Finite-Dimensional Algebras and Scalar Towers

Arthur Freitas Ramos David Barros Hulak
Ruy J. G. B. de Queiroz

November 6, 2026

Abstract

For a finite-dimensional vector space over a finite-dimensional field extension, the dimension over the smaller field is the product of the two dimensions. The result includes vector subspaces and the zero space. For compatible towers of two fields and a finite-dimensional commutative unital algebra, trace and norm over the bottom field are the iterated trace and norm through the intermediate field. Examples include $\mathbb{F}_2 \subseteq \mathbb{F}_4$ and $\mathbb{R} \subseteq \mathbb{C}$. The development follows the standard finite extension treatment in [1].

Contents

1 Restriction of scalars for carrier-based vector spaces **2**

2 Existing scalar infrastructure **12**

3 Scalar towers **16**

4 Restriction of scalars **16**

5 Composition of bases **22**

6 Dimension in a tower **30**

7 Regular representation **32**

8 Algebra trace **44**

9 Algebra norm **49**

10 Examples **83**
10.1 Finite-field towers 83
10.2 The four-element field over $\mathbb{F}_2$ 86
10.3 The quadratic extension of the reals by the imaginary unit 94

theory *Scalar-Restriction-Bridge*
imports *HOL−New-Algebra.Steinitz*
begin

1 Restriction of scalars for carrier-based vector spaces

The upper carrier is an arbitrary vector space, not a field. In particular, it may be the zero space. The lower vector space describes the intermediate field over the smaller field, using the same scalar operations. These parameters can also describe subfields of a noncommutative ambient ring and a vector subspace of that ring.

locale *scalar-restriction-tower* =
  lower: *Vector-Space K sadd smult szero sone sadd szero F smult* +
  upper: *Vector-Space F sadd smult szero sone vadd vzero E scale*
  for *K F* :: *'s set*
    and *sadd smult* :: *'s* ⇒ *'s* ⇒ *'s*
    and *szero sone* :: *'s*
    and *vadd* :: *'v* ⇒ *'v* ⇒ *'v*
    and *vzero* :: *'v*
    and *E* :: *'v set*
    and *scale* :: *'s* ⇒ *'v* ⇒ *'v* +
  assumes *base-subset*: *K* ⊆ *F*
begin

sublocale *direct*: *Vector-Space K sadd smult szero sone vadd vzero E scale*
proof (*intro Vector-Space.intro Vector-Space-axioms.intro*)
  show *Field K sadd smult szero sone* **by** (*rule lower.Field-axioms*)
  show *Abelian-Group E vadd vzero* **by** (*rule upper.vadd-group*)
  show ⋀*a v*. ⟦*a* ∈ *K*; *v* ∈ *E*⟧ ⟹ *scale a v* ∈ *E*
    using *base-subset* **by** (*blast intro*: *upper.scale-closed*)
  show ⋀*a u v*. ⟦*a* ∈ *K*; *u* ∈ *E*; *v* ∈ *E*⟧ ⟹
      scale a (*vadd u v*) = *vadd* (*scale a u*) (*scale a v*)
    using *base-subset* **by** (*blast intro*: *upper.scale-distrib-vadd*)
  show ⋀*a b v*. ⟦*a* ∈ *K*; *b* ∈ *K*; *v* ∈ *E*⟧ ⟹
      scale (*sadd a b*) *v* = *vadd* (*scale a v*) (*scale b v*)
    using *base-subset* **by** (*blast intro*: *upper.scale-distrib-add*)
  show ⋀*a b v*. ⟦*a* ∈ *K*; *b* ∈ *K*; *v* ∈ *E*⟧ ⟹
      scale (*smult a b*) *v* = *scale a* (*scale b v*)
    using *base-subset* **by** (*blast intro*: *upper.scale-scale*)
  show ⋀*v*. *v* ∈ *E* ⟹ *scale sone v* = *v*
    by (*rule upper.scale-one*)
qed

Restricting the field changes the allowed coefficients, but not the expression for a linear combination.

lemma *restricted-lincomb*:
  direct.lincomb c A = *upper.lincomb c A*
  by (*simp only*: *direct.lincomb-def*)

definition *product-basis* :: *'s set* ⇒ *'v set* ⇒ *'v set*
  where *product-basis B C* = (λ(*b*,*c*). *scale b c*) ‘ (*B* × *C*)

lemma *product-basis-finite*:
assumes *finite B finite C*
shows *finite* (*product-basis B C*)
using *assms* **by** (*simp add*: *product-basis-def*)

lemma *product-basis-subset*:
assumes $B \subseteq F$ $C \subseteq E$
shows *product-basis B C* $\subseteq E$
using *assms* **unfolding** *product-basis-def* **by** (*auto intro*: *upper.scale-closed*)

lemma *product-basis-empty* [*simp*]:
product-basis B $\{\} = \{\}$
by (*simp add*: *product-basis-def*)

lemma *lower-basis-nonzero*:
assumes *B*: *lower.basis B* **and** *b*: $b \in B$
shows $b \neq$ *szero*
proof
assume $b =$ *szero*
have *ind*: *lower.lin-indep B* **by** (*rule lower.basis-lin-indep*[*OF B*])
have $b \notin$ *lower.span* $(B - \{b\})$
by (*rule lower.lin-indep-not-in-span*[*OF ind b*])
moreover have $b \in$ *lower.span* $(B - \{b\})$
using ‹$b =$ *szero*› *lower.span-zero* **by** *simp*
ultimately show *False* **by** *contradiction*
qed

lemma *upper-basis-scaled-unique*:
assumes *C*: *upper.basis C* **and** *c*: $c \in C$ **and** *d*: $d \in C$
and *b*: $b \in F$ **and** *e*: $e \in F$
and *eq*: *scale b c* = *scale e d*
shows $b =$ (*if* $c = d$ *then e else szero*)
proof −
have *finC*: *finite C* **and** *CE*: $C \subseteq E$
using *C* **by** (*auto simp*: *upper.basis-def*)
have *ind*: *upper.mod.lin-indep C*
by (*rule iffD1*[*OF upper.lin-indep-iff-mod-lin-indep*[*OF finC CE*]
upper.basis-lin-indep[*OF C*]])
have *cE*: $c \in E$ **and** *dE*: $d \in E$ **using** *CE c d* **by** *blast*+
have *coeff-b*: *upper.mod.coeffs-on* (λ-. *b*) $\{c\}$
by (*rule upper.mod.coeffs-onI*) (*simp add*: *b*)
have *coeff-e*: *upper.mod.coeffs-on* (λ-. *e*) $\{d\}$
by (*rule upper.mod.coeffs-onI*) (*simp add*: *e*)
have *comb*:
upper.mod.lincomb (λ-. *b*) $\{c\}$ = *upper.mod.lincomb* (λ-. *e*) $\{d\}$
using *eq* **by** (*simp add*: *cE dE b e*)
have (*if* $c \in \{c\}$ *then b else szero*) =
(*if* $c \in \{d\}$ *then e else szero*)

```
    by (rule upper.mod.lin-indep-lincomb-unique-gen[OF ind - - - - coeff-b coeff-e
comb])
      (use c d in auto)
  then show ?thesis by simp
qed

lemma product-basis-injective:
  assumes B: lower.basis B and C: upper.basis C
  shows inj-on (λ(b,c). scale b c) (B × C)
proof (rule inj-onI)
  fix x y
  assume x: x ∈ B × C and y: y ∈ B × C
    and eq: (λ(b,c). scale b c) x = (λ(b,c). scale b c) y
  obtain b c where x': x = (b,c) and b: b ∈ B and c: c ∈ C
    using x by (cases x) auto
  obtain e d where y': y = (e,d) and e: e ∈ B and d: d ∈ C
    using y by (cases y) auto
  have BF: B ⊆ F using B by (auto simp: lower.basis-def)
  have bF: b ∈ F and eF: e ∈ F using BF b e by blast+
  have scaled: scale b c = scale e d using eq x' y' by simp
  have unique: b = (if c = d then e else szero)
    by (rule upper-basis-scaled-unique[OF C c d bF eF scaled])
  have c = d using unique lower-basis-nonzero[OF B b] by (cases c = d) auto
  then have b = e using unique by simp
  then show x = y using x' y' ‹c = d› by simp
qed

lemma scalar-lower-combination-in-product-span:
  assumes BF: B ⊆ F and CE: C ⊆ E and c: c ∈ C
    and fin: finite T and TB: T ⊆ B
    and coeff: ⋀b. b ∈ T ⟹ f b ∈ K
  shows scale (lower.lincomb f T) c ∈ direct.span (product-basis B C)
proof −
  have PE: product-basis B C ⊆ E by (rule product-basis-subset[OF BF CE])
  have cE: c ∈ E using c CE by blast
  have step: scale (lower.lincomb f U) c ∈ direct.span (product-basis B C)
    if finite U U ⊆ B ∀ b∈U. f b ∈ K for U
    using that
  proof (induction U rule: finite-induct)
    case empty
    then show ?case
      by (simp add: upper.mod.scale-zero-scalar cE)
  next
    case (insert b U)
    have bF: b ∈ F and UF: U ⊆ F
      using insert.prems BF by auto
    have fK: f b ∈ K and fUK: ⋀x. x ∈ U ⟹ f x ∈ K
      using insert.prems by auto
    have fF: f b ∈ F and fUF: ⋀x. x ∈ U ⟹ f x ∈ F
```

```
    using fK fUK base-subset by blast+
  have tailF: lower.lincomb f U ∈ F
    by (rule lower.lincomb-closed[OF UF fUK])
  have termF: smult (f b) b ∈ F
    by (rule lower.scale-closed[OF fK bF])
  have termsU: (λx. smult (f x) x) ∈ U → F
    using UF fUK by (auto intro: lower.scale-closed)
  have split: lower.lincomb f (insert b U) =
      sadd (smult (f b) b) (lower.lincomb f U)
    using lower.vadd.fincomp-insert[OF insert.hyps(1,2) termsU termF]
    by (simp only: lower.lincomb-def)
  have base: scale b c ∈ direct.span (product-basis B C)
  proof (rule direct.span-incl[OF PE])
    show scale b c ∈ product-basis B C
      using insert.prems c by (auto simp: product-basis-def)
  qed
  have first: scale (smult (f b) b) c ∈ direct.span (product-basis B C)
    using direct.span-scale[OF PE fK base] upper.scale-scale[OF fF bF cE] by
simp
  have rest: scale (lower.lincomb f U) c ∈ direct.span (product-basis B C)
    using insert.IH insert.prems by auto
  have vadd (scale (smult (f b) b) c) (scale (lower.lincomb f U) c)
      ∈ direct.span (product-basis B C)
    by (rule direct.span-vadd[OF PE first rest])
  then show ?case
    using upper.scale-distrib-add[OF termF tailF cE] by (simp only: split)
 qed
 show ?thesis by (rule step[OF fin TB]) (use coeff in blast)
qed

lemma fincomp-in-direct-span:
 assumes PE: P ⊆ E and fin: finite A
   and terms: ⋀a. a ∈ A ⟹ g a ∈ direct.span P
 shows upper.vadd.fincomp g A ∈ direct.span P
 using fin terms
proof (induction A rule: finite-induct)
 case empty
 then show ?case by simp
next
 case (insert a A)
 have ga: g a ∈ direct.span P using insert.prems by simp
 have rest: upper.vadd.fincomp g A ∈ direct.span P
   using insert.IH insert.prems by blast
 have gE: g ∈ insert a A → E
   using insert.prems direct.span-subset-V[OF PE] by blast
 have vadd (g a) (upper.vadd.fincomp g A) ∈ direct.span P
   by (rule direct.span-vadd[OF PE ga rest])
 then show ?case using insert.hyps gE by simp
qed
```

