. c ≤ c' ⟹ ∀ k. Q n a k c ⟹ ∀ k. Q n a k c'
    unfolding Q-def by arith
  { fix n a assume na: (n,a) ∈ set n-as
    obtain c where c: norm a < 1 ⟶ (∀ i j k. i < n ⟶ j < n ⟶ f n a i j k
≤ c)
      apply (rule exE[OF jordan-block-constant-bound2])
      unfolding f-def using Jordan-Normal-Form.jordan-block-constant-bound2
      by metis
    define c1 where c1 = max 1 c
    then have c1 ≥ 1 c1 ≥ c by auto
    have ∃ c1. ∀ k i j. i < n ⟶ j < n ⟶ ?P n a i j k c1
    proof rule+
      fix i j k assume i < n j < n
      then have 0<n by auto
      let ?jbs = map (λ(n,a). jordan-block n a) n-as
      have sq-jbs: Ball (set ?jbs) square-mat by auto
      have jordan-matrix n-as ^m k = diag-block-mat (map (λA. A ^m k) ?jbs)
        unfolding jordan-matrix-def using diag-block-pow-mat[OF sq-jbs] by auto
      show ?P n a i j k c1
      proof (cases norm a = 1)
        case True {
          have nN:n−1 ≤ N−1 using N[OF na] True by auto
          have f n a i j k ≤ max 1 (of-nat k ^ (n−1))
            using Jordan-Normal-Form.jordan-block-poly-bound True ‹i<n› ‹j<n›
            unfolding f-def by auto
          also have ... ≤ max 1 (of-nat k ^ (N−1))
            proof (cases k=0)
              case False then show ?thesis
                by (subst max.mono[OF - power-increasing[OF nN]], auto)
            qed (simp add: power-eq-if)
          also have ... ≤ max c1 (of-nat k ^ (N−1)) using ‹c1≥1› by auto
          also have ... ≤ c1 + (of-nat k ^ (N−1)) using ‹c1≥1› by auto
          finally show ?thesis by simp
        } next
        case False {
          then have na1: norm a < 1 using n-as[OF na] ‹0<n› by auto
          hence f n a i j k ≤ c using c ‹i<n› ‹j<n› by auto
          also have ... ≤ c1 using ‹c≤c1›.
          also have ... ≤ c1 + of-nat k ^ (N−1) by auto
          finally show ?thesis by auto
```

```
      }
    qed
  qed
}
hence ∀ na. ∃ c1. na ∈ set n-as ⟶ (∀ k. Q (fst na) (snd na) k c1)
  unfolding Q-def by auto
from choice[OF this] obtain c'
  where c': ⋀ na k. na ∈ set n-as ⟹ Q (fst na) (snd na) k (c' na) by blast
define c where c = max 0 (Max (set (map c' n-as)))
{ fix n a assume na: (n,a) ∈ set n-as
  then have Q: ∀ k. Q n a k (c' (n,a)) using c'[OF na] by auto
  from na have c' (n,a) ∈ set (map c' n-as) by auto
  from Max-ge[OF - this] have c' (n,a) ≤ c unfolding c-def by auto
  from Q-mono[OF this Q] have ⋀ k. Q n a k c by blast
}
hence Q: ⋀k n a. (n,a) ∈ set n-as ⟹ Q n a k c by auto
have c0: c ≥ 0 unfolding c-def by simp
{ fix k n a e
  assume na:(n,a) ∈ set n-as
  let ?jbk = jordan-block n a ^m k
  assume e ∈ elements-mat ?jbk
  from elements-matD[OF this] obtain i j
    where i < n j < n and [simp]: e = ?jbk $$ (i,j)
    by (simp only:pow-mat-dim-square[OF jordan-block-carrier],auto)
  hence norm e ≤ ?g k c using Q[OF na] unfolding Q-def f-def by simp
}
hence norm-jordan:
  ⋀k. ∀ (n,a) ∈ set n-as. ∀ e ∈ elements-mat (jordan-block n a ^m k).
   norm e ≤ ?g k c by auto
{ fix k
  let ?jmk = jordan-matrix n-as ^m k
  have dim-row ?jmk = d dim-col ?jmk = d
    using jm by (simp only:pow-mat-dim-square[OF jm])+
  let ?As = (map (λ(n,a). jordan-block n a ^m k) n-as)
  have ⋀e. e ∈ elements-mat ?jmk ⟹ norm e ≤ ?g k c
  proof -
    fix e assume e:e ∈ elements-mat ?jmk
    obtain i j where ij: i < d j < d and e = ?jmk $$ (i,j)
     using elements-matD[OF e] by (simp only:pow-mat-dim-square[OF jm],auto)
    have ?jmk = diag-block-mat ?As
      using jordan-matrix-pow[of n-as k] by auto
    hence elements-mat ?jmk ⊆ {0} ∪ ⋃ (set (map elements-mat ?As))
      using elements-diag-block-mat[of ?As] by auto
    hence e-mem: e ∈ {0} ∪ ⋃ (set (map elements-mat ?As))
      using e by blast
    show norm e ≤ ?g k c
    proof (cases e = 0)
      case False
        then have e ∈ ⋃ (set (map elements-mat ?As)) using e-mem by auto
```

```
        then obtain n a
          where e ∈ elements-mat (jordan-block n a ^m k)
          and na: (n,a) ∈ set n-as by force
        thus ?thesis using norm-jordan na by force
      qed (insert c0, auto)
    qed
  }
  thus ?thesis by auto
qed

lemma norm-bound-bridge:
  ∀ e ∈ elements-mat A. norm e ≤ b ⟹ norm-bound A b
  unfolding norm-bound-def by force

lemma norm-bound-mult: assumes A1: A1 ∈ carrier-mat nr n
  and A2: A2 ∈ 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
  let ?A = A1 * A2
  let ?n = of-nat n
  fix i j
  assume i: i < dim-row ?A and j: j < dim-col ?A
  define v1 where v1 = (λ k. row A1 i $ k)
  define v2 where v2 = (λ k. col A2 j $ k)
  from assms(1−2) have dim: dim-row A1 = nr dim-col A2 = nc dim-col A1 =
n dim-row A2 = n by auto
  {
    fix k
    assume k: k < n
    have n: norm (v1 k) ≤ b1 norm (v2 k) ≤ b2
      using i j k dim v1-def v2-def
      b1[unfolded norm-bound-def, rule-format, of i k]
      b2[unfolded norm-bound-def, rule-format, of k j] by auto
   have norm (v1 k * v2 k) ≤ norm (v1 k) * norm (v2 k) by (rule norm-mult-ineq)
    also have ... ≤ b1 * b2 by (rule mult-mono'[OF n], auto)
    finally have norm (v1 k * v2 k) ≤ b1 * b2 .
  } note bound = this
  have ?A $$ (i,j) = row A1 i · col A2 j using dim i j by simp
  also have ... = (∑ k = 0 ..< n. v1 k * v2 k) unfolding scalar-prod-def
    using dim i j v1-def v2-def by simp
  also have norm (...) ≤ (∑ k = 0 ..< n. b1 * b2)
    by (rule sum-norm-le, insert bound, simp)
  also have ... = b1 * b2 * ?n by simp
  finally show norm (?A $$ (i,j)) ≤ b1 * b2 * ?n .
qed

lemma norm-bound-max: norm-bound A (Max {norm (A $$ (i,j)) | i j. i <
```

dim-row A ∧ *j* < *dim-col A*})
(**is** *norm-bound A* (*Max ?norms*))
proof
fix *i j*
have *fin*: *finite ?norms* **by** (*simp add*: *finite-image-set2*)
assume *i* < *dim-row A* **and** *j* < *dim-col A*
hence *norm* (*A* $$ (*i*,*j*)) ∈ *?norms* **by** *auto*
from *Max-ge*[*OF fin this*] **show** *norm* (*A* $$ (*i*,*j*)) ≤ *Max ?norms* **.**
qed

lemma *jordan-matrix-poly-bound*: **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*. *norm-bound* (*jordan-matrix n-as* $\widehat{\ }_m$ *k*) (*c1* + *of-nat k* ^ (*N* − *1*))
using *jordan-matrix-poly-bound2 norm-bound-bridge N n-as*
by *metis*

lemma *jordan-nf-matrix-poly-bound*: **fixes** *n-as* :: (*nat* × *'a*)*list*
assumes *A*: *A* ∈ *carrier-mat n n*
and *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*
and *jnf*: *jordan-nf A n-as*
shows ∃ *c1 c2*. ∀ *k*. *norm-bound* (*A* $\widehat{\ }_m$ *k*) (*c1* + *c2* ∗ *of-nat k* ^ (*N* − *1*))
proof −
let *?cp2* = ∏(*n*, *a*)←*n-as*. [:− *a*, *1*:] ^ *n*
let *?J* = *jordan-matrix n-as*
from *jnf*[*unfolded jordan-nf-def*]
have *sim*: *similar-mat A ?J* **by** *auto*
then obtain *P Q* **where** *sim-wit*: *similar-mat-wit A ?J P Q* **unfolding** *similar-mat-def* **by** *auto*
from *similar-mat-wit-pow-id*[*OF this*] **have** *pow*: ⋀ *k*. *A* $\widehat{\ }_m$ *k* = *P* ∗ *?J* $\widehat{\ }_m$ *k* ∗ *Q* **.**
from *sim-wit*[*unfolded similar-mat-wit-def Let-def*] *A*
have *J*: *?J* ∈ *carrier-mat n n* **and** *P*: *P* ∈ *carrier-mat n n* **and** *Q*: *Q* ∈ *carrier-mat n n*
unfolding *carrier-mat-def* **by** *force+*
have ∃ *c1*. ∀ *k*. *norm-bound* (*?J* $\widehat{\ }_m$ *k*) (*c1* + *of-nat k* ^ (*N* − *1*))
by (*rule jordan-matrix-poly-bound*[*OF n-as N*])
then obtain *c1* **where**
bound-pow: ⋀ *k*. *norm-bound* ((*?J* $\widehat{\ }_m$ *k*)) (*c1* + *of-nat k* ^ (*N* − *1*)) **by** *blast*
obtain *bP* **where** *bP*: *norm-bound P bP* **using** *norm-bound-max*[*of P*] **by** *auto*
obtain *bQ* **where** *bQ*: *norm-bound Q bQ* **using** *norm-bound-max*[*of Q*] **by** *auto*
let *?n* = *of-nat n* :: *real*
let *?c2* = *bP* ∗ *?n* ∗ *bQ* ∗ *?n*
let *?c1* = *?c2* ∗ *c1*
{
fix *k*
have *Jk*: *?J* $\widehat{\ }_m$ *k* ∈ *carrier-mat n n* **using** *J* **by** *simp*

from *norm-bound-mult*[*OF mult-carrier-mat*[*OF P Jk*] *Q*
norm-bound-mult[*OF P Jk bP bound-pow*] *bQ*, *folded pow*]
have *norm-bound* (A $\widehat{\ }_m$ k) (*?c1* + *?c2* ∗ *of-nat k* ^ (N − *1*)) (**is** *norm-bound* - *?exp*)
by (*simp add*: *field-simps*)
} **note** *main* = *this*
show *?thesis*
by (*intro exI allI*, *rule main*)
qed
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*))
unfolding *char-matrix-def jordan-block-def* **by** *auto*

lemma *diag-jordan-block-pow*: *diag-mat* (*jordan-block n* (*a* :: ′*a*) $\widehat{\ }_m$ *k*) = *replicate n* (*a* ^ *k*)
unfolding *diag-mat-def jordan-block-pow*
by (*intro nth-equalityI*, *auto*)

lemma *jordan-block-zero-pow*: (*jordan-block n* (*0* :: ′*a*)) $\widehat{\ }_m$ *k* =
(*mat n n* (λ (*i*,*j*). *if j* ≥ *i* ∧ *j* − *i* = *k then 1 else 0*))
proof −
{
fix *i j*
assume ∗: *j* − *i* ≠ *k*
have *of-nat* (*k choose* (*j* − *i*)) ∗ *0* ^ (*k* + *i* − *j*) = (*0* :: ′*a*)
proof (*cases k* + *i* − *j* > *0*)
case *True* **thus** *?thesis* **by** (*cases k* + *i* − *j*, *auto*)
next
case *False*
with ∗ **have** *j* − *i* > *k* **by** *auto*
thus *?thesis* **by** (*simp add*: *binomial-eq-0*)
qed
}
thus *?thesis* **unfolding** *jordan-block-pow* **by** (*intro eq-matI*, *auto*)
qed
end

lemma *jordan-matrix-concat-diag-block-mat*: *jordan-matrix* (*concat jbs*) = *diag-block-mat* (*map jordan-matrix jbs*)
unfolding *jordan-matrix-def*[*abs-def*]
by (*induct jbs*, *auto simp*: *diag-block-mat-append Let-def*)

lemma *jordan-nf-diag-block-mat*: **assumes** *Ms*: ⋀ *A jbs*. (*A*,*jbs*) ∈ *set Ms* ⟹ *jordan-nf A jbs*

```
  shows jordan-nf (diag-block-mat (map fst Ms)) (concat (map snd Ms))
proof -
  let ?Ms = map (λ (A, jbs). (A, jordan-matrix jbs)) Ms
  have id: map fst ?Ms = map fst Ms by auto
  have id2: map snd ?Ms = map jordan-matrix (map snd Ms) by auto
  {
    fix A B
    assume (A,B) ∈ set ?Ms
    then obtain jbs where mem: (A,jbs) ∈ set Ms and B: B = jordan-matrix jbs
by auto
    from Ms[OF mem] have similar-mat A B unfolding B jordan-nf-def by auto
  }
  from similar-diag-mat-block-mat[of ?Ms, OF this, unfolded id id2] Ms
  show ?thesis
    unfolding jordan-nf-def jordan-matrix-concat-diag-block-mat by force
qed


lemma jordan-nf-char-poly: assumes jordan-nf A n-as
  shows char-poly A = (∏ (n,a) ← n-as. [:− a, 1:] ^ n)
  unfolding jordan-matrix-char-poly[symmetric]
  by (rule char-poly-similar, insert assms[unfolded jordan-nf-def], auto)

lemma jordan-nf-block-size-order-bound: assumes jnf: jordan-nf A n-as
  and mem: (n,a) ∈ set n-as
  shows n ≤ order a (char-poly A)
proof -
  from jnf[unfolded jordan-nf-def]
  have similar-mat A (jordan-matrix n-as) by auto
  from similar-matD[OF this] obtain m where A ∈ carrier-mat m m by auto
  from degree-monic-char-poly[OF this] have A: char-poly A ≠ 0 by auto
  from mem obtain as bs where nas: n-as = as @ (n,a) # bs
    by (meson split-list)
  from jordan-nf-char-poly[OF jnf]
  have cA: char-poly A = (∏(n, a)←n-as. [:− a, 1:] ^ n) .
  also have ... = [: −a, 1:] ^ n * (∏(n, a)← as @ bs. [:− a, 1:] ^ n) unfolding
nas by auto
  also have [: −a,1 :] ^ n dvd ... unfolding dvd-def by blast
  finally have [: −a,1 :] ^ n dvd char-poly A by auto
  from order-max[OF this A] show ?thesis .
qed

lemma similar-mat-jordan-block-smult: fixes A :: 'a :: field mat
  assumes similar-mat A (jordan-block n a)
   and k: k ≠ 0
  shows similar-mat (k ·m A) (jordan-block n (k * a))
proof -
  let ?J = jordan-block n a
  let ?Jk = jordan-block n (k * a)
```

let *?kJ* = *k* $\cdot_m$ *jordan-block n a*
from *k* **have** *inv*: *k* ^ *i* $\neq$ *0* **for** *i* **by** *auto*
let *?A* = *mat-diag n* (λ *i. k*^*i*)
let *?B* = *mat-diag n* (λ *i. inverse* (*k*^*i*))
have *similar-mat-wit ?Jk ?kJ ?A ?B*
proof (*rule similar-mat-witI*)
show *jordan-block n* (*k* $*$ *a*) = *?A* $*$ *?kJ* $*$ *?B*
by (*subst mat-diag-mult-left*[*of* - - *n*], *force*, *subst mat-diag-mult-right*[*of* - *n*], *insert k inv*, *auto simp*: *jordan-block-def field-simps intro*!: *eq-matI*)
qed (*auto simp*: *inv field-simps k*)
hence *kJ*: *similar-mat ?Jk ?kJ*
unfolding *similar-mat-def* **by** *auto*
have *similar-mat A ?J* **by** *fact*
hence *similar-mat* (*k* $\cdot_m$ *A*) (*k* $\cdot_m$ *?J*) **by** (*rule similar-mat-smult*)
with *kJ* **show** *?thesis*
using *similar-mat-sym similar-mat-trans* **by** *blast*
qed

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*)
unfolding *jordan-matrix-def* **by** (*simp*, *simp add*: *jordan-matrix-def*[*symmetric*])

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 (*induct n-as*)
case *Nil*
show *?case* **by** (*auto simp*: *jordan-matrix-def intro*!: *similar-mat-refl*)
next
case (*Cons na n-as*)
obtain *n a* **where** *na*: *na* = (*n*,*a*) **by** *force*
let *?l* = *map* (λ (*n*,*a*). (*n*, *k* $*$ *a*))
let *?n* = *sum-list* (*map fst n-as*)
have *k* $\cdot_m$ *jordan-matrix* (*Cons na n-as*) = *k* $\cdot_m$ *four-block-mat*
(*jordan-block n a*) (*0*$_m$ *n ?n*)
(*0*$_m$ *?n n*) (*jordan-matrix n-as*) (**is** *?M* = - $\cdot_m$ *four-block-mat ?A ?B ?C ?D*)
by (*simp add*: *na jordan-matrix-Cons*)
also have ... = *four-block-mat* (*k* $\cdot_m$ *?A*) *?B ?C* (*k* $\cdot_m$ *?D*)
by (*subst smult-four-block-mat*, *auto*)
finally have *jm*: *?M* = *four-block-mat* (*k* $\cdot_m$ *?A*) *?B ?C* (*k* $\cdot_m$ *?D*) **.**
have [*simp*]: *fst* (*case x of* (*n* :: *nat*, *a*) $\Rightarrow$ (*n*, *k* $*$ *a*)) = *fst x* **for** *x* **by** (*cases x*, *auto*)
have *jmk*: *jordan-matrix* (*?l* (*Cons na n-as*)) = *four-block-mat*
(*jordan-block n* (*k* $*$ *a*)) *?B*
?C (*jordan-matrix* (*?l n-as*)) (**is** *?kM* = *four-block-mat ?kA* - - *?kD*)
by (*simp add*: *na jordan-matrix-Cons o-def*)

show *?case* **unfolding** *jmk jm*
by (*rule similar-mat-four-block-0-0*[*OF similar-mat-jordan-block-smult*[*OF - k*] *Cons*],
auto intro!: *similar-mat-refl*)
qed

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 −
let *?l* = *map* (λ (*n*,*a*). (*n*, $k * a$))
from *jn*[*unfolded jordan-nf-def*] **have** *sim*: *similar-mat A* (*jordan-matrix n-as*) **by** *auto*
from *similar-mat-smult*[*OF this*, *of k*] *similar-mat-jordan-matrix-smult*[*OF k*, *of n-as*]
have *similar-mat* ($k \cdot_m A$) (*jordan-matrix* (*map* (λ(*n*, *a*). (*n*, $k * a$)) *n-as*))
using *similar-mat-trans* **by** *blast*
with *jn* **show** *?thesis* **unfolding** *jordan-nf-def* **by** *force*
qed

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 −
let *?p* = λ *n-as*. ($\prod$(*n*, *a*)←*n-as*. [:− *a*, *1*:] ^ *n*)
let *?s* = λ *n-as*. *sum-list* (*map fst* (*filter* (λ *na*. *snd na* = *a*) *n-as*))
from *jordan-nf-char-poly*[*OF assms*]
have *order a* (*char-poly A*) = *order a* (*?p n-as*) **by** *simp*
also have ... = *?s n-as*
proof (*induct n-as*)
case (*Cons nb n-as*)
obtain *n b* **where** *nb*: *nb* = (*n*,*b*) **by** *force*
have *order a* (*?p* (*nb* # *n-as*)) = *order a* ([: −*b*, *1*:] ^ *n* ∗ *?p n-as*) **unfolding** *nb* **by** *simp*
also have ... = *order a* ([: −*b*, *1*:] ^ *n*) + *order a* (*?p n-as*)
by (*rule order-mult*, *auto simp*: *prod-list-zero-iff*)
also have ... = (*if a* = *b then n else 0*) + *?s n-as* **unfolding** *Cons order-linear-power* **by** *simp*
also have ... = *?s* (*nb* # *n-as*) **unfolding** *nb* **by** *auto*
finally show *?case* **.**
qed *simp*
finally show *?thesis* **.**
qed

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: ⋀ 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 ^m k) (c1 + c2 * of-nat k ^ (N − 1))
proof −
  from jnf-exists obtain n-as
    where jnf: jordan-nf A n-as by auto
  let ?cp1 = (∏ a ← as. [:− a, 1:])
  let ?cp2 = ∏ (n, a)←n-as. [:− a, 1:] ^ n
  let ?J = jordan-matrix n-as
  from jnf[unfolded jordan-nf-def]
  have sim: similar-mat A ?J by auto
  from char-poly-similar[OF sim, unfolded linear-factors jordan-matrix-char-poly]
  have cp: ?cp1 = ?cp2 .
  show ?thesis
  proof (rule jordan-nf-matrix-poly-bound[OF A - - jnf])
    fix n a
    assume na: (n,a) ∈ set n-as
    then obtain na1 na2 where n-as: n-as = na1 @ (n,a) # na2
      unfolding in-set-conv-decomp by auto
    then obtain p where ?cp2 = [: −a, 1 :]^n * p unfolding n-as by auto
    from cp[unfolded this] have dvd: [: −a, 1 :] ^ n dvd ?cp1 by auto
    let ?as = filter ((=) a) as
    let ?pn = λ as. ∏ a←as. [:− a, 1:]
    let ?p = λ as. ∏ a←as. [: a, 1:]
    have ?pn as = ?p (map uminus as) by (induct as, auto)
    from poly-linear-exp-linear-factors[OF dvd[unfolded this]]
    have n ≤ length (filter ((=) (− a)) (map uminus as)) .
    also have . . . = length (filter ((=) a) as)
      by (induct as, auto)
    finally have filt: n ≤ length (filter ((=) a) as) .
    {
      assume 0 < n
      with filt obtain b bs where ?as = b # bs by (cases ?as, auto)
      from arg-cong[OF this, of set]
      have a ∈ set as by auto
      from le-1[rule-format, OF this]
      show norm a ≤ 1 .
      note ‹a ∈ set as›
    } note mem = this
    {
      assume norm a = 1
      from le-N[OF mem this] filt show n ≤ N by (cases n, auto)
    }
  qed
qed

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 :: $'a \Rightarrow 'a \Rightarrow 'a$ **and** A::$'a$ *set*
assumes *comp-fun-commute-restrict*: $\forall y \in A.\ \forall x \in A.\ \forall z \in A.\ f\ y\ (f\ x\ z) = f\ x\ (f\ y\ z)$
and f: $f : A \to A \to A$
begin

lemma *comp-fun-commute-on-UNIV*:
assumes $A = (UNIV :: 'a\ set)$
shows *comp-fun-commute f*
unfolding *comp-fun-commute-def*
using *assms comp-fun-commute-restrict f* **by** *auto*

lemma *fun-left-comm*:
assumes $y \in A$ **and** $x \in A$ **and** $z \in A$ **shows** $f\ y\ (f\ x\ z) = f\ x\ (f\ y\ z)$
using *comp-fun-commute-restrict assms* **by** *auto*

lemma *commute-left-comp*:
assumes $y \in A$ **and** $x \in A$ **and** $z \in A$ **and** $g \in A \to A$
shows $f\ y\ (f\ x\ (g\ z)) = f\ x\ (f\ y\ (g\ z))$
using *assms* **by** (*auto simp add*: *Pi-def o-assoc comp-fun-commute-restrict*)

lemma *fold-graph-finite*:
assumes *fold-graph f z B y*
shows *finite B*
using *assms* **by** *induct simp-all*

lemma *fold-graph-closed*:
assumes *fold-graph f z B y* **and** $B \subseteq A$ **and** $z \in A$
shows $y \in A$
using *assms*
proof (*induct set*: *fold-graph*)
case *emptyI*
then show *?case* **by** *auto*

next
  case (*insertI x B y*)
  then show *?case* **using** *insertI f* **by** *auto*
qed

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 (*induct set*: *fold-graph*)
  case *emptyI*
  then show *?case* **by** *auto*
next
  case (*insertI x B y*)
  show *?case*
  proof (*cases* $x = a$)
    case *True*
    show *?thesis*
    proof (*rule exI*[*of - y*])
      have *B*: (*insert x B* $- \{a\}$) $= B$ **using** *True insertI* **by** *auto*
      have $f\ x\ y = f\ a\ y$ **by** (*simp add*: *True*)
      moreover have *fold-graph f z* (*insert x B* $- \{a\}$) *y* **by** (*simp add*: *B insertI*)
      moreover have $y \in A$ **using** *insertI fold-graph-closed*[*of z B*] **by** *auto*
      ultimately show $f\ x\ y = f\ a\ y \wedge$ *fold-graph f z* (*insert x B* $- \{a\}$) $y \wedge y \in$
A **by** *simp*
    qed
  next
    case *False*
    then obtain y' **where** *y*: $y = f\ a\ y'$ **and** y': *fold-graph f z* $(B - \{a\})\ y'$ **and**
y'*-in-A*: $y' \in A$
      using *insertI f* **by** *auto*
    have $f\ x\ y = f\ a\ (f\ x\ y')$
      unfolding *y*
    proof (*rule fun-left-comm*)
      show $x \in A$ **using** *insertI* **by** *auto*
      show $a \in A$ **using** *insertI* **by** *auto*
      show $y' \in A$ **using** y'*-in-A* **by** *auto*
    qed
    moreover have *fold-graph f z* (*insert x B* $- \{a\}$) $(f\ x\ y')$
      using y' **and** ‹$x \neq a$› **and** ‹$x \notin B$›
      by (*simp add*: *insert-Diff-if fold-graph.insertI*)
    moreover have $(f\ x\ y') \in A$ **using** *insertI f* y'*-in-A* **by** *auto*
    ultimately show *?thesis* **using** y'*-in-A*
      by *auto*
  qed
qed

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*
using *assms* **by** (*auto dest*: *fold-graph-insertE-aux* [*OF* - *insertI1*])

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 (*induct arbitrary*: *y set*: *fold-graph*)
case *emptyI*
then show *?case*
by (*meson empty-fold-graphE*)
next
case (*insertI x B y v*)
from ‹*fold-graph f z* (*insert x B*) *v*› **and** ‹$x \notin B$› **and** ‹*insert x B* $\subseteq A$› **and** ‹$z \in A$›
obtain y' **where** $v = f\ x\ y'$ **and** *fold-graph f z B* y'
by (*rule fold-graph-insertE*)
from ‹*fold-graph f z B* y'› **and** ‹*insert x B* $\subseteq A$› **have** $y' = y$ **using** *insertI* **by** *auto*
with ‹$v = f\ x\ y'$› **show** $v = f\ x\ y$
by *simp*
qed

lemma *fold-equality*: *fold-graph f z B y* $\Longrightarrow$ $B \subseteq A$ $\Longrightarrow$ $z \in A$ $\Longrightarrow$ *Finite-Set.fold f z B* $= y$
by (*cases finite B*)
(*auto simp add*: *Finite-Set.fold-def intro*: *fold-graph-determ dest*: *fold-graph-finite*)

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 −
have $\exists\, x$. *fold-graph f z B x*
by (*rule finite-imp-fold-graph*[*OF f*])
moreover note *fold-graph-determ*
ultimately have $\exists!x$. *fold-graph f z B x* **using** *f BA z* **by** *auto*
then have *fold-graph f z B* (*The* (*fold-graph f z B*))
by (*rule theI'*)
with *assms* **show** *?thesis*
by (*simp add*: *Finite-Set.fold-def*)
qed

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 (*rule fold-equality*[*OF* - *BA z*])
from ‹*finite B*› **have** *fold-graph f z B* (*Finite-Set.fold f z B*)
using *BA fold-graph-fold z* **by** *auto*
hence *fold-graph f z* (*insert x B*) (*f x* (*Finite-Set.fold f z B*))
using *BA fold-graph.insertI assms* **by** *auto*

then show *fold-graph f z* (*insert x B*) (*f x* (*Finite-Set.fold f z B*))
by *simp*
qed
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 −
have *Finite-Set.fold f s S* = *Finite-Set.fold g s S*
using ‹*finite S*› *cong SA s*
proof (*induct S*)
case *empty*
then show *?case* **by** *simp*
next
case (*insert x F*)
interpret *f*: *comp-fun-commute-on f A* **by** (*fact f*)
interpret *g*: *comp-fun-commute-on g A* **by** (*fact g*)
show *?case* **using** *insert* **by** *auto*
qed
with *assms* **show** *?thesis* **by** *simp*
qed

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 −
have $(f\ x\ \widehat{}\widehat{}\ g\ x)\ y \in A$ **if** *y*: $y \in A$ **and** *x*: $x \in A$ **for** *x y*
using *x y*
proof (*induct g x arbitrary*: *g*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc n g*)
define *h* **where** $h\ z = g\ z - 1$ **for** *z*
have *hyp*: $(f\ x\ \widehat{}\widehat{}\ h\ x)\ y \in A$
using *h-def Suc.prems Suc.hyps diff-Suc-1* **by** *metis*
have $g\ x = Suc\ (h\ x)$ **unfolding** *h-def*
using *Suc.hyps*(*2*) **by** *auto*
then show *?case* **using** *f x hyp* **unfolding** *Pi-def* **by** *auto*
qed
thus *?thesis* **by** (*auto simp add*: *Pi-def*)
qed

lemma *comp-fun-commute-funpow*: *comp-fun-commute-on* (*λx. f x ^^ g x*) *A*
proof −
  have *f*: (*f y ^^ g y*) ((*f x ^^ g x*) *z*) = (*f x ^^ g x*) ((*f y ^^ g y*) *z*)
    if *x*: *x*∈*A* **and** *y*: *y* ∈ *A* **and** *z*: *z* ∈ *A* **for** *x y z*
  proof (*cases x* = *y*)
    case *False*
    show *?thesis*
    proof (*induct g x arbitrary*: *g*)
      case (*Suc n g*)
      have *hyp1*: (*f y ^^ g y*) (*f x k*) = *f x* ((*f y ^^ g y*) *k*) **if** *k*: *k* ∈ *A* **for** *k*
      proof (*induct g y arbitrary*: *g*)
        case *0*
        then show *?case* **by** *simp*
      next
        case (*Suc n g*)
        define *h* **where** *h z* = *g z* − *1* **for** *z*
        with *Suc* **have** *n* = *h y*
          by *simp*
        with *Suc* **have** *hyp*: (*f y ^^ h y*) (*f x k*) = *f x* ((*f y ^^ h y*) *k*)
          by *auto*
        from *Suc h-def* **have** *g*: *g y* = *Suc* (*h y*)
          by *simp*
        have ((*f y ^^ h y*) *k*) ∈ *A* **using** *y k comp-fun-Pi*[*of h*] **unfolding** *Pi-def* **by** *auto*
        then show *?case*
        by (*simp add*: *comp-assoc g hyp*) (*auto simp add*: *o-assoc comp-fun-commute-restrict x y k*)
      qed
      define *h* **where** *h a* = (*if a* = *x then g x* − *1 else g a*) **for** *a*
      with *Suc* **have** *n* = *h x*
        by *simp*
      with *Suc* **have** (*f y ^^ h y*) ((*f x ^^ h x*) *z*) = (*f x ^^ h x*) ((*f y ^^ h y*) *z*)
        by *auto*
      with *False* **have** *Suc2*: (*f x ^^ h x*) ((*f y ^^ g y*) *z*) = (*f y ^^ g y*) ((*f x ^^ h x*) *z*)
        using *h-def* **by** *auto*
      from *Suc h-def* **have** *g*: *g x* = *Suc* (*h x*)
        by *simp*
      have (*f x ^^ h x*) *z* ∈*A* **using** *comp-fun-Pi*[*of h*] *x z* **unfolding** *Pi-def* **by** *auto*
      hence ∗: (*f y ^^ g y*) (*f x* ((*f x ^^ h x*) *z*)) = *f x* ((*f y ^^ g y*) ((*f x ^^ h x*) *z*))
        using *hyp1* **by** *auto*
      thus *?case* **using** *g Suc2* **by** *auto*
    qed *simp*
  qed *simp*
  thus *?thesis* **by** (*auto simp add*: *comp-fun-commute-on-def comp-fun-Pi o-def*)

qed

lemma *fold-mset-add-mset*:
assumes *MA*: *set-mset* $M \subseteq A$ **and** *s*: $s \in A$ **and** *x*: $x \in A$
shows *fold-mset f s* (*add-mset x M*) = *f x* (*fold-mset f s M*)
proof −
interpret *mset*: *comp-fun-commute-on* λy. *f y* ^^ *count M y A*
by (*fact comp-fun-commute-funpow*)
interpret *mset-union*: *comp-fun-commute-on* λy. *f y* ^^ *count* (*add-mset x M*) *y A*
by (*fact comp-fun-commute-funpow*)
show *?thesis*
proof (*cases* $x \in$ *set-mset M*)
case *False*
then have ∗: *count* (*add-mset x M*) *x* = *1*
by (*simp add*: *not-in-iff*)
have *Finite-Set.fold* (λy. *f y* ^^ *count* (*add-mset x M*) *y*) *s* (*set-mset M*) =
Finite-Set.fold (λy. *f y* ^^ *count M y*) *s* (*set-mset M*)
by (*rule fold-cong*[*of - A*], *auto simp add*: *assms False comp-fun-commute-funpow*)
with *False* ∗ *s MA x* **show** *?thesis*
by (*simp add*: *fold-mset-def del*: *count-add-mset*)
next
case *True*
let *?f* = (λxa. *f xa* ^^ *count* (*add-mset x M*) *xa*)
let *?f2* = (λx. *f x* ^^ *count M x*)
define *N* **where** *N* = *set-mset M* − $\{x\}$
have *F*: *Finite-Set.fold ?f s* (*insert x N*) = *?f x* (*Finite-Set.fold ?f s N*)
by (*rule mset-union.fold-insert*, *auto simp add*: *assms N-def*)
have *F2*: *Finite-Set.fold ?f2 s* (*insert x N*) = *?f2 x* (*Finite-Set.fold ?f2 s N*)
by (*rule mset.fold-insert*, *auto simp add*: *assms N-def*)
from *N-def True* **have** ∗: *set-mset M* = *insert x N* $x \notin N$ *finite N* **by** *auto*
then have *Finite-Set.fold* (λy. *f y* ^^ *count* (*add-mset x M*) *y*) *s N* =
Finite-Set.fold (λy. *f y* ^^ *count M y*) *s N*
using *MA N-def s*
by (*auto intro*!: *fold-cong comp-fun-commute-funpow*)
with ∗ **show** *?thesis* **by** (*simp add*: *fold-mset-def del*: *count-add-mset*, *unfold F F2*, *auto*)
qed
qed
end

lemma *Diff-not-in*: $a \notin A - \{a\}$ **by** *auto*

context *abelian-group* **begin**

lemma *finsum-restrict*:
  assumes *fA*: $f : A \to carrier\ G$
    and *restr*: *restrict f A* = *restrict g A*
  shows *finsum G f A* = *finsum G g A*
proof (*rule finsum-cong′*;*rule?*)
  fix *a* **assume** *a*: $a : A$
  have *f a* = *restrict f A a* **using** *a* **by** *simp*
  also have ... = *restrict g A a* **using** *restr* **by** *simp*
  also have ... = *g a* **using** *a* **by** *simp*
  finally show *f a* = *g a*.
  thus $g\ a : carrier\ G$ **using** *fA a* **by** *force*
qed

lemma *minus-nonzero*: $x : carrier\ G \Longrightarrow x \neq \mathbf{0} \Longrightarrow \ominus\ x \neq \mathbf{0}$
  using *r-neg* **by** *force*

end

lemma (**in** *ordered-comm-monoid-add*) *positive-sum*:
  assumes $X : finite\ X$
    and $f : X \to \{\ y :: 'a.\ y \geq 0\ \}$
  shows $sum\ f\ X \geq 0 \wedge (sum\ f\ X = 0 \longrightarrow f\ `\ X \subseteq \{0\})$
  using *assms*
proof (*induct set:finite*)
  case (*insert x X*)
    hence *x0*: $f\ x \geq 0$ **and** *sum0*: $sum\ f\ X \geq 0$ **by** *auto*
    hence $sum\ f\ (insert\ x\ X) \geq 0$ **using** *insert* **by** *auto*
    moreover
    { assume $sum\ f\ (insert\ x\ X) = 0$
      hence $f\ x = 0\ sum\ f\ X = 0$
        using *sum0 x0 insert add-nonneg-eq-0-iff* **by** *auto*
    }
    ultimately show *?case* **using** *insert* **by** *blast*
qed *auto*

lemma *insert-union*: $insert\ x\ X = X \cup \{x\}$ **by** *auto*

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 –

```
  let ?f = λa u. a u ⊙V u
  have fa: ?f a : U → carrier V using a U by auto
  have restrict (?f a) U = restrict (?f b) U
  proof
    fix u
    have u : U ⟹ a u = b u using restr unfolding restrict-def by metis
    thus restrict (?f a) U u = restrict (?f b) U u by auto
  qed
  thus ?thesis
    unfolding lincomb-def using finsum-restrict[OF fa] by auto
qed

lemma lindep-span:
  assumes U: U ⊆ carrier V and finU: finite U
  shows lin-dep U = (∃ u ∈ U. u ∈ span (U − {u})) (is ?l = ?r)
proof
  assume l: ?l show ?r
  proof −
    from l[unfolded lin-dep-def]
    obtain A a u
    where finA: finite A
      and AU: A ⊆ U
      and aA: a : A → carrier K
      and aA0: lincomb a A = zero V
      and uA: u:A
      and au0: a u ≠ zero K by auto
    define a′ where a′ = (λv. (if v : A then a v else zero K))
    have a′U: a′ : U → carrier K unfolding a′-def using aA by auto
    have uU: u : U using uA AU by auto
    have a′u0: a′ u ≠ zero K unfolding a′-def using au0 uA by auto
    define B where B = U − A
    have B: B ⊆ carrier V unfolding B-def using U by auto
    have UAB: U = A ∪ B unfolding B-def using AU by auto
    have finB: finite B using finU B-def by auto
    have AB: A ∩ B = {} unfolding B-def by auto
    let ?f = λv. a v ⊙V v
    have fA: ?f : A → carrier V unfolding a′-def using aA AU U by auto
    let ?f′ = λv. a′ v ⊙V v
    have restrict ?f A = restrict ?f′ A unfolding a′-def by auto
    hence finsum: finsum V ?f′ A = finsum V ?f A
      using finsum-restrict[OF fA] by simp
    have f′A: ?f′ : A → carrier V
    proof
      fix x assume xA: x ∈ A
      show ?f′ x : carrier V unfolding a′-def using aA xA AU U by auto
    qed
    have f′B: ?f′ : B → carrier V
    proof
      fix x assume xB: x : B
```

```
    have x ∉ A using a'U xB unfolding B-def by auto
    thus ?f' x : carrier V using xB B unfolding a'-def by auto
  qed
  have sumB0: finsum V ?f' B = zero V
  proof −
    { fix B'
      have finite B' ⟹ B' ⊆ B ⟹ finsum V ?f' B' = zero V
      proof(induct set:finite)
        case (insert b B')
          have finB': finite B' and bB': b ∉ B' using insert by auto
          have f'B': ?f' : B' → carrier V using f'B insert by auto
          have bA: b ∉ A using insert unfolding B-def by auto
          have b: b : carrier V using insert B by auto
          have foo: a' b ⊙_V b ∈ carrier V unfolding a'-def using bA b by auto
          have IH: finsum V ?f' B' = zero V using insert by auto
          show ?case
            unfolding finsum-insert[OF finB' bB' f'B' foo]
            using IH a'-def bA b by auto
      qed auto
    }
    thus ?thesis using finB by auto
  qed
  have a'A0: lincomb a' U = zero V
    unfolding UAB
    unfolding lincomb-def
    unfolding finsum-Un-disjoint[OF finA finB AB f'A f'B]
    unfolding finsum
    unfolding aA0[unfolded lincomb-def]
    unfolding sumB0 by simp
  have uU: u:U using uA AU by auto
  moreover have u : span (U − {u})
    using lincomb-isolate(2)[OF finU U a'U uU a'u0 a'A0].
  ultimately show ?r by auto
qed
next assume r: ?r show ?l
proof −
  from r obtain u where uU: u : U and uspan: u : span (U−{u}) by auto
  hence u: u : carrier V using U by auto
  have finUu: finite (U−{u}) using finU by auto
  have Uu: U−{u} ⊆ carrier V using U by auto
  obtain a
    where ulin: u = lincomb a (U−{u})
      and a: a : U−{u} → carrier K
    using uspan unfolding finite-span[OF finUu Uu] by auto
  show ?l unfolding lin-dep-def
  proof(intro exI conjI)
    let ?a = λv. if v = u then ⊖_K one K else a v
    show ?a : U → carrier K using a by auto
    hence a': ?a : U−{u}∪{u} → carrier K by auto
```

```
      have U = U−{u}∪{u} using uU by auto
      also have lincomb ?a ... = ?a u ⊙V u ⊕V lincomb ?a (U−{u})
        unfolding lincomb-insert[OF finUu Uu a′ Diff-not-in u] by auto
      also have restrict a (U−{u}) = restrict ?a (U−{u}) by auto
        hence lincomb ?a (U−{u}) = lincomb a (U−{u})
          using lincomb-restrict[OF Uu a] by auto
      also have ?a u ⊙V u = ⊖V u using smult-minus-1[OF u] by simp
      also have lincomb a (U−{u}) = u using ulin..
      also have ⊖V u ⊕V u = zero V using l-neg[OF u].
      finally show lincomb ?a U = zero V by auto
    qed (insert uU finU, auto)
  qed
qed

lemma not-lindepD:
  assumes ~ lin-dep S
      and finite A A ⊆ S f : A → carrier K lincomb f A = zero V
    shows f : A → {zero K}
  using assms unfolding lin-dep-def by blast


lemma span-mem:
  assumes E: E ⊆ carrier V and uE: u : E shows u : span E
  unfolding span-def
proof (rule,intro exI conjI)
  show u = lincomb (λ-. one K) {u} unfolding lincomb-def using uE E by auto
  show {u} ⊆ E using uE by auto
qed auto

lemma lincomb-distrib:
  assumes U: U ⊆ carrier V
      and a: a : U → carrier K
      and c: c : carrier K
  shows c ⊙V lincomb a U = lincomb (λu. c ⊗K a u) U
    (is - = lincomb ?b U)
  using U a
proof (induct U rule: infinite-finite-induct)
  case empty show ?case unfolding lincomb-def using c by simp next
  case (insert u U)
    then have U: U ⊆ carrier V
          and u: u : carrier V
          and a: a : insert u U → carrier K
          and finU: finite U by auto
    hence aU: a : U → carrier K by auto
    have b: ?b : insert u U → carrier K using a c by auto
    show ?case
      unfolding lincomb-insert2[OF finU U a ‹u∉U› u]
      unfolding lincomb-insert2[OF finU U b ‹u∉U› u]
      using insert U aU c u smult-r-distr smult-assoc1 by auto next
```

case (*infinite U*)
thus *?case* **unfolding** *lincomb-def* **using** *assms* **by** *simp*
qed

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 −
have *Eu*[*simp*]: *insert u E* $\subseteq$ *carrier V* **by** *auto*
have *Ev*[*simp*]: *insert v E* $\subseteq$ *carrier V* **by** *auto*
have *finEu*: *finite* (*insert u E*) **and** *finEv*: *finite* (*insert v E*)
using *finE* **by** *auto*
have *uE*: $u \notin E$ **using** *usE span-mem* **by** *force*
have *vE*: $v \notin E$
proof
assume *v* : *E*
hence *EvE*: *insert v E* = *E* **using** *insert-absorb* **by** *auto*
hence *u* : *span E* **using** *usEv* **by** *auto*
thus *False* **using** *usE* **by** *auto*
qed
obtain *ua*
where *ua*[*simp*]: *ua* : (*insert v E*) $\rightarrow$ *carrier K*
and *uua*: *u* = *lincomb ua* (*insert v E*)
using *usEv finite-span*[*OF finEv Ev*] **by** *auto*
hence *uaE*[*simp*]: *ua* : $E \rightarrow$ *carrier K* **and** [*simp*]: *ua v* : *carrier K*
by *auto*

show $?L \subseteq ?R$
proof
fix *x* **assume** *x* : *?L*
then obtain *xa*
where *xa*: *xa* : *insert u E* $\rightarrow$ *carrier K*
and *xxa*: *x* = *lincomb xa* (*insert u E*)
unfolding *finite-span*[*OF finEu Eu*] **by** *auto*
hence *xaE*[*simp*]: *xa* : $E \rightarrow$ *carrier K* **and** *xau*[*simp*]: *xa u* : *carrier K* **by** *auto*
show *x* : *span* (*insert v E*)
unfolding *finite-span*[*OF finEv Ev*]
proof (*rule*,*intro exI conjI*)
define *a* **where** $a = (\lambda e.\ xa\ u \otimes_K ua\ e)$
define *a′* **where** $a' = (\lambda e.\ a\ e \oplus_K xa\ e)$
define *a″* **where** $a'' = (\lambda e.\ \mathit{if}\ e = v\ \mathit{then}\ xa\ u \otimes_K ua\ v\ \mathit{else}\ a'\ e)$
have *aE*: *a* : $E \rightarrow$ *carrier K* **unfolding** *a-def* **using** *xau uaE u* **by** *blast*
hence *a′E*: *a′* : $E \rightarrow$ *carrier K* **unfolding** *a′-def* **using** *xaE* **by** *blast*
thus *a″*: *a″* : *insert v E* $\rightarrow$ *carrier K* **unfolding** *a″-def* **by** *auto*

have *restr*: *restrict* a' *E* = *restrict* a'' *E*
  unfolding a''-*def* **using** *vE* **by** *auto*
have $x = xa\ u \odot_V u \oplus_V lincomb\ xa\ E$
  using *xxa lincomb-insert2 finE E xa uE u* **by** *auto*
also have
  $xa\ u \odot_V u = xa\ u \odot_V lincomb\ ua\ (insert\ v\ E)$
  using *uua* **by** *auto*
also have $lincomb\ ua\ (insert\ v\ E) = ua\ v \odot_V v \oplus_V lincomb\ ua\ E$
  using *lincomb-insert2 finE E ua vE v* **by** *auto*
also have $xa\ u \odot_V ... = xa\ u \odot_V (ua\ v \odot_V v) \oplus_V xa\ u \odot_V lincomb\ ua\ E$
  using *smult-r-distr* **by** *auto*
also have $xa\ u \odot_V lincomb\ ua\ E = lincomb\ a\ E$
  unfolding *a-def* **using** *lincomb-distrib*[*OF E*] **by** *auto*
finally have $x = xa\ u \odot_V (ua\ v \odot_V v) \oplus_V (lincomb\ a\ E \oplus_V lincomb\ xa\ E)$
  using *a-assoc xau v aE xaE* **by** *auto*
also have $lincomb\ a\ E \oplus_V lincomb\ xa\ E = lincomb\ a'\ E$
  unfolding a'-*def* **using** *lincomb-sum*[*OF finE E aE xaE*]**..**
also have $... = lincomb\ a''\ E$
  using *lincomb-restrict*[*OF E* $a'E$] *restr* **by** *auto*
finally have $x = ((xa\ u \otimes_K ua\ v) \odot_V v) \oplus_V lincomb\ a''\ E$
  using *smult-assoc1* **by** *auto*
also have $xa\ u \otimes_K ua\ v = a''\ v$ **unfolding** a''-*def* **by** *simp*
also have $(a''\ v \odot_V v) \oplus_V lincomb\ a''\ E = lincomb\ a''\ (insert\ v\ E)$
  using *lincomb-insert2*[*OF finE E* a'' *vE*] **by** *auto*
finally show $x = lincomb\ a''\ (insert\ v\ E)$**.**
qed
qed
qed

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*)
  unfolding *basis-def*
proof (*intro conjI*)
  have *Eu*[*simp*]: *insert u E* $\subseteq$ *carrier V*
  and *spanEu*: *carrier V* = *span* (*insert u E*)
  and *indEu*: ~ *lin-dep* (*insert u E*)
   using *b*[*unfolded basis-def*] **by** *auto*
  hence *E*[*simp*]: $E \subseteq$ *carrier V* **by** *auto*
  thus *Ev*[*simp*]: *insert v E* $\subseteq$ *carrier V* **using** *v* **by** *auto*
  have *finEu*: *finite* (*insert u E*)
  and *finEv*: *finite* (*insert v E*) **using** *finE* **by** *auto*
  have *usE*: $u \notin$ *span E*
  proof

assume u : *span E*
hence u : *span* (*insert u E* − {u}) **using** *uE* **by** *auto*
moreover have u : *insert u E* **by** *auto*
ultimately have *lin-dep* (*insert u E*)
unfolding *lindep-span*[*OF Eu finEu*] **by** *auto*
thus *False* **using** *b* **unfolding** *basis-def* **by** *auto*
qed

obtain *ua*
where *ua*[*simp*]: *ua* : *insert v E* → *carrier K*
and *uua*: u = *lincomb ua* (*insert v E*)
using *uEv finite-span*[*OF finEv Ev*] **by** *auto*
hence *uaE*[*simp*]: *ua* : *E* → *carrier K*
and *uav*[*simp*]: *ua v* : *carrier K*
by *auto*

have *vE*: $v \notin E$
proof
assume v : *E*
hence *EvE*: *insert v E* = *E* **using** *insert-absorb* **by** *auto*
hence u : *span E* **using** *uEv* **by** *auto*
thus *False* **using** *usE* **by** *auto*
qed
have *vsE*: $v \notin$ *span E*
proof
assume v : *span E*
then obtain *va*
where *va*[*simp*]: *va* : *E* → *carrier K*
and *vva*: v = *lincomb va E*
using *finite-span*[*OF finE E*] **by** *auto*
define *va′* **where** *va′* = (λe. *ua v* $\otimes_K$ *va e*)
define *va″* **where** *va″* = (λe. *va′ e* $\oplus_K$ *ua e*)
have *va′*[*simp*]: *va′* : *E* → *carrier K*
unfolding *va′-def* **using** *uav va* **by** *blast*
hence *va″*[*simp*]: *va″* : *E* → *carrier K*
unfolding *va″-def* **using** *ua* **by** *blast*
from *uua*
have u = *ua v* $\odot_V$ v $\oplus_V$ *lincomb ua E*
using *lincomb-insert2 vE* **by** *auto*
also have *ua v* $\odot_V$ v = *ua v* $\odot_V$ (*lincomb va E*)
using *vva* **by** *auto*
also have ... = *lincomb va′ E*
unfolding *va′-def* **using** *lincomb-distrib* **by** *auto*
finally have u = *lincomb va″ E*
unfolding *va″-def*
using *lincomb-sum*[*OF finE E*] **by** *auto*
hence u : *span E* **using** *finite-span*[*OF finE E*] *va″* **by** *blast*
hence *lin-dep* (*insert u E*) **using** *lindep-span* **by** *simp*
then show *False* **using** *indEu* **by** *auto*

qed

have *indE*: ~ *lin-dep E* **using** *indEu subset-li-is-li* **by** *auto*

show ~ *lin-dep* (*insert v E*) **using** *lin-dep-iff-in-span*[*OF E indE v vE*] *vsE* **by** *auto*

show *span* (*insert v E*) = *carrier V* (**is** *?L* = *?R*)
proof (*rule*)
show *?L* ⊆ *?R*
proof
have *finEv*: *finite* (*insert v E*) **using** *finE* **by** *auto*
fix *x* **assume** *x* : *?L*
then obtain *a*
where *a*: *a* : *insert v E* → *carrier K*
and *x*: *x* = *lincomb a* (*insert v E*)
unfolding *finite-span*[*OF finEv Ev*] **by** *auto*
from *a* **have** *av*: *a v* : *carrier K* **by** *auto*
from *a* **have** *a2*: *a* : *E* → *carrier K* **by** *auto*
show *x* : *?R*
unfolding *x*
unfolding *lincomb-insert2*[*OF finE E a vE v*]
using *lincomb-closed*[*OF E a2*] *av v*
by *auto*
qed
show *?R* ⊆ *?L*
using *span-swap*[*OF finE E u usE v uEv*] *spanEu* **by** *auto*
qed
qed

lemma *span-empty*: *span* {} = {*zero V*}
unfolding *finite-span*[*OF finite.emptyI empty-subsetI*]
unfolding *lincomb-def* **by** *simp*

lemma *span-self*: **assumes** [*simp*]: *v* : *carrier V* **shows** *v* : *span* {*v*}
proof −
have *v* = *lincomb* (*λx. one K*) {*v*} **unfolding** *lincomb-def* **by** *auto*
thus *?thesis* **using** *finite-span* **by** *auto*
qed

lemma *span-zero*: *zero V* : *span U* **unfolding** *span-def lincomb-def* **by** *force*

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*
apply *rule* **unfolding** *emb-def* **by** *auto*

lemma *emb-restrict*: *restrict* (*emb f D*) *D* = *restrict f D*
apply *rule* **unfolding** *restrict-def emb-def* **by** *auto*

```
lemma emb-zero: emb f D : X − D → { zero K }
  apply rule unfolding emb-def by auto

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
  using finZ Z aZ
proof(induct set:finite)
  case empty thus ?case by simp next
  case (insert z Z) show ?case
    proof (cases z : A)
      case True hence A ∪ insert z Z = A ∪ Z by auto
        thus ?thesis using insert by simp next
      case False
        have finAZ: finite (A ∪ Z) using finA insert by simp
        have AZ: A ∪ Z ⊆ carrier V using A insert by simp
        have a: a : insert z (A ∪ Z) → carrier K using insert aA by force
        have a z = zero K using insert by auto
        also have ... ⊙V z = zero V using insert by auto
        also have ... ⊕V lincomb a (A ∪ Z) = lincomb a (A ∪ Z)
          using insert AZ aA by auto
        also have ... = lincomb a A using insert by simp
        finally have a z ⊙V z ⊕V lincomb a (A ∪ Z) = lincomb a A.
        thus ?thesis
          using lincomb-insert2[OF finAZ AZ a] False insert by auto
    qed
qed

lemma span-add1:
  assumes U: U ⊆ carrier V and v: v : span U and w: w : span U
  shows v ⊕V w : span U
proof −
  from v obtain a A
    where finA: finite A
      and va: lincomb a A = v
      and AU: A ⊆ U
      and a: a : A → carrier K
    unfolding span-def by auto
  hence A: A ⊆ carrier V using U by auto
  from w obtain b B
    where finB: finite B
      and wb: lincomb b B = w
      and BU: B ⊆ U
```

and *b*: $b : B \rightarrow carrier\ K$
unfolding *span-def* **by** *auto*
hence *B*: $B \subseteq carrier\ V$ **using** *U* **by** *auto*

have *B-A*: $B - A \subseteq carrier\ V$ **and** *A-B*: $A - B \subseteq carrier\ V$ **using** *A B* **by** *auto*

have a': $emb\ a\ A : A \cup B \rightarrow carrier\ K$
apply (*rule Pi-I*) **unfolding** *emb-def* **using** *a* **by** *auto*
hence $a'A$: $emb\ a\ A : A \rightarrow carrier\ K$ **by** *auto*
have $a'Z$: $emb\ a\ A : B - A \rightarrow \{\ zero\ K\ \}$
apply (*rule Pi-I*) **unfolding** *emb-def* **using** *a* **by** *auto*

have b': $emb\ b\ B : A \cup B \rightarrow carrier\ K$
apply (*rule Pi-I*) **unfolding** *emb-def* **using** *b* **by** *auto*
hence $b'B$: $emb\ b\ B : B \rightarrow carrier\ K$ **by** *auto*
have $b'Z$: $emb\ b\ B : A - B \rightarrow \{\ zero\ K\ \}$
apply (*rule Pi-I*) **unfolding** *emb-def* **using** *b* **by** *auto*

show *?thesis*
unfolding *span-def*
proof (*rule,intro exI conjI*)
let $?v = lincomb\ (emb\ a\ A)\ (A \cup B)$
let $?w = lincomb\ (emb\ b\ B)\ (A \cup B)$
let $?ab = \lambda u.\ (emb\ a\ A)\ u \oplus_K (emb\ b\ B)\ u$
show *finAB*: *finite* $(A \cup B)$ **using** *finA finB* **by** *auto*
show $A \cup B \subseteq U$ **using** *AU BU* **by** *auto*
show $?ab : A \cup B \rightarrow carrier\ K$ **using** $a'\ b'$ **by** *auto*
have $v = ?v$
using *va lincomb-restrict*[*OF A a emb-restrict*[*symmetric*]]
using *lincomb-clean*[*OF A B-A*] $a'A\ a'Z$ *finA finB* **by** *simp*
moreover have $w = ?w$
apply (*subst Un-commute*)
using *wb lincomb-restrict*[*OF B b emb-restrict*[*symmetric*]]
using *lincomb-clean*[*OF B A-B*] *finA finB* $b'B\ b'Z$ **by** *simp*
ultimately show $v \oplus_V w = lincomb\ ?ab\ (A \cup B)$
using *lincomb-sum*[*OF finAB*] *A B* $a'\ b'$ **by** *simp*
qed
qed

lemma *span-neg*:
assumes *U*: $U \subseteq carrier\ V$ **and** *vU*: $v : span\ U$
shows $\ominus_V v : span\ U$
proof −
have *v*: $v : carrier\ V$ **using** *vU U* **unfolding** *span-def* **by** *auto*
from *vU*[*unfolded span-def*]
obtain *a A*
where *finA*: *finite A*
and *AU*: $A \subseteq U$

and *a*: $a \in A \to$ *carrier K*
and *va*: $v =$ *lincomb a A* **by** *auto*
hence *A*: $A \subseteq$ *carrier V* **using** *U* **by** *simp*
let *?a* = $\lambda u.\ \ominus_K$ *one K* $\otimes_K$ *a u*

have $\ominus_V v = \ominus_K$ *one K* $\odot_V$ *v* **using** *smult-minus-1-back*[*OF v*]**.**
also have ... $= \ominus_K$ *one K* $\odot_V$ *lincomb a A* **using** *va* **by** *simp*
finally have *main*: $\ominus_V v =$ *lincomb ?a A*
unfolding *lincomb-distrib*[*OF A a R.a-inv-closed*[*OF R.one-closed*]] **by** *auto*
show *?thesis*
unfolding *span-def*
apply *rule*
using *main a finA AU* **by** *force*
qed

lemma *span-closed*[*simp*]: $U \subseteq$ *carrier V* $\Longrightarrow v :$ *span U* $\Longrightarrow v :$ *carrier V*
unfolding *span-def* **by** *auto*

lemma *span-add*:
assumes *U*: $U \subseteq$ *carrier V* **and** *vU*: $v :$ *span U* **and** *w*[*simp*]: $w :$ *carrier V*
shows $w :$ *span U* $\longleftrightarrow v \oplus_V w :$ *span U* (**is** *?L* $\longleftrightarrow$ *?R*)
proof
show *?L* $\Longrightarrow$ *?R* **using** *span-add1*[*OF U vU*] **by** *auto*
assume *R*: *?R* **show** *?L*
proof −
have *v*[*simp*]: $v :$ *carrier V* **using** *vU U* **by** *simp*
have $w =$ *zero V* $\oplus_V w$ **using** *M.l-zero* **by** *auto*
also have ... $= \ominus_V v \oplus_V v \oplus_V w$ **using** *M.l-neg* **by** *auto*
also have ... $= \ominus_V v \oplus_V (v \oplus_V w)$
using *M.l-zero M.a-assoc M.a-closed* **by** *auto*
also have ... : *span U* **using** *span-neg*[*OF U vU*] *span-add1*[*OF U*] *R* **by** *auto*
finally show *?thesis***.**
qed
qed

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*: $a : U \cup U' \to$ *carrier K*
shows *lincomb a* $(U \cup U') =$ *lincomb a U* $\oplus_V$ *lincomb a U'*
using *finU U disj a*
proof (*induct set*:*finite*)
case *empty* **thus** *?case* **by** (*subst*(*2*) *lincomb-def*, *simp*) **next**
case (*insert u U*) **thus** *?case*
unfolding *Un-insert-left*

using *lincomb-insert2 finU′ insert a-assoc* **by** *auto*
qed

lemma *span-union1*:
assumes *U*: *U* ⊆ *carrier V* **and** *U′*: *U′* ⊆ *carrier V* **and** *UU′*: *span U* = *span U′*
and *W*: *W* ⊆ *carrier V* **and** *W′*: *W′* ⊆ *carrier V* **and** *WW′*: *span W* = *span W′*
shows *span* (*U* ∪ *W*) ⊆ *span* (*U′* ∪ *W′*) (**is** *?L* ⊆ *?R*)
proof
fix *x* **assume** *x* : *?L*
then obtain *a A*
where *finA*: *finite A*
and *AUW*: *A* ⊆ *U* ∪ *W*
and *x*: *x* = *lincomb a A*
and *a*: *a* : *A* → *carrier K*
unfolding *span-def* **by** *auto*
let *?AU* = *A* ∩ *U* **and** *?AW* = *A* ∩ *W* − *U*
have *AU*: *?AU* ⊆ *carrier V* **using** *U* **by** *auto*
have *AW*: *?AW* ⊆ *carrier V* **using** *W* **by** *auto*
have *disj*: *?AU* ∩ *?AW* = {} **by** *auto*
have *U′W′*: *U′* ∪ *W′* ⊆ *carrier V* **using** *U′ W′* **by** *auto*

have *?AU* ∪ *?AW* = *A* **using** *AUW* **by** *auto*
hence *x* = *lincomb a* (*?AU* ∪ *?AW*) **using** *x* **by** *auto*
hence *x* = *lincomb a ?AU* $\oplus_V$ *lincomb a ?AW*
using *lincomb-union*[*OF AU AW disj*] *finA a* **by** *auto*
moreover
have *lincomb a ?AU* : *span U* **and** *lincomb a ?AW* : *span W*
unfolding *span-def* **using** *AU a finA* **by** *auto*
hence *lincomb a ?AU* : *span U′* **and** *lincomb a ?AW* : *span W′*
using *UU′ WW′* **by** *auto*
hence *lincomb a ?AU* : *?R* **and** *lincomb a ?AW* : *?R*
using *span-is-monotone*[*OF Un-upper1*, *of U′*]
using *span-is-monotone*[*OF Un-upper2*, *of W′*] **by** *auto*
ultimately
show *x* : *?R* **using** *span-add1*[*OF U′W′*] **by** *auto*
qed

lemma *span-Un*:
assumes *U*: *U* ⊆ *carrier V* **and** *U′*: *U′* ⊆ *carrier V* **and** *UU′*: *span U* = *span U′*
and *W*: *W* ⊆ *carrier V* **and** *W′*: *W′* ⊆ *carrier V* **and** *WW′*: *span W* = *span W′*
shows *span* (*U* ∪ *W*) = *span* (*U′* ∪ *W′*) (**is** *?L* = *?R*)
using *span-union1*[*OF assms*]
using *span-union1*[*OF U′ U UU′*[*symmetric*] *W′ W WW′*[*symmetric*]]
by *auto*

lemma *lincomb-zero*:
  assumes *U*: $U \subseteq$ *carrier* V **and** *a*: $a : U \to \{$*zero* $K\}$
  shows *lincomb* a U = *zero* V
  using U a
proof (*induct U rule*: *infinite-finite-induct*)
  case *empty* **show** *?case* **unfolding** *lincomb-def* **by** *auto* **next**
  case (*insert u U*)
    hence $a \in$ *insert* u $U \to$ *carrier* K **using** *zero-closed* **by** *force*
    thus *?case* **using** *insert* **by** (*subst lincomb-insert2*; *auto*)
qed (*auto simp*: *lincomb-def*)

end

context *module*
begin

lemma *lincomb-empty*[*simp*]: *lincomb* a $\{\} = \mathbf{0}_M$
  unfolding *lincomb-def* **by** *auto*

end

context *linear-map*
begin

interpretation *Ker*: *vectorspace* K (*V.vs kerT*)
  using *kerT-is-subspace*
  using *V.subspace-is-vs* **by** *blast*

interpretation *im*: *vectorspace* K (*W.vs imT*)
  using *imT-is-subspace*
  using *W.subspace-is-vs* **by** *blast*

lemma *inj-imp-Ker0*:
assumes *inj-on* T (*carrier* V)
shows *carrier* (*V.vs kerT*) = $\{\mathbf{0}_V\}$
  unfolding *mod-hom.ker-def*
  using *assms inj-on-contraD* **by** *fastforce*

lemma *Ke0-imp-inj*:
assumes *c*: *carrier* (*V.vs kerT*) = $\{\mathbf{0}_V\}$
shows *inj-on* T (*carrier* V)
proof (*auto simp add*: *inj-on-def*)
  fix x y
  assume *x*: $x \in$ *carrier* V **and** *y*: $y \in$ *carrier* V
  and *Tx-Ty*: $T\ x = T\ y$
  hence $T\ x \ominus_W T\ y = \mathbf{0}_W$ **using** *W.module.M.minus-other-side* **by** *auto*
  hence $T\ (x \ominus_V y) = \mathbf{0}_W$ **by** (*simp add*: *x y*)
  hence $x \ominus_V y \in$ *carrier* (*V.vs kerT*) **by** (*simp add*: *mod-hom.ker-def x y*)
  hence $x \ominus_V y = \mathbf{0}_V$ **using** *c* **by** *fast*

thus $x = y$ **by** (*simp add*: *x y*)
qed

corollary *Ke0-iff-inj*: *inj-on T* (*carrier V*) = (*carrier* (*V.vs kerT*) = $\{\mathbf{0}_V\}$)
using *inj-imp-Ker0 Ke0-imp-inj* **by** *auto*

lemma *inj-imp-dim-ker0*:
assumes *inj-on T* (*carrier V*)
shows *vectorspace.dim K* (*V.vs kerT*) = *0*
proof (*unfold Ker.dim-def*, *rule Least-eq-0*, *rule exI*[*of* - {}])
  have *Ker-rw*: *carrier* (*V.vs kerT*) = $\{\mathbf{0}_V\}$
    unfolding *mod-hom.ker-def*
    using *assms inj-on-contraD* **by** *fastforce*
  have *finite* {} **by** *simp*
  moreover have *card* {} = *0* **by** *simp*
  moreover have {} ⊆ *carrier* (*V.vs kerT*) **by** *simp*
  moreover have *Ker.gen-set* {} **unfolding** *Ker-rw* **by** (*simp add*: *Ker.span-empty*)
  ultimately show *finite* {} ∧ *card* {} = *0* ∧ {} ⊆ *carrier* (*V.vs kerT*) ∧ *Ker.gen-set* {} **by** *simp*
qed

lemma *surj-imp-imT-carrier*:
assumes *surj*: *T‘* (*carrier V*) = *carrier W*
shows (*imT*) = *carrier W*
by (*simp add*: *surj im-def*)

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 −
  have *dim0*: *vectorspace.dim K* (*V.vs kerT*) = *0*
    by (*rule inj-imp-dim-ker0*[*OF i*])
  have *imT-W*: (*imT*) = *carrier W*
    by (*rule surj-imp-imT-carrier*[*OF surj*])
  have *rnt*: *vectorspace.dim K* (*W.vs imT*) + *vectorspace.dim K* (*V.vs kerT*) = *V.dim*
    by (*rule rank-nullity*[*OF fin-dim-V*])
  hence *V.dim* = *vectorspace.dim K* (*W.vs imT*) **using** *dim0* **by** *auto*
  also have ... = *W.dim* **using** *imT-W* **by** *auto*
  finally show *?thesis* **using** *fin-dim-V* **by** *auto*
qed

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 $(T`A) = T$ $(V.module.lincomb$ $(a \circ T)$ $A)$
using f **using** $A\text{-}in\text{-}V$ a
proof (*induct A*)
  case *empty* **thus** *?case* **by** *auto*
next
  case (*insert x A*)
  have *T-insert-rw*: T ` *insert* $x$ $A$ = *insert* $(T\ x)$ $(T`\ A)$ **by** *simp*
  have $W.module.lincomb$ a $(T$ ` *insert* $x$ $A)$ = $W.module.lincomb$ $a$ (*insert* $(T\ x)$ $(T`\ A))$
    unfolding *T-insert-rw* **..**
  also have ... = $a\ (T\ x) \odot_W (T\ x) \oplus_W$ $W.module.lincomb$ a $(T`\ A)$
  proof (*rule W.lincomb-insert2*)
    show *finite* $(T$ ` $A)$ **by** (*simp add*: *insert.hyps*(*1*))
    show T ` $A \subseteq$ *carrier W* **using** *insert.prems*(*1*) **by** *auto*
    show $a \in$ *insert* $(T\ x)$ $(T$ ` $A) \to$ *carrier K*
      using *insert.prems*(*2*) **by** *blast*
    show $T\ x \notin T$ ` A
      by (*meson inj-T inj-on-image-mem-iff insert.hyps*(*2*) *insert.prems*(*1*) *insert-subset*)
    show $T\ x \in$ *carrier W* **using** *insert.prems*(*1*) **by** *blast*
  qed
  also have ... = $a\ (T\ x) \odot_W (T\ x) \oplus_W (T\ (V.module.lincomb\ (a \circ T)\ A))$
    using *insert.hyps*(*3*) *insert.prems*(*1*) *insert.prems*(*2*) **by** *fastforce*
  also have ... = $T\ (a\ (T\ x) \odot_V x) \oplus_W (T\ (V.module.lincomb\ (a \circ T)\ A))$
    using *insert.prems*(*1*) *insert.prems*(*2*) **by** *auto*
  also have ... = $T\ ((a\ (T\ x) \odot_V x) \oplus_V (V.module.lincomb\ (a \circ T)\ A))$
  proof (*rule T-add*[*symmetric*])
    show $a\ (T\ x) \odot_V x \in$ *carrier V* **using** *insert.prems*(*1*) *insert.prems*(*2*) **by** *auto*
    show $V.module.lincomb$ $(a \circ T)$ $A \in$ *carrier V*
    proof (*rule V.module.lincomb-closed*)
      show $A \subseteq$ *carrier V* **using** *insert.prems*(*1*) **by** *blast*
      show $a \circ T \in A \to$ *carrier K* **using** *coeff-in-ring insert.prems*(*2*) **by** *auto*
    qed
  qed
  also have ... = $T\ (V.module.lincomb\ (a \circ T)$ (*insert x A*))
  proof (*rule arg-cong*[*of - - T*])
    have $a \circ T \in$ *insert* x $A \to$ *carrier K*
      using *comp-def insert.prems*(*2*) **by** *auto*
    then show $a\ (T\ x) \odot_V x \oplus_V$ $V.module.lincomb$ $(a \circ T)$ A
      = $V.module.lincomb$ $(a \circ T)$ (*insert x A*)
      using *V.lincomb-insert2 insert.hyps*(*1*) *insert.hyps*(*2*) *insert.prems*(*1*) **by** *force*
  qed
  finally show *?case* **.**
qed

lemma *surj-fin-dim*:
  assumes *fd*: *V.fin-dim* **and** *surj*: *T‘ (carrier V) = carrier W*
  shows *image-fin-dim*: *W.fin-dim*
    using *rank-nullity-main(2)[OF fd surj]* **.**

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 (*rule W.dim-li-is-basis*)
  have *lm*: *linear-map K V W T* **by** *intro-locales*
  have *inj-TB*: *inj-on T B*
    by (*meson basis-B inj-T subset-inj-on V.basis-def*)
  show *W.fin-dim* **by** (*rule surj-fin-dim[OF fin-dim-V surj]*)
  show *finite (T ‘ B)*
  proof (*rule finite-imageI*, *rule V.fin[OF fin-dim-V]*)
    show *V.module.lin-indpt B* **using** *basis-B* **unfolding** *V.basis-def* **by** *auto*
    show *B ⊆ carrier V* **using** *basis-B* **unfolding** *V.basis-def* **by** *auto*
  qed
  show *T ‘ B ⊆ carrier W* **using** *basis-B* **unfolding** *V.basis-def* **by** *auto*
  show *W.dim ≤ card (T ‘ B)*
  proof −
    have *d*: *V.dim = W.dim* **by** (*rule dim-eq[OF fin-dim-V inj-T surj]*)
    have *card (T‘ B) = card B* **by** (*simp add*: *card-image inj-TB*)
    also have ... = *V.dim* **using** *basis-B fin-dim-V V.basis-def V.dim-basis V.fin*
by *auto*
    finally show *?thesis* **using** *d* **by** *simp*
  qed
  show *W.module.lin-indpt (T ‘ B)*
  proof (*rule W.module.finite-lin-indpt2*)
     show *fin-TB*: *finite (T ‘ B)* **by** *fact*
     show *TB-W*: *T ‘ B ⊆ carrier W* **by** *fact*
     fix *a* **assume** *a*: *a ∈ T ‘ B → carrier K* **and** *lc-a*: *W.module.lincomb a (T ‘
B) =* $\mathbf{0}_W$
     show *∀ v∈T ‘ B. a v =* $\mathbf{0}_K$
     proof (*rule ballI*)
      fix *v* **assume** *v*: *v ∈ T ‘ B*
      have *W.module.lincomb a (T ‘ B) = T (V.module.lincomb (a ∘ T) B)*
      proof (*rule lincomb-linear-image[OF inj-T]*)
      show *B ⊆ carrier V* **using** *V.vectorspace-axioms basis-B vectorspace.basis-def*
by *blast*
        show *a ∈ T ‘ B → carrier K* **by** (*simp add*: *a*)
        show *finite B* **using** *fin-TB finite-image-iff inj-TB* **by** *blast*
      qed
       hence *T-lincomb*: *T (V.module.lincomb (a ∘ T) B) =* $\mathbf{0}_W$ **using** *lc-a* **by**
simp
      have *lincomb-0*: *V.module.lincomb (a ∘ T) B =* $\mathbf{0}_V$
      proof −

have $a \circ T \in B \to$ *carrier K*
using *a* **by** *auto*
then show *?thesis*
by (*metis V.module.M.zero-closed V.module.lincomb-closed T-lincomb basis-B f0-is-0 inj-T inj-onD V.basis-def*)
qed
have $(a \circ T) \in B \to \{\mathbf{0}_K\}$
proof (*rule V.not-lindepD*[*OF - - - - lincomb-0*])
show *V.module.lin-indpt B* **using** *V.basis-def basis-B* **by** *blast*
show *finite B* **using** *fin-TB finite-image-iff inj-TB* **by** *auto*
show $B \subseteq B$ **by** *auto*
show $a \circ T \in B \to$ *carrier K* **using** *a* **by** *auto*
qed
thus $a\ v = \mathbf{0}_K$ **using** *v* **by** *auto*
qed
qed
qed

end

lemma (**in** *vectorspace*) *dim1I*:
assumes *gen-set* $\{v\}$
assumes $v \neq \mathbf{0}_V$ $v \in$ *carrier V*
shows *dim = 1*
proof −
have *basis* $\{v\}$ **by** (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *basis-def empty-iff empty-subsetI finite.emptyI finite-lin-indpt2 insert-iff insert-subset insert-union lin-dep-iff-in-span span-empty*)
then show *?thesis* **using** *dim-basis* **by** *force*
qed

lemma (**in** *vectorspace*) *dim0I*:
assumes *gen-set* $\{\mathbf{0}_V\}$
shows *dim = 0*
proof −
have *basis* {} **unfolding** *basis-def* **using** *already-in-span assms finite-lin-indpt2 span-zero* **by** *auto*
then show *?thesis* **using** *dim-basis* **by** *force*
qed

lemma (**in** *vectorspace*) *dim-le1I*:
assumes *gen-set* $\{v\}$
assumes $v \in$ *carrier V*
shows $dim \leq 1$
by (*metis One-nat-def assms*(*1*) *assms*(*2*) *bot.extremum card.empty card.insert empty-iff finite.intros*(*1*)
finite.intros(*2*) *insert-subset vectorspace.gen-ge-dim vectorspace-axioms*)

definition *find-indices* **where** *find-indices x xs* $\equiv [i \leftarrow [0..<length\ xs].\ xs!i = x]$

lemma *find-indices-Nil* [*simp*]:
  find-indices x [] = []
  by (*simp add*: *find-indices-def*)

lemma *find-indices-Cons*:
  find-indices x (*y#ys*) = (*if x* = *y then Cons 0 else id*) (*map Suc* (*find-indices x ys*))
apply (*unfold find-indices-def length-Cons*, *subst upt-conv-Cons*, *simp*)
apply (*fold map-Suc-upt*, *auto simp*: *filter-map o-def*) **done**

lemma *find-indices-snoc* [*simp*]:
  find-indices x (*ys*@[*y*]) = *find-indices x ys* @ (*if x* = *y then* [*length ys*] *else* [])
  by (*unfold find-indices-def*, *auto intro*!: *filter-cong simp*: *nth-append*)

lemma *mem-set-find-indices* [*simp*]: $i \in$ *set* (*find-indices x xs*) $\longleftrightarrow$ $i <$ *length xs* $\wedge$ *xs*!*i* = *x*
  by (*auto simp*: *find-indices-def*)

lemma *distinct-find-indices*: *distinct* (*find-indices x xs*)
  unfolding *find-indices-def* **by** *simp*

context *abelian-monoid* **begin**

definition *sumlist*
  where *sumlist xs* $\equiv$ *foldr* ($\oplus$) *xs* **0**

lemma [*simp*]:
  shows *sumlist-Cons*: *sumlist* (*x#xs*) = *x* $\oplus$ *sumlist xs*
    and *sumlist-Nil*: *sumlist* [] = **0**
  by (*simp-all add*: *sumlist-def*)

lemma *sumlist-carrier* [*simp*]:
  assumes *set xs* $\subseteq$ *carrier G* **shows** *sumlist xs* $\in$ *carrier G*
  using *assms* **by** (*induct xs*, *auto*)

lemma *sumlist-neutral*:
  assumes *set xs* $\subseteq$ {**0**} **shows** *sumlist xs* = **0**
proof (*insert assms*, *induct xs*)
  case (*Cons x xs*)
  then have *x* = **0** **and** *set xs* $\subseteq$ {**0**} **by** *auto*
  with *Cons.hyps* **show** *?case* **by** *auto*
qed *simp*

lemma *sumlist-append*:
  assumes *set xs* $\subseteq$ *carrier G* **and** *set ys* $\subseteq$ *carrier G*
  shows *sumlist* (*xs* @ *ys*) = *sumlist xs* $\oplus$ *sumlist ys*
proof (*insert assms*, *induct xs arbitrary*: *ys*)

case (*Cons x xs*)
have *sumlist* (*xs* @ *ys*) = *sumlist xs* ⊕ *sumlist ys*
using *Cons.prems* **by** (*auto intro*: *Cons.hyps*)
with *Cons.prems* **show** *?case* **by** (*auto intro*!: *a-assoc*[*symmetric*])
qed *auto*

lemma *sumlist-snoc*:
assumes *set xs* ⊆ *carrier G* **and** *x* ∈ *carrier G*
shows *sumlist* (*xs* @ [*x*]) = *sumlist xs* ⊕ *x*
by (*subst sumlist-append*, *insert assms*, *auto*)

lemma *sumlist-as-finsum*:
assumes *set xs* ⊆ *carrier G* **and** *distinct xs* **shows** *sumlist xs* = ($\bigoplus$ *x*∈*set xs*. *x*)
using *assms* **by** (*induct xs*, *auto intro*:*finsum-insert*[*symmetric*])

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*)
using *assms* **by** (*induct xs*, *auto*)

definition *summset* **where** *summset M* ≡ *fold-mset* (⊕) **0** *M*

lemma *summset-empty* [*simp*]: *summset* {#} = **0** **by** (*simp add*: *summset-def*)

lemma *fold-mset-add-carrier*: *a* ∈ *carrier G* ⟹ *set-mset M* ⊆ *carrier G* ⟹ *fold-mset* (⊕) *a M* ∈ *carrier G*
proof (*induct M arbitrary*: *a*)
case (*add x M*)
thus *?case* **by**
(*subst comp-fun-commute-on.fold-mset-add-mset*[*of - carrier G*], *unfold-locales*, *auto simp*: *a-lcomm*)
qed *simp*

lemma *summset-carrier*[*intro*]: *set-mset M* ⊆ *carrier G* ⟹ *summset M* ∈ *carrier G*
unfolding *summset-def* **by** (*rule fold-mset-add-carrier*, *auto*)

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*
using *assms*
by (*auto simp add*: *summset-def*)
(*rule comp-fun-commute-on.fold-mset-add-mset*, *unfold-locales*, *auto simp add*: *a-lcomm*)

lemma *sumlist-as-summset*:
assumes *set xs* ⊆ *carrier G* **shows** *sumlist xs* = *summset* (*mset xs*)
by (*insert assms*, *induct xs*, *auto*)

lemma *sumlist-rev*:
  assumes *set xs* $\subseteq$ *carrier G*
  shows *sumlist* (*rev xs*) = *sumlist xs*
  using *assms* **by** (*simp add*: *sumlist-as-summset*)

lemma *sumlist-as-fold*:
  assumes *set xs* $\subseteq$ *carrier G*
  shows *sumlist xs* = *fold* ($\oplus$) *xs* **0**
  by (*fold sumlist-rev*[*OF assms*], *simp add*: *sumlist-def foldr-conv-fold*)

end

context *Module.module* **begin**

definition *lincomb-list*
where *lincomb-list c vs* = *sumlist* (*map* ($\lambda i.$ *c i* $\odot_M$ *vs* ! *i*) [*0..<length vs*])

lemma *lincomb-list-carrier*:
  assumes *set vs* $\subseteq$ *carrier M* **and** *c* : {*0..<length vs*} $\to$ *carrier R*
  shows *lincomb-list c vs* $\in$ *carrier M*
  by (*insert assms*, *unfold lincomb-list-def*, *intro sumlist-carrier*, *auto intro*!: *smult-closed*)

lemma *lincomb-list-Nil* [*simp*]: *lincomb-list c* [] = $\mathbf{0}_M$
  by (*simp add*: *lincomb-list-def*)

lemma *lincomb-list-Cons* [*simp*]:
  lincomb-list c (*v#vs*) = *c 0* $\odot_M$ *v* $\oplus_M$ *lincomb-list* (*c o Suc*) *vs*
  by (*unfold lincomb-list-def length-Cons*, *subst upt-conv-Cons*, *simp*, *fold map-Suc-upt*, *simp add*: *o-def*)

lemma *lincomb-list-eq-0*:
  assumes $\bigwedge i.$ *i* < *length vs* $\Longrightarrow$ *c i* $\odot_M$ *vs* ! *i* = $\mathbf{0}_M$
  shows *lincomb-list c vs* = $\mathbf{0}_M$
proof (*insert assms*, *induct vs arbitrary*:*c*)
  case (*Cons v vs*)
  from *Cons.prems*[*of 0*] **have** [*simp*]: *c 0* $\odot_M$ *v* = $\mathbf{0}_M$ **by** *auto*
  from *Cons.prems*[*of Suc -*] *Cons.hyps* **have** *lincomb-list* (*c* $\circ$ *Suc*) *vs* = $\mathbf{0}_M$ **by** *auto*
  then show *?case* **by** (*simp add*: *o-def*)
qed *simp*

definition *mk-coeff* **where** *mk-coeff vs c v* $\equiv$ *R.sumlist* (*map c* (*find-indices v vs*))

lemma *mk-coeff-carrier*:
  assumes *c* : {*0..<length vs*} $\to$ *carrier R* **shows** *mk-coeff vs c w* $\in$ *carrier R*
  by (*insert assms*, *auto simp*: *mk-coeff-def find-indices-def intro*!:*R.sumlist-carrier elim*!:*funcset-mem*)

lemma *mk-coeff-Cons*:

assumes c : $\{0..<length\ (v\#vs)\} \to carrier\ R$
shows $mk\text{-}coeff\ (v\#vs)\ c = (\lambda w.\ (if\ w = v\ then\ c\ 0\ else\ \mathbf{0}) \oplus mk\text{-}coeff\ vs\ (c\ o\ Suc)\ w)$
proof−
from *assms* **have** $c\ o\ Suc : \{0..<length\ vs\} \to carrier\ R$ **by** *auto*
from *mk-coeff-carrier*[*OF this*] *assms*
show *?thesis* **by** (*auto simp add*: *mk-coeff-def find-indices-Cons*)
qed

lemma *mk-coeff-0*[*simp*]:
assumes $v \notin set\ vs$
shows $mk\text{-}coeff\ vs\ c\ v = \mathbf{0}$
proof −
have $(find\text{-}indices\ v\ vs) = []$ **using** *assms* **unfolding** *find-indices-def*
by (*simp add*: *in-set-conv-nth*)
thus *?thesis* **unfolding** *mk-coeff-def* **by** *auto*
qed

lemma *lincomb-list-as-lincomb*:
assumes *vs-M*: $set\ vs \subseteq carrier\ M$ **and** *c*: $c : \{0..<length\ vs\} \to carrier\ R$
shows $lincomb\text{-}list\ c\ vs = lincomb\ (mk\text{-}coeff\ vs\ c)\ (set\ vs)$
proof (*insert assms*, *induct vs arbitrary*: *c*)
case (*Cons v vs*)
have *mk-coeff-Suc-closed*: $mk\text{-}coeff\ vs\ (c \circ Suc)\ a \in carrier\ R$ **for** *a*
apply (*rule mk-coeff-carrier*)
using *Cons.prems* **unfolding** *Pi-def* **by** *auto*
have *x-in*: $x \in carrier\ M$ **if** *x*: $x \in set\ vs$ **for** *x* **using** *Cons.prems x* **by** *auto*
show *?case* **apply** (*unfold mk-coeff-Cons*[*OF Cons.prems*(*2*)] *lincomb-list-Cons*)
apply (*subst Cons*) **using** *Cons* **apply** (*force*, *force*)
proof (*cases* $v \in set\ vs$, *auto simp*:*insert-absorb*)
case *False*
let $?f = (\lambda va.\ ((if\ va = v\ then\ c\ 0\ else\ \mathbf{0}) \oplus mk\text{-}coeff\ vs\ (c \circ Suc)\ va) \odot_M va)$
have *mk-0*: $mk\text{-}coeff\ vs\ (c \circ Suc)\ v = \mathbf{0}$ **using** *False* **by** *auto*
have [*simp*]: $(c\ 0 \oplus \mathbf{0}) = c\ 0$
using *Cons.prems*(*2*) **by** *force*
have *finsum-rw*: $(\bigoplus_M va{\in}insert\ v\ (set\ vs).\ ?f\ va) = (?f\ v) \oplus_M (\bigoplus_M va{\in}(set\ vs).\ ?f\ va)$
proof (*rule finsum-insert*, *auto simp add*: *False*, *rule smult-closed*, *rule R.a-closed*)
fix *x*
show $mk\text{-}coeff\ vs\ (c \circ Suc)\ x \in carrier\ R$
using *mk-coeff-Suc-closed* **by** *auto*
show $c\ 0 \odot_M v \in carrier\ M$
proof (*rule smult-closed*)
show $c\ 0 \in carrier\ R$
using *Cons.prems*(*2*) **by** *fastforce*
show $v \in carrier\ M$
using *Cons.prems*(*1*) **by** *auto*
qed
show $\mathbf{0} \in carrier\ R$

by *simp*
assume *x*: $x \in$ *set vs* **show** $x \in$ *carrier M*
using *Cons.prems*(*1*) *x* **by** *auto*
qed
have *finsum-rw2*:
$(\bigoplus_M va \in (set\ vs).\ ?f\ va) = (\bigoplus_M va \in set\ vs.\ (mk\text{-}coeff\ vs\ (c \circ Suc)\ va) \odot_M va)$
proof (*rule finsum-cong2*, *auto simp add*: *False*)
fix *i* **assume** *i*: $i \in$ *set vs*
have $c \circ Suc \in \{0..<length\ vs\} \to carrier\ R$ **using** *Cons.prems* **by** *auto*
then have [*simp*]: $mk\text{-}coeff\ vs\ (c \circ Suc)\ i \in carrier\ R$
using *mk-coeff-Suc-closed* **by** *auto*
have $\mathbf{0} \oplus mk\text{-}coeff\ vs\ (c \circ Suc)\ i = mk\text{-}coeff\ vs\ (c \circ Suc)\ i$ **by** (*rule R.l-zero*, *simp*)
then show $(\mathbf{0} \oplus mk\text{-}coeff\ vs\ (c \circ Suc)\ i) \odot_M i = mk\text{-}coeff\ vs\ (c \circ Suc)\ i \odot_M i$
by *auto*
show $(\mathbf{0} \oplus mk\text{-}coeff\ vs\ (c \circ Suc)\ i) \odot_M i \in carrier\ M$
using *Cons.prems*(*1*) *i* **by** *auto*
qed
show $c\ 0 \odot_M v \oplus_M lincomb\ (mk\text{-}coeff\ vs\ (c \circ Suc))\ (set\ vs) =$
$lincomb\ (\lambda a.\ (if\ a = v\ then\ c\ 0\ else\ \mathbf{0}) \oplus mk\text{-}coeff\ vs\ (c \circ Suc)\ a)\ (insert\ v\ (set\ vs))$
unfolding *lincomb-def*
unfolding *finsum-rw mk-0*
unfolding *finsum-rw2* **by** *auto*
next
case *True*
let $?f = \lambda va.\ ((if\ va = v\ then\ c\ 0\ else\ \mathbf{0}) \oplus mk\text{-}coeff\ vs\ (c \circ Suc)\ va) \odot_M va$
have *rw*: $(c\ 0 \oplus mk\text{-}coeff\ vs\ (c \circ Suc)\ v) \odot_M v$
$= (c\ 0 \odot_M v) \oplus_M (mk\text{-}coeff\ vs\ (c \circ Suc)\ v) \odot_M v$
using *Cons.prems*(*1*) *Cons.prems*(*2*) *atLeast0-lessThan-Suc-eq-insert-0*
using *mk-coeff-Suc-closed smult-l-distr* **by** *auto*
have *rw2*: $((mk\text{-}coeff\ vs\ (c \circ Suc)\ v) \odot_M v)$
$\oplus_M (\bigoplus_M va \in (set\ vs - \{v\}).\ ?f\ va) = (\bigoplus_M v \in set\ vs.\ mk\text{-}coeff\ vs\ (c \circ Suc)\ v \odot_M v)$
proof −
have $(\bigoplus_M va \in (set\ vs - \{v\}).\ ?f\ va) = (\bigoplus_M v \in set\ vs - \{v\}.\ mk\text{-}coeff\ vs\ (c \circ Suc)\ v \odot_M v)$
by (*rule finsum-cong2*, *unfold Pi-def*, *auto simp add*: *mk-coeff-Suc-closed x-in*)
moreover have $(\bigoplus_M v \in set\ vs.\ mk\text{-}coeff\ vs\ (c \circ Suc)\ v \odot_M v) = ((mk\text{-}coeff\ vs\ (c \circ Suc)\ v) \odot_M v)$
$\oplus_M (\bigoplus_M v \in set\ vs - \{v\}.\ mk\text{-}coeff\ vs\ (c \circ Suc)\ v \odot_M v)$
by (*rule M.add.finprod-split*, *auto simp add*: *mk-coeff-Suc-closed True x-in*)
ultimately show *?thesis* **by** *auto*
qed
have $lincomb\ (\lambda a.\ (if\ a = v\ then\ c\ 0\ else\ \mathbf{0}) \oplus mk\text{-}coeff\ vs\ (c \circ Suc)\ a)\ (set\ vs)$