lemma *product-basis-spanning*:
assumes *B*: *lower.basis B* **and** *C*: *upper.basis C*
shows *direct.spanning* (*product-basis B C*)
proof −
have *finB*: *finite B* **and** *BF*: $B \subseteq F$
using *B* **by** (*auto simp*: *lower.basis-def*)
have *finC*: *finite C* **and** *CE*: $C \subseteq E$
using *C* **by** (*auto simp*: *upper.basis-def*)
have *PE*: *product-basis B C* $\subseteq E$ **by** (*rule product-basis-subset*[*OF BF CE*])
have *finP*: *finite* (*product-basis B C*) **by** (*rule product-basis-finite*[*OF finB finC*])
have *lower-span*: *lower.spanning B* **by** (*rule lower.basis-spanning*[*OF B*])
have *upper-span*: *upper.spanning C* **by** (*rule upper.basis-spanning*[*OF C*])
have *all-span*: $x \in$ *direct.span* (*product-basis B C*) **if** *xE*: $x \in E$ **for** *x*
proof −
obtain *d* **where** *d*: $d \in C \to_E F$ **and** *x*: $x =$ *upper.lincomb d C*
using *upper-span xE* **by** (*auto simp*: *upper.spanning-def*)
have *summand*: *scale* (*d c*) *c* $\in$ *direct.span* (*product-basis B C*) **if** *c*: $c \in C$ **for** *c*
proof −
have *dcF*: *d c* $\in F$ **using** *d c* **by** *auto*
obtain *f* **where** *f*: $f \in B \to_E K$ **and** *dc*: *d c* = *lower.lincomb f B*
using *lower-span dcF* **by** (*auto simp*: *lower.spanning-def*)
have *scale* (*lower.lincomb f B*) *c* $\in$ *direct.span* (*product-basis B C*)
by (*rule scalar-lower-combination-in-product-span*[*OF BF CE c finB subset-refl*])
(*use f* **in** *auto*)
then show *?thesis* **using** *dc* **by** *simp*
qed
have *upper.vadd.fincomp* (λ*c*. *scale* (*d c*) *c*) *C*
$\in$ *direct.span* (*product-basis B C*)
by (*rule fincomp-in-direct-span*[*OF PE finC summand*])
then show *?thesis* **using** *x* **by** (*simp add*: *upper.lincomb-def*)
qed
have *direct.mod.spanning* (*product-basis B C*)
by (*rule direct.mod.spanningI*[*OF PE all-span*])
then show *?thesis*
using *direct.spanning-iff-mod-spanning*[*OF finP PE*] **by** *blast*
qed

lemma *scale-lower-lincomb*:
assumes *BF*: $B \subseteq F$ **and** *cE*: $c \in E$
and *fK*: $\bigwedge b.\ b \in B \Longrightarrow f\ b \in K$
shows *scale* (*lower.lincomb f B*) *c* =
upper.vadd.fincomp (λ*b*. *scale* (*smult* (*f b*) *b*) *c*) *B*
proof −
have *terms*: (λ*b*. *smult* (*f b*) *b*) $\in B \to F$
using *BF fK* **by** (*auto intro*: *lower.scale-closed*)
have *hom*: *scale* (*lower.vadd.fincomp* (λ*b*. *smult* (*f b*) *b*) *B*) *c* =

```
    upper.vadd.fincomp (λb. scale (smult (f b) b) c) B
  proof (rule fincomp-hom[OF lower.vadd.commutative-monoid-axioms
      upper.vadd.commutative-monoid-axioms])
    show ⋀x. x ∈ F ⟹ scale x c ∈ E
      by (rule upper.scale-closed[OF - cE])
    show scale szero c = vzero by (rule upper.mod.scale-zero-scalar[OF cE])
    show ⋀x y. ⟦x ∈ F; y ∈ F⟧ ⟹
      scale (sadd x y) c = vadd (scale x c) (scale y c)
      by (rule upper.scale-distrib-add[OF - - cE])
    show (λb. smult (f b) b) ∈ B → F by (rule terms)
  qed
  show ?thesis using hom by (simp only: lower.lincomb-def)
qed

lemma vector-fincomp-cartesian:
  assumes finB: finite B and finC: finite C
    and gE: ⋀b c. ⟦b ∈ B; c ∈ C⟧ ⟹ g (b,c) ∈ E
  shows upper.vadd.fincomp g (B × C) =
    upper.vadd.fincomp (λc. upper.vadd.fincomp (λb. g (b,c)) B) C
proof −
  have disjoint: pairwise (λc d. disjnt (B × {c}) (B × {d})) C
    by (auto simp: pairwise-def disjnt-def)
  have union: (⋃c∈C. B × {c}) = B × C by blast
  have split:
    upper.vadd.fincomp g (B × C) =
      upper.vadd.fincomp (λc. upper.vadd.fincomp g (B × {c})) C
    using upper.vadd.fincomp-UN-disjoint[OF finC - disjoint, of g] finB gE union
    by auto
  have slice: upper.vadd.fincomp g (B × {c}) =
      upper.vadd.fincomp (λb. g (b,c)) B if c: c ∈ C for c
  proof −
    have image: (λb. (b,c)) ‘ B = B × {c} by auto
    have valid: g ∈ ((λb. (b,c)) ‘ B) → E
      using gE c by auto
    have injective: inj-on (λb. (b,c)) B by (auto simp: inj-on-def)
    show ?thesis using upper.vadd.fincomp-reindex[OF valid injective] image by
simp
  qed
  show ?thesis using split slice
    by (intro trans[OF split] upper.vadd.fincomp-cong′)
      (auto intro: upper.vadd.fincomp-closed gE)
qed

lemma product-lincomb-nested:
  assumes B: lower.basis B and C: upper.basis C
    and fK: ⋀v. v ∈ product-basis B C ⟹ f v ∈ K
  shows direct.lincomb f (product-basis B C) =
    upper.lincomb (λc. lower.lincomb (λb. f (scale b c)) B) C
proof −
```

```
  have finB: finite B and BF: B ⊆ F
    using B by (auto simp: lower.basis-def)
  have finC: finite C and CE: C ⊆ E
    using C by (auto simp: upper.basis-def)
  have PE: product-basis B C ⊆ E by (rule product-basis-subset[OF BF CE])
  have fk: f (scale b c) ∈ K if b: b ∈ B and c: c ∈ C for b c
  proof (rule fK)
   show scale b c ∈ product-basis B C using b c by (auto simp: product-basis-def)
  qed
  have terms: (λv. scale (f v) v) ∈ product-basis B C → E
    using PE fK base-subset by (auto intro: upper.scale-closed)
  have inj: inj-on (λ(b,c). scale b c) (B × C)
    by (rule product-basis-injective[OF B C])
  have reindex: upper.vadd.fincomp (λv. scale (f v) v) (product-basis B C) =
      upper.vadd.fincomp (λ(b,c). scale (f (scale b c)) (scale b c)) (B × C)
    using upper.vadd.fincomp-reindex[OF terms[unfolded product-basis-def] inj]
    by (simp only: product-basis-def case-prod-unfold)
  have pairE: scale (f (scale b c)) (scale b c) ∈ E
    if b ∈ B c ∈ C for b c
    using BF CE base-subset fk[OF that] that by (auto intro: upper.scale-closed)
  have cart:
    upper.vadd.fincomp (λ(b,c). scale (f (scale b c)) (scale b c)) (B × C) =
      upper.vadd.fincomp (λc. upper.vadd.fincomp
        (λb. scale (f (scale b c)) (scale b c)) B) C
  proof −
    have pairE': (λ(b,c). scale (f (scale b c)) (scale b c)) (b,c) ∈ E
      if b ∈ B c ∈ C for b c
      using pairE[OF that] by simp
     note result = vector-fincomp-cartesian[where g=λ(b,c). scale (f (scale b c))
(scale b c),
        OF finB finC pairE']
    show ?thesis using result by simp
  qed
  have inner:
    upper.vadd.fincomp (λb. scale (f (scale b c)) (scale b c)) B =
      scale (lower.lincomb (λb. f (scale b c)) B) c
    if c: c ∈ C for c
  proof −
    have cE: c ∈ E using CE c by blast
    have rightE: (λb. scale (smult (f (scale b c)) b) c) ∈ B → E
    proof
      fix b assume b: b ∈ B
      have bF: b ∈ F using BF b by blast
      have smult (f (scale b c)) b ∈ F
        by (rule lower.scale-closed[OF fk[OF b c] bF])
      then show scale (smult (f (scale b c)) b) c ∈ E
        by (rule upper.scale-closed[OF - cE])
    qed
    have equal:
```

```
      upper.vadd.fincomp (λb. scale (f (scale b c)) (scale b c)) B =
       upper.vadd.fincomp (λb. scale (smult (f (scale b c)) b) c) B
    proof (rule upper.vadd.fincomp-cong'[OF refl rightE])
      fix b assume b: b ∈ B
      have fbF: f (scale b c) ∈ F using fk[OF b c] base-subset by blast
      have bF: b ∈ F using BF b by blast
      show scale (f (scale b c)) (scale b c) = scale (smult (f (scale b c)) b) c
        by (rule upper.scale-scale[OF fbF bF cE, symmetric])
    qed
    show ?thesis using equal scale-lower-lincomb[OF BF cE, of λb. f (scale b c)]
fk c
      by simp
  qed
  have dF: lower.lincomb (λb. f (scale b c)) B ∈ F if c ∈ C for c
    by (rule lower.lincomb-closed[OF BF]) (use fk that in auto)
  have outer:
    upper.vadd.fincomp (λc. upper.vadd.fincomp
        (λb. scale (f (scale b c)) (scale b c)) B) C =
      upper.vadd.fincomp (λc. scale (lower.lincomb (λb. f (scale b c)) B) c) C
    by (rule upper.vadd.fincomp-cong'[OF refl])
      (use dF CE inner in ‹auto intro: upper.scale-closed›)
  have whole:
    upper.vadd.fincomp (λv. scale (f v) v) (product-basis B C) =
     upper.vadd.fincomp (λc. scale (lower.lincomb (λb. f (scale b c)) B) c) C
    by (rule trans[OF reindex], rule trans[OF cart outer])
  show ?thesis
    unfolding direct.lincomb-def upper.lincomb-def by (rule whole)
qed

lemma product-basis-independent:
  assumes B: lower.basis B and C: upper.basis C
  shows direct.lin-indep (product-basis B C)
proof −
  have finB: finite B and BF: B ⊆ F
    using B by (auto simp: lower.basis-def)
  have finC: finite C and CE: C ⊆ E
    using C by (auto simp: upper.basis-def)
  have PE: product-basis B C ⊆ E by (rule product-basis-subset[OF BF CE])
  have finP: finite (product-basis B C) by (rule product-basis-finite[OF finB finC])
  have lower-ind: lower.lin-indep B by (rule lower.basis-lin-indep[OF B])
  have upper-ind: upper.lin-indep C by (rule upper.basis-lin-indep[OF C])
  have zero-coeff:
    ∀ v∈product-basis B C. f v = szero
    if f: f ∈ product-basis B C →_E K
      and zero: direct.lincomb f (product-basis B C) = vzero for f
  proof −
    define d where d c = lower.lincomb (λb. f (scale b c)) B for c
    have fk: f (scale b c) ∈ K if b: b ∈ B and c: c ∈ C for b c
      using f b c by (auto simp: product-basis-def)
```

```
    have dF: d c ∈ F if c: c ∈ C for c
      unfolding d-def by (rule lower.lincomb-closed[OF BF]) (use fk c in auto)
    have dPi: restrict d C ∈ C →E F using dF by auto
    have fkp: ⋀v. v ∈ product-basis B C ⟹ f v ∈ K
      using f by auto
    have nested: direct.lincomb f (product-basis B C) =
        upper.lincomb (λc. lower.lincomb (λb. f (scale b c)) B) C
      by (rule product-lincomb-nested[OF B C fkp])
    have d-eq: d = (λc. lower.lincomb (λb. f (scale b c)) B)
      by (rule ext) (simp only: d-def)
    have combine: upper.lincomb d C = vzero
      using nested zero by (simp only: d-eq)
    have combine-restrict: upper.lincomb (restrict d C) C = vzero
    proof −
      have upper.lincomb (restrict d C) C = upper.lincomb d C
        by (rule upper.lincomb-cong[OF CE]) (use dF in auto)
      then show ?thesis using combine by simp
    qed
    have dzero: d c = szero if c: c ∈ C for c
    proof −
      have restrict d C c = szero
        using upper-ind dPi combine-restrict c unfolding upper.lin-indep-def by
blast
      then show ?thesis using c by simp
    qed
    have fzero: f (scale b c) = szero if b: b ∈ B and c: c ∈ C for b c
    proof −
      define h where h = (λb. f (scale b c))
      have hK: ⋀b. b ∈ B ⟹ h b ∈ K
        using fk c by (simp add: h-def)
      have hPi: restrict h B ∈ B →E K using hK by auto
      have comb0: lower.lincomb (restrict h B) B = szero
      proof −
        have lower.lincomb (restrict h B) B = lower.lincomb h B
          by (rule lower.lincomb-cong[OF BF]) (use hK in auto)
        then show ?thesis using dzero[OF c] by (simp add: d-def h-def)
      qed
      have restrict h B b = szero
        using lower-ind hPi comb0 b unfolding lower.lin-indep-def by blast
      then show ?thesis using b by (simp add: h-def)
    qed
    show ?thesis using fzero by (auto simp: product-basis-def)
  qed
  show ?thesis unfolding direct.lin-indep-def using finP PE zero-coeff by blast
qed

theorem product-basis:
  assumes B: lower.basis B and C: upper.basis C
  shows direct.basis (product-basis B C)
```

proof (*rule direct.basisI*)
  show *direct.spanning* (*product-basis B C*)
    by (*rule product-basis-spanning*[*OF B C*])
  show *direct.lin-indep* (*product-basis B C*)
    by (*rule product-basis-independent*[*OF B C*])
qed

lemma *product-basis-in-direct-span*:
  assumes *B*: *lower.basis B* **and** *C*: *upper.basis C*
    and *b*: $b \in B$ **and** *c*: $c \in C$
  shows *scale b c* $\in$ *direct.span* (*product-basis B C*)
proof −
  have *BF*: $B \subseteq F$ **and** *CE*: $C \subseteq E$
    using *B C* **by** (*auto simp*: *lower.basis-def upper.basis-def*)
  have *scale b c* $\in$ *product-basis B C*
    using *b c* **by** (*auto simp*: *product-basis-def*)
  then show *?thesis*
    by (*rule direct.span-incl*[*OF product-basis-subset*[*OF BF CE*]])
qed