```
      = (⨁Mva∈set vs. ?f va) unfolding lincomb-def ..
    also have ... = ?f v ⊕M (⨁Mva∈(set vs − {v}). ?f va)
    proof (rule M.add.finprod-split)
      have c0-mkcoeff-in: c 0 ⊕ mk-coeff vs (c ∘ Suc) v ∈ carrier R
      proof (rule R.a-closed)
        show c 0 ∈ carrier R  using Cons.prems by auto
        show mk-coeff vs (c ∘ Suc) v ∈ carrier R
          using mk-coeff-Suc-closed by auto
    qed
    moreover have (0 ⊕ mk-coeff vs (c ∘ Suc) va) ⊙M va ∈ carrier M
      if va: va ∈ carrier M for va
     by (rule smult-closed[OF - va], rule R.a-closed, auto simp add: mk-coeff-Suc-closed)
    ultimately show ?f ` set vs ⊆ carrier M using Cons.prems(1) by auto
      show finite (set vs) by simp
      show v ∈ set vs using True by simp
    qed
    also have ... = (c 0 ⊕ mk-coeff vs (c ∘ Suc) v) ⊙M v
      ⊕M (⨁Mva∈(set vs − {v}). ?f va) by auto
    also have ... = ((c 0 ⊙M v) ⊕M (mk-coeff vs (c ∘ Suc) v) ⊙M v)
      ⊕M (⨁Mva∈(set vs − {v}). ?f va) unfolding rw by simp
    also have ... = (c 0 ⊙M v) ⊕M (((mk-coeff vs (c ∘ Suc) v) ⊙M v)
      ⊕M (⨁Mva∈(set vs − {v}). ?f va))
    proof (rule M.a-assoc)
      show c 0 ⊙M v ∈ carrier M
        using Cons.prems(1) Cons.prems(2) by auto
      show mk-coeff vs (c ∘ Suc) v ⊙M v ∈ carrier M
        using Cons.prems(1) mk-coeff-Suc-closed by auto
      show (⨁Mva∈set vs − {v}. ((if va = v then c 0 else 0)
        ⊕ mk-coeff vs (c ∘ Suc) va) ⊙M va) ∈ carrier M
        by (rule M.add.finprod-closed) (auto simp add: mk-coeff-Suc-closed x-in)
    qed
    also have ... = c 0 ⊙M v ⊕M (⨁Mv∈set vs. mk-coeff vs (c ∘ Suc) v ⊙M v)
      unfolding rw2 ..
    also have ... = c 0 ⊙M v ⊕M lincomb (mk-coeff vs (c ∘ Suc)) (set vs)
      unfolding lincomb-def ..
    finally show c 0 ⊙M v ⊕M lincomb (mk-coeff vs (c ∘ Suc)) (set vs)
      = lincomb (λa. (if a = v then c 0 else 0) ⊕ mk-coeff vs (c ∘ Suc) a) (set vs)
..
  qed
qed simp

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
  using assms by (auto simp: span-list-def)

lemma in-span-listE:
```

assumes $v \in$ *span-list vs*
and $\bigwedge c.\ c : \{0..<length\ vs\} \to carrier\ R \Longrightarrow v = lincomb\text{-}list\ c\ vs \Longrightarrow thesis$
shows *thesis*
using *assms* **by** (*auto simp*: *span-list-def*)

lemmas *lincomb-insert2* = *lincomb-insert*[*unfolded insert-union*[*symmetric*]]

lemma *lincomb-zero*:
assumes *U*: $U \subseteq carrier\ M$ **and** *a*: $a : U \to \{zero\ R\}$
shows *lincomb a U = zero M*
using *U a*
proof (*induct U rule*: *infinite-finite-induct*)
case *empty* **show** *?case* **unfolding** *lincomb-def* **by** *auto* **next**
case (*insert u U*)
hence $a \in insert\ u\ U \to carrier\ R$ **using** *zero-closed* **by** *force*
thus *?case* **using** *insert* **by** (*subst lincomb-insert2*; *auto*)
qed (*auto simp*: *lincomb-def*)

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 −
let *?r* = *class-ring* :: *'a ring*
show *semiring ?r*
by (*unfold-locales*, *auto simp*: *field-simps*)
then interpret *semiring ?r* **.**
{
fix *x y*
have $x\ [\widehat{\ }]_{?r}\ y = x\ \widehat{\ }\ y$
by (*induct y*, *auto simp*: *power-commutes*)
}
thus $([\widehat{\ }]_{?r})$ = (^) **by** (*intro ext*)
{
fix *f* **and** *A* :: *'b set*
have *finsum ?r f A* = *sum f A*
by (*induct A rule*: *infinite-finite-induct*, *auto*)
}
thus *finsum ?r* = *sum* **by** (*intro ext*)
qed *auto*

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 −
let *?r* = *class-ring* :: *'a ring*
interpret *semiring ?r* **..**
show *finsum ?r* = *sum pow ?r* = (^) **by** (*simp-all add*: *class-ring-simps*)
{
fix *x* :: *'a*
have $\exists\,y.\ x + y = 0$ **by** (*rule exI*[*of* - $-x$], *auto*)
} note [*simp*] = *this*
show *ring ?r*
by (*unfold-locales*, *auto simp*: *field-simps Units-def*)
then interpret *ring ?r* **.**
{
fix *x* :: *'a*
have $\ominus_{?r}\ x = -\ x$ **unfolding** *a-inv-def m-inv-def*

by (*rule the1-equality*, *rule ex1I*[*of* - $-$ *x*], *auto simp*: *minus-unique*)
} **note** *ainv* = *this*
thus *inv*: *a-inv ?r* = *uminus* **by** (*intro ext*)
{
fix *x y* :: *'a*
have $x \ominus_{?r} y = x - y$
apply (*subst a-minus-def*)
using *inv* **by** *auto*
}
thus $(\lambda x\ y.\ x \ominus_{?r} y)$ = *minus* **by** (*intro ext*)
qed (*auto simp*: *class-ring-simps*)

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 −
let *?r* = *class-ring* :: *'a ring*
interpret *ring ?r* **..**
show *cring ?r*
by (*unfold-locales*, *auto*)
then interpret *cring ?r* **.**
show *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* **by** (*simp-all add*: *class-ring-simps*)
{
fix *f* **and** *A* :: *'b set*
have *finprod ?r f A* = *prod f A*
by (*induct A rule*: *infinite-finite-induct*, *auto*)
}
thus *finprod ?r* = *prod* **by** (*intro ext*)
qed (*auto simp*: *class-ring-simps*)

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 −
interpret *cring ?r* **..**
{
fix *x* :: *'a*

have $x \neq 0 \Longrightarrow \exists xa.\ xa * x = 1 \wedge x * xa = 1$
by (*intro exI*[*of - inverse x*], *auto*)
} **note** [*simp*] = *this*
show *field ?r*
by (*unfold-locales*, *auto simp*: *Units-def*)
qed

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* = (^)
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 −
let *?r* = *class-ring* :: *'a ring*
show *field ?r* **using** *class-field*.
then interpret *field ?r*.
show *a-inv ?r* = *uminus*
and *a-minus ?r* = *minus*
and *pow ?r* = (^)
and *finsum ?r* = *sum*
and *finprod ?r* = *prod* **by** (*fact class-ring-simps*)+
show $inv_{?r}$ *x* = (*if x* = *0 then div0 else inverse x*)
proof (*cases x* = *0*)
case *True*
thus *?thesis* **unfolding** *div0-def* **by** *simp*
next
case *False*
thus *?thesis* **unfolding** *m-inv-def*
by (*intro the1-equality ex1I*[*of - inverse x*], *auto simp*: *inverse-unique*)
qed
qed (*auto simp*: *class-ring-simps*)

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** $R \equiv$ *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*
  using *ring-mat* **by** (*auto simp*: *ring-mat-simps*)

end

lemma *matrix-vs*: *vectorspace* (*class-ring* :: *'a* :: *field ring*) (*module-mat TYPE('a) nr nc*)
proof −
  interpret *abelian-group module-mat TYPE('a) nr nc*
    by (*rule abelian-group-mat*)
  show *?thesis* **unfolding** *class-field-def*
    by (*unfold-locales*, *unfold matrix-vs-simps*,
      auto simp: *add-smult-distrib-left-mat add-smult-distrib-right-mat*)
qed

locale *vec-module* =
  fixes *f-ty*::*'a*::*comm-ring-1 itself*
  and *n*::*nat*
begin

abbreviation *V* **where** $V \equiv$ *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* = ($\cdot_v$)
    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*) = (−)
    and *pow* (*class-ring* :: *'a ring*) = (^)
    and *finsum* (*class-ring* :: *'a ring*) = *sum*
    and *finprod* (*class-ring* :: *'a ring*) = *prod*
    and $\bigwedge X.\ X \subseteq UNIV = True$

and $\bigwedge x.\ x \in UNIV = True$
and $\bigwedge a\ A.\ a \in A \to UNIV \equiv True$
and $\bigwedge P.\ P \wedge True \equiv P$
and $\bigwedge P.\ (True \Longrightarrow P) \equiv Trueprop\ P$
apply *unfold-locales*
apply (*auto simp*: *module-vec-simps class-ring-simps Units-def add-smult-distrib-vec*

smult-add-distrib-vec intro!:*bexI*[*of* - − -])
done

lemma *finsum-index*:
assumes *i*: $i < n$
and *f*: $f \in A \to carrier\text{-}vec\ n$
and *A*: $A \subseteq carrier\text{-}vec\ n$
shows *finsum V f A* $\$\ i = sum\ (\lambda x.\ f\ x\ \$\ i)\ A$
using *A f*
proof (*induct A rule*: *infinite-finite-induct*)
case *empty*
then show *?case* **using** *i* **by** *simp* **next**
case (*insert x X*)
then have *Xf*: *finite X*
and *xX*: $x \notin X$
and *x*: $x \in carrier\text{-}vec\ n$
and *X*: $X \subseteq carrier\text{-}vec\ n$
and *fx*: $f\ x \in carrier\text{-}vec\ n$
and *f*: $f \in X \to carrier\text{-}vec\ n$ **by** *auto*
have *i2*: $i < dim\text{-}vec\ (finsum\ V\ f\ X)$
using *i finsum-closed*[*OF f*] **by** *auto*
have *ix*: $i < dim\text{-}vec\ x$ **using** *x i* **by** *auto*
show *?case*
unfolding *finsum-insert*[*OF Xf xX f fx*]
unfolding *sum.insert*[*OF Xf xX*]
unfolding *index-add-vec*(*1*)[*OF i2*]
using *insert lincomb-def*
by *auto*
qed (*insert i*, *auto*)

lemma *mat-of-rows-mult-as-finsum*:
assumes $v \in carrier\text{-}vec\ (length\ lst)\ \bigwedge\ i.\ i < length\ lst \Longrightarrow lst\ !\ i \in carrier\text{-}vec$
n
defines $f\ l \equiv sum\ (\lambda\ i.\ if\ l = lst\ !\ i\ then\ v\ \$\ i\ else\ 0)\ \{0..<length\ lst\}$
shows *mat-of-cols-mult-as-finsum*:$mat\text{-}of\text{-}cols\ n\ lst *_v v = lincomb\ f\ (set\ lst)$
proof −
from *assms* **have** $\forall\ i < length\ lst.\ lst\ !\ i \in carrier\text{-}vec\ n$ **by** *blast*
note *an* = *all-nth-imp-all-set*[*OF this*] **hence** *slc*:$set\ lst \subseteq carrier\text{-}vec\ n$ **by** *auto*
hence *dn* [*simp*]:$\bigwedge\ x.\ x \in set\ lst \Longrightarrow dim\text{-}vec\ x = n$ **by** *auto*
have *dl* [*simp*]:$dim\text{-}vec\ (lincomb\ f\ (set\ lst)) = n$ **using** *an*
by (*simp add*: *slc*)

show *?thesis* **proof**
show *dim-vec* (*mat-of-cols n lst* $*_v$ *v*) = *dim-vec* (*lincomb f* (*set lst*)) **using** *assms*(*1*,*2*) **by** *auto*
fix *i* **assume** *i*:$i <$ *dim-vec* (*lincomb f* (*set lst*)) **hence** *i'*:$i < n$ **by** *auto*
with *an* **have** *fcarr*:($\lambda v.$ *f v* $\cdot_v$ *v*) $\in$ *set lst* $\rightarrow$ *carrier-vec n* **by** *auto*
from *i'* **have** (*mat-of-cols n lst* $*_v$ *v*) \$ *i* = *row* (*mat-of-cols n lst*) *i* $\cdot$ *v* **by** *auto*
also have ... = ($\sum$ *ia* = *0*..<*dim-vec v*. *lst* ! *ia* \$ *i* $*$ *v* \$ *ia*)
unfolding *mat-of-cols-def row-def scalar-prod-def*
apply(*rule sum.cong*[*OF refl*]) **using** *i an assms*(*1*) **by** *auto*
also have ... = ($\sum$ *ia* = *0*..<*length lst*. *lst* ! *ia* \$ *i* $*$ *v* \$ *ia*) **using** *assms*(*1*) **by** *auto*
also have ... = ($\sum$ *x*$\in$*set lst*. *f x* $*$ *x* \$ *i*)
unfolding *f-def sum-distrib-right* **apply** (*subst sum.swap*)
apply(*rule sum.cong*[*OF refl*])
unfolding *if-distrib if-distribR mult-zero-left sum.delta*[*OF finite-set*] **by** *auto*
also have ... = ($\sum$ *x*$\in$*set lst*. (*f x* $\cdot_v$ *x*) \$ *i*)
apply(*rule sum.cong*[*OF refl*],*subst index-smult-vec*) **using** *i slc* **by** *auto*
also have ... = ($\bigoplus_V$*v*$\in$*set lst*. *f v* $\cdot_v$ *v*) \$ *i*
unfolding *finsum-index*[*OF i' fcarr slc*] **by** *auto*
finally show (*mat-of-cols n lst* $*_v$ *v*) \$ *i* = *lincomb f* (*set lst*) \$ *i*
by (*auto simp*:*lincomb-def*)
qed
qed

lemma *lincomb-as-lincomb-list*:
fixes *ws f*
assumes *s*: *set ws* $\subseteq$ *carrier-vec n*
shows *lincomb f* (*set ws*) = *lincomb-list* ($\lambda i.$ *if* $\exists j<i.$ *ws*!*i* = *ws*!*j then 0 else f* (*ws* ! *i*)) *ws*
using *assms*
proof (*induct ws rule*: *rev-induct*)
case (*snoc a ws*)
let *?f* = $\lambda i.$ *if* $\exists j<i.$ *ws* ! *i* = *ws* ! *j then 0 else f* (*ws* ! *i*)
let *?g* = $\lambda i.$ (*if* $\exists j<i.$ (*ws* @ [*a*]) ! *i* = (*ws* @ [*a*]) ! *j then 0 else f* ((*ws* @ [*a*]) ! *i*)) $\cdot_v$ (*ws* @ [*a*]) ! *i*
let *?g2*= ($\lambda i.$ (*if* $\exists j<i.$ *ws* ! *i* = *ws* ! *j then 0 else f* (*ws* ! *i*)) $\cdot_v$ *ws* ! *i*)
have [*simp*]: $\bigwedge v.$ $v \in$ *set ws* $\Longrightarrow$ $v \in$ *carrier-vec n* **using** *snoc.prems*(*1*) **by** *auto*
then have *ws*: *set ws* $\subseteq$ *carrier-vec n* **by** *auto*
have *hyp*: *lincomb f* (*set ws*) = *lincomb-list ?f ws*
by (*intro snoc.hyps ws*)
show *?case*
proof (*cases a*$\in$*set ws*)
case *True*
have *g-length*: *?g* (*length ws*) = 0_v *n* **using** *True*
by (*auto*, *metis in-set-conv-nth nth-append*)
have (*map ?g* [*0*..<*length* (*ws* @ [*a*])]) = (*map ?g* [*0*..<*length ws*]) @ [*?g* (*length ws*)]
by *auto*
also have ... = (*map ?g* [*0*..<*length ws*]) @ [0_v *n*] **using** *g-length* **by** *simp*

finally have *map-rw*: (*map ?g* [*0..<length* (*ws* @ [*a*])]) = (*map ?g* [*0..<length ws*]) @ [0_v *n*] **.**
have *M.sumlist* (*map ?g2* [*0..<length ws*]) = *M.sumlist* (*map ?g* [*0..<length ws*])
by (*rule arg-cong*[*of - - M.sumlist*], *intro nth-equalityI*, *auto simp add*: *nth-append*)
also have ... = *M.sumlist* (*map ?g* [*0..<length ws*]) + 0_v *n*
by (*metis M.r-zero calculation hyp lincomb-closed lincomb-list-def ws*)
also have ... = *M.sumlist* (*map ?g* [*0..<length ws*] @ [0_v *n*])
by (*rule M.sumlist-snoc*[*symmetric*], *auto simp add*: *nth-append*)
finally have *summlist-rw*: *M.sumlist* (*map ?g2* [*0..<length ws*])
= *M.sumlist* (*map ?g* [*0..<length ws*] @ [0_v *n*]) **.**
have *lincomb f* (*set* (*ws* @ [*a*])) = *lincomb f* (*set ws*) **using** *True* **unfolding** *lincomb-def*
by (*simp add*: *insert-absorb*)
thus *?thesis*
unfolding *hyp lincomb-list-def map-rw summlist-rw*
by *auto*
next
case *False*
have *g-length*: *?g* (*length ws*) = *f a* $\cdot_v$ *a* **using** *False* **by** (*auto simp add*: *nth-append*)
have (*map ?g* [*0..<length* (*ws* @ [*a*])]) = (*map ?g* [*0..<length ws*]) @ [*?g* (*length ws*)]
by *auto*
also have ... = (*map ?g* [*0..<length ws*]) @ [(*f a* $\cdot_v$ *a*)] **using** *g-length* **by** *simp*
finally have *map-rw*: (*map ?g* [*0..<length* (*ws* @ [*a*])]) = (*map ?g* [*0..<length ws*]) @ [(*f a* $\cdot_v$ *a*)] **.**
have *summlist-rw*: *M.sumlist* (*map ?g2* [*0..<length ws*]) = *M.sumlist* (*map ?g* [*0..<length ws*])
by (*rule arg-cong*[*of - - M.sumlist*], *intro nth-equalityI*, *auto simp add*: *nth-append*)
have *lincomb f* (*set* (*ws* @ [*a*])) = *lincomb f* (*set* (*a* # *ws*)) **by** *auto*
also have ... = ($\bigoplus_V$*v*∈*set* (*a* # *ws*). *f v* $\cdot_v$ *v*) **unfolding** *lincomb-def* **..**
also have ... = ($\bigoplus_V$*v*∈ *insert a* (*set ws*). *f v* $\cdot_v$ *v*) **by** *simp*
also have ... = (*f a* $\cdot_v$ *a*) + ($\bigoplus_V$*v*∈ (*set ws*). *f v* $\cdot_v$ *v*)
proof (*rule finsum-insert*)
show *finite* (*set ws*) **by** *auto*
show *a* ∉ *set ws* **using** *False* **by** *auto*
show (λ*v*. *f v* $\cdot_v$ *v*) ∈ *set ws* → *carrier-vec n*
using *snoc.prems*(*1*) **by** *auto*
show *f a* $\cdot_v$ *a* ∈ *carrier-vec n* **using** *snoc.prems* **by** *auto*
qed
also have ... = (*f a* $\cdot_v$ *a*) + *lincomb f* (*set ws*) **unfolding** *lincomb-def* **..**
also have ... = (*f a* $\cdot_v$ *a*) + *lincomb-list ?f ws* **using** *hyp* **by** *auto*
also have ... = *lincomb-list ?f ws* + (*f a* $\cdot_v$ *a*)
using *M.add.m-comm lincomb-list-carrier snoc.prems* **by** *auto*
also have ... = *lincomb-list* (λ*i*. *if* ∃*j*<*i*. (*ws* @ [*a*]) ! *i*
= (*ws* @ [*a*]) ! *j then 0 else f* ((*ws* @ [*a*]) ! *i*)) (*ws* @ [*a*])

proof (*unfold lincomb-list-def map-rw summlist-rw*, *rule M.sumlist-snoc*[*symmetric*])
show *set* (*map ?g* [*0..<length ws*]) ⊆ *carrier-vec n* **using** *snoc.prems*
by (*auto simp add*: *nth-append*)
show *f a* $\cdot_v$ *a* ∈ *carrier-vec n*
using *snoc.prems* **by** *auto*
qed
finally show *?thesis* **.**
qed
qed *auto*
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* = ($\cdot_m$)
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*)
by (*rule matrix-vs*, *auto simp*: *matrix-vs-simps class-field-def*)
end

lemma *vec-module*: *module* (*class-ring* :: *'a* :: *field ring*) (*module-vec TYPE*(*'a*) *n*)
proof −
interpret *abelian-group module-vec TYPE*(*'a*) *n*
apply (*unfold-locales*)
unfolding *module-vec-def Units-def*
using *add-inv-exists-vec* **by** *auto*
show *?thesis*
unfolding *class-field-def*

apply (*unfold-locales*)
unfolding *class-ring-simps*
unfolding *module-vec-simps*
using *add-smult-distrib-vec*
by (*auto simp*: *smult-add-distrib-vec*)
qed

lemma *vec-vs*: *vectorspace* (*class-ring* :: *'a* :: *field ring*) (*module-vec* $TYPE('a)$ *n*)
unfolding *vectorspace-def*
using *vec-module class-field*
by (*auto simp*: *class-field-def*)

locale *vec-space* =
fixes *f-ty*::*'a*::*field itself*
and *n*::*nat*
begin

sublocale *vec-module f-ty n*.

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*)
using *vec-vs*
unfolding *class-field-def*
by (*auto simp*: *module-vec-simps class-ring-simps*)

lemma *finsum-vec*[*simp*]: *finsum-vec* $TYPE('a)$ *n* = *finsum V*
by (*force simp*: *finsum-vec-def monoid-vec-def finsum-def finprod-def*)

lemma *finsum-scalar-prod-sum*:
assumes *f*: *f* : *U* → *carrier-vec n*
and *w*: *w*: *carrier-vec n*
shows *finsum V f U* · *w* = *sum* (λ*u*. *f u* · *w*) *U*
using *w f*
proof (*induct U rule*: *infinite-finite-induct*)
case (*insert u U*)

```
    hence f: f : U → carrier-vec n f u : carrier-vec n  by auto
    show ?case
      unfolding finsum-insert[OF insert(1) insert(2) f]
      apply (subst add-scalar-prod-distrib) using insert by auto
qed auto

lemma vec-neg[simp]: assumes x : carrier-vec n shows ⊖V x = − x
  unfolding a-inv-def m-inv-def apply simp
  apply (rule the-equality, intro conjI)
  using assms apply auto
  using M.minus-unique uminus-carrier-vec uminus-r-inv-vec by blast

lemma finsum-dim:
  finite A ⟹ f ∈ A → carrier-vec n ⟹ dim-vec (finsum V f A) = n
proof(induct set:finite)
  case (insert a A)
    hence dfa: dim-vec (f a) = n by auto
    have f: f ∈ A → carrier-vec n using insert by auto
    hence fa: f a ∈ carrier-vec n using insert by auto
    show ?case
      unfolding finsum-insert[OF insert(1) insert(2) f fa]
      using insert by auto
qed simp

lemma lincomb-dim:
  assumes fin: finite X
    and X: X ⊆ carrier-vec n
  shows dim-vec (lincomb a X) = n
proof −
  let ?f = λv. a v ·v v
  have f: ?f ∈ X → carrier-vec n apply rule using X by auto
  show ?thesis
    unfolding lincomb-def
    using finsum-dim[OF fin f].
qed


lemma lincomb-index:
  assumes i: i < n
    and X: X ⊆ carrier-vec n
  shows lincomb a X $ i = sum (λx. a x ∗ x $ i) X
proof −
  let ?f = λx. a x ·v x
  have f: ?f : X → carrier-vec n using X by auto
  have point: ⋀v. v ∈ X ⟹ (a v ·v v) $ i = a v ∗ v $ i using i X by auto
  show ?thesis
    unfolding lincomb-def
    unfolding finsum-index[OF i f X]
    using point X by simp
```

```
qed

lemma append-insert: set (xs @ [x]) = insert x (set xs) by simp

lemma lincomb-units:
  assumes i: i < n
  shows lincomb a (set (unit-vecs n)) $ i = a (unit-vec n i)
  unfolding lincomb-index[OF i unit-vecs-carrier]
  unfolding unit-vecs-first
proof -
  let ?f = λm i. ∑x∈set (unit-vecs-first n m). a x ∗ x $ i
  have zero:⋀m j. m ≤ j ⟹ j < n ⟹ ?f m j = 0
  proof -
    fix m
    show ⋀j. m ≤ j ⟹ j < n ⟹ ?f m j = 0
    proof (induction m)
      case (Suc m)
        hence mj:m≤j and mj':m≠j and jn:j<n and mn:m<n by auto
        hence mem: unit-vec n m ∉ set (unit-vecs-first n m)
          apply(subst unit-vecs-first-distinct) by auto
        show ?case
          unfolding unit-vecs-first.simps
          unfolding append-insert
          unfolding sum.insert[OF finite-set mem]
          unfolding index-unit-vec(1)[OF mn jn]
          unfolding Suc(1)[OF mj jn] using mj' by simp
    qed simp
  qed
  { fix m
    have i < m ⟹ m ≤ n ⟹ ?f m i = a (unit-vec n i)
    proof (induction m arbitrary: i)
      case (Suc m)
        hence iSm: i < Suc m and i:i<n and mn: m<n by auto
        hence mem: unit-vec n m ∉ set (unit-vecs-first n m)
          apply(subst unit-vecs-first-distinct) by auto
        show ?case
          unfolding unit-vecs-first.simps
          unfolding append-insert
          unfolding sum.insert[OF finite-set mem]
          unfolding index-unit-vec(1)[OF mn i]
          using zero Suc by (cases i = m,auto)
    qed auto
  }
  thus ?f n i = a (unit-vec n i) using assms by auto
qed

lemma lincomb-coordinates:
  assumes v: v : carrier-vec n
  defines a ≡ (λu. v $ (THE i. u = unit-vec n i))
```

shows *lincomb a* (*set* (*unit-vecs n*)) = *v*
proof −
have *a*: *a* ∈ *set* (*unit-vecs n*) → *UNIV* **by** *auto*
have *fvu*: $\bigwedge i.\ i < n \Longrightarrow v \$ i = a\ (unit\text{-}vec\ n\ i)$
unfolding *a-def* **using** *unit-vec-eq* **by** *auto*
show *?thesis*
apply *rule*
unfolding *lincomb-dim*[*OF finite-set unit-vecs-carrier*]
using *v lincomb-units fvu*
by *auto*
qed

lemma *span-unit-vecs-is-carrier*: *span* (*set* (*unit-vecs n*)) = *carrier-vec n* (**is** *?L* = *?R*)
proof (*rule*;*rule*)
fix *v* **assume** *vsU*: *v* ∈ *?L* **show** *v* ∈ *?R*
proof −
obtain *a*
where *v*: *v* = *lincomb a* (*set* (*unit-vecs n*))
using *vsU*
unfolding *finite-span*[*OF finite-set unit-vecs-carrier*] **by** *auto*
thus *?thesis* **using** *lincomb-closed*[*OF unit-vecs-carrier*] **by** *auto*
qed
next fix *v*::′*a vec* **assume** *v*: *v* ∈ *?R* **show** *v* ∈ *?L*
unfolding *span-def*
using *lincomb-coordinates*[*OF v*,*symmetric*] **by** *auto*
qed

lemma *fin-dim*[*simp*]: *fin-dim*
unfolding *fin-dim-def*
apply (*intro eqTrueI exI conjI*) **using** *span-unit-vecs-is-carrier unit-vecs-carrier*
by *auto*

lemma *unit-vecs-basis*: *basis* (*set* (*unit-vecs n*)) **unfolding** *basis-def span-unit-vecs-is-carrier*
proof (*intro conjI*)
show ¬ *lin-dep* (*set* (*unit-vecs n*))
proof
assume *lin-dep* (*set* (*unit-vecs n*))
from *this*[*unfolded lin-dep-def*] **obtain** *A a v* **where**
fin: *finite A* **and** *A*: *A* ⊆ *set* (*unit-vecs n*)
and *lc*: *lincomb a A* = 0_v *n* **and** *v*: *v* ∈ *A* **and** *av*: *a v* ≠ *0*
by *auto*
from *v A* **obtain** *i* **where** *i*: *i* < *n* **and** *vu*: *v* = *unit-vec n i* **unfolding** *unit-vecs-def* **by** *auto*
define *b* **where** *b* = (λ *x*. *if x* ∈ *A then a x else 0*)
have *id*: *A* ∪ (*set* (*unit-vecs n*) − *A*) = *set* (*unit-vecs n*) **using** *A* **by** *auto*
from *lincomb-index*[*OF i unit-vecs-carrier*]
have *lincomb b* (*set* (*unit-vecs n*)) $ *i* = ($\sum$ *x*∈ (*A* ∪ (*set* (*unit-vecs n*) − *A*)). *b x* ∗ *x* $ *i*)

```
    unfolding id .
  also have ... = (∑x∈ A. b x * x $ i) + (∑x∈ set (unit-vecs n) − A. b x *
x $ i)
    by (rule sum.union-disjoint, insert fin, auto)
  also have (∑x∈ A. b x * x $ i) = (∑x∈ A. a x * x $ i)
    by (rule sum.cong, auto simp: b-def)
  also have ... = lincomb a A $ i
    by (subst lincomb-index[OF i], insert A unit-vecs-carrier, auto)
  also have ... = 0 unfolding lc using i by simp
  also have (∑x∈ set (unit-vecs n) − A. b x * x $ i) = 0
    by (rule sum.neutral, auto simp: b-def)
  finally have lincomb b (set (unit-vecs n)) $ i = 0 by simp
  from lincomb-units[OF i, of b, unfolded this]
  have b v = 0 unfolding vu by simp
  with v av show False unfolding b-def by simp
 qed
qed (insert unit-vecs-carrier, auto)

lemma unit-vecs-length[simp]: length (unit-vecs n) = n
 unfolding unit-vecs-def by auto

lemma unit-vecs-distinct: distinct (unit-vecs n)
 unfolding distinct-conv-nth unit-vecs-length
proof (intro allI impI)
 fix i j
 assume *: i < n j < n i ≠ j
 show unit-vecs n ! i ≠ unit-vecs n ! j
 proof
  assume unit-vecs n ! i = unit-vecs n ! j
  from arg-cong[OF this, of λ v. v $ i]
  show False using * unfolding unit-vecs-def by auto
 qed
qed

lemma dim-is-n: dim = n
 unfolding dim-basis[OF finite-set unit-vecs-basis]
 unfolding distinct-card[OF unit-vecs-distinct]
 by simp

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 −
  have *vectorspace class-ring* (*span-vs A*)
   using *assms span-is-subspace subspace-def subspace-is-vs* **by** *simp*
  have *A* ⊆ *span A* **using** *assms in-own-span* **by** *simp*
  have *submodule class-ring* (*span A*) *V* **using** *assms span-is-submodule* **by** *simp*
  have *LinearCombinations.module.span class-ring* (*vs* (*span A*)) *A* = *carrier* (*vs* (*span A*))
    using *span-li-not-depend*(*1*)[*OF* ‹*A* ⊆ *span A*› ‹*submodule class-ring* (*span A*) *V*›] **by** *auto*
 then show *?thesis* **unfolding** *vectorspace.fin-dim-def*[*OF* ‹*vectorspace class-ring* (*span-vs A*)›]
    using *List.finite-set* ‹*A* ⊆ *span A*› ‹*vectorspace class-ring* (*vs* (*span A*))›
      vec-vs vectorspace.carrier-vs-is-self[*OF* ‹*vectorspace class-ring* (*span-vs A*)›]
using *assms*(*1*) **by** *auto*
qed

lemma *fin-dim-span-cols*:
assumes *A* ∈ *carrier-mat n nc*
shows *vectorspace.fin-dim class-ring* (*vs* (*span* (*set* (*cols A*))))
 using *fin-dim-span cols-dim List.finite-set assms carrier-matD*(*1*) *module-vec-simps*(*3*)
by *force*
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 (*insert assms*, *induct ws arbitrary*: *c*)
  case *Nil*
  then show *?case* **by** (*auto simp*: *mult-mat-vec-def scalar-prod-def*)
next
  case (*Cons w ws*)
  { fix *i* **assume** *i*: *i* < *n*
    have *?l* (*w*#*ws*) *c* = *c 0* $\cdot_v$ *w* + *mat-of-cols n ws* $*_v$ *vec* (*length ws*) (*c* ∘ *Suc*)
      by (*simp add*: *Cons o-def*)
    also have . . . $ *i* = *?r* (*w*#*ws*) *c* $ *i*
      using *Cons i index-smult-vec*
        by (*simp add*: *mat-of-cols-Cons-index-0 mat-of-cols-Cons-index-Suc o-def vec-Suc mult-mat-vec-def row-def length-Cons*)
    finally have *?l* (*w*#*ws*) *c* $ *i* = ...
  }
  with *Cons* **show** *?case* **by** (*intro eq-vecI*, *auto*)

qed

lemma *lincomb-vec-diff-add*:
  assumes *A*: $A \subseteq$ *carrier-vec n*
  and *BA*: $B \subseteq A$ **and** *fin-A*: *finite A*
  and *f*: $f \in A \to UNIV$ **shows** *lincomb f A* = *lincomb f* $(A-B)$ + *lincomb f B*
proof −
 have $A - B \cup B = A$ **using** *BA* **by** *auto*
 hence *lincomb f A* = *lincomb f* $(A - B \cup B)$ **by** *simp*
 also have ... = *lincomb f* $(A-B)$ + *lincomb f B*
  by (*rule lincomb-union*, *insert assms*, *auto intro*: *finite-subset*)
 finally show *?thesis* **.**
qed

lemma *dim-sumlist*:
 assumes $\forall x \in$ *set xs*. *dim-vec x* = *n*
 shows *dim-vec* (*M.sumlist xs*) = *n* **using** *assms* **by** (*induct xs*, *auto*)

lemma *sumlist-nth*:
 assumes $\forall x \in$ *set xs*. *dim-vec x* = *n* **and** $i<n$
 shows (*M.sumlist xs*) \$ *i*= *sum* (λj. (*xs* ! *j*) \$ *i*) {*0*..<*length xs*}
 using *assms*
proof (*induct xs rule*: *rev-induct*)
 case (*snoc a xs*)
 have [*simp*]: $x \in$ *carrier-vec n* **if** *x*: $x \in$*set xs* **for** *x*
  using *snoc.prems x* **unfolding** *carrier-vec-def* **by** *auto*
 have [*simp*]: $a \in$ *carrier-vec n*
  using *snoc.prems* **unfolding** *carrier-vec-def* **by** *auto*
 have *hyp*: *M.sumlist xs* \$ *i* = ($\sum j$ = *0*..<*length xs*. *xs* ! *j* \$ *i*)
  by (*rule snoc.hyps*, *auto simp add*: *snoc.prems*)
 have *M.sumlist* (*xs* @ [*a*]) = *M.sumlist xs* + *M.sumlist* [*a*]
  by (*rule M.sumlist-append*, *auto simp add*: *snoc.prems*)
 also have ... = *M.sumlist xs* + *a* **by** *auto*
 also have ... \$ *i* = (*M.sumlist xs* \$ *i*) + (*a* \$ *i*)
  by (*rule index-add-vec*(*1*), *auto simp add*: *snoc.prems*)
 also have ... = ($\sum j$ = *0*..<*length xs*. *xs* ! *j* \$ *i*) + (*a* \$ *i*) **unfolding** *hyp* **by**
simp
 also have ... = ($\sum j$ = *0*..<*length* (*xs* @ [*a*]). (*xs* @ [*a*]) ! *j* \$ *i*)
  by (*auto*, *rule sum.cong*, *auto simp add*: *nth-append*)
 finally show *?case* **.**
qed *auto*

lemma *lincomb-as-lincomb-list-distinct*:
 assumes *s*: *set ws* $\subseteq$ *carrier-vec n* **and** *d*: *distinct ws*
 shows *lincomb f* (*set ws*) = *lincomb-list* (λi. *f* (*ws* ! *i*)) *ws*
proof (*insert assms*, *induct ws*)
 case *Nil*
 then show *?case* **by** *auto*
next

case (*Cons a ws*)
have [*simp*]: $\bigwedge$*v. v* ∈ *set ws* ⟹ *v* ∈ *carrier-vec n* **using** *Cons.prems*(*1*) **by** *auto*
then have *ws*: *set ws* ⊆ *carrier-vec n* **by** *auto*
have *hyp*: *lincomb f* (*set* (*ws*)) = *lincomb-list* (λ*i. f* (*ws* ! *i*)) *ws*
proof (*intro Cons.hyps ws*)
  show *distinct ws* **using** *Cons.prems*(*2*) **by** *auto*
qed
have (*map* (λ*i. f* (*ws* ! *i*) $\cdot_v$ *ws* ! *i*) [*0*..<*length ws*]) = (*map* (λ*v. f v* $\cdot_v$ *v*) *ws*)
  by (*intro nth-equalityI*, *auto*)
with *ws* **have** *sumlist-rw*: *sumlist* (*map* (λ*i. f* (*ws* ! *i*) $\cdot_v$ *ws* ! *i*) [*0*..<*length ws*])
  = *sumlist* (*map* (λ*v. f v* $\cdot_v$ *v*) *ws*)
  by (*subst* (*1 2*) *sumlist-as-summset*, *auto*)
have *lincomb f* (*set* (*a* # *ws*)) = ($\bigoplus_V$*v*∈*set* (*a* # *ws*). *f v* $\cdot_v$ *v*) **unfolding** *lincomb-def* **..**
also have ... = ($\bigoplus_V$*v*∈ *insert a* (*set ws*). *f v* $\cdot_v$ *v*) **by** *simp*
also have ... = (*f a* $\cdot_v$ *a*) + ($\bigoplus_V$*v*∈ (*set ws*). *f v* $\cdot_v$ *v*)
  by (*rule finsum-insert*, *insert Cons.prems*, *auto*)
also have ... = *f a* $\cdot_v$ *a* + *lincomb-list* (λ*i. f* (*ws* ! *i*)) *ws* **using** *hyp lincomb-def* **by** *auto*
also have ... = *f a* $\cdot_v$ *a* + *sumlist* (*map* (λ*v. f v* $\cdot_v$ *v*) *ws*)
  unfolding *lincomb-list-def sumlist-rw* **by** *auto*
also have ... = *sumlist* (*map* (λ*v. f v* $\cdot_v$ *v*) (*a* # *ws*))
proof −
  let *?a* = (*map* (λ*v. f v* $\cdot_v$ *v*) [*a*])
  have *a*: *a* ∈ *carrier-vec n* **using** *Cons.prems*(*1*) **by** *auto*
  have *f a* $\cdot_v$ *a* = *sumlist* (*map* (λ*v. f v* $\cdot_v$ *v*) [*a*]) **using** *Cons.prems*(*1*) **by** *auto*
  hence *f a* $\cdot_v$ *a* + *sumlist* (*map* (λ*v. f v* $\cdot_v$ *v*) *ws*)
    = *sumlist ?a* + *sumlist* (*map* (λ*v. f v* $\cdot_v$ *v*) *ws*) **by** *simp*
  also have ... = *sumlist* (*?a* @ (*map* (λ*v. f v* $\cdot_v$ *v*) *ws*))
    by (*rule sumlist-append*[*symmetric*], *auto simp add*: *a*)
  finally show *?thesis* **by** *auto*
qed
also have ... = *sumlist* (*map* (λ*i. f* ((*a* # *ws*) ! *i*) $\cdot_v$ (*a* # *ws*) ! *i*) [*0*..<*length* (*a* # *ws*)])
proof −
  have *u*: (*map* (λ*i. f* ((*a* # *ws*) ! *i*) $\cdot_v$ (*a* # *ws*) ! *i*) [*0*..<*length* (*a* # *ws*)])
      = (*map* (λ*v. f v* $\cdot_v$ *v*) (*a* # *ws*))
    by (*smt* (*verit*, *del-insts*) *length-map map-equality-iff map-nth nth-map*)
  show *?thesis* **unfolding** *u* **..**
qed
also have ... = *lincomb-list* (λ*i. f* ((*a* # *ws*) ! *i*)) (*a* # *ws*)
  unfolding *lincomb-list-def* **..**
finally show *?case* **.**
qed

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 (*cases distinct ws*)
case *False*
obtain *i j* **where** *ij*: *i*≠*j* **and** *c*: *col A i* = *col A j* **and** *i*: *i*<*n* **and** *j*: *j*<*n*
using *False A* **unfolding** *A-def*
by (*metis dimv-ws distinct-conv-nth carrier-matD*(*2*)
col-mat-of-cols mat-of-cols-carrier(*3*) *nth-mem carrier-vecI*)
show *?thesis* **by** (*rule det-identical-columns*[*OF A ij i j c*])
next
case *True*
have *d1*[*simp*]: ⋀*x*. *x* ∈ *set ws* ⟹ *x* ∈ *carrier-vec n* **using** *dimv-ws* **by** *auto*
obtain *A' f' v* **where** *f'-in*: *f'* ∈ *A'* → *UNIV*
and *lc-f'*: *lincomb f' A'* = 0_v *n* **and** *f'-v*: *f' v* ≠ *0*
and *v-A'*: *v* ∈ *A'* **and** *A'-in-rows*: *A'* ⊆ *set* (*cols A*)
using *ld-cols* **unfolding** *lin-dep-def* **by** *auto*
define *f* **where** *f* ≡ λ*x*. *if x* ∉ *A' then 0 else f' x*
have *f-in*: *f* ∈ (*set* (*cols A*)) → *UNIV* **using** *f'-in* **by** *auto*
have *A'-in-carrier*: *A'* ⊆ *carrier-vec n*
by (*metis* (*no-types*) *A'-in-rows A-def cols-dim carrier-matD*(*1*) *mat-of-cols-carrier*(*1*)
subset-trans)
have *lc-f*: *lincomb f* (*set* (*cols A*)) = 0_v *n*
proof −
have *l1*: *lincomb f* (*set* (*cols A*) − *A'*) = 0_v *n*
by (*rule lincomb-zero*, *auto simp add*: *f-def*, *insert A cols-dim*, *blast*)
have *l2*: *lincomb f A'* = 0_v *n* **using** *lc-f'* **unfolding** *f-def* **using** *A'-in-carrier*
by *auto*
have *lincomb f* (*set* (*cols A*)) = *lincomb f* (*set* (*cols A*) − *A'*) + *lincomb f A'*
proof (*rule lincomb-vec-diff-add*)
show *set* (*cols A*) ⊆ *carrier-vec n*
using *A cols-dim* **by** *blast*
show *A'* ⊆ *set* (*cols A*)
using *A'-in-rows* **by** *blast*
qed *auto*
also have ... = 0_v *n* **using** *l1 l2* **by** *auto*
finally show *?thesis* **.**
qed
have *v-in*: *v* ∈ (*set* (*cols A*)) **using** *v-A' A'-in-rows* **by** *auto*
have *fv*: *f v* ≠ *0* **using** *f'-v v-A'* **unfolding** *f-def* **by** *auto*
let *?c* = (λ*i*. *f* (*ws* ! *i*))
have *lincomb f* (*set ws*) = *lincomb-list ?c ws*
by (*rule lincomb-as-lincomb-list-distinct*[*OF - True*], *auto*)
have ∃ *v*. *v* ∈ *carrier-vec n* ∧ *v* ≠ 0_v *n* ∧ *A* $*_v$ *v* = 0_v *n*
proof (*rule exI*[*of - vec* (*length ws*) *?c*], *rule conjI*)

show *vec* (*length ws*) *?c* ∈ *carrier-vec n* **using** *A A-def* **by** *auto*
have *vec-not0*: *vec* (*length ws*) *?c* ≠ 0_v *n*
proof −
obtain *i* **where** *ws-i*: (*ws* ! *i*) = *v* **and** *i*: *i*<*length ws* **using** *v-in* **unfolding** *A-def*
by (*metis d1 cols-mat-of-cols in-set-conv-nth subset-eq*)
have *vec* (*length ws*) *?c* $ *i* = *?c i* **by** (*rule index-vec*[*OF i*])
also have ... = *f v* **using** *ws-i* **by** *simp*
also have ... ≠ *0* **using** *fv* **by** *simp*
finally show *?thesis*
using *A A-def i* **by** *fastforce*
qed
have *A* $*_v$ *vec* (*length ws*) *?c* = *mat-of-cols n ws* $*_v$ *vec* (*length ws*) *?c* **unfolding** *A-def* **..**
also have ... = *lincomb-list ?c ws* **by** (*rule lincomb-list-as-mat-mult*[*symmetric*, *OF dimv-ws*])
also have ... = *lincomb f* (*set ws*)
by (*rule lincomb-as-lincomb-list-distinct*[*symmetric*, *OF - True*], *auto*)
also have ... = 0_v *n*
using *lc-f* **unfolding** *A-def* **using** *A* **by** (*simp add*: *subset-code*(*1*))
finally show *vec* (*length ws*) (λ*i*. *f* (*ws* ! *i*)) ≠ 0_v *n* ∧ *A* $*_v$ *vec* (*length ws*) (λ*i*. *f* (*ws* ! *i*)) = 0_v *n*
using *vec-not0* **by** *fast*
qed
thus *?thesis* **unfolding** *det-0-iff-vec-prod-zero*[*OF A*] **.**
qed

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 −
have *col-rw*: (*cols* (*mat-of-cols n* (*cols A*))) = *cols A*
using *A* **by** *auto*
have *m*: *mat-of-cols n* (*cols A*) = *A* **using** *A* **by** *auto*
show *?thesis*
by (*rule A lin-dep-cols-imp-det-0'*[*of cols A*, *unfolded col-rw*, *unfolded m*, *OF - A ld*])
(*metis A cols-dim carrier-matD*(*1*) *subsetCE carrier-vecD*)
qed

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*
by (*subst det-transpose*[*OF A*, *symmetric*], *rule lin-dep-cols-imp-det-0*, *auto simp add*: *ld A*)

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*))

using *lin-dep-rows-imp-det-0*[*OF A*] *det* **by** *auto*

lemma *upper-triangular-imp-lin-indpt-rows*:
assumes *A*: $A \in$ *carrier-mat n n*
and *tri*: *upper-triangular A*
and *diag*: $0 \notin$ *set* (*diag-mat A*)
shows *lin-indpt* (*set* (*rows A*))
using *det-not-0-imp-lin-indpt-rows upper-triangular-imp-det-eq-0-iff assms*
by *auto*

lemma *span-list-as-span*:
assumes *set vs* $\subseteq$ *carrier-vec n*
shows *span-list vs* = *span* (*set vs*)
using *assms*
proof (*auto simp*: *span-list-def span-def*)
fix *f* **show** $\exists$ *a A*. *lincomb-list f vs* = *lincomb a A* $\wedge$ *finite A* $\wedge$ *A* $\subseteq$ *set vs*
using *assms lincomb-list-as-lincomb* **by** *auto*
next
fix *f*::*'a vec* $\Rightarrow$*'a* **and** *A* **assume** *fA*: *finite A* **and** *A*: *A* $\subseteq$ *set vs*
have [*simp*]: $x \in$ *carrier-vec n* **if** *x*: $x \in A$ **for** *x* **using** *A x assms* **by** *auto*
have [*simp*]: $v \in$ *carrier-vec n* **if** *v*: $v \in$ *set vs* **for** *v* **using** *assms v* **by** *auto*
have *set-vs-Un*: ((*set vs*) − *A*) ∪ *A* = *set vs* **using** *A* **by** *auto*
let *?f* = (λ*x*. *if x*∈(*set vs*) − *A then 0 else f x*)
have *f0*: ($\bigoplus_V$*v*∈(*set vs*) − *A*. *?f v* $\cdot_v$ *v*) = 0_v *n* **by** (*rule M.finsum-all0*, *auto*)
have *lincomb f A* = *lincomb ?f A*
by (*auto simp add*: *lincomb-def intro*!: *finsum-cong2*)
also have ... = ($\bigoplus_V$*v*∈(*set vs*) − *A*. *?f v* $\cdot_v$ *v*) + ($\bigoplus_V$*v*∈*A*. *?f v* $\cdot_v$ *v*)
unfolding *f0 lincomb-def* **by** *auto*
also have ... = *lincomb ?f* (((*set vs*) − *A*) ∪ *A*)
unfolding *lincomb-def*
by (*rule M.finsum-Un-disjoint*[*symmetric*], *auto simp add*: *fA*)
also have ... = *lincomb ?f* (*set vs*) **using** *set-vs-Un* **by** *auto*
finally have *lincomb f A* = *lincomb ?f* (*set vs*) **.**
with *lincomb-as-lincomb-list*[*OF assms*]
show $\exists$ *c*. *lincomb f A* = *lincomb-list c vs* **by** *auto*
qed

lemma *in-spanI*[*intro*]:
assumes *v* = *lincomb a A finite A A* $\subseteq$ *W*
shows $v \in$ *span W*
unfolding *span-def* **using** *assms* **by** *auto*
lemma *in-spanE*:
assumes $v \in$ *span W*
shows $\exists$ *a A*. *v* = *lincomb a A* $\wedge$ *finite A* $\wedge$ *A* $\subseteq$ *W*
using *assms* **unfolding** *span-def* **by** *auto*

declare *in-own-span*[*intro*]

```
lemma smult-in-span:
  assumes W ⊆ carrier-vec n and insp: x ∈ span W
  shows c ·v x ∈ span W
proof −
  from in-spanE[OF insp] obtain a A where a: x = lincomb a A finite A A ⊆ W
by blast
  have c ·v x = lincomb (λ x. c ∗ a x) A using a(1) unfolding lincomb-def a
    apply(subst finsum-smult) using assms a by (auto simp:smult-smult-assoc)
  thus c ·v x ∈ span W using a(2,3) by auto
qed

lemma span-subsetI: assumes ws: ws ⊆ carrier-vec n
  us ⊆ span ws
shows span us ⊆ span ws
  by (simp add: assms(1) span-is-submodule span-is-subset subsetI ws)

end

context vec-space begin
sublocale idom-vec.

lemma sumlist-in-span: assumes W: W ⊆ carrier-vec n
  shows (⋀x. x ∈ set xs ⟹ x ∈ span W) ⟹ sumlist xs ∈ span W
proof (induct xs)
  case Nil
  thus ?case using W by force
next
  case (Cons x xs)
  from span-is-subset2[OF W] Cons(2) have xs: x ∈ carrier-vec n set xs ⊆ car-
rier-vec n by auto
  from span-add1[OF W Cons(2)[of x] Cons(1)[OF Cons(2)]]
  have x + sumlist xs ∈ span W by auto
  also have x + sumlist xs = sumlist ([x] @ xs)
    by (subst sumlist-append, insert xs, auto)
  finally show ?case by simp
qed

lemma span-span[simp]:
  assumes W ⊆ carrier-vec n
  shows span (span W) = span W
proof(standard,standard,goal-cases)
  case (1 x) with in-spanE obtain a A where a: x = lincomb a A finite A A ⊆
span W by blast
  from a(3) assms have AC:A ⊆ carrier-vec n by auto
  show ?case unfolding a(1)[unfolded lincomb-def]
  proof(insert a(3),atomize (full),rule finite-induct[OF a(2)],goal-cases)
    case 1
    then show ?case using span-zero by auto
```

next
  case (*2 x F*)
  { assume *F*:*insert x F* ⊆ *span W*
    hence *a x* $\cdot_v$ *x* ∈ *span W* **by** (*intro smult-in-span*[*OF assms*],*auto*)
    hence *a x* $\cdot_v$ *x* + ($\bigoplus_V$*v*∈*F*. *a v* $\cdot_v$ *v*) ∈ *span W*
      using *span-add1 F 2 assms* **by** *auto*
    hence ($\bigoplus_V$*v*∈*insert x F*. *a v* $\cdot_v$ *v*) ∈ *span W*
      apply(*subst M.finsum-insert*[*OF 2*(*1*,*2*)]) **using** *F assms* **by** *auto*
  }
  then show *?case* **by** *auto*
qed
next
case *2*
show *?case* **using** *assms* **by**(*intro in-own-span*, *auto*)
qed

lemma *upper-triangular-imp-basis*:
  assumes *A*: *A* ∈ *carrier-mat n n*
    and *tri*: *upper-triangular A*
    and *diag*: *0* ∉ *set* (*diag-mat A*)
  shows *basis* (*set* (*rows A*))
  using *upper-triangular-imp-distinct*[*OF assms*]
  using *upper-triangular-imp-lin-indpt-rows*[*OF assms*] *A*
  by (*auto intro*: *dim-li-is-basis simp*: *distinct-card dim-is-n set-rows-carrier*)

lemma *fin-dim-span-rows*:
assumes *A*: *A* ∈ *carrier-mat nr n*
shows *vectorspace.fin-dim class-ring* (*vs* (*span* (*set* (*rows A*))))
proof (*rule fin-dim-span*)
  show *set* (*rows A*) ⊆ *carrier V* **using** *A rows-carrier*[*of A*] **unfolding** *carrier-mat-def* **by** *auto*
  show *finite* (*set* (*rows A*)) **by** *auto*
qed

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
  unfolding *col-space-def row-space-def cols-transpose* **..**

lemma *col-space-eq-row-space-transpose*:
  shows *col-space A* = *row-space* A^T
  unfolding *col-space-def row-space-def Matrix.rows-transpose* **..**

lemma *col-space-eq*:
  assumes *A*: *A* ∈ *carrier-mat n nc*

shows *col-space* A = {y∈*carrier-vec* (*dim-row* A). ∃ x∈*carrier-vec* (*dim-col* A). $A *_v x = y$}
proof −
let *?ws* = *cols* A
have *set-cols-in*: *set* (*cols* A) ⊆ *carrier-vec* n **using** A **unfolding** *cols-def* **by** *auto*
have *lincomb f S* ∈ *carrier-vec* (*dim-row* A) **if** *finite* S **and** S: S ⊆ *set* (*cols* A) **for** *f S*
using *lincomb-closed A*
by (*metis* (*full-types*) *S carrier-matD*(*1*) *cols-dim lincomb-closed subsetCE subsetI*)
moreover have ∃ x∈*carrier-vec* (*dim-col* A). $A *_v x$ = *lincomb f S*
if *fin-S*: *finite* S **and** S: S ⊆ *set* (*cols* A) **for** *f S*
proof −
let *?g* = (λ*v*. *if v* ∈ *S then f v else 0*)
let *?g′* = (λ*i*. *if* ∃ *j*<*i*. *?ws* ! *i* = *?ws* ! *j then 0 else ?g* (*?ws* ! *i*))
let *?Z* = *set ?ws* − *S*
have *union*: *set ?ws* = *S* ∪ *?Z* **using** *S* **by** *auto*
have *inter*: *S* ∩ *?Z* = {} **by** *auto*
have *lincomb f S* = *lincomb ?g S* **by** (*rule lincomb-cong*, *insert set-cols-in A S*, *auto*)
also have ... = *lincomb ?g* (*S* ∪ *?Z*)
by (*rule lincomb-clean*[*symmetric*],*insert set-cols-in A S fin-S*, *auto*)
also have ... = *lincomb ?g* (*set ?ws*) **using** *union* **by** *auto*
also have ... = *lincomb-list ?g′ ?ws*
by (*rule lincomb-as-lincomb-list*[*OF set-cols-in*])
also have ... = *mat-of-cols n ?ws* $*_v$ *vec* (*length ?ws*) *?g′*
by (*rule lincomb-list-as-mat-mult*, *insert set-cols-in A*, *auto*)
also have ... = $A *_v$ (*vec* (*length ?ws*) *?g′*) **using** *mat-of-cols-cols A* **by** *auto*
finally show *?thesis* **by** *auto*
qed
moreover have ∃ *f S*. $A *_v x$ = *lincomb f S* ∧ *finite S* ∧ *S* ⊆ *set* (*cols* A)
if *Ax*: $A *_v x$ ∈ *carrier-vec* (*dim-row* A) **and** *x*: x ∈ *carrier-vec* (*dim-col* A) **for** x
proof −
let *?c* = λ*i*. *x* $ *i*
have *x-vec*: *vec* (*length ?ws*) *?c* = *x* **using** *x* **by** *auto*
have $A *_v x$ = *mat-of-cols n ?ws* $*_v$ *vec* (*length ?ws*) *?c* **using** *mat-of-cols-cols A x-vec* **by** *auto*
also have ... = *lincomb-list ?c ?ws*
by (*rule lincomb-list-as-mat-mult*[*symmetric*], *insert set-cols-in A*, *auto*)
also have ... = *lincomb* (*mk-coeff ?ws ?c*) (*set ?ws*)
by (*rule lincomb-list-as-lincomb*, *insert set-cols-in A*, *auto*)
finally show *?thesis* **by** *auto*
qed
ultimately show *?thesis* **unfolding** *col-space-def span-def* **by** *auto*
qed

lemma *vector-space-row-space*:

assumes A: $A \in$ *carrier-mat nr n*
shows *vectorspace class-ring* (*vs* (*row-space* A))
proof −
have *fin*: *finite* (*set* (*rows* A)) **by** *auto*
have *s*: *set* (*rows* A) $\subseteq$ *carrier* V **using** A **unfolding** *rows-def* **by** *auto*
have *span-vs* (*set* (*rows* A)) = *vs* (*span* (*set* (*rows* A))) **by** *auto*
moreover have *vectorspace class-ring* (*span-vs* (*set* (*rows* A)))
using *fin s span-is-subspace subspace-def subspace-is-vs* **by** *simp*
ultimately show *?thesis* **unfolding** *row-space-def* **by** *auto*
qed

lemma *row-space-eq*:
assumes A: $A \in$ *carrier-mat nr n*
shows *row-space* A = {w∈*carrier-vec* (*dim-col* A). ∃ y∈*carrier-vec* (*dim-row* A). $A^T *_v y = w$}
using A *col-space-eq* **unfolding** *row-space-eq-col-space-transpose* **by** *auto*

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 −
have *At*: $A^T \in$ *carrier-mat n m* **using** A **by** *auto*
have *Pt*: $P^T \in$ *carrier-mat m m* **using** P **by** *auto*
have *PA*: $P*A \in$ *carrier-mat m n* **using** P A **by** *auto*
have $w \in$ *row-space* A **if** w: $w \in$ *row-space* ($P*A$) **for** w
proof −
have *w-carrier*: $w \in$ *carrier-vec* (*dim-col* ($P*A$))
using w *mult-carrier-mat*[*OF P A*] *row-space-eq* **by** *auto*
from *that* **and** *this* **obtain** y **where** y: $y \in$ *carrier-vec* (*dim-row* ($P * A$))
and *w-By*: $w = (P*A)^T *_v y$ **unfolding** *row-space-eq*[*OF PA*] **by** *blast*
have *ym*: $y \in$ *carrier-vec* m **using** y *Pt* **by** *auto*
have $w=((P*A)^T) *_v y$ **using** *w-By* **.**
also have ... = $(A^T * P^T) *_v y$ **using** *transpose-mult*[*OF P A*] **by** *auto*
also have ... = $A^T *_v (P^T *_v y)$ **by** (*rule assoc-mult-mat-vec*[*OF At Pt*], *insert Pt y*, *auto*)
finally show $w \in$ *row-space* A **unfolding** *row-space-eq*[*OF A*] **using** *At Pt ym* **by** *auto*
qed
moreover have $w \in$ *row-space* ($P*A$) **if** w: $w \in$ *row-space* A **for** w
proof −
have *w-carrier*: $w \in$ *carrier-vec* (*dim-col* A) **using** w A **unfolding** *row-space-eq*[*OF A*] **by** *auto*
obtain P' **where** PP': *inverts-mat* P P' **and** $P'P$: *inverts-mat* P' P
using *inv-P P* **unfolding** *invertible-mat-def* **by** *blast*
have P': $P' \in$ *carrier-mat m m* **using** PP' $P'P$ P **unfolding** *inverts-mat-def*

by (*metis carrier-matD*(*1*) *carrier-matD*(*2*) *carrier-mat-triv index-mult-mat*(*3*) *index-one-mat*(*3*))

from *that* **obtain** y **where** y: $y \in$ *carrier-vec* (*dim-row* A) **and**
w-Ay: $w = A^T *_v y$ **unfolding** *row-space-eq*[*OF* A] **by** *blast*
have *Py*: $(P'^T *_v y) \in$ *carrier-vec* m **using** P' y A **by** *auto*
have $w = A^T *_v y$ **using** *w-Ay* **.**
also have ... $= ((P' * P)*A)^T *_v y$
using $P'P$ *left-mult-one-mat* A P' **unfolding** *inverts-mat-def* **by** *auto*
also have ... $= ((P' * (P*A))^T) *_v y$ **using** *assoc-mult-mat-vec* P' P A **by** *auto*
also have ... $= ((P*A)^T * P'^T) *_v y$ **using** *transpose-mult* P A P' *mult-carrier-mat* **by** *metis*
also have ... $= (P*A)^T *_v (P'^T *_v y)$ **using** A P' *PA* y **by** *auto*
finally show $w \in$ *row-space* $(P*A)$
unfolding *row-space-eq*[*OF PA*]
using *Py* *w-carrier* A P **by** *fastforce*
qed
ultimately show *?thesis* **by** *auto*
qed

end

context *vec-module* **begin**

lemma *R-sumlist*[*simp*]: *R.sumlist* = *sum-list*
proof (*intro ext*)
fix *xs*
show *R.sumlist xs* = *sum-list xs* **by** (*induct xs*, *auto*)
qed

lemma *sumlist-dim*: **assumes** $\bigwedge x.\ x \in$ *set xs* $\Longrightarrow x \in$ *carrier-vec* n
shows *dim-vec* (*sumlist xs*) $= n$
using *sumlist-carrier assms*
by *fastforce*

lemma *sumlist-vec-index*: **assumes** $\bigwedge x.\ x \in$ *set xs* $\Longrightarrow x \in$ *carrier-vec* n
and $i < n$
shows *sumlist xs* \$ i = *sum-list* (*map* (λ x. x \$ i) *xs*)
unfolding *M.sumlist-def* **using** *assms*(*1*) **proof**(*induct xs*)
case (*Cons a xs*)
hence *cond*:$\bigwedge x.\ x \in$ *set xs* $\Longrightarrow x \in$ *carrier-vec* n **by** *auto*
from *Cons*(*1*)[*OF cond*] **have** *IH*:*foldr* (+) *xs* (0_v n) \$ i = ($\sum x \leftarrow xs$. x \$ i) **by** *auto*
have (a + *foldr* (+) *xs* (0_v n)) \$ i = a \$ i + ($\sum x \leftarrow xs$. x \$ i)
apply(*subst index-add-vec*) **unfolding** *IH*
using *sumlist-dim*[*OF cond*,*unfolded M.sumlist-def*] *assms* **by** *auto*
then show *?case* **by** *auto* **next**
case *Nil* **thus** *?case* **using** *assms* **by** *auto*
qed

lemma *scalar-prod-left-sum-distrib*:

assumes *vs*: $\bigwedge$ *v*. $v \in$ *set vvs* $\Longrightarrow$ $v \in$ *carrier-vec n* **and** *w*: $w \in$ *carrier-vec n*
shows *sumlist vvs* $\cdot$ *w* = *sum-list* (*map* (λ *v*. *v* $\cdot$ *w*) *vvs*)
using *vs*
proof (*induct vvs*)
case (*Cons v vs*)
from *Cons* **have** *v*: $v \in$ *carrier-vec n* **and** *vs*: *sumlist vs* $\in$ *carrier-vec n*
by (*auto intro*!: *sumlist-carrier*)
have *sumlist* (*v* # *vs*) $\cdot$ *w* = *sumlist* ([*v*] @ *vs*) $\cdot$ *w* **by** *auto*
also have ... = (*v* + *sumlist vs*) $\cdot$ *w*
by (*subst sumlist-append*, *insert Cons v vs*, *auto*)
also have ... = *v* $\cdot$ *w* + (*sumlist vs* $\cdot$ *w*)
by (*rule add-scalar-prod-distrib*[*OF v vs w*])
finally show *?case* **using** *Cons* **by** *auto*
qed (*insert w*, *auto*)

lemma *scalar-prod-right-sum-distrib*:
assumes *vs*: $\bigwedge$ *v*. $v \in$ *set vvs* $\Longrightarrow$ $v \in$ *carrier-vec n* **and** *w*: $w \in$ *carrier-vec n*
shows *w* $\cdot$ *sumlist vvs* = *sum-list* (*map* (λ *v*. *w* $\cdot$ *v*) *vvs*)
by (*subst comm-scalar-prod*[*OF w sumlist-carrier*], *insert vs w*, *force*,
subst scalar-prod-left-sum-distrib[*OF vs w*], *force*,
rule arg-cong[*of - - sum-list*], *rule nth-equalityI*,
auto simp: *set-conv-nth intro*!: *comm-scalar-prod*)

lemma *lincomb-list-add-vec-2*: **assumes** *us*: *set us* $\subseteq$ *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* $\neq$ *j*
shows *x* = *lincomb-list* (*lc* (*j* := *lc j* + *lc i* $*$ *c*)) *us* (**is** - = *?x*)
proof −
let *?xx* = *lc j* + *lc i* $*$ *c*
let *?i* = *us* ! *i*
let *?j* = *us* ! *j*
let *?v* = *?i* + *c* $\cdot_v$ *?j*
let *?ws* = *us* [*i* := *us* ! *i* + *c* $\cdot_v$ *us* ! *j*]
from *us* **have** *usk*: *k* < *length us* $\Longrightarrow$ *us* ! *k* $\in$ *carrier-vec n* **for** *k* **by** *auto*
from *usk i* **have** *ij*: *?i* $\in$ *carrier-vec n ?j* $\in$ *carrier-vec n* **by** *auto*
hence *v*: *c* $\cdot_v$ *?j* $\in$ *carrier-vec n ?v* $\in$ *carrier-vec n* **by** *auto*
with *us* **have** *ws*: *set ?ws* $\subseteq$ *carrier-vec n* **unfolding** *set-conv-nth* **using** *i*
by (*auto*, *rename-tac k*, *case-tac k* = *i*, *auto*)
from *us* **have** *us'*: $\forall$ *w*$\in$*set us*. *dim-vec w* = *n* **by** *auto*
from *ws* **have** *ws'*: $\forall$ *w*$\in$*set ?ws*. *dim-vec w* = *n* **by** *auto*
have *mset*: *mset-set* {*0*..<*length us*} = {#*i*#} + {#*j*#} + (*mset-set* ({*0*..<*length us*} − {*i*,*j*}))
by (*rule multiset-eqI*, *insert i*, *auto*, *rename-tac x*, *case-tac x* $\in$ {*0* ..< *length us*}, *auto*)
define *M2* **where** *M2* = *M*.*summset*
{#*lc ia* $\cdot_v$ *?ws* ! *ia*. *ia* $\in$# *mset-set* ({*0*..<*length us*} − {*i*, *j*})#}
define *M1* **where** *M1* = *M*.*summset* {#(*if i* = *j then ?xx else lc i*) $\cdot_v$ *us* ! *i*. *i* $\in$# *mset-set* ({*0*..<*length us*} − {*i*, *j*})#}
have *M1*: *M1* $\in$ *carrier-vec n* **unfolding** *M1-def* **using** *usk* **by** *fastforce*

have *M2*: $M1 = M2$ **unfolding** *M2-def M1-def*
by (*rule arg-cong*[*of - - M.summset*], *rule multiset.map-cong0*, *insert i usk*, *auto*)
have *x1*: $x = lc\ j \cdot_v\ ?j + (lc\ i \cdot_v\ ?i + lc\ i \cdot_v\ (c \cdot_v\ ?j) + M1)$
unfolding *x lincomb-list-def M2 M2-def*
apply (*subst sumlist-as-summset*, (*insert us ws i v ij*, *auto simp*: *set-conv-nth*)[*1*], *insert i ij v us ws usk*,
simp add: *mset smult-add-distrib-vec*[*OF ij*(*1*) *v*(*1*)])
by (*subst M.summset-add-mset*, *auto*)+
have *x2*: $?x = ?xx \cdot_v\ ?j + (lc\ i \cdot_v\ ?i + M1)$
unfolding *x lincomb-list-def M1-def*
apply (*subst sumlist-as-summset*, (*insert us ws i v ij*, *auto simp*: *set-conv-nth*)[*1*], *insert i ij v us ws usk*,
simp add: *mset smult-add-distrib-vec*[*OF ij*(*1*) *v*(*1*)])
by (*subst M.summset-add-mset*, *auto*)+
show *?thesis* **unfolding** *x1 x2* **using** *M1 ij*
by (*intro eq-vecI*, *auto simp*: *field-simps*)
qed

lemma *lincomb-list-add-vec-1*: **assumes** *us*: $set\ us \subseteq carrier\text{-}vec\ n$
and *x*: $x = lincomb\text{-}list\ lc\ us$
and *i*: $j < length\ us\ i < length\ us\ i \neq j$
shows $x = lincomb\text{-}list\ (lc\ (j := lc\ j - lc\ i * c))\ (us\ [i := us\ !\ i + c \cdot_v\ us\ !\ j])$ (**is** $- = ?x$)
proof −
let $?i = us\ !\ i$
let $?j = us\ !\ j$
let $?v = ?i + c \cdot_v\ ?j$
let $?ws = us\ [i := us\ !\ i + c \cdot_v\ us\ !\ j]$
from *us* **have** *usk*: $k < length\ us \Longrightarrow us\ !\ k \in carrier\text{-}vec\ n$ **for** *k* **by** *auto*
from *usk i* **have** *ij*: $?i \in carrier\text{-}vec\ n\ ?j \in carrier\text{-}vec\ n$ **by** *auto*
hence *v*: $c \cdot_v\ ?j \in carrier\text{-}vec\ n\ ?v \in carrier\text{-}vec\ n$ **by** *auto*
with *us* **have** *ws*: $set\ ?ws \subseteq carrier\text{-}vec\ n$ **unfolding** *set-conv-nth* **using** *i*
by (*auto*, *rename-tac k*, *case-tac* $k = i$, *auto*)
from *us* **have** *us′*: $\forall w \in set\ us.\ dim\text{-}vec\ w = n$ **by** *auto*
from *ws* **have** *ws′*: $\forall w \in set\ ?ws.\ dim\text{-}vec\ w = n$ **by** *auto*
have *mset*: $mset\text{-}set\ \{0..<length\ us\} = \{\#i\#\} + \{\#j\#\} + (mset\text{-}set\ (\{0..<length\ us\} - \{i,j\}))$
by (*rule multiset-eqI*, *insert i*, *auto*, *rename-tac x*, *case-tac* $x \in \{0\ ..<\ length\ us\}$, *auto*)
define *M2* **where** $M2 = M.summset$
$\{\#(if\ ia = j\ then\ lc\ j - lc\ i * c\ else\ lc\ ia) \cdot_v\ ?ws\ !\ ia$
$.\ ia \in\#\ mset\text{-}set\ (\{0..<length\ us\} - \{i, j\})\#\}$
define *M1* **where** $M1 = M.summset\ \{\#lc\ i \cdot_v\ us\ !\ i.\ i \in\#\ mset\text{-}set\ (\{0..<length\ us\} - \{i, j\})\#\}$
have *M1*: $M1 \in carrier\text{-}vec\ n$ **unfolding** *M1-def* **using** *usk* **by** *fastforce*
have *M2*: $M1 = M2$ **unfolding** *M2-def M1-def*
by (*rule arg-cong*[*of - - M.summset*], *rule multiset.map-cong0*, *insert i usk*, *auto*)

```
  have x1: x = lc j ·v ?j + (lc i ·v ?i + M1)
    unfolding x lincomb-list-def M1-def
   apply (subst sumlist-as-summset, (insert us ws i v ij, auto simp: set-conv-nth)[1],
insert i ij v us ws usk,
      simp add: mset smult-add-distrib-vec[OF ij(1) v(1)])
    by (subst M.summset-add-mset, auto)+
  have x2: ?x = (lc j − lc i ∗ c) ·v ?j + (lc i ·v ?i + lc i ·v (c ·v ?j) + M1)
    unfolding x lincomb-list-def M2 M2-def
   apply (subst sumlist-as-summset, (insert us ws i v ij, auto simp: set-conv-nth)[1],
insert i ij v us ws usk,
      simp add: mset smult-add-distrib-vec[OF ij(1) v(1)])
    by (subst M.summset-add-mset, auto)+
  show ?thesis unfolding x1 x2 using M1 ij
    by (intro eq-vecI, auto simp: field-simps)
qed

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 ·v us ! j])) (is - = span (set
?ws))
proof −
  let ?i = us ! i
  let ?j = us ! j
  let ?v = ?i + c ·v ?j
  from us i have ij: ?i ∈ carrier-vec n ?j ∈ carrier-vec n by auto
  hence v: ?v ∈ carrier-vec n by auto
  with us have ws: set ?ws ⊆ carrier-vec n unfolding set-conv-nth using i
    by (auto, rename-tac k, case-tac k = i, auto)
  have span (set us) = span-list us unfolding span-list-as-span[OF us] ..
  also have ... = span-list ?ws
  proof −
    {
      fix x
      assume x ∈ span-list us
      then obtain lc where x = lincomb-list lc us by (metis in-span-listE)
      from lincomb-list-add-vec-1[OF us this i, of c]
      have x ∈ span-list ?ws unfolding span-list-def by auto
    }
    moreover
    {
      fix x
      assume x ∈ span-list ?ws
      then obtain lc where x = lincomb-list lc ?ws by (metis in-span-listE)
      from lincomb-list-add-vec-2[OF us this i]
      have x ∈ span-list us unfolding span-list-def by auto
```

```
    }
    ultimately show ?thesis by blast
  qed
  also have ... = span (set ?ws) unfolding span-list-as-span[OF ws] ..
  finally show ?thesis .
qed

lemma prod-in-span[intro!]:
  assumes b ∈ carrier-vec n S ⊆ carrier-vec n a = 0 ∨ b ∈ span S
  shows a ·v b ∈ span S
proof(cases a = 0)
  case True
  then show ?thesis by (auto simp:lmult-0[OF assms(1)] span-zero)
next
  case False with assms have b ∈ span S by auto
  from this[THEN in-spanE]
  obtain aa A where a[intro!]: b = lincomb aa A finite A A ⊆ S by auto
  hence [intro!]:(λv. aa v ·v v) ∈ A → carrier-vec n using assms by auto
  show ?thesis proof
    show a ·v b = lincomb (λ v. a * aa v) A using a(1) unfolding lincomb-def
smult-smult-assoc[symmetric]
      by(subst finsum-smult[symmetric]) force+
  qed auto
qed

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 −
  have 1m n ∈ carrier-mat n n by auto
  from det-non-zero-imp-unit[OF M det] gauss-jordan-check-invertable[OF M this]
  have gj-fst:(fst (gauss-jordan M (1m n)) = 1m n) by metis
  define Mi where Mi = snd (gauss-jordan M (1m n))
  with gj-fst have gj:gauss-jordan M (1m n) = (1m n, Mi)
    unfolding fst-def snd-def by (auto split:prod.split)
  from gauss-jordan-compute-inverse(1,3)[OF M gj]
  have Mi: Mi ∈ carrier-mat n n and is1:M * Mi = 1m n by metis+
  from arg-cong[OF eq, of λ M. M * Mi]
  show A = B unfolding carr[THEN assoc-mult-mat[OF - M Mi]] is1 carr[THEN
right-mult-one-mat].
qed

end
```

fun *find-index* :: $'b$ *list* $\Rightarrow$ $'b$ $\Rightarrow$ *nat* **where**
find-index [] - = *0* |
find-index (*x#xs*) *y* = (*if x* = *y then 0 else find-index xs y* + *1*)

lemma *find-index-not-in-set*: $x \notin$ *set xs* $\longleftrightarrow$ *find-index xs x* = *length xs*
by (*induction xs*) *auto*

lemma *find-index-in-set*: $x \in$ *set xs* $\Longrightarrow$ *xs* ! (*find-index xs x*) = *x*
by (*induction xs*) *auto*

lemma *find-index-inj*: *inj-on* (*find-index xs*) (*set xs*)
by (*induction xs*) (*auto simp add*: *inj-on-def*)

lemma *find-index-leq-length*: *find-index xs x* < *length xs* $\longleftrightarrow$ $x \in$ *set xs*
by (*induction xs*) (*auto*)

context *vec-module*
begin
definition *lattice-of* :: $'a$ *vec list* $\Rightarrow$ $'a$ *vec set* **where**
lattice-of fs = *range* (λ *c*. *sumlist* (*map* (λ *i*. *of-int* (*c i*) $\cdot_v$ *fs* ! *i*) [*0* ..< *length fs*]))

lemma *lattice-of-finsum*:
assumes *set fs* $\subseteq$ *carrier-vec n*
shows *lattice-of fs* = *range* (λ *c*. *finsum V* (λ *i*. *of-int* (*c i*) $\cdot_v$ *fs* ! *i*) {*0* ..< *length fs*})
proof −
have *sumlist* (*map* (λ *i*. *of-int* (*c i*) $\cdot_v$ *fs* ! *i*) [*0* ..< *length fs*])
= *finsum V* (λ *i*. *of-int* (*c i*) $\cdot_v$ *fs* ! *i*) {*0* ..< *length fs*} **for** *c*
using *assms* **by** (*subst sumlist-map-as-finsum*) (*fastforce*)+
then show *?thesis*
unfolding *lattice-of-def* **by** *auto*
qed

lemma *in-latticeE*: **assumes** $f \in$ *lattice-of fs* **obtains** *c* **where**
f = *sumlist* (*map* (λ *i*. *of-int* (*c i*) $\cdot_v$ *fs* ! *i*) [*0* ..< *length fs*])
using *assms* **unfolding** *lattice-of-def* **by** *auto*

lemma *in-latticeI*: **assumes** *f* = *sumlist* (*map* (λ *i*. *of-int* (*c i*) $\cdot_v$ *fs* ! *i*) [*0* ..< *length fs*])
shows $f \in$ *lattice-of fs*
using *assms* **unfolding** *lattice-of-def* **by** *auto*

lemma *finsum-over-indexes-to-vectors*:
assumes *set vs* $\subseteq$ *carrier-vec n l* = *length vs*
shows $\exists$ *c*. ($\bigoplus_V x \in \{0..<l\}$. *of-int* (*g x*) $\cdot_v$ *vs* ! *x*) = ($\bigoplus_V v \in$ *set vs*. *of-int* (*c v*) $\cdot_v$ *v*)
using *assms* **proof** (*induction l arbitrary*: *vs*)

case $(Suc\ l)$
then obtain $vs'\ v$ **where** vs'-def: $vs = vs'\ @\ [v]$
by (*metis Zero-not-Suc length-0-conv rev-exhaust*)
have c: $\exists\, c.\ (\bigoplus{}_V i{\in}\{0..{<}l\}.\ of\text{-}int\ (g\ i) \cdot_v vs'\ !\ i) = (\bigoplus{}_V v{\in}set\ vs'.\ of\text{-}int\ (c\ v) \cdot_v v)$
using *Suc vs'-def* **by** (*auto*)
then obtain c
where c-def: $(\bigoplus{}_V x{\in}\{0..{<}l\}.\ of\text{-}int\ (g\ x) \cdot_v vs'\ !\ x) = (\bigoplus{}_V v{\in}set\ vs'.\ of\text{-}int\ (c\ v) \cdot_v v)$
by *blast*
have $(\bigoplus{}_V x{\in}\{0..{<}Suc\ l\}.\ of\text{-}int\ (g\ x) \cdot_v vs\ !\ x)$
$= of\text{-}int\ (g\ l) \cdot_v vs\ !\ l + (\bigoplus{}_V x{\in}\{0..{<}l\}.\ of\text{-}int\ (g\ x) \cdot_v vs\ !\ x)$
using *Suc* **by** (*subst finsum-insert*[*symmetric*]) (*fastforce intro*!: *finsum-cong'*)+
also have $vs = vs'\ @\ [v]$
using *vs'-def* **by** *simp*
also have $(\bigoplus{}_V x{\in}\{0..{<}l\}.\ of\text{-}int\ (g\ x) \cdot_v (vs'\ @\ [v])\ !\ x) = (\bigoplus{}_V x{\in}\{0..{<}l\}.\ of\text{-}int\ (g\ x) \cdot_v vs'\ !\ x)$
using *Suc vs'-def* **by** (*intro finsum-cong'*) (*auto simp add*: *in-mono nth-append*)
also note *c-def*
also have $(vs'\ @\ [v])\ !\ l = v$
using *Suc vs'-def* **by** *auto*
also have $\exists\, d'.\ of\text{-}int\ (g\ l) \cdot_v v + (\bigoplus{}_V v{\in}set\ vs'.\ of\text{-}int\ (c\ v) \cdot_v v) = (\bigoplus{}_V v{\in}set\ vs.\ of\text{-}int\ (d'\ v) \cdot_v v)$
proof (*cases* $v \in set\ vs'$)
case *True*
then have I: $set\ vs' = insert\ v\ (set\ vs' - \{v\})$
by *blast*
define c' **where** $c'\ x = (if\ x = v\ then\ c\ x + g\ l\ else\ c\ x)$ **for** x
have $of\text{-}int\ (g\ l) \cdot_v v + (\bigoplus{}_V v{\in}set\ vs'.\ of\text{-}int\ (c\ v) \cdot_v v)$
$= of\text{-}int\ (g\ l) \cdot_v v + (of\text{-}int\ (c\ v) \cdot_v v + (\bigoplus{}_V v{\in}set\ vs' - \{v\}.\ of\text{-}int\ (c\ v) \cdot_v v))$
using *Suc vs'-def* **by** (*subst* I, *subst finsum-insert*) *fastforce*+
also have $\ldots = of\text{-}int\ (g\ l) \cdot_v v + of\text{-}int\ (c\ v) \cdot_v v + (\bigoplus{}_V v{\in}set\ vs' - \{v\}.\ of\text{-}int\ (c\ v) \cdot_v v)$
using *Suc vs'-def* **by** (*subst a-assoc*) (*auto intro*!: *finsum-closed*)
also have $of\text{-}int\ (g\ l) \cdot_v v + of\text{-}int\ (c\ v) \cdot_v v = of\text{-}int\ (c'\ v) \cdot_v v$
unfolding *c'-def* **by** (*auto simp add*: *add-smult-distrib-vec*)
also have $(\bigoplus{}_V v{\in}set\ vs' - \{v\}.\ of\text{-}int\ (c\ v) \cdot_v v) = (\bigoplus{}_V v{\in}set\ vs' - \{v\}.\ of\text{-}int\ (c'\ v) \cdot_v v)$
using *Suc vs'-def* **unfolding** *c'-def* **by** (*intro finsum-cong'*) (*auto*)
also have $of\text{-}int\ (c'\ v) \cdot_v v + (\bigoplus{}_V v{\in}set\ vs' - \{v\}.\ of\text{-}int\ (c'\ v) \cdot_v v)$
$= (\bigoplus{}_V v{\in}insert\ v\ (set\ vs').\ of\text{-}int\ (c'\ v) \cdot_v v)$
using *Suc vs'-def* **by** (*subst finsum-insert*[*symmetric*]) (*auto*)
finally show *?thesis*
using *vs'-def* **by** *force*
next
case *False*
define c' **where** $c'\ x = (if\ x = v\ then\ g\ l\ else\ c\ x)$ **for** x
have $of\text{-}int\ (g\ l) \cdot_v v + (\bigoplus{}_V v{\in}set\ vs'.\ of\text{-}int\ (c\ v) \cdot_v v)$

$= \textit{of-int}\ (c'\ v) \cdot_v v + (\bigoplus_V v{\in}\textit{set}\ vs'.\ \textit{of-int}\ (c\ v) \cdot_v v)$
unfolding *c′-def* **by** *simp*
also have $(\bigoplus_V v{\in}\textit{set}\ vs'.\ \textit{of-int}\ (c\ v) \cdot_v v) = (\bigoplus_V v{\in}\textit{set}\ vs'.\ \textit{of-int}\ (c'\ v) \cdot_v v)$
unfolding *c′-def* **using** *Suc False vs′-def* **by** (*auto intro*!: *finsum-cong′*)
also have $\textit{of-int}\ (c'\ v) \cdot_v v + (\bigoplus_V v{\in}\textit{set}\ vs'.\ \textit{of-int}\ (c'\ v) \cdot_v v)$
$= (\bigoplus_V v{\in}\textit{insert}\ v\ (\textit{set}\ vs').\ \textit{of-int}\ (c'\ v) \cdot_v v)$
using *False Suc vs′-def* **by** (*subst finsum-insert*[*symmetric*]) (*auto*)
also have $(\bigoplus_V v{\in}\textit{set}\ vs'.\ \textit{of-int}\ (c'\ v) \cdot_v v) = (\bigoplus_V v{\in}\textit{set}\ vs'.\ \textit{of-int}\ (c\ v) \cdot_v v)$
unfolding *c′-def* **using** *False Suc vs′-def* **by** (*auto intro*!: *finsum-cong′*)
finally show *?thesis*
using *vs′-def* **by** *auto*
qed
finally show *?case*
unfolding *vs′-def* **by** *blast*
qed (*auto*)

lemma *lattice-of-altdef*:
assumes *set vs* ⊆ *carrier-vec n*
shows $\textit{lattice-of}\ vs = \textit{range}\ (\lambda c.\ \bigoplus_V v{\in}\textit{set}\ vs.\ \textit{of-int}\ (c\ v) \cdot_v v)$
proof −
have $v \in \textit{lattice-of}\ vs$ **if** $v \in \textit{range}\ (\lambda c.\ \bigoplus_V v{\in}\textit{set}\ vs.\ \textit{of-int}\ (c\ v) \cdot_v v)$ **for** *v*
proof −
obtain *c* **where** *v*: $v = (\bigoplus_V v{\in}\textit{set}\ vs.\ \textit{of-int}\ (c\ v) \cdot_v v)$
using ‹$v \in \textit{range}\ (\lambda c.\ \bigoplus_V v{\in}\textit{set}\ vs.\ \textit{of-int}\ (c\ v) \cdot_v v)$› **by** (*auto*)
define *c′* **where** $c'\ i = (\textit{if find-index}\ vs\ (vs\ !\ i) = i\ \textit{then}\ c\ (vs\ !\ i)\ \textit{else}\ 0)$ **for** *i*
have $v = (\bigoplus_V v{\in}\textit{set}\ vs.\ \textit{of-int}\ (c'\ (\textit{find-index}\ vs\ v)) \cdot_v vs\ !\ (\textit{find-index}\ vs\ v))$
unfolding *v*
using *assms* **by** (*auto intro*!: *finsum-cong′ simp add*: *c′-def find-index-in-set in-mono*)
also have $\ldots = (\bigoplus_V i{\in}\textit{find-index}\ vs\ `\ (\textit{set}\ vs).\ \textit{of-int}\ (c'\ i) \cdot_v vs\ !\ i)$
using *assms find-index-in-set find-index-inj* **by** (*subst finsum-reindex*) *fastforce*+
also have $\ldots = (\bigoplus_V i{\in}\textit{set}\ [0..{<}\textit{length}\ vs].\ \textit{of-int}\ (c'\ i) \cdot_v vs\ !\ i)$
proof −
have $i \in \textit{find-index}\ vs\ `\ \textit{set}\ vs$ **if** $i < \textit{length}\ vs$ $\textit{find-index}\ vs\ (vs\ !\ i) = i$ **for** *i*
using *that* **by** (*metis imageI nth-mem*)
then show *?thesis*
unfolding *c′-def* **using** *find-index-leq-length assms*
by (*intro add.finprod-mono-neutral-cong-left*) (*auto simp add*: *in-mono find-index-leq-length*)
qed
also have $\ldots = \textit{sumlist}\ (\textit{map}\ (\lambda i.\ \textit{of-int}\ (c'\ i) \cdot_v vs\ !\ i)\ [0..{<}\textit{length}\ vs])$
using *assms* **by** (*subst sumlist-map-as-finsum*) (*fastforce*)+
finally show *?thesis*
unfolding *lattice-of-def* **by** *blast*
qed
moreover have $v \in \textit{range}\ (\lambda c.\ \bigoplus_V v{\in}\textit{set}\ vs.\ \textit{of-int}\ (c\ v) \cdot_v v)$ **if** $v \in \textit{lattice-of}\ vs$ **for** *v*
proof −

obtain *c* **where** $v = sumlist\ (map\ (\lambda i.\ of\text{-}int\ (c\ i) \cdot_v vs\ !\ i)\ [0..<length\ vs])$
using ‹$v \in lattice\text{-}of\ vs$› **unfolding** *lattice-of-def* **by** (*auto*)
also have $\ldots = (\bigoplus_V x\in\{0..<length\ vs\}.\ of\text{-}int\ (c\ x) \cdot_v vs\ !\ x)$
using *that assms* **by** (*subst sumlist-map-as-finsum*) *fastforce+*
also obtain *d* **where** $\ldots = (\bigoplus_V v\in set\ vs.\ of\text{-}int\ (d\ v) \cdot_v v)$
using *finsum-over-indexes-to-vectors assms* **by** *blast*
finally show *?thesis*
by *blast*
qed
ultimately show *?thesis*
by *fastforce*
qed

lemma *basis-in-latticeI*:
assumes *fs*: $set\ fs \subseteq carrier\text{-}vec\ n$ **and** $f \in set\ fs$
shows $f \in lattice\text{-}of\ fs$
proof −
define $c :: 'a\ vec \Rightarrow int$ **where** $c\ v = (if\ v = f\ then\ 1\ else\ 0)$ **for** *v*
have $f = (\bigoplus_V v\in\{f\}.\ of\text{-}int\ (c\ v) \cdot_v v)$
using *assms* **by** (*auto simp add*: *c-def*)
also have $\ldots = (\bigoplus_V v\in set\ fs.\ of\text{-}int\ (c\ v) \cdot_v v)$
using *assms* **by** (*intro add.finprod-mono-neutral-cong-left*) (*auto simp add*: *c-def*)
finally show *?thesis*
using *assms lattice-of-altdef* **by** *blast*
qed

lemma *lattice-of-eq-set*:
assumes $set\ fs = set\ gs$ $set\ fs \subseteq carrier\text{-}vec\ n$
shows $lattice\text{-}of\ fs = lattice\text{-}of\ gs$
using *assms lattice-of-altdef* **by** *simp*

lemma *lattice-of-swap*: **assumes** *fs*: $set\ fs \subseteq carrier\text{-}vec\ n$
and *ij*: $i < length\ fs$ $j < length\ fs$ $i \neq j$
and *gs*: $gs = fs[\ i := fs\ !\ j,\ j := fs\ !\ i]$
shows $lattice\text{-}of\ gs = lattice\text{-}of\ fs$
using *assms mset-swap* **by** (*intro lattice-of-eq-set*) *auto*

lemma *lattice-of-add*: **assumes** *fs*: $set\ fs \subseteq carrier\text{-}vec\ n$
and *ij*: $i < length\ fs$ $j < length\ fs$ $i \neq j$
and *gs*: $gs = fs[\ i := fs\ !\ i + of\text{-}int\ l \cdot_v fs\ !\ j]$
shows $lattice\text{-}of\ gs = lattice\text{-}of\ fs$
proof −
{
fix *i j l* **and** $fs :: 'a\ vec\ list$
assume $*$: $i < j$ $j < length\ fs$ **and** *fs*: $set\ fs \subseteq carrier\text{-}vec\ n$
note $* = ij(1)\ *$
let $?gs = fs[\ i := fs\ !\ i + of\text{-}int\ l \cdot_v fs\ !\ j]$
let $?len = [0..<i]\ @\ [i]\ @\ [Suc\ i..<j]\ @\ [j]\ @\ [Suc\ j..<length\ fs]$

have [*0* ..< *length fs*] = [*0* ..< *j*] @ [*j*] @ [*Suc j* ..< *length fs*] **using** ∗
by (*metis append-Cons append-self-conv2 less-Suc-eq-le less-imp-add-positive upt-add-eq-append*
upt-conv-Cons zero-less-Suc)
also have [*0* ..< *j*] = [*0* ..< *i*] @ [*i*] @ [*Suc i* ..< *j*] **using** ∗
by (*metis append-Cons append-self-conv2 less-Suc-eq-le less-imp-add-positive upt-add-eq-append*
upt-conv-Cons zero-less-Suc)
finally have *len*: [*0*..<*length fs*] = *?len* **by** *simp*
from *fs* **have** *fs*: $\bigwedge$ *i*. *i* < *length fs* ⟹ *fs* ! *i* ∈ *carrier-vec n* **unfolding** *set-conv-nth* **by** *auto*
from *fs* **have** *fsd*: $\bigwedge$ *i*. *i* < *length fs* ⟹ *dim-vec* (*fs* ! *i*) = *n* **by** *auto*
from *fsd*[*of i*] *fsd*[*of j*] ∗ **have** *fsd*: *dim-vec* (*fs* ! *i*) = *n* *dim-vec* (*fs* ! *j*) = *n* **by** *auto*
{
fix *f*
assume *f* ∈ *lattice-of fs*
from *in-latticeE*[*OF this*, *unfolded len*] **obtain** *c* **where**
f: *f* = *sumlist* (*map* (λ*i*. *of-int* (*c i*) $\cdot_v$ *fs* ! *i*) *?len*) **by** *auto*
define *sc* **where** *sc* = (λ *xs*. *sumlist* (*map* (λ*i*. *of-int* (*c i*) $\cdot_v$ *fs* ! *i*) *xs*))
define *d* **where** *d* = (λ *k*. *if k* = *j then c j* − *c i* ∗ *l else c k*)
define *sd* **where** *sd* = (λ *xs*. *sumlist* (*map* (λ*i*. *of-int* (*d i*) $\cdot_v$ *?gs* ! *i*) *xs*))
have *isc*: *set is* ⊆ {*0* ..< *length fs*} ⟹ *sc is* ∈ *carrier-vec n* **for** *is*
unfolding *sc-def* **by** (*intro sumlist-carrier*, *auto simp*: *fs*)
have *isd*: *set is* ⊆ {*0* ..< *length fs*} ⟹ *sd is* ∈ *carrier-vec n* **for** *is*
unfolding *sd-def* **using** ∗ **by** (*intro sumlist-carrier*, *auto*, *rename-tac k*,
case-tac k = *i*, *auto simp*: *fs*)
let *?a* = *sc* [*0*..<*i*] **let** *?b* = *sc* [*i*] **let** *?c* = *sc* [*Suc i* ..< *j*] **let** *?d* = *sc* [*j*]
let *?e* = *sc* [*Suc j* ..< *length fs*]
let *?A* = *sd* [*0*..<*i*] **let** *?B* = *sd* [*i*] **let** *?C* = *sd* [*Suc i* ..< *j*] **let** *?D* = *sd* [*j*]
let *?E* = *sd* [*Suc j* ..< *length fs*]
let *?CC* = *carrier-vec n*
have *ae*: *?a* ∈ *?CC* *?b* ∈ *?CC* *?c* ∈ *?CC* *?d* ∈ *?CC* *?e* ∈ *?CC*
using ∗ **by** (*auto intro*: *isc*)
have *AE*: *?A* ∈ *?CC* *?B* ∈ *?CC* *?C* ∈ *?CC* *?D* ∈ *?CC* *?E* ∈ *?CC*
using ∗ **by** (*auto intro*: *isd*)
have *sc-sd*: {*i*,*j*} ∩ *set is* ⊆ {} ⟹ *sc is* = *sd is* **for** *is*
unfolding *sc-def sd-def* **by** (*rule arg-cong*[*of - - sumlist*], *rule map-cong*, *auto simp*: *d-def*,
rename-tac k, *case-tac i* = *k*, *auto*)
have *f* = *?a* + (*?b* + (*?c* + (*?d* + *?e*)))
unfolding *f map-append sc-def* **using** *fs* ∗
by ((*subst sumlist-append*, *force*, *force*)+, *simp*)
also have . . . = *?a* + ((*?b* + *?d*) + (*?c* + *?e*)) **using** *ae* **by** *auto*
also have . . . = *?A* + ((*?b* + *?d*) + (*?C* + *?E*))
using ∗ **by** (*auto simp*: *sc-sd*)
also have *?b* + *?d* = *?B* + *?D* **unfolding** *sd-def sc-def d-def sumlist-def*
by (*rule eq-vecI*, *insert* ∗ *fsd*, *auto simp*: *algebra-simps*)
finally have *f* = *?A* + (*?B* + (*?C* + (*?D* + *?E*))) **using** *AE* **by** *auto*

also have ... = *sumlist* (*map* (λ*i*. *of-int* (*d i*) $\cdot_v$ *?gs* ! *i*) *?len*)
unfolding *f map-append sd-def* **using** *fs* $*$
by ((*subst sumlist-append*, *force*, *force*)+, *simp*)
also have ... = *sumlist* (*map* (λ*i*. *of-int* (*d i*) $\cdot_v$ *?gs* ! *i*) [*0* ..< *length ?gs*])
unfolding *len*[*symmetric*] **by** *simp*
finally have *f* = *sumlist* (*map* (λ*i*. *of-int* (*d i*) $\cdot_v$ *?gs* ! *i*) [*0* ..< *length ?gs*]) **.**
from *in-latticeI*[*OF this*] **have** *f* $\in$ *lattice-of ?gs* **.**
}
hence *lattice-of fs* $\subseteq$ *lattice-of ?gs* **by** *blast*
} **note** *main* = *this*
{
fix *i j* **and** *fs* :: *'a vec list*
assume $*$: *i* < *j j* < *length fs* **and** *fs*: *set fs* $\subseteq$ *carrier-vec n*
let *?gs* = *fs*[*i* := *fs* ! *i* + *of-int l* $\cdot_v$ *fs* ! *j*]
define *gs* **where** *gs* = *?gs*
from *main*[*OF* $*$ *fs*, *of l*, *folded gs-def*]
have *one*: *lattice-of fs* $\subseteq$ *lattice-of gs* **.**
have $*$: *i* < *j j* < *length gs set gs* $\subseteq$ *carrier-vec n* **using** $*$ *fs* **unfolding** *gs-def* *set-conv-nth*
by (*auto*, *rename-tac k*, *case-tac k* = *i*, (*force intro*!: *add-carrier-vec*)+)
from *fs* **have** *fs*: $\bigwedge$ *i*. *i* < *length fs* $\Longrightarrow$ *fs* ! *i* $\in$ *carrier-vec n* **unfolding** *set-conv-nth* **by** *auto*
from *fs* **have** *fsd*: $\bigwedge$ *i*. *i* < *length fs* $\Longrightarrow$ *dim-vec* (*fs* ! *i*) = *n* **by** *auto*
from *fsd*[*of i*] *fsd*[*of j*] $*$ **have** *fsd*: *dim-vec* (*fs* ! *i*) = *n dim-vec* (*fs* ! *j*) = *n* **by** (*auto simp*: *gs-def*)
from *main*[*OF* $*$, *of* $-l$]
have *lattice-of gs* $\subseteq$ *lattice-of* (*gs*[*i* := *gs* ! *i* + *of-int* ($-$ *l*) $\cdot_v$ *gs* ! *j*]) **.**
also have *gs*[*i* := *gs* ! *i* + *of-int* ($-$ *l*) $\cdot_v$ *gs* ! *j*] = *fs* **unfolding** *gs-def*
by (*rule nth-equalityI*, *auto*, *insert* $*$ *fsd*, *rename-tac k*, *case-tac k* = *i*, *auto*)
ultimately have *lattice-of fs* = *lattice-of ?gs* **using** *one* **unfolding** *gs-def* **by** *auto*
} **note** *main* = *this*
show *?thesis*
proof (*cases i* < *j*)
case *True*
from *main*[*OF this ij*(*2*) *fs*] **show** *?thesis* **unfolding** *gs* **by** *simp*
next
case *False*
with *ij* **have** *ji*: *j* < *i* **by** *auto*
define *hs* **where** *hs* = *fs*[*i* := *fs* ! *j*, *j* := *fs* ! *i*]
define *ks* **where** *ks* = *hs*[*j* := *hs* ! *j* + *of-int l* $\cdot_v$ *hs* ! *i*]
from *ij fs* **have** *ij'*: *i* < *length hs set hs* $\subseteq$ *carrier-vec n* **unfolding** *hs-def* **by** *auto*
hence *ij''*: *set ks* $\subseteq$ *carrier-vec n i* < *length ks j* < *length ks i* $\neq$ *j*
using *ji* **unfolding** *ks-def set-conv-nth* **by** (*auto*, *rename-tac k*, *case-tac k* = *i*,
force, *case-tac k* = *j*, (*force intro*!: *add-carrier-vec*)+)
from *lattice-of-swap*[*OF fs ij refl*]
have *lattice-of fs* = *lattice-of hs* **unfolding** *hs-def* **by** *auto*

also have ... = *lattice-of ks*
using *main*[*OF ji ij'*] **unfolding** *ks-def* .
also have ... = *lattice-of* (*ks*[*i* := *ks* ! *j*, *j* := *ks* ! *i*])
by (*rule sym*, *rule lattice-of-swap*[*OF ij'' refl*])
also have *ks*[*i* := *ks* ! *j*, *j* := *ks* ! *i*] = *gs* **unfolding** *gs ks-def hs-def*
by (*rule nth-equalityI*, *insert ij*, *auto*,
rename-tac k, *case-tac k* = *i*, *force*, *case-tac k* = *j*, *auto*)
finally show *?thesis* **by** *simp*
qed
qed

lemma *lattice-of-altdef-lincomb*:
assumes *set fs* ⊆ *carrier-vec n*
shows *lattice-of fs* = {*y*. ∃ *f*. *lincomb* (*of-int* ∘ *f*) (*set fs*) = *y*}
unfolding *lincomb-def lattice-of-altdef*[*OF assms*] *image-def* **by** *auto*

definition *orthogonal-complement W* = {*x*. *x* ∈ *carrier-vec n* ∧ (∀ *y* ∈ *W*. *x* · *y* = *0*)}

lemma *orthogonal-complement-subset*:
assumes *A* ⊆ *B*
shows *orthogonal-complement B* ⊆ *orthogonal-complement A*
unfolding *orthogonal-complement-def* **using** *assms* **by** *auto*

end

context *vec-space*
begin

lemma *in-orthogonal-complement-span*[*simp*]:
assumes [*intro*]:*S* ⊆ *carrier-vec n*
shows *orthogonal-complement* (*span S*) = *orthogonal-complement S*
proof
show *orthogonal-complement* (*span S*) ⊆ *orthogonal-complement S*
by(*fact orthogonal-complement-subset*[*OF in-own-span*[*OF assms*]])
{**fix** *x* :: *'a vec*
fix *a* **fix** *A* :: *'a vec set*
assume *x* [*intro*]:*x* ∈ *carrier-vec n* **and** *f*: *finite A* **and** *S*:*A* ⊆ *S*
assume *i0*:∀ *y*∈*S*. *x* · *y* = *0*
have *x* · *lincomb a A* = *0*
unfolding *comm-scalar-prod*[*OF x lincomb-closed*[*OF subset-trans*[*OF S assms*]]]
proof(*insert S*,*atomize*(*full*),*rule finite-induct*[*OF f*],*goal-cases*)
case *1* **thus** *?case* **using** *assms x* **by** *force*
next
case (*2 f F*)
{ **assume** *i*:*insert f F* ⊆ *S*

hence F:$F \subseteq S$ **and** f: $f \in S$ **by** *auto*
from F f *assms*
have [*intro*]:$F \subseteq$ *carrier-vec n*
and *fc*[*intro*]:$f \in$ *carrier-vec n*
and [*intro*]:$x \in F \Longrightarrow x \in$ *carrier-vec n* **for** x **by** *auto*
have *laf*:*lincomb a F* $\cdot$ $x = 0$ **using** F *2* **by** *auto*
have [*simp*]:$(\sum u \in F.\ (a\ u \cdot_v u) \cdot x) = 0$
by(*insert laf*[*unfolded lincomb-def*],*atomize*(*full*),*subst finsum-scalar-prod-sum*) *auto*
from *f i0* **have** [*simp*]:$f \cdot x = 0$ **by** (*subst comm-scalar-prod*) *auto*
from *lincomb-closed*[*OF subset-trans*[*OF i assms*]]
have *lincomb a* (*insert f F*) $\cdot$ $x = 0$ **unfolding** *lincomb-def*
apply(*subst finsum-scalar-prod-sum*,*force*,*force*)
using *2*(*1*,*2*) *smult-scalar-prod-distrib*[*OF fc x*] **by** *auto*
} **thus** *?case* **by** *auto*
qed
}
thus *orthogonal-complement S* $\subseteq$ *orthogonal-complement* (*span S*)
unfolding *orthogonal-complement-def span-def* **by** *auto*
qed

end

context
begin

interpretation *vec-module TYPE*(*int*) **.**

lemma *lattice-of-cols-as-mat-mult*:
assumes A: $A \in$ *carrier-mat n nc*
shows *lattice-of* (*cols A*) = $\{y \in$ *carrier-vec* (*dim-row A*). $\exists x \in$ *carrier-vec* (*dim-col A*). $A *_v x = y\}$
proof −
let *?ws* = *cols A*
have *set-cols-in*: *set* (*cols A*) $\subseteq$ *carrier-vec n* **using** A **unfolding** *cols-def* **by** *auto*
have *lincomb* (*of-int* $\circ$ *f*)(*set ?ws*) $\in$ *carrier-vec* (*dim-row A*) **for** *f*
using *lincomb-closed A*
by (*metis* (*full-types*) *carrier-matD*(*1*) *cols-dim lincomb-closed*)
moreover have $\exists x \in$ *carrier-vec* (*dim-col A*). $A *_v x$ = *lincomb* (*of-int* $\circ$ *f*) (*set* (*cols A*)) **for** *f*
proof −
let *?g* = (λv. *of-int* (*f v*))
let *?g'* = (λi. *if* $\exists j<i$. *?ws* ! *i* = *?ws* ! *j then 0 else ?g* (*?ws* ! *i*))
have *lincomb* (*of-int* $\circ$ *f*) (*set* (*cols A*)) = *lincomb ?g* (*set ?ws*) **unfolding** *o-def* **by** *auto*
also have ... = *lincomb-list ?g' ?ws*
by (*rule lincomb-as-lincomb-list*[*OF set-cols-in*])

also have ... = *mat-of-cols n ?ws* $*_v$ *vec* (*length ?ws*) *?g'*
by (*rule lincomb-list-as-mat-mult*, *insert set-cols-in A*, *auto*)
also have ... = $A *_v$ (*vec* (*length ?ws*) *?g'*) **using** *mat-of-cols-cols A* **by** *auto*
finally show *?thesis* **by** *auto*
qed
moreover have $\exists f.\ A *_v x = $ *lincomb* (*of-int* $\circ$ *f*) (*set* (*cols A*))
if *Ax*: $A *_v x \in$ *carrier-vec* (*dim-row A*) **and** *x*: $x \in$ *carrier-vec* (*dim-col A*) **for** *x*
proof −
let *?c* = $\lambda i.\ x \$ i$
have *x-vec*: *vec* (*length ?ws*) *?c* = *x* **using** *x* **by** *auto*
have $A *_v x$ = *mat-of-cols n ?ws* $*_v$ *vec* (*length ?ws*) *?c* **using** *mat-of-cols-cols A x-vec* **by** *auto*
also have ... = *lincomb-list ?c ?ws*
by (*rule lincomb-list-as-mat-mult*[*symmetric*], *insert set-cols-in A*, *auto*)
also have ... = *lincomb* (*mk-coeff ?ws ?c*) (*set ?ws*)
by (*rule lincomb-list-as-lincomb*, *insert set-cols-in A*, *auto*)
finally show *?thesis* **by** *auto*
qed
ultimately show *?thesis* **unfolding** *lattice-of-altdef-lincomb*[*OF set-cols-in*]
by (*metis* (*mono-tags*, *opaque-lifting*))
qed

corollary *lattice-of-as-mat-mult*:
assumes *fs*: *set fs* $\subseteq$ *carrier-vec n*
shows *lattice-of fs* = {*y*∈*carrier-vec n*. ∃ *x*∈*carrier-vec* (*length fs*). (*mat-of-cols n fs*) $*_v$ *x* = *y*}
proof −
have *cols-eq*: *cols* (*mat-of-cols n fs*) = *fs* **using** *cols-mat-of-cols*[*OF fs*] **by** *simp*
have *m*: (*mat-of-cols n fs*) ∈ *carrier-mat n* (*length fs*) **using** *mat-of-cols-carrier*(*1*) **by** *auto*
show *?thesis* **using** *lattice-of-cols-as-mat-mult*[*OF m*] **unfolding** *cols-eq* **using** *m* **by** *auto*
qed
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*
 unfolding *corthogonal-def* **by** *auto*

lemma *corthogonalI*[*intro*]:
 (⋀*i j*. *i* < *length vs* ⟹ *j* < *length vs* ⟹ *vs* ! *i* ·*c vs* ! *j* = *0* ⟷ *i* ≠ *j*) ⟹
 corthogonal vs
 unfolding *corthogonal-def* **by** *auto*

lemma *corthogonal-distinct*: *corthogonal us* ⟹ *distinct us*
proof (*induct us*)
 case (*Cons u us*)
  have *u* ∉ *set us*
  proof
   assume *u* : *set us*
   then obtain *j* **where** *uj*: *u* = *us*!*j* **and** *j*: *j* < *length us*
    using *in-set-conv-nth* **by** *metis*
   hence *j′*: *j*+*1* < *length* (*u*#*us*) **by** *auto*
   have *u* ·*c us*!*j* = *0*
    using *corthogonalD*[*OF Cons*(*2*) - *j′*,*of 0*] **by** *auto*
   hence *u* ·*c u* = *0* **using** *uj* **by** *simp*
   thus *False* **using** *corthogonalD*[*OF Cons*(*2*),*of 0 0*] **by** *auto*
  qed
  moreover have *distinct us*
  proof (*rule Cons*(*1*),*intro corthogonalI*)
   fix *i j* **assume** *i* < *length* (*us*) *j* < *length* (*us*)
   hence *len*: *i*+*1* < *length* (*u*#*us*) *j*+*1* < *length* (*u*#*us*) **by** *auto*
   show (*us*!*i* ·*c us*!*j* = *0*) = (*i*≠*j*)
    using *corthogonalD*[*OF Cons*(*2*) *len*] **by** *simp*
  qed
  ultimately show *?case* **by** *simp*
qed *simp*

lemma *corthogonal-sort*:
 assumes *dist′*: *distinct us′*
   and *mem*: *set us* = *set us′*
 shows *corthogonal us* ⟹ *corthogonal us′*
proof

assume *orth*: *corthogonal us*
hence *dist*: *distinct us* **using** *corthogonal-distinct* **by** *auto*
fix i' j' **assume** i': $i' <$ *length us'* **and** j': $j' <$ *length us'*
obtain *i* **where** *ii'*: $us!i = us'!i'$ **and** *i*: $i <$ *length us*
using *mem i' in-set-conv-nth* **by** *metis*
obtain *j* **where** *jj'*: $us!j = us'!j'$ **and** *j*: $j <$ *length us*
using *mem j' in-set-conv-nth* **by** *metis*
from *corthogonalD*[*OF orth i j*]
have $(us!i \cdot c\ us!j = 0) = (i \neq j)$**.**
hence $(us'!i' \cdot c\ us'!j' = 0) = (i \neq j)$ **using** *ii' jj'* **by** *auto*
also have ... $= (us!i \neq us!j)$ **using** *nth-eq-iff-index-eq dist i j* **by** *auto*
also have ... $= (us'!i' \neq us'!j')$ **using** *ii' jj'* **by** *auto*
also have ... $= (i' \neq j')$ **using** *nth-eq-iff-index-eq dist' i' j'* **by** *auto*
finally show $(us'!i' \cdot c\ us'!j' = 0) = (i' \neq j')$**.**
qed

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*) = $-(w \cdot c\ u)/(u \cdot c\ u) \cdot_v u$ + *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*
by (*induct ws arbitrary*:*us*, *auto simp*:*Let-def*)

lemma *gram-schmidt-code*[*code*]:
gram-schmidt n ws = *gram-schmidt-sub2 n* [] *ws*
unfolding *gram-schmidt-def*
apply(*subst gram-schmidt-sub-eq*) **by** *simp*

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* ($\lambda u. -(w \cdot c\ u)/(u \cdot c\ u) \cdot_v u$) (*set us*)
  using *assms*
proof (*induct us*)
  case *Cons* **show** *?case* **unfolding** *set-simps*
  by (*subst finsum-insert*[*OF finite-set*], *insert Cons*, *auto*)
qed *simp*

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* ($\lambda u. -(w \cdot c\ u)/(u \cdot c\ u)$) (*set us*)
    (**is** - = *lincomb ?a* -)
  using *us dist* **unfolding** *lincomb-def*
proof (*induct us*)
  case (*Cons u us*)
    let *?f* = $\lambda u.$ *?a u* $\cdot_v$ *u*
    have *?f* : (*set us*) → *carrier-vec n* **and** *?f u* : *carrier-vec n* **using** *w Cons* **by**
auto
    moreover have *u* ∉ *set us* **using** *Cons* **by** *auto*
    ultimately show *?case*
      unfolding *adjuster.simps*
      unfolding *set-simps*
      using *finsum-insert*[*OF finite-set*] *Cons* **by** *auto*
qed *simp*

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*)
  using *adjuster-lincomb*[*OF assms*]
  unfolding *finite-span*[*OF finite-set us*] **by** *auto*

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*
  using *adjuster-in-span span-closed assms* **by** *auto*

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*)
  using *span-add*[*OF us adjuster-in-span*[*OF w us dist*] *w*]
  using *comm-add-vec ind* **by** *auto*

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*
  using *adjust-not-in-span*[*OF assms*] *span-mem*[*OF us*] **by** *auto*

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 −
  let *?a* = λ*u*. −(*w* ·*c u*)/(*u* ·*c u*)
  have *?v* = *lincomb ?a* (*set us*) **using** *adjuster-lincomb*[*OF assms*].
  hence *vU*: *?v* : *span* (*set us*) **unfolding** *finite-span*[*OF finite-set us*] **by** *auto*
  hence *v*[*simp*]: *?v* : *carrier-vec n* **using** *span-closed*[*OF us*] **by** *auto*
  have *vU'*: *?v* : *span ?U* **using** *vU span-is-monotone*[*OF subset-insertI*] **by** *auto*

  have {*w*} ⊆ *?U* **by** *simp*
  from *span-is-monotone*[*OF this*]
  have *wU'*: *w* : *span ?U* **using** *span-self*[*OF w*] **by** *auto*

  have *?U* ⊆ *carrier-vec n* **using** *us w* **by** *simp*
  from *span-add*[*OF this wU' v*] *vU' comm-add-vec*[*OF w*]
  show *?thesis* **by** *simp*
qed

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 −
  have *v*: *?v* : *carrier-vec n* **using** *assms* **by** *auto*
  have *?v* ∉ *span* (*set us*)

using *adjust-not-in-span*[*OF w us dist wus*]
using *comm-add-vec*[*OF adjuster-carrier*[*OF w us dist*] *w*] **by** *auto*
thus *?thesis*
using *lin-dep-iff-in-span*[*OF us ind v*] *adjust-not-mem*[*OF w us dist wus*] **by** *auto*
qed

lemma *adjust-preserves-span*:
assumes *w*[*simp*]: $(w :: 'a\ vec) : carrier\text{-}vec\ n$
and *us*: $set\ (us :: 'a\ vec\ list) \subseteq carrier\text{-}vec\ n$
and *dist*: *distinct us*
shows $w : span\ (set\ us) \longleftrightarrow adjuster\ n\ w\ us + w : span\ (set\ us)$
(**is** - $\longleftrightarrow$ *?v* + - : -)
proof −
have *?v* : *span* (*set us*)
using *adjuster-lincomb*[*OF assms*]
unfolding *finite-span*[*OF finite-set us*] **by** *auto*
hence [*simp*]: *?v* : *carrier-vec n* **using** *span-closed*[*OF us*] **by** *auto*
show *?thesis*
using *span-add*[*OF us adjuster-in-span*[*OF w us*] *w*] *comm-add-vec*[*OF w*] *dist*
by *auto*
qed

lemma *in-span-adjust*:
assumes *w*[*simp*]: $(w :: 'a\ vec) : carrier\text{-}vec\ n$
and *us*: $set\ (us :: 'a\ vec\ list) \subseteq carrier\text{-}vec\ n$
and *dist*: *distinct us*
shows $w : span\ (insert\ (adjuster\ n\ w\ us + w)\ (set\ us))$
(**is** - : *span* (*insert ?v* -))
proof −
have *v*: *?v* : *carrier-vec n* **using** *assms* **by** *auto*
have *a*[*simp*]: *adjuster n w us* : *carrier-vec n*
and *neg*: − *adjuster n w us* : *carrier-vec n* **using** *assms* **by** *auto*
hence *vU*: *insert ?v* (*set us*) $\subseteq$ *carrier-vec n* **using** *us* **by** *auto*
have *aS*: *adjuster n w us* : *span* (*insert ?v* (*set us*))
using *adjuster-in-span*[*OF w us*] *span-is-monotone*[*OF subset-insertI*] *dist*
by *auto*
have *negS*: − *adjuster n w us* : *span* (*insert ?v* (*set us*))
using *span-neg*[*OF vU aS*] *us* **by** *simp*
have [*simp*]:− *adjuster n w us* + (*adjuster n w us* + *w*) = *w*
unfolding *a-assoc*[*OF neg a w*,*symmetric*] **by** *simp*
have {*?v*} $\subseteq$ *insert ?v* (*set us*) **by** *simp*
from *span-is-monotone*[*OF this*]
have *vS*: *?v* : *span* (*insert ?v* (*set us*)) **using** *span-self*[*OF v*] **by** *auto*
thus *?thesis* **using** *span-add*[*OF vU negS v*] **by** *auto*
qed

lemma *adjust-zero*:
assumes *U*: $set\ (us :: 'a\ vec\ list) \subseteq carrier\text{-}vec\ n$

```
    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 −
  define u where u = us!i
  have u[simp]: u : carrier-vec n using i U u-def by auto
  hence cu[simp]: conjugate u : carrier-vec n by auto
  have uU: u : set us using i u-def by auto
  let ?g = λu'::'a vec. (−(w ·c u')/(u' ·c u') ·v u')
  have g: ?g : set us → carrier-vec n using w U by auto
  hence carrier: finsum V ?g (set us) : carrier-vec n by simp
  let ?f = λu'. ?g u' ·c u
  let ?U = set us − {u}
  { fix u' assume u': (u'::'a vec) : carrier-vec n
    have [simp]: dim-vec u = n by auto
    have ?f u' = (− (w ·c u') / (u' ·c u')) ∗ (u' ·c u)
      using scalar-prod-smult-left[of u' conjugate u]
      unfolding carrier-vecD[OF u] carrier-vecD[OF u'] by auto
  } note conv = this
  have ?f : ?U → {0}
  proof (intro Pi-I)
    fix u' assume u'Uu: u' : set us − {u}
    hence u'U: u' : set us by auto
    hence u'[simp]: u' : carrier-vec n using U by auto
    obtain j where j: j < length us and u'j: u' = us ! j
      using u'U in-set-conv-nth by metis
    have i ≠ j using u'Uu u'j u-def by auto
    hence u' ·c u = 0
      unfolding u'j using corthogonalD[OF orth j i] u-def by auto
    hence ?f u' = 0 using mult-zero-right conv[OF u'] by auto
    thus ?f u' : {0} by auto
  qed
  hence restrict ?f ?U = restrict (λu. 0) ?U by force
  hence sum ?f ?U = sum (λu. 0) ?U
    by (intro R.finsum-restrict, auto)
  hence fU'0: sum ?f ?U = 0 by auto
  have uU': u ∉ ?U by auto
  have set us = insert u ?U
    using insert-Diff-single uU by auto
  hence sum ?f (set us) = ?f u + sum ?f ?U
    using R.finsum-insert[OF - uU'] by auto
  also have ... = ?f u using fU'0 by auto
  also have ... = − (w ·c u) / (u ·c u) ∗ (u ·c u)
    using conv[OF u] by auto
  finally have main: sum ?f (set us) = − (w ·c u)
    unfolding u-def
    by (simp add: i orth corthogonalD)
  show ?thesis
```

unfolding *u-def*[*symmetric*]
unfolding *adjuster-finsum*[*OF U corthogonal-distinct*[*OF orth*]]
unfolding *add-scalar-prod-distrib*[*OF carrier w cu*]
unfolding *finsum-scalar-prod-sum*[*OF g cu*]
unfolding *main*
unfolding *comm-scalar-prod*[*OF cu w*]
using *left-minus* **by** *auto*
qed

lemma *adjust-nonzero*:
assumes *U*: *set* (*us* :: *'a vec list*) $\subseteq$ *carrier-vec n*
and *dist*: *distinct us*
and *w*[*simp*]: *w* : *carrier-vec n*
and *wsU*: $w \notin$ *span* (*set us*)
shows *adjuster n w us* + $w \neq 0_v$ *n* (**is** *?a* + - $\neq$ -)
proof
have [*simp*]: *?a* : *carrier-vec n* **using** *U dist* **by** *auto*
have [*simp*]: $-$ *?a* : *carrier-vec n* **by** *auto*
have [*simp*]: *?a* + *w* : *carrier-vec n* **by** *auto*
assume *?a* + *w* = 0_v *n*
hence $-$ *?a* = $-$ *?a* + (*?a* + *w*) **by** *auto*
also have ... = ($-$ *?a* + *?a*) + *w* **apply**(*subst a-assoc*) **by** *auto*
also have $-$ *?a* + *?a* = 0_v *n* **using** *r-neg*[*OF w*] **unfolding** *vec-neg*[*OF w*] **by** *auto*
finally have $-$ *?a* = *w* **by** *auto*
moreover have $-$ *?a* : *span* (*set us*)
using *span-neg*[*OF U adjuster-in-span*[*OF w U dist*]] **by** *auto*
ultimately show *False* **using** *wsU* **by** *auto*
qed

lemma *adjust-orthogonal*:
assumes *U*: *set* (*us* :: *'a vec list*) $\subseteq$ *carrier-vec n*
and *orth*: *corthogonal us*
and *w*[*simp*]: *w* : *carrier-vec n*
and *wsU*: $w \notin$ *span* (*set us*)
shows *corthogonal* ((*adjuster n w us* + *w*) # *us*)
(**is** *corthogonal* (*?aw* # -))
proof
have *dist*: *distinct us* **using** *corthogonal-distinct orth* **by** *auto*
have *aw*[*simp*]: *?aw* : *carrier-vec n* **using** *U dist* **by** *auto*
note *adjust-nonzero*[*OF U dist w*] *wsU*
hence *aw0*: *?aw* ·c *?aw* $\neq$ *0* **using** *conjugate-square-eq-0-vec*[*OF aw*] **by** *auto*
fix *i j* **assume** *i*: *i* < *length* (*?aw* # *us*) **and** *j*: *j* < *length* (*?aw* # *us*)
show ((*?aw* # *us*) ! *i* ·c (*?aw* # *us*) ! *j* = *0*) = (*i* $\neq$ *j*)
proof (*cases i* = *0*)
case *True* **note** *i0* = *this*
show *?thesis*
proof (*cases j* = *0*)
case *True* **show** *?thesis* **unfolding** *True i0* **using** *aw0* **by** *auto*

```
      next case False
        define j' where j' = j−1
        hence jfold: j = j'+1 using False by auto
        hence j': j' < length us using j by auto
        show ?thesis unfolding i0 jfold
          using adjust-zero[OF U orth w j'] by auto
    qed
  next case False
    define i' where i' = i−1
    hence ifold: i = i'+1 using False by auto
    hence i': i' < length us using i by auto
    have [simp]: us ! i' : carrier-vec n using U i' by auto
    hence cu': conjugate (us ! i') : carrier-vec n by auto
    show ?thesis
    proof (cases j = 0)
      case True
        { assume ?aw ·c us ! i' = 0
          hence conjugate (?aw ·c us ! i') = 0 using conjugate-zero by auto
          hence conjugate ?aw · us ! i' = 0
            using conjugate-sprod-vec[OF aw cu'] by auto
        }
        thus ?thesis unfolding True ifold
        using adjust-zero[OF U orth w i']
        by (subst comm-scalar-prod[of - n], auto)
      next case False
        define j' where j' = j−1
        hence jfold: j = j'+1 using False by auto
        hence j': j' < length us using j by auto
        show ?thesis
          unfolding ifold jfold
          using orth i' j' by (auto simp: corthogonalD)
     qed
  qed
qed

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 (cases w : span (set us))
  case True
    hence ?v : span (set us)
      using adjust-preserves-span[OF assms] by auto
    thus ?thesis using already-in-span[OF us] True by auto next
  case False show ?thesis
    proof
      have wU: ?wU ⊆ carrier-vec n using us by simp
```

```
      have vswU: ?v : span ?wU using adjust-in-span[OF assms] by auto
      hence v: ?v : carrier-vec n using span-closed[OF wU] by auto
      have wsvU: w : span (insert ?v (set us)) using in-span-adjust[OF assms].
      show span ?wU ⊆ span (set (?v # us))
        using span-swap[OF finite-set us w False v wsvU] by auto
      have ?v ∉ span (set us)
        using False adjust-preserves-span[OF assms] by auto
      thus span (set (?v # us)) ⊆ span ?wU
        using span-swap[OF finite-set us v - w] vswU by auto
    qed
qed

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
  using assms
proof (induct ws arbitrary: us us')
case (Cons w ws)
  let ?v = adjuster n w us
  have wW[simp]: set (w#ws) ⊆ carrier-vec n using Cons by simp
  hence W[simp]: set ws ⊆ carrier-vec n
   and w[simp]: w : carrier-vec n by auto
  have U[simp]: set us ⊆ carrier-vec n using Cons by simp
  have UW: set (us@ws) ⊆ carrier-vec n by simp
  have wU: set (w#us) ⊆ carrier-vec n by simp
  have dist: distinct (us @ w # ws) using Cons by simp
  hence dist-U: distinct us
    and dist-W: distinct ws
    and dist-UW: distinct (us @ ws)
    and w-U: w ∉ set us
    and w-W: w ∉ set ws
    and w-UW: w ∉ set (us @ ws) by auto
  have ind: ~ lin-dep (set (us @ w # ws)) using Cons by simp
  have ind-U: ~ lin-dep (set us)
    and ind-W: ~ lin-dep (set ws)
    and ind-wU: ~ lin-dep (insert w (set us))
    and ind-UW: ~ lin-dep (set (us @ ws))
    by (subst subset-li-is-li[OF ind];auto)+
  have corth: corthogonal us using Cons by simp
  have U'def: gram-schmidt-sub n ((?v + w)#us) ws = us' using Cons by simp
```

have *v*: *?v* : *carrier-vec n* **using** *dist-U* **by** *auto*
hence *vw*: *?v* + *w* : *carrier-vec n* **by** *auto*
hence *vwU*: *set* ((*?v* + *w*) # *us*) ⊆ *carrier-vec n* **by** *auto*
have *vsU*: *?v* : *span* (*set us*) **using** *adjuster-in-span*[*OF w*] *dist* **by** *auto*
hence *vsUW*: *?v* : *span* (*set* (*us* @ *ws*))
using *span-is-monotone*[*of set us set* (*us*@*ws*)] **by** *auto*
have *wsU*: *w* ∉ *span* (*set us*)
using *lin-dep-iff-in-span*[*OF U ind-U w w-U*] *ind-wU* **by** *auto*
hence *vwU*: *?v* + *w* ∉ *span* (*set us*) **using** *adjust-not-in-span*[*OF w U dist-U*] **by** *auto*

have *w* ∉ *span* (*set* (*us*@*ws*)) **using** *lin-dep-iff-in-span*[*OF - ind-UW*] *dist ind* **by** *auto*
hence *span*: *?v* + *w* ∉ *span* (*set* (*us*@*ws*)) **using** *span-add*[*OF UW vsUW w*] **by** *auto*
hence *vwUS*: *?v* + *w* ∉ *set* (*us* @ *ws*) **using** *span-mem* **by** *auto*
hence *ind2*: ~ *lin-dep* (*set* (((*?v* + *w*) # *us*) @ *ws*))
using *lin-dep-iff-in-span*[*OF UW ind-UW vw*] *span* **by** *auto*

have *vwU*: *set* ((*?v* + *w*) # *us*) ⊆ *carrier-vec n* **using** *U w dist* **by** *auto*
have *dist2*: *distinct* (((*?v* + *w*) # *us*) @ *ws*) **using** *dist vwUS* **by** *simp*

have *orth2*: *corthogonal* ((*adjuster n w us* + *w*) # *us*)
using *adjust-orthogonal*[*OF U corth w wsU*].

show *?case*
using *Cons*(*1*)[*OF U'def W vwU dist2 ind2*] *orth2*
using *span-Un*[*OF vwU wU gram-schmidt-sub-span*[*OF w U dist-U*] *W W*] **by** *auto*

qed *simp*

lemma *gram-schmidt-hd* [*simp*]:
assumes [*simp*]: *w* : *carrier-vec n* **shows** *hd* (*gram-schmidt n* (*w*#*ws*)) = *w*
unfolding *gram-schmidt-code* **by** *simp*

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 −
have *main*: *gram-schmidt-sub n* [] *ws* = *rev us*
using *us* **unfolding** *gram-schmidt-def*

```
    using gram-schmidt-sub-eq by auto
  have orth: corthogonal [] by auto
  have span (set ws) = span (set (rev us))
   and orth2: corthogonal (rev us)
   and set us ⊆ carrier-vec n
   and length us = length ws
   and dist: distinct us
    using gram-schmidt-sub-result[OF main ws]
    by (auto simp: assms orth)
  thus span (set ws) = span (set us) by simp
  show set us ⊆ carrier-vec n by fact
  show length us = length ws by fact
  show distinct us by fact
  show corthogonal us
    using corthogonal-distinct[OF orth2] unfolding distinct-rev
    using corthogonal-sort[OF - set-rev orth2] by auto
qed
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) ·v conjugate v

lemma vec-inv-closed[simp]: v ∈ carrier-vec n ⟹ vec-inv v ∈ carrier-vec n
  unfolding vec-inv-def by auto
```

lemma *vec-inv-dim*[*simp*]: *dim-vec* (*vec-inv v*) = *dim-vec v*
  unfolding *vec-inv-def* **by** *auto*

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 −
  { **assume** *v* ·*c v* = *0*
    hence *v* = 0_v *n* **using** *conjugate-square-eq-0-vec*[*OF v*] **by** *auto*
    hence *False* **using** *v0* **by** *auto*
  }
  moreover have *conjugate v* · *v* = *v* ·*c v*
    apply (*rule comm-scalar-prod*) **using** *v* **by** *auto*
  ultimately show *?thesis*
    unfolding *vec-inv-def*
    apply (*subst smult-scalar-prod-distrib*)
    using *assms* **by** *auto*
qed

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 −
  define *l* **where** *l* = *length vs*
  have *rW*[*simp*]: *dim-row ?W* = *l* **using** *l-def* **by** *auto*
  have *cV*[*simp*]:*dim-col ?V* = *l* **using** *l-def* **by** *auto*
  have *dim*: ⋀*i*. *i* < *length vs* ⟹ *vs*!*i* ∈ *carrier-vec n* **using** *V* **by** *auto*
  show *?thesis*
    unfolding *inverts-mat-def*
    apply *rule*
    unfolding *mat-of-rows-carrier length-map l-def*[*symmetric*]
    unfolding *index-one-mat*
  proof −
    show *dim-row* (*?W* ∗ *?V*) = *l dim-col* (*?W* ∗ *?V*) = *l*
      unfolding *times-mat-def rW cV* **by** *auto*
    fix *i j* **assume** *i*:*i*<*l* **and** *j*: *j*<*l*
    hence *i2*: *i*<*length vs*
      and *i3*: *i*<*length* (*map vec-inv vs*)
      and *j2*: *j*<*length vs* **using** *l-def* **by** *auto*
    hence *id2*: *vs* ! *i* ∈ *carrier-vec n*
      and *id3*: *map vec-inv vs* ! *i* ∈ *carrier-vec n*
      and *id4*: *conjugate* (*vs* ! *i*) ∈ *carrier-vec n*
      and *jd2*: *vs* ! *j* ∈ *carrier-vec n* **using** *dim* **by** *auto*
    show (*?W* ∗ *?V*) $$ (*i*,*j*) = (*if i* = *j then 1 else 0*)
      unfolding *times-mat-def rW cV*

```
    unfolding index-mat[OF i j] split
    unfolding mat-of-rows-row[OF i3 id3]
    unfolding col-mat-of-cols[OF j2 jd2]
    unfolding nth-map[OF i2]
    unfolding vec-inv-def
    unfolding smult-scalar-prod-distrib[OF id4 jd2]
    unfolding comm-scalar-prod[OF id4 jd2]
    using corthogonalD[OF orth j2 i2] by auto
  qed
qed

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
  have ci: col A i : carrier-vec (dim-row A)
   and cj: col A j : carrier-vec (dim-row A) by auto
  note [simp] = conjugate-conjugate-sprod[OF ci cj]

  let ?B = mat-adjoint A * A
  have diag: diagonal-mat ?B and zero: ⋀i. i<dim-col A ⟹ ?B $$ (i,i) ≠ 0
    using orth unfolding corthogonal-mat-def Let-def by auto
  { assume i = j
    hence conjugate (col A i) · col A j ≠ 0
      using zero[OF i] unfolding mat-adjoint-def using i by simp
    hence conjugate (conjugate (col A i) · col A j) ≠ 0
      unfolding conjugate-zero-iff.
    hence col A i ·c col A j ≠ 0 by simp
  }
  thus col A i ·c col A j = 0 ⟹ i ≠ j by auto
  { assume i ≠ j
    hence conjugate (col A i) · col A j = 0
      using diag
      unfolding diagonal-mat-def
      unfolding mat-adjoint-def using i j by simp
    hence conjugate (conjugate (col A i) · col A j) = 0 by simp
```

```
    thus col A i ∙c col A j = 0 by simp
  }
qed

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 –
  { fix i j assume i: i < dim-col A and j: j < dim-col A and ij: i ≠ j
    have conjugate (col A i) ∙ col A j = 0
      by (metis assms col-dim i j ij conjugate-vec-sprod-comm)
  }
  moreover
  { fix i assume i < dim-col A
    hence conjugate (col A i) ∙ col A i ≠ 0
      by (metis assms comm-scalar-prod carrier-vec-conjugate carrier-vecI)
  }
  ultimately show ?thesis
  unfolding corthogonal-mat-def Let-def
  unfolding diagonal-mat-def
  unfolding mat-adjoint-def by auto
qed

lemma corthogonal-inv-result:
  assumes o: corthogonal-mat (A::'a::conjugatable-field mat)
  shows inverts-mat (corthogonal-inv A) A
proof –
  have oc: corthogonal (cols A)
    apply (intro corthogonalI) using corthogonal-matD[OF o] by auto
  show ?thesis unfolding corthogonal-inv-def
    using corthogonal-inv[OF oc cols-dim] by auto
qed
```

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 ≠ 0v 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 −
let *?b* = *basis-completion* *v*
note *d* = *basis-completion-def* *Let-def*
from *v* **have** *dim*: *dim-vec* *v* = *n* **by** *auto*
let *?is* = [*i* . *i* <− [*0*..<*n*], *v* $ *i* ≠ *0*]
{
assume *empty*: *set* *?is* = {}
have *v* = 0_v *n*
by (*rule* *eq-vecI*, *insert* *empty* *v*, *auto*)
}
with *v0* **obtain** *k* *ids* **where** *id*: *?is* = *k* # *ids* **and** *mem*: *k* ∈ *set* *?is* **by** (*cases* *?is*, *auto*)
from *mem* **have** *vk*: *v* $ *k* ≠ *0* **and** *k*: *k* < *n* **by** *auto*
{
fix *i*
assume *i*: ¬ *i* < *k*
have *id*: *k* # [*Suc* *k*..<*n*] = [*k* ..< *n*] **using** *k* **by** (*simp* *add*: *upt-conv-Cons*)
from *i* **have** *i* < *n* ⟹ (*k* # [*Suc* *k*..<*n*]) ! (*i* − *k*) = *i*
unfolding *id*
by (*subst* *nth-upt*, *auto*)
}
hence *split*: [*0* ..< *n*] = [*0* ..< *k*] @ *k* # [*Suc* *k* ..< *n*]
by (*intro* *nth-equalityI*, *insert* *k*, *auto* *simp*: *nth-append*)
{
fix *as*
assume *k* ∉ *set* *as*
hence [*unit-vec* *n* *i*. *i* <− *as*, *i* ≠ *k*] = [*unit-vec* *n* *i*. *i* <− *as*]
by (*induct* *as*, *auto*)
} **note** *conv* = *this*
have *b-all*: *?b* = *v* # [*unit-vec* *n* *i*. *i* <− [*0*..<*n*], *i* ≠ *k*]
unfolding *d* *dim* *id* **by** *simp*
also have [*unit-vec* *n* *i*. *i* <− [*0*..<*n*], *i* ≠ *k*] = [*unit-vec* *n* *i*. *i* <− [*0*..<*k*]] @ [*unit-vec* *n* *i*. *i* <− [*Suc* *k*..<*n*]]
unfolding *split* **by** (*auto* *simp*: *conv*)
finally have *b*: *?b* = *v* # [*unit-vec* *n* *i*. *i* <− [*0*..<*k*]] @ [*unit-vec* *n* *i*. *i* <− [*Suc* *k*..<*n*]] **by** *simp*
show *carr*: *set* *?b* ⊆ *carrier-vec* *n* (**is** *?S* ⊆ -)
unfolding *b* **using** *assms* **by** *auto*
show *hd* *?b* = *v* **unfolding** *b* **by** *auto*
show *len*: *length* (*basis-completion* *v*) = *n* **unfolding** *b* **using** *k*
by *auto*
define *I* **where** *I* = (λ *i*. *if* *i* < *k* *then* *i* *else* *Suc* *i*)
have *I*: ⋀ *i*. *I* *i* ≠ *k* ⋀ *i*. *Suc* *i* < *n* ⟹ *I* *i* < *n* **unfolding** *I-def* **by** *auto*
{
fix *i*

```
    assume i: i < n
    have ?b ! i = (if i = 0 then v else unit-vec n (I (i − 1)))
      unfolding b I-def using i
      by (auto split: if-splits simp: nth-append)
  } note bi = this
  show dist: distinct ?b unfolding distinct-conv-nth len
  proof (intro allI impI)
    fix i j
    assume i: i < n and j: j < n and ij: i ≠ j
    show ?b ! i ≠ ?b ! j
    proof
      assume id1: ?b ! i = ?b ! j
      hence id2: ⋀ l. ?b ! i $ l = ?b ! j $ l by auto
      have i = j
      proof (cases i = 0)
        case True
        hence biv: ?b ! i = v unfolding b by simp
          from True ij have bj: ?b ! j = unit-vec n (I (j − 1)) Suc (j − 1) = j
unfolding bi[OF j] by auto
        with id2[of k, unfolded biv bj] vk I[of j − 1] k j
        have False by simp
        thus ?thesis ..
      next
        case False note i0 = this
        hence bi': ?b ! i = unit-vec n (I (i − 1)) Suc (i − 1) = i unfolding bi[OF
i] by auto
        show ?thesis
        proof (cases j = 0)
          case True
          hence bj: ?b ! j = v unfolding b by simp
          from id2[of k, unfolded bi' bj] vk I[of i − 1] k i bi'
          have False by simp
          thus ?thesis by simp
        next
          case False note j0 = this
            hence bj: ?b ! j = unit-vec n (I (j − 1)) Suc (j − 1) = j unfolding
bi[OF j] by auto
          have 1 = ?b ! i $ I (i − 1) unfolding bi' using I[of i − 1] i i0 by auto
          also have ... = unit-vec n (I (j − 1)) $ I (i − 1) unfolding id1 bj by
simp
          also have ... = (if I (i − 1) = I (j − 1) then 1 else 0)
            using I[of i − 1] I[of j − 1] i0 j0 i j by auto
          finally have I (i − 1) = I (j − 1) by (auto split: if-splits)
          with i0 j0 show i = j unfolding I-def by (auto split: if-splits)
        qed
      qed
      thus False using ij by simp
    qed
  qed
```

have *span* (*set ?b*) ⊆ *carrier-vec n* **using** *carr* **by** *auto*
moreover
{
fix *w* :: *'a vec*
assume *w*: *w* ∈ *carrier-vec n*
define *lookup* **where** *lookup* = (*v*,*k*) # [(*unit-vec n i*, *i*). *i* <− [*0*..<*n*], *i* ≠ *k*]
define *a* **where** *a* = (λ *vi*. *case map-of lookup vi of Some i* ⇒ *if i* = *k then w* $ *k* / *v* $ *k else*
w $ *i* − *w* $ *k* / *v* $ *k* ∗ *v* $ *i*)
have *map fst lookup* = *?b* **unfolding** *b-all lookup-def*
by (*auto simp*: *map-concat o-def if-distrib*, *unfold list.simps fst-def prod.simps*, *simp*)
with *dist* **have** *dist*: *distinct* (*map fst lookup*) **by** *simp*
let *?w* = *lincomb a* (*set ?b*)
have *?w* ∈ *carrier-vec n* **using** *carr* **by** *auto*
with *w* **have** *dim*: *dim-vec w* = *n dim-vec ?w* = *n* **by** *auto*
have *w* = *?w*
proof (*rule eq-vecI*; *unfold dim*)
fix *i*
assume *i*: *i* < *n*
show *w* $ *i* = *?w* $ *i* **unfolding** *lincomb-def*
proof (*subst finsum-index*[*OF i - carr*])
show (λ*v*. *a v* $\cdot_v$ *v*) ∈ *set ?b* → *carrier-vec n* **using** *carr* **by** *auto*
{
fix *x* :: *'a vec* **and** *j*
assume *x* = *unit-vec n j j* ≠ *k j* < *n*
hence (*x*,*j*) ∈ *set lookup* **unfolding** *lookup-def* **by** *auto*
from *map-of-is-SomeI*[*OF dist this*]
have *a x* = *w* $ *j* − *w* $ *k* / *v* $ *k* ∗ *v* $ *j* **unfolding** *a-def* **using** ‹*j* ≠ *k*› **by** *auto*
} **note** *a* = *this*
have ($\sum$*x*∈*set ?b*. (*a x* $\cdot_v$ *x*) $ *i*) = (*a v* $\cdot_v$ *v*) $ *i* + ($\sum$*x*∈(*set ?b*) − {*v*}. (*a x* $\cdot_v$ *x*) $ *i*)
by (*rule sum.remove*[*OF finite-set*], *auto simp*: *b*)
also have *a v* = *w* $ *k* / *v* $ *k* **unfolding** *a-def lookup-def* **by** *auto*
also have (. . . $\cdot_v$ *v*) $ *i* = *w* $ *k* / *v* $ *k* ∗ *v* $ *i* **using** *i v* **by** *auto*
finally have ($\sum$*x*∈*set ?b*. (*a x* $\cdot_v$ *x*) $ *i*) = *w* $ *k* / *v* $ *k* ∗ *v* $ *i* + ($\sum$*x*∈(*set ?b*) − {*v*}. (*a x* $\cdot_v$ *x*) $ *i*) **.**
also have . . . = *w* $ *i*
proof (*cases i* = *k*)
case *True*
hence *w* $ *k* / *v* $ *k* ∗ *v* $ *i* = *w* $ *k* **using** *vk* **by** *auto*
moreover have ($\sum$*x*∈(*set ?b*) − {*v*}. (*a x* $\cdot_v$ *x*) $ *i*) = *0* **unfolding** *True*
proof (*rule sum.neutral*, *intro ballI*)
fix *x*
assume *x* ∈ *set ?b* − {*v*}
then obtain *j* **where** *x*: *x* = *unit-vec n j j* ≠ *k j* < *n* **using** *k* **unfolding** *b* **by** *auto*
show (*a x* $\cdot_v$ *x*) $ *k* = *0* **unfolding** *a*[*OF x*] **unfolding** *x* **using** *x k* **by**

```
auto
        qed
        ultimately show ?thesis unfolding True by simp
      next
        case False
        let ?ui = unit-vec n i :: 'a vec
        {
          assume ?ui = v
          from arg-cong[OF this, of λ v. v $ k] vk i k False have False by auto
        }
        hence diff: ?ui ≠ v by auto
        from a[OF refl False] have ai: (a ?ui ·v ?ui) $ i = w $ i − w $ k / v $ k
∗ v $ i
          using i by auto
        have ?ui ∈ set ?b unfolding b-all using False k i by auto
        with diff have mem: unit-vec n i ∈ set ?b − {v} by auto
        have w $ k / v $ k ∗ v $ i + (∑x∈(set ?b) − {v}. (a x ·v x) $ i)
          = w $ i + (∑x∈(set ?b) − {v,?ui}. (a x ·v x) $ i)
          by (subst sum.remove[OF - mem], auto simp: ai intro!: sum.cong)
        also have (∑x∈(set ?b) − {v,?ui}. (a x ·v x) $ i) = 0
          by (rule sum.neutral, unfold b-all, insert i k, auto)
        finally show ?thesis by simp
      qed
      finally show w $ i = (∑x∈set ?b. (a x ·v x) $ i) by simp
    qed
  qed auto
  hence w ∈ span (set ?b) unfolding span-def by auto
 }
 ultimately show span: span (set ?b) = carrier-vec n by blast
 show basis (set ?b)
 proof (rule dim-gen-is-basis[OF finite-set carr span])
  have card (set ?b) = dim using dist len distinct-card unfolding dim-is-n by
blast
  thus card (set ?b) ≤ dim by simp
 qed
 thus ¬ lin-dep (set ?b) unfolding basis-def by auto
qed

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
  fix i j assume i: i < dim-col ?W and j: j < dim-col ?W
  hence [simp]: ws ! i : carrier-vec n ws ! j : carrier-vec n
   using W len by auto
  have i < length ws and j < length ws using i j using len W by auto
  thus col ?W i ·c col ?W j = 0 ⟷ i ≠ j
```

using *orth*
unfolding *corthogonal-def*
by *simp*
qed

lemma *corthogonal-col-ev-0*: **fixes** $A :: 'a :: conjugatable\text{-}ordered\text{-}field\ mat$
assumes *A*: $A \in carrier\text{-}mat\ n\ n$
and *v*: $v \in carrier\text{-}vec\ n$
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*: $set\ ws \subseteq carrier\text{-}vec\ n\ corthogonal\ ws\ length\ ws = n$
defines $W == mat\text{-}of\text{-}cols\ n\ ws$
defines $W' == corthogonal\text{-}inv\ W$
defines $A' == W' * A * W$
shows $col\ A'\ 0 = vec\ n\ (\lambda\ i.\ if\ i = 0\ then\ e\ else\ 0)$
proof −
let $?f = (\lambda\ i.\ if\ i = 0\ then\ e\ else\ 0)$
from *ws* **have** *W*: $W \in carrier\text{-}mat\ n\ n$ **unfolding** *W-def* **by** *auto*
from *W* **have** *W'*: $W' \in carrier\text{-}mat\ n\ n$ **unfolding** *W'-def*
corthogonal-inv-def mat-of-rows-def **by** *auto*
from *A W W'* **have** *A'*: $A' \in carrier\text{-}mat\ n\ n$ **unfolding** *A'-def* **by** *auto*
show $col\ A'\ 0 = vec\ n\ ?f$
proof (*rule*,*unfold dim-vec*)
show *dim*: $dim\text{-}vec\ (col\ A'\ 0) = n$ **using** *A'* **by** *simp*
have *row0*: $vec\text{-}inv\ v \cdot (A *_v v) = e$
using *scalar-prod-smult-distrib*[*OF vec-inv-closed*[*OF v*] *v*]
using *vec-inv*[*OF v v0*] **by** *auto*
fix *i* **assume** *i*: $i < n$
hence *i2*: $i < length\ ws$ **using** *ws* **by** *auto*
let $?wsi = ws\ !\ i$
have *z*: $0 < dim\text{-}col\ A'$ **using** *A' n* **by** *auto*
hence *z2*: $0 < length\ ws$ **using** *A' ws* **by** *auto*
have *wsi*[*simp*]: $ws!i : carrier\text{-}vec\ n$ **using** *ws i* **by** *auto*
hence *ws0*[*simp*]: $ws!0 = v$ **using** *hd-conv-nth*[*symmetric*] *hdws z2* **by** *auto*
have $col\ A'\ 0\ \$\ i = A'\ \$\$\ (i,\ 0)$ **using** *A' i* **by** *auto*
also have $... = (W' * (A * W))\ \$\$\ (i,\ 0)$ **unfolding** *A'-def* **using** *W' A W*
by *auto*
also have $... = row\ W'\ i \cdot col\ (A * W)\ 0$
apply (*subst index-mult-mat*) **using** *W W' A i* **by** *auto*
also have $row\ W'\ i = vec\text{-}inv\ ?wsi$
unfolding *W'-def W-def* **unfolding** *corthogonal-inv-def* **using** *i ws* **by** *auto*
also have $col\ (A * W)\ 0 = A *_v col\ W\ 0$ **using** *A W z A'* **by** *auto*
also have $col\ W\ 0 = v$ **unfolding** *W-def* **using** *z2 ws0 n col-mat-of-cols v* **by**
blast
also have $A *_v v = e \cdot_v v$ **using** *eigen*.
also have $vec\text{-}inv\ ?wsi \cdot (e \cdot_v v) = e * (vec\text{-}inv\ ?wsi \cdot v)$
using *scalar-prod-smult-distrib*[*OF vec-inv-closed*[*OF wsi*] *v*].

```
  also have ... = ?f i
  proof(cases i = 0)
    case True thus ?thesis using vec-inv[OF v v0] by simp
  next
    case False
    hence z: 0 < length ws using i ws by auto
    note cwsi = carrier-vec-conjugate[OF wsi]
    have vec-inv ?wsi · v = 1 / (?wsi ·c ?wsi) * (conjugate ?wsi · v)
      unfolding vec-inv-def unfolding smult-scalar-prod-distrib[OF cwsi v]..
    also have conjugate ?wsi · v = v ·c ?wsi
      using comm-scalar-prod[OF cwsi v].
    also have ... = 0
      using corthogonalD[OF ws(2) z i2] False unfolding ws0 by auto
    finally show ?thesis using False by auto
  qed
  also have ... = vec n ?f $ i using i by simp
  finally show col A' 0 $ i = vec n ?f $ i .
 qed
qed
```

Schur decomposition

```
fun schur-decomposition :: 'a::conjugatable-field mat ⇒ 'a list ⇒ 'a mat × 'a mat
× 'a mat where
  schur-decomposition A [] = (A, 1m (dim-row A), 1m (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 = (0m 1 n1);
      z-col = (0m n1 1);
      one-1 = 1m 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
  using assms
proof (induct es arbitrary: n A B P Q)
```

case *Nil*
with *degree-monic-char-poly*[*of A n*]
show *?case* **by** (*auto intro*: *similar-mat-wit-refl simp*: *diag-mat-def*)
next
case (*Cons e es n A C P Q*)
let *?n1* = *n* − *1*
from *Cons* **have** *A*: *A* ∈ *carrier-mat n n* **and** *dim*: *dim-row A* = *n* **by** *auto*
let *?cp* = *char-poly A*
from *Cons*(*3*)
have *cp*: *?cp* = [: −*e*, *1* :] ∗ (∏ *e* ← *es*. [:− *e*, *1*:]) **by** *auto*
have *mon*: *monic* (∏ *e*← *es*. [:− *e*, *1*:]) **by** (*rule monic-prod-list*, *auto*)
have *deg*: *degree ?cp* = *Suc* (*degree* (∏ *e*← *es*. [:− *e*, *1*:])) **unfolding** *cp*
by (*subst degree-mult-eq*, *insert mon*, *auto*)
with *degree-monic-char-poly*[*OF A*] **have** *n*: *n* ≠ *0* **by** *auto*
define *v* **where** *v* = *find-eigenvector A e*
define *b* **where** *b* = *basis-completion v*
define *ws* **where** *ws* = *gram-schmidt n b*
define *W* **where** *W* = *mat-of-cols n ws*
define *W′* **where** *W′* = *corthogonal-inv W*
define *A′* **where** *A′* = *W′* ∗ *A* ∗ *W*
obtain *A1 A2 A0 A3* **where** *splitA′*: *split-block A′ 1 1* = (*A1*,*A2*,*A0*,*A3*)
by (*cases split-block A′ 1 1*, *auto*)
obtain *B P′ Q′* **where** *schur*: *schur-decomposition A3 es* = (*B*,*P′*,*Q′*)
by (*cases schur-decomposition A3 es*, *auto*)
let *?P′* = *four-block-mat* (1_m *1*) (0_m *1 ?n1*) (0_m *?n1 1*) *P′*
let *?Q′* = *four-block-mat* (1_m *1*) (0_m *1 ?n1*) (0_m *?n1 1*) *Q′*
have *C*: *C* = *four-block-mat A1* (*A2* ∗ *P′*) *A0 B* **and** *P*: *P* = *W* ∗ *?P′* **and** *Q*:
Q = *?Q′* ∗ *W′*
using *Cons*(*4*) **unfolding** *schur-decomposition.simps*
Let-def list.sel dim
v-def[*symmetric*] *b-def*[*symmetric*] *ws-def*[*symmetric*] *W′-def*[*symmetric*] *W-def*[*symmetric*]
A′-def[*symmetric*] *split splitA′ schur* **by** *auto*
have *e*: *eigenvalue A e*
unfolding *eigenvalue-root-char-poly*[*OF A*] *cp* **by** *simp*
from *find-eigenvector*[*OF A e*] **have** *ev*: *eigenvector A v e* **unfolding** *v-def* **.**
from *this*[*unfolded eigenvector-def*]
have *v*[*simp*]: *v* ∈ *carrier-vec n* **and** *v0*: *v* ≠ 0_v *n* **using** *A* **by** *auto*
interpret *cof-vec-space n TYPE*(*′a*) **.**
from *basis-completion*[*OF v v0*, *folded b-def*]
have *span-b*: *span* (*set b*) = *carrier-vec n* **and** *dist-b*: *distinct b*
and *indep*: ¬ *lin-dep* (*set b*) **and** *b*: *set b* ⊆ *carrier-vec n* **and** *hdb*: *hd b* = *v*
and *len-b*: *length b* = *n* **by** *auto*
from *hdb len-b n* **obtain** *vs* **where** *bv*: *b* = *v* # *vs* **by** (*cases b*, *auto*)
from *gram-schmidt-result*[*OF b dist-b indep refl*, *folded ws-def*]
have *ws*: *set ws* ⊆ *carrier-vec n corthogonal ws length ws* = *n*
by (*auto simp*: *len-b*)
from *gram-schmidt-hd*[*OF v*, *of vs*, *folded bv*] **have** *hdws*: *hd ws* = *v* **unfolding**
ws-def **.**
have *orth-W*: *corthogonal-mat W* **using** *orthogonal-mat-of-cols ws* **unfolding**

W-def .
have *W*: $W \in$ *carrier-mat n n*
using *ws* **unfolding** *W-def* **using** *mat-of-cols-carrier(1)[of n ws]* **by** *auto*
have *W'*: $W' \in$ *carrier-mat n n* **unfolding** *W'-def corthogonal-inv-def* **using** *W*
by (*auto simp*: *mat-of-rows-def*)
from *corthogonal-inv-result[OF orth-W]*
have *W'W*: *inverts-mat W' W* **unfolding** *W'-def* .
hence *WW'*: *inverts-mat W W'* **using** *mat-mult-left-right-inverse[OF W' W] W' W*
unfolding *inverts-mat-def* **by** *auto*
have *A'*: $A' \in$ *carrier-mat n n* **using** *W W' A* **unfolding** *A'-def* **by** *auto*
have *A'A-wit*: *similar-mat-wit A' A W' W*
by (*rule similar-mat-witI[of - - n]*, *insert W W' A A' W'W WW'*, *auto simp*: *A'-def*
inverts-mat-def)
hence *A'A*: *similar-mat A' A* **unfolding** *similar-mat-def* **by** *blast*
from *similar-mat-wit-sym[OF A'A-wit]* **have** *simAA'*: *similar-mat-wit A A' W W'* **by** *auto*
have *eigen[simp]*: $A *_v v = e \cdot_v v$ **and** *v0*: $v \neq 0_v\ n$
using *v-def find-eigenvector[OF A e] A*
unfolding *eigenvector-def* **by** *auto*
let *?f* = (λ *i*. *if i = 0 then e else 0*)
have *col0*: *col A' 0 = vec n ?f*
unfolding *A'-def W'-def W-def*
using *corthogonal-col-ev-0[OF A v v0 eigen n hdws ws]*.
from *A' n* **have** *dim-row A' = 1 + ?n1 dim-col A' = 1 + ?n1* **by** *auto*
from *split-block[OF splitA' this]* **have** *A2*: $A2 \in$ *carrier-mat 1 ?n1*
and *A3*: $A3 \in$ *carrier-mat ?n1 ?n1*
and *A'block*: *A' = four-block-mat A1 A2 A0 A3* **by** *auto*
have *A1id*: *A1 = mat 1 1* (λ -. *e*)
using *splitA'[unfolded split-block-def Let-def] arg-cong[OF col0, of* λ *v. v \$ 0] A' n*
by (*auto simp*: *col-def*)
have *A1*: $A1 \in$ *carrier-mat 1 1* **unfolding** *A1id* **by** *auto*
{
fix *i*
assume *i < ?n1*
with *arg-cong[OF col0, of* λ *v. v \$ Suc i] A'*
have *A' \$\$ (Suc i, 0) = 0* **by** *auto*
} **note** *A'0 = this*
have *A0id*: $A0 = 0_m$ *?n1 1*
using *splitA'[unfolded split-block-def Let-def] A'0 A'* **by** *auto*
have *A0*: $A0 \in$ *carrier-mat ?n1 1* **unfolding** *A0id* **by** *auto*
from *cp char-poly-similar[OF A'A]*
have *cp*: *char-poly A'* = [: $-e$,*1* :] $*$ ($\prod$ *e* $\leftarrow$ *es*. [:$-$ *e*, *1*:]) **by** *simp*
also have *char-poly A' = char-poly A1* $*$ *char-poly A3*
unfolding *A'block A0id*
by (*rule char-poly-four-block-zeros-col[OF A1 A2 A3]*)

also have *char-poly A1* = [: $-e$,*1* :]
by (*simp add*: *A1id char-poly-defs det-def sign-def*)
finally have *cp*: *char-poly A3* = ($\prod$ *e* ← *es*. [:− *e*, *1*:])
by (*metis mult-cancel-left pCons-eq-0-iff zero-neq-one*)
from *Cons*(*1*)[*OF A3 cp schur*]
have *simIH*: *similar-mat-wit A3 B P′ Q′* **and** *ut*: *upper-triangular B* **and** *diag*: *diag-mat B* = *es* **by** *auto*
from *similar-mat-witD2*[*OF A3 simIH*]
have *B*: *B* ∈ *carrier-mat ?n1 ?n1* **and** *P′*: *P′* ∈ *carrier-mat ?n1 ?n1* **and** *Q′*: *Q′* ∈ *carrier-mat ?n1 ?n1*
and *PQ′*: *P′* ∗ *Q′* = 1_m *?n1* **by** *auto*
have *A0-eq*: *A0* = *P′* ∗ *A0* ∗ 1_m *1* **unfolding** *A0id* **using** *P′* **by** *auto*
have *simA′C*: *similar-mat-wit A′ C ?P′ ?Q′* **unfolding** *A′block C*
by (*rule similar-mat-wit-four-block*[*OF similar-mat-wit-refl*[*OF A1*] *simIH* - *A0-eq A1 A3 A0*],
insert PQ′ A2 P′ Q′, *auto*)
have *ut1*: *upper-triangular A1* **unfolding** *A1id* **by** *auto*
have *ut*: *upper-triangular C* **unfolding** *C A0id*
by (*intro upper-triangular-four-block*[*OF - B ut1 ut*], *auto simp*: *A1id*)
from *A1id* **have** *diagA1*: *diag-mat A1* = [*e*] **unfolding** *diag-mat-def* **by** *auto*
from *diag-four-block-mat*[*OF A1 B*] **have** *diag*: *diag-mat C* = *e* # *es* **unfolding** *diag diagA1 C* **by** *simp*
from *ut similar-mat-wit-trans*[*OF simAA′ simA′C*, *folded P Q*] *diag*
show *?case* **by** *blast*
qed

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* = ($\prod$ *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 −
let *?B* = *schur-upper-triangular A es*
obtain *C P Q* **where** *schur*: *schur-decomposition A es* = (*C*,*P*,*Q*)
by (*cases schur-decomposition A es*, *auto*)
hence *B*: *B* = *C* **using** *A* **unfolding** *schur-upper-triangular-def B* **by** *auto*
from *schur-decomposition*[*OF A linear schur*]
have *sim*: *similar-mat-wit A B P Q* **and** *B*: *upper-triangular B* **unfolding** *B* **by** *auto*
from *sim* **show** *similar-mat A B* **unfolding** *similar-mat-def* **by** *auto*
from *similar-mat-witD2*[*OF A sim*] **show** *B* ∈ *carrier-mat n n* **by** *auto*
show *upper-triangular B* **by** *fact*
qed

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*
using *schur-upper-triangular*[*OF A linear*] **by** *blast*

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 −
from *linearB* **obtain** *bs* **where** *cB*: *char-poly B* = (∏ *a*←*bs*. [:− *a*, *1*:]) **by** *auto*
from *linearD* **obtain** *ds* **where** *cD*: *char-poly D* = (∏ *a*←*ds*. [:− *a*, *1*:]) **by** *auto*
from *schur-decomposition-exists*[*OF B cB*]
obtain *B*′ *PB QB* **where** *sB*: *schur-decomposition B bs* = (*B*′,*PB*,*QB*)
by (*cases schur-decomposition B bs*, *auto*)
obtain *D*′ *PD QD* **where** *sD*: *schur-decomposition D ds* = (*D*′,*PD*,*QD*)
by (*cases schur-decomposition D ds*, *auto*)
from *schur-decomposition*[*OF B cB sB*] *similar-mat-witD2*[*OF B*, *of B*′] **have**
simB: *similar-mat B B*′ **and** *utB*: *upper-triangular B*′ **and** *diagB*: *diag-mat B*′ = *bs*
and *B*′: *B*′ ∈ *carrier-mat n n*
by (*auto simp*: *similar-mat-def*)
from *schur-decomposition*[*OF D cD sD*] *similar-mat-witD2*[*OF D*, *of D*′] **have**
simD: *similar-mat D D*′ **and** *utD*: *upper-triangular D*′ **and** *diagD*: *diag-mat D*′ = *ds*
and *D*′: *D*′ ∈ *carrier-mat m m*
by (*auto simp*: *similar-mat-def*)
let *?z* = 0_m *m n*
from *similar-mat-four-block-0-ex*[*OF simB simD C B D*, *folded A*]
obtain *B0* **where** *B0*: *B0* ∈ *carrier-mat n m* **and** *sim*: *similar-mat A* (*four-block-mat B*′ *B0 ?z D*′)
by *auto*
let *?block* = *four-block-mat B*′ *B0 ?z D*′
let *?cp* = *char-poly*
let *?prod* = *QB* ∗ *C* ∗ *PD*
let *?diag* = λ *A*. (∏ *a*←*diag-mat A*. [:− *a*, *1*:])
from *char-poly-similar*[*OF sim*] **have** *?cp A* = *?cp ?block* **by** *simp*
also have . . . = *?diag ?block*
by (*rule char-poly-upper-triangular*[*OF four-block-carrier-mat*[*OF B*′ *D*′] *upper-triangular-four-block*[*OF B*′ *D*′ *utB utD*]])
also have . . . = *?diag B*′ ∗ *?diag D*′ **unfolding** *diag-four-block-mat*[*OF B*′ *D*′]
by *auto*
also have *?diag B*′ = *?cp B*′
by (*subst char-poly-upper-triangular*[*OF B*′ *utB*], *simp*)

```
  also have … = ?cp B
    by (rule char-poly-similar[OF similar-mat-sym[OF simB]])
  also have ?diag D′ = ?cp D′
    by (subst char-poly-upper-triangular[OF D′ utD], simp)
  also have … = ?cp D
    by (rule char-poly-similar[OF similar-mat-sym[OF simD]])
  finally show ?thesis .
qed


lemma char-poly-0-block′: fixes A :: ′a :: conjugatable-ordered-field mat
  assumes A: A = four-block-mat B (0m 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 −
  let ?A = four-block-mat B (0m n m) C D
  let ?B = transpose-mat B
  let ?D = transpose-mat D
  have AC: ?A ∈ carrier-mat (n + m) (n + m) using B D by auto
  from arg-cong[OF transpose-four-block-mat[OF B zero-carrier-mat C D], of char-poly,
    unfolded char-poly-transpose-mat[OF AC], folded A]
  have char-poly A =
    char-poly (four-block-mat ?B (transpose-mat C) (0m m n) ?D) by auto
  also have … = char-poly ?B ∗ char-poly ?D
    by (rule char-poly-0-block[OF refl], insert B C D linearB linearD, auto)
  also have … = char-poly B ∗ char-poly D using B D
    by simp
  finally show ?thesis .
qed

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* $(1\ /\ x)\ k$.

   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* :: $'a \Rightarrow nat \Rightarrow 'a\ mat \Rightarrow nat\ option$ **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* :: $nat \Rightarrow nat \Rightarrow 'a\ mat \Rightarrow 'a\ mat$ **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)
by *pat-completeness auto*
termination by *(relation measure (λ (i,j,A). j − i)) auto*

fun *identify-block* :: $'a :: one\ mat \Rightarrow nat \Rightarrow nat$ **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* :: $'a :: ring\text{-}1\ mat \Rightarrow nat \Rightarrow (nat \times nat)\ list \Rightarrow (nat \times nat)\ list$ **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)
)
by *pat-completeness auto*

definition *identify-blocks* :: $'a :: ring\text{-}1\ mat \Rightarrow nat \Rightarrow (nat \times nat) list$ **where**
identify-blocks A i = identify-blocks-main A i []

fun *find-largest-block* :: $nat \times nat \Rightarrow (nat \times nat) list \Rightarrow nat \times nat$ **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* :: $'a \Rightarrow nat \Rightarrow 'a\ mat \Rightarrow nat\ option$ **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* :: $'a\ mat \Rightarrow 'a\ mat\ list \Rightarrow 'a\ mat\ list$ **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)
  by *pat-completeness auto*
termination by (*relation measures* [λ (*i*,*j*,*A*). *n* − *j*, λ (*i*,*j*,*A*). *i*]) *auto*

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*)
  by *pat-completeness auto*
termination by (*relation measure* (λ (*j*,*A*). *n* − *j*)) *auto*

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*)
by *pat-completeness auto*
termination by (*relation measure* (λ (*k*,*A*). *n* − *k*)) *auto*

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*))])
by *pat-completeness auto*

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 (*induct i j A rule*: *swap-cols-rows-block.induct*)
case (*1 i j A*)
thus *?case* **unfolding** *swap-cols-rows-block.simps*[*of i j A*]
by (*auto split*: *if-splits*)
qed

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*
using *swap-cols-rows-block-dims-main* **unfolding** *carrier-mat-def* **by** *auto*

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 (*induct i j A taking*: *n rule*: *step-1-main.induct*)
case (*1 i j A*)
thus *?case* **unfolding** *step-1-main.simps*[*of n i j A*]
by (*auto split*: *if-splits simp*: *Let-def*)
qed

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*
using *step-1-main-dims-main* **by** *blast+*

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 (*induct j A taking*: *n rule*: *step-2-main.induct*)
case (*1 j A*)
thus *?case* **unfolding** *step-2-main.simps*[*of n j A*]
by (*auto split*: *if-splits option.splits simp*: *Let-def*)
qed

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*
using *step-2-main-dims-main* **by** *blast+*

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*
by (*induct i j A rule*: *step-3-a.induct*, *auto split*: *if-splits simp*: *Let-def*)

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*
using *step-3-a-dims-main* **by** *blast+*

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*
by (*induct val l i j A rule*: *step-3-c-inner-loop.induct*, *auto*)

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*
using *step-3-c-inner-loop-dims-main* **by** *blast+*

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*
by (*induct x l k i A rule*: *step-3-c.induct*, *auto simp*: *Let-def*)

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*
using *step-3-c-dims-main* **by** *blast+*

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 (*induct k A taking*: *n rule*: *step-3-main.induct*)
case (*1 k A*)
thus *?case* **unfolding** *step-3-main.simps*[*of n k A*]
by (*auto split*: *if-splits prod.splits option.splits simp*: *Let-def*)
qed

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*
using *step-3-main-dims-main* **by** *blast+*

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*

unfolding *step-1-def step-2-def step-3-def o-def* **by** *auto*

18.4 Properties of Auxiliary Algorithms

lemma *lookup-ev-Some*:
$lookup\text{-}ev\ ev\ j\ A = Some\ i \Longrightarrow$
$i < j \wedge A\ \$\$\ (i,i) = ev \wedge (\forall\ k.\ i < k \longrightarrow k < j \longrightarrow A\ \$\$\ (k,k) \neq ev)$
by (*induct j*, *auto split*: *if-splits*, *case-tac k* = *j*, *auto*)

lemma *lookup-ev-None*: $lookup\text{-}ev\ ev\ j\ A = None \Longrightarrow i < j \Longrightarrow A\ \$\$\ (i,i) \neq ev$
by (*induct j*, *auto split*: *if-splits*, *case-tac i* = *j*, *auto*)

lemma *swap-cols-rows-block-index*[*simp*]:
$i < dim\text{-}row\ A \Longrightarrow i < dim\text{-}col\ A \Longrightarrow j < dim\text{-}row\ A \Longrightarrow j < dim\text{-}col\ A$
$\Longrightarrow low \leq high \Longrightarrow high < dim\text{-}row\ A \Longrightarrow high < dim\text{-}col\ A$
$\Longrightarrow swap\text{-}cols\text{-}rows\text{-}block\ low\ high\ A\ \$\$\ (i,j) = A\ \$\$$
(*if* $i = low$ *then high else if* $i > low \wedge i \leq high$ *then* $i - 1$ *else i*,
if $j = low$ *then high else if* $j > low \wedge j \leq high$ *then* $j - 1$ *else j*)
proof (*induct low high A rule*: *swap-cols-rows-block.induct*)
case (*1 low high A*)
let *?r* = *dim-row A* **let** *?c* = *dim-col A*
let *?A* = *swap-cols-rows-block low high A*
let *?B* = *swap-cols-rows low high A*
let *?C* = *swap-cols-rows-block* (*Suc low*) *high ?B*
note [*simp*] = *swap-cols-rows-block.simps*[*of low high A*]
from *1*(*2*−) **have** *lh*: $low \leq high$ **by** *simp*
show *?case*
proof (*cases* $low < high$)
case *False*
with *lh* **have** *lh*: *low* = *high* **by** *auto*
from *False* **have** *?A* = *A* **by** *simp*
with *lh* **show** *?thesis* **by** *auto*
next
case *True*
hence *A*: *?A* = *?C* **by** *simp*
from *True lh* **have** *Suc low* $\leq$ *high* **by** *simp*
note *IH* = *1*(*1*)[*unfolded swap-cols-rows-carrier*, *OF True 1*(*2*−*5*) *this 1*(*7*−)]
note ∗ = *1*(*2*−)
let *?i* = *if i* = *Suc low then high else if Suc low* < *i* ∧ *i* ≤ *high then i* − *1 else*
i
let *?j* = *if j* = *Suc low then high else if Suc low* < *j* ∧ *j* ≤ *high then j* − *1 else*
j
have *cong*: $\bigwedge\ i\ j\ i'\ j'.\ i = i' \Longrightarrow j = j' \Longrightarrow A\ \$\$\ (i,j) = A\ \$\$\ (i',j')$ **by** *auto*
have *ij*: *?i* < *?r ?i* < *?c ?j* < *?r ?j* < *?c low* < *?r high* < *?r* **using** ∗ *True*
by *auto*
show *?thesis* **unfolding** *A IH*
by (*subst swap-cols-rows-index*[*OF ij*], *rule cong*, *insert* ∗ *True*, *auto*)
qed
qed

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*)
using *assms*
proof (*induct block blocks rule*: *find-largest-block.induct*)
case (*2 m-start m-end i-start i-end blocks*)
let *?res* = *find-largest-block* (*m-start*,*m-end*) ((*i-start*,*i-end*) # *blocks*)
show *?case*
proof (*cases i-end* − *i-start* ≥ *m-end* − *m-start*)
case *True*
with *2*(*3*−) **have** *find-largest-block* (*i-start*,*i-end*) *blocks* = (*m-b*, *m-e*) **by** *auto*
note *IH* = *2*(*1*)[*OF True this*]
thus *?thesis* **using** *True* **by** *auto*
next
case *False*
with *2*(*3*−) **have** *find-largest-block* (*m-start*,*m-end*) *blocks* = (*m-b*, *m-e*) **by** *auto*
note *IH* = *2*(*2*)[*OF False this*]
thus *?thesis* **using** *False* **by** *auto*
qed
qed *simp*

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 −
from *bl* **have** *id*: *insert* (*hd blocks*) (*set* (*tl blocks*)) = *set blocks* **by** (*cases blocks*, *auto*)
from *find-largest-block-main*[*OF find*, *unfolded id*]
show (*m-begin*,*m-end*) ∈ *set blocks*
⋀ *i-begin i-end*. (*i-begin*,*i-end*) ∈ *set blocks* ⟹ *m-end* − *m-begin* ≥ *i-end* − *i-begin* **by** *auto*
qed

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*)
using *assms*
proof (*induct j*)

```
  case (Suc j)
  show ?case
  proof (cases A $$ (j, Suc j) = 1)
    case False
    with Suc(2) show ?thesis by auto
  next
    case True
    with Suc(2) have identify-block A j = i by simp
    note IH = Suc(1)[OF this]
    {
      fix k
      assume k ≥ i and k < Suc j
      hence A $$ (k, Suc k) = 1
      proof (cases k < j)
        case True
        with IH ‹k ≥ i› show ?thesis by auto
      next
        case False
        with ‹k < Suc j› have k = j by auto
        with True show ?thesis by auto
      qed
    }
    with IH show ?thesis by auto
  qed
qed simp


lemma identify-block-le: identify-block A i ≤ i
  using identify-block-main[OF refl] by blast
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 −
  let ?ev = A $$ (j,j)
  note main = identify-block-main[OF assms]
  from main show i ≤ j by blast
  from main show i = 0 ∨ A $$ (i − 1, i) ≠ 1 by blast
  assume i ≤ k
  with main have main: k < j ⟹ A $$ (k, Suc k) = 1 by blast
  thus k < j ⟹ A $$ (k, Suc k) = 1 by blast
qed

lemmas identify-block-le' = identify-block(1)

lemma identify-block-le-rev: j = identify-block A i ⟹ j ≤ i
```

using *identify-block-le'*[*of A i j*] **by** *auto*

termination *identify-blocks-main* **by** (*relation measure* (λ (-,*i*,*list*). *i*), *auto simp*: *identify-block-le le-imp-less-Suc*)

termination *jnf-vector-main* **by** (*relation measure* (λ (*i*,*A*). *i*), *auto simp*: *identify-block-le le-imp-less-Suc*)

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* **using** *assms*
proof (*induct A i list rule*: *identify-blocks-main.induct*)
case (*2 A i-e list*)
obtain *i-b* **where** *id*: *identify-block A i-e* = *i-b* **by** *force*
note *IH* = *2*(*1*)[*OF id*[*symmetric*]]
let *?res* = *identify-blocks-main A* (*Suc i-e*) *list*
let *?rec* = *identify-blocks-main A i-b* ((*i-b*, *i-e*) # *list*)
have *res*: *?res* = *?rec* **using** *id* **by** *simp*
from *2*(*2*)[*unfolded res*] **have** (*i-start*, *i-end*) ∈ *set ?rec* **.**
note *IH* = *IH*[*OF this*]
from *2*(*3*−) **have** *iek*: *i-e* < *k* **by** *simp*
from *identify-block-le'*[*OF id*] **have** *ibe*: *i-b* ≤ *i-e* **.**
from *ibe iek* **have** *i-b* ≤ *k* **by** *simp*
note *IH* = *IH*[*OF - this*]
show *?thesis*
by (*rule IH*, *insert ibe iek 2*(*3*−), *auto*)
qed *simp*

lemma *identify-blocks*: **assumes** (*i-start*,*i-end*) ∈ *set* (*identify-blocks B k*)
shows *i-start* ≤ *i-end* ∧ *i-end* < *k*
using *identify-blocks-main*[*OF assms*[*unfolded identify-blocks-def*]] **by** *auto*

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*
using *assms*
proof (*induct i j A rule*: *swap-cols-rows-block.induct*)
case (*1 i j A*)
hence *A*: *A* ∈ *carrier-mat n n*
and *jn*: *j* < *n* **and** *ij*: *i* ≤ *j* **by** *auto*
note [*simp*] = *swap-cols-rows-block.simps*[*of i j*]
show *?case*
proof (*cases i* < *j*)

case *False*
thus *?thesis* **using** *similar-mat-refl*[*OF A*] **by** *simp*
next
case *True* **note** *ij* = *this*
let *?B* = *swap-cols-rows i j A*
let *?C* = *swap-cols-rows-block* (*Suc i*) *j ?B*
from *swap-cols-rows-similar*[*OF A - jn*, *of i*] *ij jn*
have *BA*: *similar-mat ?B A* **by** *auto*
have *CB*: *similar-mat ?C ?B*
by (*rule 1*(*1*)[*OF ij - jn*], *insert ij A*, *auto*)
from *similar-mat-trans*[*OF CB BA*] **show** *?thesis* **using** *ij* **by** *simp*
qed
qed

private lemma *step-1-main-similar*: $i \leq j \Longrightarrow A \in$ *carrier-mat n n* $\Longrightarrow$ *similar-mat* (*step-1-main n i j A*) *A*
proof (*induct i j A taking*: *n rule*: *step-1-main.induct*)
case (*1 i j A*)
note [*simp*] = *step-1-main.simps*[*of n i j A*]
from *1*(*3*−) **have** *ij*: $i \leq j$ **and** *A*: $A \in$ *carrier-mat n n* **by** *auto*
show *?case*
proof (*cases* $j \geq n$)
case *True*
thus *?thesis* **using** *similar-mat-refl*[*OF A*] **by** *simp*
next
case *False*
hence *jn*: $j < n$ **by** *simp*
note *IH* = *1*(*1*−*2*)[*OF False*]
show *?thesis*
proof (*cases* $i = 0$)
case *True*
from *IH*(*1*)[*OF this - A*]
show *?thesis* **using** *jn* **by** (*simp*, *simp add*: *True*)
next
case *False*
let *?evi* = *A* $$ (*i* − *1*, *i* − *1*)
let *?evj* = *A* $$ (*j*,*j*)
let *?B* = *if ?evi* ≠ *?evj* ∧ *A* $$ (*i* − *1*, *j*) ≠ *0 then*
add-col-sub-row (*A* $$ (*i* − *1*, *j*) / (*?evj* − *?evi*)) (*i* − *1*) *j A else A*
obtain *B* **where** *B*: *B* = *?B* **by** *auto*
have *Bn*: *B* ∈ *carrier-mat n n* **unfolding** *B* **using** *A* **by** *simp*
from *False ij jn* **have** ∗: *i* − *1* < *n j* < *n i* − *1* ≠ *j* **by** *auto*
have *BA*: *similar-mat B A* **unfolding** *B* **using** *similar-mat-refl*[*OF A*]
add-col-sub-row-similar[*OF A* ∗] **by** *auto*
from *ij* **have** *i* − *1* ≤ *j* **by** *simp*
note *IH* = *IH*(*2*)[*OF False refl refl refl refl B this Bn*]
from *False jn* **have** *id*: *step-1-main n i j A* = *step-1-main n* (*i* − *1*) *j B*
unfolding *B* **by** (*simp add*: *Let-def*)
show *?thesis* **unfolding** *id*

by (*rule similar-mat-trans*[*OF IH BA*])
qed
qed
qed

private lemma *step-2-main-similar*: $A \in$ *carrier-mat n n* $\Longrightarrow$ *similar-mat* (*step-2-main n j A*) *A*
proof (*induct j A taking*: *n rule*: *step-2-main.induct*)
case (*1 j A*)
note [*simp*] = *step-2-main.simps*[*of n j A*]
from *1*(*2*) **have** *A*: $A \in$ *carrier-mat n n* **.**
show *?case*
proof (*cases* $j \geq n$)
case *True*
thus *?thesis* **using** *similar-mat-refl*[*OF A*] **by** *simp*
next
case *False*
hence *jn*: $j < n$ **by** *simp*
note *IH* = *1*(*1*)[*OF False*]
let *?look* = *lookup-ev* (*A* $$ (*j*,*j*)) *j A*
let *?B* = *case ?look of*
None ⇒ *A*
| *Some i* ⇒ *swap-cols-rows-block* (*Suc i*) *j A*
obtain *B* **where** *B*: *B* = *?B* **by** *auto*
have *id*: *step-2-main n j A* = *step-2-main n* (*Suc j*) *B* **unfolding** *B* **using** *False* **by** *simp*
have *Bn*: $B \in$ *carrier-mat n n* **unfolding** *B* **using** *A* **by** (*auto split*: *option.splits*)
have *BA*: *similar-mat B A*
proof (*cases ?look*)
case *None*
thus *?thesis* **unfolding** *B* **using** *similar-mat-refl*[*OF A*] **by** *simp*
next
case (*Some i*)
from *lookup-ev-Some*[*OF this*]
show *?thesis* **unfolding** *B Some*
by (*auto intro*: *swap-cols-rows-block-similar*[*OF A jn*])
qed
show *?thesis* **unfolding** *id*
by (*rule similar-mat-trans*[*OF IH*[*OF refl B Bn*] *BA*])
qed
qed

private lemma *step-3-a-similar*: $A \in$ *carrier-mat n n* $\Longrightarrow i < j \Longrightarrow j < n \Longrightarrow$ *similar-mat* (*step-3-a i j A*) *A*
proof (*induct i j A rule*: *step-3-a.induct*)
case (*1 j A*)
thus *?case* **by** (*simp add*: *similar-mat-refl*)
next