Cardinality uniqueness turns a product basis into the dimension formula.

lemma *dimension-from-product-basis*:
  assumes *B*: *lower.basis B* **and** *C*: *upper.basis C*
    and *P*: *direct.basis* (*product-basis B C*)
    and *inj*: *inj-on* (λ(*b*,*c*). *scale b c*) (*B* × *C*)
  shows *direct.dimension* = *lower.dimension* ∗ *upper.dimension*
proof −
  have *cardP*: *card* (*product-basis B C*) = *card B* ∗ *card C*
    unfolding *product-basis-def*
    by (*simp add*: *card-image*[*OF inj*] *card-cartesian-product*)
  have *direct.dimension* = *card* (*product-basis B C*)
    by (*rule direct.dimension-eq-any-field*[*OF P*])
  also have ... = *card B* ∗ *card C* **by** (*rule cardP*)
  also have ... = *lower.dimension* ∗ *upper.dimension*
    using *lower.dimension-eq-any-field*[*OF B*] *upper.dimension-eq-any-field*[*OF C*]
    by *simp*
  finally show *?thesis* **.**
qed

theorem *finite-dimension-tower*:
  assumes *lower-finite*: ∃ *B*. *lower.basis B*
    and *upper-finite*: ∃ *C*. *upper.basis C*
  shows ∃ *P*. *direct.basis P*
    and *direct.dimension* = *lower.dimension* ∗ *upper.dimension*
proof −
  obtain *B* **where** *B*: *lower.basis B* **using** *lower-finite* **by** *blast*
  obtain *C* **where** *C*: *upper.basis C* **using** *upper-finite* **by** *blast*
  have *P*: *direct.basis* (*product-basis B C*) **by** (*rule product-basis*[*OF B C*])
  show ∃ *P*. *direct.basis P* **using** *P* **by** *blast*

show *direct.dimension = lower.dimension * upper.dimension*
by (*rule dimension-from-product-basis*[*OF B C P product-basis-injective*[*OF B C*]])
qed

The zero space has an empty basis over either field. This case must not be excluded by an index type or by requiring the upper carrier to be a field.

lemma *zero-space-tower*:
assumes *E = {vzero}*
shows *direct.basis {} upper.basis {}*
and *direct.dimension = lower.dimension * upper.dimension*
proof −
show *D*: *direct.basis {}* **by** (*rule direct.basis-empty-trivial*[*OF assms*])
show *U*: *upper.basis {}* **by** (*rule upper.basis-empty-trivial*[*OF assms*])
have *direct.dimension = 0*
using *direct.dimension-eq-any-field*[*OF D*] **by** *simp*
moreover have *upper.dimension = 0*
using *upper.dimension-eq-any-field*[*OF U*] **by** *simp*
ultimately show *direct.dimension = lower.dimension * upper.dimension* **by** *simp*
qed

end

end
theory *Compatible-Scalar-Actions*
imports
HOL−Analysis.Determinants
begin

2 Existing scalar infrastructure

Isabelle's type-based algebra library represents a module by the locale *module*, parameterized by an explicit scalar action. Consequently, several scalar actions may coexist without introducing new type classes. Linear maps are already represented by *module-hom*, and over fields by the definitionally equivalent locale *vector-space*. We use its existing operations for subspaces, spans, independence, and coordinate representations. The Cartesian matrix library supplies trace and determinant.

locale *compatible-scalar-actions* =
RS: *Modules.module rS* +
SA: *Modules.module sA* +
RA: *Modules.module rA*
for *rS* :: *'r*::*comm-ring-1* ⇒ *'s*::*comm-ring-1* ⇒ *'s*
and *sA* :: *'s* ⇒ *'a*::*comm-ring-1* ⇒ *'a*
and *rA* :: *'r* ⇒ *'a* ⇒ *'a* +
assumes *scalar-compat* [*simp*]:

```
    rA r (sA s a) = sA (rS r s) a
begin

abbreviation scalar-embed :: 'r ⇒ 's
  where scalar-embed r ≡ rS r 1

lemma direct-eq-upper-embed [simp]:
  rA r a = sA (scalar-embed r) a
  using scalar-compat[of r 1 a] by simp

lemma module-hom-restrict-scalars:
  assumes module-hom sA sA f
  shows module-hom rA rA f
proof −
  interpret f: module-hom sA sA f
    by fact
  show ?thesis
    by unfold-locales (simp-all add: f.add f.scale)
qed

end

locale compatible-vector-spaces =
  compatible-scalar-actions rS sA rA
  for rS :: 'r::field ⇒ 's::field ⇒ 's
    and sA :: 's ⇒ 'a::comm-ring-1 ⇒ 'a
    and rA :: 'r ⇒ 'a ⇒ 'a
begin

lemma linear-restrict-scalars:
  assumes Vector-Spaces.linear sA sA f
  shows Vector-Spaces.linear rA rA f
  using assms module-hom-restrict-scalars
  by (simp add: module-hom-iff-linear)

end

locale commutative-algebra =
  Modules.module scale
  for scale :: 'r::comm-ring-1 ⇒ 'a::comm-ring-1 ⇒ 'a +
  assumes scale-mult-right [algebra-simps]:
    scale r x * y = scale r (x * y)
begin

abbreviation scalar-embed :: 'r ⇒ 'a
  where scalar-embed r ≡ scale r 1

lemma scale-eq-embed-mult:
  scale r x = scalar-embed r * x
```

using *scale-mult-right*[*of r 1 x*] **by** *simp*

lemma *scale-mult-left* [*algebra-simps*]:
x ∗ *scale r y* = *scale r* (*x* ∗ *y*)
proof −
have *x* ∗ *scale r y* = *scale r y* ∗ *x*
by (*rule mult.commute*)
also have ... = *scale r* (*y* ∗ *x*)
by (*rule scale-mult-right*)
also have ... = *scale r* (*x* ∗ *y*)
by (*simp only*: *mult.commute*)
finally show *?thesis* **.**
qed

lemma *scalar-embed-zero* [*simp*]:
scalar-embed 0 = *0*
by *simp*

lemma *scalar-embed-one* [*simp*]:
scalar-embed 1 = *1*
by *simp*

lemma *scalar-embed-add* [*simp*]:
scalar-embed (*r* + *s*) = *scalar-embed r* + *scalar-embed s*
by (*simp add*: *scale-left-distrib*)

lemma *scalar-embed-minus* [*simp*]:
scalar-embed (− *r*) = − *scalar-embed r*
by *simp*

lemma *scalar-embed-diff* [*simp*]:
scalar-embed (*r* − *s*) = *scalar-embed r* − *scalar-embed s*
by (*simp add*: *scale-left-diff-distrib*)

lemma *scalar-embed-mult* [*simp*]:
scalar-embed (*r* ∗ *s*) = *scalar-embed r* ∗ *scalar-embed s*
proof −
have *scalar-embed* (*r* ∗ *s*) = *scale r* (*scale s 1*)
by (*simp only*: *scale-scale*)
also have ... = *scalar-embed r* ∗ *scalar-embed s*
by (*rule scale-eq-embed-mult*)
finally show *?thesis* **.**
qed

lemma *scalar-embed-of-nat* [*simp*]:
scalar-embed (*of-nat n*) = *of-nat n*
by (*induction n*) *simp-all*

lemma *scalar-embed-of-int* [*simp*]:

scalar-embed (*of-int z*) = *of-int z*
by (*cases z*) *simp-all*

lemma *scalar-embed-power* [*simp*]:
scalar-embed (*r* ^ *n*) = *scalar-embed r* ^ *n*
by (*induction n*) *simp-all*

end

locale *algebra-tower* =
RS: *commutative-algebra rS* +
SA: *commutative-algebra sA* +
RA: *commutative-algebra rA*
for *rS* :: *'r*::*comm-ring-1* ⇒ *'s*::*comm-ring-1* ⇒ *'s*
and *sA* :: *'s* ⇒ *'a*::*comm-ring-1* ⇒ *'a*
and *rA* :: *'r* ⇒ *'a* ⇒ *'a* +
assumes *scalar-compat* [*simp*]:
rA r (*sA s a*) = *sA* (*rS r s*) *a*
begin

sublocale *compatible-scalar-actions rS sA rA*
by *unfold-locales* (*fact scalar-compat*)

lemma *composed-scalar-embed* [*simp*]:
rA r 1 = *sA* (*rS r 1*) *1*
using *scalar-compat*[*of r 1 1*] **by** *simp*

lemma *scalar-compat-mult* [*simp*]:
rA r (*sA s a* ∗ *b*) = *sA* (*rS r s*) (*a* ∗ *b*)
proof −
have *rA r* (*sA s a* ∗ *b*) = *rA r* (*sA s a*) ∗ *b*
by (*rule RA.scale-mult-right*[*symmetric*])
also have ... = *sA* (*rS r s*) *a* ∗ *b*
by (*simp only*: *scalar-compat*)
also have ... = *sA* (*rS r s*) (*a* ∗ *b*)
by (*rule SA.scale-mult-right*)
finally show *?thesis* **.**
qed

lemma *scalar-compat-linear*:
assumes *module-hom sA sA f*
shows *f* (*rA r a*) = *rA r* (*f a*)
proof −
interpret *f*: *module-hom sA sA f*
by *fact*
have *f* (*rA r a*) = *f* (*sA* (*rS r 1*) *a*)
by *simp*
also have ... = *sA* (*rS r 1*) (*f a*)
by (*rule f.scale*)

also have ... = *rA r* (*f a*)
by *simp*
finally show *?thesis* **.**
qed

end

end
theory *Algebra-Towers*
imports *Compatible-Scalar-Actions*
begin

3 Scalar towers

The locale *algebra-tower* packages three existing module actions and the single additional compatibility law required by a tower. The remaining theories derive functorial, basis, dimension, trace, and norm interfaces from it.

end
theory *Restriction-of-Scalars*
imports *Algebra-Towers*
begin

4 Restriction of scalars

Linear maps in *HOL.Modules* are plain functions equipped with a locale proposition. Restricting scalars therefore changes only the proposition used to view a function, never the function itself.

definition *restrict-scalars* :: ('*a* ⇒ '*b*) ⇒ '*a* ⇒ '*b*
where *restrict-scalars f x* = *f x*

lemma *restrict-scalars-apply* [*simp*]:
restrict-scalars f x = *f x*
by (*simp add*: *restrict-scalars-def*)

definition *subspace-rel* :: '*a*::*ab-group-add set* ⇒ ('*a* × '*a*) *set*
where *subspace-rel U* = {(*x*, *y*). *x* − *y* ∈ *U*}

definition *subspace-coset* :: '*a*::*ab-group-add set* ⇒ '*a* ⇒ '*a set*
where *subspace-coset U x* = ((+) *x*) ‘ *U*