```
  case (2 i j A)
  from 2(2−) have A: A ∈ carrier-mat n n and ij: Suc i < j and j: j < n by
auto
  let ?B = if A $$ (i, i + 1) = 1 ∧ A $$ (i, j) ≠ 0
    then add-col-sub-row (− A $$ (i, j)) (Suc i) j A else A
  obtain B where B: B = ?B by auto
  from A have Bn: B ∈ carrier-mat n n unfolding B by simp
  note IH = 2(1)[OF refl B Bn - j]
   have id: step-3-a (Suc i) j A = step-3-a i j B unfolding B by (simp add:
Let-def)
  from ij j have ∗: Suc i < n j < n Suc i ≠ j by auto
  have BA: similar-mat B A unfolding B
    using similar-mat-refl[OF A] add-col-sub-row-similar[OF A ∗] by auto
  show ?case unfolding id
    by (rule similar-mat-trans[OF IH BA], insert ij, auto)
qed

private 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 (induct val l i k A rule: step-3-c-inner-loop.induct)
  case (1 val l i A)
  thus ?case using similar-mat-refl[of A] by simp
next
  case (2 val l i k A)
  let ?res = step-3-c-inner-loop val l i (Suc k) A
  from 2(2−) have A: A ∈ carrier-mat n n and
    ∗: l ≠ i k ≤ l k ≤ i l < n i < n by auto
  let ?B = add-col-sub-row val i l A
  have BA: similar-mat ?B A
    by (rule add-col-sub-row-similar[OF A], insert ∗, auto)
  have B: ?B ∈ carrier-mat n n using A unfolding carrier-mat-def by simp
  show ?case
  proof (cases k)
    case 0
    hence id: ?res = ?B by simp
    thus ?thesis using BA by simp
  next
    case (Suc kk)
    with ∗ have l − 1 ≠ i − 1 k − 1 ≤ l − 1 k − 1 ≤ i − 1 l − 1 < n i − 1 <
n by auto
    note IH = 2(1)[OF B this]
    show ?thesis unfolding step-3-c-inner-loop.simps
      by (rule similar-mat-trans[OF IH BA])
  qed
qed

private 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 (*induct x l k blocks A rule*: *step-3-c.induct*)
  case (*1 x l k A*)
  thus *?case* **using** *similar-mat-refl*[*OF 1*(*1*)] **by** *simp*
next
  case (*2 x l k i-begin i-end blocks A*)
  let *?res* = *step-3-c x l k* ((*i-begin*,*i-end*) # *blocks*) *A*
  from *2*(*2*−*4*) **have** *A*: *A* ∈ *carrier-mat n n* **and** *lki*: *l* < *k k* < *n* **by** *auto*
  from *2*(*5*) **have** *i*: *i-end* ≤ *k i-end* − *i-begin* ≤ *l* **by** *auto*
  let *?y* = *A* $$ (*i-end*,*k*)
  let *?inner* = *step-3-c-inner-loop* (*?y* / *x*) *l i-end* (*Suc i-end* − *i-begin*) *A*
  obtain *B* **where** *B*:
    B = (*if i-end* = *l then A else ?inner*) **by** *auto*
  hence *id*: *?res* = *step-3-c x l k blocks B*
    by *simp*
  have *BA*: *similar-mat B A*
  proof (*cases i-end* = *l*)
    case *True*
    thus *?thesis* **unfolding** *B* **using** *similar-mat-refl*[*OF A*] **by** *simp*
  next
    case *False*
    hence *B*: *B* = *?inner* **and** *li*: *l* ≠ *i-end* **by** (*auto simp*: *B*)
    show *?thesis* **unfolding** *B*
      by (*rule step-3-c-inner-loop-similar*[*OF A li*], *insert lki i*, *auto*)
  qed
  have *Bn*: *B* ∈ *carrier-mat n n* **using** *A* **unfolding** *B carrier-mat-def* **by** *simp*
  note *IH* = *2*(*1*)[*OF B Bn lki*(*1*−*2*) *2*(*5*)]
  show *?case* **unfolding** *id*
    by (*rule similar-mat-trans*[*OF IH BA*], *auto*)
qed

private lemma *step-3-main-similar*: *A* ∈ *carrier-mat n n* ⟹ *k* > *0* ⟹ *similar-mat* (*step-3-main n k A*) *A*

proof (*induct k A taking*: *n rule*: *step-3-main.induct*)
  case (*1 k A*)
  from *1*(*2*−) **have** *A*: *A* ∈ *carrier-mat n n* **and** *k*: *k* > *0* **by** *auto*
  note [*simp*] = *step-3-main.simps*[*of n k A*]
  show *?case*
  proof (*cases k* ≥ *n*)
    case *True*
    thus *?thesis* **using** *similar-mat-refl*[*OF A*] **by** *simp*
  next
    case *False*
    hence *kn*: *k* < *n* **by** *simp*
    obtain *B* **where** *B*: *B* = *step-3-a* (*k* − *1*) *k A* **by** *auto*
    note *IH* = *1*(*1*)[*OF False B*]

from *A* **have** *Bn*: $B \in$ *carrier-mat n n* **unfolding** *B carrier-mat-def* **by** *simp*
from *k* **have** $k - 1 < k$ **by** *simp*
from *step-3-a-similar*[*OF A this kn*] **have** *BA*: *similar-mat B A* **unfolding** *B*
.
obtain *all-blocks* **where** *ab*: *all-blocks* = *identify-blocks B k* **by** *simp*
obtain *blocks* **where** *blocks*: *blocks* = *filter* (λ *block*. *B* \$\$ (*snd block*, *k*) $\neq$ *0*) *all-blocks* **by** *simp*
obtain *F* **where** *F*: *F* = (*if blocks* = [] *then B*
else let (*l-begin*,*l*) = *find-largest-block* (*hd blocks*) (*tl blocks*); *x* = *B* \$\$ (*l*, *k*); *C* = *step-3-c x l k blocks B*;
D = *mult-col-div-row* (*inverse x*) *k C*; *E* = *swap-cols-rows-block* (*Suc l*) *k D*
in E) **by** *simp*
note *IH* = *IH*[*OF ab blocks F*]
have *Fn*: $F \in$ *carrier-mat n n* **unfolding** *F Let-def carrier-mat-def* **using** *Bn*
by (*simp split*: *prod.splits*)
have *FB*: *similar-mat F B*
proof (*cases blocks* = [])
case *True*
thus *?thesis* **unfolding** *F* **using** *similar-mat-refl*[*OF Bn*] **by** *simp*
next
case *False*
obtain *l-start l* **where** *l*: *find-largest-block* (*hd blocks*) (*tl blocks*) = (*l-start*, *l*) **by** *force*
obtain *x* **where** *x*: *x* = *B* \$\$ (*l*,*k*) **by** *simp*
obtain *C* **where** *C*: *C* = *step-3-c x l k blocks B* **by** *simp*
obtain *D* **where** *D*: *D* = *mult-col-div-row* (*inverse x*) *k C* **by** *auto*
obtain *E* **where** *E*: *E* = *swap-cols-rows-block* (*Suc l*) *k D* **by** *auto*
from *find-largest-block*[*OF False l*] **have** *lb*: (*l-start*,*l*) $\in$ *set blocks*
and *llarge*: $\bigwedge$ *i-begin i-end*. (*i-begin*,*i-end*) $\in$ *set blocks* $\Longrightarrow$ *l* − *l-start* $\geq$ *i-end* − *i-begin* **by** *auto*
from *lb* **have** *x0*: $x \neq 0$ **unfolding** *blocks x* **by** *simp*
{
fix *i-start i-end*
assume (*i-start*,*i-end*) $\in$ *set blocks*
hence (*i-start*,*i-end*) $\in$ *set* (*identify-blocks B k*) **unfolding** *blocks ab* **by** *simp*
from *identify-blocks*[*OF this*]
have *i-end* < *k* **by** *simp*
} **note** *block-bound* = *this*
from *block-bound*[*OF lb*]
have *lk*: $l < k$ **.**
from *False* **have** *F*: *F* = *E* **unfolding** *E D C x F l Let-def* **by** *simp*
from *Bn* **have** *Cn*: $C \in$ *carrier-mat n n* **unfolding** *C carrier-mat-def* **by** *simp*
have *CB*: *similar-mat C B* **unfolding** *C*
proof (*rule step-3-c-similar*[*OF Bn lk kn*])
fix *i-begin i-end*
assume *i*: (*i-begin*, *i-end*) $\in$ *set blocks*

from *llarge*[*OF i*] *block-bound*[*OF i*]
show *i-end* ≤ *k* ∧ *i-end* − *i-begin* ≤ *l* **by** *auto*
qed
from *x0* **have** *inverse x* ≠ *0* **by** *simp*
from *mult-col-div-row-similar*[*OF Cn kn this*]
have *DC*: *similar-mat D C* **unfolding** *D* **.**
from *Cn* **have** *Dn*: *D* ∈ *carrier-mat n n* **unfolding** *D carrier-mat-def* **by** *simp*
from *lk* **have** *Suc l* ≤ *k* **by** *auto*
from *swap-cols-rows-block-similar*[*OF Dn kn this*]
have *ED*: *similar-mat E D* **unfolding** *E* **.**
from *similar-mat-trans*[*OF ED similar-mat-trans*[*OF DC*
similar-mat-trans[*OF CB similar-mat-refl*[*OF Bn*]]]]
show *?thesis* **unfolding** *F* **.**
qed
have *0* < *Suc k* **by** *simp*
note *IH* = *IH*[*OF Fn this*]
have *id*: *step-3-main n k A* = *step-3-main n* (*Suc k*) *F* **using** *kn*
by (*simp add*: *F Let-def blocks ab B*)
show *?thesis* **unfolding** *id*
by (*rule similar-mat-trans*[*OF IH similar-mat-trans*[*OF FB BA*]])
qed
qed

lemma *step-1-similar*: *A* ∈ *carrier-mat n n* ⟹ *similar-mat* (*step-1 A*) *A*
unfolding *step-1-def* **by** (*rule step-1-main-similar*, *auto*)

lemma *step-2-similar*: *A* ∈ *carrier-mat n n* ⟹ *similar-mat* (*step-2 A*) *A*
unfolding *step-2-def* **by** (*rule step-2-main-similar*, *auto*)

lemma *step-3-similar*: *A* ∈ *carrier-mat n n* ⟹ *similar-mat* (*step-3 A*) *A*
unfolding *step-3-def* **by** (*rule step-3-main-similar*, *auto*)

end

18.6 Invariants for Proving that Result is in JNF

context
fixes *n* :: *nat*
and *ty* :: *'a* :: *field itself*
begin

definition *uppert* :: *'a mat* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
uppert A i j ≡ *j* < *i* ⟶ *A* $$ (*i*,*j*) = *0*

definition *diff-ev* :: *'a mat* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
diff-ev A i j ≡ *i* < *j* ⟶ *A* $$ (*i*,*i*) ≠ *A* $$ (*j*,*j*) ⟶ *A* $$ (*i*,*j*) = *0*

definition *ev-blocks-part* :: *nat* ⇒ *'a mat* ⇒ *bool* **where**
ev-blocks-part m A ≡ ∀ *i j k*. *i* < *j* ⟶ *j* < *k* ⟶ *k* < *m* ⟶ *A* $$ (*k*,*k*) = *A* $$ (*i*,*i*) ⟶ *A* $$ (*j*,*j*) = *A* $$ (*i*,*i*)

definition *ev-blocks* :: *'a mat* ⇒ *bool* **where**
ev-blocks ≡ *ev-blocks-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 *one-zero* :: *'a mat* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
one-zero A i j ≡
(*Suc i* < *j* ⟶ *A* $$ (*i*,*Suc i*) = *1* ⟶ *A* $$ (*i*,*j*) = *0*) ∧
(*j* < *i* ⟶ *A* $$ (*i*,*j*) = *0*) ∧
(*i* < *j* ⟶ *A* $$ (*i*,*i*) ≠ *A* $$ (*j*,*j*) ⟶ *A* $$ (*i*,*j*) = *0*)

private definition *single-non-zero* :: *nat* ⇒ *nat* ⇒ *'a* ⇒ *'a mat* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
single-non-zero ≡ λ *l k x*. (λ *A i j*. (*i* ∉ {*k*,*l*} ⟶ *A* $$ (*i*,*k*) = *0*) ∧ *A* $$ (*l*,*k*) = *x*)

private definition *single-one* :: *nat* ⇒ *nat* ⇒ *'a mat* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
single-one ≡ λ *l k*. (λ *A i j*. (*i* ∉ {*k*,*l*} ⟶ *A* $$ (*i*,*k*) = *0*) ∧ *A* $$ (*l*,*k*) = *1*)

private definition *lower-one* :: *nat* ⇒ *'a mat* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
lower-one k A i j = (*j* = *k* ⟶
(*A* $$ (*i*,*j*) = *0* ∨ *i* = *j* ∨ (*A* $$ (*i*,*j*) = *1* ∧ *j* = *Suc i* ∧ *A* $$ (*i*,*i*) = *A* $$ (*j*,*j*))))

definition *jb* :: *'a mat* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
jb A i j ≡ (*Suc i* = *j* ⟶ *A* $$ (*i*,*j*) ∈ {*0*,*1*})
∧ (*i* ≠ *j* ⟶ (*Suc i* ≠ *j* ∨ *A* $$ (*i*,*i*) ≠ *A* $$ (*j*,*j*)) ⟶ *A* $$ (*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* ⇒ *'a mat* ⇒ *bool* **where**
same-diag A B ≡ ∀ *i* < *n*. *A* $$ (*i*,*i*) = *B* $$ (*i*,*i*)

private definition *same-upto* :: *nat* ⇒ *'a mat* ⇒ *'a mat* ⇒ *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-i m-j* $\equiv \forall\ i\ j.\ i < n \longrightarrow j < n \longrightarrow j <$ *m-j* $\lor j =$ *m-j* $\land i \geq$ *m-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* $A\ i\ j$
unfolding *jb-def uppert-def* **by** *auto*

private lemma *ev-blocks-partD*:
ev-blocks-part $m\ A \Longrightarrow i < j \Longrightarrow j < k \Longrightarrow k < m \Longrightarrow A \$\$ (k,k) = A \$\$ (i,i) \Longrightarrow A \$\$ (j,j) = A \$\$ (i,i)$
unfolding *ev-blocks-part-def* **by** *auto*

private 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 −
show *?thesis*
proof (*cases* $i = j \lor j = k$)
case *False*
with *assms*(*2*−*3*) **have** $i < j$ $j < k$ **by** *auto*
from *ev-blocks-partD*[*OF assms*(*1*) *this assms*(*4*−)] **show** *?thesis* **.**
next
case *True*
thus *?thesis* **using** *assms*(*5*) **by** *auto*

  qed
qed

private 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*
  using *assms* **unfolding** *ev-blocks-part-def* **by** *blast*

private lemma *ev-blocksD*:
  $ev\text{-}blocks\ A \Longrightarrow i < j \Longrightarrow j < k \Longrightarrow k < n \Longrightarrow A\ \$\$\ (k,k) = A\ \$\$\ (i,i) \Longrightarrow A\ \$\$\ (j,j) = A\ \$\$\ (i,i)$
  unfolding *ev-blocks-def* **by** (*rule ev-blocks-partD*)

private lemma *ev-blocks-leD*:
  $ev\text{-}blocks\ A \Longrightarrow i \le j \Longrightarrow j \le k \Longrightarrow k < n \Longrightarrow A\ \$\$\ (k,k) = A\ \$\$\ (i,i) \Longrightarrow A\ \$\$\ (j,j) = A\ \$\$\ (i,i)$
  unfolding *ev-blocks-def* **by** (*rule ev-blocks-part-leD*)

lemma *inv-allD*: $inv\text{-}all\ p\ A \Longrightarrow i < n \Longrightarrow j < n \Longrightarrow p\ A\ i\ j$
  unfolding *inv-all-def* **by** *auto*

private lemma *inv-allI*: **assumes** $\bigwedge i\ j.\ i < n \Longrightarrow j < n \Longrightarrow p\ A\ i\ j$
  shows *inv-all p A*
  using *assms* **unfolding** *inv-all-def* **by** *blast*

private lemma *inv-partI*: **assumes** $\bigwedge i\ j.\ i < n \Longrightarrow j < n \Longrightarrow j < m\text{-}j \lor j = m\text{-}j \land i \ge m\text{-}i \Longrightarrow p\ A\ i\ j$
  shows *inv-part p A m-i m-j*
  using *assms* **unfolding** *inv-part-def* **by** *auto*

private lemma *inv-partD*: **assumes** $inv\text{-}part\ p\ A\ m\text{-}i\ m\text{-}j\ i < n\ j < n$
  shows $j < m\text{-}j \Longrightarrow p\ A\ i\ j$
  and $j = m\text{-}j \Longrightarrow i \ge m\text{-}i \Longrightarrow p\ A\ i\ j$
  and $j < m\text{-}j \lor j = m\text{-}j \land i \ge m\text{-}i \Longrightarrow p\ A\ i\ j$
  using *assms* **unfolding** *inv-part-def* **by** *auto*

private 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*
  using *assms* **unfolding** *inv-upto-def* **by** *auto*

private lemma *inv-uptoD*: **assumes** $inv\text{-}upto\ p\ A\ m\ i < n\ j < n\ j < m$
  shows *p A i j*
  using *assms* **unfolding** *inv-upto-def* **by** *auto*

private 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 (*intro inv-uptoI*)
  fix *i j*
  assume $i < n$ $j < n$ $j < Suc\ m$
  thus *p A i j* **using** *inv-uptoD*[*OF assms*(*1*), *of i j*] *assms*(*2*)[*of i*] **by** (*cases* $j = m$, *auto*)
qed

private 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*
  using *assms* **unfolding** *inv-upto-def* **by** *auto*

private 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*
  using *assms* **unfolding** *inv-from-def* **by** *auto*

private lemma *inv-fromD*: **assumes** *inv-from p A m* $i < n$ $j < n$ $j > m$
  shows *p A i j*
  using *assms* **unfolding** *inv-from-def* **by** *auto*

private lemma *inv-atI*[*intro*]: **assumes** $\bigwedge i.\ i < n \Longrightarrow p\ A\ i\ m$
  shows *inv-at p A m*
  using *assms* **unfolding** *inv-at-def* **by** *auto*

private lemma *inv-atD*: **assumes** *inv-at p A m* $i < n$
  shows *p A i m*
  using *assms* **unfolding** *inv-at-def* **by** *auto*

private lemma *inv-all-imp-inv-part*: $m\ i \leq n \Longrightarrow m\text{-}j \leq n \Longrightarrow$ *inv-all p A* $\Longrightarrow$ *inv-part p A m-i m-j*
  unfolding *inv-all-def inv-part-def* **by** *auto*

private lemma *inv-all-eq-inv-part*: *inv-all p A* = *inv-part p A n n*
  unfolding *inv-all-def inv-part-def* **by** *auto*

private lemma *inv-part-0-Suc*: $m\text{-}j < n \Longrightarrow$ *inv-part p A 0 m-j* = *inv-part p A n* (*Suc m-j*)
  unfolding *inv-part-def* **by** (*auto*, *case-tac* $j = m\text{-}j$, *auto*)

private lemma *inv-all-uppertD*: *inv-all uppert A* $\Longrightarrow j < i \Longrightarrow i < n \Longrightarrow A\ \$\$\ (i,j) = 0$
  unfolding *inv-all-def uppert-def* **by** *auto*

private lemma *inv-all-diff-evD*: *inv-all diff-ev A* $\Longrightarrow i < j \Longrightarrow j < n$
  $\Longrightarrow A\ \$\$\ (i,\ i) \neq A\ \$\$\ (j,\ j) \Longrightarrow A\ \$\$\ (i,j) = 0$
  unfolding *inv-all-def diff-ev-def* **by** *auto*

private 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 −
from *neg* **have** $i \neq j$ **by** *auto*
hence $i < j \lor j < i$ **by** *arith*
thus *?thesis*
proof
assume $i < j$
from *inv-all-diff-evD*[*OF assms*(*1*) *this* ‹$j < n$› *neg*] **show** *?thesis* **.**
next
assume $j < i$
from *inv-all-uppertD*[*OF assms*(*2*) *this* ‹$i < n$›] **show** *?thesis* **.**
qed
qed

private lemma *inv-from-bot-step*: *p A i* $\Longrightarrow$ *inv-from-bot p A* (*Suc i*) $\Longrightarrow$ *inv-from-bot p A i*
unfolding *inv-from-bot-def* **by** (*auto*, *case-tac ia* = *i*, *auto*)

private lemma *same-diag-refl*[*simp*]: *same-diag A A* **unfolding** *same-diag-def* **by** *auto*
private lemma *same-diag-trans*: *same-diag A B* $\Longrightarrow$ *same-diag B C* $\Longrightarrow$ *same-diag A C*
unfolding *same-diag-def* **by** *auto*

private lemma *same-diag-ev-blocks*: *same-diag A B* $\Longrightarrow$ *ev-blocks A* $\Longrightarrow$ *ev-blocks B*
unfolding *same-diag-def ev-blocks-def ev-blocks-part-def* **by** *auto*

private lemma *same-uptoI*[*intro*]: **assumes** $\bigwedge i'\ j'.\ i' < n \Longrightarrow j' < j \Longrightarrow A$ $$ $(i',j') = B$ $$ (i',j')
shows *same-upto j A B*
using *assms* **unfolding** *same-upto-def* **by** *blast*

private lemma *same-uptoD*[*dest*]: **assumes** *same-upto j A B* $i' < n$ $j' < j$
shows A $$ $(i',j') = B$ $$ (i',j')
using *assms* **unfolding** *same-upto-def* **by** *blast*

private lemma *same-upto-refl*[*simp*]: *same-upto j A A* **unfolding** *same-upto-def* **by** *auto*

private lemma *same-upto-trans*: *same-upto j A B* $\Longrightarrow$ *same-upto j B C* $\Longrightarrow$ *same-upto j A C*
unfolding *same-upto-def* **by** *auto*

private lemma *same-upto-inv-upto-jb*: *same-upto j A B* $\Longrightarrow$ *inv-upto jb A j* $\Longrightarrow$ *inv-upto jb B j*

```
  unfolding inv-upto-def same-upto-def jb-def by auto

lemma jb-imp-diff-ev: jb A i j ⟹ diff-ev A i j
  unfolding jb-def diff-ev-def by auto

private lemma ev-blocks-diag:
  same-diag A B ⟹ ev-blocks B ⟹ ev-blocks A
  unfolding ev-blocks-def ev-blocks-part-def same-diag-def by auto

private lemma inv-all-imp-inv-from: inv-all p A ⟹ inv-from p A k
  unfolding inv-all-def inv-from-def by auto

private lemma inv-all-imp-inv-at: inv-all p A ⟹ k < n ⟹ inv-at p A k
  unfolding inv-all-def inv-at-def by auto

private lemma inv-from-upto-at-all:
  assumes inv-upto jb A k inv-from diff-ev A k inv-from uppert A k inv-at p A k
  and ⋀ i. i < n ⟹ p A i k ⟹ diff-ev A i k ∧ uppert A i k
  shows inv-all diff-ev A inv-all uppert A
proof −
  {
    fix i j
    assume ij: i < n j < n
    have diff-ev A i j ∧ uppert A i j
    proof (cases j < k)
      case True
      with assms(1) ij have jb A i j unfolding inv-upto-def by auto
      thus ?thesis using ij unfolding jb-def diff-ev-def uppert-def by auto
    next
      case False note ge = this
      show ?thesis
      proof (cases j = k)
        case True
        with assms(4−) ij show ?thesis unfolding inv-at-def by auto
      next
        case False
        with ge have j > k by auto
        with assms(2−3) ij show ?thesis unfolding inv-from-def by auto
      qed
    qed
  }
  thus inv-all diff-ev A inv-all uppert A unfolding inv-all-def by auto
qed

private lemma lower-one-diff-uppert:
  i < n ⟹ lower-one k B i k ⟹ diff-ev B i k ∧ uppert B i k
   unfolding lower-one-def diff-ev-def uppert-def by auto

definition ev-block :: nat ⇒ 'a mat ⇒ bool where
```

$\bigwedge$ *n. ev-block n A* = (∀ *i j. i* < *n* ⟶ *j* < *n* ⟶ *A* $$ (*i*,*i*) = *A* $$ (*j*,*j*))

lemma *ev-blockD*: $\bigwedge$ *n. ev-block n A* ⟹ *i* < *n* ⟹ *j* < *n* ⟹ *A* $$ (*i*,*i*) = *A* $$ (*j*,*j*)
  unfolding *ev-block-def carrier-mat-def* **by** *blast*

lemma *same-diag-ev-block*: *same-diag A B* ⟹ *ev-block n A* ⟹ *ev-block n B*
  unfolding *ev-block-def carrier-mat-def same-diag-def* **by** *metis*

18.7 Alternative Characterization of *identify-blocks* in Presence of *local.ev-block*

private lemma *identify-blocks-main-iff*: **assumes** ∗: *k* ≤ *k′*
  k′ ≠ *k* ⟶ *k* > *0* ⟶ *A* $$ (*k* − *1*, *k*) ≠ *1* **and** *k′* < *n*
  shows *set* (*identify-blocks-main A k list*) =
  set list ∪ {(*i*,*j*) | *i j. i* ≤ *j* ∧ *j* < *k* ∧ (∀ *l. i* ≤ *l* ⟶ *l* < *j* ⟶ *A* $$ (*l*, *Suc l*) = *1*)
  ∧ (*Suc j* ≠ *k′* ⟶ *A* $$ (*j*, *Suc j*) ≠ *1*) ∧ (*i* > *0* ⟶ *A* $$ (*i* − *1*, *i*) ≠ *1*)} (**is** - = - ∪ *?ss A k*)
  using ∗
proof (*induct A k list rule*: *identify-blocks-main.induct*)
  case *1*
  show *?case* **unfolding** *identify-blocks-main.simps* **by** *auto*
next
  case (*2 A i-e list*)
  let *?s* = *?ss A*
  obtain *i-b* **where** *id*: *identify-block A i-e* = *i-b* **by** *force*
  note *IH* = *2*(*1*)[*OF id*[*symmetric*]]
  let *?res* = *identify-blocks-main A* (*Suc i-e*) *list*
  let *?rec* = *identify-blocks-main A i-b* ((*i-b*, *i-e*) # *list*)
  note *idb* = *identify-block*[*OF id*]
  hence *res*: *?res* = *?rec* **using** *id* **by** *simp*
  from *2*(*2*−) **have** *iek*: *i-e* < *k′* **by** *simp*
  from *identify-block-le′*[*OF id*] **have** *ibe*: *i-b* ≤ *i-e* **.**
  from *ibe iek* **have** *i-b* ≤ *k′* **by** *simp*
  have *k′* ≠ *i-b* ⟶ *0* < *i-b* ⟶ *A* $$ (*i-b* − *1*, *i-b*) ≠ *1*
    using *idb*(*2*) **by** *auto*
  note *IH* = *IH*[*OF* ‹*i-b* ≤ *k′*› *this*]
  have *cong*: $\bigwedge$ *a b c d. insert a c* = *d* ⟹ *set* (*a* # *b*) ∪ *c* = *set b* ∪ *d* **by** *auto*
  show *?case* **unfolding** *res IH*
  proof (*rule cong*)
    from *ibe* **have** *?s i-b* ⊆ *?s* (*Suc i-e*) **by** *auto*
    moreover
    have *inter*: $\bigwedge$*l. i-b* ≤ *l* ⟹ *l* < *i-e* ⟹ *A* $$ (*l*, *Suc l*) = *1* **using** *idb* **by** *blast*
    have *last*: *Suc i-e* ≠ *k′* ⟹ *A* $$ (*i-e*, *Suc i-e*) ≠ *1* **using** *2*(*3*) **by** *auto*
    have (*i-b*, *i-e*) ∈ *?s* (*Suc i-e*) **using** *ibe idb*(*2*) *inter last* **by** *blast*
    ultimately have *insert* (*i-b*, *i-e*) (*?s i-b*) ⊆ *?s* (*Suc i-e*) **by** *auto*
    moreover
    {

fix *i j*
assume *ij*: $(i,j) \in$ *?s* (*Suc i-e*)
hence $(i,j) \in$ *insert* (*i-b*, *i-e*) (*?s i-b*)
proof (*cases* $j <$ *i-b*)
case *True*
with *ij* **show** *?thesis* **by** *blast*
next
case *False*
with *ij* **have** *i-b* $\leq j$ $j \leq$ *i-e* **by** *auto*
{
assume *j*: $j <$ *i-e*
from *idb*(*3*)[*OF* ‹*i-b* $\leq j$› *this*] **have** *1*: *A* $$ (*j*, *Suc j*) = *1* **.**
from *j* ‹*Suc i-e* $\leq k'$› **have** *Suc j* $\neq k'$ **by** *auto*
with *ij 1* **have** *False* **by** *auto*
}
with ‹$j \leq$ *i-e*› **have** *j*: $j =$ *i-e* **by** (*cases* $j =$ *i-e*, *auto*)
{
assume *i*: $i <$ *i-b* $\vee$ $i >$ *i-b*
hence *False*
proof
assume $i <$ *i-b*
hence *i-b* > 0 **by** *auto*
with *idb*(*2*) **have** $*$: *A* $$ (*i-b* $-$ *1*, *i-b*) $\neq$ *1* **by** *auto*
from ‹$i <$ *i-b*› ‹*i-b* $\leq$ *i-e*› ‹*i-e* $< k'$› **have** $i \leq$ *i-b* $-$ *1* *i-b* $-$ *1* $\leq k'$ **by** *auto*
from ‹$i <$ *i-b*› ‹*i-b* $\leq$ *i-e*› *j* **have** $**$: $i \leq$ *i-b* $-$ *1* *i-b* $-$ *1* $< j$ *Suc* (*i-b* $-$ *1*) = *i-b* **by** *auto*
from *ij* **have** $\bigwedge$ *l*. $l \geq i \Longrightarrow l < j \Longrightarrow$ *A* $$ (*l*, *Suc l*) = *1* **by** *auto*
from *this*[*OF* $**$(*1*−*2*)] $**$(*3*) $*$ **show** *False* **by** *auto*
next
assume $i >$ *i-b*
with *ij j* **have** *A* $$ ($i - 1$, *i*) $\neq$ *1* **and**
$*$: $i - 1 \geq$ *i-b* $i - 1 \leq$ *i-e* $i - 1 <$ *i-e* *Suc* ($i - 1$) = *i* **by** *auto*
with *idb*(*3*)[*OF* $*$(*1*,*3*)] **show** *False* **by** *auto*
qed
}
hence *i*: $i =$ *i-b* **by** *arith*
show *?thesis* **unfolding** *i j* **by** *simp*
qed
}
hence *?s* (*Suc i-e*) $\subseteq$ *insert* (*i-b*, *i-e*) (*?s i-b*) **by** *blast*
ultimately
show *insert* (*i-b*, *i-e*) (*?s i-b*) = *?s* (*Suc i-e*) **by** *blast*
qed
qed

private lemma *identify-blocks-iff*: **assumes** $k < n$
shows *set* (*identify-blocks A k*) =

$\{(i,j) \mid i\ j.\ i \leq j \wedge j < k \wedge (\forall\ l.\ i \leq l \longrightarrow l < j \longrightarrow A\ \$\$\ (l,\ Suc\ l) = 1)$
$\wedge\ (Suc\ j \neq k \longrightarrow A\ \$\$\ (j,\ Suc\ j) \neq 1) \wedge (i > 0 \longrightarrow A\ \$\$\ (i - 1,\ i) \neq 1)\}$
unfolding *identify-blocks-def* **using** *identify-blocks-main-iff*[*OF le-refl* - ‹$k < n$›]
by *auto*

private lemma *identify-blocksD*: **assumes** $k < n$ **and** $(i,j) \in set\ (identify\text{-}blocks\ A\ k)$
shows $i \leq j\ j < k$
$\bigwedge\ l.\ i \leq l \Longrightarrow l < j \Longrightarrow A\ \$\$\ (l,\ Suc\ l) = 1$
$Suc\ j \neq k \Longrightarrow A\ \$\$\ (j,\ Suc\ j) \neq 1$
$i > 0 \Longrightarrow A\ \$\$\ (i - 1,\ i - 1) \neq A\ \$\$\ (k,k) \vee A\ \$\$\ (i - 1,\ i) \neq 1$
using *assms* **unfolding** *identify-blocks-iff*[*OF assms*(*1*)] **by** *auto*

private lemma *identify-blocksI*: **assumes** *inv*: $k < n$
$i \leq j\ j < k\ \bigwedge\ l.\ i \leq l \Longrightarrow l < j \Longrightarrow A\ \$\$\ (l,\ Suc\ l) = 1$
$Suc\ j \neq k \Longrightarrow A\ \$\$\ (j,\ Suc\ j) \neq 1\ i > 0 \Longrightarrow A\ \$\$\ (i - 1,\ i) \neq 1$
shows $(i,j) \in set\ (identify\text{-}blocks\ A\ k)$
unfolding *identify-blocks-iff*[*OF inv*(*1*)] **using** *inv* **by** *blast*

private lemma *identify-blocks-rev*: **assumes** $A\ \$\$\ (i,\ Suc\ i) = 0 \wedge Suc\ i < k \vee Suc\ i = k$
and *inv*: $k < n$
shows $(identify\text{-}block\ A\ i,\ i) \in set\ (identify\text{-}blocks\ A\ k)$
proof −
obtain *j* **where** *id*: *identify-block* $A\ i = j$ **by** *force*
note *idb* = *identify-block*[*OF this*]
show *?thesis* **unfolding** *id*
by (*rule identify-blocksI*[*OF inv*], *insert idb assms*, *auto*)
qed

18.8 Proving the Invariants

private lemma *add-col-sub-row-diag*: **assumes** *A*: $A \in carrier\text{-}mat\ n\ n$
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 −
from *inv-all-uppertD*[*OF ut*]
show *?thesis*
by (*subst add-col-sub-index-row*, *insert A ijk*, *auto*)
qed

private lemma *add-col-sub-row-diff-ev-part-old*: **assumes** *A*: $A \in carrier\text{-}mat\ n\ n$
and *ij*: $i \leq j\ i \neq 0\ i < n\ j < n\ i' < n\ j' < n$
and *choice*: $j' < j \vee j' = j \wedge i' \geq i$
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*) $i'\ j'$

unfolding *diff-ev-def*
proof (*intro impI*)
assume *ij′*: $i' < j'$
let *?A* = *add-col-sub-row a* (*i* − *1*) *j A*
assume *neq*: *?A* \$\$ (*i′*,*i′*) ≠ *?A* \$\$ (*j′*,*j′*)
from *A* **have** *dim*: *dim-row A* = *n dim-col A* = *n* **by** *auto*
note *utd* = *inv-all-uppertD*[*OF ut*]
let *?i* = *i* − *1*
have *?i* < *j* **using** ‹*i* ≤ *j*› ‹*i* ≠ *0*› ‹*i* < *n*› **by** *auto*
from *utd*[*OF this* ‹*j* < *n*›] **have** *Aji*: *A* \$\$ (*j*,*?i*) = *0* **by** *simp*
from *add-col-sub-row-diag*[*OF A ut* ‹*?i* < *j*› ‹*j* < *n*›]
have *diag*: ⋀ *k*. *k* < *n* ⟹ *?A* \$\$ (*k*,*k*) = *A* \$\$ (*k*,*k*) **.**
from *neq*[*unfolded diag*[*OF* ‹*i′* < *n*›] *diag*[*OF* ‹*j′* < *n*›]]
have *neq*: *A* \$\$ (*i′*,*i′*) ≠ *A* \$\$ (*j′*,*j′*) **by** *auto*
{
from *inv-partD*(*3*)[*OF old* ‹*i′* < *n*› ‹*j′* < *n*› *choice*]
have *diff-ev A i′ j′* **by** *auto*
with *neq ij′* **have** *A* \$\$ (*i′*,*j′*) = *0* **unfolding** *diff-ev-def* **by** *auto*
} **note** *zero* = *this*
{
assume *i′* ≠ *?i j′* = *j*
with *ij′ ij*(*1*) *choice* **have** *i′* > *?i* **by** *auto*
from *utd*[*OF this*] *ij*
have *A* \$\$ (*i′*, *?i*) = *0* **by** *auto*
} **note** *1* = *this*
{
assume *j′* ≠ *j i′* = *?i*
with *ij′ ij*(*1*) *choice* **have** *j* > *j′* **by** *auto*
from *utd*[*OF this*] *ij*
have *A* \$\$ (*j*, *j′*) = *0* **by** *auto*
} **note** *2* = *this*
from *ij′ ij choice* **have** (*i′* = *?i* ∧ *j′* = *j*) = *False* **by** *arith*
note *id* = *add-col-sub-index-row*[*of i′ A j′ j a ?i*, *unfolded dim this if-False*,
OF ‹*i′* < *n*› ‹*i′* < *n*› ‹*j′* < *n*› ‹*j′* < *n*› ‹*j* < *n*›]
show *?A* \$\$ (*i′*,*j′*) = *0* **unfolding** *id zero* **using** *1 2* **by** *auto*
qed

private 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*)
unfolding *inv-all-def uppert-def*
proof (*intro allI impI*)
fix *i′ j′*
assume ∗: *i′* < *n j′* < *n j′* < *i′*
note *inv* = *inv-allD*[*OF inv*, *unfolded uppert-def*]
show *add-col-sub-row a i j A* \$\$ (*i′*, *j′*) = *0*
by (*subst add-col-sub-index-row*, *insert assms* ∗ *inv*, *auto*)

qed

private lemma *step-1-main-inv*: $i \leq j$
$\Longrightarrow A \in carrier\text{-}mat\ n\ n$
$\Longrightarrow inv\text{-}all\ uppert\ A$
$\Longrightarrow inv\text{-}part\ diff\text{-}ev\ A\ i\ j$
$\Longrightarrow inv\text{-}all\ uppert\ (step\text{-}1\text{-}main\ n\ i\ j\ A) \wedge inv\text{-}all\ diff\text{-}ev\ (step\text{-}1\text{-}main\ n\ i\ j\ A)$
proof (*induct i j A taking*: *n rule*: *step-1-main.induct*)
case (*1 i j A*)
let $?i = i - 1$
note [*simp*] = *step-1-main.simps*[*of n i j A*]
from *1*(*3*−) **have** *ij*: $i \leq j$ **and** *A*: $A \in carrier\text{-}mat\ n\ n$ **and** *inv*: *inv-all uppert A*
inv-part diff-ev A i j **by** *auto*
show *?case*
proof (*cases* $j \geq n$)
case *True*
thus *?thesis* **using** *inv* **by** (*simp add*: *inv-all-eq-inv-part*, *auto simp*: *inv-part-def*)
next
case *False*
hence *jn*: $j < n$ **by** *simp*
note $IH = 1(1-2)$[*OF False*]
show *?thesis*
proof (*cases* $i = 0$)
case *True*
from *inv*[*unfolded True inv-part-0-Suc*[*OF jn*]]
have *inv2*: *inv-part diff-ev A n* $(j + 1)$ **by** *simp*
have *inv-part diff-ev A* $(j + 1)$ $(j + 1)$
proof (*intro inv-partI*)
fix $i'\ j'$
assume *ij*: $i' < n$ $j' < n$ **and** *choice*: $j' < j + 1 \vee j' = j + 1 \wedge j + 1 \leq i'$
from *inv-partD*[*OF inv2 ij*] *choice*
show *diff-ev A* $i'\ j'$ **using** *jn* **unfolding** *diff-ev-def* **by** *auto*
qed
from *IH*(*1*)[*OF True - A inv*(*1*) *this*]
show *?thesis* **using** *jn* **by** (*simp*, *simp add*: *True*)
next
case *False*
let $?evi = A\ \$\$\ (?i,?i)$
let $?evj = A\ \$\$\ (j,j)$
let $?choice = ?evi \neq ?evj \wedge A\ \$\$\ (?i,\ j) \neq 0$
let $?A = add\text{-}col\text{-}sub\text{-}row\ (A\ \$\$\ (?i,\ j)\ /\ (?evj - ?evi))\ ?i\ j\ A$
let $?B = if\ ?choice\ then\ ?A\ else\ A$
obtain *B* **where** *B*: $B = ?B$ **by** *auto*
have *Bn*: $B \in carrier\text{-}mat\ n\ n$ **unfolding** *B* **using** *A* **by** *simp*
from *False ij jn* **have** $*$: $?i < j$ $j < n$ $?i < n$ **by** *auto*
have *inv1*: *inv-all uppert B* **unfolding** *B* **using** *inv add-col-sub-row-uppert*[*OF*
A $*(1-2)$ *inv*(*1*)]
by *auto*

note *inv2* = *inv-partD*[*OF inv*(*2*)]
have *inv2*: *inv-part diff-ev B ?i j*
proof (*cases ?choice*)
case *False*
hence *B*: $B = A$ **unfolding** *B* **by** *auto*
show *?thesis* **unfolding** *B*
proof (*rule inv-partI*)
fix i' j'
assume *ij*: $i' < n$ $j' < n$ **and** $j' < j \lor j' = j \land ?i \leq i'$
hence *choice*: $(j' < j \lor j' = j \land i \leq i') \lor j' = j \land i' = ?i$ **by** *auto*
note *inv2* = *inv2*[*OF ij*]
from *choice*
show *diff-ev A* i' j'
proof
assume $j' < j \lor j' = j \land i \leq i'$
from *inv2*(*3*)[*OF this*] **show** *?thesis* **.**
next
assume $j' = j \land i' = ?i$
thus *?thesis* **using** *False* **unfolding** *diff-ev-def* **by** *auto*
qed
qed
next
case *True*
hence *B*: $B = ?A$ **unfolding** *B* **by** *auto*
from ∗ *True* **have** $i < n$ **by** *auto*
note *old* = *add-col-sub-row-diff-ev-part-old*[*OF A* ‹$i \leq j$› ‹$i \neq 0$› ‹$i < n$› ‹j
$< n$›
\- - - *inv*(*2*) *inv*(*1*)]
show *?thesis* **unfolding** *B*
proof (*rule inv-partI*)
fix i' j'
assume *ij*: $i' < n$ $j' < n$ **and** $j' < j \lor j' = j \land ?i \leq i'$
hence *choice*: $(j' < j \lor j' = j \land i \leq i') \lor j' = j \land i' = ?i$ **by** *auto*
note *inv2* = *inv2*[*OF ij*]
from *choice*
show *diff-ev ?A* i' j'
proof
assume $j' < j \lor j' = j \land i \leq i'$
from *old*[*OF ij this*] **show** *?thesis* **.**
next
assume $j' = j \land i' = ?i$
hence *ij'*: $j' = j$ $i' = ?i$ **by** *auto*
note *diag* = *add-col-sub-row-diag*[*OF A inv*(*1*) ‹$?i < j$› ‹$j < n$›]
show *?thesis* **unfolding** *ij'* *diff-ev-def diag*[*OF* ‹$j < n$›] *diag*[*OF* ‹$?i <$
n›]
proof (*intro impI*)
from *True* **have** *neq*: $?evi \neq ?evj$ **by** *simp*
note *ut* = *inv-all-uppertD*[*OF inv*(*1*)]
obtain i' **where** i': $i' = i - Suc\ 0$ **by** *auto*

obtain *diff* **where** *diff*: *diff* = *?evj* − *A* $$ (*i′*,*i′*) **by** *auto*
from *neq* **have** [*simp*]: *diff* ≠ *0* **unfolding** *diff i′* **by** *auto*
from *ut*[*OF* ‹*?i* < *j*› ‹*j* < *n*›] **have** [*simp*]: *A* $$ (*j*,*i′*) = *0* **unfolding** *diff i′* **by** *simp*
have *?A* $$ (*?i*, *j*) =
A $$ (*i′*, *j*) + (*A* $$ (*i′*, *j*) ∗ *A* $$ (*i′*, *i′*) −
A $$ (*i′*, *j*) ∗ *A* $$ (*j*, *j*)) / *diff*
by (*subst add-col-sub-index-row*, *insert A* ∗, *auto simp*: *diff*[*symmetric*] *i′*[*symmetric*] *field-simps*)
also have *A* $$ (*i′*, *j*) ∗ *A* $$ (*i′*, *i′*) − *A* $$ (*i′*, *j*) ∗ *A* $$ (*j*, *j*)
= − *A* $$ (*i′*,*j*) ∗ *diff* **by** (*simp add*: *diff i′ field-simps*)
also have . . . / *diff* = − *A* $$ (*i′*,*j*) **by** *simp*
finally show *?A* $$ (*?i*,*j*) = *0* **by** *simp*
qed
qed
qed
qed
from *ij* **have** *i* − *1* ≤ *j* **by** *simp*
note *IH* = *IH*(*2*)[*OF False refl refl refl refl B this Bn inv1 inv2*]
from *False jn* **have** *id*: *step-1-main n i j A* = *step-1-main n* (*i* − *1*) *j B*
unfolding *B* **by** (*simp add*: *Let-def*)
show *?thesis* **unfolding** *id* **by** (*rule IH*)
qed
qed
qed

private lemma *step-2-main-inv*: *A* ∈ *carrier-mat n n*
⟹ *inv-all uppert A*
⟹ *inv-all diff-ev A*
⟹ *ev-blocks-part j A*
⟹ *inv-all uppert* (*step-2-main n j A*) ∧ *inv-all diff-ev* (*step-2-main n j A*)
∧ *ev-blocks* (*step-2-main n j A*)
proof (*induct j A taking*: *n rule*: *step-2-main.induct*)
case (*1 j A*)
note [*simp*] = *step-2-main.simps*[*of n j A*]
from *1*(*2*−) **have** *A*: *A* ∈ *carrier-mat n n*
and *inv*: *inv-all uppert A inv-all diff-ev A ev-blocks-part j A* **by** *auto*
show *?case*
proof (*cases j* ≥ *n*)
case *True*
with *inv*(*3*) **have** *ev-blocks A* **unfolding** *ev-blocks-def ev-blocks-part-def* **by** *auto*
thus *?thesis* **using** *True inv*(*1*−*2*) **by** *auto*
next
case *False*
hence *jn*: *j* < *n* **by** *simp*
note *intro* = *ev-blocks-partI*
note *dest* = *ev-blocks-partD*
note *IH* = *1*(*1*)[*OF False*]

let *?look* = *lookup-ev* (*A* \$\$ (*j*,*j*)) *j* *A*
let *?B* = *case* *?look* *of*
None ⇒ *A*
| *Some* *i* ⇒ *swap-cols-rows-block* (*Suc* *i*) *j* *A*
obtain *B* **where** *B*: *B* = *?B* **by** *auto*
have *id*: *step-2-main* *n* *j* *A* = *step-2-main* *n* (*Suc* *j*) *B* **unfolding** *B* **using** *False* **by** *simp*
have *Bn*: *B* ∈ *carrier-mat* *n* *n* **unfolding** *B* **using** *A* **by** (*auto* *split*: *option.splits*)
have *inv-all* *uppert* *B* ∧ *inv-all* *diff-ev* *B* ∧ *ev-blocks-part* (*Suc* *j*) *B*
proof (*cases* *?look*)
case *None*
have *ev-blocks-part* (*Suc* *j*) *A*
proof (*intro* *intro*)
fix *i′* *j′* *k′*
assume ∗: *i′* < *j′* *j′* < *k′* *k′* < *Suc* *j* *A* \$\$ (*k′*,*k′*) = *A* \$\$ (*i′*,*i′*)
show *A* \$\$ (*j′*,*j′*) = *A* \$\$ (*i′*,*i′*)
proof (*cases* *j* = *k′*)
case *False*
with ∗ **have** *k′* < *j* **by** *auto*
from *dest*[*OF* *inv*(*3*) ∗(*1*−*2*) *this* ∗(*4*)]
show *?thesis* **.**
next
case *True*
with *lookup-ev-None*[*OF* *None*, *of* *i′*] ∗ **have** *False* **by** *simp*
thus *?thesis* **..**
qed
qed
with *None* **show** *?thesis* **unfolding** *B* **using** *inv* **by** *auto*
next
case (*Some* *i*)
from *lookup-ev-Some*[*OF* *Some*]
have *ij*: *i* < *j* **and** *id*: *A* \$\$ (*i*, *i*) = *A* \$\$ (*j*, *j*)
and *neq*: ⋀ *k*. *i* < *k* ⟹ *k* < *j* ⟹ *A* \$\$ (*k*,*k*) ≠ *A* \$\$ (*j*,*j*) **by** *auto*
let *?A* = *swap-cols-rows-block* (*Suc* *i*) *j* *A*
let *?perm* = λ *i′*. *if* *i′* = *Suc* *i* *then* *j* *else* *if* *Suc* *i* < *i′* ∧ *i′* ≤ *j* *then* *i′* − *1* *else* *i′*
from *Some* **have** *B*: *B* = *?A* **unfolding** *B* **by** *simp*
have *Aind*: ⋀ *i′* *j′*. *i′* < *n* ⟹ *j′* < *n* ⟹ *?A* \$\$ (*i′*, *j′*) = *A* \$\$ (*?perm* *i′*, *?perm* *j′*)
by (*subst* *swap-cols-rows-block-index*, *insert* *False* *A* *ij*, *auto*)
have *inv-ev*: *ev-blocks-part* (*Suc* *j*) *?A*
proof (*intro* *intro*)
fix *i′* *j′* *k*
assume ∗: *i′* < *j′* *j′* < *k* *k* < *Suc* *j* **and** *ki*: *?A* \$\$ (*k*,*k*) = *?A* \$\$ (*i′*,*i′*)
from ∗ *jn* **have** *j′* < *n* *i′* < *n* *k* < *n* **by** *auto*
note *id′* = *Aind*[*OF* ‹*j′* < *n*› ‹*j′* < *n*›] *Aind*[*OF* ‹*i′* < *n*› ‹*i′* < *n*›] *Aind*[*OF* ‹*k* < *n*› ‹*k* < *n*›]
note *inv-ev* = *dest*[*OF* *inv*(*3*)]

```
    show ?A $$ (j',j') = ?A $$ (i',i')
    proof (cases i' < Suc i)
      case True note i' = this
      hence pi: ?perm i' = i' by simp
      show ?thesis
      proof (cases j' < Suc i)
        case True note j' = this
        hence pj: ?perm j' = j' by simp
        show ?thesis
        proof (cases k < Suc i)
          case True note k = this
          hence pk: ?perm k = k by simp
          from True ij have k < j by simp
          from inv-ev[OF *(1−2) this] ki
          show ?thesis unfolding id' pi pj pk by auto
        next
          case False note kf1 = this
          show ?thesis
          proof (cases k = Suc i)
            case True note k = this
            hence pk: ?perm k = j by simp
             from ki id have ii': A $$ (i, i) = A $$ (i', i') unfolding id' pi pj
pk by simp
            have ji: A $$ (j',j') = A $$ (i',i')
            proof (cases j' = i)
              case True
              with ii' show ?thesis by simp
            next
              case False
              with ‹j' < Suc i› have j' < i by auto
              from ki id inv-ev[OF ‹i' < j'› this ij] show ?thesis
                unfolding id' pi pj pk by simp
            qed
            thus ?thesis unfolding id' pi pj pk .
          next
            case False note kf2 = this
            with kf1 have k: k > Suc i by auto
            hence pk: ?perm k = k − 1 and kj: k − 1 < j
              using * ‹k < Suc j› by auto
            from k j' have j' < k − 1 by auto
            from inv-ev[OF *(1) this kj] ki
            show ?thesis unfolding id' pi pj pk by simp
          qed
        qed
      next
        case False note j'f1 = this
        show ?thesis
        proof (cases j' = Suc i)
          case True note j' = this
```

```
            hence pj: ?perm j' = j by simp
            from j' * have k: k > Suc i by auto
            hence pk: ?perm k = k - 1 and kj: k - 1 < j
              using * ‹k < Suc j› by auto
             from ki[unfolded id' pi pj pk] have eq: A $$ (k - 1, k - 1) = A $$
(i', i') .
            from * i' k have le: i' ≤ i and lt: i < k - 1 k - 1 < j by auto
            from inv-ev[OF - lt eq] le have A $$ (i, i) = A $$ (i', i')
              by (cases i = i', auto)
            with id show ?thesis unfolding id' pi pj pk by simp
          next
            case False note j'f2 = this
            with j'f1 have j' > Suc i by auto
            hence pj: ?perm j' = j' - 1 and pk: ?perm k = k - 1
              and kj: i' < j' - 1 j' - 1 < k - 1 k - 1 < j
              using * i' ‹k < Suc j› by auto
            from inv-ev[OF kj] ki
            show ?thesis unfolding id' pi pj pk by simp
          qed
        qed
      next
        case False note i'f1 = this
        show ?thesis
        proof (cases i' = Suc i)
          case True note i' = this
          with * have gt: i < k - 1 k - 1 < j
            and perm: ?perm i' = j ?perm k = k - 1 by auto
          from ki[unfolded id' perm] neq[OF gt] have False by auto
          thus ?thesis ..
        next
          case False note i'f2 = this
          with i'f1 have i' > Suc i by auto
          with * have gt: i' - 1 < j' - 1  j' - 1 < k - 1 k - 1 < j
            and perm: ?perm i' = i' - 1 ?perm j' = j' - 1 ?perm k = k - 1 by
auto
          show ?thesis using inv-ev[OF gt] ki
            unfolding id' perm by simp
        qed
      qed
    qed
    let ?both = λ A i j. uppert A i j ∧ diff-ev A i j
    have inv-all ?both ?A
    proof (intro inv-allI)
      fix ii jj
      assume ii: ii < n and jj: jj < n
      note id = Aind[OF ii ii] Aind[OF jj jj] Aind[OF ii jj]
      note ut = inv-all-uppertD[OF inv(1)]
      note diff = inv-all-diff-evD[OF inv(2)]
      have upper: uppert ?A ii jj unfolding uppert-def
```

```
proof
  assume ji: jj < ii
  show ?A $$ (ii,jj) = 0
  proof (cases ii < Suc i)
    case True note i = this
    with ji have perm: ?perm ii = ii ?perm jj = jj by auto
    show ?thesis unfolding id perm using ut[OF ji ii] .
  next
    case False note if1 = this
    show ?thesis
    proof (cases ii = Suc i)
      case True note i = this
     with ji ij have perm: ?perm ii = j ?perm jj = jj and jj: jj < j by auto
      show ?thesis unfolding id perm
        by (rule ut[OF jj jn])
    next
      case False
      with if1 have if1: ii > Suc i by auto
      show ?thesis
      proof (cases ii ≤ j)
        case True note i = this
        with if1 have pi: ?perm ii = ii − 1 by auto
        show ?thesis
        proof (cases jj = Suc i)
          case True note j = this
          hence pj: ?perm jj = j by simp
          from i ji if1 ii j have ij: ii − 1 < j and ii: i < ii − 1 by auto
          show ?thesis unfolding id pi pj
            by (rule diff[OF ij jn neq[OF ii ij]])
        next
          case False
          with i ji if1 ii have ?perm jj < ii − 1 ii − 1 < n by auto
          from ut[OF this]
          show ?thesis unfolding id pi .
        qed
      next
        case False
        hence i: ii > j by auto
        with if1 have pi: ?perm ii = ii by simp
        from i ji if1 ii have ?perm jj < ii by auto
        from ut[OF this ii]
        show ?thesis unfolding id pi .
      qed
    qed
  qed
qed
have diff: diff-ev ?A ii jj unfolding diff-ev-def
proof (intro impI)
  assume ij': ii < jj and neq: ?A $$ (ii,ii) ≠ ?A $$ (jj,jj)
```

show *?A* $$ (*ii*,*jj*) = *0*
proof (*cases jj* < *Suc i*)
case *True* **note** *j* = *this*
with *ij′* **have** *perm*: *?perm ii* = *ii ?perm jj* = *jj* **by** *auto*
show *?thesis* **using** *neq* **unfolding** *id perm* **using** *diff*[*OF ij′ jj*] **by** *simp*
next
case *False* **note** *jf1* = *this*
show *?thesis*
proof (*cases jj* = *Suc i*)
case *True* **note** *j* = *this*
with *ij′ ij* **have** *perm*: *?perm jj* = *j ?perm ii* = *ii* **and** *ii*: *ii* < *j* **by** *auto*
show *?thesis* **using** *neq* **unfolding** *id perm*
by (*intro diff*[*OF ii jn*])
next
case *False*
with *jf1* **have** *jf1*: *jj* > *Suc i* **by** *auto*
show *?thesis*
proof (*cases jj* ≤ *j*)
case *True* **note** *j* = *this*
with *jf1* **have** *pj*: *?perm jj* = *jj* − *1* **by** *auto*
show *?thesis*
proof (*cases ii* = *Suc i*)
case *True* **note** *i* = *this*
hence *pi*: *?perm ii* = *j* **by** *simp*
from *i ij′ jf1 jj j* **have** *ij*: *jj* − *1* < *j* **by** *auto*
show *?thesis* **unfolding** *id pi pj*
by (*rule ut*[*OF ij jn*])
next
case *False*
with *j ij′ jf1 jj* **have** *?perm ii* < *jj* − *1 jj* − *1* < *n* **by** *auto*
from *diff*[*OF this*] *neq*
show *?thesis* **unfolding** *id pj* **.**
qed
next
case *False*
hence *j*: *jj* > *j* **by** *auto*
with *jf1* **have** *pj*: *?perm jj* = *jj* **by** *simp*
from *j ij′ jf1 jj* **have** *?perm ii* < *jj* **by** *auto*
from *diff*[*OF this jj*] *neq*
show *?thesis* **unfolding** *id pj* **.**
qed
qed
qed
qed
from *upper diff*
show *?both ?A ii jj* **..**
qed
hence *inv-all diff-ev ?A inv-all uppert ?A*

unfolding *inv-all-def* **by** *blast+*
with *inv-ev* **show** *?thesis* **unfolding** *B* **by** *auto*
qed
with *IH*[*OF refl B Bn*]
show *?thesis* **unfolding** *id* **by** *auto*
qed
qed

private 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*)
by (*intro same-uptoI*, *subst add-col-sub-index-row*, *insert assms*, *auto simp*: *uppert-def inv-upto-def*)

private 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 −
note ∗ = ∗ ∗∗
let *?A* = *add-col-sub-row v i j A*
show *inv-from uppert ?A j* **unfolding** *inv-from-def*
proof (*intro allI impI*)
fix i' j'
assume ∗∗: $i' < n$ $j' < n$ $j < j'$
from ∗ ∗∗ **have** $i' <$ *dim-row A* $i' <$ *dim-col A* $j' <$ *dim-row A* $j' <$ *dim-col A* $j <$ *dim-row A* **by** *auto*
note *id2* = *add-col-sub-index-row*[*OF this*]
show *uppert ?A* i' j' **unfolding** *uppert-def*
proof (*intro conjI impI*)
assume $j' < i'$
with *inv-fromD*[*OF* ‹*inv-from uppert A j*›, *unfolded uppert-def*, *of* i' j'] ∗ ∗∗
show *?A* $$ (i',j') = *0* **unfolding** *id2* **using** ∗ ∗∗ ‹$j' < i'$› **by** *simp*
qed
qed
qed

private 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*
∧ *inv-upto jb* (*step-3-a i j A*) *j*
∧ *inv-at one-zero* (*step-3-a i j A*) *j* ∧ *same-diag A* (*step-3-a i j A*)
proof (*induct i j A rule*: *step-3-a.induct*)
case (*1 j A*)
thus *?case* **by** (*simp add*: *inv-from-bot-def inv-at-def*)

next
case (*2 i j A*)
from *2*(*2*−) **have** *A*: *A* ∈ *carrier-mat n n* **and** *ij*: *Suc i* < *j i* < *j* **and** *j*: *j* < *n* **by** *auto*
let *?cond* = *A* $$ (*i*, *i* + *1*) = *1* ∧ *A* $$ (*i*, *j*) ≠ *0*
let *?B* = *add-col-sub-row* (− *A* $$ (*i*, *j*)) (*Suc i*) *j A*
obtain *B* **where** *B*: *B* = (*if ?cond then ?B else A*) **by** *auto*
from *A* **have** *Bn*: *B* ∈ *carrier-mat n n* **unfolding** *B* **by** *simp*
note *IH* = *2*(*1*)[*OF refl B Bn ij*(*2*) *j*]
have *id*: *step-3-a* (*Suc i*) *j A* = *step-3-a i j B* **unfolding** *B* **by** (*simp add*: *Let-def*)
from *ij j* **have** ∗: *Suc i* < *n j* < *n Suc i* ≠ *j* **by** *auto*
from *2*(*2*−) **have** *inv*: *inv-upto jb A j inv-from uppert A j ev-block n A*
inv-from-bot (λ*A i*. *one-zero A i j*) *A* (*Suc i*) **by** *auto*
note *evbA* = *ev-blockD*[*OF inv*(*3*)]
show *?case*
proof (*cases ?cond*)
case *False*
hence *B*: *B* = *A* **unfolding** *B* **by** *auto*
have *inv2*: *inv-from-bot* (λ*A i*. *one-zero A i j*) *A i*
by (*rule inv-from-bot-step*[*OF - inv*(*4*)],
insert False ij evbA[*of i j*] ∗, *auto simp*: *one-zero-def*)
show *?thesis* **unfolding** *id B*
by (*rule IH*[*unfolded B*], *insert inv inv2*, *auto*)
next
case *True*
hence *B*: *B* = *?B* **unfolding** *B* **by** *auto*
let *?C* = *step-3-a i j B*
from *inv-uptoD*[*OF inv*(*1*) *j* ∗(*1*) *ij*(*1*), *unfolded jb-def*] *ij*
have *Aji*: *A* $$ (*j*, *Suc i*) = *0* **by** *auto*
have *diag*: *same-diag A B* **unfolding** *same-diag-def*
by (*intro allI impI*, *insert ij j A Aji B*, *auto*)
have *upto*: *same-upto j A B* **unfolding** *B*
by (*rule add-col-sub-row-same-upto*[*OF* ‹*Suc i* < *j*› ‹*j* < *n*› *A inv-upto-mono*[*OF jb-imp-uppert inv*(*1*)]]])
from *add-col-sub-row-inv-from-uppert*[*OF inv*(*2*) *A* ‹*Suc i* < *n*› ‹*Suc i* < *j*› ‹*j* < *n*›]
have *from-j*: *inv-from uppert B j* **unfolding** *B* **by** *blast*
have *ev*: *A* $$ (*Suc i*, *Suc i*) = *A* $$ (*j*,*j*) **using** *evbA*[*of Suc i j*] *ij j* **by** *auto*
have *evb-B*: *ev-block n B*
by (*rule same-diag-ev-block*[*OF diag inv*(*3*)])
note *evbB* = *ev-blockD*[*OF evb-B*]
{
fix *k*
assume *k* < *n*
with *A* ∗ **have** *k*: *k* < *dim-row A k* < *dim-col A j* < *dim-row A j* < *dim-col A j* < *dim-row A* **by** *auto*
note *id* = *B add-col-sub-index-row*[*OF k*]
have *B* $$ (*k*,*j*) = (*if k* = *i then 0 else A* $$ (*k*,*j*)) **unfolding** *id*

using *inv-uptoD*[*OF inv*(*1*), *of k Suc i*, *unfolded jb-def*]
by (*insert* ∗ *Aji True ij* ‹*k* < *n*›, *auto simp*: *ev*)
} **note** *id2* = *this*
have *inv-from-bot* (*λA i*. *one-zero A i j*) *B i* **unfolding** *inv-from-bot-def*
proof (*intro allI impI*)
fix *k*
assume *i* ≤ *k k* < *n*
thus *one-zero B k j* **using** *inv*(*4*)[*unfolded inv-from-bot-def*]
upto[*unfolded same-upto-def*] *evbB*[*OF* ‹*k* < *n*› ‹*j* < *n*›]
unfolding *one-zero-def id2*[*OF* ‹*k* < *n*›] **by** *auto*
qed
from *IH*[*OF same-upto-inv-upto-jb*[*OF upto inv*(*1*)] *from-j this evb-B*]
same-diag-trans[*OF diag*]
show *?thesis* **unfolding** *id* **by** *blast*
qed
qed

private lemma *identify-block-cong*: **assumes** *su*: *same-upto k A B* **and** *kn*: *k* < *n*
shows *i* < *k* ⟹ *identify-block A i* = *identify-block B i*
proof (*induct i*)
case (*Suc i*)
hence *i* < *k* **by** *auto*
note *IH* = *Suc*(*1*)[*OF this*]
let *?c* = *λ A*. *A* $$ (*i*,*Suc i*) = *1*
from *same-uptoD*[*OF su*, *of i Suc i*] *kn Suc*(*2*) **have** *1*: *A* $$ (*i*, *Suc i*) = *B* $$ (*i*, *Suc i*) **by** *auto*
from *1* **have** *id*: *?c A* = *?c B* **by** *simp*
show *?case*
proof (*cases ?c A*)
case *True*
with *True*[*unfolded id*] *IH* **show** *?thesis* **by** *simp*
next
case *False*
with *False*[*unfolded id*] **show** *?thesis* **by** *auto*
qed
qed *simp*

private lemma *identify-blocks-main-cong*:
k < *n* ⟹ *same-upto k A B* ⟹ *identify-blocks-main A k xs* = *identify-blocks-main B k xs*
proof (*induct k arbitrary*: *xs rule*: *less-induct*)
case (*less k list*)
show *?case*
proof (*cases k* = *0*)
case *False*
then obtain *i-e* **where** *k*: *k* = *Suc i-e* **by** (*cases k*, *auto*)
obtain *i-b* **where** *idA*: *identify-block A i-e* = *i-b* **by** *force*
from *identify-block-le′*[*OF idA*] **have** *ibe*: *i-b* ≤ *i-e* **.**

have *idB*: *identify-block B i-e* = *i-b* **unfolding** *idA*[*symmetric*]
by (*rule sym*, *rule identify-block-cong*, *insert k less*(*2*−*3*), *auto*)
let *?I* = *identify-blocks-main*
let *?resA* = *?I A* (*Suc i-e*) *list*
let *?recA* = *?I A i-b* ((*i-b*, *i-e*) # *list*)
let *?resB* = *?I B* (*Suc i-e*) *list*
let *?recB* = *?I B i-b* ((*i-b*, *i-e*) # *list*)
have *res*: *?resA* = *?recA ?resB* = *?recB* **using** *idA idB* **by** *auto*
from *k ibe* **have** *ibk*: *i-b* < *k* **by** *simp*
with *less*(*3*) **have** *same-upto i-b A B* **unfolding** *same-upto-def* **by** *auto*
from *less*(*1*)[*OF ibk - this*] *ibk* ‹*k* < *n*› **have** *?recA* = *?recB* **by** *auto*
thus *?thesis* **unfolding** *k res* **by** *simp*
qed *simp*
qed

private lemma *identify-blocks-cong*:
k < *n* ⟹ *same-diag A B* ⟹ *same-upto k A B* ⟹ *identify-blocks A k* = *identify-blocks B k*
unfolding *identify-blocks-def*
by (*intro identify-blocks-main-cong*, *auto simp*: *same-diag-def*)

private 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*: ⋀ *i*. *i* < *n* ⟹ *p A i k* ⟹ *uppert A i k*
and *k*: *k* < *n*
shows *inv-all uppert A*
proof (*rule inv-from-upto-at-all*[*OF jb - ut at*])
from *ev-blockD*[*OF evb*]
show *inv-from diff-ev A k* **unfolding** *inv-from-def diff-ev-def* **by** *blast*
fix *i*
assume *i* < *n p A i k*
with *ev-blockD*[*OF evb k*, *of i*] *p*[*OF this*] *k*
show *diff-ev A i k* ∧ *uppert A i k*
unfolding *diff-ev-def* **by** *auto*
qed

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* ∈ *carrier-mat n n*
and *lbl*: (*lb*,*l*) ∈ *set* (*identify-blocks A k*)

```
  and ib-block: (i-begin,i-end) ∈ set (identify-blocks A k)
  and il: i-end ≠ l
  and large: l − lb ≥ 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 −
  let ?Alk = A $$ (l,k)
  let ?Aik = A $$ (i-end,k)
  define quot where quot = ?Aik / ?Alk
  let ?idiff = i-end − i-begin
  let ?m = λ iter diff i j. if (i,j) = (i-end,k) then if diff = (Suc ?idiff) then ?Aik else 0
    else if i ≥ i-begin + diff ∧ i ≤ i-end ∧ k < j then A $$ (i, j) − quot ∗ A $$ (l + i − i-end, j)
    else if (i,j) = (i-end − iter, Suc l − iter) ∧ iter ∉ {0, Suc ?idiff} then quot
    else A $$ (i,j)
  let ?mm = λ iter diff i j. if (i,j) = (i-end,k) then 0
    else if i ≥ i-begin + diff ∧ i ≤ i-end ∧ k < j then A $$ (i, j) − quot ∗ A $$ (l + i − i-end, j)
    else if (i,j) = (i-end − Suc iter, l − iter) ∧ iter ≠ ?idiff then quot
    else A $$ (i,j)
  let ?mat = λ iter diff. mat n n (λ (i,j). ?m iter diff i j)
  from identify-blocks[OF ib-block] have ib: i-begin ≤ i-end i-end < k by auto
  from identify-blocks[OF lbl] have lb: lb ≤ l l < k by auto
  have mend: ?mat 0 (Suc ?idiff) = A
    by (rule eq-matI, insert A ib, auto)
  {
    fix ll ii diff iter
    assume diff ≠ 0 ⟹ ii + iter = i-end diff ≠ 0 ⟹ ll + iter = l diff + iter = Suc ?idiff
    hence step-3-c-inner-loop quot ll ii diff (?mat iter diff) = ?R
    proof (induct diff arbitrary: ii ll iter)
      case 0
      hence iter: iter = Suc ?idiff by auto
      have step-3-c-inner-loop quot ll ii 0 (?mat iter 0) = ?mat (Suc ?idiff) 0
        unfolding iter step-3-c-inner-loop.simps ..
      also have . . . = ?R
        by (rule eq-matI, insert ib, auto simp: quot-def)
      finally show ?case .
    next
      case (Suc diff ii ll)
      note prems = Suc(2−)
      let ?B = ?mat iter (Suc diff)
```

have *step-3-c-inner-loop quot ll ii* (*Suc diff*) *?B*
= *step-3-c-inner-loop quot* (*ll* − *1*) (*ii* − *1*) *diff* (*add-col-sub-row quot ii ll ?B*)
by *simp*
also have *add-col-sub-row quot ii ll ?B*
= *?mat* (*Suc iter*) *diff* (**is** *?C* = *?D*)
proof (*rule eq-matI*, *unfold dim-row-mat dim-col-mat*)
fix *i j*
assume *i*: *i* < *n* **and** *j*: *j* < *n*
have *ll*: *ll* < *n* **using** *prems lb k* **by** *auto*
from *prems ib k* **have** *ii*: *ii* ≥ *i-begin ii* < *n ii* < *k ii* ≤ *i-end*
and *eqs*: *ii* + *iter* = *i-end ll* + *iter* = *l Suc diff* + *iter* = *Suc ?idiff* **by** *auto*
from *eqs* **have** *diff*: *diff* < *Suc ?idiff* **by** *auto*
from *eqs lb* ‹*k* < *n*› **have** *ll* < *k l* < *n* **by** *auto*
note *index* = *ib lb k i j ll il large ii this*
let *?Aij* = *A* $$ (*i*,*j*)
have *D*: *?D* $$ (*i*,*j*) = *?mm iter diff i j* **using** *diff i j* **by** (*auto split*: *if-splits*)
define *B* **where** *B* = *?B*
have *BB*: ⋀ *i j*. *i* < *n* ⟹ *j* < *n* ⟹ *B* $$ (*i*,*j*) = *?m iter* (*Suc diff*) *i j* **unfolding** *B-def* **by** *auto*
have *B*: *B* $$ (*i*,*j*) = *?m iter* (*Suc diff*) *i j* **by** (*rule BB*[*OF i j*])
have *C*: *?C* $$ (*i*, *j*) =
(*if i* = *ii* ∧ *j* = *ll then B* $$ (*i*, *j*) + *quot* ∗ *B* $$ (*i*, *i*) − *quot* ∗ *quot* ∗ *B* $$ (*j*, *i*) − *quot* ∗ *B* $$ (*j*, *j*)
else if i = *ii* ∧ *j* ≠ *ll then B* $$ (*i*, *j*) − *quot* ∗ *B* $$ (*ll*, *j*)
else if i ≠ *ii* ∧ *j* = *ll then B* $$ (*i*, *j*) + *quot* ∗ *B* $$ (*i*, *ii*)
else B $$ (*i*, *j*)) **unfolding** *B-def*
by (*rule add-col-sub-index-row*(*1*), *insert i j ll*, *auto*)
from *inv-from-upto-at-all-ev-block*[*OF inv*(*1*−*4*) - ‹*k* < *n*›]
have *invA*: *inv-all uppert A*
unfolding *one-zero-def uppert-def* **by** *auto*
note *ut* = *inv-all-uppertD*[*OF invA*]
note *jb* = *inv-uptoD*[*OF inv*(*1*), *unfolded jb-def*]
note *oz* = *inv-atD*[*OF inv*(*3*), *unfolded one-zero-def*]
note *evb* = *ev-blockD*[*OF inv*(*4*)]
note *iblock* = *identify-blocksD*[*OF* ‹*k* < *n*›]
note *ibe* = *iblock*[*OF ib-block*]
let *?ev* = λ *i*. *A* $$ (*i*,*i*)

{
fix *i ib ie*
assume (*ib*,*ie*) ∈ *set* (*identify-blocks A k*) **and** *i*: *ib* ≤ *i i* < *ie*
note *ibe* = *iblock*[*OF this*(*1*)]
from *ibe*(*3*)[*OF i*] **have** *id*: *A* $$ (*i*, *Suc i*) = *1* **by** *auto*
from *i ibe* ‹*k* < *n*› **have** *i* < *n Suc i* < *k* **by** *auto*
with *oz*[*OF this*(*1*)] *id*
have *A* $$ (*i*,*k*) = *0* **by** *auto*
} **note** *A-ik* = *this*

{
fix *i*
assume *i*: $i < n$ **and** $\neg$ ($i \geq$ *i-begin* $\wedge$ $i \leq$ *i-end*)
hence *choice*: $i >$ *i-end* $\vee$ $i <$ *i-begin* **by** *auto*
note *index* = *index i*
from *index eqs choice* **have** $i \neq ii$ **by** *auto*
{
assume *0*: *A* $$ (i,ii) $\neq$ *0*
from *0 ut*[*of ii*, *OF* - *i*] ‹$i \neq ii$› **have** $i < ii$ **by** *force*
from *choice index eqs this* **have** $i <$ *i-begin* **by** *auto*
with *index* **have** $i < k$ **by** *auto*
from *jb*[*OF i* ‹$ii < n$› ‹$ii < k$›] *0* ‹$i \neq ii$›
have $*$: *Suc i* = *ii A* $$ (i,ii) = *1 ?ev i* = *?ev ii* **by** *auto*
with *index* ‹$i <$ *i-begin*› **have** *ii* = *i-begin* **by** *auto*
with *evb*[*OF* ‹$i < n$› ‹$k < n$›] *ibe*(*5*) $*$ **have** *False* **by** *auto*
}
hence *Aii*: *A* $$ (i,ii) = *0* **by** *auto*
{
fix *j* **assume** *j*: $j < n$
have *B*: *B* $$ (i,j) = *?m iter* (*Suc diff*) *i j* **using** *i j* **unfolding** *B-def* **by** *simp*
from *choice* **have** *id*: ((i, j) = (*i-end* − *iter*, *Suc l* − *iter*) $\wedge$ *iter* $\notin$ {*0*, *Suc ?idiff*}) = *False*
using *ib index eqs* **by** *auto*
have *B* $$ (i,j) = *A* $$ (i,j) **unfolding** *B id* **using** *choice ib index* **by** *auto*
}
note *Aii this*
}
hence *A-outside-ii*: $\bigwedge$ *i*. $i < n \Longrightarrow \neg$ (*i-begin* $\leq i \wedge i \leq$ *i-end*) $\Longrightarrow$ *A* $$ (i, ii) = *0*
and *B-outside*: $\bigwedge$ *i j*. $i < n \Longrightarrow j < n \Longrightarrow \neg$ (*i-begin* $\leq i \wedge i \leq$ *i-end*) $\Longrightarrow$ *B* $$ (i, j) = *A* $$ (i, j) **by** *auto*

from *diff eqs* **have** *iter*: *iter* $\neq$ *Suc ?idiff* **by** *auto*
{
fix *ib ie jb je*
assume *i*: (*ib*,*ie*) $\in$ *set* (*identify-blocks A k*) **and**
j: (*jb*,*je*) $\in$ *set* (*identify-blocks A k*) **and** *lt*: *ie* < *je*
note *i* = *iblock*[*OF i*]
note *j* = *iblock*[*OF j*]
from *i j lt* **have** *Suc ie* < *k* **by** *auto*
with *i* **have** *Aie*: *A* $$ (*ie*, *Suc ie*) $\neq$ *1* **by** *auto*
have *ie* < *jb*
proof (*rule ccontr*)
assume $\neg$ *ie* < *jb*
hence *ie* $\geq$ *jb* **by** *auto*
from *j*(*3*)[*OF this lt*] *Aie* **show** *False* **by** *auto*

qed
} **note** *block-bounds* = *this*
{
assume *i-end* < *l*
from *block-bounds*[*OF ib-block lbl this*]
have *i-end* < *lb* **.**
} **note** *i-less-l* = *this*
{
assume *l* < *i-end*
from *block-bounds*[*OF lbl ib-block this*]
have *l* < *i-begin* **.**
} **note** *l-less-i* = *this*
{
assume *i-end* − *iter* = *Suc l* − *iter*
with *iter large eqs* **have** *i-end* = *Suc l* **by** *auto*
with *l-less-i* **have** *l* < *i-begin* **by** *auto*
with *index* ‹*i-end* = *Suc l*› **have** *i-begin* = *i-end* **by** *auto*
} **note** *block* = *this*
have *Alie*: *A* $$ (*l*, *i-end*) = *0*
proof (*cases l* < *i-end*)
case *True*
{
assume *nz*: *A* $$ (*l*, *i-end*) ≠ *0*
from *l-less-i*[*OF True*] *index* **have** *0* < *i-begin l* < *i-begin i-end* < *n* *i-end* < *k* **by** *auto*
from *jb*[*OF* ‹*l* < *n*› *this*(*3*−*4*)] *il nz*
have *i-end* = *Suc l A* $$ (*l*, *Suc l*) = *1* **by** *auto*
with *iblock*[*OF lbl*] **have** *k* = *Suc l* **by** *auto*
with ‹*i-end* = *Suc l*› ‹*i-end* < *k*› **have** *False* **by** *auto*
}
thus *?thesis* **by** *auto*
next
case *False*
with *il* **have** *i-end* < *l* **by** *auto*
from *ut*[*OF this* ‹*l* < *n*›] **show** *?thesis* **.**
qed
show *?C* $$ (*i*,*j*) = *?D* $$ (*i*,*j*)
proof (*cases i* ≥ *i-begin* ∧ *i* ≤ *i-end*)
case *False*
hence *choice*: *i* > *i-end* ∨ *i* < *i-begin* **by** *auto*
from *choice* **have** *id*: ((*i*, *j*) = (*i-end* − *Suc iter*, *l* − *iter*) ∧ *iter* ≠ *?idiff*) = *False*
using *ib index eqs* **by** *auto*
have *D*: *?D* $$ (*i*,*j*) = *?Aij* **unfolding** *D id* **using** *choice ib index* **by** *auto*
have *B*: *B* $$ (*i*,*j*) = *?Aij* **unfolding** *B-outside*[*OF i j False*] **..**
from *index eqs False* **have** *i* ≠ *ii* **by** *auto*
have *Bii*: *B* $$ (*i*, *ii*) = *A* $$ (*i*,*ii*) **unfolding** *B-outside*[*OF i* ‹*ii* < *n*› *False*] **..**
hence *C*: *?C* $$ (*i*,*j*) = *?Aij* **unfolding** *C B Bii* **using** ‹*i* ≠ *ii*›

A-outside-ii[*OF i False*] **by** *auto*
show *?thesis* **unfolding** *D C* **..**
next
case *True*
with *index* **have** *i-begin* $\leq$ *i i* $\leq$ *i-end i* $<$ *k* **by** *auto*
note *index* = *index this*
show *?thesis*
proof (*cases j* $>$ *k*)
case *True*
note *index* = *index this*
have *D*: *?D* $$ (*i*,*j*) = (*if i-begin* + *diff* $\leq$ *i then ?Aij* − *quot* ∗ *A* $$ (*l* + *i* − *i-end*, *j*) *else ?Aij*) **unfolding** *D*
using *index* **by** *auto*
have *B*: *B* $$ (*i*,*j*) = (*if i-begin* + *Suc diff* $\leq$ *i then ?Aij* − *quot* ∗ *A* $$ (*l* + *i* − *i-end*, *j*) *else ?Aij*) **unfolding** *B*
using *index* **by** *auto*
from *index eqs* **have** *j* $>$ *ll* **by** *auto*
hence *C*: *?C* $$ (*i*,*j*) = (*if i* = *ii then B* $$ (*i*, *j*) − *quot* ∗ *B* $$ (*ll*, *j*) *else B* $$ (*i*, *j*)) **unfolding** *C*
using *index* **by** *auto*
show *?thesis*
proof (*cases i-begin* + *Suc diff* $\leq$ *i* ∨ ¬ (*i-begin* + *diff* $\leq$ *i*))
case *True*
from *True eqs index* **have** *i* ≠ *ii* **by** *auto*
from *True* **have** *?D* $$ (*i*,*j*) = *B* $$ (*i*,*j*) **unfolding** *D B* **by** *auto*
also have *B* $$ (*i*,*j*) = *?C* $$ (*i*,*j*) **unfolding** *C* **using** ‹*i* ≠ *ii*› **by** *auto*
finally show *?thesis* **..**
next
case *False*
hence *i*: *i* = *i-begin* + *diff* **by** *simp*
with *eqs index* **have** *ii*: *ii* = *i* **by** *auto*
from *index eqs i ii* **have** *ll*: *ll* = *l* + *i* − *i-end* **by** *auto*
have *not*: ¬ (*i-begin* + *Suc diff* $\leq$ *ll* ∧ *ll* $\leq$ *i-end*)
proof
from *eqs* **have** *ll* $\leq$ *l* **by** *auto*
assume *i-begin* + *Suc diff* $\leq$ *ll* ∧ *ll* $\leq$ *i-end*
hence *i-begin* $<$ *ll ll* $\leq$ *i-end* **by** *auto*
with ‹*ll* $\leq$ *l*› **have** *i-begin* $<$ *l* **by** *auto*
with *l-less-i* **have** ¬ *l* $<$ *i-end* **by** *auto*
hence *l* $\geq$ *i-end* **by** *simp*
with *il i-less-l* **have** *i-end* $<$ *lb* **by** *auto*
from *index large eqs* **have** *lb* $\leq$ *ll* **by** *auto*
with ‹*i-end* $<$ *lb*› **have** *i-end* $<$ *ll* **by** *auto*
with ‹*ll* $\leq$ *i-end*›
show *False* **by** *auto*
qed
have *D*: *?D* $$ (*i*,*j*) = *?Aij* − *quot* ∗ *A* $$ (*ll*, *j*) **unfolding** *D* **unfolding** *i ll* **by** *simp*

have *C*: *?C* $$ (*i*,*j*) = *?Aij* − *quot* ∗ *B* $$ (*ll*, *j*) **unfolding** *C B* **unfolding** *ii i* **by** *simp*
have *B*: *B* $$ (*ll*, *j*) = *A* $$ (*ll*, *j*) **unfolding** *BB*[*OF* ‹*ll* < *n*› *j*] **using** *index not* **by** *auto*
show *?thesis* **unfolding** *C D B* **unfolding** *ii i* **by** (*simp split*: *if-splits*)
qed
next
case *False*
hence *j* < *k* ∨ *j* = *k* **by** *auto*
thus *?thesis*
proof
assume *jk*: *j* = *k*
hence *j* ≠ *Suc l* − *Suc iter* **using** *index* **by** *auto*
hence *?D* $$ (*i*,*j*) = (*if i* = *i-end then 0 else ?Aij*) **unfolding** *D* **using** *jk* **by** *auto*
also have ... = *0* **using** *A-ik*[*OF ib-block* ‹*i-begin* ≤ *i*›] ‹*i* ≤ *i-end*› **unfolding** *jk* **by** *auto*
finally have *D*: *?D* $$ (*i*,*j*) = *0* **.**
from *jk index* **have** *j* ≠ *ll* **by** *auto*
hence *C*: *?C* $$ (*i*,*j*) = (*if i* = *ii then B* $$ (*i*, *j*) − *quot* ∗ *B* $$ (*ll*, *j*) *else B* $$ (*i*, *j*))
unfolding *C* **unfolding** *jk* **by** *simp*
have *C*: *?C* $$ (*i*,*j*) = *0*
proof (*cases i* = *i-end*)
case *False*
with *index ii jk* **have** *i*: *i-begin* ≤ *i i* < *i-end* **by** *auto*
from *A-ik*[*OF ib-block this*] **have** *Aij*: *A* $$ (*i*,*j*) = *0* **unfolding** *jk* **.**
from *index i jk* **have** ¬ ((*i*, *j*) = (*i-end* − *iter*, *Suc l* − *iter*) ∧ *iter* ∉ {*0*, *Suc ?idiff*}) **by** *auto*
hence *Bij*: *B* $$ (*i*,*j*) = *0*
unfolding *B Aij* **using** *i jk* **by** *auto*
hence *C*: *?C* $$ (*i*,*j*) = (*if i* = *ii then* − *quot* ∗ *B* $$ (*ll*,*j*) *else 0*) **unfolding** *C* **by** *auto*
let *?l* = *l* − *iter*
from *index eqs* **have** *ll*: *ll* = *?l* **by** *auto*
show *?C* $$ (*i*,*j*) = *0*
proof (*cases i* = *ii*)
case *True*
with *index eqs i* **have** *l*: *lb* ≤ *?l ?l* < *l* **and** *diff*: *Suc diff* ≠ *Suc ?idiff* **by** *auto*
from *A-ik*[*OF lbl l*] **have** *Alj*: *A* $$ (*ll*,*j*) = *0* **unfolding** *jk ll* **.**
from *index l jk eqs* **have** ¬ ((*ll*, *j*) = (*i-end* − *iter*, *Suc l* − *iter*) ∧ *iter* ∉ {*0*, *Suc ?idiff*}) **by** *auto*
hence *Bij*: *B* $$ (*ll*,*j*) = *0* **unfolding** *BB*[*OF* ‹*ll* < *n*› *j*] *Alj*
using *l jk diff* **by** *auto*
thus *?thesis* **unfolding** *C* **by** *simp*
next
case *False*
thus *?thesis* **unfolding** *C* **by** *simp*

```
        qed
      next
        case True note i = this
        hence Bij: B $$ (i,j) = (if diff = ?idiff then A $$ (i-end, k) else 0)
unfolding B unfolding jk by auto
        show ?thesis
        proof (cases i = ii)
          case True
          with i eqs have diff = ?idiff ll = l iter = 0 by auto
          hence B: B $$ (i,j) = A $$ (i-end,k) unfolding Bij by auto
          have C: ?C $$ (i,j) = A $$ (i-end,k) − quot ∗ B $$ (l, k)
            unfolding C B unfolding True ‹ll =l› jk by simp
          also have B $$ (l,k) = A $$ (l,k)
            unfolding BB[OF ‹l < n› ‹k < n›] using il ‹iter = 0› by auto
             also have A $$ (i-end,k) − quot ∗ . . . = 0 unfolding quot-def
using Alk by auto
          finally show ?thesis .
        next
          case False
          with i eqs have diff ≠ ?idiff by auto
          thus ?thesis unfolding C Bij using False by auto
        qed
      qed
      show ?thesis unfolding C D ..
    next
      assume jk: j < k
      from eqs il have ii ≠ ll by auto
      show ?thesis
      proof (cases diff = 0 ∨ (i,j) ≠ (ii − 1,ll))
        case False
        with eqs have ∗∗: i = i-end − Suc iter j = l − iter iter ≠ ?idiff
          and ∗: diff ≠ 0 i = ii − 1 j = ll ii ≠ 0 i ≠ ii by auto
        hence D: ?D $$ (i,j) = quot unfolding D using jk index by auto
        from ∗ index eqs False jk have i: ii = Suc i i < i-end by auto
        from iblock(3)[OF ib-block ‹i-begin ≤ i› ‹i < i-end›]
        have Ai: A $$ (i, ii) = 1 unfolding i .
        have ii < k i ≠ i-end − iter using index ∗ ∗∗ eqs
          by (blast, force)
         hence Bi: B $$ (i,ii) = 1 unfolding BB[OF ‹i < n› ‹ii < n›] Ai
by auto
        have B $$ (i,ll) = A $$ (i,ll) unfolding BB[OF ‹i < n› ‹ll < n›]
          using ‹i ≠ i-end − iter› ‹ll < k› by auto
        also have A $$ (i,ll) = 0
        proof (rule ccontr)
          assume nz: A $$ (i,ll) ≠ 0
          from i eqs il have neq: Suc i ≠ ll by auto
          from jb[OF ‹i < n› ‹ll < n› ‹ll < k›] nz neq
          have i = ll by auto
          with i have ii = Suc ll by simp
```

hence *i-end* − *iter* = *Suc l* − *iter* **using** *eqs* **by** *auto*
from *block*[*OF this*] **have** *i-begin* = *i-end* **by** *auto*
with *large ib lb index* **have** *i* = *ii* **by** *auto*
with ∗ **show** *False* **by** *auto*
qed
finally have *C*: *?C* $$ (*i*,*j*) = *quot* **unfolding** *C* **using** ∗ *Bi* **by** *auto*
show *?thesis* **unfolding** *C D* **..**
next
case *True*
with *eqs* **have** ¬ ((*i*, *j*) = (*i-end* − *Suc iter*, *l* − *iter*) ∧ *iter* ≠ *?idiff*)

and *not*: ¬ (*i* = *ii* − *1* ∧ *j* = *ll* ∧ *iter* ≠ *?idiff*) **by** *auto*
from *this*(*1*) **have** *D*: *?D* $$ (*i*,*j*) = *?Aij* **unfolding** *D* **using** *jk index* **by** *auto*
{
fix *i*
assume *i* < *n*
with *index* **have** *id*: ((*i*,*i*) = (*i-end*,*k*)) = *False* (*i-begin* + *Suc diff* ≤ *i* ∧ *i* ≤ *i-end* ∧ *k* < *i*) = *False* **by** *auto*
{
assume ∗: (*i*, *i*) = (*i-end* − *iter*, *Suc l* − *iter*) ∧ *iter* ∉ {*0*, *Suc ?idiff*}
hence *i-end* − *iter* = *Suc l* − *iter* **by** *auto*
from *block*[*OF this*] ∗ *index large eqs* **have** *False* **by** *auto*
}
hence *B* $$ (*i*,*i*) = *?ev i* **unfolding** *BB*[*OF* ‹*i* < *n*› ‹*i* < *n*›] *id if-False* **by** *auto*
} **note** *Bdiag* = *this*
from *eqs* **have** *ii*: *ii* = *i-end* − *iter Suc l* − *iter* = *Suc ll* **by** *auto*
have *B*: *B* $$ (*i*,*j*) =
(*if* (*i*, *j*) = (*ii*, *Suc ll*) ∧ *iter* ≠ *0 then quot else A* $$ (*i*, *j*))
unfolding *B* **using** *ii jk iter* **by** *auto*
have *ll-i*: *ll* ≠ *i-end* − *iter* **using** ‹*ii* ≠ *ll*› *eqs* **by** *auto*
have *B* $$ (*ll*,*ii*) = *A* $$ (*ll*,*ii*) **unfolding** *BB*[*OF* ‹*ll* < *n*› ‹*ii* < *n*›]
using ‹*ii* < *k*› *ll-i* **by** *auto*
also have . . . = *0*
proof (*rule ccontr*)
assume *nz*: *A* $$ (*ll*,*ii*) ≠ *0*
with *jb*[*OF* ‹*ll* < *n*› ‹*ii* < *n*› ‹*ii* < *k*›] ‹*ii* ≠ *ll*› **have** *Suc ll* = *ii* **by** *auto*
with *eqs* **have** *i-end* − *iter* = *Suc l* − *iter* **by** *auto*
from *block*[*OF this*] *index eqs* **have** *iter* = *0* **by** *auto*
with *ii* **have** *ll* = *l ii* = *i-end* **by** *auto*
with *Alie nz* **show** *False* **by** *auto*
qed
finally have *Bli*: *B* $$ (*ll*,*ii*) = *0* **.**
have *C*: *?C* $$ (*i*,*j*) = *?Aij*
proof (*cases i* = *j*)
case *True*

show *?thesis* **unfolding** *C* **unfolding** *Bdiag*[*OF* ‹*j* < *n*›] *True* **using** ‹*ii* ≠ *ll*› *Bli*
by *auto*
next
case *False*
from *lb eqs index large* **have** *lb* ≤ *ll ll* ≤ *l* **by** *auto*
note *C* = *C*[*unfolded Bdiag*[*OF* ‹*i* < *n*›] *Bdiag*[*OF* ‹*j* < *n*›]]
show *?thesis*
proof (*cases* (*i*, *j*) = (*ii*, *Suc ll*) ∧ *iter* ≠ *0*)
case *True*
hence ∗: *i* = *ii j* = *Suc ll iter* ≠ *0* **by** *auto*
from ∗ *eqs index* **have** *ll* < *l Suc ll* < *k Suc ll* < *n* **by** *auto*
have *B*: *B* $$ (*i*,*j*) = *quot* **unfolding** *B* **using** ∗ **by** *auto*
have ¬ ((*ll*, *j*) = (*i-end* − *iter*, *Suc l* − *iter*) ∧ *iter* ∉ {*0*, *Suc ?idiff*})
using ∗ *index eqs* **by** *auto*
hence *B'*: *B* $$ (*ll*, *j*) = *A* $$ (*ll*, *j*)
unfolding *BB*[*OF* ‹*ll* < *n*› ‹*j* < *n*›] **using** *jk* **by** *auto*
have *?C* $$ (*i*,*j*) = *quot* − *quot* ∗ *A* $$ (*ll*, *Suc ll*) **unfolding** *C B* **using** ∗ *B'* **by** *auto*
with *iblock*(*3*)[*OF lbl* ‹*lb* ≤ *ll*› ‹*ll* < *l*›] **have** *C*: *?C* $$ (*i*,*j*) = *0* **by** *simp*
{
assume *A* $$ (*ii*, *Suc ll*) ≠ *0*
with *jb*[*OF* ‹*ii* < *n*› ‹*Suc ll* < *n*› ‹*Suc ll* < *k*›] ‹*ii* ≠ *ll*›
have *ii* = *Suc ll* **by** *auto*
with *eqs* **have** *i-end* − *iter* = *Suc l* − *iter* **by** *auto*
from *block*[*OF this*] ‹*iter* ≠ *0*› ‹*iter* ≠ *Suc ?idiff*› *eqs large* **have** *False* **by** *auto*
}
hence *A* $$ (*ii*,*Suc ll*) = *0* **by** *auto*
thus *?thesis* **unfolding** *C* **unfolding** ∗ **by** *simp*
next
case *False*
hence *B*: *B* $$ (*i*,*j*) = *?Aij* **unfolding** *B* **by** *auto*
from *eqs index* **have** *lb* ≤ *ll ll* ≤ *l* **by** *auto*
note *index* = *index this ll-i*
from *evb*[*of ll k*] *index* **have** *evl*: *?ev ll* = *?ev k* **by** *auto*
from *evb*[*of i k*] *index* **have** *evi*: *?ev i* = *?ev k* **by** *auto*
from *not* **have** *not*: *i* ≠ *ii* − *1* ∨ *j* ≠ *ll* ∨ *iter* = *?idiff* **by** *auto*
from *False* **have** *not2*: *i* ≠ *ii* ∨ *j* ≠ *Suc ll* ∨ *iter* = *0* **by** *auto*
show *?thesis*
proof (*cases i* = *ii*)
case *True*
let *?diff* = *if j* = *ll then 0 else* − *quot* ∗ *A* $$ (*ll*, *j*)
have *Bli*: *B* $$ (*ll*, *i*) = *0* **using** *True Bli* **by** *simp*
have *Blj*: *B* $$ (*ll*, *j*) = *A* $$ (*ll*,*j*) **unfolding** *BB*[*OF* ‹*ll* < *n*› ‹*j* < *n*›]
using *index jk* **by** *auto*