lemma *subspace-coset-class*:
subspace-coset U x = {*y*. (*y*, *x*) ∈ *subspace-rel U*}
proof (*rule equalityI*)
show *subspace-coset U x* ⊆ {*y*. (*y*, *x*) ∈ *subspace-rel U*}
proof

```
    fix y
    assume y ∈ subspace-coset U x
    then obtain u where u ∈ U y = x + u
      by (auto simp: subspace-coset-def)
    then show y ∈ {y. (y, x) ∈ subspace-rel U}
      by (simp add: subspace-rel-def)
  qed
  show {y. (y, x) ∈ subspace-rel U} ⊆ subspace-coset U x
  proof
    fix y
    assume y ∈ {y. (y, x) ∈ subspace-rel U}
    then have y − x ∈ U
      by (simp add: subspace-rel-def)
    then have x + (y − x) ∈ ((+) x) ‘ U
      by (rule imageI)
    then show y ∈ subspace-coset U x
      by (simp add: subspace-coset-def)
  qed
qed

context Modules.module
begin

lemma subspace-rel-refl:
  assumes subspace U
  shows (x, x) ∈ subspace-rel U
  using assms by (simp add: subspace-rel-def subspace-0)

lemma subspace-rel-sym:
  assumes U: subspace U
    and xy: (x, y) ∈ subspace-rel U
  shows (y, x) ∈ subspace-rel U
proof −
  have x − y ∈ U
    using xy by (simp add: subspace-rel-def)
  then have − (x − y) ∈ U
    by (rule subspace-neg[OF U])
  then show ?thesis
    by (simp add: subspace-rel-def)
qed

lemma subspace-rel-trans:
  assumes U: subspace U
    and xy: (x, y) ∈ subspace-rel U
    and yz: (y, z) ∈ subspace-rel U
  shows (x, z) ∈ subspace-rel U
proof −
  have x − y ∈ U and y − z ∈ U
    using xy yz by (simp-all add: subspace-rel-def)
```

```
  then have (x − y) + (y − z) ∈ U
    by (rule subspace-add[OF U])
  then show ?thesis
    by (simp add: subspace-rel-def)
qed

lemma subspace-rel-equiv:
  assumes U: subspace U
  shows equiv UNIV (subspace-rel U)
proof (rule equivI)
  show subspace-rel U ⊆ UNIV × UNIV
    by simp
  show refl-on UNIV (subspace-rel U)
    unfolding refl-on-def
    using subspace-rel-refl[OF U] by simp
  show sym (subspace-rel U)
  proof (unfold sym-def, intro allI impI)
    fix x y
    assume (x, y) ∈ subspace-rel U
    then show (y, x) ∈ subspace-rel U
      by (rule subspace-rel-sym[OF U])
  qed
  show trans (subspace-rel U)
  proof (unfold trans-def, intro allI impI)
    fix x y z
    assume (x, y) ∈ subspace-rel U (y, z) ∈ subspace-rel U
    then show (x, z) ∈ subspace-rel U
      by (rule subspace-rel-trans[OF U])
  qed
qed

end

locale scalar-tower-pair =
  source: compatible-scalar-actions rS sA rA +
  target: compatible-scalar-actions rS sB rB
  for rS :: 'r::comm-ring-1 ⇒ 's::comm-ring-1 ⇒ 's
    and sA :: 's ⇒ 'a::comm-ring-1 ⇒ 'a
    and rA :: 'r ⇒ 'a ⇒ 'a
    and sB :: 's ⇒ 'b::comm-ring-1 ⇒ 'b
    and rB :: 'r ⇒ 'b ⇒ 'b
begin

lemma restrict-scalars-module-hom:
  assumes module-hom sA sB f
  shows module-hom rA rB (restrict-scalars f)
proof −
  interpret f: module-hom sA sB f
    by fact
```

```
  show ?thesis
    by unfold-locales (simp-all add: f.add f.scale)
qed

lemma restrict-scalars-id [simp]:
  restrict-scalars id = id
  by (rule ext) simp

lemma restrict-scalars-comp [simp]:
  restrict-scalars (g ∘ f) =
    restrict-scalars g ∘ restrict-scalars f
  by (rule ext) simp

lemma restrict-scalars-inj [simp]:
  inj (restrict-scalars f) ⟷ inj f
  by (simp add: inj-def)

lemma restrict-scalars-surj [simp]:
  surj (restrict-scalars f) ⟷ surj f
  by (simp add: surj-def)

lemma restrict-scalars-bij [simp]:
  bij (restrict-scalars f) ⟷ bij f
  by (simp add: bij-def)

lemma restrict-scalars-kernel [simp]:
  {x. restrict-scalars f x = 0} = {x. f x = 0}
  by simp

lemma restrict-scalars-image [simp]:
  restrict-scalars f ‘ X = f ‘ X
  by auto

lemma restrict-scalars-range [simp]:
  range (restrict-scalars f) = range f
  by auto

lemma source-subspace-restrict-scalars:
  assumes U: source.SA.subspace U
  shows source.RA.subspace U
proof (rule source.RA.subspaceI)
  show 0 ∈ U
    by (rule source.SA.subspace-0[OF U])
  show x + y ∈ U if x ∈ U y ∈ U for x y
    using that by (rule source.SA.subspace-add[OF U])
  show rA r x ∈ U if x ∈ U for r x
  proof −
    have sA (rS r 1) x ∈ U
      using that by (rule source.SA.subspace-scale[OF U])
```

```
    then show ?thesis
      by simp
  qed
qed

lemma target-subspace-restrict-scalars:
  assumes U: target.SA.subspace U
  shows target.RA.subspace U
proof (rule target.RA.subspaceI)
  show 0 ∈ U
    by (rule target.SA.subspace-0[OF U])
  show x + y ∈ U if x ∈ U y ∈ U for x y
    using that by (rule target.SA.subspace-add[OF U])
  show rB r x ∈ U if x ∈ U for r x
  proof −
    have sB (rS r 1) x ∈ U
      using that by (rule target.SA.subspace-scale[OF U])
    then show ?thesis
      by simp
  qed
qed

lemma restrict-scalars-kernel-subspace:
  assumes f: module-hom sA sB f
  shows source.RA.subspace {x. restrict-scalars f x = 0}
proof −
  interpret rf: module-hom rA rB restrict-scalars f
    by (rule restrict-scalars-module-hom[OF f])
  show ?thesis
    by (rule rf.subspace-kernel)
qed

lemma restrict-scalars-image-subspace:
  assumes f: module-hom sA sB f
    and U: source.SA.subspace U
  shows target.RA.subspace (restrict-scalars f ‘ U)
proof −
  interpret rf: module-hom rA rB restrict-scalars f
    by (rule restrict-scalars-module-hom[OF f])
  show ?thesis
    by (rule rf.subspace-image[OF source-subspace-restrict-scalars[OF U]])
qed

lemma restrict-scalars-quotient:
  assumes f: module-hom sA sB f
    and maps: f ‘ U ⊆ V
    and xy: (x, y) ∈ subspace-rel U
  shows (restrict-scalars f x, restrict-scalars f y) ∈ subspace-rel V
proof −
```

```
  interpret f: module-hom sA sB f
    by fact
  have f (x − y) ∈ f ‘ U
    using xy by (auto simp: subspace-rel-def)
  then have f (x − y) ∈ V
    using maps by blast
  then show ?thesis
    by (simp add: subspace-rel-def f.diff)
qed

lemma restrict-scalars-coset-image:
  assumes module-hom sA sB f
  shows restrict-scalars f ‘ subspace-coset U x =
    subspace-coset (f ‘ U) (f x)
proof −
  interpret f: module-hom sA sB f
    by fact
  show ?thesis
  proof (rule equalityI)
    show restrict-scalars f ‘ subspace-coset U x ⊆
      subspace-coset (f ‘ U) (f x)
    proof
      fix y
      assume y ∈ restrict-scalars f ‘ subspace-coset U x
      then obtain z u where u ∈ U z = x + u
        and y = f z
        by (auto simp: subspace-coset-def)
      then have y = f x + f u
        by (simp add: f.add)
      moreover have f u ∈ f ‘ U
        using ‹u ∈ U› by blast
      ultimately show y ∈ subspace-coset (f ‘ U) (f x)
        by (auto simp: subspace-coset-def)
    qed
    show subspace-coset (f ‘ U) (f x) ⊆
      restrict-scalars f ‘ subspace-coset U x
    proof
      fix y
      assume y ∈ subspace-coset (f ‘ U) (f x)
      then obtain u where u ∈ U y = f x + f u
        by (auto simp: subspace-coset-def)
      then have y = restrict-scalars f (x + u)
        by (simp add: f.add)
      moreover have x + u ∈ subspace-coset U x
        using ‹u ∈ U› by (auto simp: subspace-coset-def)
      ultimately show y ∈ restrict-scalars f ‘ subspace-coset U x
        by blast
    qed
  qed
```

qed

end

locale *scalar-tower-pair-fields* =
  scalar-tower-pair rS sA rA sB rB
  for *rS* :: *'r*::*field* ⇒ *'s*::*field* ⇒ *'s*
    and *sA* :: *'s* ⇒ *'a*::*comm-ring-1* ⇒ *'a*
    and *rA* :: *'r* ⇒ *'a* ⇒ *'a*
    and *sB* :: *'s* ⇒ *'b*::*comm-ring-1* ⇒ *'b*
    and *rB* :: *'r* ⇒ *'b* ⇒ *'b*
begin

lemma *restrict-scalars-linear*:
  assumes *Vector-Spaces.linear sA sB f*
  shows *Vector-Spaces.linear rA rB* (*restrict-scalars f*)
  using *assms restrict-scalars-module-hom*
  by (*simp add*: *module-hom-iff-linear*)

end

end
theory *Tower-Bases*
  imports *Restriction-of-Scalars*
begin

5 Composition of bases

An indexed basis below is only a convenient locale wrapper around Isabelle's existing set-valued notions of independence and span. It adds no competing basis predicate.

locale *indexed-basis* =
  Modules.module scale
  for *scale* :: *'r*::*comm-ring-1* ⇒ *'a*::*ab-group-add* ⇒ *'a* +
  fixes *basis* :: *'i*::*finite* ⇒ *'a*
  assumes *basis-inj*: *inj basis*
    and *independent-basis*: *independent* (*range basis*)
    and *span-basis*: *span* (*range basis*) = *UNIV*
begin

definition *coordinate* :: *'a* ⇒ *'i* ⇒ *'r*
  where *coordinate x i* = *representation* (*range basis*) *x* (*basis i*)

lemma *basis-mem-span* [*simp*]:
  basis i ∈ *span* (*range basis*)
  by (*rule span-base*) *blast*

lemma *all-mem-span* [*simp*]:

$x \in span\ (range\ basis)$
using *span-basis* **by** *simp*

lemma *coordinate-zero* [*simp*]:
$coordinate\ 0\ i = 0$
by (*simp add*: *coordinate-def representation-zero*)

lemma *coordinate-add* [*simp*]:
$coordinate\ (x + y)\ i = coordinate\ x\ i + coordinate\ y\ i$
by (*simp add*: *coordinate-def representation-add independent-basis*)

lemma *coordinate-diff* [*simp*]:
$coordinate\ (x - y)\ i = coordinate\ x\ i - coordinate\ y\ i$
by (*simp add*: *coordinate-def representation-diff independent-basis*)

lemma *coordinate-scale* [*simp*]:
$coordinate\ (scale\ r\ x)\ i = r * coordinate\ x\ i$
by (*simp add*: *coordinate-def representation-scale independent-basis*)

lemma *coordinate-sum*:
$coordinate\ (\sum x{\in}X.\ f\ x)\ i =$
$(\sum x{\in}X.\ coordinate\ (f\ x)\ i)$
by (*induction X rule*: *infinite-finite-induct*) *simp-all*

lemma *coordinate-basis* [*simp*]:
$coordinate\ (basis\ i)\ j = (if\ j = i\ then\ 1\ else\ 0)$
proof −
have $representation\ (range\ basis)\ (basis\ i) =$
$(\lambda b.\ if\ b = basis\ i\ then\ 1\ else\ 0)$
by (*rule representation-basis*[*OF independent-basis*]) *blast*
then show *?thesis*
by (*simp add*: *coordinate-def basis-inj inj-eq*)
qed

lemma *basis-expansion*:
$(\sum i{\in}UNIV.\ scale\ (coordinate\ x\ i)\ (basis\ i)) = x$
proof −
have $(\sum b{\in}range\ basis.$
$scale\ (representation\ (range\ basis)\ x\ b)\ b) = x$
by (*rule sum-representation-eq*[*OF independent-basis*])
(*simp-all add*: *span-basis*)
also have $(\sum b{\in}range\ basis.$
$scale\ (representation\ (range\ basis)\ x\ b)\ b) =$
$(\sum i{\in}UNIV.\ scale\ (coordinate\ x\ i)\ (basis\ i))$
by (*simp add*: *coordinate-def sum.reindex*[*OF basis-inj*])
finally show *?thesis* **.**
qed

lemma *coordinate-eq-iff*:

```
  x = y ⟷ (∀ i. coordinate x i = coordinate y i)
proof
  show x = y ⟹ ∀ i. coordinate x i = coordinate y i
    by simp
next
  assume coordinates: ∀ i. coordinate x i = coordinate y i
  have (∑ i∈UNIV. scale (coordinate x i) (basis i)) =
    (∑ i∈UNIV. scale (coordinate y i) (basis i))
    using coordinates by simp
  then show x = y
    by (simp only: basis-expansion)
qed

end

locale indexed-algebra-tower =
  algebra-tower rS sA rA
  for rS :: 'r::comm-ring-1 ⇒ 's::comm-ring-1 ⇒ 's
    and sA :: 's ⇒ 'a::comm-ring-1 ⇒ 'a
    and rA :: 'r ⇒ 'a ⇒ 'a +
  fixes lower-basis :: 'i::finite ⇒ 's
    and upper-basis :: 'j::finite ⇒ 'a
  assumes lower-basis-inj: inj lower-basis
    and lower-basis-independent: RS.independent (range lower-basis)
    and lower-basis-span: RS.span (range lower-basis) = UNIV
    and upper-basis-inj: inj upper-basis
    and upper-basis-independent: SA.independent (range upper-basis)
    and upper-basis-span: SA.span (range upper-basis) = UNIV
begin

sublocale lower: indexed-basis rS lower-basis
  by unfold-locales
    (fact lower-basis-inj lower-basis-independent lower-basis-span)+

sublocale upper: indexed-basis sA upper-basis
  by unfold-locales
    (fact upper-basis-inj upper-basis-independent upper-basis-span)+

definition tower-basis :: ('i × 'j) ⇒ 'a
  where tower-basis ij =
    sA (lower-basis (fst ij)) (upper-basis (snd ij))

definition tower-coordinate :: 'a ⇒ ('i × 'j) ⇒ 'r
  where tower-coordinate x ij =
    lower.coordinate (upper.coordinate x (snd ij)) (fst ij)

lemma tower-coordinate-zero [simp]:
  tower-coordinate 0 ij = 0
  by (simp add: tower-coordinate-def)
```

lemma *tower-coordinate-add* [*simp*]:
tower-coordinate (*x* + *y*) *ij* =
tower-coordinate x ij + *tower-coordinate y ij*
by (*simp add*: *tower-coordinate-def*)

lemma *tower-coordinate-diff* [*simp*]:
tower-coordinate (*x* − *y*) *ij* =
tower-coordinate x ij − *tower-coordinate y ij*
by (*simp add*: *tower-coordinate-def*)

lemma *upper-coordinate-lower-scale*:
upper.coordinate (*rA r x*) *j* =
rS r (*upper.coordinate x j*)
proof −
have *upper.coordinate* (*rA r x*) *j* =
upper.coordinate (*sA* (*rS r 1*) *x*) *j*
by *simp*
also have ... = (*rS r 1*) ∗ *upper.coordinate x j*
by *simp*
also have ... = *rS r* (*upper.coordinate x j*)
by (*rule RS.scale-eq-embed-mult*[*symmetric*])
finally show *?thesis* **.**
qed

lemma *tower-coordinate-scale* [*simp*]:
tower-coordinate (*rA r x*) *ij* = *r* ∗ *tower-coordinate x ij*
by (*cases ij*)
(*simp only*: *tower-coordinate-def fst-conv snd-conv*
upper-coordinate-lower-scale lower.coordinate-scale)

lemma *tower-coordinate-sum*:
tower-coordinate (∑ *x*∈*X*. *f x*) *ij* =
(∑ *x*∈*X*. *tower-coordinate* (*f x*) *ij*)
by (*induction X rule*: *infinite-finite-induct*) *simp-all*

lemma *tower-coordinate-basis* [*simp*]:
tower-coordinate (*tower-basis ij*) *kl* =
(*if kl* = *ij then 1 else 0*)
by (*cases ij*; *cases kl*)
(*simp add*: *tower-coordinate-def tower-basis-def*)

lemma *tower-basis-inj*:
inj tower-basis
proof (*rule injI*)
fix *ij kl*
assume *equal*: *tower-basis ij* = *tower-basis kl*
have *tower-coordinate* (*tower-basis ij*) *ij* =
tower-coordinate (*tower-basis kl*) *ij*

```
    using equal by simp
  then show ij = kl
    by (cases ij = kl) simp-all
qed

lemma tower-basis-span:
  RA.span (range tower-basis) = UNIV
proof −
  have upper-term:
    sA (upper.coordinate x j) (upper-basis j) =
      (∑ i∈UNIV. rA
        (lower.coordinate (upper.coordinate x j) i)
        (tower-basis (i, j)))
    for x j
  proof −
    have sA (upper.coordinate x j) (upper-basis j) =
      sA (∑ i∈UNIV.
        rS (lower.coordinate (upper.coordinate x j) i) (lower-basis i))
        (upper-basis j)
      by (simp only: lower.basis-expansion)
    also have ... = (∑ i∈UNIV.
      sA (rS (lower.coordinate (upper.coordinate x j) i) (lower-basis i))
        (upper-basis j))
      by (rule SA.scale-sum-left)
    also have ... = (∑ i∈UNIV. rA
      (lower.coordinate (upper.coordinate x j) i)
      (tower-basis (i, j)))
    proof (rule sum.cong)
      show (UNIV :: 'i set) = UNIV
        by simp
      fix i
      assume i ∈ (UNIV :: 'i set)
      show sA (rS (lower.coordinate (upper.coordinate x j) i)
          (lower-basis i)) (upper-basis j) =
        rA (lower.coordinate (upper.coordinate x j) i)
          (tower-basis (i, j))
        unfolding tower-basis-def
        by (simp only: fst-conv snd-conv scalar-compat)
    qed
    finally show ?thesis .
  qed

  have expansion:
    (∑ j∈UNIV. ∑ i∈UNIV. rA
      (lower.coordinate (upper.coordinate x j) i)
      (tower-basis (i, j))) = x
    for x
  proof −
    have (∑ j∈UNIV. ∑ i∈UNIV. rA
```

```
    (lower.coordinate (upper.coordinate x j) i)
    (tower-basis (i, j))) =
    (∑ j∈UNIV. sA (upper.coordinate x j) (upper-basis j))
  proof (rule sum.cong)
    show (UNIV :: 'j set) = UNIV
      by simp
    fix j
    assume j ∈ (UNIV :: 'j set)
    show (∑ i∈UNIV. rA
      (lower.coordinate (upper.coordinate x j) i)
      (tower-basis (i, j))) =
      sA (upper.coordinate x j) (upper-basis j)
      by (rule upper-term[symmetric])
  qed
  also have ... = x
    by (rule upper.basis-expansion)
  finally show ?thesis .
qed

have inner-span:
  (∑ i∈UNIV. rA
    (lower.coordinate (upper.coordinate x j) i)
    (tower-basis (i, j))) ∈ RA.span (range tower-basis)
  for x j
proof (rule RA.span-sum)
  fix i
  assume i ∈ (UNIV :: 'i set)
  show rA (lower.coordinate (upper.coordinate x j) i)
    (tower-basis (i, j)) ∈ RA.span (range tower-basis)
    by (rule RA.span-scale, rule RA.span-base) blast
qed

have x ∈ RA.span (range tower-basis) for x
proof −
  have (∑ j∈UNIV. ∑ i∈UNIV. rA
    (lower.coordinate (upper.coordinate x j) i)
    (tower-basis (i, j))) ∈ RA.span (range tower-basis)
  proof (rule RA.span-sum)
    fix j
    assume j ∈ (UNIV :: 'j set)
    show (∑ i∈UNIV. rA
      (lower.coordinate (upper.coordinate x j) i)
      (tower-basis (i, j))) ∈ RA.span (range tower-basis)
      by (rule inner-span)
  qed
  then show ?thesis
    by (simp only: expansion)
qed
then show ?thesis
```

```
    by auto
qed

lemma tower-basis-independent:
  RA.independent (range tower-basis)
  unfolding RA.independent-explicit-module
proof (intro allI impI)
  fix T c w
  assume finite-T: finite T
    and subset: T ⊆ range tower-basis
    and sum-zero: (∑ x∈T. rA (c x) x) = 0
    and w: w ∈ T
  obtain ij where w-eq: w = tower-basis ij
    using subset w by blast
  have 0 = tower-coordinate (∑ x∈T. rA (c x) x) ij
    using sum-zero by simp
  also have ... =
    (∑ x∈T. tower-coordinate (rA (c x) x) ij)
    by (rule tower-coordinate-sum)
  also have ... = (∑ x∈T. c x * tower-coordinate x ij)
  proof (rule sum.cong)
    show T = T
      by simp
    fix x
    assume x ∈ T
    show tower-coordinate (rA (c x) x) ij =
      c x * tower-coordinate x ij
      by (rule tower-coordinate-scale)
  qed
  also have ... = c w
  proof −
    have summand:
      c x * tower-coordinate x ij = (if x = w then c w else 0)
      if x ∈ T for x
    proof (cases x = w)
      case True
      then show ?thesis
        by (simp add: w-eq)
    next
      case False
      have x ∈ range tower-basis
        using subset that by blast
      obtain kl where x-eq: x = tower-basis kl
        using ‹x ∈ range tower-basis› by blast
      have kl ≠ ij
        using False x-eq w-eq tower-basis-inj
        by (auto simp: inj-def)
      then show ?thesis
        using False by (simp add: x-eq)
```

```
    qed
    have (∑x∈T. c x * tower-coordinate x ij) =
      (∑x∈T. if x = w then c w else 0)
      by (rule sum.cong) (simp-all add: summand)
    also have ... = c w
      using finite-T w by simp
    finally show ?thesis .
  qed
  finally show c w = 0
    by simp
qed

theorem basis-tower:
  RA.independent (range tower-basis) ∧
    RA.span (range tower-basis) = UNIV
  using tower-basis-independent tower-basis-span by blast

sublocale composed: indexed-basis rA tower-basis
  by unfold-locales
    (fact tower-basis-inj tower-basis-independent tower-basis-span)+

lemma composed-coordinate:
  composed.coordinate x ij = tower-coordinate x ij
proof −
  have mapped: tower-coordinate
      (∑kl∈UNIV.
        rA (composed.coordinate x kl) (tower-basis kl)) ij =
    tower-coordinate x ij
    by (rule arg-cong[OF composed.basis-expansion])
  have coefficients: tower-coordinate
      (∑kl∈UNIV.
        rA (composed.coordinate x kl) (tower-basis kl)) ij =
    composed.coordinate x ij
  proof −
    have tower-coordinate
        (∑kl∈UNIV.
          rA (composed.coordinate x kl) (tower-basis kl)) ij =
      (∑kl∈UNIV.
        tower-coordinate
          (rA (composed.coordinate x kl) (tower-basis kl)) ij)
      by (rule tower-coordinate-sum)
    also have ... =
      (∑kl∈UNIV.
        composed.coordinate x kl *
          tower-coordinate (tower-basis kl) ij)
    proof (rule sum.cong)
      show (UNIV :: ('i × 'j) set) = UNIV
        by simp
      fix kl
```

```
    assume kl ∈ (UNIV :: ('i × 'j) set)
    show tower-coordinate
        (rA (composed.coordinate x kl) (tower-basis kl)) ij =
      composed.coordinate x kl *
        tower-coordinate (tower-basis kl) ij
      by (rule tower-coordinate-scale)
  qed
  also have ... =
    (∑ kl∈UNIV.
      if kl = ij then composed.coordinate x ij else 0)
  proof (rule sum.cong)
    show (UNIV :: ('i × 'j) set) = UNIV
      by simp
    fix kl
    assume kl ∈ (UNIV :: ('i × 'j) set)
    show composed.coordinate x kl *
        tower-coordinate (tower-basis kl) ij =
      (if kl = ij then composed.coordinate x ij else 0)
      by (cases kl = ij) simp-all
  qed
  also have ... = composed.coordinate x ij
    by simp
  finally show ?thesis .
 qed
 show ?thesis
   using mapped coefficients by simp
qed

end

end
theory Finite-Dimensional-Towers
 imports Tower-Bases
   Scalar-Restriction-Bridge
begin
```

6 Dimension in a tower

```
context indexed-algebra-tower
begin

theorem finite-rank-tower:
  card (range tower-basis) = card (range lower-basis) * card (range upper-basis)
proof −
 have tower-card: card (range tower-basis) = CARD('i × 'j)
   using card-image tower-basis-inj by blast
 have lower-card: card (range lower-basis) = CARD('i)
   using card-image lower-basis-inj by blast
 have upper-card: card (range upper-basis) = CARD('j)
```

```
    using card-image upper-basis-inj by blast
  show ?thesis
    by (simp add: tower-card lower-card upper-card)
qed

end

locale indexed-field-tower =
  indexed-algebra-tower rS sA rA lower-basis upper-basis
  for rS :: 'r::field ⇒ 's::field ⇒ 's
    and sA :: 's ⇒ 'a::comm-ring-1 ⇒ 'a
    and rA :: 'r ⇒ 'a ⇒ 'a
    and lower-basis :: 'i::finite ⇒ 's
    and upper-basis :: 'j::finite ⇒ 'a
begin

sublocale lower-space: vector-space rS
  using RS.module-axioms
  by (simp add: module-iff-vector-space)

sublocale upper-space: vector-space sA
  using SA.module-axioms
  by (simp add: module-iff-vector-space)

sublocale direct-space: vector-space rA
  using RA.module-axioms
  by (simp add: module-iff-vector-space)

theorem finrank-tower:
  direct-space.dim UNIV =
    lower-space.dim UNIV * upper-space.dim UNIV
proof −
  have lower-dim:
    card (range lower-basis) = lower-space.dim UNIV
    by (rule lower-space.basis-card-eq-dim)
      (simp-all add: lower-basis-span lower-basis-independent)
  have upper-dim:
    card (range upper-basis) = upper-space.dim UNIV
    by (rule upper-space.basis-card-eq-dim)
      (simp-all add: upper-basis-span upper-basis-independent)
  have direct-dim:
    card (range tower-basis) = direct-space.dim UNIV
    by (rule direct-space.basis-card-eq-dim)
      (simp-all add: tower-basis-span tower-basis-independent)
  show ?thesis
    using finite-rank-tower
    by (simp add: lower-dim upper-dim direct-dim)
qed
```

end

The carrier-based formulation permits the top carrier to be a vector subspace, including the zero space; it does not assume that the top carrier is a field. The two finite-basis assumptions express finite dimensionality of the intermediate field over the base and of the vector space over the intermediate field.