from *True* **have** *C*: *?C* $$ (*i*,*j*) = *?Aij* + *?diff*
unfolding *C B evi* **using** *Bli Blj evl* **by** *auto*
also have *?diff* = *0*
proof (*rule ccontr*)
assume *?diff* ≠ *0*
hence *jl*: *j* ≠ *ll* **and** *Alj*: *A* $$ (*ll*,*j*) ≠ *0* **by** (*auto split*: *if-splits*)
with *jb*[*OF* ‹*ll* < *n*› ‹*j* < *n*› *jk*] **have** *j* = *Suc ll ?ev ll* = *?ev j* **by** *auto*
with *not2 True* **have** *iter* = *0* **by** *auto*
with *eqs index jk* **have** *id*: *A* $$ (*ll*, *j*) = *A* $$ (*l*, *Suc l*) **and**
j = *Suc l Suc l* < *k ll* = *l*
unfolding ‹*j* = *Suc ll*› **by** *auto*
from *iblock*[*OF lbl*] ‹*Suc l* < *k*› **have** *A* $$ (*l*, *Suc l*) ≠ *1* **by** *auto*
from *jb*[*OF* ‹*l* < *n*› ‹*j* < *n*› *jk*] *Alj this* **show** *False* **unfolding** ‹*j* = *Suc l*› ‹*ll* = *l*› **by** *auto*
qed
finally show *?thesis* **by** *simp*
next
case *False*
let *?diff* = *if j* = *ll then quot* ∗ *B* $$ (*i*, *ii*) *else 0*
from *False* **have** *C*: *?C* $$ (*i*,*j*) = *?Aij* + *?diff*
unfolding *C B* **by** *auto*
also have *?diff* = *0*
proof (*rule ccontr*)
assume *?diff* ≠ *0*
hence *j*: *j* = *ll* **and** *Bi*: *B* $$ (*i*, *ii*) ≠ *0* **by** (*auto split*: *if-splits*)
from *eqs* **have** *ii*: *i-end* − *iter* = *ii* **by** *auto*
have *Bii*: *B* $$ (*i*,*ii*) = *A* $$ (*i*, *ii*)
unfolding *BB*[*OF* ‹*i* < *n*› ‹*ii* < *n*›] **using** ‹*ii* < *k*› *iter ii False* **by** *auto*
from *Bi Bii* **have** *Ai*: *A* $$ (*i*,*ii*) ≠ *0* **by** *auto*
from *jb*[*OF* ‹*i* < *n*› ‹*ii* < *n*› ‹*ii* < *k*›] *False Ai* **have** *ii*: *ii* = *Suc i*
and *Ai*: *A* $$ (*i*,*ii*) = *1* **by** *auto*
from *not ii j* **have** *iter*: *iter* = *?idiff* **by** *auto*
with *eqs index* **have** *ii* = *i-begin* **by** *auto*
with *ii* ‹*i* ≥ *i-begin*›
show *False* **by** *auto*
qed
finally show *?thesis* **by** *simp*
qed
qed
qed
show *?thesis* **unfolding** *D C* **..**
qed
qed
qed
qed

```
      qed auto
      also have step-3-c-inner-loop quot (ll − 1) (ii − 1) diff . . . = ?R
        by (rule Suc(1), insert prems large, auto)
      finally show ?case .
    qed
  }
  note main = this[of Suc ?idiff i-end 0 l]
  from ib have suc: Suc i-end − i-begin = Suc ?idiff by simp
  have step-3-c-inner-loop (A $$ (i-end, k) / A $$ (l, k)) l i-end (Suc ?idiff) A
    = step-3-c-inner-loop quot l i-end (Suc ?idiff) (?mat 0 (Suc ?idiff))
    unfolding mend unfolding quot-def ..
  also have . . . = ?R by (rule main, auto)
  finally show ?thesis unfolding suc .
qed

private 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 (induct bs arbitrary: A)
  case (Nil A)
  note inv = Nil(4−7)
  from inv-from-upto-at-all-ev-block[OF inv(1−4) ‹k < n›]
  have inv-all uppert A unfolding one-zero-def diff-ev-def uppert-def by auto
  moreover
  have inv-at (single-non-zero l k x) A k unfolding single-non-zero-def inv-at-def
    by (intro allI impI conjI, insert Nil, auto)
  ultimately show ?case by auto
next
  case (Cons p bs A)
  obtain i-begin i-end where p: p = (i-begin, i-end) by force
  note Cons = Cons[unfolded p]
  note IH = Cons(1)
  note A = Cons(2)
  note kn = Cons(3)
  note lbl = Cons(4)
  note inv = Cons(5−8)
```

note *blocks* = *Cons*(*9*−*11*)
note *x* = *Cons*(*12*−*13*)
from *identify-blocks*[*OF lbl*] *kn* **have** *lk*: *l* < *k* **and** *ln*: *l* < *n* **and** *lb* ≤ *l* **by** *auto*
define *B* **where** *B* = *step-3-c-inner-loop* (*A* $$ (*i-end*,*k*) / *x*) *l i-end* (*Suc i-end* − *i-begin*) *A*
show *?case*
proof (*cases i-end* = *l*)
case *True*
hence *id*: *step-3-c x l k* (*p* # *bs*) *A* = *step-3-c x l k bs A* **unfolding** *p* **by** *simp*
show *?thesis* **unfolding** *id*
by (*rule IH*[*OF A kn lbl inv - blocks*(*2*−*3*) *x*], *insert blocks*(*1*), *auto simp*: *p True*)
next
case *False* **note** *il* = *this*
hence *id*: *step-3-c x l k* (*p* # *bs*) *A* = *step-3-c x l k bs B* **unfolding** *B-def p* **by** *simp*
from *blocks*[*unfolded p*] **have**
ib-block: (*i-begin*,*i-end*) ∈ *set* (*identify-blocks A k*) **and** *large*: *i-end* − *i-begin* ≤ *l* − *lb* **by** *auto*
from *identify-blocks*[*OF this*(*1*)]
have *ibe*: *i-begin* ≤ *i-end i-end* < *k* **by** *auto*
have *B*: *B* = *mat n n* (λ (*i*,*j*). *if* (*i*,*j*) = (*i-end*,*k*) *then 0 else*
if i-begin ≤ *i* ∧ *i* ≤ *i-end* ∧ *j* > *k then A* $$ (*i*,*j*) − *A* $$ (*i-end*,*k*) / *x* ∗ *A* $$ (*l* + *i* − *i-end*,*j*) *else A* $$ (*i*,*j*))
unfolding *B-def x*
by (*rule step-3-c-inner-result*[*OF inv kn A lbl ib-block il large*], *insert x*, *auto*)
have *Bn*: *B* ∈ *carrier-mat n n* **unfolding** *B* **by** *auto*
have *sdAB*: *same-diag A B* **unfolding** *B same-diag-def* **using** *ibe* **by** *auto*
have *suAB*: *same-upto k A B* **using** *A kn* **unfolding** *B same-upto-def* **by** *auto*
have *inv-ev*: *ev-block n B* **using** *same-diag-ev-block*[*OF sdAB inv*(*4*)] **.**
have *inv-jb*: *inv-upto jb B k* **using** *same-upto-inv-upto-jb*[*OF suAB inv*(*1*)] **.**
have *ib*: *identify-blocks A k* = *identify-blocks B k* **using** *identify-blocks-cong*[*OF kn sdAB suAB*] **.**
have *inv-ut*: *inv-from uppert B k* **using** *inv*(*2*) *ibe* **unfolding** *B inv-from-def uppert-def* **by** *auto*
from *x il ibe kn lk* **have** *xB*: *x* = *B* $$ (*l*,*k*) **by** (*auto simp*: *B*)
{
fix *be*
assume ∗: *be* ∉ *snd* ‘ *set bs be* ∉ {*l*, *k*} *be* < *n*
hence *B* $$ (*be*, *k*) = *0* **using** *kn blocks*(*2*)[*of be*] **unfolding** *B*
by (*cases be* = *i-end*, *auto*)
} **note** *blocksB* = *this*
have *bs*: *set bs* ⊆ *set* (*identify-blocks A k*) **using** *blocks*(*1*) **by** *auto*
have *inv-oz*: *inv-at one-zero B k* **using** *inv*(*3*) *ibe kn* **unfolding** *B inv-at-def one-zero-def* **by** *simp*
show *?thesis* **unfolding** *id*
using *IH*[*OF Bn kn*, *folded ib*, *OF lbl inv-jb inv-ut inv-oz inv-ev bs blocksB blocks*(*3*) *xB x*(*2*)]
using *same-diag-trans*[*OF sdAB*] *same-upto-trans*[*OF suAB*]

```
      by auto
  qed
qed

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 (induct k A taking: n rule: step-3-main.induct)
  case (1 k A)
  from 1(2−) have A: A ∈ carrier-mat n n and k: k > 0 and
    inv: inv-all uppert A ev-block n A inv-upto jb A k by auto
  note [simp] = step-3-main.simps[of n k A]
  show ?case
  proof (cases k ≥ n)
    case True
    thus ?thesis using inv-uptoD[OF inv(3)]
      by (intro conjI inv-allI, auto)
  next
    case False
    hence kn: k < n by simp
    obtain B where B: B = step-3-a (k − 1) k A by auto
    note IH = 1(1)[OF False B]
    from A have Bn: B ∈ carrier-mat n n unfolding B carrier-mat-def by simp
    from k have k − 1 < k by simp
    {
      fix i
      assume i < k
       with ev-blockD[OF inv(2) - ‹k < n›, of i] ‹k < n› have A $$ (i,i) = A $$
(k,k) by auto
    }
    hence inv-from-bot (λA i. one-zero A i k) A (k − 1)
      using inv-all-uppertD[OF inv(1), of k]
      unfolding inv-from-bot-def one-zero-def by auto
    from step-3-a-inv[OF A ‹k − 1 < k› ‹k < n› inv(3) inv-all-imp-inv-from[OF
inv(1)]
      this inv(2)] same-diag-ev-block[OF - inv(2)]
    have inv: inv-from uppert B k ev-block n B inv-upto jb B k
      inv-at one-zero B k and sd: same-diag A B unfolding B by auto
    note evb = ev-blockD[OF inv(2)]
    obtain all-blocks where ab: all-blocks = identify-blocks B k by simp
    obtain blocks where blocks: blocks = filter (λ block. B $$ (snd block, k) ≠ 0)
all-blocks by simp
    obtain F where F: F = (if blocks = [] then B
       else let (l-begin,l) = find-largest-block (hd blocks) (tl blocks); x = B $$ (l, k);
C = step-3-c x l k blocks B;
          D = mult-col-div-row (inverse x) k C; E = swap-cols-rows-block (Suc l)
```

k D
in E) **by** *simp*
note *IH* = *IH*[*OF ab blocks F*]
have *Fn*: *F* ∈ *carrier-mat n n* **unfolding** *F Let-def carrier-mat-def* **using** *Bn*
by (*simp split*: *prod.splits*)
have *inv*: *inv-all uppert F* ∧ *same-diag A F* ∧ *inv-upto jb F* (*Suc k*)
proof (*cases blocks* = [])
case *True*
hence *F*: *F* = *B* **unfolding** *F* **by** *simp*
have *lo*: *inv-at* (*lower-one k*) *B k*
proof
fix *i*
assume *i*: *i* < *n*
note *lower-one-def*[*simp*]
show *lower-one k B i k*
proof (*cases B* $$ (*i*,*k*) = *0*)
case *False* **note** *nz* = *this*
note *oz* = *inv-atD*[*OF inv*(*4*) *i*, *unfolded one-zero-def*]
from *nz oz* **have** *i* ≤ *k* **by** *auto*
show *?thesis*
proof (*cases i* = *k*)
case *False*
with ‹*i* ≤ *k*› **have** *i* < *k* **by** *auto*
with *nz oz* **have** *ev*: *B* $$ (*i*,*i*) = *B* $$ (*k*,*k*) **unfolding** *diff-ev-def* **by**
auto
have (*identify-block B i*, *i*) ∈ *set all-blocks* **unfolding** *ab*
proof (*rule identify-blocks-rev*[*OF* - ‹*k* < *n*›])
show *B* $$ (*i*, *Suc i*) = *0* ∧ *Suc i* < *k* ∨ *Suc i* = *k*
proof (*cases Suc i* = *k*)
case *False*
with ‹*i* < *k*› ‹*k* < *n*› **have** *Suc i* < *k Suc i* < *n* **by** *simp-all*
with *nz oz* **have** *B* $$ (*i*, *Suc i*) ≠ *1* **by** *simp*
with *inv-uptoD*[*OF inv*(*3*) ‹*i* < *n*› ‹*Suc i* < *n*› ‹*Suc i* < *k*›, *unfolded*
jb-def]
have *B* $$ (*i*, *Suc i*) = *0* **by** *simp*
thus *?thesis* **using** ‹*Suc i* < *k*› **by** *simp*
qed *simp*
qed
with *arg-cong*[*OF* ‹*blocks* = []›[*unfolded blocks*], *of set*] **have** *B* $$ (*i*,*k*)
= *0* **by** *auto*
with *nz* **show** *?thesis* **by** *auto*
qed *auto*
qed *auto*
qed
have *inv-jb*: *inv-upto jb B* (*Suc k*)
proof (*rule inv-upto-Suc*[*OF inv*(*3*)])
fix *i*
assume *i* < *n*
from *inv-atD*[*OF lo* ‹*i* < *n*›, *unfolded lower-one-def*]

show *jb B i k* **unfolding** *jb-def* **by** *auto*
qed
from *inv-from-upto-at-all-ev-block*[*OF inv*(*3*,*1*) *lo inv*(*2*) - ‹*k* < *n*›] *lower-one-diff-uppert*
have *inv-all uppert B* **by** *auto*
with *inv inv-jb sd*
show *?thesis* **unfolding** *F* **by** *simp*
next
case *False*
obtain *l-start l* **where** *l*: *find-largest-block* (*hd blocks*) (*tl blocks*) = (*l-start*, *l*) **by** *force*
obtain *x* **where** *x*: *x* = *B* $$ (*l*,*k*) **by** *simp*
obtain *C* **where** *C*: *C* = *step-3-c x l k blocks B* **by** *simp*
obtain *D* **where** *D*: *D* = *mult-col-div-row* (*inverse x*) *k C* **by** *auto*
obtain *E* **where** *E*: *E* = *swap-cols-rows-block* (*Suc l*) *k D* **by** *auto*
from *find-largest-block*[*OF False l*] **have** *lb*: (*l-start*,*l*) ∈ *set blocks*
and *llarge*: ⋀ *i-begin i-end*. (*i-begin*,*i-end*) ∈ *set blocks* ⟹ *l* − *l-start* ≥ *i-end* − *i-begin* **by** *auto*
from *lb* **have** *x0*: *x* ≠ *0* **unfolding** *blocks x* **by** *simp*
{
fix *i-start i-end*
assume (*i-start*,*i-end*) ∈ *set blocks*
hence (*i-start*,*i-end*) ∈ *set* (*identify-blocks B k*) **unfolding** *blocks ab* **by** *simp*
with *identify-blocks*[*OF this*]
have *i-end* < *k* (*i-start*,*i-end*) ∈ *set* (*identify-blocks B k*) **by** *auto*
} **note** *block-bound* = *this*
from *block-bound*[*OF lb*]
have *lk*: *l* < *k* **and** *lblock*: (*l-start*, *l*) ∈ *set* (*identify-blocks B k*) **by** *auto*
from *lk* ‹*k* < *n*› **have** *ln*: *l* < *n* **by** *simp*
from *evb*[*OF* ‹*l* < *n*› ‹*k* < *n*›]
have *Bll*: *B* $$ (*l*,*l*) = *B* $$ (*k*,*k*) **.**
from *False* **have** *F*: *F* = *E* **unfolding** *E D C x F l Let-def* **by** *simp*
from *Bn* **have** *Cn*: *C* ∈ *carrier-mat n n* **unfolding** *C carrier-mat-def* **by** *simp*
{
fix *be*
assume *nmem*: *be* ∉ *snd* ‘ *set blocks* **and** *belk*: *be* ∉ {*l*, *k*} **and** *be*: *be* < *n*
have *B* $$ (*be*, *k*) = *0*
proof (*rule ccontr*)
assume *nz*: ¬ *?thesis*
note *oz* = *inv-atD*[*OF inv*(*4*) *be*, *unfolded one-zero-def*]
from *belk oz be nz* **have** *be* < *k* **by** *auto*
obtain *bb* **where** *ib*: *identify-block B be* = *bb* **by** *force*
note *ib-inv* = *identify-block*[*OF ib*]
have *B* $$ (*be*, *Suc be*) = *0* ∧ *Suc be* < *k* ∨ *Suc be* = *k*
proof (*cases Suc be* = *k*)
case *False*
with ‹*be* < *k*› **have** *sbek*: *Suc be* < *k* **by** *auto*
from *inv-uptoD*[*OF inv*(*3*) ‹*be* < *n*› - *sbek*] *sbek kn* **have** *jb B be* (*Suc*

be) **by** *auto*
from *this*[*unfolded jb-def*] **have** *01*: *B* $$ (*be*, *Suc be*) ∈ {*0*,*1*} **by** *auto*
from *01 oz sbek nz* **have** *B* $$ (*be*, *Suc be*) = *0* **by** *auto*
with *sbek* **show** *?thesis* **by** *auto*
qed *auto*
from *identify-blocks-rev*[*OF this kn*]
nz nmem **show** *False* **unfolding** *ab blocks* **by** *force*
qed
}
note *inv3* = *step-3-c-inv*[*OF Bn* ‹*k* < *n*› *lblock inv*(*3*,*1*,*4*,*2*) - *this llarge x x0*, *of blocks*, *folded C*,
unfolded ab blocks]
from *inv3* **have** *sdC*: *same-diag B C* **and** *suC*: *same-upto k B C* **by** *auto*
note *sd* = *same-diag-trans*[*OF sd sdC*]
from *Bll sdC ln* ‹*k* < *n*›
have *Cll*: *C* $$ (*l*,*l*) = *C* $$ (*k*,*k*) **unfolding** *same-diag-def* **by** *auto*
from *same-diag-ev-block*[*OF sdC inv*(*2*)] *same-upto-inv-upto-jb*[*OF suC inv*(*3*)] *inv3*
have *inv*: *inv-all uppert C ev-block n C*
inv-upto jb C k inv-at (*single-non-zero l k x*) *C k* **by** *auto*
from *x0* **have** *inverse x* ≠ *0* **by** *simp*
from *Cn* **have** *Dn*: *D* ∈ *carrier-mat n n* **unfolding** *D carrier-mat-def* **by** *simp*
{
fix *i j*
assume *i*: *i* < *n* **and** *j*: *j* < *n*
with *Cn* **have** *dC*: *i* < *dim-row C i* < *dim-col C j* < *dim-row C j* < *dim-col C* **by** *auto*
let *?c* = *C* $$ (*i*,*j*)
let *?x* = *inverse x*
have *D* $$ (*i*,*j*) = (*if i* = *l* ∧ *j* = *k then 1 else if i* = *k* ∧ *j* ≠ *k then x* ∗ *?c else ?c*)
unfolding *D*
proof (*subst mult-col-div-index-row*[*OF dC* ‹*inverse x* ≠ *0*›], *unfold inverse-inverse-eq*)
note *at* = *inv-atD*[*OF inv*(*4*) ‹*i* < *n*›, *unfolded single-non-zero-def*]
show (*if i* = *k* ∧ *j* ≠ *i then x* ∗ *?c*
else if j = *k* ∧ *j* ≠ *i then ?x* ∗ *?c else ?c*) =
(*if i* = *l* ∧ *j* = *k then 1 else if i* = *k* ∧ *j* ≠ *k then x* ∗ *?c else ?c*) (**is** *?l* = *?r*)
proof (*cases* (*i*,*j*) = (*l*,*k*))
case *True*
with *lk* **have** *?l* = *?x* ∗ *?c* **by** *auto*
also have . . . = *1* **using** *at True* ‹*inverse x* ≠ *0*› **by** *auto*
finally show *?thesis* **using** *True* **by** *simp*
next
case *False* **note** *neq* = *this*
have *?l* = (*if i* = *k* ∧ *j* ≠ *k then x* ∗ *?c else ?c*)
proof (*cases i* = *k* ∧ *j* ≠ *k* ∨ *j* = *k* ∧ *i* ≠ *k*)

case *True*
thus *?thesis*
proof
  assume ∗: $i = k \wedge j \neq k$
  hence *l*: $?l = x * ?c$ **by** *simp*
  show *?thesis* **using** ∗ *neq* **unfolding** *l* **by** *simp*
next
  assume ∗: $j = k \wedge i \neq k$
  hence $?l = ?x * ?c$ **using** *lk* **by** *auto*
  from ∗ *neq* **have** $i \neq l$ **and** ∗∗: $\neg (i = k \wedge j \neq k)$ **by** *auto*
  from *at* ‹$i \neq l$› ∗ **have** $?c = 0$ **by** *auto*
  with ‹$?l = ?x * ?c$› ∗∗ **show** *?thesis* **by** *auto*
qed
qed *auto*
also have $\ldots = ?r$ **using** *False* **by** *auto*
finally show *?thesis* **.**
qed
qed
} **note** $D = this$
have *sD*[*simp*]: $\bigwedge i.\ i < n \Longrightarrow D\ \$\$\ (i,i) = C\ \$\$\ (i,i)$ **using** *lk* **by** (*auto simp*: *D*)
from ‹$C\ \$\$\ (l,l) = C\ \$\$\ (k,k)$› ‹$l < n$› ‹$k < n$›
have *Dll*: $D\ \$\$\ (l,l) = D\ \$\$\ (k,k)$ **by** *simp*
have *sdD*: *same-diag C D* **unfolding** *same-diag-def* **by** *simp*
note *sd* = *same-diag-trans*[*OF sd sdD*]
from *same-diag-ev-block*[*OF sdD inv*(*2*)] **have** *invD*: *ev-block n D* **.**
note *inv* = *inv-uptoD*[*OF inv*(*3*), *unfolded jb-def*] *inv-all-uppertD*[*OF inv*(*1*)]

inv-atD[*OF inv*(*4*), *unfolded single-non-zero-def*]
moreover have *inv-all uppert D*
by (*intro inv-allI*, *insert inv*(*2*) *lk*, *auto simp*: *uppert-def D*)
moreover have *suD*: *same-upto k C D*
proof
  fix *i j*
  assume *i*: $i < n$ **and** *j*: $j < k$
  with *kn* **have** *jn*: $j < n$ **by** *simp*
  show $C\ \$\$\ (i, j) = D\ \$\$\ (i, j)$
    unfolding *D*[*OF i jn*] **using** *j k*
    inv(*1*)[*OF i jn j*] *i j* **by** *auto*
qed
from *same-upto-inv-upto-jb*[*OF suD* ‹*inv-upto jb C k*›]
have *inv-upto jb D k* **.**
moreover
let *?single-one* = *single-one l k*
have *inv-at ?single-one D k*
by (*intro inv-atI*, *insert inv*(*3*) *D*[*OF -* ‹$k < n$›] *ln*, *auto simp*: *single-one-def*)
ultimately
have *inv*: *inv-all uppert D ev-block n D*
inv-upto jb D k inv-at ?single-one D k **using** *invD* **by** *blast+*

```
note inv = inv-uptoD[OF inv(3), unfolded jb-def]
  inv-all-uppertD[OF inv(1)]
  inv-atD[OF inv(4), unfolded single-one-def]
  ev-blockD[OF inv(2)]
from suC suD have suD: same-upto k B D unfolding same-upto-def by auto
let ?I = λ j. if j = Suc l then k else if Suc l < j ∧ j ≤ k then j − 1 else j
let ?I' = λ j. if j = Suc l then k else j − 1
{
  fix i j
  assume i: i < n and j: j < n
  with Dn lk ‹k < n›
  have dims: i < dim-row D i < dim-col D j < dim-row D j < dim-col D
    Suc l ≤ k k < dim-row D k < dim-col D by auto
  have E $$ (i,j) = D $$ (?I i, ?I j)
    unfolding E by (rule subst swap-cols-rows-block-index[OF dims])
} note E = this
{
  fix i
  assume i: i < n
  from ‹l < k› have l ≤ Suc l Suc l ≤ k by auto
  have E $$ (i,i) = D $$ (i,i) unfolding E[OF i i]
    by (rule inv(4), insert i ‹k < n›, auto)
} note Ed = this
from Ed have ed: same-diag D E unfolding same-diag-def by auto
note sd = same-diag-trans[OF sd ed]
have ev-block n E using same-diag-ev-block[OF ed ‹ev-block n D›] by auto
moreover have Eut: inv-all uppert E
proof (intro inv-allI, unfold uppert-def, intro impI)
  fix i j
  assume i: i < n and j: j < n and ji: j < i
  have ?I i < n using i ‹k < n› by auto
  show E $$ (i,j) = 0
  proof (cases ?I j < ?I i)
    case True
    from inv(2)[OF this ‹?I i < n›] show ?thesis unfolding E[OF i j] .
  next
    case False
    have ?I i ≠ ?I j using ji lk by (auto split: if-splits)
    with False have ij: ?I i < ?I j by simp
    from ij ji have jl: j = Suc l using lk by (auto split: if-splits)
    with ji ij have il: i > Suc l i ≤ k by (auto split: if-splits)
    from jl il have Eij: E $$ (i,j) = D $$ (i−1,k) unfolding E[OF i j] by
simp
    have i − 1 < n i − 1 ∉ {k, l} using i il by auto
    with inv(3)[of i−1] have D: D $$ (i−1,k) = 0 by auto
    show ?thesis unfolding Eij D by simp
  qed
qed
moreover
```

```
  from same-diag-trans[OF ‹same-diag B C› ‹same-diag C D›] have same-diag
B D .
   from identify-blocks-cong[OF ‹k < n› this suD]
   have idb: identify-blocks B k = identify-blocks D k .
   have inv-upto jb E (Suc k)
   proof (intro inv-uptoI)
    fix i j
    assume i: i < n and j: j < n and j < Suc k
    hence jk: j ≤ k by simp
    show jb E i j
    proof (cases E $$ (i,j) = 0 ∨ j = i)
     case True
     thus ?thesis unfolding jb-def by auto
    next
     case False note enz = this
     from inv(4)[OF i j] have same-ev: D $$ (i,i) = D $$ (j,j) .
     note inv2 = inv-all-uppertD[OF Eut - i, of j]
     from False inv2 have ¬ j < i by auto
     with False have ji: j > i by auto
     have E $$ (i,j) ∈ {0,1} ∧ (j ≠ Suc i ⟶ E $$ (i,j) = 0)
     proof (cases j ≤ l)
      case True note jl = this
      with ji lk have il: i ≤ l and jk: j < k by auto
     have id: E $$ (i,j) = D $$ (i,j) unfolding E[OF i j] using jl il by simp
      from inv(1)[OF i j jk] ji
      show ?thesis unfolding id by auto
     next
      case False note jl = this
      show ?thesis
      proof (cases j = Suc l)
       case True note jl = this
       with ji lk have il: i ≤ l i ≠ k by auto
        have id: E $$ (i,j) = D $$ (i,k) unfolding E[OF i j] using jl il by
auto
       from inv(3)[OF i] jl il
       show ?thesis unfolding id by (cases i = l, auto)
      next
       case False
       with jl jk kn have jl: j > Suc l and jk: j − 1 < k and jn: j − 1 < n
by auto
       with jk have id: ?I j = j − 1 by auto
       note jb = inv(1)[OF - jn jk]
       show ?thesis
       proof (cases i < Suc l)
        case True note il = this
         with id have id: E $$ (i,j) = D $$ (i,j − 1) unfolding E[OF i j]
by auto
        show ?thesis
        proof (cases i = j − 2)
```

```
                case False
                thus ?thesis unfolding id using jb[OF i] il jl by auto
              next
                case True
                with il jl have *: j = Suc (Suc l) i = l by auto
                with id have id: E $$ (i,j) = D $$ (l,Suc l) by auto
                from * jl jk have neq: Suc l ≠ k by auto
                 from lblock[unfolded idb] have (l-start, l) ∈ set (identify-blocks D
k) .
                from this[unfolded identify-blocks-iff[OF kn]] neq
                have D $$ (l, Suc l) ≠ 1 by auto
                with jb[OF i] il jl ji * have D $$ (l, Suc l) = 0 by auto
                thus ?thesis unfolding id by simp
              qed
            next
              case False note il = this
              show ?thesis
              proof (cases i = Suc l)
                case True
               with id have id: E $$ (i,j) = D $$ (k,j − 1) unfolding E[OF i j]
by auto
                from inv(2)[OF jk kn] show ?thesis unfolding id by simp
              next
                case False
                 with il jl ji jk kn have il: i > Suc l and ik: i < k and i-n: i − 1
< n by auto
               with id have id: E $$ (i,j) = D $$ (i − 1, j − 1) unfolding E[OF
i j] by auto
                show ?thesis unfolding id using jb[OF i-n] il jl ji by auto
              qed
            qed
          qed
        qed
        thus jb E i j unfolding jb-def Ed[OF i] Ed[OF j] same-ev by auto
      qed
    qed
    ultimately show ?thesis using sd unfolding F by simp
  qed
  hence inv: inv-all uppert F ev-block n F inv-upto jb F (Suc k)
     and sd: same-diag A F using same-diag-ev-block[OF - ‹ev-block n A›] by
auto
  have 0 < Suc k by simp
  note IH = IH[OF Fn this inv(1−3)]
  have id: step-3-main n k A = step-3-main n (Suc k) F using kn
    by (simp add: F Let-def blocks ab B)
  from same-diag-trans[OF sd] IH
  show ?thesis unfolding id by auto
 qed
qed
```

lemma *step-1-2-inv*:
  assumes *A*: $A \in$ *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 −
  from *A* **have** *d*: *dim-row A* $= n$ **by** *simp*
  let *?B* $=$ *step-2* (*step-1 A*)
  from *upper-triangularD*[*OF upper-t*] **have** *inv*: *inv-all uppert A*
    unfolding *inv-all-def uppert-def* **using** *A* **by** *auto*
  from *upper-t* **have** *inv2*: *inv-part diff-ev A 0 0*
    unfolding *inv-part-def diff-ev-def* **by** *auto*
  have *inv3*: *ev-blocks-part 0* (*step-1 A*)
    by (*rule ev-blocks-partI*, *auto*)
  have *A1*: *step-1 A* $\in$ *carrier-mat n n* **using** *A* **unfolding** *carrier-mat-def* **by** *auto*
  from *A1* **have** *d1*: *dim-row* (*step-1 A*) $= n$ **unfolding** *carrier-mat-def* **by** *simp*
  have *B*: *?B* $\in$ *carrier-mat n n* **using** *A* **unfolding** *carrier-mat-def* **by** *auto*
  from *B* **have** *d2*: *dim-row ?B* $= n$ **unfolding** *carrier-mat-def* **by** *simp*
  have *inv-all uppert* (*step-1 A*) $\wedge$ *inv-all diff-ev* (*step-1 A*) **unfolding** *step-1-def d*
    by (*rule step-1-main-inv*[*OF - A inv inv2*], *simp*)
  hence *inv-all uppert* (*step-1 A*) **and** *inv-all diff-ev* (*step-1 A*) **by** *auto*
  from *step-2-main-inv*[*OF A1 this inv3*]
  show *inv-all uppert B inv-all diff-ev B ev-blocks B*
    unfolding *step-2-def d d1 Bid* **by** *auto*
qed

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*. $i <$ *dim-row A* $\longrightarrow j <$ *dim-row A* $\longrightarrow$ *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*
  using *assms* **by** (*induct ev k A rule*: *lookup-other-ev.induct*, *auto split*: *if-splits*)
  (*insert less-antisym*, *blast*)

private lemma *lookup-other-ev-Some*: **assumes** *lookup-other-ev ev k A* $=$ *Some i*
  shows $i < k \wedge$ *A* $$ (*i*,*i*) $\neq$ *ev* $\wedge$ ($\forall$ *j*. $i < j \wedge j < k \longrightarrow$ *A* $$ (*j*,*j*) $=$ *ev*)
  using *assms* **by** (*induct ev k A rule*: *lookup-other-ev.induct*, *auto split*: *if-splits*)
  (*insert less-SucE*, *blast*)

lemma *partition-jb*: **assumes** *A*: (*A* :: *′a mat*) $\in$ *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*. $B \in$ *set bs* $\Longrightarrow$ *inv-all′ uppert B* $\wedge$ *ev-block* (*dim-col B*) *B* $\wedge$ *dim-row B* $=$ *dim-col B*

proof −
have *diag*: *diag-block-mat* [*A*] = *A* **using** *A* **by** *auto*
{
fix *cs*
assume ∗: $\bigwedge$ *C*. *C* ∈ *set cs* ⟹ *dim-row C* = *dim-col C* ∧ *inv-all′ uppert C* ∧ *ev-block* (*dim-col C*) *C partition-ev-blocks A cs* = *bs*
from *inv* **have** *inv*: *inv-all′ uppert A inv-all′ diff-ev A ev-blocks-part n A*
unfolding *inv-all-def inv-all′-def ev-blocks-def* **using** *A* **by** *auto*
hence *diag-block-mat* (*A* # *cs*) = *diag-block-mat bs* ∧ (∀ *B* ∈ *set bs*. *inv-all′ uppert B* ∧ *ev-block* (*dim-col B*) *B* ∧ *dim-row B* = *dim-col B*)
using *A* ∗
proof (*induct n arbitrary*: *A cs bs rule*: *less-induct*)
case (*less n A cs bs*)
from *less*(*5*) **have** *A*: *A* ∈ *carrier-mat n n* **by** *auto*
hence *dim*: *dim-row A* = *n dim-col A* = *n* **by** *auto*
let *?dim* = *sum-list* (*map dim-col cs*)
let *?C* = *diag-block-mat cs*
define *C* **where** *C* = *?C*
from *less*(*6*) **have** *cs*: $\bigwedge$ *C*. *C* ∈ *set cs* ⟹ *inv-all′ uppert C* ∧ *ev-block* (*dim-col C*) *C* ∧ *dim-row C* = *dim-col C* **by** *auto*
hence *dimcs*[*simp*]: *sum-list* (*map dim-row cs*) = *?dim* **by** (*induct cs*, *auto*)
from *dim-diag-block-mat*[*of cs*, *unfolded dimcs*] **obtain** *nc* **where** *C*: *?C* ∈ *carrier-mat nc nc* **unfolding** *carrier-mat-def* **by** *auto*
hence *dimC*: *dim-row C* = *nc dim-col C* = *nc* **unfolding** *C-def* **by** *auto*
note *bs* = *less*(*7*)[*unfolded partition-ev-blocks.simps*[*of A cs*] *Let-def dim*, *symmetric*]
show *?case*
proof (*cases n* = *0*)
case *True*
hence *bs*: *bs* = *cs* **unfolding** *bs* **by** *simp*
thus *?thesis* **using** *cs A* **by** (*auto simp*: *Let-def True*)
next
case *False*
let *?n1* = *n* − *1*
let *?look* = *lookup-other-ev* (*A* $$ (*?n1*, *?n1*)) *?n1 A*
show *?thesis*
proof (*cases ?look*)
case *None*
from *False None* **have** *bs*: *bs* = *A* # *cs* **unfolding** *bs* **by** *auto*
have *ut*: *inv-all′ uppert A* **using** *less*(*2*) **by** *auto*
from *lookup-other-ev-None*[*OF None*] **have** $\bigwedge$ *i*. *i* < *n* ⟹ *A* $$ (*i*,*i*) = *A* $$ (*?n1*, *?n1*)
by (*case-tac i* = *?n1*, *auto*)
hence *evb*: *ev-block n A* **unfolding** *ev-block-def dim* **by** *metis*
from *cs A ut evb* **show** *?thesis* **unfolding** *bs* **by** *auto*
next
case (*Some i*)
let *?si* = *Suc i*
from *lookup-other-ev-Some*[*OF Some*] **have** *i*: *i* < *?n1* **and** *neq*: *A* $$

$(i,i) \neq A$ $$ $(?n1, ?n1)$
and *between*: $\bigwedge j.\ i < j \Longrightarrow j < ?n1 \Longrightarrow A$ $$ $(j,j) = A$ $$ $(?n1, ?n1)$
by *auto*
define *m* **where** $m = n - ?si$
from *i False* **have** *si*: $?si < n$ **by** *auto*
from *False i* **have** *nsi*: $n = ?si + m$ **unfolding** *m-def* **by** *auto*
obtain *UL UR LL LR* **where** *split*: *split-block A ?si ?si* = (*UL*, *UR*, *LL*, *LR*) **by** (*rule prod-cases4*)
from *split-block*[*OF split dim*[*unfolded nsi*]]
have *carr*: *UL* ∈ *carrier-mat ?si ?si UR* ∈ *carrier-mat ?si m LL* ∈ *carrier-mat m ?si LR* ∈ *carrier-mat m m*
and *Ablock*: *A* = *four-block-mat UL UR LL LR* **by** *auto*
hence *dimLR*: *dim-row LR* = *m dim-col LR* = *m* **and** *dimUL*: *dim-col UL* = *?si dim-row UL* = *?si* **by** *auto*
from *less*(*3*)[*unfolded inv-all′-def diff-ev-def*] *dim*
have *diff*: $\bigwedge i\ j.\ i < n \Longrightarrow j < n \Longrightarrow i < j \Longrightarrow A$ $$ $(i, j) \neq A$ $$ (j, j) $\Longrightarrow A$ $$ $(i, j) = 0$ **by** *auto*
from *less*(*2*)[*unfolded inv-all′-def uppert-def*] *dim*
have *ut*: $\bigwedge i\ j.\ i < n \Longrightarrow j < n \Longrightarrow j < i \Longrightarrow A$ $$ $(i, j) = 0$ **by** *auto*
let *?UR* = 0_m *?si m*
have *UR*: *UR* = *?UR*
proof (*rule eq-matI*)
fix *ia j*
assume *ij*: *ia* < *dim-row* (0_m (*Suc i*) *m*) *j* < *dim-col* (0_m (*Suc i*) *m*)
let *?j* = *?si* + *j*
have *UL* $$ (*ia*,*ia*) = *A* $$ (*ia*,*ia*) **using** *ij carr* **unfolding** *Ablock* **by** *auto*
also have ... ≠ *A* $$ (*?j*, *?j*)
proof
assume *eq*: *A* $$ (*ia*,*ia*) = *A* $$ (*?j*, *?j*)
from *ij* **have** *rel*: *ia* ≤ *i i* ≤ *?j ?j* < *n* **using** *nsi i* **by** *auto*
from *ev-blocks-part-leD*[*OF less*(*4*) *this eq*[*symmetric*]] *eq*
have *eq*: *A* $$ (*i*,*i*) = *A* $$ (*?j*,*?j*) **by** *auto*
also have ... = *A* $$ (*?n1*, *?n1*) **using** *between*[*of ?j*] *rel* **by** (*cases ?j* = *?n1*, *auto*)
finally show *False* **using** *neq* **by** *auto*
qed
also have *A* $$ (*?si* + *j*, *?si* + *j*) = *LR* $$ (*j*,*j*) **using** *ij carr* **unfolding** *Ablock* **by** *auto*
finally show *UR* $$ (*ia*, *j*) = 0_m (*Suc i*) *m* $$ (*ia*, *j*)
using *diff*[*of ia ?si* + *j*, *unfolded Ablock*] *ij nsi carr* **by** *auto*
qed (*insert carr*, *auto*)
let *?LL* = 0_m *m ?si*
have *LL*: *LL* = *?LL*
proof (*rule eq-matI*)
fix *ia j*
show *ia* < *dim-row* (0_m *m* (*Suc i*)) ⟹ *j* < *dim-col* (0_m *m* (*Suc i*)) ⟹ *LL* $$ (*ia*, *j*) = 0_m *m* (*Suc i*) $$ (*ia*, *j*)
using *ut*[*of ?si* + *ia j*, *unfolded Ablock*] *nsi carr* **by** *auto*

qed (*insert carr*, *auto*)
have *utUL*: *inv-all′ uppert UL***unfolding** *inv-all′-def uppert-def*
proof (*intro allI impI*)
fix *i j*
show $i < \textit{dim-row } UL \Longrightarrow j < \textit{dim-row } UL \Longrightarrow j < i \Longrightarrow UL \ \$\$ \ (i, j) = 0$
using *ut*[*of i j*, *unfolded Ablock*] **using** *nsi carr* **by** *auto*
qed
have *diffUL*: *inv-all′ diff-ev UL***unfolding** *inv-all′-def diff-ev-def*
proof (*intro allI impI*)
fix *i j*
show $i < \textit{dim-row } UL \Longrightarrow j < \textit{dim-row } UL \Longrightarrow i < j \Longrightarrow UL \ \$\$ \ (i, i) \neq UL \ \$\$ \ (j, j) \Longrightarrow UL \ \$\$ \ (i, j) = 0$
using *diff*[*of i j*, *unfolded Ablock*] **using** *nsi carr* **by** *auto*
qed
have *evbUL*: *ev-blocks-part ?si UL***unfolding** *ev-blocks-part-def*
proof (*intro allI impI*)
fix *ia j k*
show $ia < j \Longrightarrow j < k \Longrightarrow k < \textit{Suc } i \Longrightarrow UL \ \$\$ \ (k, k) = UL \ \$\$ \ (ia, ia) \Longrightarrow UL \ \$\$ \ (j, j) = UL \ \$\$ \ (ia, ia)$
using *less*(*4*)[*unfolded Ablock ev-blocks-part-def*, *rule-format*, *of ia j k*] **using** *nsi carr* **by** *auto*
qed
have *utLR*: *inv-all′ uppert LR* **unfolding** *inv-all′-def uppert-def*
proof (*intro allI impI*)
fix *i j*
show $i < \textit{dim-row } LR \Longrightarrow j < \textit{dim-row } LR \Longrightarrow j < i \Longrightarrow LR \ \$\$ \ (i, j) = 0$
using *ut*[*of ?si + i ?si + j*, *unfolded Ablock*] *nsi carr* **by** *auto*
qed
have *evbLR*: *ev-block* (*dim-row LR*) *LR* **unfolding** *ev-block-def*
proof (*intro allI impI*)
fix *i j*
show $i < \textit{dim-row } LR \Longrightarrow j < \textit{dim-row } LR \Longrightarrow LR \ \$\$ \ (i, i) = LR \ \$\$ \ (j, j)$
using *between*[*of ?si + i*] *between*[*of ?si + j*] *carr nsi*
unfolding *Ablock* **by** *auto* (*metis One-nat-def Suc-lessI diff-Suc-1*)
qed
from *False Some split* **have** *bs*: *partition-ev-blocks UL* (*LR # cs*) = *bs* **unfolding** *bs* **by** *auto*
have *IH*: *diag-block-mat* (*UL # LR # cs*) = *diag-block-mat bs* ∧ (∀ *B*∈*set bs*. *inv-all′ uppert B* ∧ *ev-block* (*dim-col B*) *B* ∧ *dim-row B* = *dim-col B*)
by (*rule less*(*1*)[*OF si utUL diffUL evbUL carr*(*1*) - *bs*], *insert dimLR evbLR utLR cs*, *auto*)
have *diag-block-mat* (*A # cs*) = *diag-block-mat* (*UL # LR # cs*)
unfolding *diag-block-mat.simps dim C-def*[*symmetric*] *dimC dimLR dimUL Let-def*
index-mat-four-block(*2−3*) *Ablock UR LL*
using *assoc-four-block-mat*[*of UL LR C*] *dimC carr* **by** *simp*

with *IH* **show** *?thesis* **by** *auto*
qed
qed
qed
}
from *this*[*of Nil*, *OF - part*] **show** *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*
unfolding *diag* **by** *fastforce+*
qed

lemma *uppert-to-jb*: **assumes** *ut*: *inv-all uppert A* **and** *A* ∈ *carrier-mat n n*
shows *inv-upto jb A 1*
proof (*rule inv-uptoI*)
fix *i j*
assume *i* < *n j* < *n* **and** *j* < *1*
hence *j*: *j* = *0* **and** *jn*: *0* < *n* **by** *auto*
show *jb A i j* **unfolding** *jb-def j* **using** *inv-all-uppertD*[*OF ut - ‹i* < *n›*, *of 0*]
by *auto*
qed

lemma *jnf-vector*: **assumes** *A*: *A* ∈ *carrier-mat n n*
and *jb*: $\bigwedge$ *i j*. *i* < *n* ⟹ *j* < *n* ⟹ *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 −
from *A* **have** *dim-row A* = *n* **by** *simp*
hence *id*: *jnf-vector A* = *jnf-vector-main n A* **unfolding** *jnf-vector-def* **by** *auto*
let *?map* = *map* (λ(*n*, *a*). *jordan-block n* (*a* :: *′a*))
let *?B* = λ *k*. *diag-block-mat* (*?map* (*jnf-vector-main k A*))
{
fix *k*
assume *k* ≤ *n*
hence (∀ *i j*. *i* < *k* ⟶ *j* < *k* ⟶ *?B k* $$ (*i*,*j*) = *A* $$ (*i*,*j*))
∧ *diag-block-mat* (*?map* (*jnf-vector-main k A*)) ∈ *carrier-mat k k*
∧ *0* ∉ *fst ‘ set* (*jnf-vector-main k A*)
proof (*induct k rule*: *less-induct*)
case (*less sk*)
show *?case*
proof (*cases sk*)
case (*Suc k*)
obtain *b* **where** *ib*: *identify-block A k* = *b* **by** *force*
let *?ev* = *A* $$ (*b*,*b*)
from *ib* **have** *id*: *jnf-vector-main sk A* = *jnf-vector-main b A* @ [(*Suc k* − *b*, *?ev*)] **unfolding** *Suc* **by** *simp*
let *?c* = *Suc k* − *b*
define *B* **where** *B* = *?B b*
define *C* **where** *C* = *jordan-block ?c ?ev*
have *C*: *C* ∈ *carrier-mat ?c ?c* **unfolding** *C-def* **by** *auto*

let *?FB* = λ *Bb Cc. four-block-mat Bb* (0_m (*dim-row Bb*) (*dim-col Cc*)) (0_m (*dim-row Cc*) (*dim-col Bb*)) *Cc*
from *identify-block-le'*[*OF ib*] **have** *bk*: $b \le k$ **.**
with *Suc less*(*2*) **have** $b < sk$ $b \le n$ **by** *auto*
note *IH* = *less*(*1*)[*OF this*, *folded B-def*]
have *B*: $B \in$ *carrier-mat b b* **using** *IH* **by** *simp*
from *bk Suc* **have** *sk*: *sk* = *b* + *?c* **by** *auto*
show *?thesis* **unfolding** *id map-append list.simps diag-block-mat-last split B-def*[*symmetric*] *C-def*[*symmetric*] *Let-def*
proof (*intro allI conjI impI*)
show *?FB B C* $\in$ *carrier-mat sk sk* **unfolding** *sk* **using** *four-block-carrier-mat*[*OF B C*] **.**
fix *i j*
assume *i*: *i* < *sk* **and** *j*: *j* < *sk*
with *jb* ‹*sk* $\le$ *n*›
have *jb*: *jb A i j* **by** *auto*
have *ut*: *uppert A i j* **by** (*rule jb-imp-uppert*[*OF jb*])
have *de*: *diff-ev A i j* **by** (*rule jb-imp-diff-ev*[*OF jb*])
from *B C* **have** *dim*: *dim-row B* = *b* *dim-col B* = *b* *dim-col C* = *?c* *dim-row C* = *?c* **by** *auto*
from *sk B C i j* **have** *i* < *dim-row B* + *dim-row C* *j* < *dim-col B* + *dim-col C* **by** *auto*
note *id* = *index-mat-four-block*(*1*)[*OF this*, *unfolded dim*]
have *id*: *?FB B C* $$ (*i*,*j*) =
(*if i* < *b then if j* < *b then B* $$ (*i*, *j*) *else 0*
else if j < *b then 0 else C* $$ (*i* − *b*, *j* − *b*))
unfolding *id dim* **using** *i j sk* **by** *auto*
show *?FB B C* $$ (*i*,*j*) = *A* $$ (*i*,*j*)
proof (*cases i* < *b* $\wedge$ *j* < *b*)
case *True*
hence *?FB B C* $$ (*i*,*j*) = *B* $$ (*i*,*j*) **unfolding** *id* **by** *auto*
with *IH True* **show** *?thesis* **by** *auto*
next
case *False* **note** *not-ul* = *this*
note *ib* = *identify-block*[*OF ib*]
show *?thesis*
proof (*cases* ¬ *i* < *b* $\wedge$ *j* < *b* $\vee$ *i* < *b* $\wedge$ ¬ *j* < *b*)
case *True*
hence *id*: *?FB B C* $$ (*i*,*j*) = *0* **unfolding** *id* **by** *auto*
show *?thesis*
proof (*cases j* < *i*)
case *True*
with *ut* **show** *?thesis* **unfolding** *id uppert-def* **by** *auto*
next
case *False*
with *True* **have** $*$: $j \ge b$ $i < b$ $j > i$ **by** *auto*
have *A* $$ (*i*,*j*) = *0*
proof (*rule ccontr*)
assume *A* $$ (*i*,*j*) $\neq$ *0*

with *jb*[*unfolded jb-def*] $*$
have *ji*: $j = b$ $i = b - 1$ $b > 0$ **and** *no-border*: A \$\$ $(i, i) = A$ \$\$ (j, j) A \$\$ $(i,j) = 1$ **by** *auto*
from *no-border*[*unfolded ji*] *ib*(*2*) ‹$b > 0$› **show** *False* **by** *auto*

qed
thus *?thesis* **unfolding** *id* **by** *simp*
qed
next
case *False*
with *not-ul* **have** $*$: $\neg\ i < b$ $\neg\ j < b$ **by** *auto*
hence *id*: *?FB B C* \$\$ $(i,j) = C$ \$\$ $(i - b, j - b)$ **unfolding** *id* **by** *auto*
from $*$ *i j* **have** *ijc*: $i - b <$ *?c* $j - b <$ *?c* **unfolding** *sk* **by** *auto*
have *id*: *?FB B C* \$\$ $(i,j) =$ (*if* $i - b = j - b$ *then ?ev else if Suc* $(i - b) = j - b$ *then 1 else 0*)
unfolding *id* **unfolding** *C-def jordan-block-index*(*1*)[*OF ijc*] **..**
show *?thesis*
proof (*cases* $i - b = j - b$)
case *True*
hence *id*: *?FB B C* \$\$ $(i,j) =$ *?ev* **unfolding** *id* **by** *simp*
from *True* $*$ **have** *ij*: $j = i$ **by** *auto*
have *i-n*: $i < n$ **using** *i* ‹$sk \leq n$› **by** *auto*
have *b-n*: $b < n$ **using** ‹$b < sk$› ‹$sk \leq n$› **by** *auto*
from *ib*(*3*)[*of i*] *True* $*$ *i j Suc ev-blockD*[*OF evb i-n b-n*] **have** A \$\$ $(i,j) =$ *?ev* **unfolding** *ij* **by** *auto*
with *id* **show** *?thesis* **by** *simp*
next
case *False* **note** *neq* = *this*
show *?thesis*
proof (*cases* $j - b =$ *Suc* $(i - b)$)
case *True*
hence *id*: *?FB B C* \$\$ $(i,j) = 1$ **unfolding** *id* **by** *simp*
from *True* $*$ **have** *ij*: $j =$ *Suc i* **by** *auto*
from *ib*(*3*)[*of i*] *True* $*$ *i j Suc* **have** A \$\$ $(i,j) = 1$ **unfolding** *ij* **by** *auto*
with *id* **show** *?thesis* **by** *simp*
next
case *False*
with *neq* **have** *id*: *?FB B C* \$\$ $(i,j) = 0$ **unfolding** *id* **by** *simp*
from $*$ *neq False* **have** $i \neq j$ *Suc* $i \neq j$ **by** *auto*
with *jb*[*unfolded jb-def*] **have** A \$\$ $(i,j) = 0$ **by** *auto*
with *id* **show** *?thesis* **by** *simp*
qed
qed
qed
qed
qed (*insert bk IH*, *auto*)
qed *auto*

qed
}
from *this*[*OF le-refl*] *A*
show *jordan-matrix* (*jnf-vector A*) = *A* *0* ∉ *fst* ‘ *set* (*jnf-vector A*)
unfolding *id jordan-matrix-def* **by** *auto*
qed

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 −
from *A* **have** *d*: *dim-row A* = *n* **by** *simp*
let *?B* = *step-2* (*step-1 A*)
let *?J* = *triangular-to-jnf-vector A*
have *A1*: *step-1 A* ∈ *carrier-mat n n* **using** *A* **unfolding** *carrier-mat-def* **by** *simp*
from *similar-mat-trans*[*OF step-2-similar step-1-similar*, *OF A1 A*]
have *sim*: *similar-mat ?B A* **.**
have *A1*: *step-1 A* ∈ *carrier-mat n n* **using** *A* **unfolding** *carrier-mat-def* **by** *auto*
from *A1* **have** *d1*: *dim-row* (*step-1 A*) = *n* **unfolding** *carrier-mat-def* **by** *simp*
have *B*: *?B* ∈ *carrier-mat n n* **using** *A* **unfolding** *carrier-mat-def* **by** *auto*
from *B* **have** *d2*: *dim-row ?B* = *n* **unfolding** *carrier-mat-def* **by** *simp*
define *Cs* **where** *Cs* = *partition-ev-blocks ?B* []
from *step-1-2-inv*[*OF A upper-t refl*]
have *inv*: *inv-all n uppert ?B inv-all n diff-ev ?B ev-blocks n ?B* **by** *auto*
from *partition-jb*[*OF B inv*, *of Cs*] **have** *BC*: *?B* = *diag-block-mat Cs*
and *Cs*: ⋀ *C*. *C* ∈ *set Cs* ⟹ *inv-all′ uppert C* ∧ *ev-block* (*dim-col C*) *C* ∧ *dim-row C* = *dim-col C* **unfolding** *Cs-def* **by** *auto*
define *D* **where** *D* = *map step-3 Cs*
let *?D* = *diag-block-mat D*
let *?CD* = *map* (λ *C*. (*C*, (*jnf-vector o step-3*) *C*)) *Cs*
{
fix *C D*
assume *mem*: (*C*,*D*) ∈ *set ?CD*
hence *DC*: *D* = *jnf-vector* (*step-3 C*) **and** *C*: *C* ∈ *set Cs* **by** *auto*
let *?D* = *step-3 C*
define *n* **where** *n* = *dim-col C*
from *Cs*[*OF C*] **have** *C*: *inv-all n uppert C ev-block n C C* ∈ *carrier-mat n n*
unfolding *inv-all′-def inv-all-def n-def carrier-mat-def* **by** *auto*
from *step-3-similar*[*OF C*(*3*)] **have** *sim*: *similar-mat C ?D* **by** (*rule similar-mat-sym*)
from *similar-matD*[*OF sim*] *C* **have** *D*: *?D* ∈ *carrier-mat n n* **unfolding** *carrier-mat-def* **by** *auto*
from *C*(*3*) **have** *dimC*: *dim-row C* = *n* **by** *auto*

from *step-3-main-inv*[*OF C*(*3*) - *C*(*1*,*2*) *uppert-to-jb*[*OF C*(*1*) *C*(*3*)]]
have *inv-all n jb* (*step-3 C*) **and** *sd*: *same-diag n C* (*step-3 C*) **unfolding** *step-3-def dimC* **by** *auto*
hence *jbD*: $\bigwedge$ *i j*. $i < n \Longrightarrow j < n \Longrightarrow$ *jb ?D i j* **unfolding** *inv-all-def DC* **by** *auto*
from *same-diag-ev-block*[*OF sd C*(*2*)] **have** *ev-block n* (*step-3 C*) **by** *auto*
from *jnf-vector*[*OF D jbD this*] **have** *jordan-matrix D* = *?D 0* $\notin$ *fst ' set D* **unfolding** *DC* **by** *auto*
with *sim* **have** *jordan-nf C D* **unfolding** *jordan-nf-def* **by** *simp*
} **note** *jnf-blocks* = *this*
have *id*: *map fst ?CD* = *Cs* **by** (*induct Cs*, *auto*)
have *id2*: *map snd ?CD* = *map* (*jnf-vector o step-3*) *Cs* **by** (*induct Cs*, *auto*)
have *J*: *?J* = *concat* (*map* (*jnf-vector* ∘ *step-3*) *Cs*) **unfolding** *triangular-to-jnf-vector-def Let-def Cs-def* **..**
from *jordan-nf-diag-block-mat*[*of ?CD*, *OF jnf-blocks*, *unfolded id id2*]
have *jnf*: *jordan-nf* (*diag-block-mat Cs*) *?J* **unfolding** *J* **.**
hence *similar-mat* (*diag-block-mat Cs*) (*jordan-matrix ?J*) **unfolding** *jordan-nf-def* **by** *auto*
from *similar-mat-sym*[*OF similar-mat-trans*[*OF similar-mat-sym*[*OF this*] *sim*[*unfolded BC*]]] *jnf*
show *?thesis* **unfolding** *jordan-nf-def* **by** *auto*
qed

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* = ($\prod$ *a* ← *es*. [:− *a*, *1*:])
shows *jordan-nf A* (*jordan-nf-via-factored-charpoly A es*)
proof −

let *?B* = *schur-upper-triangular A es*
let *?J* = *jordan-nf-via-factored-charpoly A es*
from *schur-upper-triangular*[*OF A linear*]
have *B*: *?B* ∈ *carrier-mat n n upper-triangular ?B* **and** *AB*: *similar-mat A ?B* **by** *auto*
from *triangular-to-jnf-vector*[*OF B*] **have** *jordan-nf ?B ?J*
unfolding *jordan-nf-via-factored-charpoly-def* **.**
with *similar-mat-trans*[*OF AB*] **show** *jordan-nf A ?J* **unfolding** *jordan-nf-def* **by** *blast*
qed

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*
using *jordan-nf-via-factored-charpoly*[*OF A linear*] **by** *blast*

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
assume *?r*
thus *?l* **using** *jordan-nf-exists*[*OF A*] **by** *auto*
next
assume *?l*
then obtain *n-as* **where** *jnf*: *jordan-nf A n-as* **by** *auto*
show *?r* **unfolding** *jordan-nf-char-poly*[*OF jnf*] *expand-powers*[*of* λ *a*. [: −*a*, *1*:] *n-as*] **by** *blast*
qed

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
show *similar-mat A B* ⟹ *jordan-nf A* = *jordan-nf B*
by (*intro ext*, *auto simp*: *jordan-nf-def*, *insert similar-mat-trans similar-mat-sym*, *blast*+)
assume *id*: *jordan-nf A* = *jordan-nf B*
from *char-poly-factorized*[*OF A*] **obtain** *as* **where** *char-poly A* = (∏ *a*←*as*. [:− *a*, *1*:]) **by** *auto*
from *jordan-nf-exists*[*OF A this*] **obtain** *n-as* **where** *jnfA*: *jordan-nf A n-as* **..**
with *id* **have** *jnfB*: *jordan-nf B n-as* **by** *simp*
from *jnfA jnfB* **show** *similar-mat A B*
unfolding *jordan-nf-def* **using** *similar-mat-trans similar-mat-sym* **by** *blast*
qed

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 −
from *char-poly-factorized*[*OF A*] **obtain** *as* **where** *char-poly A* = (∏ *a*←*as*. [:− *a*, *1*:]) **by** *auto*
from *jordan-nf-exists*[*OF A this*] **obtain** *n-as* **where** *jnf*: *jordan-nf A n-as* **..**
show *?thesis* **unfolding** *jordan-nf-order*[*OF jnf*] *jordan-nf-order*[*OF jordan-nf-smult*[*OF jnf k*]]
by (*induct n-as*, *auto simp*: *k*)
qed

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* $\widehat{\ }_m$ *k*) (*c1* + *c2* ∗ *of-nat k* ^ (*N* − *1*))
by (*rule factored-char-poly-norm-bound*[*OF A linear-factors jordan-nf-exists*[*OF A linear-factors*] *le-1 le-N*])

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 −
from *triangular-to-jnf-vector*[*OF A upper-t*] **have** *jnf*: ∃ *n-as*. *jordan-nf A n-as* **..**
show *?thesis*
by (*rule factored-char-poly-norm-bound*[*OF A char-poly-upper-triangular*[*OF A upper-t*] *jnf le-1 le-N*])
qed

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*: $A \in$ *carrier-mat n n*
  and *upper-t*: *upper-triangular* (*A* :: *'a* :: *real-normed-field mat*)
  and *le-1*: $\bigwedge$ *n a*. (*n*,*a*) $\in$ *set* (*triangular-to-jnf-vector A*) $\Longrightarrow$ *norm a* $\leq$ *1*
  and *le-N*: $\bigwedge$ *n a*. (*n*,*a*) $\in$ *set* (*triangular-to-jnf-vector A*) $\Longrightarrow$ *norm a* = *1* $\Longrightarrow$ *n* $\leq$ *N*
  shows $\exists$ *c1 c2*. $\forall$ *k*. *norm-bound* ($A \;\widehat{}_m\; k$) (*c1* + *c2* $*$ *of-nat k* ^ (*N* − *1*))
proof −
  let *?jnf* = *triangular-to-jnf-vector A*
  from *triangular-to-jnf-vector*[*OF A upper-t*] **have** *jnf*: *jordan-nf A ?jnf* **.**
  show *?thesis*
    by (*rule jordan-nf-matrix-poly-bound*[*OF A le-1 le-N jnf*])
qed

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)
by *pat-completeness auto*

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*)
unfolding *strassen-measure-def strassen-even-def* **by** *auto*

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 −
{
fix *Ai Bi*
assume *Ai*: *Ai* ∈ {*A1*,*A2*,*A3*,*A4*} **and** *Bi*: *Bi* ∈ {*B1*,*B2*,*B3*,*B4*}
from *large*[*unfolded strassen-too-small-def strassen-constant-def*]
have ¬ *dim-row A* < *2* **by** *auto*
with *assms Ai Bi* **have** *Ar*:
dim-row Ai < *dim-row A*
dim-col Ai ≤ *dim-col A*
dim-col Bi ≤ *dim-col B*
unfolding *split-block-def Let-def* **by** *auto*
hence *strassen-measure* (*Ai*,*Bi*) < *strassen-measure* (*A*,*B*)
unfolding *strassen-measure-def split* **by** *auto*
}
thus
strassen-measure (*A1*,*B2*) < *strassen-measure* (*A*,*B*)
strassen-measure (*A1*,*B4*) < *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*)
by *auto*
qed

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 −
from *assms* **have** *Ar*:
dim-row A1 < *dim-row A* ∨ *dim-row A1* = *dim-row A* ∧ *even* (*dim-row A*)

unfolding *split-block-def Let-def* **by** *auto presburger*
from *assms* **have** *Ac*:
dim-col A1 $<$ *dim-col A* $\vee$ *dim-col A1* $=$ *dim-col A* $\wedge$ *even* (*dim-col A*)
unfolding *split-block-def Let-def* **by** *auto presburger*
from *assms* **have** *Bc*:
dim-col B1 $<$ *dim-col B* $\vee$ *dim-col B1* $=$ *dim-col B* $\wedge$ *even* (*dim-col B*)
unfolding *split-block-def Let-def* **by** *auto presburger*
from *Ar Ac Bc odd* **show** *?thesis* **unfolding** *strassen-measure-def strassen-even-def split*
by (*auto split*: *if-splits*)
qed

termination by (*relation measure strassen-measure*,
auto elim: *strassen-measure-div-2 strassen-measure-odd*)

lemma *strassen-mat-mult*:
dim-col A $=$ *dim-row B* $\Longrightarrow$ *strassen-mat-mult A B* $=$ *A* $*$ *B*
proof (*induct A B rule*: *strassen-mat-mult.induct*)
case (*1 A B*)
let *?nr* $=$ *dim-row A*
let *?nc* $=$ *dim-col B*
let *?n* $=$ *dim-col A*
show *?case*
proof (*cases strassen-too-small A B*)
case *False* **note** *large* $=$ *this*
let *?smm* $=$ *strassen-mat-mult*
note *IH* $=$ *1*(*1*$-$*8*)[*OF refl refl refl False - refl refl refl - refl refl refl - refl refl refl*]
show *?thesis*
proof (*cases strassen-even A B*)
case *True*
note *even* $=$ *True*[*unfolded strassen-even-def*]
let *?nr2* $=$ *?nr div 2*
let *?n2* $=$ *?n div 2*
let *?nc2* $=$ *?nc div 2*
from *even* **have** *nr*: *?nr* $=$ *?nr2* $+$ *?nr2* **by** *presburger*
from *even* **have** *n*: *?n* $=$ *?n2* $+$ *?n2* **by** *presburger*
from *even* **have** *nc*: *?nc* $=$ *?nc2* $+$ *?nc2* **by** *presburger*
from *1*(*9*) *even* **have** *n'*: *dim-row B* $=$ *?n2* $+$ *?n2*
by *auto*
obtain *A1 A2 A3 A4* **where** *splitA*:
split-block A ?nr2 ?n2 $=$ (*A1*,*A2*,*A3*,*A4*) **by** (*rule prod-cases4*)
obtain *B1 B2 B3 B4* **where** *splitB*:
split-block B ?n2 ?nc2 $=$ (*B1*,*B2*,*B3*,*B4*) **by** (*rule prod-cases4*)
note *IH* $=$ *IH*(*1*$-$*7*)[*OF True splitA*[*symmetric*] *splitB*[*symmetric*]]
from *split-block*[*OF splitA nr n*]
have *blockA*: *A* $=$ *four-block-mat A1 A2 A3 A4*
and *A1*: *A1* $\in$ *carrier-mat ?nr2 ?n2*

and *A2*: *A2* ∈ *carrier-mat ?nr2 ?n2*
and *A3*: *A3* ∈ *carrier-mat ?nr2 ?n2*
and *A4*: *A4* ∈ *carrier-mat ?nr2 ?n2*
by *blast+*
from *split-block*[*OF splitB n′ nc*]
have *blockB*: *B* = *four-block-mat B1 B2 B3 B4*
and *B1*: *B1* ∈ *carrier-mat ?n2 ?nc2*
and *B2*: *B2* ∈ *carrier-mat ?n2 ?nc2*
and *B3*: *B3* ∈ *carrier-mat ?n2 ?nc2*
and *B4*: *B4* ∈ *carrier-mat ?n2 ?nc2*
by *blast+*
note *carr* = *A1 A2 A3 A4 B1 B2 B3 B4*
let *?M11* = *A1* + *A4* **let** *?M12* = *B1* + *B4*
let *?M21* = *A3* + *A4* **let** *?M22* = *B1*
let *?M31* = *A1* **let** *?M32* = *B2* − *B4*
let *?M41* = *A4* **let** *?M42* = *B3* − *B1*
let *?M51* = *A1* + *A2* **let** *?M52* = *B4*
let *?M61* = *A3* − *A1* **let** *?M62* = *B1* + *B2*
let *?M71* = *A2* − *A4* **let** *?M72* = *B3* + *B4*
let *?M1* = *?smm ?M11 ?M12*
let *?M2* = *?smm ?M21 ?M22*
let *?M3* = *?smm ?M31 ?M32*
let *?M4* = *?smm ?M41 ?M42*
let *?M5* = *?smm ?M51 ?M52*
let *?M6* = *?smm ?M61 ?M62*
let *?M7* = *?smm ?M71 ?M72*
let *?C1* = *?M1* + *?M4* − *?M5* + *?M7*
let *?C2* = *?M3* + *?M5*
let *?C3* = *?M2* + *?M4*
let *?C4* = *?M1* − *?M2* + *?M3* + *?M6*
have *res*: *?smm A B* = *four-block-mat ?C1 ?C2 ?C3 ?C4*
using *large True*
unfolding *strassen-mat-mult.simps*[*of A B*] *Let-def splitA splitB split*
by *auto*
have *M1*: *?M1* = *?M11* ∗ *?M12*
by (*rule IH*(*1*), *insert carr*, *auto*)
note *IH* = *IH*(*2*−)[*OF refl*]
have *M2*: *?M2* = *?M21* ∗ *?M22*
by (*rule IH*(*1*), *insert carr*, *auto*)
note *IH* = *IH*(*2*−)[*OF refl*]
have *M3*: *?M3* = *?M31* ∗ *?M32*
by (*rule IH*(*1*), *insert carr*, *auto*)
note *IH* = *IH*(*2*−)[*OF refl*]
have *M4*: *?M4* = *?M41* ∗ *?M42*
by (*rule IH*(*1*), *insert carr*, *auto*)
note *IH* = *IH*(*2*−)[*OF refl*]
have *M5*: *?M5* = *?M51* ∗ *?M52*
by (*rule IH*(*1*), *insert carr*, *auto*)
note *IH* = *IH*(*2*−)[*OF refl*]

```
    have M6: ?M6 = ?M61 * ?M62
      by (rule IH(1), insert carr, auto)
    note IH = IH(2−)[OF refl]
    have M7: ?M7 = ?M71 * ?M72
      by (rule IH(1), insert carr, auto)
    note distr =
      add-mult-distrib-mat[of - ?nr2 ?n2 - - ?nc2]
      minus-mult-distrib-mat[of - ?nr2 ?n2 - - ?nc2]
      mult-add-distrib-mat[of - ?nr2 ?n2 - ?nc2]
      mult-minus-distrib-mat[of - ?nr2 ?n2 - ?nc2]
    note closed = add-carrier-mat[of - ?nr2 ?nc2]
       uminus-carrier-iff-mat[of - ?nr2 ?nc2]
    note ac = assoc-add-mat[of - ?nr2 ?nc2] comm-add-mat[of - ?nr2 ?nc2]
    show ?thesis unfolding res M1 M2 M3 M4 M5 M6 M7
      unfolding blockA blockB
        mult-four-block-mat[OF carr]
      by (rule cong-four-block-mat)
         (insert carr, auto simp: distr ac closed)
  next
    case False
    let ?nr2 = ?nr div 2 * 2 let ?nr2' = ?nr − ?nr2
    let ?n2 = ?n div 2 * 2  let ?n2' = ?n − ?n2
    let ?nc2 = ?nc div 2 * 2 let ?nc2' = ?nc − ?nc2
    have nr: ?nr = ?nr2 + ?nr2' by presburger
    have n: ?n = ?n2 + ?n2' by presburger
    have nc: ?nc = ?nc2 + ?nc2' by presburger
    from 1(9) have n': dim-row B = ?n2 + ?n2' by auto
    obtain A1 A2 A3 A4 where splitA:
      split-block A ?nr2 ?n2 = (A1,A2,A3,A4) by (rule prod-cases4)
    obtain B1 B2 B3 B4 where splitB:
      split-block B ?n2 ?nc2 = (B1,B2,B3,B4) by (rule prod-cases4)
    note IH = IH(8)[OF False splitA[symmetric] splitB[symmetric]]
    from split-block[OF splitA nr n]
    have blockA: A = four-block-mat A1 A2 A3 A4
      and A1: A1 ∈ carrier-mat ?nr2 ?n2
      and A2: A2 ∈ carrier-mat ?nr2 ?n2'
      and A3: A3 ∈ carrier-mat ?nr2' ?n2
      and A4: A4 ∈ carrier-mat ?nr2' ?n2'
      by blast+
    from split-block[OF splitB n' nc]
    have blockB: B = four-block-mat B1 B2 B3 B4
      and B1: B1 ∈ carrier-mat ?n2 ?nc2
      and B2: B2 ∈ carrier-mat ?n2 ?nc2'
      and B3: B3 ∈ carrier-mat ?n2' ?nc2
      and B4: B4 ∈ carrier-mat ?n2' ?nc2'
      by blast+
    note carr = A1 A2 A3 A4 B1 B2 B3 B4
    from carr have dim-col A1 = dim-row B1 by simp
    note IH = IH[OF this]
```

```
    have ?smm A B = four-block-mat
      (A1 * B1 + A2 * B3)
      (A1 * B2 + A2 * B4)
      (A3 * B1 + A4 * B3)
      (A3 * B2 + A4 * B4)
      unfolding strassen-mat-mult.simps[of A B] Let-def
        splitA splitB split IH using large False by auto
    also have ... = A * B
      unfolding blockA blockB
       mult-four-block-mat[OF carr] by simp
    finally show ?thesis by simp
  qed
 qed simp
qed

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 · col B j)
  unfolding basic-mat-mult-def by auto
```

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)
  using strassen-mat-mult[of A B] unfolding basic-mat-mult-def by auto

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 $A \geq_m B$
  using *assms* **unfolding** *mat-ge-def* **by** *auto*

lemma *mat-geD*[*dest*]: **assumes** $A \geq_m B$ **and** $i < dim\text{-}row\ A\ j < dim\text{-}col\ A$
  shows $A\ \$\$\ (i,j) \geq B\ \$\$\ (i,j)$
  using *assms* **unfolding** *mat-ge-def* **by** *auto*

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-gt gt sd A B*
  using *assms* **unfolding** *mat-gt-def* **by** *auto*

lemma *mat-gtD*[*dest*]: **assumes** *mat-gt gt sd A B*
  shows $A \geq_m B\ \exists\ i < sd.\ \exists\ j < sd.\ gt\ (A\ \$\$\ (i,j))\ (B\ \$\$\ (i,j))$
  using *assms* **unfolding** *mat-gt-def* **by** *auto*

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*]:
  $max_m\ A\ B \in carrier\text{-}mat\ (dim\text{-}row\ A)\ (dim\text{-}col\ A)$

unfolding *mat-max-def* **by** *auto*

lemma *mat-max-closed*[*intro*]:
$A \in$ *carrier-mat nr nc* $\Longrightarrow$ $B \in$ *carrier-mat nr nc* $\Longrightarrow$ max_m A $B \in$ *carrier-mat nr nc*
unfolding *mat-max-def* **by** *auto*

lemma *mat-max-index*:
assumes $i <$ *dim-row* A $j <$ *dim-col* A
shows (*mat-max* A B) \$\$ $(i,j) = max$ (A \$\$ (i,j)) (B \$\$ (i,j))
unfolding *mat-max-def* **using** *index-mat assms* **by** *auto*

definition (**in** *zero*) *mat-default* :: $'a \Rightarrow nat \Rightarrow 'a$ *mat* (‹$default_m$›) **where**
$default_m$ d n = *mat* n n (λ (i,j). *if* $i = j$ *then* d *else* 0)

lemma *mat-default-carrier*[*simp*]: $default_m$ d $n \in$ *carrier-mat* n n
unfolding *mat-default-def* **by** *auto*
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-mat nr nc* $B \in$ *carrier-mat nr nc*
shows $A \geq_m C$
using *assms ge-trans*[*of* B \$\$ (i,j) A \$\$ (i,j) **for** i j]
unfolding *mat-ge-def carrier-mat-def* **by** *auto*

lemma *mat-ge-refl*: $A \geq_m A$
unfolding *mat-ge-def* **by** (*auto simp*: *ge-refl*)

lemma *mat-max-comm*: $A \in$ *carrier-mat nr nc* $\Longrightarrow$ $B \in$ *carrier-mat nr nc* $\Longrightarrow$ max_m A B = max_m B A
unfolding *mat-max-def* **by** (*intro eq-matI*, *auto simp*: *max-comm*)

lemma *mat-max-ge*: max_m A $B \geq_m A$
unfolding *mat-max-def* **by** (*intro mat-geI*[*of - dim-row A dim-col A*], *auto*)

lemma *mat-max-ge-0*: $A \in$ *carrier-mat nr nc* $\Longrightarrow$ $B \in$ *carrier-mat nr nc* $\Longrightarrow$ $A \geq_m B \Longrightarrow max_m$ A $B = A$
unfolding *mat-max-def* **by** (*intro eq-matI*, *auto simp*: *max-id*)

lemma *mat-max-mono*: $A \geq_m B \Longrightarrow$
$A \in$ *carrier-mat nr nc* $\Longrightarrow$ $B \in$ *carrier-mat nr nc* $\Longrightarrow$ $C \in$ *carrier-mat nr nc* $\Longrightarrow$
max_m C $A \geq_m max_m$ C B

by (*intro mat-geI*[*of* - *nr nc*], *auto simp*: *max-mono mat-max-def*)
end

lemma *mat-plus-left-mono*: $A \geq_m (B :: 'a :: \textit{ordered-ab-semigroup mat})$
$\Longrightarrow A \in \textit{carrier-mat nr nc} \Longrightarrow B \in \textit{carrier-mat nr nc} \Longrightarrow C \in \textit{carrier-mat nr nc}$
$\Longrightarrow A + C \geq_m B + C$
by (*intro mat-geI*[*of* - *nr nc*], *auto simp*: *plus-left-mono*)

lemma *mat-plus-right-mono*: $B \geq_m (C :: 'a :: \textit{ordered-ab-semigroup mat})$
$\Longrightarrow A \in \textit{carrier-mat nr nc} \Longrightarrow B \in \textit{carrier-mat nr nc} \Longrightarrow C \in \textit{carrier-mat nr nc}$
$\Longrightarrow A + B \geq_m A + C$
by (*intro mat-geI*[*of* - *nr nc*], *auto simp*: *plus-right-mono*)

lemma *plus-mono*: $x_1 \geq (x_2 :: 'a :: \textit{ordered-ab-semigroup}) \Longrightarrow$
$y_1 \geq y_2 \Longrightarrow x_1 + y_1 \geq x_2 + y_2$
using *ge-trans*[*OF plus-left-mono*[*of* x_2 x_1] *plus-right-mono*[*of* y_2 y_1]] **.**

Since one cannot use $(\bigwedge i.\ i \in ?K \Longrightarrow ?f\ i \leq ?g\ i) \Longrightarrow \textit{sum}\ ?f\ ?K \leq \textit{sum}\ ?g\ ?K$ (it requires other class constraints like *order*), we make our own copy of this fact.

lemma *sum-mono-ge*:
assumes *ge*: $\bigwedge i.\ i{\in}K \Longrightarrow f\ (i{::}'a) \geq ((g\ i){::}('b{::}\textit{ordered-semiring-0}))$
shows $(\sum i{\in}K.\ f\ i) \geq (\sum i{\in}K.\ g\ i)$
proof (*cases finite K*)
case *True*
thus *?thesis* **using** *ge*
proof *induct*
case *empty*
show *?case* **by** (*simp add*: *ge-refl*)
next
case *insert*
thus *?case* **using** *plus-mono* **by** *fastforce*
qed
next
case *False* **then show** *?thesis* **by** (*simp add*: *ge-refl*)
qed

lemma (**in** *one-mono-ordered-semiring-1*) *sum-mono-gt*:
assumes *le*: $\bigwedge i.\ i{\in}K \Longrightarrow f\ (i{::}'b) \geq ((g\ i){::}'a)$
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 −
have *id*: $\bigwedge f.\ (\sum i{\in}K.\ f\ i) = f\ i + (\sum i{\in}\ K - \{i\}.\ f\ i)$
by (*rule sum.remove*[*OF K i*])
have *ge*: $(\sum i{\in}\ K - \{i\}.\ f\ i) \geq (\sum i{\in}\ K - \{i\}.\ g\ i)$

by (*rule sum-mono-ge*[*OF le*], *auto*)
show *?thesis* **unfolding** *id* **using** *compat*[*OF plus-right-mono*[*OF ge*] *plus-gt-left-mono*[*OF gt*]] .
qed

lemma *scalar-left-mono*: **assumes**
$u \in$ *carrier-vec* n $v \in$ *carrier-vec* n $w \in$ *carrier-vec* n
and $\bigwedge i.\ i < n \Longrightarrow u \$ i \geq v \$ i$
and $\bigwedge i.\ i < n \Longrightarrow w \$ i \geq (0 :: 'a :: \mathit{ordered\text{-}semiring\text{-}0})$
shows $u \cdot w \geq v \cdot w$ **unfolding** *scalar-prod-def*
by (*intro sum-mono-ge times-left-mono*, *insert assms*, *auto*)

lemma *scalar-right-mono*: **assumes**
$u \in$ *carrier-vec* n $v \in$ *carrier-vec* n $w \in$ *carrier-vec* n
and $\bigwedge i.\ i < n \Longrightarrow v \$ i \geq w \$ i$
and $\bigwedge i.\ i < n \Longrightarrow u \$ i \geq (0 :: 'a :: \mathit{ordered\text{-}semiring\text{-}0})$
shows $u \cdot v \geq u \cdot w$
proof −
have *dim*: *dim-vec* v = *dim-vec* w **using** *assms* **by** *auto*
show *?thesis* **unfolding** *scalar-prod-def dim*
by (*intro sum-mono-ge times-right-mono*, *insert assms*, *auto*)
qed

lemma *mat-mult-left-mono*: **assumes** *C0*: $C \geq_m 0_m\ n\ n$
and *AB*: $A \geq_m (B :: 'a :: \mathit{ordered\text{-}semiring\text{-}0}\ \mathit{mat})$
and *carr*: $A \in$ *carrier-mat* n n $B \in$ *carrier-mat* n n $C \in$ *carrier-mat* n n
shows $A * C \geq_m B * C$
proof −
{
fix *i j*
assume *i*: $i < n$ $j < n$
have *row* A i $\cdot$ *col* C $j \geq$ *row* B i $\cdot$ *col* C j
by (*rule scalar-left-mono*[*of - n*], *insert C0 AB carr i*, *auto*)
}
thus *?thesis*
by (*intro mat-geI*[*of - n n*], *insert carr*, *auto*)
qed

lemma *mat-mult-right-mono*: **assumes** *A0*: $A \geq_m 0_m\ n\ n$
and *BC*: $B \geq_m (C :: 'a :: \mathit{ordered\text{-}semiring\text{-}0}\ \mathit{mat})$
and *carr*: $A \in$ *carrier-mat* n n $B \in$ *carrier-mat* n n $C \in$ *carrier-mat* n n
shows $A * B \geq_m A * C$
proof −
{
fix *i j*
assume *i*: $i < n$ $j < n$
have *row* A i $\cdot$ *col* B $j \geq$ *row* A i $\cdot$ *col* C j
by (*rule scalar-right-mono*[*of - n*], *insert A0 BC carr i*, *auto*)
}

thus *?thesis*
by (*intro mat-geI*[*of* - *n n*], *insert carr*, *auto*)
qed

lemma *one-mat-ge-zero*: (1_m *n* :: *'a* :: *ordered-semiring-1 mat*) $\geq_m$ 0_m *n n*
by (*intro mat-geI*[*of* - *n n*], *auto simp*: *one-ge-zero ge-refl*)

context *order-pair*
begin
lemma *mat-ge-gt-trans*: **assumes** *sd*: *sd* $\leq$ *n* **and** *AB*: *A* $\geq_m$ *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 −
from *mat-gtD*[*OF BC*] **obtain** *i j* **where** *ij*: *i* < *sd j* < *sd* **and** *gt*: *B* $$ (*i*, *j*) $\succ$ *C* $$ (*i*, *j*)
and *BC*: *B* $\geq_m$ *C* **by** *auto*
from *mat-ge-trans*[*OF AB BC A B*] **have** *AC*: *A* $\geq_m$ *C* **.**
from *mat-geD*[*OF AB*, *of i j*] *A sd ij* **have** *ge*: *A* $$ (*i*,*j*) $\geq$ *B* $$ (*i*,*j*) **by** *auto*
from *compat*[*OF ge gt*] **have** *gt*: *A* $$ (*i*, *j*) $\succ$ *C* $$ (*i*, *j*) **.**
with *ij AC* **show** *?thesis* **by** *auto*
qed

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 −
from *mat-gtD*[*OF AB*] **obtain** *i j* **where** *ij*: *i* < *sd j* < *sd* **and** *gt*: *A* $$ (*i*, *j*) $\succ$ *B* $$ (*i*, *j*)
and *AB*: *A* $\geq_m$ *B* **by** *auto*
from *mat-ge-trans*[*OF AB BC A B*] **have** *AC*: *A* $\geq_m$ *C* **.**
from *mat-geD*[*OF BC*, *of i j*] *B sd ij* **have** *ge*: *B* $$ (*i*,*j*) $\geq$ *C* $$ (*i*,*j*) **by** *auto*
from *compat2*[*OF gt ge*] **have** *gt*: *A* $$ (*i*, *j*) $\succ$ *C* $$ (*i*, *j*) **.**
with *ij AC* **show** *?thesis* **by** *auto*
qed

lemma *mat-gt-imp-mat-ge*: *mat-gt gt sd A B* $\Longrightarrow$ *A* $\geq_m$ *B*
by (*rule mat-gtD*)

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*
using *mat-ge-gt-trans*[*OF sd mat-gt-imp-mat-ge*[*OF AB*] *BC A B*] **.**

lemma *mat-default-ge-0*: $default_m$ *default n* $\geq_m$ 0_m *n n*
by (*intro mat-geI*[*of* - *n n*], *auto simp*: *mat-default-def default-ge-zero ge-refl*)
end

definition *mat-ordered-semiring* :: *nat* ⇒ *nat* ⇒ (*'a* :: *ordered-semiring-1* ⇒ *'a* ⇒ *bool*) ⇒ *'b* ⇒ (*'a mat*,*'b*) *ordered-semiring-scheme* **where**
mat-ordered-semiring n sd gt b ≡ *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* ≤ *n*
shows *ordered-semiring*
(*mat-ordered-semiring n sd* (≻) *b* :: (*'a mat*,*'b*) *ordered-semiring-scheme*)
(**is** *ordered-semiring ?R*)
proof −
interpret *semiring ?R* **unfolding** *mat-ordered-semiring-def* **by** (*rule semiring-mat*)
show *?thesis*
by (*unfold-locales*, *unfold ring-mat-def mat-ordered-semiring-def ordered-semiring-record-simps*,
auto intro: *mat-ge-trans mat-ge-refl mat-ge-gt-trans*[*OF sd-n*] *mat-gt-ge-trans*[*OF sd-n*] *mat-max-comm*
mat-max-ge mat-max-ge-0 mat-max-mono one-mat-ge-zero mat-gt-trans[*OF sd-n*] *mat-gt-imp-mat-ge*
mat-plus-left-mono mat-mult-left-mono mat-mult-right-mono)
qed

context *weak-SN-strict-mono-ordered-semiring-1*
begin

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* ⟹ *mat-gt weak-gt sd A B*
shows ∃ *gt*. *SN-strict-mono-ordered-semiring-1 default gt mono* ∧
(∀ *A B*. *A* ∈ *carrier-mat n n* ⟶ *B* ∈ *carrier-mat n n* ⟶ (*A*, *B*) ∈ *set ABs* ⟶ *mat-gt gt sd A B*)
proof −
let *?n* = [*0* ..< *n*]
let *?m1x* = [*A* $$ (*i*,*j*) . *A* <− *map fst ABs*, *i* <− *?n*, *j* <− *?n*]
let *?m2y* = [*B* $$ (*i*,*j*) . *B* <− *map snd ABs*, *i* <− *?n*, *j* <− *?n*]
let *?pairs* = *concat* (*map* (λ *x*. *map* (λ *y*. (*x*,*y*)) *?m2y*) *?m1x*)
let *?strict* = *filter* (λ (*x*,*y*). *weak-gt x y*) *?pairs*
have ∀ *x y*. (*x*,*y*) ∈ *set ?strict* ⟶ *weak-gt x y* **by** *auto*
from *weak-gt-mono*[*OF this*] **obtain** *gt* **where** *order*: *SN-strict-mono-ordered-semiring-1 default gt mono*
and *orient2*: ⋀ *x y*. (*x*, *y*) ∈ *set ?strict* ⟹ *gt x y* **by** *auto*
show *?thesis*
proof (*intro exI allI conjI impI*, *rule order*)
fix *A B*
assume *A*: *A* ∈ *carrier-mat n n* **and** *B*: *B* ∈ *carrier-mat n n*

and *AB*: (*A*, *B*) ∈ *set ABs*
from *orient*[*OF this*] **have** *mat-gt weak-gt sd A B* **by** *auto*
from *mat-gtD*[*OF this*] **obtain** *i j* **where**
ge: $A \geq_m B$ **and** *ij*: *i* < *sd j* < *sd* **and** *wgt*: *weak-gt* (*A* $$ (*i*,*j*)) (*B* $$ (*i*,*j*))
by *auto*
from *ij* ‹*sd* ≤ *n*› **have** *ij′*: *i* < *n j* < *n* **by** *auto*
have *gt*: *gt* (*A* $$ (*i*,*j*)) (*B* $$ (*i*,*j*))
by (*rule orient2*, *insert ij′ AB wgt*, *force*)
show *mat-gt gt sd A B* **using** *ij gt ge* **by** *auto*
qed
qed
end

lemma *sum-mat-mono*:
assumes *A*: *A* ∈ *carrier-mat nr nc* **and** *B*: *B* ∈ *carrier-mat nr nc*
and *AB*: $A \geq_m$ (*B* :: ′*a* :: *ordered-semiring-0 mat*)
shows *sum-mat A* ≥ *sum-mat B*
proof −
from *A B* **have** *id*: *dim-row B* = *dim-row A dim-col B* = *dim-col A* **by** *auto*
show *?thesis* **unfolding** *sum-mat-def id*
by (*rule sum-mono-ge*, *insert mat-geD*[*OF AB*] *id*, *auto*)
qed

context *one-mono-ordered-semiring-1*
begin
lemma *sum-mat-mono-gt*:
assumes *sd* ≤ *n*
and *A*: *A* ∈ *carrier-mat n n* **and** *B*: *B* ∈ *carrier-mat n n*
and *AB*: *mat-gt* (≻) *sd A* (*B* :: ′*a mat*)
shows *sum-mat A* ≻ *sum-mat B*
proof −
from *A B* **have** *id*: *dim-row B* = *dim-row A dim-col B* = *dim-col A* **by** *auto*
from *mat-gtD*[*OF AB*] **obtain** *i j* **where** *AB*: $A \geq_m B$ **and**
ij: *i* < *sd j* < *sd* **and** *gt*: *A* $$ (*i*,*j*) ≻ *B* $$ (*i*,*j*) **by** *auto*
show *?thesis* **unfolding** *sum-mat-def id*
by (*rule sum-mono-gt*[*of - - -* (*i*,*j*)], *insert ij gt mat-geD*[*OF AB*] *A B* ‹*sd* ≤ *n*›, *auto*)
qed

lemma *mat-plus-gt-left-mono*: **assumes** *sd-n*: *sd* ≤ *n* **and** *gt*: *mat-gt* (≻) *sd A B*

and *A*: *A* ∈ *carrier-mat n n* **and** *B*: *B* ∈ *carrier-mat n n* **and** *C*: *C* ∈ *carrier-mat n n*
shows *mat-gt* (≻) *sd* (*A* + *C*) (*B* + *C*)
proof −
note *wf* = *A B C*
from *mat-gtD*[*OF gt*] **obtain** *i j*
where *AB*: $A \geq_m B$ **and** *ij*: *i* < *sd j* < *sd* **and** *gt*: *A* $$ (*i*,*j*) ≻ *B* $$ (*i*,*j*) **by** *auto*

from *plus-gt-left-mono*[*OF gt*, *of C* $$ (*i*,*j*)]
show *?thesis*
by (*intro mat-gtI*[*OF mat-geI*[*of - n n*] *ij*], *insert mat-geD*[*OF AB*] *wf ij sd-n*, *auto intro*: *plus-left-mono*)
qed

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 $(A + C)$ $(B + D)$
by (*rule mat-gt-ge-trans*[*OF - mat-plus-gt-left-mono mat-plus-right-mono*[*OF mat-gt-imp-mat-ge*]], *auto*)

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 −
from *assms* **have** *n*: $n > 0$ **by** *auto*
show *?thesis*
by (*intro mat-gtI*[*OF mat-default-ge-0 sd-pos sd-pos*], *insert sd-pos sd-n*, *auto simp*: *mat-default-def default-gt-zero*)
qed
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
fix *A*
assume $\forall\ i.\ (A\ i,\ A\ (Suc\ i)) \in \{(m1,m2).\ m1 \succ_m n\ sd\ m2\}$
hence $\bigwedge\ i.\ (A\ i,\ A\ (Suc\ i)) \in \{(m1,m2).\ m1 \succ_m n\ sd\ m2\}$ **by** *blast*
hence *A*: $\bigwedge\ i.\ A\ i \in$ *carrier-mat n n*
and *ge*: $\bigwedge\ i.\ A\ (Suc\ i) \geq_m 0_m\ n\ n$
and *gt*: $\bigwedge\ i.$ *mat-gt* $(\succ)$ *sd* $(A\ i)$ $(A\ (Suc\ i))$ **by** *auto*
define *s* **where** $s = (\lambda\ i.$ *sum-mat* $(A\ i))$
{
fix *i*
from *sum-mat-mono-gt*[*OF sd-n A A gt*[*of i*]]
have *gt*: $s\ i \succ s\ (Suc\ i)$ **unfolding** *s-def* **.**

from *sum-mat-mono*[*OF A - ge*[*of i*]]
have *ge*: *s* (*Suc i*) $\geq$ *0* **unfolding** *s-def* **by** *auto*
note *ge gt*
}
with *SN* **show** *False* **by** *auto*
qed
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* ($\succ$) *sd B C* **and** *gez*: *A* $\geq_m$ 0_m *n n* **and** *mmono*: *mat-mono mono sd A*
shows *mat-gt* ($\succ$) *sd* (*A* $*$ *B*) (*A* $*$ *C*) (**is** *mat-gt - - ?AB ?AC*)
proof −
from *mat-gtD*[*OF gt*] **obtain** *i j* **where**
i: *i* < *sd* **and** *j*: *j* < *sd* **and** *gt*: *B* $$ (*i*,*j*) $\succ$ *C* $$ (*i*,*j*) **and** *BC*: *B* $\geq_m$ *C* **by** *auto*
from *mat-mult-right-mono*[*OF gez BC A B C*] **have** *ge*: *?AB* $\geq_m$ *?AC* **.**
from *mmono*[*unfolded mat-mono-def*] *i* **obtain** *k* **where** *k*: *k* < *sd* **and** *mon*: *mono* (*A* $$ (*k*,*i*)) **by** *auto*
from *mat-geD*[*OF gez*] *k i sd-n A* **have** *A* $$ (*k*, *i*) $\geq$ *0* **by** *auto*
note *mono* = *mono*[*OF mon gt this*]
have *id*: *dim-vec* (*col B j*) = *n dim-vec* (*col C j*) = *n* **using** *j sd-n B C* **by** *auto*
{
fix *i*
assume *i* < *n*
hence *row A k* \$ *i* $*$ *col B j* \$ *i* $\geq$ *row A k* \$ *i* $*$ *col C j* \$ *i*
by (*intro times-right-mono*, *insert j k sd-n A B C mat-geD*[*OF gez*] *mat-geD*[*OF BC*], *auto*)
} note *sge* = *this*
have *gt*: *row A k* $\cdot$ *col B j* $\succ$ *row A k* $\cdot$ *col C j* **unfolding** *scalar-prod-def id*
by (*rule sum-mono-gt*[*of - - - i*, *OF sge*], *insert mono k i j A B C sd-n*, *auto*)
show *?thesis*
by (*rule mat-gtI*[*OF ge k j*], *insert k j sd-n A B C gt*, *auto*)
qed
end

definition *mat-comp-all* :: ($'a \Rightarrow 'a \Rightarrow$ *bool*) $\Rightarrow$ $'a$ *mat* $\Rightarrow$ $'a$ *mat* $\Rightarrow$ *bool*
where *mat-comp-all r A B* =
($\forall$ *i* < *dim-row A*. $\forall$ *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 $\bigwedge$ *i j*. *i* < *nr* $\Longrightarrow$ *j* < *nc* $\Longrightarrow$ *r* (*A* $$(*i*,*j*)) (*B* $$ (*i*,*j*))
shows *mat-comp-all r A B*
unfolding *mat-comp-all-def* **using** *assms* **by** *simp*

```
lemma mat-comp-allE:
  assumes mat-comp-all r A B
  and A ∈ carrier-mat nr nc B ∈ carrier-mat nr nc
  shows ⋀ i j. i < nr ⟹ j < nc ⟹ r (A $$ (i,j)) (B $$(i,j))
  using assms unfolding mat-comp-all-def by auto

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 −
  let ?pairs = [ (fst AB $$ (i,j), snd AB $$ (i,j)) . AB <− ABs, i <− [0 ..< n],
j <− [0 ..< n]]
  let ?strict = filter (λ (x,y). weak-gt x y) ?pairs
  have ∀ x y. (x,y) ∈ set ?strict ⟶ weak-gt x y by auto
  from weak-gt-both-mono[OF this]
  obtain gt
    where order: SN-both-mono-ordered-semiring-1 default gt arc-pos
    and orient2: ⋀ x y. (x, y) ∈ set ?strict ⟹ gt x y
    by auto
  {
    fix A B assume AB: (A,B) ∈ set ABs
    hence A: A ∈ carrier-mat n n and B: B ∈ carrier-mat n n
      using AB ABs by auto
    have mat-comp-all gt A B
    proof (rule mat-comp-allI[OF A B])
      fix i j
      assume i: i < n and j: j < n
      from mat-comp-allE[OF - A B this] orient AB
      have weak-gt: weak-gt (A $$(i,j)) (B $$ (i,j)) (is weak-gt ?x ?y) by auto
      have (?x,?y) ∈ set ?pairs using A AB i j by force
      with weak-gt
      have gt: (?x,?y) ∈ set ?strict by simp
      show gt ?x ?y by (rule orient2[OF gt])
    qed
  }
  hence ∀ (A, B)∈set ABs. mat-comp-all gt A B by auto
  thus ?thesis using order by auto
qed
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*
  using *mat-max-ge-0*[*OF A B ge*] **.**

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 (*rule mat-comp-allI*[*OF A C*])
  fix *i j*
  assume *i*: *i* < *nr* **and** *j*: *j* < *nc*
  from *mat-comp-allE*[*OF A-B A B i j*] *mat-comp-allE*[*OF B-C B C i j*]
  show *A* $$ (*i*,*j*) ≻ *C* $$ (*i*,*j*) **by** (*rule gt-trans*)
qed

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 (*rule mat-comp-allI*[*OF A C*])
  fix *i j* **assume** *i*: $i < nr$ **and** *j*: $j < nc$
  have *A* \$\$ (*i*,*j*) $\geq$ *B* \$\$ (*i*,*j*) **using** *ge A i j* **by** *auto*
  also have *B* \$\$ (*i*,*j*) $\succ$ *C* \$\$ (*i*,*j*)
    using *mat-comp-allE*[*OF gt B C i j*] **by** *auto*
  finally show *A* \$\$ (*i*,*j*) $\succ$ *C* \$\$ (*i*,*j*) **by** *auto*
qed

lemma *mat-gt-arc-compat2*:
  assumes *gt*: *mat-gt-arc A B*
  and *ge*: *mat-ge B C*
  and *A*: $A \in$ *carrier-mat nr nc*
  and *B*: $B \in$ *carrier-mat nr nc*
  and *C*: $C \in$ *carrier-mat nr nc*
  shows *mat-gt-arc A C*
proof (*rule mat-comp-allI*[*OF A C*])
  fix *i j* **assume** *i*: $i < nr$ **and** *j*: $j < nc$
  have *A* \$\$ (*i*,*j*) $\succ$ *B* \$\$ (*i*,*j*)
    using *mat-comp-allE*[*OF gt*] *A B i j* **by** *auto*
  also have *B* \$\$ (*i*,*j*) $\geq$ *C* \$\$ (*i*,*j*)
    using *ge B i j* **by** *auto*
  finally show *A* \$\$ (*i*,*j*) $\succ$ *C* \$\$ (*i*,*j*) **by** *auto*
qed

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*
  using *subst mat-geI*[*OF A*]
  using *mat-comp-allE*[*OF gt A B*] *gt-imp-ge* **by** *auto*

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 –
  interpret *semiring ?R* **unfolding** *mat-both-ordered-semiring-def* **by** (*rule semiring-mat*)
  show *?thesis*
    apply (*unfold-locales*)
    unfolding *ring-mat-def mat-both-ordered-semiring-def ordered-semiring-record-simps*
    apply(
      auto intro: *mat-max-comm mat-ge-trans*
      mat-plus-left-mono mat-mult-left-mono mat-mult-right-mono mat-ge-refl
      one-mat-ge-zero mat-max-mono mat-max-ge mat-max-id
      mat-gt-arc-trans mat-gt-arc-imp-mat-ge
      mat-gt-arc-compat mat-gt-arc-compat2)

done
qed

lemma *mat0-leastI*:
assumes *A*: $A \in$ *carrier-mat nr nc*
shows *mat-gt-arc A* (0_m *nr nc*)
proof (*rule mat-comp-allI*[*OF A*])
fix *i j*
assume *i*: $i <$ *nr* **and** *j*: $j <$ *nc*
thus *A* $$ (*i*,*j*) $\succ$ 0_m *nr nc* $$ (*i*,*j*) **by** (*auto simp*: *zero-leastI*)
qed *auto*

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*
apply (*rule eq-matI*)
unfolding *index-zero-mat*
using *A*
proof −
fix *i j*
assume *i*: $i <$ *nr* **and** *j*: $j <$ *nc*
show *A* $$ (*i*,*j*) = *0*
using *zero-leastII mat-comp-allE*[*OF gt - A*] *i j* **by** *auto*
qed *auto*

lemma *mat0-leastIII*:
assumes *A*: $A \in$ *carrier-mat nr nc*
shows *mat-ge A* ((0_m *nr nc*) :: *'a mat*)
proof (*rule mat-geI*[*OF A*]; *unfold index-zero-mat*)
fix *i j*
assume *i*: $i <$ *nr* **and** *j*: $j <$ *nc*
show *A* $$ (*i*,*j*) $\geq$ *0* **using** *zero-leastIII* **by** *simp*
qed

lemma *mat-max-0-id*: **fixes** *A* :: *'a mat*
assumes *A*: $A \in$ *carrier-mat nr nc*
shows *mat-max* (0_m *nr nc*) $A = A$
unfolding *mat-max-comm*[*OF zero-carrier-mat A*]
by (*rule mat-max-id*[*OF mat0-leastIII*[*OF A*] *A*], *simp*)

lemma *mat-arc-pos-one*:
assumes *n0*: $n > 0$
shows *mat-arc-posI arc-pos* (1_m *n*)
unfolding *mat-arc-posI-def*
unfolding *arc-pos-one index-one-mat*(*1*)[*OF n0 n0*]
using *arc-pos-one* **by** *simp*

lemma *mat-arc-pos-zero*:
  assumes *n0*: $n > 0$
  shows ¬ *mat-arc-posI arc-pos* (0_m *n n*)
  unfolding *mat-arc-posI-def*
  unfolding *index-zero-mat*(*1*)[*OF n0 n0*] **using** *arc-pos-zero* **by** *simp*

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 −
  show *?thesis*
  proof (*rule mat-comp-allI*)
    fix *i j*
    assume *i*: *i* < *nr* **and** *j*: *j* < *nc*
    hence *ijC*: *i* < *dim-row C j* < *dim-col C*
      and *ijD*: *i* < *dim-row D j* < *dim-col D*
      using *C D* **by** *auto*
    show *?AC* $$ (*i*,*j*) ≻ *?BD* $$ (*i*,*j*)
      unfolding *index-add-mat*(*1*)[*OF ijC*]
      unfolding *index-add-mat*(*1*)[*OF ijD*]
      using *plus-gt-both-mono*
      using *mat-comp-allE*[*OF gt1 A B*] *mat-comp-allE*[*OF gt2 C D*] *i j* **by** *auto*
  qed (*insert A B C D*, *auto*)
qed

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*
  unfolding *vec-comp-all-def* **using** *assms* **by** *auto*

lemma *vec-comp-allE*:
  vec-comp-all r v w ⟹ *i* < *dim-vec v* ⟹ *r* (*v* $ *i*) (*w* $ *i*)
  unfolding *vec-comp-all-def* **by** *auto*

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 −

{ **fix** m **assume** $m \leq n$
  hence $(\sum i<m.\ (u \ \$ \ i) * (w \ \$ \ i)) \succ (\sum i<m.\ (v \ \$ \ i) * (w \ \$ \ i))$
  proof (*induct m*)
    case *0* **show** *?case* **using** *zero-leastI* **by** *simp* **next**
    case (*Suc m*)
      hence *uv*: $u \ \$ \ m \succ v \ \$ \ m$
        using *vec-comp-allE*[*OF uv*] *u* **by** *auto*
      show *?case*
        unfolding *sum.lessThan-Suc*
        apply (*subst plus-gt-both-mono*)
        using *times-gt-left-mono Suc times-gt-left-mono*[*OF uv*] **by** *auto*
  qed
}
from *this*[*OF order.refl*]
show *?thesis*
  unfolding *scalar-prod-def atLeast0LessThan*
  using *w* **by** *auto*
qed

lemma *scalar-prod-right-mono*:
  assumes *u*: $u \in$ *carrier-vec n*
  and *v*: $v \in$ *carrier-vec n*
  and *w*: $w \in$ *carrier-vec n*
  and *vw*: *vec-comp-all gt v w*
  shows *scalar-prod u v* $\succ$ *scalar-prod u w*
proof −
  { **fix** m **assume** $m \leq n$
    hence $(\sum i<m.\ (u \ \$ \ i) * (v \ \$ \ i)) \succ (\sum i<m.\ (u \ \$ \ i) * (w \ \$ \ i))$
    proof (*induct m*)
      case *0* **show** *?case* **using** *zero-leastI* **by** *simp* **next**
      case (*Suc m*)
        hence *vw*: $v \ \$ \ m \succ w \ \$ \ m$
          using *vec-comp-allE*[*OF vw*] *v* **by** *auto*
        show *?case*
          unfolding *sum.lessThan-Suc*
          apply (*subst plus-gt-both-mono*)
          using *times-gt-left-mono Suc times-gt-right-mono*[*OF vw*] **by** *auto*
    qed
  }
  from *this*[*OF order.refl*]
  show *?thesis*
    unfolding *scalar-prod-def atLeast0LessThan*
    using *v w* **by** *auto*
qed

lemma *mat-gt-arc-mult-left-mono*:
  assumes *gt1*: *mat-gt-arc A B*
  and *A*: (*A*::′*a mat*) $\in$ *carrier-mat nr n*
  and *B*: (*B*::′*a mat*) $\in$ *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 (rule mat-comp-allI)
  fix i j assume i: i < nr and j: j < nc
  hence iA: i < dim-row A
    and iB: i < dim-row B
    and jC: j < dim-col C
    using A B C by auto
  show ?AC $$ (i,j) ≻ ?BC $$ (i,j)
    unfolding index-mult-mat(1)[OF iA jC]
    unfolding index-mult-mat(1)[OF iB jC]
  proof(rule scalar-prod-left-mono)
    show row A i ∈ carrier-vec n using A by auto
    show row B i ∈ carrier-vec n using B by auto
    show col C j ∈ carrier-vec n using C by auto
    show rowAB: vec-comp-all (≻) (row A i) (row B i)
    proof (intro vec-comp-allI)
      fix j assume j: j < dim-vec (row A i)
      have A $$ (i,j) ≻ B $$ (i,j)
        using mat-comp-allE[OF gt1 A B i] j A by simp
      thus row A i $ j ≻ row B i $ j
        using A B C i j by simp
    qed
  qed
qed (insert A B C, auto)

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 (rule mat-comp-allI)
  fix i j assume i: i < nr and j: j < nc
  hence iA: i < dim-row A
    and jB: j < dim-col B
    and jC: j < dim-col C
    using A B C by auto
  show ?AB $$ (i,j) ≻ ?AC $$ (i,j)
    unfolding index-mult-mat(1)[OF iA jB]
    unfolding index-mult-mat(1)[OF iA jC]
  proof(rule scalar-prod-right-mono)
    show row A i ∈ carrier-vec n using A by auto
    show col B j ∈ carrier-vec n using B by auto
    show col C j ∈ carrier-vec n using C by auto
    show rowAB: vec-comp-all (≻) (col B j) (col C j)
    proof (intro vec-comp-allI)
      fix i assume i: i < dim-vec (col B j)
      have B $$ (i,j) ≻ C $$ (i,j)
```

```
      using mat-comp-allE[OF gt1 B C] i j B by simp
    thus col B j $ i ≻ col C j $ i
      using A B C i j by simp
  qed
 qed
qed (insert A B C, auto)

lemma mat-arc-pos-plus:
 assumes n0: n > 0
 and A: A ∈ carrier-mat n n
 and B: B ∈ carrier-mat n n
 and arc-pos: mat-arc-pos A
 shows mat-arc-pos (A + B)
 unfolding mat-arc-posI-def
 apply (subst index-add-mat(1))
 using arc-pos-plus[OF arc-pos[unfolded mat-arc-posI-def]]
 assms by auto

lemma scalar-prod-split-head: assumes
 A ∈ carrier-mat n n B ∈ 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))
 unfolding scalar-prod-def
 using assms sum.atLeast-Suc-lessThan by auto


lemma mat-arc-pos-mult:
 assumes n0: n > 0
 and A: A ∈ carrier-mat n n
 and B: B ∈ carrier-mat n n
 and apA: mat-arc-pos A
 and apB: mat-arc-pos B
 shows mat-arc-pos (A * B)
 unfolding mat-arc-posI-def
 apply(subst index-mult-mat(1))
proof −
 let ?prod = row A 0 · col B 0
 let ?head = A $$ (0,0) * B $$ (0,0)
 let ?rest = ∑ i = 1..<n. A $$ (0, i) * B $$ (i, 0)
 have ap: arc-pos ?head
  using apA apB
  unfolding mat-arc-posI-def
  using arc-pos-mult by auto
 have split: ?prod = ?head + ?rest
  by (rule scalar-prod-split-head[OF A B n0])
 show arc-pos (row A 0 · col B 0)
  unfolding split
  using ap arc-pos-plus by auto
qed (insert A B n0, auto)
```

```
lemma mat-arc-pos-mat-default:
  assumes n0: n > 0 shows mat-arc-pos (mat-default default n)
  unfolding mat-arc-posI-def
  unfolding mat-default-def
  unfolding index-mat(1)[OF n0 n0]
  using arc-pos-default by simp

lemma mat-not-all-ge:
  assumes n-pos: n > 0
  and A: A ∈ carrier-mat n n
  and B: B ∈ carrier-mat n n
  and apB: mat-arc-pos B
  shows ∃ C. C ∈ carrier-mat n n ∧ mat-ge C (0m n n) ∧ mat-arc-pos C ∧ ¬
mat-ge A (B ∗ C)
proof −
  define c where c = A $$ (0,0)
  from apB have arc-pos (B $$ (0,0)) unfolding mat-arc-posI-def .
  from not-all-ge[OF this, of c] obtain e where e0: e ≥ 0 and ae: arc-pos e
    and nc: ¬ c ≥ B $$ (0,0) ∗ e by auto
  let ?f = λ i j. if i = 0 ∧ j = 0 then e else 0
  let ?C = mat n n (λ (i,j). ?f i j)
  have C: ?C ∈ carrier-mat n n by auto
  have C00: ?C $$ (0,0) = e using n-pos by auto
  show ?thesis
  proof(intro exI conjI)
    show ?C ≥m 0m n n
      by (rule mat-geI[of - n n], auto simp: ge-refl e0)
    show mat-arc-pos ?C
      unfolding mat-arc-posI-def
      unfolding C00 by (rule ae)
    let ?mult = B ∗ ?C
    from n-pos obtain nn where n: n = Suc nn by (cases n, auto)
    have col: col ?C 0 = vec n (?f 0) using n-pos by auto
    let ?prod = row B 0 · col ?C 0
    let ?head = B $$ (0,0) ∗ ?C $$ (0,0)
    let ?rest = ∑ i = 1..<n. B $$ (0, i) ∗ ?C $$ (i, 0)

    from n-pos B have ?mult $$ (0,0) = ?prod by auto
    also have ... = ?head + ?rest
      by (rule scalar-prod-split-head[OF B C n-pos])
    also have ?rest = 0
      by (rule sum.neutral, auto)
    finally have ?mult $$ (0,0) = B $$ (0,0) ∗ e using n-pos by simp
    with nc c-def have not-ge: ¬ A $$ (0,0) ≥ ?mult $$ (0,0) by simp
    show ¬ A ≥m ?mult
    proof
      assume A ≥m ?mult
      from mat-geD[OF this, of 0 0] A B not-ge n-pos show False by auto
```

    qed
  qed auto
qed

end

context *SN-both-mono-ordered-semiring-1*
begin

lemma *mat-gt-arc-SN*:
  assumes *n-pos*: $n > 0$
   shows *SN* $\{(A,B) \in$ *carrier-mat n n* $\times$ *carrier-mat n n*. *mat-arc-pos B* $\wedge$ *mat-gt-arc A B*$\}$
  (**is** *SN ?rel*)
proof (*rule ccontr*)
  assume $\neg$ *SN ?rel*
  then obtain *f A* **where** *f* (*0* :: *nat*) = *A* **and** *steps*: $\forall$ *i*. (*f i*, *f* (*Suc i*)) $\in$ *?rel* **unfolding** *SN-defs* **by** *blast*
  hence *pos*: $\forall$ *i*. *arc-pos* (*f* (*Suc i*) $$ (*0*,*0*)) **unfolding** *mat-arc-posI-def* **by** *blast*
  have *gt*: $\forall$ *i*. *f i* $$ (*0*,*0*) $\succ$ *f* (*Suc i*) $$ (*0*,*0*)
  proof
    fix *i*
    from *steps*
    have *wf1*: *f i* $\in$ *carrier-mat n n*
      and *wf2*: *f* (*Suc i*) $\in$ *carrier-mat n n*
      and *gt*: *mat-gt-arc* (*f i*) (*f* (*Suc i*)) **by** *auto*
    show *f i* $$ (*0*,*0*) $\succ$ *f* (*Suc i*) $$ (*0*,*0*)
      using *mat-comp-allE*[*OF gt wf1 wf2*]
      using *index-zero-mat n-pos* **by** *force*
  qed
  from *pos gt SN* **show** *False* **unfolding** *SN-defs* **by** *force*
qed

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*)
by (*unfold-locales*; *fact real-add real-mult real-zero real-one real-le*)

instantiation *real* :: *real-embedding*
begin
definition *real-of-real* :: *real* ⇒ *real* **where**
real-of-real $x = x$

instance
by (*intro-classes*, *auto simp*: *real-of-real-def*, *linarith*)
end

instantiation *int* :: *real-embedding*
begin

definition *real-of-int* :: *int* ⇒ *real* **where**
real-of-int $x = x$

instance
by (*intro-classes*, *auto simp*: *real-of-int-def*, *linarith*)
end

lemma *real-of-rat-ineq*: **assumes** *real-of-rat* $x \leq z$
shows $x \leq$ *of-int* $\lceil z \rceil$
proof −
have $z \leq$ *of-int* $\lceil z \rceil$ **by** *linarith*
from *order-trans*[*OF assms this*]
have *real-of-rat* $x \leq$ *real-of-rat* (*of-int* $\lceil z \rceil$) **by** *auto*
thus $x \leq$ *of-int* $\lceil z \rceil$ **using** *of-rat-less-eq* **by** *blast*
qed

instantiation *rat* :: *real-embedding*
begin
definition *real-of-rat* :: *rat* ⇒ *real* **where**
  real-of-rat x = *of-rat x*

instance
  by (*intro-classes*, *auto simp*: *real-of-rat-def of-rat-add of-rat-mult real-of-rat-ineq*)
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*
  by (*auto simp*: *deriv-bound-def*) (*metis lessI relpow-Suc-I*)

lemma *deriv-boundE*:
  assumes *deriv-bound r a n*
    and (⋀*b m*. *n* < *m* ⟹ (*a*, *b*) ∈ *r* ^^ *m* ⟹ *False*) ⟹ *P*
  shows *P*
  using *assms*(*1*)
  by (*intro assms*)
    (*auto simp*: *deriv-bound-def relpow-add relcomp.simps dest*!: *less-imp-Suc-add*, *metis relpow-E2*)

lemma *deriv-bound-iff*:
  deriv-bound r a n ⟷ (∀ *b m*. *n* < *m* ⟶ (*a*, *b*) ∉ *r* ^^ *m*)
  by (*auto elim*: *deriv-boundE intro*: *deriv-boundI*)

lemma *deriv-bound-empty* [*simp*]:
  deriv-bound {} *a n*

by (*simp add*: *deriv-bound-def*)

lemma *deriv-bound-mono*:
  assumes $m \leq n$ **and** *deriv-bound r a m*
  shows *deriv-bound r a n*
  using *assms* **by** (*auto simp*: *deriv-bound-iff*)

lemma *deriv-bound-image*:
  assumes *b*: *deriv-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
  fix *b m*
  assume $(a,\ b) \in r$ ^^ m
  from *relpow-image* [*OF step this*] **have** $(f\ a,\ f\ b) \in r'^{+}$ ^^ m **.**
  from *trancl-steps-relpow* [*OF subset-refl this*]
    obtain *k* **where** $k \geq m$ **and** $(f\ a,\ f\ b) \in r'$ ^^ k **by** *auto*
  moreover assume $n < m$
  moreover with *deriv-bound-mono* [*OF* - *b*, *of* $m - 1$]
    have *deriv-bound* r' $(f\ a)$ $(m - 1)$ **by** *simp*
  ultimately show *False* **using** *b* **by** (*simp add*: *deriv-bound-iff*)
qed

lemma *deriv-bound-subset*:
  assumes $r \subseteq r'^{+}$
    and *b*: *deriv-bound* r' *a n*
  shows *deriv-bound r a n*
  using *assms* **by** (*intro deriv-bound-image* [*of* - $\lambda x.\ x$, *OF b*]) *auto*

lemma *deriv-bound-SN-on*:
  assumes *deriv-bound r a n*
  shows *SN-on r* $\{a\}$
proof
  fix *f*
  assume *steps*: $\forall\ i.\ (f\ i,\ f\ (Suc\ i)) \in r$ **and** $f\ 0 \in \{a\}$
  with *assms* **have** $(f\ 0,\ f\ (Suc\ n)) \notin r$ ^^ *Suc n* **by** (*blast elim*: *deriv-boundE*)
  moreover have $(f\ 0,\ f\ (Suc\ n)) \in r$ ^^ *Suc n*
    using *steps* **unfolding** *relpow-fun-conv* **by** (*intro exI* [*of* - *f*]) *auto*
  ultimately show *False* **..**
qed

lemma *deriv-bound-steps*:
  assumes $(a,\ b) \in r$ ^^ n
    and *deriv-bound r a m*
  shows $n \leq m$
  using *assms* **by** (*auto iff*: *not-less deriv-bound-iff*)
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* **..**
instance *int* :: *large-real-ordered-semiring-1* **..**
instance *rat* :: *large-real-ordered-semiring-1* **..**

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* $\Rightarrow$ $'a$ $\Rightarrow$ *bool* (**infix** ‹≻› *50*) **and** *default* :: $'a$ +
  fixes *bound* :: $'a \Rightarrow$ *nat*
  assumes *bound-mono*: $\bigwedge$ *a b*. $a \geq b \Longrightarrow$ *bound a* $\geq$ *bound b*
   and *bound-plus*: $\bigwedge$ *a b*. *bound* $(a + b) \leq$ *bound a* + *bound b*
   and *bound-plus-of-nat*: $\bigwedge$ *a n*. $a \geq 0 \Longrightarrow$ *bound* (*a* + *of-nat n*) = *bound a* + *bound* (*of-nat n*)
   and *bound-zero*[*simp*]: *bound 0* = *0*
   and *bound-one*: *bound 1* $\geq$ *1*
   and *bound*: $\bigwedge$ *a*. *deriv-bound* {(*a*,*b*). $b \geq 0 \wedge a \succ b$} *a* (*bound a*)
begin

lemma *bound-linear*: $\exists$ *c*. $\forall$ *n*. *bound* (*of-nat n*) $\leq c * n$
proof (*rule exI*[*of - bound 1*], *intro allI*)
  fix *n*
  show *bound* (*of-nat n*) $\leq$ *bound 1* $*$ *n*
  proof (*induct n*)
    case (*Suc n*)
    have *bound* (*of-nat* (*Suc n*)) = *bound* (*1* + *of-nat n*) **by** *simp*
    also have ... $\leq$ *bound 1* + *bound* (*of-nat n*)
      by (*rule bound-plus*)
    also have ... $\leq$ *bound 1* + *bound 1* $*$ *n*
      using *Suc* **by** *auto*
    finally show *?case* **by** *auto*
  qed *simp*

qed

lemma *bound-of-nat-times*: *bound* (*of-nat n* $*$ *v*) $\leq$ *n* $*$ *bound v*
proof (*induct n*)
 case (*Suc n*)
 have *bound* (*of-nat* (*Suc n*) $*$ *v*) = *bound* (*v* + *of-nat n* $*$ *v*) **by** (*simp add*: *field-simps*)
 also have ... $\leq$ *bound v* + *bound* (*of-nat n* $*$ *v*) **by** (*rule bound-plus*)
 also have ... $\leq$ *bound v* + *n* $*$ *bound v* **using** *Suc* **by** *auto*
 finally show *?case* **by** *simp*
qed *simp*

lemma *bound-mult-of-nat*: *bound* (*a* $*$ *of-nat n*) $\leq$ *bound a* $*$ *bound* (*of-nat n*)
proof (*induct n*)
 case (*Suc n*)
 have *bound* (*a* $*$ *of-nat* (*Suc n*)) = *bound* (*a* + *a* $*$ *of-nat n*) **by** (*simp add*: *field-simps*)
 also have ... $\leq$ *bound a* + *bound* (*a* $*$ *of-nat n*)
  by (*rule bound-plus*)
 also have ... $\leq$ *bound a* + *bound a* $*$ *bound* (*of-nat n*) **using** *Suc* **by** *auto*
 also have ... = *bound a* $*$ (*1* + *bound* (*of-nat n*)) **by** (*simp add*: *field-simps*)
 also have ... $\leq$ *bound a* $*$ (*bound* (*1* + *of-nat n*))
 proof (*rule mult-le-mono2*)
  show *1* + *bound*(*of-nat n*) $\leq$ *bound* (*1* + *of-nat n*) **using** *bound-one*
  using *bound-plus*
   unfolding *bound-plus-of-nat*[*OF one-ge-zero*] **by** *simp*
 qed
 finally show *?case* **by** *simp*
qed *simp*

lemma *bound-pow-of-nat*: *bound* (*a* $*$ *of-nat n* ^ *deg*) $\leq$ *bound a* $*$ *of-nat n* ^ *deg*
proof (*induct deg*)
 case (*Suc deg*)
 have *bound* (*a* $*$ *of-nat n* ^ *Suc deg*) = *bound* (*of-nat n* $*$ (*a* $*$ *of-nat n* ^ *deg*))
  by (*simp add*: *field-simps*)
 also have ... $\leq$ *n* $*$ *bound* (*a* $*$ *of-nat n* ^ *deg*)
  by (*rule bound-of-nat-times*)
 also have ... $\leq$ *n* $*$ (*bound a* $*$ *of-nat n* ^ *deg*)
  using *Suc* **by** *auto*
 finally show *?case* **by** (*simp add*: *field-simps*)
qed *simp*
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*
  by (*cases a*) (*auto simp*: *shows-prec-arctic-def show-law-simps*)

definition *shows-list* (*as* :: *arctic list*) = *showsp-list shows-prec 0 as*

instance
  by *standard* (*simp-all add*: *shows-list-arctic-def show-law-simps*)

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*
  by (*cases a*) (*auto simp*: *shows-prec-arctic-delta-def show-law-simps*)

definition *shows-list* (*ps* :: *'a arctic-delta list*) = *showsp-list shows-prec 0 ps*

instance
  by *standard* (*simp-all add*: *shows-list-arctic-delta-def show-law-simps*)

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*: $\bigwedge$ *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: $\bigwedge$ *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 −
let *?n* = [*0* ..< *n*]
let *?m1x* = [*A* $$ (*i*,*j*) . *A* <− *map fst ABs*, *i* <− *?n*, *j* <− *?n*]
let *?m2y* = [*B* $$ (*i*,*j*) . *B* <− *map snd ABs*, *i* <− *?n*, *j* <− *?n*]
let *?pairs* = *concat* (*map* (λ *x*. *map* (λ *y*. (*x*,*y*)) *?m2y*) *?m1x*)

let *?strict* = *filter* (λ (*x*,*y*). *weak-gt x y*) *?pairs*
have ∀ *x y*. (*x*,*y*) ∈ *set ?strict* ⟶ *weak-gt x y* **by** *auto*
from *weak-gt-mono*[*OF this*] **obtain** *gt bound* **where** *order*: *mono-matrix-carrier gt default bound mono*
and *orient2*: ⋀ *x y*. (*x*, *y*) ∈ *set ?strict* ⟹ *gt x y* **by** *auto*
show *?thesis*
proof (*intro exI allI conjI impI*, *rule order*)
fix *A B*
assume *A*: *A* ∈ *carrier-mat n n* **and** *B*: *B* ∈ *carrier-mat n n*
and *AB*: (*A*, *B*) ∈ *set ABs*
from *orient*[*OF this*] **have** *mat-gt weak-gt sd A B* **by** *auto*
from *mat-gtD*[*OF this*] **obtain** *i j* **where**
ge: $A \geq_m B$ **and** *ij*: *i* < *sd j* < *sd* **and** *wgt*: *weak-gt* (*A* $$ (*i*,*j*)) (*B* $$ (*i*,*j*))
by *auto*
from *ij* ‹*sd* ≤ *n*› **have** *ij′*: *i* < *n j* < *n* **by** *auto*
have *gt*: *gt* (*A* $$ (*i*,*j*)) (*B* $$ (*i*,*j*))
by (*rule orient2*, *insert ij′ AB wgt*, *force*)
show *mat-gt gt sd A B* **using** *ij gt ge* **by** *auto*
qed
qed
end

sublocale *mono-matrix-carrier* ⊆ *SN-strict-mono-ordered-semiring-1 d gt mono*
proof
show *SN* {(*x*,*y*). *y* ≥ *0* ∧ *x* ≻ *y*}
unfolding *SN-def*
by (*intro allI deriv-bound-SN-on*[*OF bound*])
qed (*rule mono*)

sublocale *mono-matrix-carrier* ⊆ *matrix-carrier* **..**

27.2 The Integers as Carrier

lemma *int-complexity*:
mono-matrix-carrier ((>) :: *int* ⇒ *int* ⇒ *bool*) *1 nat int-mono*
proof (*unfold-locales*)
fix *x*
let *?R* = {(*x*, *y*). *0* ≤ (*y* :: *int*) ∧ *y* < *x*}
show *deriv-bound ?R x* (*nat x*)
unfolding *deriv-bound-def*
proof
assume (∃ *y*. (*x*,*y*) ∈ *?R* ^^ *Suc* (*nat x*))
then obtain *y* **where** *xy*: (*x*,*y*) ∈ *?R* ^^ *Suc* (*nat x*) **..**
from *xy* **have** *y*: *0* ≤ *y* **by** *auto*
obtain *n* **where** *n*: *n* = *Suc* (*nat x*) **by** *auto*
from *xy*[*unfolded n*[*symmetric*]]
have *x* ≥ *y* + *int n*
proof (*induct n arbitrary*: *x y*)
case *0* **thus** *?case* **by** *auto*

next
  case (*Suc n*)
  from *Suc*(*2*) **obtain** *z* **where** *xz*: (*x*,*z*) ∈ *?R* ^^ *n* **and** *zy*: (*z*,*y*) ∈ *?R*
    by *auto*
  from *Suc*(*1*)[*OF xz*] **have** *le*: *z* + *int n* ≤ *x* **.**
  from *zy* **have** *le2*: *y* + *1* ≤ *z* **by** *simp*
  with *le* **show** *?case* **by** *auto*
qed
with *y* **have** *nx*: *int n* ≤ *x* **by** *simp*
from *nx* **have** *x0*: *x* ≥ *0* **by** *simp*
with *nx n*
show *False* **by** *simp*
qed
qed (*insert int-SN.mono*, *auto*)

lemma *int-weak-complexity*:
weak-complexity-linear-poly-order-carrier (>) *1 int-mono*
by (*unfold-locales*, *intro exI*[*of - (>)*] *exI*[*of - nat*] *conjI*, *rule int-complexity*, *auto*)

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 −
from *d0* **have** *d00*: *0* ≤ *d* **by** *simp*
define *N* **where** *N* = *ceiling* (*1* / *d*)
let *?N* = *of-int N* :: *'a*
from *d0* **have** *1* / *d* > *0* **by** (*auto simp*: *field-simps*)
with *ceiling-correct*[*of 1* / *d*] **have** *Nd*: *1* / *d* ≤ *?N* **and** *N*: *N* > *0* **unfolding** *N-def* **by** *auto*
let *?nat* = λ *x*. *nat* (*ceiling* (*x* ∗ *?N*))
let *?gt* = *delta-gt d*
have *nn*: *delta-bound d* = *?nat* **unfolding** *fun-eq-iff N-def* **by** (*simp add*: *delta-bound-def*)
from *delta-interpretation*[*OF d0 d1*]
interpret *SN-strict-mono-ordered-semiring-1 def ?gt delta-mono* **.**
show *?thesis* **unfolding** *nn*
proof(*unfold-locales*)
  show *?nat 0* = *0* **by** *auto*
next
  fix *x y* :: *'a*
  assume *xy*: *x* ≥ *y*
  show *?nat x* ≥ *?nat y*
    by (*rule nat-mono*, *rule ceiling-mono*, *insert xy N*, *auto simp*: *field-simps*)

next
have $1 \leq nat\ 1$ **by** *simp*
also have ... $\leq$ *?nat 1*
proof (*rule nat-mono*)
have $1 = ceiling\ (1 :: rat)$ **by** *simp*
also have ... $\leq ceiling\ (1 * ?N)$ **using** N **by** *simp*
finally show $1 \leq ceiling\ (1 * ?N)$ **.**
qed
finally show $1 \leq ?nat\ 1$ **.**
next
fix $x\ y :: 'a$
have $ceiling\ ((x + y) * ?N) = ceiling\ (x * ?N + y * ?N)$ **by** (*simp add*: *field-simps*)
also have ... $\leq ceiling\ (x * ?N) + ceiling\ (y * ?N)$ **by** (*rule ceiling-add-le*)
finally show $?nat\ (x + y) \leq ?nat\ x + ?nat\ y$ **by** *auto*
next
fix $x :: 'a$ **and** $n :: nat$
assume x: $0 \leq x$
interpret *mono-matrix-carrier* ($>$) *1 nat int-mono* **by** (*rule int-complexity*)
have $?nat\ (x + of\text{-}nat\ n) = nat\ (ceiling\ (x * ?N + of\text{-}nat\ n * ?N))$
by (*simp add*: *field-simps*)
also have *id*: $of\text{-}nat\ n * ?N = of\text{-}int\ (of\text{-}nat\ (n * nat\ N))$ **using** N **by** (*simp add*: *field-simps*)
also have $ceiling\ (x * ?N + of\text{-}int\ (of\text{-}nat\ (n * nat\ N))) = ceiling\ (x * ?N) + of\text{-}nat\ (n * nat\ N)$ **unfolding** *ceiling-add-of-int* **..**
also have $nat\ (ceiling\ (x * ?N) + of\text{-}nat\ (n * nat\ N)) = ?nat\ x + nat\ (int\ (n * nat\ N))$
proof (*rule bound-plus-of-nat*)
have $x * ?N \geq 0$
by (*rule mult-nonneg-nonneg*, *insert* x N, *auto*)
thus $ceiling\ (x * ?N) \geq 0$ **by** *auto*
qed
also have $(nat\ (int\ (n * nat\ N))) = n * nat\ N$ **by** *presburger*
also have $n * nat\ N = ?nat\ (of\text{-}nat\ n)$ **using** N **by** (*metis id ceiling-of-int nat-int*)
finally
show $?nat\ (x + of\text{-}nat\ n) = ?nat\ x + ?nat\ (of\text{-}nat\ n)$ **.**
next
fix $x\ y\ z :: 'a$
assume $*$: *delta-mono x delta-gt d y z* **and** x: $0 \leq x$
from *mono*[*OF* $*$ x]
show *delta-gt d* $(x * y)\ (x * z)$ **.**
next
fix $x :: 'a$
let $?R = \{(x,y).\ 0 \leq y \wedge ?gt\ x\ y\}$
show *deriv-bound ?R x* (*?nat x*) **unfolding** *deriv-bound-def*
proof
assume $(\exists\ y.\ (x,y) \in ?R \ \widehat{}\widehat{}\ Suc\ (?nat\ x))$
then obtain y **where** *xy*: $(x,y) \in ?R \ \widehat{}\widehat{}\ Suc\ (?nat\ x)$ **..**

```
      from xy have y: 0 ≤ y by auto
      obtain n where n: n = Suc (?nat x) by auto
      from xy[unfolded n[symmetric]]
      have x ≥ y + d ∗ of-nat n
      proof (induct n arbitrary: x y)
        case 0 thus ?case by auto
      next
        case (Suc n)
        from Suc(2) obtain z where xz: (x,z) ∈ ?R ^^ n and zy: (z,y) ∈ ?R
          by auto
        from Suc(1)[OF xz] have le: z + d ∗ of-nat n ≤ x .
        from zy[unfolded delta-gt-def] have le2: y + d ≤ z by simp
        with le show ?case by (auto simp: field-simps)
      qed
      with y have nx: d ∗ of-nat n ≤ x by simp
      have 0 ≤ d ∗ of-nat n by (rule mult-nonneg-nonneg, insert d00, auto)
      with nx have x0: x ≥ 0 by auto
      have xd0: 0 ≤ x / d
        by (rule divide-nonneg-pos[OF x0 d0])
      from nx[unfolded n]
      have d + d ∗ of-nat (?nat x) ≤ x by (simp add: field-simps)
      with d0 have less: d ∗ of-nat (?nat x) < x by simp
      from Nd d0 have 1 ≤ d ∗ ?N by (auto simp: field-simps)
      from mult-left-mono[OF this x0]
      have x ≤ d ∗ (x ∗ ?N) by (simp add: ac-simps)
      also have ... ≤ d ∗ of-nat (?nat x)
      proof (rule mult-left-mono[OF - d00])
        show x ∗ ?N ≤ of-nat (nat ⌈x ∗ ?N⌉) using x0 ceiling-correct[of x ∗ ?N]
          by (metis int-nat-eq le-cases of-int-0-le-iff of-int-of-nat-eq order-trans)
      qed
      also have ... < x using less .
      finally show False by simp
    qed
  qed
qed


lemma delta-weak-complexity-carrier:
  assumes d0: def > 0
  shows weak-complexity-linear-poly-order-carrier (>) def delta-mono
proof
  fix xys :: ('a × 'a) list
  assume ass: ∀ x y. (x, y) ∈ set xys ⟶ y < x
  let ?cs = map (λ (x,y). x − y) xys
  let ?ds = def # ?cs
  define d where d = Min (set ?ds)
  have d: d ≤ def and dcs: ⋀ x. x ∈ set ?cs ⟹ d ≤ x unfolding d-def by auto
  have d ∈ set ?ds unfolding d-def by (rule Min-in, auto)
  hence d = def ∨ d ∈ set ?cs by auto
```

hence *d0*: *d* > *0*
by (*cases*, *insert d0 ass*, *auto simp*: *field-simps*)
show ∃ *gt bound*. *mono-matrix-carrier gt def bound delta-mono* ∧ (∀ *x y*. (*x*, *y*) ∈ *set xys* ⟶ *gt x y*)
by (*intro exI conjI*, *rule delta-complexity*[*OF d0 d*], *insert dcs*, *force simp*: *delta-gt-def*)
qed

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 **..**

lemma *arctic-weak-carrier*:
weak-SN-both-mono-ordered-semiring-1 (>) *1 pos-arctic*
proof −
have *SN*: *SN-both-mono-ordered-semiring-1 1* (>) *pos-arctic* **..**
show *?thesis*
by (*unfold-locales*, *intro conjI exI*, *rule SN*, *auto*)
qed

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* $\Rightarrow$ $'a$ *vec set* **where**
mat-kernel $A = \{\ v\ .\ v \in$ *carrier-vec* (*dim-col* A) $\wedge$ $A *_v v = 0_v$ (*dim-row* A)$\}$

lemma *mat-kernelI*: **assumes** $A \in$ *carrier-mat nr nc* $v \in$ *carrier-vec nc* $A *_v v = 0_v$ *nr*
shows $v \in$ *mat-kernel* A
using *assms* **unfolding** *mat-kernel-def* **by** *auto*

lemma *mat-kernelD*: **assumes** $A \in$ *carrier-mat nr nc* $v \in$ *mat-kernel* A
shows $v \in$ *carrier-vec nc* $A *_v v = 0_v$ *nr*
using *assms* **unfolding** *mat-kernel-def* **by** *auto*

lemma *mat-kernel*: **assumes** $A \in$ *carrier-mat nr nc*
shows *mat-kernel* $A = \{v.\ v \in$ *carrier-vec nc* $\wedge$ $A *_v v = 0_v$ *nr*$\}$
unfolding *mat-kernel-def* **using** *assms* **by** *auto*

lemma *mat-kernel-carrier*:
assumes $A \in$ *carrier-mat nr nc* **shows** *mat-kernel* $A \subseteq$ *carrier-vec nc*
using *assms mat-kernel* **by** *auto*

lemma *mat-kernel-mult-subset*: **assumes** A: $A \in$ *carrier-mat nr nc*
and B: $B \in$ *carrier-mat n nr*
shows *mat-kernel* $A \subseteq$ *mat-kernel* ($B * A$)
proof −
from A B **have** BA: $B * A \in$ *carrier-mat n nc* **by** *auto*
show *?thesis* **unfolding** *mat-kernel*[*OF BA*] *mat-kernel*[*OF A*] **using** A B **by** *auto*
qed

lemma *mat-kernel-smult*: **assumes** A: $A \in$ *carrier-mat nr nc*
and v: $v \in$ *mat-kernel* A
shows $a \cdot_v v \in$ *mat-kernel* A
proof −
from *mat-kernelD*[*OF A v*] **have** v: $v \in$ *carrier-vec nc*
and z: $A *_v v = 0_v$ *nr* **by** *auto*
from *arg-cong*[*OF z*, *of* $\lambda\ v.\ a \cdot_v v$] v
have $a \cdot_v (A *_v v) = 0_v$ *nr* **by** *auto*
also have $a \cdot_v (A *_v v) = A *_v (a \cdot_v v)$ **using** A v **by** *auto*
finally show *?thesis* **using** v A
by (*intro mat-kernelI*, *auto*)
qed

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
from *B A* **have** *BA*: $B * A \in$ *carrier-mat nr nc* **by** *auto*
show *mat-kernel A* $\subseteq$ *mat-kernel* $(B * A)$ **by** (*rule mat-kernel-mult-subset*[*OF A B*])
{
fix *v*
assume *v*: $v \in$ *mat-kernel* $(B * A)$
from *mat-kernelD*[*OF BA this*] **have** *v*: $v \in$ *carrier-vec nc* **and** *z*: $B * A *_v v = 0_v$ *nr* **by** *auto*
from *arg-cong*[*OF z*, *of* λ *v*. $C *_v v$]
have $C *_v (B * A *_v v) = 0_v$ *nr* **using** *C v* **by** *auto*
also have $C *_v (B * A *_v v) = ((C * B) * A) *_v v$
unfolding *assoc-mult-mat-vec*[*symmetric*, *OF C BA v*]
unfolding *assoc-mult-mat*[*OF C B A*] **by** *simp*
also have ... $= A *_v v$ **unfolding** *inv* **using** *A v* **by** *auto*
finally have $v \in$ *mat-kernel A*
by (*intro mat-kernelI*[*OF A v*])
}
thus *mat-kernel* $(B * A) \subseteq$ *mat-kernel A* **by** *auto*
qed

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* **.**

abbreviation *VK* $\equiv$ *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*)
apply (*intro vectorspace.intro*)
apply (*rule NC.submodule-is-module*)
apply (*unfold-locales*)
by (*insert A mult-add-distrib-mat-vec*[*OF A*] *mult-mat-vec*[*OF A*] *mat-kernel*[*OF A*], *auto simp*: *class-ring-simps*)

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*
using *assms*
proof (*induct S rule*: *infinite-finite-induct*)
case (*insert s S*)
hence *base*: *finite S s* ∉ *S*
and *f-VK*: *f* : *S* → *mat-kernel A f s* : *mat-kernel A* **by** *auto*
hence *f-NC*: *f* : *S* → *carrier-vec nc f s* : *carrier-vec nc* **using** *mat-kernel*[*OF A*] **by** *auto*
have *IH*: *finsum VK f S* = *finsum NC.V f S* **using** *insert f-VK* **by** *auto*
thus *?case*
unfolding *NC.M.finsum-insert*[*OF base f-NC*]
unfolding *Ker.finsum-insert*[*OF base f-VK*]
by *simp*
qed *auto*

lemma *lincomb-same*:
assumes *S-kernel*: *S* ⊆ *mat-kernel A*
shows *lincomb a S* = *NC.lincomb a S*
unfolding *Ker.lincomb-def*
unfolding *NC.lincomb-def*
apply(*subst finsum-same*)
using *S-kernel Ker.smult-closed*[*unfolded module-vec-simps class-ring-simps*] **by** *auto*

lemma *span-same*:
assumes *S-kernel*: *S* ⊆ *mat-kernel A*
shows *span S* = *NC.span S*
proof (*rule*;*rule*)
fix *v* **assume** *L*: *v* : *span S* **show** *v* : *NC.span S*

proof −
obtain *a U* **where** *know*: *finite U U* ⊆ *S a* : *U* → *UNIV v* = *lincomb a U*
using *L* **unfolding** *Ker.span-def* **by** *auto*
hence *v*: *v* = *NC.lincomb a U* **using** *lincomb-same S-kernel* **by** *auto*
show *?thesis*
unfolding *NC.span-def* **by** (*rule*,*intro exI conjI*;*fact*)
qed
next fix *v* **assume** *R*: *v* : *NC.span S* **show** *v* : *span S*
proof −
obtain *a U* **where** *know*: *finite U U* ⊆ *S v* = *NC.lincomb a U*
using *R* **unfolding** *NC.span-def* **by** *auto*
hence *v*: *v* = *lincomb a U* **using** *lincomb-same S-kernel* **by** *auto*
show *?thesis* **unfolding** *Ker.span-def* **by** (*rule*, *intro exI conjI*, *insert v know*, *auto*)
qed
qed

lemma *lindep-same*:
assumes *S-kernel*: *S* ⊆ *mat-kernel A*
shows *Ker.lin-dep S* = *NC.lin-dep S*
proof
note [*simp*] = *module-vec-simps class-ring-simps*
{ assume *L*: *Ker.lin-dep S*
then obtain *v a U*
where *finU*: *finite U* **and** *US*: *U* ⊆ *S*
and *lc*: *lincomb a U* = 0_v *nc*
and *vU*: *v* ∈ *U*
and *av0*: *a v* ≠ *0*
unfolding *Ker.lin-dep-def* **by** *auto*
have *lc'*: *NC.lincomb a U* = 0_v *nc*
using *lc lincomb-same US S-kernel* **by** *auto*
show *NC.lin-dep S* **unfolding** *NC.lin-dep-def*
by (*intro exI conjI*, *insert finU US lc' vU av0*, *auto*)
}
assume *R*: *NC.lin-dep S*
then obtain *v a U*
where *finU*: *finite U* **and** *US*: *U* ⊆ *S*
and *lc*: *NC.lincomb a U* = 0_v *nc*
and *vU*: *v* : *U*
and *av0*: *a v* ≠ *0*
unfolding *NC.lin-dep-def* **by** *auto*
have *lc'*: *lincomb a U* = *zero VK*
using *lc lincomb-same US S-kernel* **by** *auto*
show *Ker.lin-dep S* **unfolding** *Ker.lin-dep-def*
by (*intro exI conjI*,*insert finU US lc' vU av0*, *auto*)
qed

lemma *lincomb-index*:
assumes *i*: *i* < *nc*

and *Xk*: $X \subseteq$ *mat-kernel A*
shows *lincomb a X* \$ *i* = *sum* (*λx. a x ∗ x* \$ *i*) *X*
proof −
have *X*: $X \subseteq$ *carrier-vec nc* **using** *Xk mat-kernel-def A* **by** *auto*
show *?thesis*
using *vec-space.lincomb-index*[*OF i X*]
using *lincomb-same*[*OF Xk*] **by** *auto*
qed

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* ∧ *row A i* ≠ 0_v *nc*}
length (*pivot-positions A*) = *card* { *i. i* < *nr* ∧ *row A i* ≠ 0_v *nc*}
kernel.dim nc A = *nc* − *card* { *i. i* < *nr* ∧ *row A i* ≠ 0_v *nc*}
proof −
note *non-pivot-base* = *non-pivot-base*[*OF ref A*]
let *?B* = *set* (*find-base-vectors A*)
let *?pp* = *set* (*pivot-positions A*)
from *A* **have** *dim*: *dim-row A* = *nr dim-col A* = *nc* **by** *auto*
from *ref*[*unfolded row-echelon-form-def*] **obtain** *p*
where *pivot*: *pivot-fun A p nc* **using** *dim* **by** *auto*
note *piv* = *pivot-funD*[*OF dim*(*1*) *pivot*]
{
fix *v*
assume *v* ∈ *?B*
from *this*[*unfolded find-base-vectors-def Let-def dim*]
obtain *c* **where** *c*: *c* < *nc c* ∉ *snd* ‘ *?pp*
and *res*: *v* = *non-pivot-base A* (*pivot-positions A*) *c* **by** *auto*
from *non-pivot-base*[*OF c*, *folded res*] *c*
have *v* ∈ *mat-kernel A v* ≠ 0_v *nc*
by (*intro mat-kernelI*[*OF A*], *auto*)
}
thus *sub*: *?B* $\subseteq$ *mat-kernel A* **and**
0_v *nc* ∉ *?B* **by** *auto*
{
fix *j j′*
assume *j*: *j* < *nc j* ∉ *snd* ‘ *?pp* **and** *j′*: *j′* < *nc j′* ∉ *snd* ‘ *?pp* **and** *neq*: *j′* ≠ *j*
from *non-pivot-base*(*2*)[*OF j*] *non-pivot-base*(*4*)[*OF j′ j neq*]
have *non-pivot-base A* (*pivot-positions A*) *j* ≠ *non-pivot-base A* (*pivot-positions A*) *j′* **by** *auto*
}
hence *inj*: *inj-on* (*non-pivot-base A* (*pivot-positions A*))
(*set* [*j*←[*0..*<*nc*] . *j* ∉ *snd* ‘ *?pp*]) **unfolding** *inj-on-def* **by** *auto*
note *pp* = *pivot-positions*[*OF A pivot*]

have *lc*: *length* (*pivot-positions A*) = *card* (*snd* ‘ *?pp*)
using *distinct-card*[*OF pp*(*3*)] **by** *auto*
show *card*: *card ?B* = *nc* − *card* { *i*. *i* < *nr* ∧ *row A i* ≠ 0_v *nc*}
length (*pivot-positions A*) = *card* { *i*. *i* < *nr* ∧ *row A i* ≠ 0_v *nc*}
unfolding *find-base-vectors-def Let-def dim set-map card-image*[*OF inj*] *pp*(*4*)[*symmetric*]
unfolding *pp*(*1*) *lc*
proof −
have *nc* − *card* (*snd* ‘ {(*i*, *p i*) |*i*. *i* < *nr* ∧ *p i* ≠ *nc*})
= *card* {*0* ..< *nc*} − *card* (*snd* ‘ {(*i*, *p i*) |*i*. *i* < *nr* ∧ *p i* ≠ *nc*}) **by** *auto*
also have . . . = *card* ({*0* ..< *nc*} − *snd* ‘ {(*i*, *p i*) |*i*. *i* < *nr* ∧ *p i* ≠ *nc*})
by (*rule card-Diff-subset*[*symmetric*], *insert piv*(*1*), *force*+)
also have {*0* ..< *nc*} − *snd* ‘ {(*i*, *p i*) |*i*. *i* < *nr* ∧ *p i* ≠ *nc*} = (*set* [*j*←[*0*..<*nc*]
. *j* ∉ *snd* ‘ {(*i*, *p i*) |*i*. *i* < *nr* ∧ *p i* ≠ *nc*}])
by *auto*
finally show *card* (*set* [*j*←[*0*..<*nc*] . *j* ∉ *snd* ‘ {(*i*, *p i*) |*i*. *i* < *nr* ∧ *p i* ≠ *nc*}])
=
nc − *card* (*snd* ‘ {(*i*, *p i*) |*i*. *i* < *nr* ∧ *p i* ≠ *nc*}) **by** *simp*
qed *auto*
interpret *kernel nr nc A* **by** (*unfold-locales*, *rule A*)
show *basis*: *basis ?B*
unfolding *Ker*.*basis-def*
proof (*intro conjI*)
show *span ?B* = *mat-kernel A*
proof
show *span ?B* ⊆ *mat-kernel A*
using *sub* **by** (*rule Ker*.*span-is-subset2*)
show *mat-kernel A* ⊆ *Ker*.*span ?B*
proof
fix *v*
assume *v* ∈ *mat-kernel A*
from *mat-kernelD*[*OF A this*]
have *v*: *v* ∈ *carrier-vec nc* **and** *Av*: *A* $*_v$ *v* = 0_v *nr* **by** *auto*
let *?bi* = *non-pivot-base A* (*pivot-positions A*)
let *?ran* = *set* [*j*←[*0*..<*nc*] . *j* ∉ *snd* ‘ *?pp*]
let *?ran′* = *set* [*j*←[*0*..<*nc*] . *j* ∈ *snd* ‘ *?pp*]
have *dimv*: *dim-vec v* = *nc* **using** *v* **by** *auto*
define *I* **where** *I* = (λ *b*. *SOME i*. *i* ∈ *?ran* ∧ *?bi i* = *b*)
{
fix *j*
assume *j*: *j* ∈ *?ran*
hence ∃ *i*. *i* ∈ *?ran* ∧ *?bi i* = *?bi j* **unfolding** *find-base-vectors-def Let-def*
dim **by** *auto*
from *someI-ex*[*OF this*] **have** *I*: *I* (*?bi j*) ∈ *?ran* **and** *id*: *?bi* (*I* (*?bi j*))
= *?bi j* **unfolding** *I-def* **by** *blast*+
from *inj-onD*[*OF inj id I j*] **have** *I* (*?bi j*) = *j* **.**
} **note** *I* = *this*
define *a* **where** *a* = (λ *b*. *v* $ (*I b*))
from *Ker*.*lincomb-closed*[*OF sub*] **have** *diml*: *dim-vec* (*lincomb a ?B*) = *nc*
unfolding *mat-kernel-def* **using** *dim lincomb-same* **by** *auto*

have $v = lincomb\ a\ ?B$
proof (*rule eq-vecI*; *unfold diml dimv*)
fix j
assume j: $j < nc$
have $Ker.lincomb\ a\ ?B\ \$\ j = (\sum b\in\ ?B.\ a\ b * b\ \$\ j)$ **by** (*rule lincomb-index*[*OF j sub*])
also have $\ldots = (\sum\ i\in\ ?ran.\ v\ \$\ i * ?bi\ i\ \$\ j)$
proof (*subst sum.reindex-cong*[*OF inj*])
show $?B = ?bi$ ‘ $?ran$ **unfolding** *find-base-vectors-def Let-def dim* **by** *auto*
fix i
assume $i \in\ ?ran$
hence $I\ (?bi\ i) = i$ **by** (*rule I*)
hence $a\ (?bi\ i) = v\ \$\ i$ **unfolding** *a-def* **by** *simp*
thus $a\ (?bi\ i) * ?bi\ i\ \$\ j = v\ \$\ i * ?bi\ i\ \$\ j$ **by** *simp*
qed *auto*
also have $\ldots = v\ \$\ j$
proof (*cases* $j \in\ ?ran$)
case *True*
hence *nmem*: $j \notin snd$ ‘ *set* (*pivot-positions A*) **by** *auto*
note *npb* = *non-pivot-base*[*OF j nmem*]
have $(\sum\ i\in\ ?ran.\ v\ \$\ i * (?bi\ i)\ \$\ j) =$
$v\ \$\ j * ?bi\ j\ \$\ j + (\sum\ i\in\ ?ran - \{j\}.\ v\ \$\ i * ?bi\ i\ \$\ j)$
by (*subst sum.remove*[*OF - True*], *auto*)
also have $?bi\ j\ \$\ j = 1$ **using** *npb* **by** *simp*
also have $(\sum\ i \in\ ?ran - \{j\}.\ v\ \$\ i * ?bi\ i\ \$\ j) = 0$
using *insert non-pivot-base*(*4*)[*OF - - j nmem*] **by** (*intro sum.neutral*, *auto*)
finally show *?thesis* **by** *simp*
next
case *False*
with j **have** *jpp*: $j \in snd$ ‘ $?pp$ **by** *auto*
with j *pp* **obtain** i **where** i: $i < nr$ **and** *ji*: $j = p\ i$ **and** *pi*: $p\ i < nc$ **by** *auto*
from *arg-cong*[*OF Av*, *of* $\lambda\ u.\ u\ \$\ i$] i A
have $v\ \$\ j = v\ \$\ j - row\ A\ i \cdot v$ **by** *auto*
also have $row\ A\ i \cdot v = (\sum\ j = 0\ ..< nc.\ A\ \$\$\ (i,j) * v\ \$\ j)$ **unfolding** *scalar-prod-def* **using** v A i **by** *auto*
also have $\ldots = (\sum\ j \in\ ?ran.\ A\ \$\$\ (i,j) * v\ \$\ j) + (\sum\ j \in\ ?ran'.\ A\ \$\$\ (i,j) * v\ \$\ j)$
by (*subst sum.union-disjoint*[*symmetric*], *auto intro*: *sum.cong*)
also have $(\sum\ j \in\ ?ran'.\ A\ \$\$\ (i,j) * v\ \$\ j) =$
$A\ \$\$\ (i,p\ i) * v\ \$\ j + (\sum\ j \in\ ?ran' - \{p\ i\}.\ A\ \$\$\ (i,j) * v\ \$\ j)$
using *jpp* **by** (*subst sum.remove*, *auto simp*: *ji i pi*)
also have $A\ \$\$\ (i,\ p\ i) = 1$ **using** *piv*(*4*)[*OF i*] *pi ji* **by** *auto*
also have $(\sum\ j \in\ ?ran' - \{p\ i\}.\ A\ \$\$\ (i,j) * v\ \$\ j) = 0$
proof (*rule sum.neutral*, *intro ballI*)
fix j'
assume $j' \in\ ?ran' - \{p\ i\}$

then obtain i' **where** i': $i' < nr$ **and** j': $j' = p\ i'$ **and** pi': $p\ i' \neq nc$ **and** *neq*: $p\ i' \neq p\ i$
unfolding *pp* **by** *auto*
from pi' *piv*[*OF* i'] **have** pi': $p\ i' < nc$ **by** *auto*
from *pp* pi' *neq* $j\ i'\ i$ **have** $i \neq i'$ **by** *auto*
from *piv*(5)[*OF* i' pi' *i this*]
show $A\ \$\$\ (i,j') * v\ \$\ j' = 0$ **unfolding** j' **by** *simp*
qed
also have $(\sum j \in ?ran.\ A\ \$\$\ (i,j) * v\ \$\ j) = - (\sum j \in ?ran.\ v\ \$\ j * - A\ \$\$\ (i,j))$
unfolding *sum-negf*[*symmetric*] **by** (*rule sum.cong*, *auto*)
finally have *vj*: $v\ \$\ j = (\sum j \in ?ran.\ v\ \$\ j * - A\ \$\$\ (i,j))$ **by** *simp*
show *?thesis* **unfolding** *vj j*
proof (*rule sum.cong*[*OF refl*])
fix j'
assume j': $j' \in ?ran$
from *jpp* j' **have** jj': $j \neq j'$ **by** *auto*
let *?map* = *map prod.swap* (*pivot-positions A*)
from *ji i j* **have** $(i,j) \in$ *set* (*pivot-positions A*) **unfolding** *pp* **by** *auto*
hence *mem*: $(j,i) \in$ *set ?map* **by** *auto*
from *pp* **have** *distinct* (*map fst ?map*) **unfolding** *map-map o-def prod.swap-def fst-conv* **by** *auto*
from *map-of-is-SomeI*[*OF this mem*] **have** *map-of ?map j = Some i* **by** *auto*
hence $?bi\ j'\ \$\ j = - A\ \$\$\ (i, j')$
unfolding *non-pivot-base-def Let-def dim* **using** $j\ jj'$ **by** *auto*
thus $v\ \$\ j' * ?bi\ j'\ \$\ j = v\ \$\ j' * - A\ \$\$\ (i,j')$ **by** *simp*
qed
qed
finally show $v\ \$\ j =$ *lincomb a ?B* $\$\ j$ **..**
qed *auto*
thus $v \in$ *span ?B* **unfolding** *Ker.span-def* **by** *auto*
qed
qed
show $?B \subseteq$ *mat-kernel A* **by** (*rule sub*)
{
fix *a v*
assume *lc*: *lincomb a ?B* $= 0_v\ nc$ **and** *vB*: $v \in ?B$
from *vB*[*unfolded find-base-vectors-def Let-def dim*]
obtain *j* **where** *j*: $j < nc\ j \notin snd$ ‘ *?pp* **and** *v*: $v =$ *non-pivot-base A* (*pivot-positions A*) *j*
by *auto*
from *arg-cong*[*OF lc*, *of* $\lambda\ v.\ v\ \$\ j$] *j*
have $0 =$ *lincomb a ?B* $\$\ j$ **by** *auto*
also have $\ldots = (\sum v \in ?B.\ a\ v * v\ \$\ j)$
by (*subst lincomb-index*[*OF j*(1) *sub*], *simp*)
also have $\ldots = a\ v * v\ \$\ j + (\sum w \in ?B - \{v\}.\ a\ w * w\ \$\ j)$
by (*subst sum.remove*[*OF - vB*], *auto*)
also have $a\ v * v\ \$\ j = a\ v$ **using** *non-pivot-base*[*OF j*, *folded v*] **by** *simp*

also have ($\sum w{\in}?B - \{v\}.\ a\ w * w\ \$\ j) = 0$
proof (*rule sum.neutral*, *intro ballI*)
fix *w*
assume *wB*: $w \in ?B - \{v\}$
from *this*[*unfolded find-base-vectors-def Let-def dim*]
obtain j' **where** j': $j' < nc\ j' \notin snd$ ' *?pp* **and** *w*: *w* = *non-pivot-base A* (*pivot-positions A*) j'
by *auto*
with *wB v* **have** $j' \neq j$ **by** *auto*
from *non-pivot-base*(*4*)[*OF* j' *j this*]
show $a\ w * w\ \$\ j = 0$ **unfolding** *w* **by** *simp*
qed
finally have $a\ v = 0$ **by** *simp*
}
thus ¬ *lin-dep ?B*
by (*intro Ker.finite-lin-indpt2*[*OF finite-set sub*], *auto simp*: *class-field-def*)
qed
show *dim* = *nc* − *card* { *i*. $i < nr \wedge$ *row A i* $\neq 0_v$ *nc*}
using *Ker.dim-basis*[*OF finite-set basis*] *card* **by** *simp*
qed

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* **unfolding** *kernel-dim-def*
using *A* **by** *simp*

lemma *kernel-dim-code*[*code*]:
kernel-dim A = *dim-col A* − *length* (*pivot-positions* (*gauss-jordan-single A*))
proof −
define *nr* **where** *nr* = *dim-row A*
define *nc* **where** *nc* = *dim-col A*
let *?B* = *gauss-jordan-single A*
have *A*: $A \in$ *carrier-mat nr nc* **unfolding** *nr-def nc-def* **by** *auto*
from *gauss-jordan-single*[*OF A refl*]
obtain *P Q* **where** *AB*: *?B* = *P* ∗ *A* **and** *QP*: *Q* ∗ *P* = 1_m *nr* **and**
P: $P \in$ *carrier-mat nr nr* **and** *Q*: $Q \in$ *carrier-mat nr nr* **and** *B*: *?B* ∈ *carrier-mat nr nc*
and *row*: *row-echelon-form ?B* **by** *auto*
interpret *K*: *kernel nr nc ?B*
by (*unfold-locales*, *rule B*)
from *mat-kernel-mult-eq*[*OF A P Q QP*, *folded AB*]
have *kernel-dim A* = *K.dim* **unfolding** *kernel-dim-def* **using** *A* **by** *simp*
also have . . . = *nc* − *length* (*pivot-positions ?B*) **using** *find-base-vectors*[*OF row B*] **by** *auto*
also have . . . = *dim-col A* − *length* (*pivot-positions ?B*)
unfolding *nc-def* **by** *simp*
finally show *?thesis* **.**

qed

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 −
  have *Ac*: $A \in$ *carrier-mat* $n\ n$ **unfolding** A **by** *auto*
  have *pivot-fun* A *id* n
    unfolding A **by** (*rule pivot-funI*, *auto*)
  hence *row*: *row-echelon-form* A **unfolding** *row-echelon-form-def* A **by** *auto*
  have $\{i.\ i < n \wedge$ *row* $A\ i \neq 0_v\ n\} = \{0\ ..< n\}$ **unfolding** A **by** *auto*
  hence *id*: *card* $\{i.\ i < n \wedge$ *row* $A\ i \neq 0_v\ n\} = n$ **by** *auto*
  interpret *kernel* $n\ n\ A$ **by** (*unfold-locales*, *rule Ac*)
  from *find-base-vectors*[*OF row Ac*, *unfolded id*]
  show *dim* $= 0$ *basis* $\{\}$ **by** *auto*
qed

lemma *kernel-upper-triangular*: **assumes** A: $A \in$ *carrier-mat* $n\ n$
  and *ut*: *upper-triangular* A **and** 0: $0 \notin$ *set* (*diag-mat* A)
  shows *kernel.dim* $n\ A = 0$ *kernel.basis* $n\ A\ \{\}$
proof −
  define *ma* **where** *ma* = *diag-mat* A
  from *det-upper-triangular*[*OF ut A*] **have** *det* A = *prod-list* (*diag-mat* A) **.**
  also have $\ldots \neq 0$ **using** 0 **unfolding** *ma-def*[*symmetric*]
    by (*induct ma*, *auto*)
  finally have *det* $A \neq 0$ **.**
  from *det-non-zero-imp-unit*[*OF A this*, *unfolded Units-def*, *of* ()]
   obtain B **where** B: $B \in$ *carrier-mat* $n\ n$ **and** *BA*: $B * A = 1_m\ n$ **and** *AB*: A
$* B = 1_m\ n$
    by (*auto simp*: *ring-mat-def*)
  from *mat-kernel-mult-eq*[*OF A B A AB*, *unfolded BA*]
  have *id*: *mat-kernel* A = *mat-kernel* ($1_m\ n$) **..**
  show *kernel.dim* $n\ A = 0$ *kernel.basis* $n\ A\ \{\}$
    unfolding *id* **by** (*rule kernel-one-mat*)+
qed

lemma *kernel-basis-exists*: **assumes** A: $A \in$ *carrier-mat nr nc*
  shows $\exists\ B.$ *finite* $B \wedge$ *kernel.basis nc* $A\ B$
proof −
  obtain C **where** *gj*: *gauss-jordan-single* $A = C$ **by** *auto*
  from *gauss-jordan-single*[*OF A gj*]
  obtain $P\ Q$ **where** *CPA*: $C = P * A$ **and** *QP*: $Q * P = 1_m$ *nr*
    and P: $P \in$ *carrier-mat nr nr* **and** Q: $Q \in$ *carrier-mat nr nr*
    and C: $C \in$ *carrier-mat nr nc* **and** *row*: *row-echelon-form* C
    by *auto*
  from *find-base-vectors*[*OF row C*] **have** $\exists\ B.$ *finite* $B \wedge$ *kernel.basis nc* $C\ B$ **by**

blast
  also have *mat-kernel C* = *mat-kernel A* **unfolding** *CPA*
    by (*rule mat-kernel-mult-eq*[*OF A P Q QP*])
  finally show *?thesis* **.**
qed

lemma *mat-kernel-mult-right-gen-set*: **assumes** *A*: $A \in$ *carrier-mat nr nc*
  and *B*: $B \in$ *carrier-mat nc nc*
  and *C*: $C \in$ *carrier-mat nc nc*
  and *inv*: $B * C = 1_m$ *nc*
  and *gen-set*: *kernel.gen-set nc* $(A * B)$ *gen* **and** *gen*: *gen* $\subseteq$ *mat-kernel* $(A * B)$
  shows *kernel.gen-set nc A* $(((*_v)\ B)$ ' *gen*$)$ $(*_v)$ *B* ' *gen* $\subseteq$ *mat-kernel A card* $(((*_v)\ B)$ ' *gen*$)$ = *card gen*
proof −
  let *?AB* = $A * B$
  let *?gen* = $((*_v)\ B)$ ' *gen*
  from *A B* **have** *AB*: $A * B \in$ *carrier-mat nr nc* **by** *auto*
  from *B* **have** *dimB*: *dim-row B* = *nc* **by** *auto*
  from *inv B C* **have** *CB*: $C * B = 1_m$ *nc* **by** (*metis mat-mult-left-right-inverse*)
  interpret *AB*: *kernel nr nc ?AB*
    by (*unfold-locales*, *rule AB*)
  interpret *A*: *kernel nr nc A*
    by (*unfold-locales*, *rule A*)
  {
    fix *w*
    assume $w \in$ *?gen*
    then obtain *v* **where** *w*: $w = B *_v v$ **and** *v*: $v \in$ *gen* **by** *auto*
    from *v* **have** $v \in$ *mat-kernel ?AB* **using** *gen* **by** *auto*
    hence *v*: $v \in$ *carrier-vec nc* **and** *0*: *?AB* $*_v\ v = 0_v$ *nr* **unfolding** *mat-kernel*[*OF AB*] **by** *auto*
    have *?AB* $*_v\ v = A *_v w$ **unfolding** *w* **using** *v A B* **by** *simp*
    with *0* **have** *0*: $A *_v w = 0_v$ *nr* **by** *auto*
    from *w B v* **have** *w*: $w \in$ *carrier-vec nc* **by** *auto*
    from *0 w* **have** $w \in$ *mat-kernel A* **unfolding** *mat-kernel*[*OF A*] **by** *auto*
  }
  thus *genn*: *?gen* $\subseteq$ *mat-kernel A* **by** *auto*
  hence *one-dir*: *A.span ?gen* $\subseteq$ *mat-kernel A* **by** *fastforce*
  {
    fix *v v'*
    assume *v*: $v \in$ *gen* **and** *v'*: $v' \in$ *gen* **and** *id*: $B *_v v = B *_v v'$
    from *v v'* **have** *v*: $v \in$ *carrier-vec nc* **and** *v'*: $v' \in$ *carrier-vec nc*
      using *gen* **unfolding** *mat-kernel*[*OF AB*] **by** *auto*
    from *arg-cong*[*OF id*, *of* λ *v*. $C *_v v$]
    have $v = v'$ **using** *v v'*
      unfolding *assoc-mult-mat-vec*[*symmetric*, *OF C B v*]
        assoc-mult-mat-vec[*symmetric*, *OF C B v'*] *CB*
      by *auto*
  } note *inj* = *this*

hence *inj-gen*: *inj-on* $((*_v)\ B)$ *gen* **unfolding** *inj-on-def* **by** *auto*
show *card ?gen = card gen* **using** *inj-gen* **by** (*rule card-image*)
{
fix *v*
let *?Cv* $= C *_v v$
assume $v \in$ *mat-kernel A*
from *mat-kernelD*[*OF A this*] **have** *v*: $v \in$ *carrier-vec nc* **and** *0*: $A *_v v = 0_v$ *nr* **by** *auto*
have *?AB* $*_v$ *?Cv* $= (A * (B * C)) *_v v$ **using** *A B C v*
by (*subst assoc-mult-mat-vec*[*symmetric*, *OF AB C v*], *subst assoc-mult-mat*[*OF A B C*], *simp*)
also have ... $= 0_v$ *nr* **unfolding** *inv* **using** *0 A v* **by** *simp*
finally have *0*: *?AB* $*_v$ *?Cv* $= 0_v$ *nr* **and** *Cv*: *?Cv* $\in$ *carrier-vec nc* **using** *C v* **by** *auto*
hence *?Cv* $\in$ *mat-kernel ?AB* **unfolding** *mat-kernel*[*OF AB*] **by** *auto*
with *gen-set* **have** *?Cv* $\in$ *AB.span gen* **by** *auto*
from *this*[*unfolded AB.Ker.span-def*] **obtain** *a gen'* **where**
Cv: *?Cv = AB.lincomb a gen'* **and** *sub*: *gen'* $\subseteq$ *gen* **and** *fin*: *finite gen'* **by** *auto*
let *?gen'* $= ((*_v)\ B)$ ‘ *gen'*
from *sub gen* **have** *gen'*: *gen'* $\subseteq$ *mat-kernel ?AB* **by** *auto*
have *lin1*: *AB.lincomb a gen'* $\in$ *carrier-vec nc*
using *AB.Ker.lincomb-closed*[*OF gen'*, *of a*]
unfolding *mat-kernel*[*OF AB*] **by** (*auto simp*: *class-field-def*)
hence *dim1*: *dim-vec* (*AB.lincomb a gen'*) = *nc* **by** *auto*
hence *dim1b*: *dim-vec* $(B *_v$ (*AB.Ker.lincomb a gen'*)) = *nc* **using** *B* **by** *auto*
from *genn sub* **have** *genn'*: *?gen'* $\subseteq$ *mat-kernel A* **by** *auto*
from *gen sub* **have** *gen'nc*: *gen'* $\subseteq$ *carrier-vec nc* **unfolding** *mat-kernel*[*OF AB*] **by** *auto*
define *a'* **where** $a' = (\lambda\ b.\ a\ (C *_v b))$
from *A.Ker.lincomb-closed*[*OF genn'*]
have *lin2*: *A.Ker.lincomb a' ?gen'* $\in$ *carrier-vec nc*
unfolding *mat-kernel*[*OF A*] **by** (*auto simp*: *class-field-def*)
hence *dim2*: *dim-vec* (*A.Ker.lincomb a' ?gen'*) = *nc* **by** *auto*
have $v = B *_v$ *?Cv*
by (*unfold assoc-mult-mat-vec*[*symmetric*, *OF B C v*] *inv*, *insert v*, *simp*)
hence $v = B *_v$ *AB.Ker.lincomb a gen'* **unfolding** *Cv* **by** *simp*
also have ... = *A.Ker.lincomb a' ?gen'*
proof (*rule eq-vecI*; *unfold dim1 dim1b dim2*)
fix *i*
assume *i*: $i < nc$
with *dimB* **have** *ii*: $i <$ *dim-row B* **by** *auto*
from *sub inj* **have** *inj*: *inj-on* $((*_v)\ B)$ *gen'* **unfolding** *inj-on-def* **by** *auto*
{
fix *v*
assume $v \in$ *gen'*
with *gen'nc* **have** *v*: $v \in$ *carrier-vec nc* **by** *auto*
hence $a'\ (B *_v v) = a\ v$ **unfolding** *a'-def assoc-mult-mat-vec*[*symmetric*, *OF C B v*] *CB* **by** *auto*

} **note** *a′ = this*
have *A.Ker.lincomb a′ ?gen′* $ *i* = (∑ *v*∈(∗$_v$) *B* ‘ *gen′*. *a′ v* ∗ *v* $ *i*)
unfolding *A.lincomb-index*[*OF i genn′*] **by** *simp*
also have . . . = (∑ *v*∈*gen′*. *a v* ∗ ((*B* ∗$_v$ *v*) $ *i*))
by (*rule sum.reindex-cong*[*OF inj refl*], *auto simp*: *a′*)
also have . . . = (∑ *v*∈*gen′*. (∑ *j* = *0*..< *nc*. *a v* ∗ *row B i* $ *j* ∗ *v* $ *j*))
unfolding *mult-mat-vec-def dimB scalar-prod-def index-vec*[*OF i*]
by (*rule sum.cong*, *insert gen′nc*, *auto simp*: *sum-distrib-left ac-simps*)
also have . . . = (∑ *j* = *0* ..< *nc*. (∑ *v* ∈ *gen′*. *a v* ∗ *row B i* $ *j* ∗ *v* $ *j*))
by (*rule sum.swap*)
also have . . . = (∑ *j* = *0*..<*nc*. *row B i* $ *j* ∗ (∑ *v*∈*gen′*. *a v* ∗ *v* $ *j*))
by (*rule sum.cong*, *auto simp*: *sum-distrib-left ac-simps*)
also have . . . = (*B* ∗$_v$ *AB.Ker.lincomb a gen′*) $ *i*
unfolding *index-mult-mat-vec*[*OF ii*]
unfolding *scalar-prod-def dim1*
by (*rule sum.cong*[*OF refl*], *subst AB.lincomb-index*[*OF - gen′*], *auto*)
finally show (*B* ∗$_v$ *AB.Ker.lincomb a gen′*) $ *i* = *A.Ker.lincomb a′ ?gen′* $ *i* **..**
qed *auto*
finally have *v* ∈ *A.Ker.span ?gen* **using** *sub fin*
unfolding *A.Ker.span-def* **by** (*auto simp*: *class-field-def intro*!: *exI*[*of - a′*] *exI*[*of - ?gen′*])
}
hence *other-dir*: *A.Ker.span ?gen* ⊇ *mat-kernel A* **by** *fastforce*
from *one-dir other-dir* **show** *kernel.gen-set nc A* (((∗$_v$) *B*) ‘ *gen*) **by** *auto*
qed

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 −
let *?AB* = *A* ∗ *B*
let *?gen* = ((∗$_v$) *B*) ‘ *gen*
from *A B* **have** *AB*: *?AB* ∈ *carrier-mat nr nc* **by** *auto*
from *B* **have** *dimB*: *dim-row B* = *nc* **by** *auto*
from *inv B C* **have** *CB*: *C* ∗ *B* = *1*$_m$ *nc* **by** (*metis mat-mult-left-right-inverse*)
interpret *AB*: *kernel nr nc ?AB*
by (*unfold-locales*, *rule AB*)
interpret *A*: *kernel nr nc A*
by (*unfold-locales*, *rule A*)
from *basis*[*unfolded AB.Ker.basis-def*] **have** *gen-set*: *AB.gen-set gen* **and** *genAB*: *gen* ⊆ *mat-kernel ?AB* **by** *auto*
from *mat-kernel-mult-right-gen-set*[*OF A B C inv gen-set genAB*]
have *gen*: *A.gen-set ?gen* **and** *sub*: *?gen* ⊆ *mat-kernel A* **and** *card*: *card ?gen* =

card gen **.**
  from *card* **show** *card ?gen = card gen* **.**
  from *fin* **have** *fing*: *finite ?gen* **by** *auto*
  from *gen* **have** *gen*: *A.Ker.span ?gen = mat-kernel A* **by** *auto*
  have *ABC*: $A * B * C = A$ **using** *A B C inv* **by** *simp*
  from *kernel-basis-exists*[*OF A*] **obtain** *bas* **where** *finb*: *finite bas* **and** *bas*: *A.basis bas* **by** *auto*
  from *bas* **have** *bas'*: *A.gen-set bas bas* $\subseteq$ *mat-kernel A* **unfolding** *A.Ker.basis-def* **by** *auto*
  let *?bas* = $(*_v)$ *C* ‘ *bas*
  from *mat-kernel-mult-right-gen-set*[*OF AB C B CB*, *unfolded ABC*, *OF bas'*]
  have *bas'*: *?bas* $\subseteq$ *mat-kernel ?AB AB.Ker.span ?bas = mat-kernel ?AB card ?bas = card bas* **by** *auto*
  from *finb bas* **have** *cardb*: *A.dim = card bas* **by** (*rule A.Ker.dim-basis*)
  from *fin basis* **have** *cardg*: *AB.dim = card gen* **by** (*rule AB.Ker.dim-basis*)
  from *AB.Ker.gen-ge-dim*[*OF - bas'*(*1−2*)] *finb bas'*(*3*) *cardb cardg*
  have *ineq1*: *card gen* $\leq$ *A.dim* **by** *auto*
  from *A.Ker.dim-gen-is-basis*[*OF fing sub gen*, *unfolded card*, *OF this*]
  show *A.basis ?gen* **.**
qed

lemma *mat-kernel-dim-mult-eq-right*: **assumes** *A*: $A \in$ *carrier-mat nr nc*
  and *B*: $B \in$ *carrier-mat nc nc*
  and *C*: $C \in$ *carrier-mat nc nc*
  and *BC*: $B * C = 1_m$ *nc*
  shows *kernel.dim nc* $(A * B)$ = *kernel.dim nc A*
proof −
  let *?AB* = $A * B$
  from *A B* **have** *AB*: *?AB* $\in$ *carrier-mat nr nc* **by** *auto*
  interpret *AB*: *kernel nr nc ?AB*
    by (*unfold-locales*, *rule AB*)
  interpret *A*: *kernel nr nc A*
    by (*unfold-locales*, *rule A*)
  from *kernel-basis-exists*[*OF AB*] **obtain** *bas* **where** *finb*: *finite bas* **and** *bas*: *AB.basis bas* **by** *auto*
  let *?bas* = $((*_v)$ *B*) ‘ *bas*
  from *mat-kernel-mult-right-basis*[*OF A B C BC finb bas*] *finb*
  have *bas'*: *A.basis ?bas* **and** *finb'*: *finite ?bas* **and** *card*: *card ?bas = card bas* **by** *auto*
  show *AB.dim = A.dim* **unfolding** *A.Ker.dim-basis*[*OF finb' bas'*] *AB.Ker.dim-basis*[*OF finb bas*] *card* **..**
qed

locale *vardim* =
  fixes *f-ty* :: *'a* :: *field itself*
begin

abbreviation $M == \lambda k.$ *module-vec TYPE*$('a)$ k

abbreviation *span* $== \lambda k.$ *LinearCombinations.module.span class-ring* $(M\ k)$
abbreviation *lincomb* $== \lambda k.$ *module.lincomb* $(M\ k)$
abbreviation *lin-dep* $== \lambda k.$ *module.lin-dep class-ring* $(M\ k)$
abbreviation *padr* $m\ v == v\ @_v\ 0_v\ m$
definition *unpadr* $m\ v ==$ *vec* (*dim-vec* $v - m$) $(\lambda i.\ v\ \$\ i)$
abbreviation *padl* $m\ v == 0_v\ m\ @_v\ v$
definition *unpadl* $m\ v ==$ *vec* (*dim-vec* $v - m$) $(\lambda i.\ v\ \$\ (m{+}i))$

lemma *unpadr-padr*[*simp*]: *unpadr* m (*padr* $m\ v$) $= v$ **unfolding** *unpadr-def* **by** *auto*
lemma *unpadl-padl*[*simp*]: *unpadl* m (*padl* $m\ v$) $= v$ **unfolding** *unpadl-def* **by** *auto*

lemma *padr-unpadr*[*simp*]: v : *padr* m ‘ $U \Longrightarrow$ *padr* m (*unpadr* $m\ v$) $= v$ **by** *auto*
lemma *padl-unpadl*[*simp*]: v : *padl* m ‘ $U \Longrightarrow$ *padl* m (*unpadl* $m\ v$) $= v$ **by** *auto*

lemma *padr-image*:
  assumes $U \subseteq$ *carrier-vec* n **shows** *padr* m ‘ $U \subseteq$ *carrier-vec* $(n + m)$
proof(*rule subsetI*)
  fix v **assume** v : *padr* m ‘ U
  then obtain u **where** $u : U$ **and** *vmu*: $v =$ *padr* $m\ u$ **by** *auto*
  hence u : *carrier-vec* n **using** *assms* **by** *auto*
  thus v : *carrier-vec* $(n + m)$
    unfolding *vmu*
    using *zero-carrier-vec*[*of m*] *append-carrier-vec* **by** *metis*
qed
lemma *padl-image*:
  assumes $U \subseteq$ *carrier-vec* n **shows** *padl* m ‘ $U \subseteq$ *carrier-vec* $(m + n)$
proof(*rule subsetI*)
  fix v **assume** v : *padl* m ‘ U
  then obtain u **where** $u : U$ **and** *vmu*: $v =$ *padl* $m\ u$ **by** *auto*
  hence u : *carrier-vec* n **using** *assms* **by** *auto*
  thus v : *carrier-vec* $(m + n)$
    unfolding *vmu*
    using *zero-carrier-vec*[*of m*] *append-carrier-vec* **by** *metis*
qed

lemma *padr-inj*:
  shows *inj-on* (*padr* m) (*carrier-vec* n :: $'a$ *vec set*)
  apply(*intro inj-onI*) **using** *append-vec-eq* **by** *auto*

lemma *padl-inj*:
  shows *inj-on* (*padl* m) (*carrier-vec* n :: $'a$ *vec set*)
  apply(*intro inj-onI*)
  using *append-vec-eq*[*OF zero-carrier-vec zero-carrier-vec*] **by** *auto*

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 −
  interpret *N*: *vectorspace class-ring M n* **using** *vec-vs*.
  interpret *NM*: *vectorspace class-ring M* (*n*+*m*) **using** *vec-vs*.
  note [*simp*] = *module-vec-simps class-ring-simps*
  have *?R* ∧ *?L* **using** *finU U*
  proof (*induct set*:*finite*)
    case *empty* **thus** *?case*
      unfolding *goal-def* **unfolding** *N.lincomb-def NM.lincomb-def* **by** *auto* **next**
    case (*insert u U*)
      hence *finU*: *finite U*
        and *U*: *U* ⊆ *carrier-vec n*
        and *u*[*simp*]: *u* : *carrier-vec n*
        and *uU*: *u* ∉ *U*
        and *auU*: *a* : *insert u U* → *UNIV*
        and *aU*: *a* : *U* → *UNIV*
        and *au*: *a u* : *UNIV*
        by *auto*
      have *IHr*: *goal padr unpadr U* **and** *IHl*: *goal padl unpadl U*
        using *insert*(*3*) *U aU* **by** *auto*
      note *N-lci* = *N.lincomb-insert2*[*unfolded module-vec-simps*]
      note *NM-lci* = *NM.lincomb-insert2*[*unfolded module-vec-simps*]
      have *auu*[*simp*]: *a u* $\cdot_v$ *u* : *carrier-vec n* **using** *au u* **by** *simp*
      have *laU*[*simp*]: *lincomb n a U* : *carrier-vec n*
        using *N.lincomb-closed*[*unfolded module-vec-simps class-ring-simps*, *OF U aU*].
      let *?m0* = 0_v *m* :: *'a vec*
      have *m0*: *?m0* : *carrier-vec m* **by** *auto*
      have *ins*: *lincomb n a* (*insert u U*) = *a u* $\cdot_v$ *u* + *lincomb n a U*
        using *N-lci*[*OF finU U*] *auU uU u* **by** *auto*
      show *?case*
      proof
        have *padr m* (*a u* $\cdot_v$ *u* + *lincomb n a U*) =
          (*a u* $\cdot_v$ *u* + *lincomb n a U*) $@_v$ (*?m0* + *?m0*) **by** *auto*
        also have ... = *padr m* (*a u* $\cdot_v$ *u*) + *padr m* (*lincomb n a U*)
          using *append-vec-add*[*symmetric*, *OF auu laU*]
          using *zero-carrier-vec*[*of m*] **by** *metis*
        also have *padr m* (*lincomb n a U*) = *lincomb* (*n*+*m*) (*a o unpadr m*) (*padr m* ‘ *U*)
          using *IHr* **unfolding** *goal-def*.
        also have *padr m* (*a u* $\cdot_v$ *u*) = *a u* $\cdot_v$ *padr m u* **by** *auto*
        also have ... = (*a o unpadr m*) (*padr m u*) $\cdot_v$ *padr m u* **by** *auto*
        also have ... + *lincomb* (*n*+*m*) (*a o unpadr m*) (*padr m* ‘ *U*) =

lincomb (*n*+*m*) (*a o unpadr m*) (*insert* (*padr m u*) (*padr m ' U*))
apply(*subst NM-lci*[*symmetric*])
using *finU uU U append-vec-eq*[*OF u*] **by** *auto*
also have *insert* (*padr m u*) (*padr m ' U*) = *padr m ' insert u U*
by *auto*
finally show *goal padr unpadr* (*insert u U*) **unfolding** *goal-def ins*.
have [*simp*]: *n*+*m* = *m*+*n* **by** *auto*
have *padl m* (*a u* $\cdot_v$ *u* + *lincomb n a U*) =
(*?m0* + *?m0*) $@_v$ (*a u* $\cdot_v$ *u* + *lincomb n a U*) **by** *auto*
also have ... = *padl m* (*a u* $\cdot_v$ *u*) + *padl m* (*lincomb n a U*)
using *append-vec-add*[*symmetric*, *OF - - auu laU*]
using *zero-carrier-vec*[*of m*] **by** *metis*
also have *padl m* (*lincomb n a U*) = *lincomb* (*n*+*m*) (*a o unpadl m*) (*padl m ' U*)
using *IHl* **unfolding** *goal-def*.
also have *padl m* (*a u* $\cdot_v$ *u*) = *a u* $\cdot_v$ *padl m u* **by** *auto*
also have ... = (*a o unpadl m*) (*padl m u*) $\cdot_v$ *padl m u* **by** *auto*
also have ... + *lincomb* (*n*+*m*) (*a o unpadl m*) (*padl m ' U*) =
lincomb (*n*+*m*) (*a o unpadl m*) (*insert* (*padl m u*) (*padl m ' U*))
apply(*subst NM-lci*[*symmetric*])
using *finU uU U append-vec-eq*[*OF m0*] **by** *auto*
also have *insert* (*padl m u*) (*padl m ' U*) = *padl m ' insert u U*
by *auto*
finally show *goal padl unpadl* (*insert u U*) **unfolding** *goal-def ins*.
qed
qed
thus *?R ?L* **by** *auto*
qed