context *scalar-restriction-tower*
begin

corollary *embedded-field-dimension-tower*:
  assumes $\exists\, B.$ *lower.basis* B $\exists\, C.$ *upper.basis* C
  shows $\exists\, P.$ *direct.basis* P
    and *direct.dimension* $=$ *lower.dimension* $*$ *upper.dimension*
  using *finite-dimension-tower*[*OF assms*] **.**

end

end
theory *Regular-Representation*
  imports *Finite-Dimensional-Towers*
begin

7 Regular representation

context *commutative-algebra*
begin

definition *left-multiplication* :: $'a \Rightarrow 'a \Rightarrow 'a$
  where *left-multiplication a x* $=$ *a* $*$ *x*

lemma *left-multiplication-module-hom*:
  module-hom scale scale (*left-multiplication a*)
  by *unfold-locales*
    (*simp-all add*: *left-multiplication-def algebra-simps*)

lemma *left-multiplication-zero* [*simp*]:
  left-multiplication 0 x $=$ *0*
  by (*simp add*: *left-multiplication-def*)

lemma *left-multiplication-one* [*simp*]:
  left-multiplication 1 $=$ *id*
  by (*rule ext*) (*simp add*: *left-multiplication-def*)

lemma *left-multiplication-add*:
  left-multiplication (*a* $+$ *b*) *x* $=$
    left-multiplication a x $+$ *left-multiplication b x*
  by (*simp add*: *left-multiplication-def algebra-simps*)

lemma *left-multiplication-mult*:
left-multiplication $(a * b) =$
left-multiplication $a \circ$ *left-multiplication* b
by (*rule ext*) (*simp add*: *left-multiplication-def mult.assoc*)

definition *regular-representation* :: $'a \Rightarrow 'a \Rightarrow 'a$
where *regular-representation* $a =$ *left-multiplication* a

lemma *regular-representation-module-hom*:
module-hom scale scale (*regular-representation* a)
unfolding *regular-representation-def*
by (*rule left-multiplication-module-hom*)

lemma *regular-representation-add*:
regular-representation $(a + b)$ $x =$
regular-representation a $x +$ *regular-representation* b x
by (*simp add*: *regular-representation-def left-multiplication-add*)

lemma *regular-representation-mult*:
regular-representation $(a * b) =$
regular-representation $a \circ$ *regular-representation* b
by (*simp add*: *regular-representation-def left-multiplication-mult*)

lemma *regular-representation-one* [*simp*]:
regular-representation $1 = id$
by (*simp add*: *regular-representation-def*)

end

context *indexed-basis*
begin

definition *matrix-of* :: $('a \Rightarrow 'a) \Rightarrow 'r\hat{}'i\hat{}'i$
where *matrix-of* $f = (\chi\ i\ j.$ *coordinate* $(f\ (basis\ j))\ i)$

lemma *matrix-of-component* [*simp*]:
matrix-of $f\ \$\ i\ \$\ j =$ *coordinate* $(f\ (basis\ j))\ i$
by (*simp add*: *matrix-of-def*)

lemma *matrix-of-id* [*simp*]:
matrix-of id $=$ *mat 1*
by (*vector matrix-of-def mat-def coordinate-basis*)

lemma *matrix-of-zero* [*simp*]:
matrix-of $(\lambda$-. *0*$) = 0$
by (*vector matrix-of-def coordinate-zero*)

lemma *matrix-of-add*:

```
  matrix-of (λx. f x + g x) = matrix-of f + matrix-of g
  by (vector matrix-of-def coordinate-add)

lemma matrix-of-comp:
  assumes f: module-hom scale scale f
    and g: module-hom scale scale g
  shows matrix-of (g ∘ f) = matrix-of g ** matrix-of f
proof −
  interpret f: module-hom scale scale f
    by fact
  interpret g: module-hom scale scale g
    by fact
  have expand: g (f (basis j)) = (∑ k∈UNIV. scale (coordinate (f (basis j)) k) (g
(basis k)))
    for j
  proof −
    have g (f (basis j)) =
      g (∑ k∈UNIV. scale (coordinate (f (basis j)) k) (basis k))
      by (simp only: basis-expansion)
    also have ... =
      (∑ k∈UNIV. g (scale (coordinate (f (basis j)) k) (basis k)))
      by (rule g.sum)
    also have ... =
      (∑ k∈UNIV. scale (coordinate (f (basis j)) k) (g (basis k)))
      using g.scale by presburger
    finally show ?thesis .
  qed

  have entry: matrix-of (g ∘ f) $ i $ j = (matrix-of g ** matrix-of f) $ i $ j for
i j
  proof −
    have matrix-of (g ∘ f) $ i $ j = coordinate (∑ k∈UNIV. scale (coordinate (f
(basis j)) k) (g (basis k))) i
      by (simp add: matrix-of-def expand)
    also have ... = (∑ k∈UNIV. coordinate (scale (coordinate (f (basis j)) k) (g
(basis k))) i)
      by (rule coordinate-sum)
    also have ... = (∑ k∈UNIV. coordinate (g (basis k)) i * coordinate (f (basis
j)) k)
      by (simp add: mult.commute)
    also have ... = (matrix-of g ** matrix-of f) $ i $ j
      by (simp add: matrix-matrix-mult-def matrix-of-def)
    finally show ?thesis .
  qed

  show ?thesis
    using entry by (meson vec-eq-iff)
qed
```

end

locale *indexed-basis-pair* =
  first: *indexed-basis scale first-basis* +
  second: *indexed-basis scale second-basis*
  for *scale* :: *'r*::*comm-ring-1* ⇒ *'a*::*ab-group-add* ⇒ *'a*
    and *first-basis* :: *'i*::*finite* ⇒ *'a*
    and *second-basis* :: *'i* ⇒ *'a*
begin

definition *to-second* :: *'r^'i^'i*
  where *to-second* =
    (χ *i j*. *second.coordinate* (*first-basis j*) *i*)

definition *to-first* :: *'r^'i^'i*
  where *to-first* =
    (χ *i j*. *first.coordinate* (*second-basis j*) *i*)

definition *first-to-second-matrix* :: (*'a* ⇒ *'a*) ⇒ *'r^'i^'i*
  where *first-to-second-matrix f* =
    (χ *i j*. *second.coordinate* (*f* (*first-basis j*)) *i*)

lemma *to-second-component* [*simp*]:
  to-second $ *i* $ *j* = *second.coordinate* (*first-basis j*) *i*
  by (*simp add*: *to-second-def*)

lemma *to-first-component* [*simp*]:
  to-first $ *i* $ *j* = *first.coordinate* (*second-basis j*) *i*
  by (*simp add*: *to-first-def*)

lemma *first-to-second-component* [*simp*]:
  first-to-second-matrix f $ *i* $ *j* =
    second.coordinate (*f* (*first-basis j*)) *i*
  by (*simp add*: *first-to-second-matrix-def*)

lemma *to-second-to-first*:
  to-second ** *to-first* = *mat 1*
proof −
  have *entry*: (*to-second* ** *to-first*) $ *i* $ *j* = (*mat 1* :: *'r^'i^'i*) $ *i* $ *j* **for** *i j*
  proof −
    have (*to-second* ** *to-first*) $ *i* $ *j* = (∑ *k*∈*UNIV*. *second.coordinate* (*first-basis k*) *i* * *first.coordinate* (*second-basis j*) *k*)
      by (*simp add*: *matrix-matrix-mult-def*)
    also have ... = (∑ *k*∈*UNIV*. *first.coordinate* (*second-basis j*) *k* * *second.coordinate* (*first-basis k*) *i*)
      by (*simp add*: *mult.commute*)
    also have ... = *second.coordinate* (∑ *k*∈*UNIV*. *scale* (*first.coordinate* (*second-basis j*) *k*) (*first-basis k*)) *i*
      by (*simp add*: *second.coordinate-sum*)

```
    also have ... = (mat 1 :: 'r^'i^'i) $ i $ j
      by (simp add: first.basis-expansion mat-def)
    finally show ?thesis .
  qed
  show ?thesis
    by (simp add: entry vec-eq-iff)
qed

lemma to-first-to-second:
  to-first ** to-second = mat 1
proof −
  have entry:
    (to-first ** to-second) $ i $ j = (mat 1 :: 'r^'i^'i) $ i $ j
    for i j
  proof −
    have (to-first ** to-second) $ i $ j =
      (∑ k∈UNIV.
        first.coordinate (second-basis k) i *
        second.coordinate (first-basis j) k)
      by (simp add: matrix-matrix-mult-def)
   also have ... = (∑ k∈UNIV. second.coordinate (first-basis j) k * first.coordinate
(second-basis k) i)
      by (simp add: mult.commute)
   also have ... = first.coordinate (∑ k∈UNIV. scale (second.coordinate (first-basis
j) k) (second-basis k)) i
      by (simp add: first.coordinate-sum)
    also have ... = (mat 1 :: 'r^'i^'i) $ i $ j
      by (simp add: second.basis-expansion mat-def)
    finally show ?thesis .
  qed
  show ?thesis
    by (simp add: entry vec-eq-iff)
qed

lemma first-to-second-matrix-left:
  assumes f: module-hom scale scale f
  shows first-to-second-matrix f =
    to-second ** first.matrix-of f
proof −
  interpret f: module-hom scale scale f
    by fact
  have entry: first-to-second-matrix f $ i $ j = (to-second ** first.matrix-of f) $ i
$ j for i j
  proof −
    have first-to-second-matrix f $ i $ j =
        second.coordinate (∑ k∈UNIV. scale (first.coordinate (f (first-basis j)) k)
(first-basis k)) i
      by (simp add: first.basis-expansion)
   also have ... = (∑ k∈UNIV. second.coordinate (first-basis k) i * first.coordinate
```

```
(f (first-basis j)) k)
      by (simp add: second.coordinate-sum mult.commute)
    also have ... = (to-second ** first.matrix-of f) $ i $ j
      by (simp add: matrix-matrix-mult-def)
    finally show ?thesis .
  qed
  show ?thesis
    using entry by (meson vec-eq-iff)
qed

lemma second-matrix-from-mixed:
  assumes f: module-hom scale scale f
  shows second.matrix-of f =
    first-to-second-matrix f ** to-first
proof −
  interpret f: module-hom scale scale f
    by fact
  have entry: second.matrix-of f $ i $ j = (first-to-second-matrix f ** to-first) $ i
$ j for i j
  proof −
    have second.matrix-of f $ i $ j =
       second.coordinate (f (∑ k∈UNIV. scale (first.coordinate (second-basis j) k)
(first-basis k))) i
      by (simp add: first.basis-expansion)
    also have ... =
       second.coordinate (∑ k∈UNIV. scale (first.coordinate (second-basis j) k) (f
(first-basis k))) i
      by (simp add: f.sum f.scale)
   also have ... = (∑ k∈UNIV. second.coordinate (f (first-basis k)) i * first.coordinate
(second-basis j) k)
      by (simp add: second.coordinate-sum mult.commute)
    also have ... = (first-to-second-matrix f ** to-first) $ i $ j
      by (simp add: matrix-matrix-mult-def)
    finally show ?thesis .
  qed
  show ?thesis
    using entry by (meson vec-eq-iff)
qed

theorem matrix-change-basis:
  assumes module-hom scale scale f
  shows second.matrix-of f =
    (to-second ** first.matrix-of f) ** to-first
  using second-matrix-from-mixed[OF assms]
    first-to-second-matrix-left[OF assms]
  by simp

end
```

locale *indexed-algebra* =
  commutative-algebra scale
  for *scale* :: *'r*::*comm-ring-1* ⇒ *'a*::*comm-ring-1* ⇒ *'a* +
  fixes *basis* :: *'i*::*finite* ⇒ *'a*
  assumes *basis-inj*: *inj basis*
    and *basis-independent*: *independent* (*range basis*)
    and *basis-span*: *span* (*range basis*) = *UNIV*
begin

sublocale *coordinates*: *indexed-basis scale basis*
  by *unfold-locales* (*fact basis-inj basis-independent basis-span*)+

definition *regular-matrix* :: *'a* ⇒ *'r*^*'i*^*'i*
  where *regular-matrix a* =
    coordinates.matrix-of (*left-multiplication a*)

lemma *regular-matrix-zero* [*simp*]:
  regular-matrix 0 = *0*
  by (*vector regular-matrix-def left-multiplication-def*
    coordinates.matrix-of-def coordinates.coordinate-zero)

lemma *regular-matrix-add*:
  regular-matrix (*a* + *b*) = *regular-matrix a* + *regular-matrix b*
  by (*vector regular-matrix-def left-multiplication-def*
    coordinates.matrix-of-def coordinates.coordinate-add algebra-simps)

lemma *regular-matrix-scale*:
  regular-matrix (*scale r a*) =
    (χ *i j*. *r* ∗ *regular-matrix a* $ *i* $ *j*)
  by (*vector regular-matrix-def left-multiplication-def*
    coordinates.matrix-of-def coordinates.coordinate-scale
    scale-mult-right)

lemma *regular-matrix-one* [*simp*]:
  regular-matrix 1 = *mat 1*
  unfolding *regular-matrix-def left-multiplication-one*
  by (*rule coordinates.matrix-of-id*)

lemma *regular-matrix-scalar-embed*:
  regular-matrix (*scale r 1*) = *mat r*
  by (*vector regular-matrix-def left-multiplication-def*
    coordinates.matrix-of-def coordinates.coordinate-scale
    coordinates.coordinate-basis mat-def scale-mult-right)

lemma *regular-matrix-mult*:
  regular-matrix (*a* ∗ *b*) = *regular-matrix a* ∗∗ *regular-matrix b*
proof −
  have *regular-matrix* (*a* ∗ *b*) =
    coordinates.matrix-of

```
    (left-multiplication a ∘ left-multiplication b)
   by (simp add: regular-matrix-def left-multiplication-mult)
  also have ... =
   coordinates.matrix-of (left-multiplication a) **
    coordinates.matrix-of (left-multiplication b)
   by (rule coordinates.matrix-of-comp)
    (rule left-multiplication-module-hom)+
  finally show ?thesis
   by (simp add: regular-matrix-def)
qed

end
```

Finite basis index types are bookkeeping rather than mathematical data. The following locale transports an indexed algebra structure along an explicit bijection between finite index types, so later trace and determinant results can compare bases with genuinely different index types.

```
locale indexed-algebra-reindex =
  original: indexed-algebra scale original-basis
  for scale :: 'r::comm-ring-1 ⇒ 'a::comm-ring-1 ⇒ 'a
   and original-basis :: 'j::finite ⇒ 'a +
  fixes index-iso :: 'i::finite ⇒ 'j
  assumes index-iso-bij: bij index-iso
begin

definition reindexed-basis :: 'i ⇒ 'a
  where reindexed-basis i = original-basis (index-iso i)

lemma reindexed-basis-range:
  range reindexed-basis = range original-basis
proof
  show range reindexed-basis ⊆ range original-basis
   by (auto simp: reindexed-basis-def)
next
  show range original-basis ⊆ range reindexed-basis
  proof
    fix x
    assume x ∈ range original-basis
    then obtain j where x: x = original-basis j
     by blast
    have surj: surj index-iso
     using index-iso-bij by (rule bij-is-surj)
    have ∃ i. j = index-iso i
     using surj by (simp add: surj-def)
    then obtain i where i: j = index-iso i
     by blast
    have reindexed-basis i = original-basis j
     by (simp add: reindexed-basis-def i)
    have reindexed-basis i ∈ range reindexed-basis
```

```
    by (rule rangeI)
  show x ∈ range reindexed-basis
    using x ‹reindexed-basis i = original-basis j›
      ‹reindexed-basis i ∈ range reindexed-basis›
    by simp
 qed
qed

lemma reindexed-basis-inj:
 inj reindexed-basis
 using original.basis-inj index-iso-bij
 by (auto simp: reindexed-basis-def bij-def inj-def)

lemma reindexed-basis-independent:
 original.independent (range reindexed-basis)
 by (simp add: reindexed-basis-range original.basis-independent)

lemma reindexed-basis-span:
 original.span (range reindexed-basis) = UNIV
 by (simp add: reindexed-basis-range original.basis-span)

sublocale reindexed: indexed-algebra scale reindexed-basis
 by unfold-locales
  (fact reindexed-basis-inj reindexed-basis-independent
   reindexed-basis-span)+

lemma coordinate-reindexed:
 reindexed.coordinates.coordinate x i =
  original.coordinates.coordinate x (index-iso i)
 unfolding reindexed.coordinates.coordinate-def
  original.coordinates.coordinate-def
 apply (subst reindexed-basis-range)
 apply (simp add: reindexed-basis-def)
 done

lemma regular-matrix-reindexed-component [simp]:
 reindexed.regular-matrix a $ i $ k =
  original.regular-matrix a $ index-iso i $ index-iso k
 by (simp add: reindexed.regular-matrix-def
  original.regular-matrix-def reindexed.coordinates.matrix-of-def
  original.coordinates.matrix-of-def coordinate-reindexed
  reindexed-basis-def)

end

locale indexed-algebra-isomorphism =
 source: indexed-algebra source-scale source-basis +
 target: indexed-algebra target-scale target-basis
 for source-scale :: 'r::comm-ring-1 ⇒ 'a::comm-ring-1 ⇒ 'a
```

```
    and source-basis :: 'i::finite ⇒ 'a
    and target-scale :: 'r ⇒ 'b::comm-ring-1 ⇒ 'b
    and target-basis :: 'i ⇒ 'b +
  fixes iso :: 'a ⇒ 'b
  assumes iso-module-hom: module-hom source-scale target-scale iso
    and iso-mult: iso (x * y) = iso x * iso y
    and iso-one: iso 1 = 1
    and iso-bij: bij iso
    and iso-basis: iso (source-basis i) = target-basis i
begin

interpretation iso: module-hom source-scale target-scale iso
  by (rule iso-module-hom)

lemma iso-basis-expansion:
  (∑ i∈UNIV. target-scale (source.coordinates.coordinate x i) (target-basis i)) =
iso x
  by (simp add: source.coordinates.basis-expansion flip: iso-basis iso.sum iso.scale)

lemma coordinate-iso:
  target.coordinates.coordinate (iso x) i =
    source.coordinates.coordinate x i
proof −
  have mapped: target.coordinates.coordinate
      (∑ j∈UNIV.
        target-scale (source.coordinates.coordinate x j) (target-basis j)) i =
    target.coordinates.coordinate (iso x) i
    by (rule arg-cong[OF iso-basis-expansion])
  have coefficients: target.coordinates.coordinate
      (∑ j∈UNIV. target-scale (source.coordinates.coordinate x j) (target-basis j)) i
=
    source.coordinates.coordinate x i
  proof −
    have target.coordinates.coordinate
        (∑ j∈UNIV. target-scale (source.coordinates.coordinate x j) (target-basis j))
i =
      (∑ j∈UNIV. source.coordinates.coordinate x j * target.coordinates.coordinate
(target-basis j) i)
      by (simp add: target.coordinates.coordinate-sum)
     also have ... = (∑ j∈UNIV. if j = i then source.coordinates.coordinate x i
else 0)
     using target.coordinates.coordinate-basis by (metis mult.comm-neutral mult-not-zero)
    also have ... = source.coordinates.coordinate x i
      by simp
    finally show ?thesis .
  qed
  show ?thesis
    using mapped coefficients by simp
qed
```

```
lemma regular-matrix-iso:
  target.regular-matrix (iso a) = source.regular-matrix a
proof −
  have entry:
    target.regular-matrix (iso a) $ i $ j =
      source.regular-matrix a $ i $ j for i j
  proof −
    have target.regular-matrix (iso a) $ i $ j =
      target.coordinates.coordinate (iso a ∗ target-basis j) i
      by (simp add: target.regular-matrix-def
        target.left-multiplication-def)
    also have ... =
      target.coordinates.coordinate
        (iso a ∗ iso (source-basis j)) i
      by (simp only: iso-basis)
    also have ... =
      target.coordinates.coordinate (iso (a ∗ source-basis j)) i
      by (simp only: iso-mult)
    also have ... =
      source.coordinates.coordinate (a ∗ source-basis j) i
      by (rule coordinate-iso)
    also have ... = source.regular-matrix a $ i $ j
      by (simp add: source.regular-matrix-def
        source.left-multiplication-def)
    finally show ?thesis .
  qed
  show ?thesis
    by (simp add: entry vec-eq-iff)
qed

end

locale indexed-algebra-isomorphism-general =
  source: indexed-algebra source-scale source-basis +
  target: indexed-algebra target-scale target-basis
  for source-scale :: 'r::comm-ring-1 ⇒ 'a::comm-ring-1 ⇒ 'a
    and source-basis :: 'i::finite ⇒ 'a
    and target-scale :: 'r ⇒ 'b::comm-ring-1 ⇒ 'b
    and target-basis :: 'j::finite ⇒ 'b +
  fixes iso :: 'a ⇒ 'b
    and index-iso :: 'i ⇒ 'j
  assumes iso-module-hom: module-hom source-scale target-scale iso
    and iso-mult: iso (x ∗ y) = iso x ∗ iso y
    and iso-one: iso 1 = 1
    and iso-bij: bij iso
    and index-iso-bij: bij index-iso
begin
```

interpretation *iso*: *module-hom source-scale target-scale iso*
by (*rule iso-module-hom*)

definition *transported-basis* :: *'i* ⇒ *'b*
where *transported-basis i* = *iso* (*source-basis i*)

lemma *transported-basis-range*:
range transported-basis = *iso* ‘ *range source-basis*
by (*auto simp*: *transported-basis-def*)

lemma *transported-basis-inj*:
inj transported-basis
using *source.basis-inj iso-bij*
by (*auto simp*: *transported-basis-def bij-def inj-def*)

lemma *transported-basis-independent*:
target.independent (*range transported-basis*)
unfolding *transported-basis-range*
apply (*rule iso.independent-inj-image*)
apply (*fact source.basis-independent*)
using *iso-bij*
by (*simp add*: *bij-def*)

lemma *transported-basis-span*:
target.span (*range transported-basis*) = *UNIV*
proof −
have *target.span* (*range transported-basis*) =
iso ‘ *source.span* (*range source-basis*)
by (*simp only*: *transported-basis-range iso.span-image*)
also have . . . = *iso* ‘ *UNIV*
by (*simp only*: *source.basis-span*)
also have . . . = *UNIV*
using *iso-bij* **by** (*simp add*: *bij-def*)
finally show *?thesis* **.**
qed

sublocale *transported*: *indexed-algebra target-scale transported-basis*
by *unfold-locales*
(*fact transported-basis-inj transported-basis-independent*
transported-basis-span)+

sublocale *basis-iso*: *indexed-algebra-isomorphism*
source-scale source-basis target-scale transported-basis iso
apply *unfold-locales*
apply (*fact iso-mult*)
apply (*fact iso-one*)
apply (*fact iso-bij*)
apply (*simp add*: *transported-basis-def*)
done

sublocale *reindexed-target*:
indexed-algebra-reindex target-scale target-basis index-iso
by *unfold-locales* (*fact index-iso-bij*)

end

end
theory *Algebra-Trace*
imports *Regular-Representation*
begin

8 Algebra trace

context *indexed-algebra*
begin

definition *algebra-trace* :: $'a \Rightarrow 'r$
where *algebra-trace a* = *trace* (*regular-matrix a*)

lemma *algebra-trace-zero* [*simp*]:
algebra-trace 0 = *0*
by (*simp add*: *algebra-trace-def trace-def*)

lemma *algebra-trace-add*:
algebra-trace (*a* + *b*) = *algebra-trace a* + *algebra-trace b*
by (*simp add*: *algebra-trace-def regular-matrix-add trace-add*)

lemma *algebra-trace-sum*:
algebra-trace ($\sum x{\in}X.$ *f x*) =
($\sum x{\in}X.$ *algebra-trace* (*f x*))
by (*induction X rule*: *infinite-finite-induct*)
(*simp-all add*: *algebra-trace-add*)

lemma *algebra-trace-scale*:
algebra-trace (*scale r a*) = *r* ∗ *algebra-trace a*
by (*simp add*: *algebra-trace-def regular-matrix-scale trace-def*
sum-distrib-left)

theorem *algebra-trace-linear*:
module-hom scale ((∗) :: $'r \Rightarrow 'r \Rightarrow 'r$) *algebra-trace*
by *unfold-locales*
(*simp-all add*: *algebra-trace-add algebra-trace-scale algebra-simps*)

lemma *algebra-trace-scalar*:
algebra-trace (*scale r 1*) = *of-nat CARD*($'i$) ∗ *r*
by (*simp add*: *algebra-trace-def regular-matrix-scalar-embed*
trace-def mat-def)