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 −
interpret *N*: *vectorspace class-ring M n* **using** *vec-vs*.
interpret *NM*: *vectorspace class-ring M* (*n*+*m*) **using** *vec-vs*.
{ fix *pad* :: '*a vec* ⇒ '*a vec* **and** *unpad* :: '*a vec* ⇒ '*a vec*
assume *main*: ⋀*A a*. *A* ⊆ *U* ⟹ *finite A* ⟹
pad (*lincomb n a A*) = *lincomb* (*n*+*m*) (*a o unpad*) (*pad ' A*)
assume [*simp*]: ⋀*v*. *unpad* (*pad v*) = *v*
assume *pU*: *pad ' U* ⊆ *carrier-vec* (*n*+*m*)
have *pad '* (*span n U*) = *span* (*n*+*m*) (*pad ' U*)
proof (*intro Set.equalityI subsetI*)
fix *x* **assume** *x* : *pad '* (*span n U*)
then obtain *v* **where** *v* : *span n U* **and** *xv*: *x* = *pad v* **by** *auto*
then obtain *a A*
where *AU*: *A* ⊆ *U* **and** *finA*: *finite A* **and** *a*: *a* : *A* → *UNIV*
and *vaA*: *v* = *lincomb n a A*
unfolding *N.span-def* **by** *auto*

hence A: $A \subseteq$ *carrier-vec n* **using** U **by** *auto*
show x : *span* $(n+m)$ (*pad* ‘ U) **unfolding** *NM.span-def*
proof (*intro CollectI exI conjI*)
show x = *lincomb* $(n+m)$ (a *o unpad*) (*pad* ‘ A)
using *xv vaA main*[*OF AU finA*] **by** *auto*
show *pad* ‘ $A \subseteq$ *pad* ‘ U **using** *AU* **by** *auto*
qed (*insert finA*, *auto simp*: *class-ring-simps*)
next
fix x **assume** x : *span* $(n+m)$ (*pad* ‘ U)
then obtain a' A'
where $A'U$: $A' \subseteq$ *pad* ‘ U **and** *finA'*: *finite* A' **and** a': a' : $A' \to$ *UNIV*
and $xa'A'$: x = *lincomb* $(n+m)$ a' A'
unfolding *NM.span-def* **by** *auto*
then obtain A **where** *finA*: *finite* A **and** *AU*: $A \subseteq U$ **and** $A'A$: A' = *pad* ‘
A
using *finite-subset-image*[*OF finA' A'U*] **by** *auto*
hence A: $A \subseteq$ *carrier-vec n* **using** U **by** *auto*
have A': $A' \subseteq$ *carrier-vec* $(n+m)$ **using** $A'U$ *pU* **by** *auto*
define a **where** $a = a'$ *o pad*
define a'' **where** a'' = (a' *o pad*) *o unpad*
have a: a : $A \to$ *UNIV* **by** *auto*
have *restr*: *restrict* a' A' = *restrict* a'' A'
proof(*rule restrict-ext*)
fix u' **assume** u' : A'
then obtain u **where** u : A **and** u' = *pad* u **unfolding** $A'A$ **by** *auto*
thus a' u' = a'' u' **unfolding** *a''-def a-def* **by** *auto*
qed
have x = *lincomb* $(n+m)$ a' A' **using** $xa'A'$ **unfolding** $A'A$**.**
also have ... = *lincomb* $(n+m)$ a'' A'
apply (*subst NM.lincomb-restrict*)
using *finA' A' restr* **by** (*auto simp*: *module-vec-simps class-ring-simps*)
also have ... = *lincomb* $(n+m)$ a'' (*pad* ‘ A) **unfolding** $A'A$**..**
also have ... = *pad* (*lincomb n a A*)
unfolding *a''-def* **using** *main*[*OF AU finA*] **unfolding** *a-def* **by** *auto*
finally show x : *pad* ‘ (*span n U*) **unfolding** *N.span-def*
apply(*rule image-eqI*, *intro CollectI exI conjI*)
using *finA AU* **by** (*auto simp*: *class-ring-simps*)
qed
}
note *main* = *this*
have *AUC*: $\bigwedge A.\ A \subseteq U \Longrightarrow A \subseteq$ *carrier-vec n* **using** U **by** *simp*
have [*simp*]: $n+m = m+n$ **by** *auto*
show *goal padr m* **unfolding** *goal-def*
apply (*subst main*[*OF - - padr-image*[*OF U*]])
using *lincomb-pad*[*OF AUC*] *unpadr-padr* **by** *auto*
show *goal padl m* **unfolding** *goal-def*
apply (*subst main*)
using *lincomb-pad*[*OF AUC*] *unpadl-padl padl-image*[*OF U*] **by** *auto*
qed

```
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)
  unfolding mat-kernel-def
proof (rule, intro conjI)
  have [simp]: dim-row A = nr1 dim-row D = nr2 dim-row ?ABCD = nr1 + nr2
using A D by auto
  have a: a : carrier-vec nc1 using mat-kernel-carrier[OF A] aA by auto
  show ?ABCD *_v padr nc2 a = 0_v (dim-row ?ABCD) (is ?l = ?r)
  proof
    fix i assume i: i < dim-vec ?r
    hence ?l $ i = row ?ABCD i · padr nc2 a by auto
    also have ... = 0
    proof (cases i < nr1)
      case True
        hence rows: row A i : carrier-vec nc1 row B i : carrier-vec nc2
          using A B by auto
        have row ?ABCD i = row A i @_v row B i
          using row-four-block-mat(1)[OF A B - D True] by auto
        also have ... · padr nc2 a = row A i · a + row B i · 0_v nc2
          using scalar-prod-append[OF rows] a by auto
        also have row A i · a = (A *_v a) $ i using True A by auto
        also have ... = 0 using mat-kernelD[OF A aA] True by auto
        also have row B i · 0_v nc2 = 0 using True rows by auto
        finally show ?thesis by simp
      next case False
        let ?C = 0_m nr2 nc1
        let ?i = i − nr1
        have rows:
            row ?C ?i : carrier-vec nc1 row D ?i : carrier-vec nc2
          using D i False A by auto
        have row ?ABCD i = row ?C ?i @_v row D ?i
          using row-four-block-mat(2)[OF A B - D False] i A D by auto
        also have ... · padr nc2 a = row ?C ?i · a + row D ?i · 0_v nc2
          using scalar-prod-append[OF rows] a by auto
        also have row ?C ?i · a = 0_v nc1 · a using False A i by auto
        also have ... = 0 using a by auto
        also have row D ?i · 0_v nc2 = 0 using False rows by auto
        finally show ?thesis by simp
    qed
    finally show ?l $ i = ?r $ i using i by auto
  qed auto
  show padr nc2 a : carrier-vec (dim-col ?ABCD) using a A D by auto
qed
```

```
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 (0m nr1 nc2) C D) (is - ∈
mat-kernel ?ABCD)
  unfolding mat-kernel-def
proof (rule, intro conjI)
  have [simp]: dim-row A = nr1 dim-row D = nr2 dim-row ?ABCD = nr1 + nr2
using A D by auto
  have d: d : carrier-vec nc2 using mat-kernel-carrier[OF D] dD by auto
  show ?ABCD *v padl nc1 d = 0v (dim-row ?ABCD) (is ?l = ?r)
  proof
    fix i assume i: i < dim-vec ?r
    hence ?l $ i = row ?ABCD i · padl nc1 d by auto
    also have ... = 0
    proof (cases i < nr1)
      case True
        let ?B = 0m nr1 nc2
        have rows: row A i : carrier-vec nc1 row ?B i : carrier-vec nc2
          using A True by auto
        have row ?ABCD i = row A i @v row ?B i
          using row-four-block-mat(1)[OF A - C D True] by auto
        also have ... · padl nc1 d = row A i · 0v nc1 + row ?B i · d
          using scalar-prod-append[OF rows] d by auto
        also have row A i · 0v nc1 = 0 using A True by auto
        also have row ?B i · d = 0 using True d by auto
        finally show ?thesis by simp
      next case False
        let ?i = i − nr1
        have rows:
            row C ?i : carrier-vec nc1 row D ?i : carrier-vec nc2
          using C D i False A by auto
        have row ?ABCD i = row C ?i @v row D ?i
          using row-four-block-mat(2)[OF A - C D False] i A D by auto
        also have ... · padl nc1 d = row C ?i · 0v nc1 + row D ?i · d
          using scalar-prod-append[OF rows] d by auto
        also have row C ?i · 0v nc1 = 0 using False A C i by auto
        also have row D ?i · d = (D *v d) $ ?i using D d False i by auto
        also have ... = 0 using mat-kernelD[OF D dD] using False i by auto
        finally show ?thesis by simp
    qed
    finally show ?l $ i = ?r $ i using i by auto
  qed auto
  show padl nc1 d : carrier-vec (dim-col ?ABCD) using d A D by auto
qed
```

lemma *mat-kernel-split*:
  assumes *A*: $A \in$ *carrier-mat n n*
    and *D*: $D \in$ *carrier-mat m m*
    and *kAD*: $k \in$ *mat-kernel* (*four-block-mat A* (0_m *n m*) (0_m *m n*) *D*)
      (**is** - $\in$ *mat-kernel ?A00D*)
  shows *vec-first k n* $\in$ *mat-kernel A* (**is** *?a* $\in$ -)
   and *vec-last k m* $\in$ *mat-kernel D* (**is** *?d* $\in$ -)
proof −
  have 0_v *n* $@_v$ 0_v *m* = 0_v (*n*+*m*) **by** *auto*
  also
   have *A00D*: *?A00D* : *carrier-mat* (*n*+*m*) (*n*+*m*) **using** *four-block-carrier-mat*[*OF A D*]**.**
   hence *k*: *k* : *carrier-vec* (*n*+*m*) **using** *kAD mat-kernel-carrier* **by** *auto*
   hence *?a* $@_v$ *?d* = *k* **by** *simp*
   hence 0_v (*n*+*m*) = *?A00D* $*_v$ (*?a* $@_v$ *?d*) **using** *mat-kernelD*[*OF A00D*] *kAD* **by** *auto*
  also have ... = *A* $*_v$ *?a* $@_v$ *D* $*_v$ *?d*
   using *mult-mat-vec-split*[*OF A D*] **by** *auto*
  finally have 0_v *n* $@_v$ 0_v *m* = *A* $*_v$ *?a* $@_v$ *D* $*_v$ *?d***.**
  hence 0_v *n* = *A* $*_v$ *?a* $\wedge$ 0_v *m* = *D* $*_v$ *?d*
   apply(*subst append-vec-eq*[*of* - *n*, *symmetric*]) **using** *A D* **by** *auto*
  thus *?a* : *mat-kernel A ?d* : *mat-kernel D* **unfolding** *mat-kernel-def* **using** *A D* **by** *auto*
qed

lemma *padr-padl-eq*:
  assumes *v*: *v* : *carrier-vec n*
  shows *padr m v* = *padl n u* $\longleftrightarrow$ *v* = 0_v *n* $\wedge$ *u* = 0_v *m*
  apply (*subst append-vec-eq*) **using** *v* **by** *auto*

lemma *pad-disjoint*:
  assumes *A*: $A \subseteq$ *carrier-vec n* **and** *A0*: 0_v $n \notin A$ **and** *B*: $B \subseteq$ *carrier-vec m*
  shows *padr m* ‘ *A* $\cap$ *padl n* ‘ *B* = {} (**is** *?A* $\cap$ *?B* = -)
proof (*intro equals0I*)
  fix *ab* **assume** *ab* : *?A* $\cap$ *?B*
  then obtain *a b*
   where *ab* = *padr m a ab* = *padl n b* **and** *dim*: *a* : *A b* : *B* **by** *force*
  hence *padr m a* = *padl n b* **by** *auto*
  hence *a* = 0_v *n* **using** *dim A B* **by** *auto*
  thus *False* **using** *dim A0* **by** *auto*
qed

lemma *padr-padl-lindep*:
  assumes *A*: $A \subseteq$ *carrier-vec n* **and** *liA*: ~ *lin-dep n A*
    and *B*: $B \subseteq$ *carrier-vec m* **and** *liB*: ~ *lin-dep m B*
  shows ~ *lin-dep* (*n*+*m*) (*padr m* ‘ *A* $\cup$ *padl n* ‘ *B*) (**is** ~ *lin-dep* - (*?A* $\cup$ *?B*))
proof −
  interpret *N*: *vectorspace class-ring M n* **using** *vec-vs***.**

interpret *M*: *vectorspace class-ring M m* **using** *vec-vs*.
interpret *NM*: *vectorspace class-ring M* $(n+m)$ **using** *vec-vs*.
note [*simp*] = *module-vec-simps class-ring-simps*
have *AB*: *?A* ∪ *?B* ⊆ *carrier-vec* $(n+m)$
  using *padr-image*[*OF A*] *padl-image*[*OF B*] **by** *auto*
show *?thesis*
  unfolding *NM.lin-dep-def*
  unfolding *not-ex not-imp*[*symmetric*] *not-not*
proof(*intro allI impI*)
  fix *U f u*
  assume *finU*: *finite U*
    and *UAB*: *U* ⊆ *?A* ∪ *?B*
    and *f*: *f* : *U* → *carrier class-ring*
    and *0*: *lincomb* $(n+m)$ *f U* = $\mathbf{0}_{M\ (n+m)}$
    and *uU*: *u* : *U*
  let *?UA* = *U* ∩ *?A* **and** *?UB* = *U* ∩ *?B*
  have *?UA* ⊆ *?A* *?UB* ⊆ *?B* **by** *auto*
  then obtain *A′ B′*
   where *A′A*: *A′* ⊆ *A* **and** *B′B*: *B′* ⊆ *B*
    and *UAA′*: *?UA* = *padr m* ‘ *A′* **and** *UBB′*: *?UB* = *padl n* ‘ *B′*
   unfolding *subset-image-iff* **by** *auto*
  hence *A′*: *A′* ⊆ *carrier-vec n* **and** *B′*: *B′* ⊆ *carrier-vec m* **using** *A B* **by** *auto*
  have *finA′*: *finite A′* **and** *finB′*: *finite B′*
  proof −
   have *padr m* ‘ *A′* ⊆ *U* *padl n* ‘ *B′* ⊆ *U* **using** *UAA′ UBB′* **by** *auto*
   hence *pre*: *finite* (*padr m* ‘ *A′*) *finite* (*padl n* ‘ *B′*)
    using *finite-subset*[*OF - finU*] **by** *auto*
   show *finite A′*
    apply (*rule finite-imageD*) **using** *subset-inj-on*[*OF padr-inj A′*] *pre* **by** *auto*
   show *finite B′*
    apply (*rule finite-imageD*) **using** *subset-inj-on*[*OF padl-inj B′*] *pre* **by** *auto*
  qed
 have 0_v *n* ∉ *A* **using** *N.zero-nin-lin-indpt*[*OF - liA*] *A class-semiring.one-zeroI*
by *auto*
  hence *?A* ∩ *?B* = {} **using** *pad-disjoint A B* **by** *auto*
  hence *disj*: *?UA* ∩ *?UB* = {} **by** *auto*
  have *split*: *U* = *padr m* ‘ *A′* ∪ *padl n* ‘ *B′*
   unfolding *UAA′*[*symmetric*] *UBB′*[*symmetric*] **using** *UAB* **by** *auto*
  show *f u* = $\mathbf{0}_{(class\text{-}ring::'a\ ring)}$
  proof −
   let *?a* = *f* ∘ *padr m*
   let *?b* = *f* ∘ *padl n*
   have *lcA′*: *lincomb n ?a A′* : *carrier-vec n* **using** *N.lincomb-closed A′* **by** *auto*
    have *lcB′*: *lincomb m ?b B′* : *carrier-vec m* **using** *M.lincomb-closed B′* **by**
auto

   have 0_v *n* $@_v$ 0_v *m* = 0_v $(n+m)$ **by** *auto*
   also have ... = *lincomb* $(n+m)$ *f U* **using** *0* **by** *auto*
   also have *U* = *?UA* ∪ *?UB* **using** *UAB* **by** *auto*

also have *lincomb* (*n*+*m*) *f* ... = *lincomb* (*n*+*m*) *f ?UA* + *lincomb* (*n*+*m*) *f ?UB*
apply(*subst NM.lincomb-union*) **using** *A B finU disj* **by** *auto*
also have *lincomb* (*n*+*m*) *f ?UA* = *lincomb* (*n*+*m*) (*restrict f ?UA*) *?UA*
apply (*subst NM.lincomb-restrict*) **using** *A finU* **by** *auto*
also have *restrict f ?UA* = *restrict* (*?a* ∘ *unpadr m*) *?UA*
apply(*rule restrict-ext*) **by** *auto*
also have *lincomb* (*n*+*m*) ... *?UA* = *lincomb* (*n*+*m*) (*?a* ∘ *unpadr m*) *?UA*
apply(*subst NM.lincomb-restrict*) **using** *A finU* **by** *auto*
also have *?UA* = *padr m ‘ A′* **using** *UAA′*.
also have *lincomb* (*n*+*m*) (*?a* ∘ *unpadr m*) ... =
padr m (*lincomb n ?a A′*)
using *lincomb-pad*(*1*)[*OF A′ finA′*,*symmetric*].
also have *lincomb* (*n*+*m*) *f ?UB* = *lincomb* (*n*+*m*) (*restrict f ?UB*) *?UB*
apply (*subst NM.lincomb-restrict*) **using** *B finU* **by** *auto*
also have *restrict f ?UB* = *restrict* (*?b* ∘ *unpadl n*) *?UB*
apply(*rule restrict-ext*) **by** *auto*
also have *lincomb* (*n*+*m*) ... *?UB* = *lincomb* (*n*+*m*) (*?b* ∘ *unpadl n*) *?UB*
apply(*subst NM.lincomb-restrict*) **using** *B finU* **by** *auto*
also have *n*+*m* = *m*+*n* **by** *auto*
also have *?UB* = *padl n ‘ B′* **using** *UBB′*.
also have *lincomb* (*m*+*n*) (*?b* ∘ *unpadl n*) ... =
padl n (*lincomb m ?b B′*)
using *lincomb-pad*(*2*)[*OF B′ finB′*,*symmetric*].
also have *padr m* (*lincomb n ?a A′*) + ... =
(*lincomb n ?a A′* + 0_v *n*) $@_v$ (0_v *m* + *lincomb m ?b B′*)
apply (*rule append-vec-add*) **using** *lcA′ lcB′* **by** *auto*
also have ... = *lincomb n ?a A′* $@_v$ *lincomb m ?b B′* **using** *lcA′ lcB′* **by** *auto*
finally have 0_v *n* $@_v$ 0_v *m* = *lincomb n ?a A′* $@_v$ *lincomb m ?b B′*.
hence 0_v *n* = *lincomb n ?a A′* ∧ 0_v *m* = *lincomb m ?b B′*
apply(*subst append-vec-eq*[*symmetric*]) **using** *lcA′ lcB′* **by** *auto*
from *conjunct1*[*OF this*] *conjunct2*[*OF this*]
have *?a* : *A′* → {*0*} *?b* : *B′* → {*0*}
using *N.not-lindepD*[*OF liA finA′ A′A*]
using *M.not-lindepD*[*OF liB finB′ B′B*] **by** *auto*
hence *f* : *padr m ‘ A′* → {*0*} *f* : *padl n ‘ B′* → {*0*} **by** *auto*
hence *f* : *padr m ‘ A′* ∪ *padl n ‘ B′* → {*0*} **by** *auto*
hence *f* : *U* → {*0*} **using** *split* **by** *auto*
hence *f u* = *0* **using** *uU* **by** *auto*
thus *?thesis* **by** *simp*
qed
qed
qed

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 −
  have [simp]: n + m = m + n by auto
  have A: A ∈ carrier-mat (n+m) (n+m)
    using Adef four-block-carrier-mat[OF B D] by auto
  interpret vardim TYPE('a).
  interpret MN: vectorspace class-ring M (n+m) using vec-vs.
  interpret KA: kernel n+m n+m A by (unfold-locales, rule A)
  interpret KB: kernel n n B by (unfold-locales, rule B)
  interpret KD: kernel m m D by (unfold-locales, rule D)

  note [simp] = module-vec-simps

  from kernel-basis-exists[OF B]
    obtain baseB where fin-bB: finite baseB and bB: KB.basis baseB by blast
  hence bBkB: baseB ⊆ mat-kernel B unfolding KB.Ker.basis-def by auto
  hence bBc: baseB ⊆ carrier-vec n using mat-kernel-carrier[OF B] by auto
  have bB0: 0v n ∉ baseB
    using bB unfolding KB.Ker.basis-def
    using KB.Ker.vs-zero-lin-dep[OF bBkB] by auto
  have bBkA: padr m ‘ baseB ⊆ mat-kernel A
  proof
    fix a assume a : padr m ‘ baseB
    then obtain b where ab: a = padr m b and b : baseB by auto
    hence b : mat-kernel B using bB unfolding KB.Ker.basis-def by auto
    hence padr m b : mat-kernel A
      unfolding Adef using kernel-padr[OF - B - D] by auto
    thus a : mat-kernel A using ab by auto
  qed
  from kernel-basis-exists[OF D]
    obtain baseD where fin-bD: finite baseD and bD: KD.basis baseD by blast
  hence bDkD: baseD ⊆ mat-kernel D unfolding KD.Ker.basis-def by auto
  hence bDc: baseD ⊆ carrier-vec m using mat-kernel-carrier[OF D] by auto
  have bDkA: padl n ‘ baseD ⊆ mat-kernel A
  proof
    fix a assume a : padl n ‘ baseD
    then obtain d where ad: a = padl n d and d : baseD by auto
    hence d : mat-kernel D using bD unfolding KD.Ker.basis-def by auto
    hence padl n d : mat-kernel A
      unfolding Adef using kernel-padl[OF - B - D] by auto
    thus a : mat-kernel A using ad by auto
  qed
  let ?BD = (padr m ‘ baseB ∪ padl n ‘ baseD)
  have finBD: finite ?BD using fin-bB fin-bD by auto
  have KA.basis ?BD
    unfolding KA.Ker.basis-def
  proof (intro conjI Set.equalityI)
    show BDk: ?BD ⊆ mat-kernel A using bBkA bDkA by auto
```

also have *mat-kernel A* ⊆ *carrier-vec* (*m*+*n*) **using** *mat-kernel-carrier A* **by** *auto*
finally have *BD*: *?BD* ⊆ *carrier* (*M* (*n* + *m*)) **by** *auto*
show *mat-kernel A* ⊆ *KA.Ker.span ?BD*
  unfolding *KA.span-same*[*OF BDk*]
proof
  have *BD*: *?BD* ⊆ *carrier-vec* (*n*+*m*) (**is** - ⊆ *?R*)
  proof(*rule*)
    fix *v* **assume** *v* : *?BD*
    moreover
    { assume *v* : *padr m ‘ baseB*
      then obtain *b* **where** *b* : *baseB* **and** *vb*: *v* = *padr m b* **by** *auto*
      hence *b* : *carrier-vec n* **using** *bBc* **by** *auto*
      hence *v* : *?R* **unfolding** *vb* **apply**(*subst append-carrier-vec*) **by** *auto*
    }
    moreover
    { assume *v* : *padl n ‘ baseD*
      then obtain *d* **where** *d* : *baseD* **and** *vd*: *v* = *padl n d* **by** *auto*
      hence *d* : *carrier-vec m* **using** *bDc* **by** *auto*
      hence *v* : *?R* **unfolding** *vd* **apply**(*subst append-carrier-vec*) **by** *auto*
    }
    ultimately show *v*: *?R* **by** *auto*
  qed
  fix *a* **assume** *a*: *a* : *mat-kernel A*
  hence *a* : *carrier-vec* (*n*+*m*) **using** *a mat-kernel-carrier*[*OF A*] **by** *auto*
  hence *a* = *vec-first a n* $@_v$ *vec-last a m* (**is** - = *?b* $@_v$ *?d*) **by** *simp*
  also have ... = *padr m ?b* + *padl n ?d* **by** *auto*
  finally have *1*: *a* = *padr m ?b* + *padl n ?d*.

  have *subkernel*: *?b* : *mat-kernel B ?d* : *mat-kernel D*
    using *mat-kernel-split*[*OF B D*] *a Adef* **by** *auto*
  hence *?b* : *span n baseB*
    using *bB* **unfolding** *KB.Ker.basis-def* **using** *KB.span-same* **by** *auto*
  hence *padr m ?b* : *padr m ‘ span n baseB* **by** *auto*
  also have *padr m ‘ span n baseB* = *span* (*n*+*m*) (*padr m ‘ baseB*)
    using *span-pad*[*OF bBc*] **by** *auto*
  also have ... ⊆ *span* (*n*+*m*) *?BD* **using** *MN.span-is-monotone* **by** *auto*
  finally have *2*: *padr m ?b* : *span* (*n*+*m*) *?BD*.
  have *?d* : *span m baseD*
    using *subkernel bD* **unfolding** *KD.Ker.basis-def* **using** *KD.span-same* **by** *auto*
  hence *padl n ?d* : *padl n ‘ span m baseD* **by** *auto*
  also have *padl n ‘ span m baseD* = *span* (*n*+*m*) (*padl n ‘ baseD*)
    using *span-pad*[*OF bDc*] **by** *auto*
  also have ... ⊆ *span* (*n*+*m*) *?BD* **using** *MN.span-is-monotone* **by** *auto*
  finally have *3*: *padl n ?d* : *span* (*n*+*m*) *?BD*.

  have *padr m ?b* + *padl n ?d* : *span* (*n*+*m*) *?BD*
    using *MN.span-add1*[*OF - 2 3*] *BD* **by** *auto*

thus *a* ∈ *span* (*n*+*m*) *?BD* **using** *1* **by** *auto*
qed
show *KA.Ker.span ?BD* ⊆ *mat-kernel A* **using** *KA.Ker.span-closed*[*OF BDk*] **by** *auto*
have *li*: ~ *lin-dep n baseB* ~ *lin-dep m baseD*
using *bB*[*unfolded KB.Ker.basis-def*]
unfolding *KB.lindep-same*[*OF bBkB*]
using *bD*[*unfolded KD.Ker.basis-def*]
unfolding *KD.lindep-same*[*OF bDkD*] **by** *auto*
show ~ *KA.Ker.lin-dep ?BD*
unfolding *KA.lindep-same*[*OF BDk*]
apply(*rule padr-padl-lindep*) **using** *bBc bDc li* **by** *auto*
qed
hence *KA.dim* = *card ?BD* **using** *KA.Ker.dim-basis*[*OF finBD*] **by** *auto*
also have *card ?BD* = *card* (*padr m ' baseB*) + *card* (*padl n ' baseD*)
apply(*rule card-Un-disjoint*)
using *pad-disjoint*[*OF bBc bB0 bDc*] *fin-bB fin-bD* **by** *auto*
also have ... = *card baseB* + *card baseD*
using *card-image*[*OF subset-inj-on*[*OF padr-inj*]]
using *card-image*[*OF subset-inj-on*[*OF padl-inj*]] *bBc bDc* **by** *auto*
also have *card baseB* = *KB.dim* **using** *KB.Ker.dim-basis*[*OF fin-bB*] *bB* **by** *auto*
also have *card baseD* = *KD.dim* **using** *KD.Ker.dim-basis*[*OF fin-bD*] *bD* **by** *auto*
finally show *?thesis*.

qed

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 −
from *similar-mat-witD2*[*OF A wit*]
have *QP*: *Q* ∗ *P* = 1_m *n* **and** *AB*: *A* = *P* ∗ *B* ∗ *Q* **and**
A: *A* ∈ *carrier-mat n n* **and** *B*: *B* ∈ *carrier-mat n n* **and** *P*: *P* ∈ *carrier-mat n n* **and** *Q*: *Q* ∈ *carrier-mat n n* **by** *auto*
from *P B* **have** *PB*: *P* ∗ *B* ∈ *carrier-mat n n* **by** *auto*
show *?thesis* **unfolding** *AB mat-kernel-dim-mult-eq-right*[*OF PB Q P QP*] *mat-kernel-mult-eq*[*OF B P Q QP*]
by *simp*
qed

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 −
define *n* **where** $n = \mathit{dim\text{-}row}\ A$
let $?C = \mathit{carrier\text{-}mat}\ n\ n$
from *similar-mat-witD*[*OF refl wit*, *folded n-def*] **have**
A: $A \in ?C$ **and** *B*: $B \in ?C$ **and** *P*: $P \in ?C$ **and** *Q*: $Q \in ?C$
and *PQ*: $P * Q = 1_m\ n$ **and** *QP*: $Q * P = 1_m\ n$
and *AB*: $A = P * B * Q$
by *auto*
have $\mathit{char\text{-}matrix}\ A\ ev = (P * B * Q + (-ev) \cdot_m (P * Q))$
unfolding *char-matrix-def n-def*[*symmetric*] **unfolding** *AB PQ*
by (*intro eq-matI*, *insert P B Q*, *auto*)
also have $(-ev) \cdot_m (P * Q) = P * ((-ev) \cdot_m 1_m\ n) * Q$ **using** *P Q*
by (*metis mult-smult-assoc-mat mult-smult-distrib one-carrier-mat right-mult-one-mat*)
also have $P * B * Q + \ldots = (P * B + P * ((-ev) \cdot_m 1_m\ n)) * Q$ **using** *P B*
by (*intro add-mult-distrib-mat*[*symmetric*, *OF - - Q*, *of - n*], *auto*)
also have $P * B + P * ((-ev) \cdot_m 1_m\ n) = P * (B + (-ev) \cdot_m 1_m\ n)$
by (*intro mult-add-distrib-mat*[*symmetric*, *OF P B*], *auto*)
also have $(B + (-ev) \cdot_m 1_m\ n) = \mathit{char\text{-}matrix}\ B\ ev$ **unfolding** *char-matrix-def*
by (*intro eq-matI*, *insert B*, *auto*)
finally have *AB*: $\mathit{char\text{-}matrix}\ A\ ev = P * \mathit{char\text{-}matrix}\ B\ ev * Q$ **.**
show *similar-mat-wit* (*char-matrix A ev*) (*char-matrix B ev*) *P Q*
by (*intro similar-mat-witI*[*OF PQ QP AB - - P Q*], *insert A B*, *auto*)
qed

context fixes *ty* :: *'a* :: *field itself*
begin

lemma *dim-kernel-non-zero-jordan-block-pow*: **assumes** *a*: $a \neq 0$
shows $\mathit{kernel.dim}\ n\ (\mathit{jordan\text{-}block}\ n\ (a :: 'a)\ \widehat{}_m\ k) = 0$
by (*rule kernel-upper-triangular*[*OF pow-carrier-mat*[*OF jordan-block-carrier*]],
unfold jordan-block-pow, *insert a*, *auto simp*: *diag-mat-def*)

lemma *dim-kernel-zero-jordan-block-pow*:
$\mathit{kernel.dim}\ n\ ((\mathit{jordan\text{-}block}\ n\ (0 :: 'a))\ \widehat{}_m\ k) = \min k\ n$ (**is** *kernel.dim* - $?A =$ $?c$)
proof −
have *A*: $?A \in \mathit{carrier\text{-}mat}\ n\ n$ **by** *auto*
hence *dim*: $\mathit{dim\text{-}row}\ ?A = n$ **by** *simp*
let $?f = \lambda\ i.\ \min\ (k + i)\ n$

```
  have piv: pivot-fun ?A ?f n unfolding jordan-block-zero-pow
    by (intro pivot-funI, auto)
  hence row: row-echelon-form ?A unfolding row-echelon-form-def by auto
  from find-base-vectors(5−6)[OF row A]
  have kernel.dim n ?A = n − length (map fst (pivot-positions ?A)) by auto
  also have length (map fst (pivot-positions ?A)) = card (fst ‘ set (pivot-positions
?A))
    by (subst distinct-card[OF pivot-positions(2)[OF A piv], symmetric], simp)
  also have fst ‘ set (pivot-positions ?A) = { 0 ..< (n − ?c)} unfolding pivot-positions(1)[OF
A piv]
    by force
  also have card ... = n − ?c by simp
  finally show ?thesis by simp
qed

definition dim-gen-eigenspace :: 'a mat ⇒ 'a ⇒ nat ⇒ nat where
  dim-gen-eigenspace A ev k = kernel-dim ((char-matrix A ev) ^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 −
  let ?JM = λ n-as. jordan-matrix n-as
  let ?CM = λ n-as. char-matrix (?JM n-as) ev
  let ?A = λ n-as. (?CM n-as) ^m k
  let ?n = λ n-as. sum-list (map fst n-as)
  let ?C = λ n-as. carrier-mat (?n n-as) (?n n-as)
  let ?sum = λ n-as. ∑ n ← map fst [(n, e)←n-as . e = ev]. min k n
  let ?dim = λ n-as. sum-list (map fst n-as)
  let ?kdim = λ n-as. kernel.dim (?dim n-as) (?A n-as)
  have JM: ⋀ n-as. ?JM n-as ∈ ?C n-as by auto
  have CM: ⋀ n-as. ?CM n-as ∈ ?C n-as by auto
  have A: ⋀ n-as. ?A n-as ∈ ?C n-as by auto
  have dimc: dim-col (?JM n-as) = ?dim n-as by simp
  interpret K: kernel ?dim n-as ?dim n-as ?A n-as
    by (unfold-locales, rule A)
  show ?thesis unfolding dim-gen-eigenspace-def K.kernel-dim
  proof (induct n-as)
    case Nil
    have ?JM Nil = 1m 0 unfolding jordan-matrix-def
      by (intro eq-matI, auto)
    hence id: ?A Nil = 1m 0 unfolding char-matrix-def by auto
    show ?case unfolding id using kernel-one-mat[of 0] by auto
  next
    case (Cons ne n-as')
    let ?n-as = Cons ne n-as'
    let ?d = ?dim ?n-as
    let ?d' = ?dim n-as'
    obtain n e where ne: ne = (n,e) by force
```

have *dim*: *?d* = *n* + *?d′* **unfolding** *ne* **by** *simp*
let *?jb* = *jordan-block n e*
let *?cm* = *char-matrix ?jb ev*
let *?a* = *?cm* $\widehat{\ }_m$ *k*
have *a*: *?a* ∈ *carrier-mat n n* **by** *simp*
from *JM*[*of n-as′*] **have** *dim-rec*: *dim-row* (*?JM n-as′*) = *?d′ dim-col* (*?JM n-as′*) = *?d′* **by** *auto*
hence *JM-id*: *?JM ?n-as* = *four-block-mat ?jb* (0_m *n ?d′*) (0_m *?d′ n*) (*?JM n-as′*)
unfolding *ne jordan-matrix-def* **using** *JM*[*of n-as′*]
by (*simp add*: *Let-def*)
have *CM-id*: *?CM ?n-as* = *four-block-mat ?cm* (0_m *n ?d′*) (0_m *?d′ n*) (*?CM n-as′*)
unfolding *JM-id*
unfolding *char-matrix-def*
by (*intro eq-matI*, *auto*)
have *A-id*: *?A ?n-as* = *four-block-mat ?a* (0_m *n ?d′*) (0_m *?d′ n*) (*?A n-as′*)
unfolding *CM-id* **by** (*rule pow-four-block-mat*[*OF - CM*], *auto*)
have *kdim*: *?kdim ?n-as* = *kernel.dim n ?a* + *?kdim n-as′*
unfolding *dim A-id*
by (*rule kernel-four-block-0-mat*[*OF refl a A*])
also have *?kdim n-as′* = *?sum n-as′* **by** (*rule Cons*)
also have *kernel.dim n ?a* = (*if e* = *ev then min k n else 0*)
using *dim-kernel-zero-jordan-block-pow*[*of n k*]
dim-kernel-non-zero-jordan-block-pow[*of e* − *ev n k*]
unfolding *char-matrix-jordan-block*
by (*cases e* = *ev*, *auto*)
also have ... + *?sum n-as′* = *?sum ?n-as* **unfolding** *ne* **by** *auto*
finally show *?case* **.**
qed
qed

lemma *dim-gen-eigenspace-similar*: **assumes** *sim*: *similar-mat A B*
shows *dim-gen-eigenspace A* = *dim-gen-eigenspace B*
proof (*intro ext*)
fix *ev k*
define *n* **where** *n* = *dim-row A*
from *sim*[*unfolded similar-mat-def*] **obtain** *P Q* **where**
wit: *similar-mat-wit A B P Q* **by** *auto*
let *?C* = *carrier-mat n n*
from *similar-mat-witD*[*OF refl wit*, *folded n-def*]
have *A*: *A* ∈ *?C* **and** *B*: *B* ∈ *?C* **and** *P*: *P* ∈ *?C* **and** *Q*: *Q* ∈ *?C*
and *PQ*: *P* ∗ *Q* = 1_m *n* **and** *QP*: *Q* ∗ *P* = 1_m *n*
by *auto*
from *similar-mat-wit-pow*[*OF similar-mat-wit-char-matrix*[*OF wit*, *of ev*], *of k*]
have *wit*: *similar-mat-wit* (*char-matrix A ev* $\widehat{\ }_m$ *k*) (*char-matrix B ev* $\widehat{\ }_m$ *k*) *P Q* **.**
from *A B* **have** *cA*: *char-matrix A ev* $\widehat{\ }_m$ *k* ∈ *carrier-mat n n*

and *cB*: *char-matrix B ev* $\widehat{\ }_m$ *k* ∈ *carrier-mat n n* **by** *auto*
hence *dim*: *dim-col* (*char-matrix A ev* $\widehat{\ }_m$ *k*) = *n dim-col* (*char-matrix B ev* $\widehat{\ }_m$ *k*) = *n* **by** *auto*
have *dim-gen-eigenspace A ev k* = *kernel-dim* (*char-matrix A ev* $\widehat{\ }_m$ *k*)
unfolding *dim-gen-eigenspace-def* **using** *A* **by** *simp*
also have ... = *kernel-dim* (*char-matrix B ev* $\widehat{\ }_m$ *k*) **unfolding** *kernel-dim-def dim*
by (*rule similar-mat-wit-kernel-dim*[*OF cA wit*])
also have ... = *dim-gen-eigenspace B ev k*
unfolding *dim-gen-eigenspace-def* **using** *B* **by** *simp*
finally show *dim-gen-eigenspace A ev k* = *dim-gen-eigenspace B ev k* **.**
qed

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 −
from *assms*[*unfolded jordan-nf-def*]
have *sim*: *similar-mat A* (*jordan-matrix n-as*) **by** *auto*
from *dim-gen-eigenspace-jordan-matrix*[*of n-as*, *folded dim-gen-eigenspace-similar*[*OF this*]]
show *?thesis* **.**
qed

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* ≠ *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 −
from *no-0* **obtain** *k1* **where** *k*: *k* = *Suc k1* **by** (*cases k*, *auto*)
let *?k* = *Suc k1* **let** *?k2* = *Suc ?k*
let *?dim* = *dim-gen-eigenspace A ev*
let *?sizes* = *map fst* [(*n*, *e*)←*n-as* . *e* = *ev*]
define *sizes* **where** *sizes* = *?sizes*
let *?two* = *length* (*filter* ((=) (*k*, *ev*)) *n-as*)
have *compute-nr-of-jordan-blocks A ev k* =
?dim ?k + *?dim ?k* − *?dim k1* − *?dim ?k2* **unfolding** *compute-nr-of-jordan-blocks-def k* **by** *simp*
also have ... = *length* (*filter* ((=) *k*) *?sizes*)
unfolding *dim-gen-eigenspace*[*OF jnf*] *k sizes-def*[*symmetric*]
proof (*rule sym*, *induct sizes*)

```
    case (Cons s sizes)
    show ?case
    proof (cases s = ?k)
      case True
      let ?sum = λ k sizes. sum-list (map (min k) sizes)
      let ?len = λ sizes. length (filter ((=) ?k) sizes)
      have len: ?len (s # sizes) = Suc (?len sizes) unfolding True by simp
      have IH: ?len sizes = ?sum ?k sizes + ?sum ?k sizes −
        ?sum k1 sizes − ?sum ?k2 sizes by (rule Cons)
      have ?sum ?k (s # sizes) + ?sum ?k (s # sizes) −
        ?sum k1 (s # sizes) − ?sum ?k2 (s # sizes)
        = Suc (?sum ?k sizes + ?sum ?k sizes) −
         (?sum k1 sizes + ?sum ?k2 sizes)
        using True by simp
       also have ... = Suc (?sum ?k sizes + ?sum ?k sizes − (?sum k1 sizes +
?sum ?k2 sizes))
        by (rule Suc-diff-le, induct sizes, auto)
      also have ... = ?len (s # sizes) unfolding len IH by simp
      finally show ?thesis by simp
    qed (insert Cons, auto)
  qed simp
  also have ... = length (filter ((=) (k, ev)) n-as) by (induct n-as, force+)
  finally show ?thesis .
qed

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 −
  let ?N = set n-as ∩ UNIV × {ev} − {0} × UNIV
  have N: ?N' = ?N using jnf[unfolded jordan-nf-def] by force
  note cjb = compute-nr-of-jordan-blocks[OF jnf]
  note d = compute-set-of-jordan-blocks-def Let-def
  define kk where kk = Polynomial.order ev (char-poly A)
  define as where as = map (dim-gen-eigenspace A ev) [0 ..< Suc (Suc kk)]
  define cards where cards = map (λ k. (k, 2 ∗ as ! k − as ! (k − 1) − as ! Suc
k)) [1 ..< Suc kk]
  have C: ?C = set (map (λ (k,c). (k,ev)) (filter (λ (k,c). c ≠ 0) cards))
    unfolding d as-def kk-def cards-def by (rule refl)
  {
    fix i
    assume i < Suc (Suc kk)
```

hence *as* ! *i* = *dim-gen-eigenspace A ev i*
unfolding *as-def* **by** (*auto simp del*: *upt-Suc*)
} **note** *as* = *this*

have *cards*: *cards* = *map* (λ *k*. (*k*, *compute-nr-of-jordan-blocks A ev k*)) [*1* ..< *Suc kk*]
unfolding *cards-def*
by (*rule map-cong*[*OF refl*], *insert as*, *unfold compute-nr-of-jordan-blocks-def*, *auto*)
have *C*: *?C* = { (*k*,*ev*) | *k*. *compute-nr-of-jordan-blocks A ev k* ≠ *0* ∧ *k* ≠ *0* ∧ *k* < *Suc kk* }
unfolding *C cards* **by** *force*
{
fix *k*
have (*k*,*ev*) ∈ *?C* ⟷ (*k*,*ev*) ∈ *?N*
proof (*cases k* = *0*)
case *True*
thus *?thesis* **unfolding** *C* **by** *auto*
next
case *False*
show *?thesis*
proof (*cases k* < *Suc kk*)
case *True*
have *length* (*filter* ((=) (*k*, *ev*)) *n-as*) ≠ *0* ⟷
set (*filter* ((=) (*k*, *ev*)) *n-as*) ≠ {} **by** *blast*
have (*k*,*ev*) ∈ *?N* ⟷ *set* (*filter* ((=) (*k*, *ev*)) *n-as*) ≠ {} **using** *False* **by** *auto*
also have ... ⟷ *length* (*filter* ((=) (*k*, *ev*)) *n-as*) ≠ *0* **by** *blast*
also have ... ⟷ *compute-nr-of-jordan-blocks A ev k* ≠ *0*
unfolding *compute-nr-of-jordan-blocks*[*OF jnf False*] **by** *simp*
also have ... ⟷ (*k*,*ev*) ∈ *?C* **unfolding** *C* **using** *False True* **by** *auto*
finally show *?thesis* **by** *auto*
next
case *False*
hence (*k*,*ev*) ∉ *?C* **unfolding** *C* **by** *auto*
moreover from *False kk-def* **have** *k*: *k* > *Polynomial.order ev* (*char-poly A*) **by** *auto*
with *jordan-nf-block-size-order-bound*[*OF jnf*, *of k ev*]
have (*k*,*ev*) ∉ *?N* **by** *auto*
ultimately show *?thesis* **by** *simp*
qed
qed
}
thus *?thesis* **unfolding** *C N*[*symmetric*] **by** *auto*
qed

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 −
from *compute-set-of-jordan-blocks*[*OF assms*(*1*), *unfolded compute-set-of-jordan-blocks*[*OF assms*(*2*)]]
show *?thesis* **by** *auto*
qed

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*) $\in$ *carrier-mat* n n
shows *spectrum* $A = \{k.$ *poly* (*char-poly* A) $k = 0\}$
unfolding *spectrum-def eigenvalue-root-char-poly*[*OF* A, *symmetric*] **by** *auto*

lemma *card-finite-spectrum*: **assumes** A: (A :: $'a$ :: *field mat*) $\in$ *carrier-mat* n n
shows *finite* (*spectrum* A) *card* (*spectrum* A) $\leq n$
proof −
define *CP* **where** *CP* = *char-poly* A
from *spectrum-root-char-poly*[*OF* A] **have** *id*: *spectrum* $A = \{ k.$ *poly CP* $k = 0\}$
unfolding *CP-def* **by** *auto*
from *degree-monic-char-poly*[*OF* A] **have** *d*: *degree CP* $= n$ **and** *c*: *coeff CP* $n = 1$
unfolding *CP-def* **by** *auto*
from *c* **have** *CP* $\neq 0$ **by** *auto*
from *poly-roots-finite*[*OF this*]
show *finite* (*spectrum* A) **unfolding** *id* **.**
from *poly-roots-degree*[*OF* ‹*CP* $\neq 0$›]
show *card* (*spectrum* A) $\leq n$ **unfolding** *id* **using** *d* **by** *simp*
qed

lemma *spectrum-non-empty*: **assumes** A: (A :: *complex mat*) $\in$ *carrier-mat* n n
and *n*: $n > 0$

shows *spectrum A* $\neq$ {}
proof –
define *CP* **where** *CP* = *char-poly A*
from *spectrum-root-char-poly*[*OF A*] **have** *id*: *spectrum A* = { *k*. *poly CP k* = *0*}
unfolding *CP-def* **by** *auto*
from *degree-monic-char-poly*[*OF A*] **have** *d*: *degree CP* > *0* **using** *n*
unfolding *CP-def* **by** *auto*
hence ¬ *constant* (*poly CP*) **by** (*simp add*: *constant-degree*)
from *fundamental-theorem-of-algebra*[*OF this*] **show** *?thesis* **unfolding** *id* **by** *auto*
qed

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 –
define *SA* **where** *SA* = *norm* ‘ *spectrum A*
from *card-finite-spectrum*[*OF A*]
have *fin*: *finite SA* **unfolding** *SA-def* **by** *auto*
from *spectrum-non-empty*[*OF A n*] **have** *ne*: *SA* $\neq$ {} **unfolding** *SA-def* **by** *auto*
note *d* = *spectral-radius-def SA-def*[*symmetric*] *Sup-fin-Max*[*symmetric*]
show *?one* **unfolding** *d*
by (*rule Sup-fin.closed*[*OF fin ne*], *auto simp*: *sup-real-def*)
assume *a* ∈ *norm* ‘ *spectrum A*
thus *a* ≤ *spectral-radius A* **unfolding** *d*
by (*intro Sup-fin.coboundedI*[*OF fin*])
qed

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 –
let *?p* = *char-poly A*
from *char-poly-factorized*[*OF A*] **obtain** *as* **where** *cA*: *char-poly A* = (∏ *a*←*as*. [:− *a*, *1*:])
and *len*: *length as* = *n* **by** *auto*
show *?thesis*
proof (*rule factored-char-poly-norm-bound*[*OF A cA jnf-exists*])
fix *a*
show *length* (*filter* ((=) *a*) *as*) ≤ *n* **using** *len* **by** *auto*

assume *a* ∈ *set as*
from *linear-poly-root*[*OF this*]
have *poly ?p a* = *0* **unfolding** *cA* **by** *simp*
with *spectrum-root-char-poly*[*OF A*]
have *mem*: *norm a* ∈ *norm ‘ spectrum A* **by** *auto*
with *card-finite-spectrum*[*OF A*] **have** *n* > *0* **by** (*cases n*, *auto*)
from *spectral-radius-mem-max*(*2*)[*OF A this mem*] *sr-1*
show *norm a* ≤ *1* **by** *auto*
qed
qed

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 −
let *?p* = *char-poly A*
from *char-poly-factorized*[*OF A*] **obtain** *as* **where** *cA*: *char-poly A* = (∏ *a*←*as*. [:− *a*, *1*:]) **by** *auto*
have ∃ *c1 c2*. ∀ *k*. *norm-bound* (*A* $\widehat{\ }_m$ *k*) (*c1* + *c2* ∗ *of-nat k* ^ (*0* − *1*))
proof (*rule factored-char-poly-norm-bound*[*OF A cA jnf-exists*])
fix *a*
assume *a* ∈ *set as*
from *linear-poly-root*[*OF this*]
have *poly ?p a* = *0* **unfolding** *cA* **by** *simp*
with *spectrum-root-char-poly*[*OF A*]
have *mem*: *norm a* ∈ *norm ‘ spectrum A* **by** *auto*
with *card-finite-spectrum*[*OF A*] **have** *n* > *0* **by** (*cases n*, *auto*)
from *spectral-radius-mem-max*(*2*)[*OF A this mem*] *sr-1*
have *lt*: *norm a* < *1* **by** *auto*
thus *norm a* ≤ *1* **by** *auto*
from *lt* **show** *norm a* = *1* ⟹ *length* (*filter* ((=) *a*) *as*) ≤ *0* **by** *auto*
qed
thus *?thesis* **by** *auto*
qed

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* $\widehat{\ }_m$ *k* $*_v$ *v* = *c*^*k* $\cdot_v$ *v*
proof −
from *sr-1 spectral-radius-mem-max*[*OF A n*] **obtain** *ev*
where *ev*: *ev* ∈ *spectrum A* **and** *gt*: *norm ev* > *1* **by** *auto*
from *ev*[*unfolded spectrum-def eigenvalue-def*[*abs-def*]]
obtain *v* **where** *ev*: *eigenvector A v ev* **by** *auto*

from *eigenvector-pow*[*OF A this*] *this*[*unfolded eigenvector-def*] *A gt*
show *?thesis*
by (*intro exI*[*of - v*], *intro exI*[*of - ev*], *auto*)
qed

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*) $\in$ *carrier-mat n n*
and *sr-1*: *spectral-radius A* $\leq$ *1*
shows $\exists$ *c1 c2*. $\forall$ *k*. *norm-bound* (*A* $\widehat{\ }_m$ *k*) (*c1* + *c2* $*$ *of-nat k* ^ (*n* − *1*))
by (*rule spectral-radius-jnf-norm-bound-le-1*[*OF A sr-1*],
insert char-poly-factorized[*OF A*] *jordan-nf-exists*[*OF A*], *blast*)

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*) $\in$ *carrier-mat n n*
and *sr-1*: *spectral-radius A* < *1*
shows $\exists$ *c*. $\forall$ *k*. *norm-bound* (*A* $\widehat{\ }_m$ *k*) *c*
by (*rule spectral-radius-jnf-norm-bound-less-1*[*OF A sr-1*],
insert char-poly-factorized[*OF A*] *jordan-nf-exists*[*OF A*], *blast*)

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* $\in$ *carrier-mat n n*
and *sr-1*: *spectral-radius A* $\leq$ *1*
and *eq-1*: $\bigwedge$ *ev k*. *poly* (*char-poly A*) *ev* = *0* $\Longrightarrow$ *norm ev* = *1* $\Longrightarrow$ *Polynomial.order ev* (*char-poly A*) $\leq$ *d*
shows $\exists$ *c1 c2*. $\forall$ *k*. *norm-bound* (*A* $\widehat{\ }_m$ *k*) (*c1* + *c2* $*$ *of-nat k* ^ (*d* − *1*))
proof −
{
fix *ev*
assume *poly* (*char-poly A*) *ev* = *0*
with *eigenvalue-root-char-poly*[*OF A*] **have** *ev*: *eigenvalue A ev* **by** *simp*
hence *norm ev* $\in$ *norm* ' *spectrum A* **unfolding** *spectrum-def* **by** *auto*
from *spectral-radius-mem-max*(*2*)[*OF A eigenvalue-imp-nonzero-dim*[*OF A ev*] *this*] *sr-1*
have *norm ev* $\leq$ *1* **by** *auto*
} **note** *le-1* = *this*
let *?p* = *char-poly A*
from *char-poly-factorized*[*OF A*] **obtain** *as* **where** *cA*: *char-poly A* = ($\prod$ *a*$\leftarrow$*as*. [:− *a*, *1*:])
and *lenn*: *length as* = *n* **by** *auto*
from *degree-monic-char-poly*[*OF A*] **have** *deg*: *degree* (*char-poly A*) = *n* **by** *auto*
show *?thesis*
proof (*rule factored-char-poly-norm-bound*[*OF A cA jordan-nf-exists*[*OF A*]], *rule cA*)

fix *ev*
assume *ev* ∈ *set as*
hence *root*: *poly* (*char-poly A*) *ev* = *0* **unfolding** *cA* **by** (*rule linear-poly-root*)
from *le-1*[*OF root*] **show** *norm ev* ≤ *1* **.**
let *?k* = *length* (*filter* ((=) *ev*) *as*)
have *len*: *length* (*filter* ((=) (− *ev*)) (*map uminus as*)) = *length* (*filter* ((=) *ev*) *as*)
by (*induct as*, *auto*)
have *prod*: (∏ *a*←*map uminus as*. [:*a*, *1*:]) = (∏ *a*←*as*. [:− *a*, *1*:])
by (*induct as*, *auto*)
have *dvd*: [:− *ev*, *1*:] ^ *?k dvd char-poly A* **unfolding** *cA* **using**
poly-linear-exp-linear-factors-rev[*of* − *ev map uminus as*]
unfolding *len prod* **.**
from ‹*ev* ∈ *set as*› *deg lenn*
have *degree* (*char-poly A*) ≠ *0* **by** (*cases as*, *auto*)
hence *char-poly A* ≠ *0* **by** *auto*
from *order-max*[*OF dvd this*] **have** *k*: *?k* ≤ *Polynomial.order ev* (*char-poly A*)
.
assume *norm ev* = *1*
from *eq-1*[*OF root this*] *k*
show *?k* ≤ *d* **by** *simp*
qed
qed

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*)
using *nth-zip nth-map length-zip* **by** (*simp add*: *assms*(*1*) *assms*(*2*))

lemma *nth-map-zip2*:
assumes *i* < *length* (*map f* (*zip xs ys*))
shows *map f* (*zip xs ys*) ! *i* = *f* (*xs* ! *i*, *ys* ! *i*)
using *nth-zip nth-map length-zip assms* **by** *simp*

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*
using *assms* **proof** (*induction xs*)
  case (*Cons x xs*)
  then show *?case*
    using *find-first.simps*(*2*) *nth-Cons-0 nth-Cons-Suc set-ConsD* **by** *auto*
qed *auto*

lemma *nth-find-first*:
assumes *a* ∈ *set xs*
shows *xs* ! (*find-first a xs*) = *a*
using *assms* **proof** (*induction xs*)
  case (*Cons x xs*)
  then show *?case*
    using *find-first.simps*(*2*) *nth-Cons-0 nth-Cons-Suc set-ConsD* **by** *auto*
qed *auto*

lemma *find-first-unique*:
assumes *distinct xs*
and *i* < *length xs*
shows *find-first* (*xs* ! *i*) *xs* = *i*
using *assms* **proof** (*induction xs arbitrary*:*i*)
  case (*Cons x xs i*)
  then show *?case* **by** (*cases i*; *auto*)
qed *auto*

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 −
  have *vectorspace K* (*span-vs S*)
   using *field.field-axioms vectorspace-def submodule-is-module*[*OF span-is-submodule*[*OF assms*(*1*)]] **by** *metis*
  have *S* ⊆ *carrier* (*span-vs S*) **by** (*simp add*: *assms*(*1*) *in-own-span*)
  have *LinearCombinations.module.span K* (*vs* (*span S*)) *S* = *carrier* (*vs* (*span S*))
    using *module.span-li-not-depend*[*OF - span-is-submodule*[*OF assms*(*1*)]]

by (*simp add*: *assms*(*1*) *in-own-span*)
have *vectorspace.basis K* (*vs* (*span S*)) *S*
using *vectorspace.dim-gen-is-basis*[*OF* ‹*vectorspace K* (*span-vs S*)› ‹*finite S*› ‹$S \subseteq$ *carrier* (*span-vs S*)›
‹*LinearCombinations.module.span K* (*vs* (*span S*)) *S* = *carrier* (*vs* (*span S*))›] ‹*vectorspace.dim K* (*span-vs S*) = *card S*›
by *simp*
then have *LinearCombinations.module.lin-indpt K* (*vs* (*span S*)) *S*
using *vectorspace.basis-def*[*OF* ‹*vectorspace K* (*span-vs S*)›] **by** *blast*
then show *?thesis* **using** *module.span-li-not-depend*[*OF* - *span-is-submodule*[*OF* *assms*(*1*)]]
by (*simp add*: *assms*(*1*) *in-own-span*)
qed

lemma (**in** *vectorspace*) *dim-span*:
assumes $S \subseteq$ *carrier V*
and *finite S*
and *maximal U* ($\lambda T.\ T \subseteq S \wedge$ *lin-indpt T*)
shows *vectorspace.dim K* (*span-vs S*) = *card U*
proof −
have *lin-indpt U* $U \subseteq S$ **by** (*metis assms*(*3*) *maximal-def*)+
then have $U \subseteq$ *span S* **using** *in-own-span*[*OF assms*(*1*)] **by** *blast*
then have *lin-indpt*: *LinearCombinations.module.lin-indpt K* (*span-vs S*) *U*
using *module.span-li-not-depend*(*2*)[*OF* ‹$U \subseteq$ *span S*›] ‹*lin-indpt U*› *assms*(*1*) *span-is-submodule* **by** *blast*
have *span U* = *span S*
proof (*rule ccontr*)
assume *span U* $\neq$ *span S*
have *span U* $\subseteq$ *span S* **using** *span-is-monotone* ‹$U \subseteq S$› **by** *metis*
then have $\neg$ $S \subseteq$ *span U* **by** (*meson* ‹$U \subseteq S$› ‹*span U* $\neq$ *span S*› *assms*(*1*) *span-is-submodule*
span-is-subset subset-antisym subset-trans)
then obtain *s* **where** $s \in S$ $s \notin$ *span U* **by** *blast*
then have *lin-indpt* ($U \cup \{s\}$) **using** *lindep-span*
by (*meson* ‹$U \subseteq S$› ‹*lin-indpt U*› *assms*(*1*) *lin-dep-iff-in-span rev-subsetD span-mem subset-trans*)
have $s \notin U$ **using** ‹$U \subseteq S$› ‹$s \notin$ *span U*› *assms*(*1*) *span-mem* **by** *auto*
then have ($U \cup \{s\}$) $\subseteq S \wedge$ *lin-indpt* ($U \cup \{s\}$) **using** ‹$U \subseteq S$› ‹*lin-indpt* ($U \cup \{s\}$)› ‹$s \in S$› **by** *auto*
then have $\neg$*maximal U* ($\lambda T.\ T \subseteq S \wedge$ *lin-indpt T*)
unfolding *maximal-def* **using** *Un-subset-iff* ‹$s \notin U$› *insert-subset order-refl* **by** *auto*
then show *False* **using** *assms* **by** *metis*
qed
then have *span*:*LinearCombinations.module.span K* (*vs* (*span S*)) *U* = *span S*
using *module.span-li-not-depend*[*OF* ‹$U \subseteq$ *span S*›]
by (*simp add*: *LinearCombinations.module.span-is-submodule assms*(*1*) *module-axioms*)
have *vectorspace K* (*vs* (*span S*))

using *field.field-axioms vectorspace-def submodule-is-module*[*OF span-is-submodule*[*OF assms*(*1*)]] **by** *metis*
then have *vectorspace.basis K* (*vs* (*span S*)) *U* **using** *vectorspace.basis-def*[*OF* ‹*vectorspace K* (*vs* (*span S*))›]
by (*simp add*: *span* ‹$U \subseteq$ *span S*› *lin-indpt*)
then show *?thesis*
using ‹$U \subseteq S$› ‹*vectorspace K* (*vs* (*span S*))› *assms*(*2*) *infinite-super vectorspace.dim-basis* **by** *blast*
qed

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* ($\lambda T.\ T \subseteq$ *set* (*cols A*) $\wedge$ *lin-indpt T*)
shows *rank A* = *card S*
proof −
have *set* (*cols A*) $\subseteq$ *carrier-vec n* **using** *cols-dim assms*(*1*) **by** *blast*
have *finite* (*set* (*cols A*)) **by** *blast*
show *?thesis* **using** *dim-span*[*OF* ‹*set* (*cols A*) $\subseteq$ *carrier-vec n*› ‹*finite* (*set* (*cols A*))› *assms*(*2*)]
unfolding *rank-def* **by** *blast*
qed

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 −
have *finite* ($S-B$) **using** *assms*(*1*) *assms*(*3*) *infinite-super maxc* **by** *blast*
then show *?thesis* **using** ‹*P B*›
proof (*induction* $S-B$ *arbitrary*:*B rule*: *finite-psubset-induct*)
case (*psubset B*)
then show *?case*
proof (*cases maximal B P*)
case *True*
then show *?thesis* **using** *order-refl psubset.hyps* **by** (*metis assms*(*1*) *maxc psubset.prems rev-finite-subset*)
next
case *False*
then obtain B' **where** $B \subset B'$ $P\ B'$ **using** *maximal-def psubset.prems* **by** (*metis dual-order.order-iff-strict*)
then have $B' \subseteq S$ $B \subseteq S$ **using** *maxc* ‹*P B*› **by** *auto*
then have $S - B' \subset S - B$ **using** ‹$B \subset B'$› **by** *blast*
then show *?thesis* **using** *psubset*(*2*)[*OF* ‹$S - B' \subset S - B$› ‹$P\ B'$›] **using** ‹$B \subset B'$› **by** *fast*
qed
qed

qed

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 −
obtain *S* **where** *maximal S* (λT. $T \subseteq$ *set* (*cols A*) $\wedge$ *lin-indpt T*) *U*$\subseteq$*S finite S*
using *maximal-exists-superset*[*of set* (*cols A*) (λT. $T \subseteq$ *set* (*cols A*) $\wedge$ *lin-indpt T*) *U*]
using *List.finite-set assms*(*2*) *assms*(*3*) *maximal-exists-superset* **by** *blast*
then show *?thesis*
unfolding *rank-card-indpt*[*OF* ‹$A \in$ *carrier-mat n nc*› ‹*maximal S* (λT. $T \subseteq$ *set* (*cols A*) $\wedge$ *lin-indpt T*)›]
using *card-mono* **by** *blast*
qed

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 −
have *maximal* (*set* (*cols A*)) (λT. $T \subseteq$ *set* (*cols A*) $\wedge$ *lin-indpt T*)
by (*simp add*: *assms*(*3*) *maximal-def subset-antisym*)
then have *rank A* = *card* (*set* (*cols A*)) **using** *assms*(*1*) *vec-space.rank-card-indpt* **by** *blast*
then show *?thesis* **using** *assms*(*1*) *assms*(*2*) *distinct-card* **by** *fastforce*
qed

lemma (**in** *vec-space*) *rank-le-nc*:
assumes $A \in$ *carrier-mat n nc*
shows *rank A* $\leq$ *nc*
proof −
obtain *S* **where** *maximal S* (λT. $T \subseteq$ *set* (*cols A*) $\wedge$ *lin-indpt T*)
using *maximal-exists*[*of* (λT. $T \subseteq$ *set* (*cols A*) $\wedge$ *lin-indpt T*) *card* (*set* (*cols A*)) {}]
by (*meson List.finite-set card-mono empty-iff empty-subsetI finite-lin-indpt2 rev-finite-subset*)
then have *card S* $\leq$ *card* (*set* (*cols A*)) **by** (*simp add*: *card-mono maximal-def*)
then have *card S* $\leq$ *nc*
using *assms*(*1*) *cols-length card-length carrier-matD*(*2*) **by** (*metis dual-order.trans*)
then show *?thesis*
using *rank-card-indpt*[*OF* ‹$A \in$ *carrier-mat n nc*› ‹*maximal S* (λT. $T \subseteq$ *set* (*cols A*) $\wedge$ *lin-indpt T*)›]
by *simp*
qed

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 −
  have *1*:*set* (*cols A*) ⊆ *carrier-vec n* **using** *assms*(*1*) *cols-dim* **by** *blast*
  have *2*:*finite* (*set* (*cols A*)) **by** *simp*
  have *card* (*set* (*cols A*)) = *nc*
    using *assms*(*1*) *assms*(*3*) *distinct-card* **by** *fastforce*
  have *3*:*vectorspace.dim class-ring* (*span-vs* (*set* (*cols A*))) = *card* (*set* (*cols A*))
    using ‹*rank A* = *nc*›[*unfolded rank-def*]
    using *assms*(*1*) *assms*(*3*) *distinct-card* **by** *fastforce*
  show *?thesis* **using** *full-dim-span*[*OF 1 2 3*] **.**
qed

lemma (**in** *vec-space*) *mat-mult-eq-lincomb*:
assumes $A \in$ *carrier-mat n nc*
assumes *distinct* (*cols A*)
shows $A *_v$ (*vec nc* (λ*i*. *a* (*col A i*))) = *lincomb a* (*set* (*cols A*))
proof (*rule eq-vecI*)
  have *finite* (*set* (*cols A*)) **using** *assms*(*1*) **by** *simp*
  then show *dim-vec* ($A *_v$ (*vec nc* (λ*i*. *a* (*col A i*)))) = *dim-vec* (*lincomb a* (*set* (*cols A*)))
    using *assms cols-dim vec-space.lincomb-dim* **by** (*metis dim-mult-mat-vec carrier-matD*(*1*))
  fix *i* **assume** *i* < *dim-vec* (*lincomb a* (*set* (*cols A*)))
  then have *i* < *n* **using** ‹*dim-vec* ($A *_v$ (*vec nc* (λ*i*. *a* (*col A i*)))) = *dim-vec* (*lincomb a* (*set* (*cols A*)))› *assms* **by** *auto*
  have *set* (*cols A*) ⊆ *carrier-vec n* **using** *cols-dim* ‹$A \in$ *carrier-mat n nc*› *carrier-matD*(*1*) **by** *blast*
  have *bij-betw* (*nth* (*cols A*)) {..<*length* (*cols A*)} (*set* (*cols A*))
    unfolding *bij-betw-def* **by** (*rule conjI*, *simp add*: *inj-on-nth* ‹*distinct* (*cols A*)›;
    metis subset-antisym in-set-conv-nth lessThan-iff rev-image-eqI subsetI
    image-subsetI lessThan-iff nth-mem)
  then have (∑*x*∈*set* (*cols A*). *a x* ∗ *x* $ *i*) =
    (∑*j*∈{..<*length* (*cols A*)}. *a* (*cols A* ! *j*) ∗ (*cols A* ! *j*) $ *i*)
     using *bij-betw-imp-surj-on bij-betw-imp-inj-on* **by** (*metis* (*no-types*, *lifting*) *sum.reindex-cong*)
  also have ... = (∑*j*∈{..<*length* (*cols A*)}. *a* (*col A j*) ∗ (*cols A* ! *j*) $ *i*)
    using *assms*(*1*) *assms*(*2*) *find-first-unique*[*OF* ‹*distinct* (*cols A*)›] ‹*i* < *n*› **by** *auto*
  also have ... = (∑*j*∈{..<*length* (*cols A*)}. (*cols A* ! *j*) $ *i* ∗ *a* (*col A j*)) **by** (*metis mult-commute-abs*)
  also have ... = (∑*j*∈{..<*length* (*cols A*)}. *row A i* $ *j* ∗ *a* (*col A j*)) **using** ‹*i* < *n*› *assms*(*1*) *assms*(*2*) **by** *auto*
  finally show ($A *_v$ (*vec nc* (λ*i*. *a* (*col A i*)))) $ *i* = *lincomb a* (*set* (*cols A*)) $ *i*
    unfolding *lincomb-index*[*OF* ‹*i* < *n*› ‹*set* (*cols A*) ⊆ *carrier-vec n*›]

unfolding *mult-mat-vec-def scalar-prod-def*
using ‹$i < n$› *assms*(*1*) *atLeast0LessThan lessThan-def carrier-matD*(*1*) *index-vec sum.cong* **by** *auto*
qed

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* ($\lambda a.$ *v* \$ *find-first a* (*cols A*)) (*set* (*cols A*)) = ($A *_v v$)
proof −
have $\bigwedge i.\ i < nc \Longrightarrow$ *find-first* (*col A i*) (*cols A*) = *i*
using *assms*(*1*) *assms*(*3*) *find-first-unique* **by** *fastforce*
then have *vec nc* ($\lambda i.$ *v* \$ *find-first* (*col A i*) (*cols A*)) = *v*
using *assms*(*2*) **by** *auto*
then show *?thesis*
using *mat-mult-eq-lincomb*[**where** *a* = ($\lambda a.$ *v* \$ *find-first a* (*cols A*)), *OF assms*(*1*) *assms*(*3*)] **by** *auto*
qed

lemma (**in** *vec-space*) *lin-depI*:
assumes $A \in$ *carrier-mat n nc*
assumes $v \in$ *carrier-vec nc* $v \neq 0_v$ *nc* $A *_v v = 0_v$ *n*
assumes *distinct* (*cols A*)
shows *lin-dep* (*set* (*cols A*))
proof −
have *1*: *finite* (*set* (*cols A*)) **by** *simp*
have *2*: *set* (*cols A*) $\subseteq$ *set* (*cols A*) **by** *auto*
have *3*: ($\lambda a.$ *v* \$ *find-first a* (*cols A*)) $\in$ *set* (*cols A*) $\rightarrow$ *UNIV* **by** *simp*
obtain *i* **where** *v* \$ $i \neq 0$ $i < nc$
using ‹$v \neq 0_v$ *nc*›
by (*metis assms*(*2*) *dim-vec carrier-vecD vec-eq-iff zero-vec-def index-zero-vec*(*1*))
then have $i <$ *dim-col A* **using** *assms*(*1*) **by** *blast*
have *4*:*col A i* $\in$ *set* (*cols A*)
using *cols-nth*[*OF* ‹$i <$ *dim-col A*›] ‹$i <$ *dim-col A*› *in-set-conv-nth* **by** *fastforce*
have *5*:*v* \$ *find-first* (*col A i*) (*cols A*) $\neq 0$
using *find-first-unique*[*OF* ‹*distinct* (*cols A*)›] *cols-nth*[*OF* ‹$i <$ *dim-col A*›] ‹$i < nc$› ‹*v* \$ $i \neq 0$›
assms(*1*) **by** *auto*
have *6*:*lincomb* ($\lambda a.$ *v* \$ *find-first a* (*cols A*)) (*set* (*cols A*)) = 0_v *n*
using *assms*(*1*) *assms*(*2*) *assms*(*4*) *assms*(*5*) *lincomb-eq-mat-mult* **by** *auto*
show *?thesis* **using** *lin-dep-crit*[*OF 1 2 - 4 5 6*] **by** *metis*
qed

lemma (**in** *vec-space*) *lin-depE*:
assumes $A \in$ *carrier-mat n nc*
assumes *lin-dep* (*set* (*cols A*))
assumes *distinct* (*cols A*)
obtains *v* **where** $v \in$ *carrier-vec nc* $v \neq 0_v$ *nc* $A *_v v = 0_v$ *n*

proof −
  have *finite* (*set* (*cols A*)) **by** *simp*
  obtain *a w* **where** $a \in set\ (cols\ A) \to UNIV\ lincomb\ a\ (set\ (cols\ A)) = 0_v\ n\ w \in set\ (cols\ A)\ a\ w \neq 0$
    using *finite-lin-dep*[*OF* ‹*finite* (*set* (*cols A*))› ‹*lin-dep* (*set* (*cols A*))›]
    using *assms*(*1*) *cols-dim carrier-matD*(*1*) **by** *blast*
  define *v* **where** $v = vec\ nc\ (\lambda i.\ a\ (col\ A\ i))$
  have *1*:$v \in carrier\text{-}vec\ nc$ **by** (*simp add*: *v-def*)
  have *2*:$v \neq 0_v\ nc$
  proof −
    obtain *i* **where** $w = col\ A\ i\ i < length\ (cols\ A)$
      by (*metis* ‹$w \in set\ (cols\ A)$› *cols-length cols-nth in-set-conv-nth*)
    have $v\ \$\ i \neq 0$
      unfolding *v-def*
      using ‹$a\ w \neq 0$›[*unfolded* ‹$w = col\ A\ i$›] *index-vec*[*OF* ‹$i < length\ (cols\ A)$›]
      assms(*1*) *cols-length carrier-matD*(*2*) **by** (*metis* (*no-types*) ‹$A \in carrier\text{-}mat\ n\ nc$›
        ‹$\bigwedge f.\ vec\ (length\ (cols\ A))\ f\ \$\ i = f\ i$› ‹$a\ (col\ A\ i) \neq 0$› *cols-length carrier-matD*(*2*))
    then show *?thesis* **using** ‹$i < length\ (cols\ A)$› *assms*(*1*) **by** *auto*
  qed
  have *3*:$A *_v v = 0_v\ n$ **unfolding** *v-def*
    using ‹$lincomb\ a\ (set\ (cols\ A)) = 0_v\ n$› *mat-mult-eq-lincomb*[*OF* ‹$A \in carrier\text{-}mat\ n\ nc$› ‹*distinct* (*cols A*)›] **by** *auto*
  show *thesis* **using** *1 2 3* **by** (*simp add*: *that*)
qed

lemma (**in** *vec-space*) *non-distinct-low-rank*:
assumes $A \in carrier\text{-}mat\ n\ n$
and ¬ *distinct* (*cols A*)
shows $rank\ A < n$
proof −
  obtain *S* **where** *maximal S* ($\lambda T.\ T \subseteq set\ (cols\ A) \wedge lin\text{-}indpt\ T$)
    using *maximal-exists*[*of* ($\lambda T.\ T \subseteq set\ (cols\ A) \wedge lin\text{-}indpt\ T$) *card* (*set* (*cols A*)) {}]
    by (*meson List.finite-set card-mono empty-iff empty-subsetI finite-lin-indpt2 rev-finite-subset*)
  then have $card\ S \leq card\ (set\ (cols\ A))$ **by** (*simp add*: *card-mono maximal-def*)
  then have $card\ S < n$
    using *assms*(*1*) *cols-length card-length* ‹¬ *distinct* (*cols A*)› *card-distinct carrier-matD*(*2*) *nat-less-le*
    by (*metis dual-order.antisym dual-order.trans*)
  then show *?thesis*
    using *rank-card-indpt*[*OF* ‹$A \in carrier\text{-}mat\ n\ n$› ‹*maximal S* ($\lambda T.\ T \subseteq set\ (cols\ A) \wedge lin\text{-}indpt\ T$)›]
    by *simp*
qed

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 $A \in$ *carrier-mat n n*
and *det* $A = 0$
shows *rank* $A < n$
proof (*rule ccontr*)
  assume $\neg$ *rank* $A < n$
  then have *rank* $A = n$ **using** *rank-le-nc assms le-neq-implies-less* **by** *blast*
  obtain v **where** $v \in$ *carrier-vec n* $v \neq 0_v\ n$ $A *_v v = 0_v\ n$
    using *det-0-iff-vec-prod-zero-field*[*OF assms*(*1*)] *assms*(*2*) **by** *blast*
  then show *False*
  proof (*cases distinct* (*cols A*))
    case *True*
    then have *lin-indpt* (*set* (*cols A*)) **using** *full-rank-lin-indpt* **using** ‹*rank* $A = n$› *assms*(*1*) **by** *auto*
    then show *False* **using** *lin-depI*[*OF assms*(*1*) ‹$v \in$ *carrier-vec n*› ‹$v \neq 0_v\ n$› ‹$A *_v v = 0_v\ n$›] *True* **by** *blast*
  next
    case *False*
    then show *False* **using** *non-distinct-low-rank* ‹*rank* $A = n$› ‹$\neg$ *rank* $A < n$› *assms*(*1*) **by** *blast*
  qed
qed

lemma *det-identical-cols*:
  assumes *A*: $A \in$ *carrier-mat n n*
    and *ij*: $i \neq j$
    and *i*: $i < n$ **and** *j*: $j < n$
    and *r*: *col A i* = *col A j*
  shows *det* $A = 0$
  using *det-identical-rows det-transpose*
  by (*metis A i ij j carrier-matD*(*2*) *transpose-carrier-mat r row-transpose*)

lemma (**in** *vec-space*) *low-rank-det-zero*:
assumes $A \in$ *carrier-mat n n*
and *det* $A \neq 0$
shows *rank* $A = n$
proof −
  have *distinct* (*cols A*)
  proof (*rule ccontr*)
    assume $\neg$ *distinct* (*cols A*)
    then obtain *i j* **where** $i \neq j$ (*cols A*) ! *i* = (*cols A*) ! *j* *i*<*length* (*cols A*) *j*<*length* (*cols A*)
      using *distinct-conv-nth* **by** *blast*
    then have *col A i* = *col A j* $i<n$ $j<n$ **using** *assms*(*1*) **by** *auto*
    then have *det* $A = 0$ **using** *det-identical-cols* **using** ‹$i \neq j$› *assms*(*1*) **by** *blast*
    then show *False* **using** ‹*det* $A \neq 0$› **by** *auto*
  qed
  have $\bigwedge v.\ v \in$ *carrier-vec n* $\Longrightarrow v \neq 0_v\ n \Longrightarrow A *_v v \neq 0_v\ n$
    using *det-0-iff-vec-prod-zero-field*[*OF assms*(*1*)] *assms*(*2*) **by** *auto*

then have *lin-indpt* (*set* (*cols A*)) **using** *lin-depE*[*OF assms*(*1*) - ‹*distinct* (*cols A*)›] **by** *auto*
then show *?thesis* **using** *lin-indpt-full-rank*[*OF assms*(*1*) ‹*distinct* (*cols A*)›] **by** *metis*
qed

lemma (**in** *vec-space*) *det-rank-iff*:
assumes $A \in$ *carrier-mat n n*
shows *det* $A \neq 0 \longleftrightarrow$ *rank* $A = n$
using *assms det-zero-low-rank low-rank-det-zero* **by** *force*

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* ($b1 \cup b2$)
assumes $b1 \cup b2 \subseteq$ *carrier M*
assumes $x1 =$ *lincomb a1 b1 a1* $\in$ ($b1 \rightarrow$ *carrier R*)
assumes $x2 =$ *lincomb a2 b2 a2* $\in$ ($b2 \rightarrow$ *carrier R*)
assumes $x = x1 \oplus_M x2$
shows *lincomb* ($\lambda v.$ ($\lambda v.$ *if* $v \in b1$ *then a1 v else* $\mathbf{0}$) $v \oplus$ ($\lambda v.$ *if* $v \in b2$ *then a2 v else* $\mathbf{0}$) v) ($b1 \cup b2$) $= x$
proof −
have *finite* ($b1 \cup (b2-b1)$) *finite* ($b2 \cup (b1-b2)$)
$b1 \cup (b2 - b1) \subseteq$ *carrier M* $b2 \cup (b1-b2) \subseteq$ *carrier M*
$b1 \cap (b2 - b1) = \{\}$ $b2 \cap (b1 - b2) = \{\}$
($\lambda b.\ \mathbf{0}_R$) $\in b2 - b1 \rightarrow$ *carrier R* ($\lambda b.\ \mathbf{0}_R$) $\in b1 - b2 \rightarrow$ *carrier R*
using ‹*finite* ($b1 \cup b2$)› ‹$b1 \cup b2 \subseteq$ *carrier M*› ‹$a2 \in$ ($b2 \rightarrow$ *carrier R*)› **by** *auto*
have *lincomb* ($\lambda b.\ \mathbf{0}_R$) ($b2 - b1$) $= \mathbf{0}_M$ *lincomb* ($\lambda b.\ \mathbf{0}_R$) ($b1 - b2$) $= \mathbf{0}_M$
unfolding *lincomb-def* **using** *M.finsum-all0 assms*(*2*) *lmult-0 subset-iff*
by (*metis* (*no-types*, *lifting*) *Un-Diff-cancel2 inf-sup-aci*(*5*) *le-sup-iff*)+
then have $x1 =$ *lincomb* ($\lambda v.$ *if* $v \in b1$ *then a1 v else* $\mathbf{0}$) ($b1 \cup b2$)
$x2 =$ *lincomb* ($\lambda v.$ *if* $v \in b2$ *then a2 v else* $\mathbf{0}$) ($b1 \cup b2$)
using *lincomb-union2*[*OF* ‹*finite* ($b1 \cup (b2-b1)$)› ‹$b1 \cup (b2 - b1) \subseteq$ *carrier M*› ‹$b1 \cap (b2 - b1) = \{\}$› ‹$a1 \in$ ($b1 \rightarrow$ *carrier R*)› ‹($\lambda b.\ \mathbf{0}_R$) $\in b2 - b1 \rightarrow$ *carrier R*›]
lincomb-union2[*OF* ‹*finite* ($b2 \cup (b1-b2)$)› ‹$b2 \cup (b1-b2) \subseteq$ *carrier M*› ‹$b2 \cap (b1 - b2) = \{\}$› ‹$a2 \in$ ($b2 \rightarrow$ *carrier R*)› ‹($\lambda b.\ \mathbf{0}_R$) $\in b1 - b2 \rightarrow$ *carrier R*›]
using *assms*(*2*) *assms*(*3*) *assms*(*4*) *assms*(*5*) *assms*(*6*) **by** (*simp-all add*: *Un-commute*)
have ($\lambda v.$ *if* $v \in b1$ *then a1 v else* $\mathbf{0}$) $\in$ ($b1 \cup b2$) $\rightarrow$ *carrier R*
($\lambda v.$ *if* $v \in b2$ *then a2 v else* $\mathbf{0}$) $\in$ ($b1 \cup b2$) $\rightarrow$ *carrier R* **using** *assms*(*4*) *assms*(*6*) **by** *auto*
show *lincomb* ($\lambda v.$ ($\lambda v.$ *if* $v \in b1$ *then a1 v else* $\mathbf{0}$) $v \oplus$ ($\lambda v.$ *if* $v \in b2$ *then a2 v else* $\mathbf{0}$) v) ($b1 \cup b2$) $= x$
using *lincomb-sum*[*OF* ‹*finite* ($b1 \cup b2$)› ‹$b1 \cup b2 \subseteq$ *carrier M*›
‹($\lambda v.$ *if* $v \in b1$ *then a1 v else* $\mathbf{0}$) $\in$ ($b1 \cup b2$) $\rightarrow$ *carrier R*› ‹($\lambda v.$ *if* $v \in b2$ *then a2 v else* $\mathbf{0}$) $\in$ ($b1 \cup b2$) $\rightarrow$ *carrier R*›]

‹*x1* = *lincomb* (λ*v*. *if v* ∈ *b1 then a1 v else* **0**) (*b1* ∪ *b2*)› ‹*x2* = *lincomb* (λ*v*. *if v* ∈ *b2 then a2 v else* **0**) (*b1* ∪ *b2*)› *assms*(*7*) **by** *blast*
qed

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 −
have *vectorspace K* (*vs W1*) *vectorspace K* (*vs W2*) *submodule K W1 V submodule K W2 V*
by (*simp add*: ‹*subspace K W1 V*› ‹*subspace K W2 V*› *subspace-is-vs*)+
obtain *b1 b2* **where** *vectorspace.basis K* (*vs W1*) *b1 vectorspace.basis K* (*vs W2*) *b2 finite b1 finite b2*
using *vectorspace.finite-basis-exists*[*OF* ‹*vectorspace K* (*vs W1*)› ‹*vectorspace.fin-dim K* (*vs W1*)›]
using *vectorspace.finite-basis-exists*[*OF* ‹*vectorspace K* (*vs W2*)› ‹*vectorspace.fin-dim K* (*vs W2*)›]
by *blast*
then have *LinearCombinations.module.gen-set K* (*vs W1*) *b1 LinearCombinations.module.gen-set K* (*vs W2*) *b2*
using ‹*vectorspace K* (*vs W1*)› ‹*vectorspace K* (*vs W2*)› *vectorspace.basis-def* **by** *blast*+
then have *span b1* = *W1 span b2* = *W2*
using *module.span-li-not-depend*(*1*) ‹*submodule K W1 V*› ‹*submodule K W2 V*›
‹*vectorspace K* (*vs W1*)› ‹*vectorspace.basis K* (*vs W1*) *b1*› ‹*vectorspace K* (*vs W2*)›
‹*vectorspace.basis K* (*vs W2*) *b2*› *vectorspace.basis-def* **by** *force*+
have *W1* ⊆ *carrier V W2* ⊆ *carrier V* **using** ‹*subspace K W1 V*› ‹*subspace K W2 V*› *subspace-def submodule-def* **by** *metis*+
have *b1* ⊆ *carrier V*
using ‹*vectorspace.basis K* (*vs W1*) *b1*› ‹*vectorspace K* (*vs W1*)› *vectorspace.basis-def* ‹*W1* ⊆ *carrier V*› **by** *fastforce*
have *b2* ⊆ *carrier V*
using ‹*vectorspace.basis K* (*vs W2*) *b2*› ‹*vectorspace K* (*vs W2*)› *vectorspace.basis-def* ‹*W2* ⊆ *carrier V*› **by** *fastforce*
have *finite* (*b1* ∪ *b2*) *b1* ∪ *b2* ⊆ *carrier V*
by (*simp-all add*: ‹*finite b1*› ‹*finite b2*› ‹*b2* ⊆ *carrier V*› ‹*b1* ⊆ *carrier V*›)
have *subspace-sum W1 W2* ⊆ *span* (*b1*∪*b2*)
proof (*rule subsetI*)
fix *x* **assume** *x* ∈ *subspace-sum W1 W2*
obtain *x1 x2* **where** *x1* ∈ *W1 x2* ∈ *W2 x* = *x1* $\oplus_V$ *x2*
using *imageE*[*OF* ‹*x* ∈ *subspace-sum W1 W2*›[*unfolded submodule-sum-def*]]
by (*metis* (*no-types*, *lifting*) *BNF-Def.Collect-case-prodD split-def*)
obtain *a1* **where** *x1* = *lincomb a1 b1 a1*∈ (*b1*→*carrier K*)

using ‹*span b1* = *W1*› *finite-span*[*OF* ‹*finite b1*› ‹*b1* ⊆ *carrier V*›] ‹*x1* ∈ *W1*› **by** *auto*
obtain *a2* **where** *x2* = *lincomb a2 b2 a2*∈ (*b2*→*carrier K*)
using ‹*span b2* = *W2*› *finite-span*[*OF* ‹*finite b2*› ‹*b2* ⊆ *carrier V*›] ‹*x2* ∈ *W2*› **by** *auto*
obtain *a* **where** *x* = *lincomb a* (*b1* ∪ *b2*) **using** *lincomb-add*[*OF* ‹*finite* (*b1* ∪ *b2*)› ‹*b1* ∪ *b2* ⊆ *carrier V*›
‹*x1* = *lincomb a1 b1*› ‹*a1*∈ (*b1*→*carrier K*)› ‹*x2* = *lincomb a2 b2*› ‹*a2*∈ (*b2*→*carrier K*)› ‹$x = x1 \oplus_V x2$›] **by** *blast*
then show *x* ∈ *span* (*b1* ∪ *b2*) **using** *finite-span*[*OF* ‹*finite* (*b1* ∪ *b2*)› ‹(*b1* ∪ *b2*) ⊆ *carrier V*›]
using ‹*b1* ⊆ *carrier V*› ‹*b2* ⊆ *carrier V*› ‹*span b1* = *W1*› ‹*span b2* = *W2*› ‹*x* ∈ *subspace-sum W1 W2*› *span-union-is-sum* **by** *auto*
qed
have *b1* ⊆ *W1 b2* ⊆ *W2*
using ‹*vectorspace K* (*vs W1*)› ‹*vectorspace K* (*vs W2*)› ‹*vectorspace.basis K* (*vs W1*) *b1*›
‹*vectorspace.basis K* (*vs W2*) *b2*› *vectorspace.basis-def local.carrier-vs-is-self* **by** *blast+*
then have *b1*∪*b2* ⊆ *subspace-sum W1 W2* **using** ‹*submodule K W1 V*› ‹*submodule K W2 V*› *in-sum*
by (*metis assms*(*1*) *assms*(*3*) *dual-order.trans sup-least vectorspace.vsum-comm vectorspace-axioms*)
have *subspace-sum W1 W2* = *LinearCombinations.module.span K* (*vs* (*subspace-sum W1 W2*)) (*b1*∪*b2*)
proof (*rule subset-antisym*)
have *submodule K* (*subspace-sum W1 W2*) *V* **by** (*simp add*: ‹*submodule K W1 V*› ‹*submodule K W2 V*› *sum-is-submodule*)
show *subspace-sum W1 W2* ⊆ *LinearCombinations.module.span K* (*vs* (*subspace-sum W1 W2*)) (*b1*∪*b2*)
using *module.span-li-not-depend*(*1*)[*OF* ‹*b1*∪*b2* ⊆ *subspace-sum W1 W2*› ‹*submodule K* (*subspace-sum W1 W2*) *V*›]
by (*simp add*: ‹*subspace-sum W1 W2* ⊆ *span* (*b1* ∪ *b2*)›)
show *subspace-sum W1 W2* ⊇ *LinearCombinations.module.span K* (*vs* (*subspace-sum W1 W2*)) (*b1*∪*b2*)
using ‹*b1*∪*b2* ⊆ *subspace-sum W1 W2*› **by** (*metis* (*full-types*) *LinearCombinations.module.span-is-subset2*
LinearCombinations.module.submodule-is-module ‹*submodule K* (*subspace-sum W1 W2*) *V*› *local.carrier-vs-is-self submodule-def*)
qed
have *vectorspace K* (*vs* (*subspace-sum W1 W2*)) **using** *assms*(*1*) *assms*(*3*) *subspace-def sum-is-subspace vectorspace.subspace-is-vs* **by** *blast*
then have *vectorspace.dim K* (*vs* (*subspace-sum W1 W2*)) ≤ *card* (*b1* ∪ *b2*)
using *vectorspace.gen-ge-dim*[*OF* ‹*vectorspace K* (*vs* (*subspace-sum W1 W2*))› ‹*finite* (*b1* ∪ *b2*)›]
‹*b1* ∪ *b2* ⊆ *subspace-sum W1 W2*›
‹*subspace-sum W1 W2* = *LinearCombinations.module.span K* (*vs* (*subspace-sum W1 W2*)) (*b1* ∪ *b2*)›
local.carrier-vs-is-self **by** *blast*

also have ... ≤ *card b1* + *card b2* **by** (*simp add*: *card-Un-le*)
also have ... = *vectorspace.dim K* (*vs W1*) + *vectorspace.dim K* (*vs W2*)
by (*metis* ‹*finite b1*› ‹*finite b2*› ‹*vectorspace K* (*vs W1*)› ‹*vectorspace K* (*vs W2*)›
‹*vectorspace.basis K* (*vs W1*) *b1*› ‹*vectorspace.basis K* (*vs W2*) *b2*› *vectorspace.dim-basis*)
finally show *?thesis* **by** *auto*
qed

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*)
unfolding *submodule-def*
using ‹*X* ⊆ *W*› *submodule-is-module*[*OF* ‹*submodule R W M*›] **using** ‹*submodule R X M*›[*unfolded submodule-def*] **by** *auto*

lemma (**in** *vectorspace*) *nested-subspaces*:
assumes *subspace K W V*
assumes *subspace K X V*
assumes *X* ⊆ *W*
shows *subspace K X* (*vs W*)
using *assms nested-submodules subspace-def subspace-is-vs* **by** *blast*

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 −
have *vectorspace K* (*vs X*) **using** *assms*(*1*) *subspace-is-vs* **by** *auto*
then obtain *b* **where** *vectorspace.basis K* (*vs X*) *b* **using** *vectorspace.finite-basis-exists*
using *assms*(*3*) **by** *blast*
then have *b* ⊆ *carrier V LinearCombinations.module.lin-indpt K* (*vs X*) *b*
using *vectorspace.basis-def*[*OF* ‹*vectorspace K* (*vs X*)›] ‹*subspace K X V*›[*unfolded subspace-def submodule-def*] **by** *auto*
then have *lin-indpt b*
by (*metis LinearCombinations.module.span-li-not-depend*(*2*) ‹*vectorspace K* (*vs X*)› ‹*vectorspace.basis K* (*vs X*) *b*›
assms(*1*) *is-module local.carrier-vs-is-self submodule-def vectorspace.basis-def*)
show *?thesis* **using** *li-le-dim*(*2*)[*OF* ‹*fin-dim*› ‹*b* ⊆ *carrier V*› ‹*lin-indpt b*›]
using ‹*b* ⊆ *carrier V*› ‹*lin-indpt b*› ‹*vectorspace K* (*vs X*)› ‹*vectorspace.basis K* (*vs X*) *b*› *assms*(*2*)
fin-dim-li-fin vectorspace.dim-basis **by** *fastforce*
qed

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 −
obtain *b1* **where** *finite b1 b1* ⊆ *W1 LinearCombinations.module.gen-set K* (*vs W1*) *b1*
using *assms vectorspace.fin-dim-def subspace-is-vs* **by** *force*
obtain *b2* **where** *finite b2 b2* ⊆ *W2 LinearCombinations.module.gen-set K* (*vs W2*) *b2*
using *assms vectorspace.fin-dim-def subspace-is-vs* **by** *force*
have *1*:*finite* (*b1* ∪ *b2*) **by** (*simp add*: ‹*finite b1*› ‹*finite b2*›)
have *2*:*b1* ∪ *b2* ⊆ *subspace-sum W1 W2*
by (*metis* (*no-types*, *lifting*) ‹*b1* ⊆ *W1*› ‹*b2* ⊆ *W2*› *assms*(*1*) *assms*(*2*)
le-sup-iff subset-Un-eq vectorspace.in-sum-vs vectorspace.vsum-comm vectorspace-axioms)
have *3*:*LinearCombinations.module.gen-set K* (*vs* (*subspace-sum W1 W2*)) (*b1* ∪ *b2*)
proof (*rule subset-antisym*)
have *0*:*LinearCombinations.module.span K* (*vs* (*subspace-sum W1 W2*)) (*b1* ∪ *b2*) = *span* (*b1* ∪ *b2*)
using *span-li-not-depend*(*1*)[*OF* ‹*b1* ∪ *b2* ⊆ *subspace-sum W1 W2*›] *sum-is-subspace*[*OF assms*(*1*) *assms*(*2*)] **by** *auto*
then show *LinearCombinations.module.span K* (*vs* (*subspace-sum W1 W2*)) (*b1* ∪ *b2*) ⊆ *carrier* (*vs* (*subspace-sum W1 W2*))
using ‹*b1* ∪ *b2* ⊆ *subspace-sum W1 W2*› *span-is-subset sum-is-subspace*[*OF assms*(*1*) *assms*(*2*)] **by** *auto*
show *carrier* (*vs* (*subspace-sum W1 W2*)) ⊆ *LinearCombinations.module.span K* (*vs* (*subspace-sum W1 W2*)) (*b1* ∪ *b2*)
unfolding *0*
proof
fix *x* **assume** *assumption*:*x* ∈ *carrier* (*vs* (*subspace-sum W1 W2*))
then have *x*∈*subspace-sum W1 W2* **by** *auto*
then obtain *x1 x2* **where** *x* = *x1* $\oplus_V$ *x2 x1*∈*W1 x2*∈*W2*
using *imageE*[*OF* ‹*x* ∈ *subspace-sum W1 W2*›[*unfolded submodule-sum-def*]]
by (*metis* (*no-types*, *lifting*) *BNF-Def.Collect-case-prodD split-def*)
have *x1*∈*span b1* *x2*∈*span b2*
using ‹*LinearCombinations.module.span K* (*vs W1*) *b1* = *carrier* (*vs W1*)› ‹*b1* ⊆ *W1*› ‹*x1* ∈ *W1*›
‹*LinearCombinations.module.span K* (*vs W2*) *b2* = *carrier* (*vs W2*)› ‹*b2* ⊆ *W2*› ‹*x2* ∈ *W2*›
assms(*1*) *assms*(*2*) *span-li-not-depend*(*1*) **by** *auto*
then have *x1*∈*span* (*b1* ∪ *b2*) *x2*∈*span* (*b1* ∪ *b2*) **by** (*meson le-sup-iff subsetD span-is-monotone subsetI*)+
then show *x* ∈ *span* (*b1* ∪ *b2*) **unfolding** ‹*x* = *x1* $\oplus_V$ *x2*›
by (*meson* ‹*b1* ∪ *b2* ⊆ *subspace-sum W1 W2*› *assms*(*1*) *assms*(*2*) *is-module submodule.subset*
subset-trans sum-is-submodule vectorspace.span-add1 vectorspace-axioms)
qed
qed
show *?thesis* **using** *1 2 3 vectorspace.fin-dim-def*
by (*metis assms*(*1*) *assms*(*2*) *local.carrier-vs-is-self subspace-def sum-is-subspace vectorspace.subspace-is-vs*)
qed

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 −
define *W1* **where** *W1* = *span* (*set* (*cols A*))
define *W2* **where** *W2* = *span* (*set* (*cols B*))
have *set* (*cols* (*A* + *B*)) ⊆ *subspace-sum W1 W2*
proof
fix *x* **assume** *x* ∈ *set* (*cols* (*A* + *B*))
obtain *i* **where** *x* = *col* (*A* + *B*) *i* *i* < *length* (*cols* (*A* + *B*))
using ‹*x* ∈ *set* (*cols* (*A* + *B*))› *nth-find-first cols-nth find-first-le* **by** (*metis cols-length*)
then have *x* = *col A i* + *col B i* **using** ‹*i* < *length* (*cols* (*A* + *B*))› *assms*(*1*) *assms*(*2*) **by** *auto*
have *col A i* ∈ *span* (*set* (*cols A*)) *col B i* ∈ *span* (*set* (*cols B*))
using ‹*i* < *length* (*cols* (*A* + *B*))› *assms*(*1*) *assms*(*2*) *in-set-conv-nth*
by (*metis cols-dim cols-length cols-nth carrier-matD*(*1*) *carrier-matD*(*2*) *index-add-mat*(*3*) *span-mem*)+
then show *x* ∈ *subspace-sum W1 W2*
unfolding *W1-def W2-def* ‹*x* = *col A i* + *col B i*› *submodule-sum-def* **by** *blast*
qed
have *subspace class-ring* (*subspace-sum W1 W2*) *V*
by (*metis W1-def W2-def assms*(*1*) *assms*(*2*) *cols-dim carrier-matD*(*1*) *span-is-submodule subspace-def sum-is-submodule vec-vs*)
then have *span* (*set* (*cols* (*A* + *B*))) ⊆ *subspace-sum W1 W2*
by (*simp add*: ‹*set* (*cols* (*A* + *B*)) ⊆ *subspace-sum W1 W2*› *span-is-subset*)
have *subspace class-ring* (*span* (*set* (*cols* (*A* + *B*)))) *V* **by** (*metis assms*(*2*) *cols-dim add-carrier-mat carrier-matD*(*1*) *span-is-subspace*)
have *subspace*:*subspace class-ring* (*span* (*set* (*cols* (*A* + *B*)))) (*vs* (*subspace-sum W1 W2*))
using *nested-subspaces*[*OF* ‹*subspace class-ring* (*subspace-sum W1 W2*) *V*› ‹*subspace class-ring* (*span* (*set* (*cols* (*A* + *B*)))) *V*›
‹*span* (*set* (*cols* (*A* + *B*))) ⊆ *subspace-sum W1 W2*›] **.**
have *vectorspace.fin-dim class-ring* (*vs W1*) *vectorspace.fin-dim class-ring* (*vs W2*)
subspace class-ring W1 V subspace class-ring W2 V
using *span-is-subspace W1-def W2-def assms*(*1*) *assms*(*2*) *cols-dim carrier-matD fin-dim-span-cols* **by** *auto*
then have *fin-dim*: *vectorspace.fin-dim class-ring* (*vs* (*subspace-sum W1 W2*)) **using** *fin-dim-subspace-sum* **by** *auto*
have *vectorspace.fin-dim class-ring* (*span-vs* (*set* (*cols* (*A* + *B*)))) **using** *assms*(*2*) *add-carrier-mat vec-space.fin-dim-span-cols* **by** *blast*
then have *rank* (*A* + *B*) ≤ *vectorspace.dim class-ring* (*vs* (*subspace-sum W1 W2*)) **unfolding** *rank-def*
using *vectorspace.subspace-dim*[*OF subspace-is-vs*[*OF* ‹*subspace class-ring* (*subspace-sum W1 W2*) *V*›] *subspace fin-dim*] **by** *auto*

also have *vectorspace.dim class-ring* (*vs* (*subspace-sum W1 W2*)) $\leq$ *rank A* + *rank B* **unfolding** *rank-def*
using *W1-def W2-def* ‹*subspace class-ring W1 V*› ‹*subspace class-ring W2 V*› ‹*vectorspace.fin-dim class-ring* (*vs W1*)›
‹*vectorspace.fin-dim class-ring* (*vs W2*)› *subspace-def vectorspace.dim-subadditive* **by** *blast*
finally show *?thesis* **by** *auto*
qed

lemma (**in** *vec-space*) *span-zero*: *span* {*zero V*} = {*zero V*}
by (*metis* (*no-types*, *lifting*) *empty-subsetI in-own-span span-is-submodule span-is-subset span-is-subset2 subset-antisym vectorspace.span-empty vectorspace-axioms*)

lemma (**in** *vec-space*) *dim-zero-vs*: *vectorspace.dim class-ring* (*span-vs* {}) = *0*
proof −
have *vectorspace class-ring* (*span-vs* {}) **using** *field.field-axioms span-is-submodule submodule-is-module vectorspace-def* **by** *auto*
have {} ⊆ *carrier-vec n* ∧ *lin-indpt* {}
by (*metis* (*no-types*) *empty-subsetI fin-dim finite-basis-exists subset-li-is-li vec-vs vectorspace.basis-def*)
then have *vectorspace.basis class-ring* (*span-vs* {}) {} **using** *vectorspace.basis-def*
by (*simp add*: ‹*vectorspace class-ring* (*vs* (*span* {}))› *span-is-submodule span-li-not-depend*(*1*) *span-li-not-depend*(*2*) *vectorspace.basis-def*)
then show *?thesis* **using** ‹*vectorspace class-ring* (*vs* (*span* {}))› *vectorspace.dim-basis* **by** *fastforce*
qed

lemma (**in** *vec-space*) *rank-0I*: *rank* (0_m *n nc*) = *0*
proof −
have *set* (*cols* (0_m *n nc*)) ⊆ {0_v *n*}
by (*metis col-zero cols-length cols-nth in-set-conv-nth insertCI index-zero-mat*(*3*) *subsetI*)
have *set* (*cols* (0_m *n nc*::'*a mat*)) = {} ∨ *set* (*cols* (0_m *n nc*)) = {0_v *n*::'*a vec*}
by (*meson* ‹*set* (*cols* (0_m *n nc*)) ⊆ {0_v *n*}› *subset-singletonD*)
then have *span* (*set* (*cols* (0_m *n nc*))) = {0_v *n*}
by (*metis* (*no-types*) *span-empty span-zero vectorspace.span-empty vectorspace-axioms*)
then show *?thesis* **unfolding** *rank-def* ‹*span* (*set* (*cols* (0_m *n nc*))) = {0_v *n*}›
using *span-empty dim-zero-vs* **by** *simp*
qed

lemma (**in** *vec-space*) *rank-le-1-product-entries*:
fixes *f g*::*nat* ⇒ '*a*
assumes *A* ∈ *carrier-mat n nc*
assumes $\bigwedge$*r c*. *r*<*dim-row A* ⟹ *c*<*dim-col A* ⟹ *A* \$\$ (*r*,*c*) = *f r* ∗ *g c*
shows *rank A* ≤ *1*
proof −
have *set* (*cols A*) ⊆ *span* {*vec n f*}
proof

fix *v* **assume** $v \in$ *set* (*cols A*)
then obtain *c* **where** $c <$ *dim-col A* $v =$ *col A c* **by** (*metis cols-length cols-nth in-set-conv-nth*)
have $g\ c \cdot_v$ *vec n f* $= v$
proof (*rule eq-vecI*)
show *dim-vec* ($g\ c \cdot_v$ *Matrix.vec n f*) $=$ *dim-vec v* **using** ‹$v =$ *col A c*› *assms*(*1*) **by** *auto*
fix *r* **assume** $r <$ *dim-vec v*
then have $r <$ *dim-vec* (*Matrix.vec n f*) **using** ‹*dim-vec* ($g\ c \cdot_v$ *Matrix.vec n f*) $=$ *dim-vec v*› **by** *auto*
then have $r < n$ $r <$ *dim-row A***using** *index-smult-vec*(*2*) ‹$A \in$ *carrier-mat n nc*› **by** *auto*
show ($g\ c \cdot_v$ *Matrix.vec n f*) \$ $r = v$ \$ r
unfolding ‹$v =$ *col A c*› *col-def index-smult-vec*(*1*)[*OF* ‹$r <$ *dim-vec* (*Matrix.vec n f*)›]
index-vec[*OF* ‹$r < n$›] *index-vec*[*OF* ‹$r <$ *dim-row A*›] **by** (*simp add*: ‹$c <$ *dim-col A*› ‹$r <$ *dim-row A*› *assms*(*2*))
qed
then show $v \in$ *span* {*vec n f*} **using** *submodule.smult-closed*[*OF span-is-submodule*]
using *UNIV-I empty-subsetI insert-subset span-self dim-vec module-vec-simps*(*4*) **by** *auto*
qed
have *vectorspace class-ring* (*vs* (*span* {*Matrix.vec n f*})) **using** *span-is-subspace*[*THEN subspace-is-vs*, *of* {*vec n f*}] **by** *auto*
have *submodule class-ring* (*span* {*Matrix.vec n f*}) *V* **by** (*simp add*: *span-is-submodule*)
have *subspace class-ring*(*span* (*set* (*cols A*))) (*vs* (*span* {*Matrix.vec n f*}))
using *vectorspace.span-is-subspace*[*OF* ‹*vectorspace class-ring* (*vs* (*span* {*Matrix.vec n f*}))›, *of set* (*cols A*), *unfolded*
span-li-not-depend(*1*)[*OF* ‹*set* (*cols A*) $\subseteq$ *span* {*vec n f*}› ‹*submodule class-ring* (*span* {*Matrix.vec n f*}) *V*›]]
‹*set* (*cols A*) $\subseteq$ *span* {*vec n f*}› **by** *auto*
have *fin-dim*:*vectorspace.fin-dim class-ring* (*vs* (*span* {*Matrix.vec n f*}))
vectorspace.fin-dim class-ring (*vs* (*span* {*Matrix.vec n f*})(|*carrier* := *span* (*set* (*cols A*))|))
using *fin-dim-span fin-dim-span-cols* ‹$A \in$ *carrier-mat n nc*› **by** *auto*
have *vectorspace.dim class-ring* (*vs* (*span* {*Matrix.vec n f*})) $\leq$ *1*
using *vectorspace.dim-le1I*[*OF* ‹*vectorspace class-ring* (*vs* (*span* {*Matrix.vec n f*}))›]
span-mem span-li-not-depend(*1*)[*OF* - ‹*submodule class-ring* (*span* {*Matrix.vec n f*}) *V*›] **by** *simp*
then show *?thesis* **unfolding** *rank-def* **using** *vectorspace.subspace-dim*[*OF*
‹*vectorspace class-ring* (*vs* (*span* {*Matrix.vec n f*}))› ‹*subspace class-ring* (*span* (*set* (*cols A*))) (*vs* (*span* {*Matrix.vec n f*}))›
fin-dim(*1*) *fin-dim*(*2*)] **by** *simp*
qed

end

34 Missing Lemmas of Sublist

theory *DL-Missing-Sublist*
imports *Main*
begin

lemma *nths-only-one*:
assumes $\{i.\ i < length\ xs \wedge i{\in}I\} = \{j\}$
shows $nths\ xs\ I = [xs!j]$
proof −
  have $set\ (nths\ xs\ I) = \{xs!j\}$
    unfolding *set-nths* **using** *subset-antisym assms* **by** *fastforce*
  moreover have $length\ (nths\ xs\ I) = 1$
    unfolding *length-nths assms* **by** *auto*
  ultimately show *?thesis*
    by (*metis One-nat-def length-0-conv length-Suc-conv the-elem-eq the-elem-set*)
qed

lemma *nths-replicate*:
$nths\ (replicate\ n\ x)\ A = (replicate\ (card\ \{i.\ i < n \wedge i \in A\})\ x)$
proof (*induction n*)
  case *0*
  then show *?case* **by** *simp*
next
  case (*Suc n*)
  then show *?case*
  proof (*cases n∈A*)
    case *True*
    then have $0{:}(if\ 0 \in \{j.\ j + length\ (replicate\ n\ x) \in A\}\ then\ [x]\ else\ []) = [x]$ **by** *simp*
    have $\{i.\ i < Suc\ n \wedge i \in A\} = insert\ n\ \{i.\ i < n \wedge i \in A\}$ **using** *True* **by** *auto*
    have $Suc\ (card\ \{i.\ i < n \wedge i \in A\}) = card\ \{i.\ i < Suc\ n \wedge i \in A\}$
      unfolding ‹$\{i.\ i < Suc\ n \wedge i \in A\} = insert\ n\ \{i.\ i < n \wedge i \in A\}$›
      using *finite-Collect-conjI*[*THEN card-insert-if*] *finite-Collect-less-nat*
       less-irrefl-nat mem-Collect-eq **by** *simp*
    then show *?thesis* **unfolding** *replicate-Suc replicate-append-same*[*symmetric*] *nths-append Suc nths-singleton 0*
      unfolding *replicate-append-same replicate-Suc*[*symmetric*] **by** *simp*
  next
    case *False*
    then have $0{:}(if\ 0 \in \{j.\ j + length\ (replicate\ n\ x) \in A\}\ then\ [x]\ else\ []) = []$ **by** *simp*
    have $\{i.\ i < Suc\ n \wedge i \in A\} = \{i.\ i < n \wedge i \in A\}$ **using** *False* **using** *le-less less-Suc-eq-le* **by** *auto*
    then show *?thesis* **unfolding** *replicate-Suc replicate-append-same*[*symmetric*] *nths-append Suc nths-singleton 0*
      by *simp*
  qed

qed

lemma *length-nths-even*:
assumes *even* (*length xs*)
shows *length* (*nths xs* (*Collect even*)) = *length* (*nths xs* (*Collect odd*))
using *assms* **proof** (*induction length xs div 2 arbitrary*:*xs*)
  case *0*
  then have *length xs* = *0*
    by (*auto elim*: *evenE*)
  then show *?case* **by** *simp*
next
  case (*Suc l xs*)
  then have *length-drop2*: *length* (*nths* (*drop 2 xs*) (*Collect even*)) = *length* (*nths* (*drop 2 xs*) {*a*. *odd a*}) **by** *simp*

  have *length* (*take 2 xs*) = *2* **using** *Suc.hyps*(*2*) **by** *auto*
  then have *plus-odd*: {*j*. *j* + *length* (*take 2 xs*) ∈ *Collect odd*} = *Collect odd* **and**
        plus-even: {*j*. *j* + *length* (*take 2 xs*) ∈ *Collect even*} = *Collect even* **by** *simp-all*
  have *nths-take2*: *nths* (*take 2 xs*) (*Collect even*) = [*take 2 xs* ! *0*] *nths* (*take 2 xs*) (*Collect odd*) = [*take 2 xs* ! *1*]
    using ‹*length* (*take 2 xs*) = *2*› *less-2-cases nths-only-one*[*of take 2 xs Collect even 0*]
    nths-only-one[*of take 2 xs Collect odd 1*]
    by *fastforce*+
  then have *length* (*nths* (*take 2 xs* @ *drop 2 xs*) (*Collect even*))
        = *length* (*nths* (*take 2 xs* @ *drop 2 xs*) {*a*. *odd a*})
    unfolding *nths-append length-append plus-odd plus-even nths-take2 length-drop2*
    by *auto*
  then show *?case* **using** *append-take-drop-id*[*of 2 xs*] **by** *simp*
qed

lemma *nths-map*:
nths (*map f xs*) *A* = *map f* (*nths xs A*)
proof (*induction xs arbitrary*:*A*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons x xs*)
  then show *?case*
  by (*simp add*: *nths-Cons*)
qed

35 Pick

fun *pick* :: *nat set* ⇒ *nat* ⇒ *nat* **where**
pick S 0 = (*LEAST a*. *a*∈*S*) |
pick S (*Suc n*) = (*LEAST a*. *a*∈*S* ∧ *a* > *pick S n*)

lemma *pick-in-set-inf*:
assumes *infinite S*
shows *pick S n* $\in$ *S*
proof (*cases n*)
  show $n = 0 \Longrightarrow$ *pick S n* $\in$ *S*
    unfolding *pick.simps* **using** ‹*infinite S*› *LeastI pick.simps*(*1*) **by** (*metis Collect-mem-eq not-finite-existsD*)
next
  fix n' **assume** $n = Suc\ n'$
  obtain *a* **where** $a{\in}S \wedge a > pick\ S\ n'$ **using** *assms* **by** (*metis bounded-nat-set-is-finite less-Suc-eq nat-neq-iff*)
  show *pick S n* $\in$ *S* **unfolding** ‹$n = Suc\ n'$› *pick.simps*(*2*)
    using *LeastI*[*of* $\lambda a.\ a \in S \wedge pick\ S\ n' < a\ a$, *OF* ‹$a{\in}S \wedge a > pick\ S\ n'$›] **by** *blast*
qed

lemma *pick-mono-inf*:
assumes *infinite S*
shows $m < n \Longrightarrow pick\ S\ m < pick\ S\ n$
using *assms* **proof** (*induction n*)
  case *0*
  then show *?case* **by** *auto*
next
  case (*Suc n*)
  then obtain *a* **where** $a \in S \wedge pick\ S\ n < a$ **by** (*metis bounded-nat-set-is-finite less-Suc-eq nat-neq-iff*)
  then have $pick\ S\ n < pick\ S\ (Suc\ n)$ **unfolding** *pick.simps*
    using *LeastI*[*of* $\lambda a.\ a \in S \wedge pick\ S\ n < a\ a$, *OF* ‹$a{\in}S \wedge a > pick\ S\ n$›] **by** *simp*
  then show *?case* **using** *Suc.IH Suc.prems*(*1*) *assms dual-order.strict-trans less-Suc-eq* **by** *auto*
qed

lemma *pick-eq-iff-inf*:
assumes *infinite S*
shows $x = y \longleftrightarrow pick\ S\ x = pick\ S\ y$
  by (*metis assms nat-neq-iff pick-mono-inf*)

lemma *card-le-pick-inf*:
assumes *infinite S*
and $pick\ S\ n \geq i$
shows $card\ \{a{\in}S.\ a < i\} \leq n$
using *assms* **proof** (*induction n arbitrary*:*i*)
  case *0*
  then show *?case* **unfolding** *pick.simps* **using** *not-less-Least*
   by (*metis* (*no-types*, *lifting*) *Collect-empty-eq card-0-eq card-ge-0-finite dual-order.strict-trans1 leI le-0-eq*)
next
  case (*Suc n*)

then show *?case*
proof −
have $card\ \{a \in S.\ a < pick\ S\ n\} \leq n$ **using** *Suc* **by** *blast*
have $\{a \in S.\ a < i\} \subseteq \{a \in S.\ a < pick\ S\ (Suc\ n)\}$ **using** *Suc.prems(2)* **by** *auto*
have $\{a \in S.\ a < pick\ S\ (Suc\ n)\} = \{a \in S.\ a < pick\ S\ n\} \cup \{pick\ S\ n\}$
apply (*rule subset-antisym*; *rule subsetI*)
using *not-less-Least UnCI mem-Collect-eq nat-neq-iff singleton-conv pick-mono-inf*[*OF Suc.prems*(*1*), *of n Suc n*] *pick-in-set-inf*[*OF Suc.prems*(*1*), *of n*] **by** *fastforce+*
then have $card\ \{a \in S.\ a < i\} \leq card\ \{a \in S.\ a < pick\ S\ n\} + card\ \{pick\ S\ n\}$
using *card-Un-disjoint card-mono*[*OF - ‹*$\{a \in S.\ a < i\} \subseteq \{a \in S.\ a < pick\ S\ (Suc\ n)\}$*›*] **by** *simp*
then show *?thesis* **using** ‹$card\ \{a \in S.\ a < pick\ S\ n\} \leq n$› **by** *auto*
qed
qed

lemma *card-pick-inf*:
assumes *infinite S*
shows $card\ \{a{\in}S.\ a < pick\ S\ n\} = n$
using *assms* **proof** (*induction n*)
case *0*
then show *?case* **unfolding** *pick.simps* **using** *not-less-Least* **by** *auto*
next
case (*Suc n*)
then show $card\ \{a{\in}S.\ a < pick\ S\ (Suc\ n)\} = Suc\ n$
proof −
have $\{a \in S.\ a < pick\ S\ (Suc\ n)\} = \{a \in S.\ a < pick\ S\ n\} \cup \{pick\ S\ n\}$
apply (*rule subset-antisym*; *rule subsetI*)
using *not-less-Least UnCI mem-Collect-eq nat-neq-iff singleton-conv pick-mono-inf*[*OF Suc.prems*, *of n Suc n*] *pick-in-set-inf*[*OF Suc.prems*, *of n*] **by** *fastforce+*
then have $card\ \{a \in S.\ a < pick\ S\ (Suc\ n)\} = card\ \{a \in S.\ a < pick\ S\ n\} + card\ \{pick\ S\ n\}$ **using** *card-Un-disjoint* **by** *auto*
then show *?thesis* **by** (*metis One-nat-def Suc-eq-plus1 Suc card.empty card-insert-if empty-iff finite.emptyI*)
qed
qed

lemma
assumes $n < card\ S$
shows
pick-in-set-le: $pick\ S\ n \in S$ **and**
card-pick-le: $card\ \{a{\in}S.\ a < pick\ S\ n\} = n$ **and**
pick-mono-le: $m < n \Longrightarrow pick\ S\ m < pick\ S\ n$
using *assms* **proof** (*induction n*)
assume $0 < card\ S$
then obtain *x* **where** $x{\in}S$ **by** *fastforce*

then show *pick S 0* $\in$ *S* **unfolding** *pick.simps* **by** (*meson LeastI*)
then show *card* $\{a \in S.\ a < pick\ S\ 0\} = 0$ **using** *not-less-Least* **by** *auto*
show $m < 0 \Longrightarrow pick\ S\ m < pick\ S\ 0$ **by** *auto*
next
fix *n*
assume $n < card\ S \Longrightarrow pick\ S\ n \in S$
and $n < card\ S \Longrightarrow card\ \{a \in S.\ a < pick\ S\ n\} = n$
and $Suc\ n < card\ S$
and $m < n \Longrightarrow n < card\ S \Longrightarrow pick\ S\ m < pick\ S\ n$
then have $card\ \{a \in S.\ a < pick\ S\ n\} = n\ pick\ S\ n \in S$ **by** *linarith+*
have $card\ \{a \in S.\ a > pick\ S\ n\} > 0$
proof −
have $S = \{a \in S.\ a < pick\ S\ n\} \cup \{a \in S.\ a \geq pick\ S\ n\}$ **by** *fastforce*
then have $card\ \{a \in S.\ a \geq pick\ S\ n\} > 1$
using ‹$Suc\ n < card\ S$› ‹$card\ \{a \in S.\ a < pick\ S\ n\} = n$›
card-Un-le[*of* $\{a \in S.\ a < pick\ S\ n\}$ $\{a \in S.\ pick\ S\ n \leq a\}$] **by** *force*
then have $0{:}\{a \in S.\ a \geq pick\ S\ n\} \subseteq \{pick\ S\ n\} \cup \{a \in S.\ a > pick\ S\ n\}$ **by** *auto*
have $1{:}finite\ (\{pick\ S\ n\} \cup \{a \in S.\ pick\ S\ n < a\})$
unfolding *finite-Un* **using** *Collect-mem-eq assms card.infinite conjI* **by** *force*
have $1 < card\ \{pick\ S\ n\} + card\ \{a \in S.\ pick\ S\ n < a\}$
using *card-mono*[*OF 1 0*] *card-Un-le*[*of* $\{pick\ S\ n\}$ $\{a \in S.\ a > pick\ S\ n\}$] ‹$card\ \{a \in S.\ a \geq pick\ S\ n\} > 1$›
by *linarith*
then show *?thesis* **by** *simp*
qed
then show $pick\ S\ (Suc\ n) \in S$ **unfolding** *pick.simps*
by (*metis* (*no-types*, *lifting*) *Collect-empty-eq LeastI card-0-eq card.infinite less-numeral-extra*(*3*))
have $pick\ S\ (Suc\ n) > pick\ S\ n$
by (*metis* (*no-types*, *lifting*) *pick.simps*(*2*) ‹$card\ \{a \in S.\ a > pick\ S\ n\} > 0$› *Collect-empty-eq LeastI card-0-eq card.infinite less-numeral-extra*(*3*))
then show $m < Suc\ n \Longrightarrow pick\ S\ m < pick\ S\ (Suc\ n)$
using ‹$m < n \Longrightarrow n < card\ S \Longrightarrow pick\ S\ m < pick\ S\ n$›
using ‹$Suc\ n < card\ S$› *dual-order.strict-trans less-Suc-eq* **by** *auto*
then show $card\ \{a{\in}S.\ a < pick\ S\ (Suc\ n)\} = Suc\ n$
proof −
have $\{a \in S.\ a < pick\ S\ (Suc\ n)\} = \{a \in S.\ a < pick\ S\ n\} \cup \{pick\ S\ n\}$
apply (*rule subset-antisym*; *rule subsetI*)
using *pick.simps not-less-Least* ‹$pick\ S\ (Suc\ n) > pick\ S\ n$› ‹$pick\ S\ n \in S$› **by** *fastforce+*
then have $card\ \{a \in S.\ a < pick\ S\ (Suc\ n)\} = card\ \{a \in S.\ a < pick\ S\ n\} + card\ \{pick\ S\ n\}$ **using** *card-Un-disjoint* **by** *auto*
then show *?thesis* **by** (*metis One-nat-def Suc-eq-plus1* ‹$card\ \{a \in S.\ a < pick\ S\ n\} = n$› *card.empty card-insert-if empty-iff finite.emptyI*)
qed
qed

lemma *card-le-pick-le*:

assumes *n* < *card S*
and *pick S n* ≥ *i*
shows *card* {*a*∈*S*. *a* < *i*} ≤ *n*
using *assms* **proof** (*induction n arbitrary*:*i*)
  case *0*
  then show *?case* **unfolding** *pick.simps* **using** *not-less-Least*
   by (*metis* (*no-types*, *lifting*) *Collect-empty-eq card-0-eq card-ge-0-finite dual-order.strict-trans1 leI le-0-eq*)
next
  case (*Suc n*)
  have *card* {*a* ∈ *S*. *a* < *pick S n*} ≤ *n* **using** *Suc* **by** (*simp add*: *less-eq-Suc-le nat-less-le*)
  have {*a* ∈ *S*. *a* < *i*} ⊆ {*a* ∈ *S*. *a* < *pick S* (*Suc n*)} **using** *Suc.prems*(*2*) **by** *auto*
  have {*a* ∈ *S*. *a* < *pick S* (*Suc n*)} = {*a* ∈ *S*. *a* < *pick S n*} ∪ {*pick S n*}
   apply (*rule subset-antisym*; *rule subsetI*)
   using *pick.simps not-less-Least pick-mono-le*[*OF Suc.prems*(*1*), *of n*, *OF lessI*] *pick-in-set-le*[*of n S*] *Suc* **by** *fastforce*+
  then have *card* {*a* ∈ *S*. *a* < *i*} ≤ *card* {*a* ∈ *S*. *a* < *pick S n*} + *card* {*pick S n*}
   using *card-Un-disjoint card-mono*[*OF* - ‹{*a* ∈ *S*. *a* < *i*} ⊆ {*a* ∈ *S*. *a* < *pick S* (*Suc n*)}›] **by** *simp*
  then show *?case* **using** ‹*card* {*a* ∈ *S*. *a* < *pick S n*} ≤ *n*› **by** *auto*
qed

lemma
assumes *n* < *card S* ∨ *infinite S*
shows
  pick-in-set:*pick S n* ∈ *S* **and**
  card-le-pick: *i* ≤ *pick S n* ==> *card* {*a*∈*S*. *a* < *i*} ≤ *n* **and**
  card-pick: *card* {*a*∈*S*. *a* < *pick S n*} = *n* **and**
  pick-mono: *m* < *n* ⟹ *pick S m* < *pick S n*
   using *assms pick-in-set-inf pick-in-set-le card-pick-inf card-pick-le card-le-pick-le card-le-pick-inf*
    pick-mono-inf pick-mono-le **by** *auto*

lemma *pick-card*:
pick I (*card* {*a*∈*I*. *a* < *i*}) = (*LEAST a*. *a*∈*I* ∧ *a* ≥ *i*)
proof (*induction i*)
  case *0*
  then show *?case* **by** (*simp add*: *pick-in-set-le*)
next
  case (*Suc i*)
  then show *?case*
  proof (*cases i*∈*I*)
   case *True*
   then have *1*:*pick I* (*card* {*a*∈*I*. *a* < *i*}) = *i* **by** (*metis* (*mono-tags*, *lifting*) *Least-equality Suc.IH order-refl*)
   have {*a* ∈ *I*. *a* < *Suc i*} = {*a* ∈ *I*. *a* < *i*} ∪ {*i*} **using** *True* **by** *auto*
   then have *2*:*card* {*a* ∈ *I*. *a* < *Suc i*} = *Suc* (*card* {*a* ∈ *I*. *a* < *i*}) **by** *auto*

then show *?thesis* **unfolding** *2 pick.simps 1* **using** *Suc-le-eq* **by** *auto*
next
case *False*
then have *1*:$\{a \in I.\ a < Suc\ i\} = \{a \in I.\ a < i\}$ **using** *Collect-cong less-Suc-eq* **by** *auto*
have *2*:$\bigwedge a.\ (a \in I \wedge Suc\ i \leq a) = (a \in I \wedge i \leq a)$ **using** *False Suc-leD le-less-Suc-eq not-le* **by** *blast*
then show *?thesis* **unfolding** *1 2* **using** *Suc.IH* **by** *blast*
qed
qed

lemma *pick-card-in-set*: $i{\in}I \Longrightarrow pick\ I\ (card\ \{a{\in}I.\ a < i\}) = i$
unfolding *pick-card* **using** *Least-equality order-refl* **by** (*metis* (*no-types*, *lifting*))

36 Sublist

lemma *nth-nths-card*:
assumes $j{<}length\ xs$
and $j{\in}J$
shows $nths\ xs\ J\ !\ card\ \{j0.\ j0 < j \wedge j0 \in J\} = xs!j$
using *assms* **proof** (*induction xs rule*:*rev-induct*)
case *Nil*
then show *?case* **using** *gr-implies-not0 list.size*(*3*) **by** *auto*
next
case (*snoc x xs*)
then show *?case*
proof (*cases* $j < length\ xs$)
case *True*
have $\{j0.\ j0 < j \wedge j0 \in J\} \subset \{i.\ i < length\ xs \wedge i \in J\}$
using *True snoc.prems*(*2*) **by** *auto*
then have $card\ \{j0.\ j0 < j \wedge j0 \in J\} < length\ (nths\ xs\ J)$ **unfolding** *length-nths*
using *psubset-card-mono*[*of* $\{i.\ i < length\ xs \wedge i \in J\}$] **by** *simp*
then show *?thesis* **unfolding** *nths-append nth-append* **by** (*simp add*: *True snoc.IH snoc.prems*(*2*))
next
case *False*
then have $length\ xs = j$
using *length-append-singleton less-antisym snoc.prems*(*1*) **by** *auto*
then show *?thesis* **unfolding** *nths-append nth-append length-nths* ‹$length\ xs = j$›
by (*simp add*: *snoc.prems*(*2*))
qed
qed

lemma *pick-reduce-set*:
assumes $i{<}card\ \{a.\ a{<}m \wedge a{\in}I\}$
shows $pick\ I\ i = pick\ \{a.\ a < m \wedge a \in I\}\ i$
using *assms* **proof** (*induction i*)

let *?L* = *LEAST a. a* ∈ {*a. a* < *m* ∧ *a* ∈ *I*}
case *0*
then have {*a. a* < *m* ∧ *a* ∈ *I*} ≠ {} **using** *card.empty less-numeral-extra*(*3*) **by** *fastforce*
then have *?L* ∈ *I ?L* < *m* **by** (*metis* (*mono-tags*, *lifting*) *Collect-empty-eq LeastI mem-Collect-eq*)+
have ⋀*x. x* ∈ {*a. a* < *m* ∧ *a* ∈ *I*} ⟹ *?L* ≤ *x* **by** (*simp add*: *Least-le*)
have ⋀*x. x* ∈ *I* ⟹ *?L* ≤ *x*
by (*metis* (*mono-tags*) ‹*?L* < *m*› ‹⋀*x. x* ∈ {*a. a* < *m* ∧ *a* ∈ *I*} ⟹ *?L* ≤ *x*› *dual-order.strict-trans2 le-cases mem-Collect-eq*)
then show *?case* **unfolding** *pick.simps* **using** *Least-equality*[*of* λ*x. x*∈*I*, *OF* ‹*?L* ∈ *I*›] **by** *blast*
next
case (*Suc i*)
let *?L* = *LEAST x. x* ∈ {*a. a* < *m* ∧ *a* ∈ *I*} ∧ *pick I i* < *x*
have *0*:*pick* {*a. a* < *m* ∧ *a* ∈ *I*} *i* = *pick I i* **using** *Suc-lessD Suc* **by** *linarith*
then have *?L* ∈ {*a. a* < *m* ∧ *a* ∈ *I*} *pick I i* < *?L*
using *LeastI*[*of* λ*a. a* ∈ {*a. a* < *m* ∧ *a* ∈ *I*} ∧ *pick I i* < *a*] **using** *Suc.prems pick-in-set-le pick-mono-le* **by** *fastforce*+
then have *?L* ∈ *I* **by** *blast*
show *?case* **unfolding** *pick.simps 0* **using** *Least-equality*[*of* λ*a. a* ∈ *I* ∧ *pick I i* < *a ?L*]
by (*metis* (*no-types*, *lifting*) *Least-le* ‹*?L* ∈ {*a. a* < *m* ∧ *a* ∈ *I*}› ‹*pick I i* < *?L*› *mem-Collect-eq not-le not-less-iff-gr-or-eq order.trans*)
qed

lemma *nth-nths*:
assumes *i*<*card* {*i. i*<*length xs* ∧ *i*∈*I*}
shows *nths xs I* ! *i* = *xs* ! *pick I i*
proof −
have {*a* ∈ {*i. i* < *length xs* ∧ *i* ∈ *I*}. *a* < *pick* {*i. i* < *length xs* ∧ *i* ∈ *I*} *i*}
= {*a. a* < *pick* {*i. i* < *length xs* ∧ *i* ∈ *I*} *i* ∧ *a* ∈ *I*}
using *assms pick-in-set* **by** *fastforce*
then have *card* {*a. a* < *pick* {*i. i* < *length xs* ∧ *i* ∈ *I*} *i* ∧ *a* ∈ *I*} = *i*
using *card-pick-le*[*OF assms*] **by** *simp*
then have *nths xs I* ! *i* = *xs* ! *pick* {*i. i* < *length xs* ∧ *i* ∈ *I*} *i*
using *nth-nths-card*[**where** *j* = *pick* {*i. i* < *length xs* ∧ *i* ∈ *I*} *i*, *of xs I*]
assms pick-in-set pick-in-set **by** *auto*
then show *?thesis* **using** *pick-reduce-set* **using** *assms* **by** *auto*
qed

lemma *pick-UNIV*: *pick UNIV j* = *j*
by (*induction j*, *simp*, *metis* (*no-types*, *lifting*) *LeastI pick.simps*(*2*) *Suc-mono UNIV-I less-Suc-eq not-less-Least*)

lemma *pick-le*:
assumes *n* < *card* {*a. a* < *i* ∧ *a* ∈ *S*}
shows *pick S n* < *i*
proof −

have *0*:{$a \in \{a.\ a < i \wedge a \in S\}.\ a < i\} = \{a.\ a < i \wedge a \in S\}$ **by** *blast*
show *?thesis* **apply** (*rule ccontr*)
using *card-le-pick-le*[*OF assms*, *unfolded pick-reduce-set*[*OF assms*, *symmetric*], *of i*, *unfolded 0*]
assms not-less not-le **by** *blast*
qed

lemma *prod-list-complementary-nthss*:
fixes f ::$'a \Rightarrow\ 'b$::*comm-monoid-mult*
shows *prod-list* (*map f xs*) = *prod-list* (*map f* (*nths xs A*)) $*$ *prod-list* (*map f* (*nths xs* ($-A$)))
proof (*induction xs rule*:*rev-induct*)
case *Nil*
then show *?case* **by** *simp*
next
case (*snoc x xs*)
show *?case* **unfolding** *map-append prod-list.append nths-append nths-singleton snoc*
by (*cases* (*length xs*)$\in$*A*; *simp*;*metis mult.assoc mult.commute*)
qed

lemma *nths-zip*: *nths* (*zip xs ys*) *I* = *zip* (*nths xs I*) (*nths ys I*)
proof (*rule nth-equalityI*)
show *length* (*nths* (*zip xs ys*) *I*) = *length* (*zip* (*nths xs I*) (*nths ys I*))
proof (*cases length xs* $\leq$ *length ys*)
case *True*
then have $\{i.\ i <$ *length xs* $\wedge\ i \in I\} \subseteq \{i.\ i <$ *length ys* $\wedge\ i \in I\}$ **by** (*simp add*: *Collect-mono less-le-trans*)
then have *card* $\{i.\ i <$ *length xs* $\wedge\ i \in I\} \leq$ *card* $\{i.\ i <$ *length ys* $\wedge\ i \in I\}$
by (*metis* (*mono-tags*, *lifting*) *card-mono finite-nat-set-iff-bounded mem-Collect-eq*)
then show *?thesis* **unfolding** *length-nths length-zip* **using** *True* **using** *min-def* **by** *linarith*
next
case *False*
then have $\{i.\ i <$ *length ys* $\wedge\ i \in I\} \subseteq \{i.\ i <$ *length xs* $\wedge\ i \in I\}$ **by** (*simp add*: *Collect-mono less-le-trans*)
then have *card* $\{i.\ i <$ *length ys* $\wedge\ i \in I\} \leq$ *card* $\{i.\ i <$ *length xs* $\wedge\ i \in I\}$
by (*metis* (*mono-tags*, *lifting*) *card-mono finite-nat-set-iff-bounded mem-Collect-eq*)
then show *?thesis* **unfolding** *length-nths length-zip* **using** *False* **using** *min-def* **by** *linarith*
qed
show *nths* (*zip xs ys*) *I* ! *i* = *zip* (*nths xs I*) (*nths ys I*) ! *i* **if** *i* < *length* (*nths* (*zip xs ys*) *I*) **for** *i*
proof –
have *i* < *length* (*nths xs I*) *i* < *length* (*nths ys I*)
using *that* **by** (*simp-all add*: ‹*length* (*nths* (*zip xs ys*) *I*) = *length* (*zip* (*nths xs I*) (*nths ys I*))›)
show *nths* (*zip xs ys*) *I* ! *i* = *zip* (*nths xs I*) (*nths ys I*) ! *i*
unfolding *nth-nths*[*OF* ‹*i* < *length* (*nths* (*zip xs ys*) *I*)›[*unfolded length-nths*]]

unfolding *nth-zip*[*OF* ‹*i* < *length* (*nths xs I*)› ‹*i* < *length* (*nths ys I*)›]
unfolding *nth-zip*[*OF pick-le*[*OF* ‹*i* < *length* (*nths xs I*)›[*unfolded length-nths*]]
pick-le[*OF* ‹*i* < *length* (*nths ys I*)›[*unfolded length-nths*]]]
by (*metis* (*full-types*) ‹*i* < *length* (*nths xs I*)› ‹*i* < *length* (*nths ys I*)› *length-nths nth-nths*)
qed
qed

37 weave

definition *weave* :: *nat set* ⇒ *'a list* ⇒ *'a list* ⇒ *'a list* **where**
weave A xs ys = *map* (λ*i*. *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*
unfolding *weave-def length-map* **by** *simp*

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 −
have *i* < *length xs* + *length ys* **using** *length-weave* **using** *assms* **by** *metis*
then have *i* < *length* [*0*..<*length xs* + *length ys*] **by** *auto*
then have [*0*..<*length xs* + *length ys*] ! *i* = *i*
by (*metis* ‹*i* < *length xs* + *length ys*› *add.left-neutral nth-upt*)
then show *?thesis*
unfolding *weave-def nth-map*[*OF* ‹*i* < *length* [*0*..<*length xs* + *length ys*]›] **by** *presburger*
qed

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 (*rule nth-equalityI*)
show *length* (*weave A* (*xs* @ [*x*]) *ys*) = *length* (*weave A xs ys* @ [*x*])
unfolding *weave-def length-map* **by** *simp*
show *weave A* (*xs* @ [*x*]) *ys* ! *i* = (*weave A xs ys* @ [*x*]) ! *i*
if *i* < *length* (*weave A* (*xs* @ [*x*]) *ys*) **for** *i*
proof −
show *weave A* (*xs* @ [*x*]) *ys* ! *i* = (*weave A xs ys* @ [*x*]) ! *i*
proof (*cases i* = *length xs* + *length ys*)
case *True*
then have (*weave A xs ys* @ [*x*]) ! *i* = *x* **using** *length-weave* **by** (*metis nth-append-length*)
have *card* {*a* ∈ *A*. *a* < *i*} = *length xs* **using** *assms*(*2*) *True* **by** *auto*
then show *?thesis* **unfolding** *nth-weave*[*OF* ‹*i* < *length* (*weave A* (*xs* @ [*x*])

ys)›]
‹(*weave A xs ys* @ [*x*]) ! $i = x$› **using** *True assms*(*1*) **by** *simp*
next
case *False*
have $i <$ *length* (*weave A xs ys*) **using** ‹$i <$ *length* (*weave A* (*xs* @ [*x*]) *ys*)›
‹*length* (*weave A* (*xs* @ [*x*]) *ys*) = *length* (*weave A xs ys* @ [*x*])› *length-append-singleton*
length-weave less-antisym False **by** *fastforce*
then have (*weave A xs ys* @ [*x*]) ! i = (*weave A xs ys*) ! i **by** (*simp add*: *nth-append*)
{
assume $i \in A$
have $i <$*length xs* + *length ys* **by** (*metis* ‹$i <$ *length* (*weave A xs ys*)› *length-weave*)
then have $\{a \in A.\ a < i\} \subset \{a \in A.\ a <$ *length xs* + *length ys*$\}$
using *assms*(*1*) ‹$i<$*length xs* + *length ys*› ‹$i \in A$› **by** *auto*
then have *card* $\{a \in A.\ a < i\} <$ *card* $\{a \in A.\ a <$ *length xs* + *length ys*$\}$
using *psubset-card-mono*[*of* $\{a \in A.\ a <$ *length xs* + *length ys*$\}$ $\{a \in A.\ a < i\}$] **by** *simp*
then have (*xs* @ [*x*]) ! *card* $\{a \in A.\ a < i\}$ = *xs* ! *card* $\{a \in A.\ a < i\}$
by (*metis* (*no-types*, *lifting*) *assms*(*2*) *nth-append*)
}
then show *?thesis* **unfolding** *nth-weave*[*OF* ‹$i <$ *length* (*weave A* (*xs* @ [*x*]) *ys*)›]
‹(*weave A xs ys* @ [*x*]) ! i = (*weave A xs ys*) ! i› *nth-weave*[*OF* ‹$i <$ *length* (*weave A xs ys*)›]
by *simp*
qed
qed
qed

lemma *weave-append2*:
assumes *length xs* + *length ys* $\notin A$
assumes *length ys* = *card* $\{a \in -A.\ a <$ *length xs* + *length ys*$\}$
shows *weave A xs* (*ys* @ [*y*]) = *weave A xs ys* @ [*y*]
proof (*rule nth-equalityI*)
show *length* (*weave A xs* (*ys* @ [*y*])) = *length* (*weave A xs ys* @ [*y*])
unfolding *weave-def length-map* **by** *simp*
show *weave A xs* (*ys* @ [*y*]) ! i = (*weave A xs ys* @ [*y*]) ! i **if** $i <$ *length* (*weave A xs* (*ys* @ [*y*])) **for** i
proof −
show *weave A xs* (*ys* @ [*y*]) ! i = (*weave A xs ys* @ [*y*]) ! i
proof (*cases* i = *length xs* + *length ys*)
case *True*
then have (*weave A xs ys* @ [*y*]) ! $i = y$ **using** *length-weave* **by** (*metis nth-append-length*)
have *card* $\{a \in -A.\ a < i\}$ = *length ys* **using** *assms*(*2*) *True* **by** *auto*
then show *?thesis* **unfolding** *nth-weave*[*OF* ‹$i <$ *length* (*weave A xs* (*ys* @ [*y*]))›]
‹(*weave A xs ys* @ [*y*]) ! $i = y$› **using** *True assms*(*1*) **by** *simp*

next
case *False*
have $i < length\ (weave\ A\ xs\ ys)$ **using** ‹$i < length\ (weave\ A\ xs\ (ys\ @\ [y]))$›
‹$length\ (weave\ A\ xs\ (ys\ @\ [y])) = length\ (weave\ A\ xs\ ys\ @\ [y])$› *length-append-singleton*
length-weave less-antisym False **by** *fastforce*
then have $(weave\ A\ xs\ ys\ @\ [y])\ !\ i = (weave\ A\ xs\ ys)\ !\ i$ **by** (*simp add*: *nth-append*)
{
assume $i \notin A$
have $i < length\ xs + length\ ys$ **by** (*metis* ‹$i < length\ (weave\ A\ xs\ ys)$› *length-weave*)
then have $\{a \in -A.\ a < i\} \subset \{a \in -A.\ a < length\ xs + length\ ys\}$
using *assms*(*1*) ‹$i < length\ xs + length\ ys$› ‹$i \notin A$› **by** *auto*
then have $card\ \{a \in -A.\ a < i\} < card\ \{a \in -A.\ a < length\ xs + length\ ys\}$
using *psubset-card-mono*[*of* $\{a \in -A.\ a < length\ xs + length\ ys\}$ $\{a \in -A.\ a < i\}$] **by** *simp*
then have $(ys\ @\ [y])\ !\ card\ \{a \in -A.\ a < i\} = ys\ !\ card\ \{a \in -A.\ a < i\}$
by (*metis* (*no-types*, *lifting*) *assms*(*2*) *nth-append*)
}
then show *?thesis* **unfolding** *nth-weave*[*OF* ‹$i < length\ (weave\ A\ xs\ (ys\ @\ [y]))$›]
‹$(weave\ A\ xs\ ys\ @\ [y])\ !\ i = (weave\ A\ xs\ ys)\ !\ i$› *nth-weave*[*OF* ‹$i < length\ (weave\ A\ xs\ ys)$›]
by *simp*
qed
qed
qed

lemma *nths-nth*:
assumes $n \in A$ $n < length\ xs$
shows $nths\ xs\ A\ !\ (card\ \{i.\ i<n \wedge i \in A\}) = xs\ !\ n$
using *assms* **proof** (*induction xs rule*:*rev-induct*)
case (*snoc x xs*)
then show *?case*
proof (*cases* $n = length\ xs$)
case *True*
then show *?thesis* **unfolding** *nths-append*[*of xs* [*x*] *A*] *nth-append*
using *length-nths*[*of xs A*] *nths-singleton snoc.prems*(*1*) **by** *auto*
next
case *False*
then have $n < length\ xs$ **using** *snoc* **by** *auto*
then have $0{:}nths\ xs\ A\ !\ card\ \{i.\ i < n \wedge i \in A\} = xs\ !\ n$ **using** *snoc* **by** *auto*

have $\{i.\ i < n \wedge i \in A\} \subset \{i.\ i < length\ xs \wedge i \in A\}$ **using** ‹$n < length\ xs$› *snoc* **by** *force*
then have $card\ \{i.\ i < n \wedge i \in A\} < length\ (nths\ xs\ A)$ **unfolding** *length-nths*
by (*simp add*: *psubset-card-mono*)
then show *?thesis* **unfolding** *nths-append*[*of xs* [*x*] *A*] *nth-append* **using** *0*

by (*simp add*: ‹*n* < *length xs*›)
qed
qed *simp*

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 −
have *length xs* = *length ys*
proof (*rule ccontr*; *cases length xs* < *length ys*)
case *True*
then show *False*
proof (*cases length xs* ∈ *A*)
case *False*
have {*i*. *i* < *length xs* ∧ *i* ∈ − *A*} ⊂ {*i*. *i* < *length ys* ∧ *i* ∈ − *A*}
using *False* ‹*length xs* < *length ys*› **by** *force*
then have *length* (*nths ys* (−*A*)) > *length* (*nths xs* (−*A*))
unfolding *length-nths* **by** (*simp add*: *psubset-card-mono*)
then show *False* **using** *assms*(*2*) *list-all2-lengthD not-less-iff-gr-or-eq* **by** *blast*
next
case *True*
have {*i*. *i* < *length xs* ∧ *i* ∈ *A*} ⊂ {*i*. *i* < *length ys* ∧ *i* ∈ *A*}
using *True* ‹*length xs* < *length ys*› **by** *force*
then have *length* (*nths ys A*) > *length* (*nths xs A*)
unfolding *length-nths* **by** (*simp add*: *psubset-card-mono*)
then show *False* **using** *assms*(*1*) *list-all2-lengthD not-less-iff-gr-or-eq* **by** *blast*
qed
next
assume *length xs* ≠ *length ys*
case *False*
then have *length xs* > *length ys* **using** ‹*length xs* ≠ *length ys*› **by** *auto*
then show *False*
proof (*cases length ys* ∈ *A*)
case *False*
have {*i*. *i* < *length ys* ∧ *i* ∈ −*A*} ⊂ {*i*. *i* < *length xs* ∧ *i* ∈ −*A*}
using *False* ‹*length xs* > *length ys*› **by** *force*
then have *length* (*nths xs* (−*A*)) > *length* (*nths ys* (−*A*))
unfolding *length-nths* **by** (*simp add*: *psubset-card-mono*)
then show *False* **using** *assms*(*2*) *list-all2-lengthD dual-order.strict-implies-not-eq* **by** *blast*
next
case *True*
have {*i*. *i* < *length ys* ∧ *i* ∈ *A*} ⊂ {*i*. *i* < *length xs* ∧ *i* ∈ *A*}
using *True* ‹*length xs* > *length ys*› **by** *force*
then have *length* (*nths xs A*) > *length* (*nths ys A*)
unfolding *length-nths* **by** (*simp add*: *psubset-card-mono*)

then show *False* **using** *assms*(*1*) *list-all2-lengthD dual-order.strict-implies-not-eq*
by *blast*
qed
qed

have $\bigwedge n.\ n < length\ xs \Longrightarrow P\ (xs\ !\ n)\ (ys\ !\ n)$
proof −
fix *n* **assume** $n < length\ xs$
then have $n < length\ ys$ **using** ‹$length\ xs = length\ ys$› **by** *auto*
then show $P\ (xs\ !\ n)\ (ys\ !\ n)$
proof (*cases* $n \in A$)
case *True*
have $\{i.\ i < n \wedge i \in A\} \subset \{i.\ i < length\ xs \wedge i \in A\}$ **using** ‹$n < length\ xs$› ‹$n \in A$› **by** *force*
then have $card\ \{i.\ i < n \wedge i \in A\} < length\ (nths\ xs\ A)$ **unfolding** *length-nths*
by (*simp add*: *psubset-card-mono*)
show *?thesis* **using** *nths-nth*[*OF* ‹$n \in A$› ‹$n < length\ xs$›] *nths-nth*[*OF* ‹$n \in A$› ‹$n < length\ ys$›]
list-all2-nthD[*OF assms*(*1*), *of card* $\{i.\ i < n \wedge i \in A\}$] *length-nths*
by (*simp add*: ‹$card\ \{i.\ i < n \wedge i \in A\} < length\ (nths\ xs\ A)$›)
next
case *False* **then have** $n \in -A$ **by** *auto*
have $\{i.\ i < n \wedge i \in -A\} \subset \{i.\ i < length\ xs \wedge i \in -A\}$ **using** ‹$n < length\ xs$› ‹$n \in -A$› **by** *force*
then have $card\ \{i.\ i < n \wedge i \in -A\} < length\ (nths\ xs\ (-A))$ **unfolding** *length-nths*
by (*simp add*: *psubset-card-mono*)
show *?thesis* **using** *nths-nth*[*OF* ‹$n \in -A$› ‹$n < length\ xs$›] *nths-nth*[*OF* ‹$n \in -A$› ‹$n < length\ ys$›]
list-all2-nthD[*OF assms*(*2*), *of card* $\{i.\ i < n \wedge i \in -A\}$] *length-nths*
using ‹$card\ \{i.\ i < n \wedge i \in -\ A\} < length\ (nths\ xs\ (-\ A))$› **by** *auto* **next**
qed
qed
then show *?thesis* **using** ‹$length\ xs = length\ ys$› *list-all2-all-nthI* **by** *blast*
qed

lemma *nths-weave*:
assumes $length\ xs = card\ \{a \in A.\ a < length\ xs + length\ ys\}$
assumes $length\ ys = card\ \{a \in (-A).\ a < length\ xs + length\ ys\}$
shows $nths\ (weave\ A\ xs\ ys)\ A = xs \wedge nths\ (weave\ A\ xs\ ys)\ (-A) = ys$
using *assms* **proof** (*induction length xs* + *length ys arbitrary*: *xs ys*)
case *0*
then show *?case*
unfolding *weave-def nths-map* **by** *simp*
next
case (*Suc l*)
then show *?case*
proof (*cases* $l \in A$)
case *True*

then have $l \in \{a \in A.\ a < \mathit{length}\ \mathit{xs} + \mathit{length}\ \mathit{ys}\}$ **using** *Suc.hyps mem-Collect-eq zero-less-Suc* **by** *auto*
then have $\mathit{length}\ \mathit{xs} > 0$ **using** *Suc* **by** *fastforce*
then obtain $\mathit{xs}'\ x$ **where** $\mathit{xs} = \mathit{xs}'\ @\ [x]$ **by** (*metis append-butlast-last-id length-greater-0-conv*)
then have $l = \mathit{length}\ \mathit{xs}' + \mathit{length}\ \mathit{ys}$ **using** *Suc.hyps* **by** *simp*
have *length-xs'*:$\mathit{length}\ \mathit{xs}' = \mathit{card}\ \{a \in A.\ a < \mathit{length}\ \mathit{xs}' + \mathit{length}\ \mathit{ys}\}$
proof −
have $\{a \in A.\ a < \mathit{length}\ \mathit{xs} + \mathit{length}\ \mathit{ys}\} = \{a \in A.\ a < \mathit{length}\ \mathit{xs}' + \mathit{length}\ \mathit{ys}\} \cup \{l\}$
using ‹$\mathit{xs} = \mathit{xs}'\ @\ [x]$› ‹$l \in \{a \in A.\ a < \mathit{length}\ \mathit{xs} + \mathit{length}\ \mathit{ys}\}$› ‹$l = \mathit{length}\ \mathit{xs}' + \mathit{length}\ \mathit{ys}$›
by *force*
then have $\mathit{card}\ \{a \in A.\ a < \mathit{length}\ \mathit{xs} + \mathit{length}\ \mathit{ys}\} = \mathit{card}\ \{a \in A.\ a < \mathit{length}\ \mathit{xs}' + \mathit{length}\ \mathit{ys}\} + 1$
using ‹$l = \mathit{length}\ \mathit{xs}' + \mathit{length}\ \mathit{ys}$› **by** *fastforce*
then show *?thesis* **by** (*metis One-nat-def Suc.prems*(*1*) ‹$\mathit{xs} = \mathit{xs}'\ @\ [x]$› *add-right-imp-eq*
length-Cons length-append list.size(*3*))
qed
have *length-ys*:$\mathit{length}\ \mathit{ys} = \mathit{card}\ \{a \in -\ A.\ a < \mathit{length}\ \mathit{xs}' + \mathit{length}\ \mathit{ys}\}$
proof −
have $l \notin \{a \in -\ A.\ a < \mathit{length}\ \mathit{xs} + \mathit{length}\ \mathit{ys}\}$ **using** ‹$l \in A$› ‹$l = \mathit{length}\ \mathit{xs}' + \mathit{length}\ \mathit{ys}$› **by** *blast*
have $\{a \in -A.\ a < \mathit{length}\ \mathit{xs} + \mathit{length}\ \mathit{ys}\} = \{a \in -A.\ a < \mathit{length}\ \mathit{xs}' + \mathit{length}\ \mathit{ys}\}$
apply (*rule subset-antisym*)
using ‹$l = \mathit{length}\ \mathit{xs}' + \mathit{length}\ \mathit{ys}$› ‹$\mathit{Suc}\ l = \mathit{length}\ \mathit{xs} + \mathit{length}\ \mathit{ys}$› ‹$l \notin \{a \in -\ A.\ a < \mathit{length}\ \mathit{xs} + \mathit{length}\ \mathit{ys}\}$›
apply (*metis* (*no-types*, *lifting*) *Collect-mono less-Suc-eq mem-Collect-eq*)
using *Collect-mono Suc.hyps*(*2*) ‹$l = \mathit{length}\ \mathit{xs}' + \mathit{length}\ \mathit{ys}$› **by** *auto*
then show *?thesis* **using** *Suc.prems*(*2*) **by** *auto*
qed
have $\mathit{length}\ \mathit{xs}' + \mathit{length}\ \mathit{ys} \in A$ **using** ‹$l \in A$› ‹$l = \mathit{length}\ \mathit{xs}' + \mathit{length}\ \mathit{ys}$› **by** *blast*

then have $\mathit{nths}\ (\mathit{weave}\ A\ \mathit{xs}\ \mathit{ys})\ A = \mathit{nths}\ (\mathit{weave}\ A\ \mathit{xs}'\ \mathit{ys}\ @\ [x])\ A$ **unfolding** ‹$\mathit{xs} = \mathit{xs}'\ @\ [x]$› **using** *weave-append1*[*OF* ‹$\mathit{length}\ \mathit{xs}' + \mathit{length}\ \mathit{ys} \in A$› *length-xs'*] **by** *metis*
also have ... $= \mathit{nths}\ (\mathit{weave}\ A\ \mathit{xs}'\ \mathit{ys})\ A\ @\ \mathit{nths}\ [x]\ \{a.\ a + (\mathit{length}\ \mathit{xs}' + \mathit{length}\ \mathit{ys}) \in A\}$
using *nths-append length-weave* **by** *metis*
also have ... $= \mathit{nths}\ (\mathit{weave}\ A\ \mathit{xs}'\ \mathit{ys})\ A\ @\ [x]$
using *nths-singleton* ‹$\mathit{length}\ \mathit{xs}' + \mathit{length}\ \mathit{ys} \in A$› **by** *auto*
also have ... $= \mathit{xs}$ **using** *Suc.hyps*(*1*)[*OF* ‹$l = \mathit{length}\ \mathit{xs}' + \mathit{length}\ \mathit{ys}$› *length-xs' length-ys*]
‹$\mathit{xs} = \mathit{xs}'\ @\ [x]$› **by** *presburger*
finally have $\mathit{nths}\ (\mathit{weave}\ A\ \mathit{xs}\ \mathit{ys})\ A = \mathit{xs}$ **by** *metis*

have *nths* (*weave A xs ys*) ($-A$) = *nths* (*weave A xs' ys* @ [*x*]) ($-A$) **unfolding** ‹*xs* = *xs'* @ [*x*]› **using** *weave-append1*[*OF* ‹*length xs'* + *length ys* ∈ *A*› *length-xs'*] **by** *metis*
also have ... = *nths* (*weave A xs' ys*) ($-A$) @ *nths* [*x*] {*a*. *a* + (*length xs'* + *length ys*) ∈ ($-A$)}
using *nths-append length-weave* **by** *metis*
also have ... = *nths* (*weave A xs' ys*) ($-A$)
using *nths-singleton* ‹*length xs'* + *length ys* ∈ *A*› **by** *auto*
also have ... = *ys*
using *Suc.hyps*(*1*)[*OF* ‹*l* = *length xs'* + *length ys*› *length-xs' length-ys*] **by** *presburger*
finally show *?thesis* **using** ‹*nths* (*weave A xs ys*) *A* = *xs*› **by** *auto*
next
case *False*
then have *l*∉{*a* ∈ *A*. *a* < *length xs* + *length ys*} **using** *Suc.hyps mem-Collect-eq zero-less-Suc* **by** *auto*
then have *length ys* > *0* **using** *Suc* **by** *fastforce*
then obtain *ys' y* **where** *ys* = *ys'* @ [*y*] **by** (*metis append-butlast-last-id length-greater-0-conv*)
then have *l* = *length xs* + *length ys'* **using** *Suc.hyps* **by** *simp*
have *length-ys'*:*length ys'* = *card* {*a* ∈ $-A$. *a* < *length xs* + *length ys'*}
proof −
have {*a* ∈ $-A$. *a* < *length xs* + *length ys*} = {*a* ∈ $-A$. *a* < *length xs* + *length ys'*} ∪ {*l*}
using ‹*ys* = *ys'* @ [*y*]› ‹*l*∉{*a* ∈ *A*. *a* < *length xs* + *length ys*}› ‹*l* = *length xs* + *length ys'*›
by *force*
then have *card* {*a* ∈ $-A$. *a* < *length xs* + *length ys*} = *card* {*a* ∈ $-A$. *a* < *length xs* + *length ys'*} + *1*
using ‹*l* = *length xs* + *length ys'*› **by** *fastforce*
then show *?thesis* **by** (*metis One-nat-def Suc.prems*(*2*) ‹*ys* = *ys'* @ [*y*]› *add-right-imp-eq*
length-Cons length-append list.size(*3*))
qed
have *length-xs*:*length xs* = *card* {*a* ∈ *A*. *a* < *length xs* + *length ys'*}
proof −
have *l*∉{*a* ∈ *A*. *a* < *length xs* + *length ys*} **using** ‹*l*∉*A*› ‹*l* = *length xs* + *length ys'*› **by** *blast*
have {*a* ∈ *A*. *a* < *length xs* + *length ys*} = {*a* ∈ *A*. *a* < *length xs* + *length ys'*}
apply (*rule subset-antisym*)
using ‹*l* = *length xs* + *length ys'*› ‹*Suc l* = *length xs* + *length ys*› ‹*l*∉{*a* ∈ *A*. *a* < *length xs* + *length ys*}›
apply (*metis* (*no-types*, *lifting*) *Collect-mono less-Suc-eq mem-Collect-eq*)
using *Collect-mono Suc.hyps*(*2*) ‹*l* = *length xs* + *length ys'*› **by** *auto*
then show *?thesis* **using** *Suc.prems*(*1*) **by** *auto*
qed
have *length xs* + *length ys'* ∉ *A* **using** ‹*l*∉*A*› ‹*l* = *length xs* + *length ys'*› **by** *blast*

then have *nths* (*weave A xs ys*) *A* = *nths* (*weave A xs ys'* @ [*y*]) *A* **unfolding** ‹*ys* = *ys'* @ [*y*]› **using** *weave-append2*[*OF* ‹*length xs* + *length ys'* ∉ *A*› *length-ys'*] **by** *metis*
also have ... = *nths* (*weave A xs ys'*) *A* @ *nths* [*y*] {*a*. *a* + (*length xs* + *length ys'*) ∈ *A*}
using *nths-append length-weave* **by** *metis*
also have ... = *nths* (*weave A xs ys'*) *A*
using *nths-singleton* ‹*length xs* + *length ys'* ∉ *A*› **by** *auto*
also have ... = *xs*
using *Suc.hyps*(*1*)[*OF* ‹*l* = *length xs* + *length ys'*› *length-xs length-ys'*] **by** *auto*
finally have *nths* (*weave A xs ys*) *A* = *xs* **by** *auto*

have *nths* (*weave A xs ys*) (−*A*) = *nths* (*weave A xs ys'* @ [*y*]) (−*A*) **unfolding** ‹*ys* = *ys'* @ [*y*]› **using** *weave-append2*[*OF* ‹*length xs* + *length ys'* ∉ *A*› *length-ys'*] **by** *metis*
also have ... = *nths* (*weave A xs ys'*) (−*A*) @ *nths* [*y*] {*a*. *a* + (*length xs* + *length ys'*) ∈ (−*A*)}
using *nths-append length-weave* **by** *metis*
also have ... = *nths* (*weave A xs ys'*) (−*A*) @ [*y*]
using *nths-singleton* ‹*length xs* + *length ys'* ∉ *A*› **by** *auto*
also have ... = *ys*
using *Suc.hyps*(*1*)[*OF* ‹*l* = *length xs* + *length ys'*› *length-xs length-ys'*] ‹*ys* = *ys'* @ [*y*]› **by** *simp*
finally show *?thesis* **using** ‹*nths* (*weave A xs ys*) *A* = *xs*› **by** *auto*
qed
qed

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
show *set* (*weave A xs ys*) ⊆ *set xs* ∪ *set ys*
proof
fix *x* **assume** *x*∈*set* (*weave A xs ys*)
then obtain *i* **where** *weave A xs ys* ! *i* = *x* *i*<*length* (*weave A xs ys*) **by** (*meson in-set-conv-nth*)
show *x* ∈ *set xs* ∪ *set ys*
proof (*cases i*∈*A*)
case *True*
then have *i* ∈ {*a*∈*A*. *a* < *length xs* + *length ys*} **unfolding** *length-weave*
by (*metis* ‹*i* < *length* (*weave A xs ys*)› *length-weave mem-Collect-eq*)
then have {*a* ∈ *A*. *a* < *i*} ⊂ {*a*∈*A*. *a* < *length xs* + *length ys*}
using *Collect-mono* ‹*i* < *length* (*weave A xs ys*)›[*unfolded length-weave*] *le-Suc-ex less-imp-le-nat trans-less-add1*
le-neq-trans less-irrefl mem-Collect-eq **by** *auto*
then have *card* {*a* ∈ *A*. *a* < *i*} < *card* {*a*∈*A*. *a* < *length xs* + *length ys*} **by**

(*simp add*: *psubset-card-mono*)
then show $x \in$ *set xs* $\cup$ *set ys*
unfolding *nth-weave*[*OF* ‹*i*<*length* (*weave A xs ys*)›, *unfolded* ‹*weave A xs ys* ! $i = x$›] **using** *True*
using *UnI1 assms*(*1*) *nth-mem* **by** *auto*
next
case *False*
have $i \notin A \Longrightarrow i \in \{a \in -A.\ a <$ *length xs* + *length ys*$\}$ **unfolding** *length-weave*
by (*metis ComplI* ‹*i* < *length* (*weave A xs ys*)› *length-weave mem-Collect-eq*)
then have $\{a \in -A.\ a < i\} \subset \{a \in -A.\ a <$ *length xs* + *length ys*$\}$
using *Collect-mono* ‹*i* < *length* (*weave A xs ys*)›[*unfolded length-weave*] *le-Suc-ex less-imp-le-nat trans-less-add1*
le-neq-trans less-irrefl mem-Collect-eq **using** *False* **by** *auto*
then have *card* $\{a \in -A.\ a < i\} <$ *card* $\{a \in -A.\ a <$ *length xs* + *length ys*$\}$ **by** (*simp add*: *psubset-card-mono*)
then show $x \in$ *set xs* $\cup$ *set ys*
unfolding *nth-weave*[*OF* ‹*i*<*length* (*weave A xs ys*)›, *unfolded* ‹*weave A xs ys* ! $i = x$›] **using** *False*
using *UnI1 assms*(*2*) *nth-mem* **by** *auto*
qed
qed
show *set xs* $\cup$ *set ys* $\subseteq$ *set* (*weave A xs ys*)
using *nths-weave*[*OF assms*] **by** (*metis Un-subset-iff set-nths-subset*)
qed

lemma *weave-complementary-nthss*[*simp*]:
weave A (*nths xs A*) (*nths xs* ($-A$)) = *xs*
proof (*induction xs rule*:*rev-induct*)
case *Nil*
then show *?case* **by** (*metis gen-length-def length-0-conv length-code length-weave nths-nil*)
next
case (*snoc x xs*)
have *length-xs*:*length xs* = *length* (*nths xs A*) + *length* (*nths xs* ($-A$)) **by** (*metis length-weave snoc.IH*)
show *?case*
proof (*cases* (*length xs*)$\in$*A*)
case *True*
have *0*:*length* (*nths xs A*) + *length* (*nths xs* ($-A$)) $\in A$ **using** *length-xs True* **by** *metis*
have *1*:*length* (*nths xs A*) = *card* $\{a \in A.\ a <$ *length* (*nths xs A*) + *length* (*nths xs* ($-A$))$\}$
using *length-nths*[*of xs A*] **by** (*metis* (*no-types*, *lifting*) *Collect-cong length-xs*)
have *2*:*nths* (*xs* @ [*x*]) *A* = *nths xs A* @ [*x*]
unfolding *nths-append*[*of xs* [*x*] *A*] **using** *nths-singleton True* **by** *auto*
have *3*:*nths* (*xs* @ [*x*]) ($-A$) = *nths xs* ($-A$)
unfolding *nths-append*[*of xs* [*x*] $-A$] **using** *True* **by** *auto*
show *?thesis* **unfolding** *2 3 weave-append1*[*OF 0 1*] *snoc.IH* **by** *metis*

next
case *False*
have *0*:*length* (*nths xs A*) + *length* (*nths xs* (−*A*)) ∉ *A* **using** *length-xs False* **by** *metis*
have *1*:*length* (*nths xs* (−*A*)) = *card* {*a* ∈ −*A*. *a* < *length* (*nths xs A*) + *length* (*nths xs* (−*A*))}
using *length-nths*[*of xs* −*A*] **by** (*metis* (*no-types*, *lifting*) *Collect-cong length-xs*)
have *2*:*nths* (*xs* @ [*x*]) *A* = *nths xs A*
unfolding *nths-append*[*of xs* [*x*] *A*] **using** *nths-singleton False* **by** *auto*
have *3*:*nths* (*xs* @ [*x*]) (−*A*) = *nths xs* (−*A*) @ [*x*]
unfolding *nths-append*[*of xs* [*x*] −*A*] **using** *False* **by** *auto*
show *?thesis* **unfolding** *2 3 weave-append2*[*OF 0 1*] *snoc.IH* **by** *metis*
qed
qed

lemma *length-nths′*: *length* (*nths xs I*) = *card* {*i*∈*I*. *i* < *length xs*}
unfolding *length-nths* **by** *meson*

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*}
unfolding *submatrix-def* **by** *simp-all*

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*)
unfolding *submatrix-def* **by** (*simp add*: *assms*(*1*) *assms*(*2*))

lemma *set-le-in*:{*a*. *a* < *n* ∧ *a* ∈ *I*} = {*a* ∈ *I*. *a* < *n*} **by** *meson*

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 −

have *i* = *pick I* (*card* {*a*∈*I*. *a* < *i*})
j = *pick J* (*card* {*a*∈*J*. *a* < *j*}) **using** *pick-card-in-set assms* **by** *auto*
have {*a*∈*I*. *a* < *i*} ⊂ {*i*. *i* < *dim-row A* ∧ *i* ∈ *I*}
{*a*∈*J*. *a* < *j*} ⊂ {*j*. *j* < *dim-col A* ∧ *j* ∈ *J*}
unfolding *set-le-in* **using** ‹*i*<*dim-row A*› ‹*j*<*dim-col A*› *Collect-mono less-imp-le less-le-trans* ‹*i*∈*I*› ‹*j*∈*J*› **by** *auto*
then have *card* {*a*∈*I*. *a* < *i*} < *card* {*i*. *i* < *dim-row A* ∧ *i* ∈ *I*}
card {*a*∈*J*. *a* < *j*} < *card* {*j*. *j* < *dim-col A* ∧ *j* ∈ *J*} **by** (*simp-all add*: *psubset-card-mono*)
then show *?thesis*
using ‹*i* = *pick I* (*card* {*a* ∈ *I*. *a* < *i*})› ‹*j* = *pick J* (*card* {*a* ∈ *J*. *a* < *j*})› *submatrix-index* **by** *fastforce*
qed

lemma *submatrix-split*: *submatrix A I J* = *submatrix* (*submatrix A UNIV J*) *I UNIV*
proof (*rule eq-matI*)
show *dim-row* (*submatrix A I J*) = *dim-row* (*submatrix* (*submatrix A UNIV J*) *I UNIV*)
by (*simp add*: *dim-submatrix*(*1*))
show *dim-col* (*submatrix A I J*) = *dim-col* (*submatrix* (*submatrix A UNIV J*) *I UNIV*)
by (*simp add*: *dim-submatrix*(*2*))
fix *i j* **assume** *ij-le*:*i* < *dim-row* (*submatrix* (*submatrix A UNIV J*) *I UNIV*) *j* < *dim-col* (*submatrix* (*submatrix A UNIV J*) *I UNIV*)
then have *ij-le1*:*i*<*card* {*i*. *i* < *dim-row A* ∧ *i* ∈ *I*} *j*<*card* {*i*. *i* < *dim-col A* ∧ *i* ∈ *J*}
by (*simp-all add*: *dim-submatrix*)
then have *ij-le2*:*i*<*card* {*i*. *i* < *dim-row* (*submatrix A UNIV J*) ∧ *i* ∈ *I*} *j*<*card* {*i*. *i* < *dim-col* (*submatrix A UNIV J*) ∧ *i* ∈ *UNIV*}
by (*simp-all add*: *dim-submatrix*)
then have *i-le3*:*pick I i*<*card* {*i*. *i* < *dim-row A* ∧ *i* ∈ *UNIV*}
using *ij-le1*(*1*) *pick-le* **by** *auto*
have *j-le3*: *pick UNIV j*<*card* {*i*. *i* < *dim-col A* ∧ *i* ∈ *J*} **unfolding** *pick-UNIV* **by** (*simp add*: *ij-le1*(*2*))
then show *submatrix A I J* $$ (*i*, *j*) = *submatrix* (*submatrix A UNIV J*) *I UNIV* $$ (*i*, *j*)
unfolding *submatrix-index*[*OF ij-le1*] *submatrix-index*[*OF ij-le2*] *submatrix-index*[*OF i-le3 j-le3*]
unfolding *pick-UNIV* **by** *metis*
qed

end

40 Rank and Submatrices

theory *DL-Rank-Submatrix*
imports *DL-Rank DL-Submatrix Matrix*
begin

lemma *row-submatrix-UNIV*:
assumes $i < card\ \{i.\ i < dim\text{-}row\ A \wedge i \in I\}$
shows *row* (*submatrix A I UNIV*) i = *row A* (*pick I i*)
proof (*rule eq-vecI*)
show *dim-eq*:*dim-vec* (*row* (*submatrix A I UNIV*) *i*) = *dim-vec* (*row A* (*pick I i*))
unfolding *carrier-vecD*[*OF row-carrier*] *dim-submatrix* **by** *auto*
fix *j* **assume** $j <$ *dim-vec* (*row A* (*pick I i*))
then have $j <$ *dim-col* (*submatrix A I UNIV*) $j <$ *dim-col A* $j <$ *card* $\{j.\ j <$ *dim-col A* $\wedge j \in$ *UNIV*$\}$ **using** *dim-eq* **by** *auto*
show *row* (*submatrix A I UNIV*) *i* \$ *j* = *row A* (*pick I i*) \$ *j*
unfolding *row-def index-vec*[*OF* ‹$j <$ *dim-col* (*submatrix A I UNIV*)›] *index-vec*[*OF* ‹$j <$ *dim-col A*›]
using *submatrix-index*[*OF assms* ‹$j <$ *card* $\{j.\ j <$ *dim-col A* $\wedge j \in$ *UNIV*$\}$›] **using** *pick-UNIV* **by** *auto*
qed

lemma *distinct-cols-submatrix-UNIV*:
assumes *distinct* (*cols* (*submatrix A I UNIV*))
shows *distinct* (*cols A*)
using *assms* **proof** (*rule contrapos-pp*)
assume ¬ *distinct* (*cols A*)
then obtain *i j* **where** $i <$ *dim-col A* $j <$ *dim-col A* (*cols A*)!*i* = (*cols A*)!*j* $i \neq j$
using *distinct-conv-nth cols-length* **by** *metis*
have $i <$ *dim-col* (*submatrix A I UNIV*) $j <$ *dim-col* (*submatrix A I UNIV*)
unfolding *dim-submatrix* **using** ‹$i <$ *dim-col A*› ‹$j <$ *dim-col A*›**by** *simp-all*
then have $i <$ *length* (*cols* (*submatrix A I UNIV*)) $j <$ *length* (*cols* (*submatrix A I UNIV*))
unfolding *cols-length* **by** *simp-all*
have (*cols* (*submatrix A I UNIV*))!*i* = (*cols* (*submatrix A I UNIV*))!*j*
proof (*rule eq-vecI*)
show *dim-vec* (*cols* (*submatrix A I UNIV*) ! *i*) = *dim-vec* (*cols* (*submatrix A I UNIV*) ! *j*)
by (*simp add*: ‹$i <$ *dim-col* (*submatrix A I UNIV*)› ‹$j <$ *dim-col* (*submatrix A I UNIV*)›)
fix *k* **assume** $k <$ *dim-vec* (*cols* (*submatrix A I UNIV*) ! *j*)
then have $k <$ *dim-row* (*submatrix A I UNIV*)
using ‹$j <$ *length* (*cols* (*submatrix A I UNIV*))› **by** *auto*
then have $k <$ *card* $\{j.\ j <$ *dim-row A* $\wedge j \in I\}$ **using** *dim-submatrix*(*1*) **by** *metis*
have *i-transfer*:*cols* (*submatrix A I UNIV*) ! *i* \$ *k* = (*cols A*) ! *i* \$ (*pick I k*)
unfolding *cols-nth*[*OF* ‹$i <$ *dim-col* (*submatrix A I UNIV*)›] *col-def index-vec*[*OF* ‹$k <$ *dim-row* (*submatrix A I UNIV*)›]
unfolding *submatrix-index*[*OF* ‹$k <$ *card* $\{j.\ j <$ *dim-row A* $\wedge j \in I\}$› ‹$i <$ *dim-col* (*submatrix A I UNIV*)›[*unfolded dim-submatrix*]]
unfolding *pick-UNIV cols-nth*[*OF* ‹$i <$ *dim-col A*›] *col-def index-vec*[*OF pick-le*[*OF* ‹$k <$ *card* $\{j.\ j <$ *dim-row A* $\wedge j \in I\}$›]]
by *metis*

have *j-transfer*:*cols* (*submatrix A I UNIV*) ! *j* \$ *k* = (*cols A*) ! *j* \$ (*pick I k*)
unfolding *cols-nth*[*OF* ‹*j* < *dim-col* (*submatrix A I UNIV*)›] *col-def index-vec*[*OF* ‹*k* < *dim-row* (*submatrix A I UNIV*)›]
unfolding *submatrix-index*[*OF* ‹*k* < *card* {*j*. *j* < *dim-row A* ∧ *j* ∈ *I*}› ‹*j* < *dim-col* (*submatrix A I UNIV*)›[*unfolded dim-submatrix*]]
unfolding *pick-UNIV cols-nth*[*OF* ‹*j* < *dim-col A*›] *col-def index-vec*[*OF pick-le*[*OF* ‹*k* < *card* {*j*. *j* < *dim-row A* ∧ *j* ∈ *I*}›]]
by *metis*
show *cols* (*submatrix A I UNIV*) ! *i* \$ *k* = *cols* (*submatrix A I UNIV*) ! *j* \$ *k*
using ‹*cols A* ! *i* = *cols A* ! *j*› *i-transfer j-transfer* **by** *auto*
qed
then show ¬ *distinct* (*cols* (*submatrix A I UNIV*)) **unfolding** *distinct-conv-nth*
using ‹*i* < *length* (*cols* (*submatrix A I UNIV*))› ‹*j* < *length* (*cols* (*submatrix A I UNIV*))› ‹*i* ≠ *j*› **by** *blast*
qed

lemma *cols-submatrix-subset*: *set* (*cols* (*submatrix A UNIV J*)) ⊆ *set* (*cols A*)
proof
fix *c* **assume** *c* ∈ *set* (*cols* (*submatrix A UNIV J*))
then obtain *j* **where** *j* < *length* (*cols* (*submatrix A UNIV J*)) *cols* (*submatrix A UNIV J*) ! *j* = *c*
by (*meson in-set-conv-nth*)
then have *j* < *dim-col* (*submatrix A UNIV J*) **by** *simp*
then have *j* < *card* {*j*. *j* < *dim-col A* ∧ *j* ∈ *J*} **by** (*simp add*: *dim-submatrix*(*2*))
have *cols* (*submatrix A UNIV J*) ! *j* = *cols A* ! (*pick J j*)
unfolding *cols-nth*[*OF* ‹*j* < *dim-col* (*submatrix A UNIV J*)›] *cols-nth*[*OF pick-le*[*OF* ‹*j* < *card* {*j*. *j* < *dim-col A* ∧ *j* ∈ *J*}›]]
proof (*rule eq-vecI*)
show *dim-vec* (*col* (*submatrix A UNIV J*) *j*) = *dim-vec* (*col A* (*pick J j*)) **unfolding** *dim-col dim-submatrix* **by** *auto*
fix *i* **assume** *i* < *dim-vec* (*col A* (*pick J j*))
then have *i* < *dim-row A* **by** *simp*
then have *i* < *dim-row* (*submatrix A UNIV J*) **using** ‹*dim-vec* (*col* (*submatrix A UNIV J*) *j*) = *dim-vec* (*col A* (*pick J j*))› **by** *auto*
show *col* (*submatrix A UNIV J*) *j* \$ *i* = *col A* (*pick J j*) \$ *i*
unfolding *col-def index-vec*[*OF* ‹*i* < *dim-row* (*submatrix A UNIV J*)›] *index-vec*[*OF* ‹*i* < *dim-row A*›]
using *submatrix-index* **by** (*metis* (*no-types*, *lifting*) ‹*dim-vec* (*col* (*submatrix A UNIV J*) *j*) = *dim-vec* (*col A* (*pick J j*))›
‹*i* < *dim-vec* (*col A* (*pick J j*))› ‹*j* < *dim-col* (*submatrix A UNIV J*)› *dim-col dim-submatrix*(*1*) *dim-submatrix*(*2*) *pick-UNIV*)
qed
then show *c* ∈ *set* (*cols A*)
using ‹*cols* (*submatrix A UNIV J*) ! *j* = *c*›
using *pick-le*[*OF* ‹*j* < *card* {*j*. *j* < *dim-col A* ∧ *j* ∈ *J*}›] **by** (*metis cols-length nth-mem*)
qed

lemma (**in** *vec-space*) *lin-dep-submatrix-UNIV*:

assumes $A \in$ *carrier-mat n nc*
assumes *lin-dep* (*set* (*cols A*))
assumes *distinct* (*cols* (*submatrix A I UNIV*))
shows *LinearCombinations.module.lin-dep class-ring* (*module-vec TYPE*('*a*) (*card* $\{i.\ i < n \wedge i \in I\}$)) (*set* (*cols* (*submatrix A I UNIV*)))
(**is** *LinearCombinations.module.lin-dep class-ring ?M* (*set ?S'*))
proof −
obtain *v* **where** *2*:$v \in$ *carrier-vec nc* **and** *3*:$v \neq 0_v$ *nc* **and** $A *_v v = 0_v\ n$
using *vec-space.lin-depE*[*OF assms*(*1*) *assms*(*2*) *distinct-cols-submatrix-UNIV*[*OF assms*(*3*)]] **by** *auto*
have *1*: *submatrix A I UNIV* $\in$ *carrier-mat* (*card* $\{i.\ i < n \wedge i \in I\}$) *nc*
apply (*rule carrier-matI*) **unfolding** *dim-submatrix* **using** ‹$A \in$ *carrier-mat n nc*› **by** *auto*
have *4*:*submatrix A I UNIV* $*_v\ v = 0_v$ (*card* $\{i.\ i < n \wedge i \in I\}$)
proof (*rule eq-vecI*)
show *dim-eq*:*dim-vec* (*submatrix A I UNIV* $*_v\ v$) = *dim-vec* (0_v (*card* $\{i.\ i < n \wedge i \in I\}$)) **using** *1* **by** *auto*
fix *i* **assume** $i <$ *dim-vec* (0_v (*card* $\{i.\ i < n \wedge i \in I\}$))
then have *i-le*:$i <$ *card* $\{i.\ i < n \wedge i \in I\}$ **by** *auto*
have (*submatrix A I UNIV* $*_v\ v$) \$ *i* = *row* (*submatrix A I UNIV*) $i \cdot v$ **using** *dim-eq i-le* **by** *auto*
also have ... = *row A* (*pick I i*) $\cdot\ v$ **using** *row-submatrix-UNIV*
by (*metis* (*no-types*, *lifting*) *dim-eq dim-mult-mat-vec dim-submatrix*(*1*) ‹$i <$ *dim-vec* (0_v (*card* $\{i.\ i < n \wedge i \in I\}$))›)
also have ... = *0*
using ‹$A *_v v = 0_v\ n$› *i-le*[*THEN pick-le*] **by** (*metis assms*(*1*) *index-mult-mat-vec carrier-matD*(*1*) *index-zero-vec*(*1*))
also have ... = 0_v (*card* $\{i.\ i < n \wedge i \in I\}$) \$ *i* **by** (*simp add*: *i-le*)
finally show (*submatrix A I UNIV* $*_v\ v$) \$ *i* = 0_v (*card* $\{i.\ i < n \wedge i \in I\}$) \$ *i* **by** *metis*
qed
show *?thesis* **using** *vec-space.lin-depI*[*OF 1 2 3 4*] **using** *assms*(*3*) **by** *auto*
qed

lemma (**in** *vec-space*) *rank-gt-minor*:
assumes $A \in$ *carrier-mat n nc*
assumes *det* (*submatrix A I J*) $\neq 0$
shows *card* $\{j.\ j < nc \wedge j \in J\} \leq$ *rank A*
proof −
have *square*:*dim-row* (*submatrix A I J*) = *dim-col* (*submatrix A I J*)
using *det-def* ‹*det* (*submatrix A I J*) $\neq 0$› **by** *metis*
then have *full-rank*:*vec-space.rank* (*dim-row* (*submatrix A I J*)) (*submatrix A I J*) = *dim-row* (*submatrix A I J*)
using *vec-space.low-rank-det-zero assms*(*2*) *carrier-matI* **by** *auto*
then have *distinct*:*distinct* (*cols* (*submatrix A I J*))
using *vec-space.non-distinct-low-rank square less-irrefl carrier-matI* **by** *metis*
then have *indpt*:*LinearCombinations.module.lin-indpt class-ring* (*module-vec TYPE*('*a*) (*dim-row* (*submatrix A I J*))) (*set* (*cols* (*submatrix A I J*)))
using *vec-space.full-rank-lin-indpt*[*OF - full-rank distinct*] *square* **by** *fastforce*

have *distinct2*: *distinct* (*cols* (*submatrix* (*submatrix A UNIV J*) *I UNIV*)) **using** *submatrix-split distinct* **by** *metis*
have *indpt2*:*LinearCombinations.module.lin-indpt class-ring* (*module-vec TYPE*('*a*) (*card* $\{i.\ i < n \wedge i \in I\}$)) (*set* (*cols* (*submatrix* (*submatrix A UNIV J*) *I UNIV*)))
using *submatrix-split dim-submatrix*(*1*) *indpt* **by** (*metis* (*full-types*) *assms*(*1*) *carrier-matD*(*1*))

have *submatrix A UNIV J* ∈ *carrier-mat n* (*dim-col* (*submatrix A UNIV J*))
apply (*rule carrier-matI*) **unfolding** *dim-submatrix*(*1*) **using** ‹*A* ∈ *carrier-mat n nc*› *carrier-matD* **by** *simp-all*
have *lin-indpt* (*set* (*cols* (*submatrix A UNIV J*)))
using *indpt2 vec-space.lin-dep-submatrix-UNIV*[*OF* ‹*submatrix A UNIV J* ∈ *carrier-mat n* (*dim-col* (*submatrix A UNIV J*))› - *distinct2*] **by** *blast*
have *distinct3*:*distinct* (*cols* (*submatrix A UNIV J*)) **by** (*metis distinct distinct-cols-submatrix-UNIV submatrix-split*)
show *?thesis* **using**
rank-ge-card-indpt[*OF* ‹*A* ∈ *carrier-mat n nc*› *cols-submatrix-subset* ‹*lin-indpt* (*set* (*cols* (*submatrix A UNIV J*)))›,
unfolded distinct-card[*OF distinct3*, *unfolded cols-length dim-submatrix*], *unfolded carrier-matD*(*2*)[*OF* ‹*A* ∈ *carrier-mat n nc*›]]
by *blast*
qed

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.