end

context *indexed-basis-pair*
begin

theorem *trace-matrix-change-basis*:
assumes *module-hom scale scale f*
shows *trace* (*second.matrix-of f*) = *trace* (*first.matrix-of f*)
proof −
have *trace* (*second.matrix-of f*) =
trace ((*to-second* ∗∗ *first.matrix-of f*) ∗∗ *to-first*)
by (*simp only*: *matrix-change-basis*[*OF assms*])
also have ... = *trace* (*to-first* ∗∗ (*to-second* ∗∗ *first.matrix-of f*))
by (*rule trace-mul-sym*)
also have ... =
trace ((*to-first* ∗∗ *to-second*) ∗∗ *first.matrix-of f*)
by (*simp only*: *matrix-mul-assoc*)
also have ... = *trace* (*first.matrix-of f*)
by (*simp only*: *to-first-to-second matrix-mul-lid*)
finally show *?thesis* **.**
qed

end

context *indexed-algebra-reindex*
begin

theorem *algebra-trace-reindexed*:
reindexed.algebra-trace a = *original.algebra-trace a*
unfolding *reindexed.algebra-trace-def original.algebra-trace-def*
trace-def
apply (*simp only*: *regular-matrix-reindexed-component*)
apply (*rule sum.reindex-bij-betw*)
using *index-iso-bij*
by (*simp add*: *bij-betw-def bij-def*)

end

context *indexed-algebra*
begin

theorem *algebra-trace-basis-independent*:
fixes *other-basis* :: *'i* ⇒ *'a*
assumes *other-inj*: *inj other-basis*
and *other-independent*: *independent* (*range other-basis*)
and *other-span*: *span* (*range other-basis*) = *UNIV*
shows *algebra-trace a* =
indexed-algebra.algebra-trace scale other-basis a
proof −

```
  interpret other: indexed-algebra scale other-basis
    by unfold-locales
      (fact other-inj other-independent other-span)+
  interpret change: indexed-basis-pair scale basis other-basis
    by unfold-locales
      (fact basis-inj basis-independent basis-span
        other-inj other-independent other-span)+
  have trace (other.regular-matrix a) = trace (regular-matrix a)
    unfolding other.regular-matrix-def regular-matrix-def
    by (rule change.trace-matrix-change-basis)
      (rule left-multiplication-module-hom)
  then show ?thesis
    by (simp add: algebra-trace-def other.algebra-trace-def)
qed

theorem algebra-trace-basis-independent-bij:
  fixes other-basis :: 'j::finite ⇒ 'a
    and index-iso :: 'i ⇒ 'j
  assumes other-inj: inj other-basis
    and other-independent: independent (range other-basis)
    and other-span: span (range other-basis) = UNIV
    and index-iso-bij: bij index-iso
  shows algebra-trace a =
    indexed-algebra.algebra-trace scale other-basis a
proof −
  interpret other: indexed-algebra scale other-basis
    by unfold-locales
      (fact other-inj other-independent other-span)+
  interpret reindex:
    indexed-algebra-reindex scale other-basis index-iso
    by unfold-locales
      (fact other-inj other-independent other-span index-iso-bij)+
  have algebra-trace a = reindex.reindexed.algebra-trace a
    by (rule algebra-trace-basis-independent)
      (fact reindex.reindexed.basis-inj
        reindex.reindexed.basis-independent
        reindex.reindexed.basis-span)+
  also have ... = other.algebra-trace a
    by (rule reindex.algebra-trace-reindexed)
  finally show ?thesis .
qed

end

context indexed-algebra-isomorphism
begin

theorem algebra-trace-iso:
  target.algebra-trace (iso a) = source.algebra-trace a
```

by (*simp add*: *target.algebra-trace-def source.algebra-trace-def*
regular-matrix-iso)

end

context *indexed-algebra-isomorphism-general*
begin

theorem *algebra-trace-iso*:
target.algebra-trace (*iso a*) = *source.algebra-trace a*
proof −
have *basis-change*:
transported.algebra-trace (*iso a*) =
target.algebra-trace (*iso a*)
by (*rule transported.algebra-trace-basis-independent-bij*)
(*fact target.basis-inj target.basis-independent*
target.basis-span index-iso-bij)+
have *transported*:
transported.algebra-trace (*iso a*) =
source.algebra-trace a
by (*rule basis-iso.algebra-trace-iso*)
show *?thesis*
using *basis-change transported* **by** *simp*
qed

end

context *indexed-algebra-tower*
begin

sublocale *lower-algebra*: *indexed-algebra rS lower-basis*
by *unfold-locales*
(*fact lower-basis-inj lower-basis-independent lower-basis-span*)+

sublocale *upper-algebra*: *indexed-algebra sA upper-basis*
by *unfold-locales*
(*fact upper-basis-inj upper-basis-independent upper-basis-span*)+

sublocale *tower-algebra*: *indexed-algebra rA tower-basis*
by *unfold-locales*
(*fact tower-basis-inj tower-basis-independent tower-basis-span*)+

lemma *restricted-matrix-diagonal*:
assumes *f*: *module-hom sA sA f*
shows *composed.matrix-of f* $ (*i*, *j*) $ (*i*, *j*) = *lower-algebra.regular-matrix*
(*upper.matrix-of f* $ *j* $ *j*) $ *i* $ *i*
proof −
interpret *f*: *module-hom sA sA f*
by *fact*

```
  have composed.matrix-of f $ (i, j) $ (i, j) =
    composed.coordinate
      (f (tower-basis (i, j))) (i, j)
    by simp
  also have ... = tower-coordinate
      (f (tower-basis (i, j))) (i, j)
    by (rule composed-coordinate)
  also have ... =
    tower-coordinate
      (f (sA (lower-basis i) (upper-basis j))) (i, j)
    by (simp only: tower-basis-def fst-conv snd-conv)
  also have ... =
    tower-coordinate
      (sA (lower-basis i) (f (upper-basis j))) (i, j)
    by (simp only: f.scale)
  also have ... =
    lower.coordinate
      (lower-basis i *
        upper.coordinate (f (upper-basis j)) j) i
    by (simp add: tower-coordinate-def)
  also have ... =
    lower.coordinate
      (upper.coordinate (f (upper-basis j)) j *
        lower-basis i) i
    by (simp only: mult.commute)
  also have ... =
    lower-algebra.regular-matrix
      (upper.matrix-of f $ j $ j) $ i $ i
    by (simp add: lower-algebra.regular-matrix-def
      RS.left-multiplication-def)
  finally show ?thesis .
qed

theorem trace-restrict-scalars:
  assumes f: module-hom sA sA f
  shows trace (composed.matrix-of f) =
    lower-algebra.algebra-trace
      (trace (upper.matrix-of f))
proof −
  have pair-sum: (∑ ij∈(UNIV :: ('i × 'j) set). composed.matrix-of f $ ij $ ij) =
    (∑ j∈UNIV. ∑ i∈UNIV. composed.matrix-of f $ (i, j) $ (i, j))
   using sum.swap by (force simp add: sum.cartesian-product' simp flip: UNIV-Times-UNIV)
  have trace (composed.matrix-of f) = (∑ j∈UNIV. ∑ i∈UNIV. composed.matrix-of
f $ (i, j) $ (i, j))
    by (simp only: trace-def pair-sum)
  also have ... = (∑ j∈UNIV. ∑ i∈UNIV. lower-algebra.regular-matrix (upper.matrix-of
f $ j $ j) $ i $ i)
    using f restricted-matrix-diagonal by presburger
  also have ... = (∑ j∈UNIV. lower-algebra.algebra-trace (upper.matrix-of f $ j
```

```
$ j))
    by (simp add: lower-algebra.algebra-trace-def trace-def)
  also have ... = lower-algebra.algebra-trace (∑ j∈UNIV. upper.matrix-of f $ j
$ j)
    by (rule lower-algebra.algebra-trace-sum[symmetric])
  also have ... = lower-algebra.algebra-trace (trace (upper.matrix-of f))
    by (simp only: trace-def)
  finally show ?thesis .
qed

theorem algebra-trace-trans:
  tower-algebra.algebra-trace a =
    lower-algebra.algebra-trace (upper-algebra.algebra-trace a)
  unfolding tower-algebra.algebra-trace-def
    upper-algebra.algebra-trace-def
    tower-algebra.regular-matrix-def
    upper-algebra.regular-matrix-def
    RA.left-multiplication-def
    SA.left-multiplication-def
  by (rule trace-restrict-scalars)
    (rule SA.left-multiplication-module-hom[
      unfolded SA.left-multiplication-def])

end

end
theory Algebra-Norm
  imports Algebra-Trace
begin
```

9 Algebra norm

Simultaneously reindexing the rows and columns of a matrix along a bijection does not change its determinant. Isabelle's *map-permutation* transports the permutations in the Leibniz formula and preserves their signs.

```
lemma bij-betw-map-permutations:
  assumes h: bij-betw h A B
  shows bij-betw (map-permutation A h)
    {p. p permutes A} {q. q permutes B}
  apply (rule bij-betwI[where g = map-permutation B (inv-into A h)])
  apply (simp add: h map-permutation-permutes)
  apply (simp add: bij-betw-inv-into h map-permutation-permutes)
  using bij-betw-imp-inj-on h map-permutation-compose-inv apply fastforce
  using bij-betw-inv-into bij-betw-inv-into-right h map-permutation-compose-inv by
fastforce

theorem det-reindex-bij:
  fixes A :: 'r::comm-ring-1^'j::finite^'j
```

```
    and index-iso :: 'i::finite ⇒ 'j
  assumes index-iso-bij: bij index-iso
  shows det (χ i k. A $ index-iso i $ index-iso k) = det A
proof −
  let ?P = {p :: 'i ⇒ 'i. p permutes UNIV}
  let ?Q = {q :: 'j ⇒ 'j. q permutes UNIV}
  let ?F = map-permutation (UNIV :: 'i set) index-iso
  have index-betw:
    bij-betw index-iso (UNIV :: 'i set) (UNIV :: 'j set)
    using index-iso-bij
    by (simp add: bij-betw-def bij-def)
  have index-inj: inj-on index-iso (UNIV :: 'i set)
    using index-betw by (simp add: bij-betw-def)
  have permutation-betw: bij-betw ?F ?P ?Q
    by (rule bij-betw-map-permutations[OF index-betw])
  have sign-reindexed:
    sign (?F p) = sign p if p ∈ ?P for p
  proof −
    have p permutes (UNIV :: 'i set)
      using that by simp
    show ?thesis
      apply (rule sign-map-permutation)
       apply (fact index-inj)
      apply fact
      apply simp
      done
  qed
  have product-reindexed:
    (∏ i∈UNIV.
        A $ index-iso i $ index-iso (p i)) =
      (∏ j∈UNIV. A $ j $ ?F p j)
    for p
  proof −
    have (∏ i∈UNIV.
        A $ index-iso i $ ?F p (index-iso i)) =
      (∏ j∈UNIV. A $ j $ ?F p j)
      by (rule prod.reindex-bij-betw[OF index-betw])
    then show ?thesis
      using index-inj
      by (simp add: map-permutation-apply)
  qed
  have summand-reindexed:
    (of-int (sign p) :: 'r) *
       (∏ i∈UNIV.
          A $ index-iso i $ index-iso (p i)) =
      of-int (sign (?F p)) *
        (∏ j∈UNIV. A $ j $ ?F p j)
    if p ∈ ?P for p
    using sign-reindexed[OF that] product-reindexed
```

```
  by simp
 have sum-reindexed:
  (∑ p∈?P.
     (of-int (sign p) :: 'r) *
       (∏ i∈UNIV.
         A $ index-iso i $ index-iso (p i))) =
    (∑ q∈?Q.
      of-int (sign q) * (∏ j∈UNIV. A $ j $ q j))
 proof −
  have (∑ p∈?P.
     (of-int (sign p) :: 'r) *
       (∏ i∈UNIV.
         A $ index-iso i $ index-iso (p i))) =
    (∑ p∈?P.
      ((λq. of-int (sign q) *
        (∏ j∈UNIV. A $ j $ q j)) ∘ ?F) p)
  proof (rule sum.cong)
   show ?P = ?P
    by simp
   fix p
   assume p ∈ ?P
   then show (of-int (sign p) :: 'r) *
      (∏ i∈UNIV.
        A $ index-iso i $ index-iso (p i)) =
     ((λq. of-int (sign q) *
       (∏ j∈UNIV. A $ j $ q j)) ∘ ?F) p
     by (simp only: o-apply summand-reindexed)
  qed
  also have ... =
    (∑ q∈?Q.
      of-int (sign q) * (∏ j∈UNIV. A $ j $ q j))
    unfolding o-def
    by (rule sum.reindex-bij-betw[OF permutation-betw])
  finally show ?thesis .
 qed
 show ?thesis
  unfolding det-def
  by (simp only: vec-lambda-beta sum-reindexed)
qed
```

The Cartesian-space library provides matrix induction by elementary operations only for real matrices. The proof uses no order or topology, so we expose the same argument over an arbitrary field.

```
lemma induct-matrix-row-operations-field:
 fixes P :: 'a::field^'n^'n ⇒ bool
 assumes zero-row: ⋀A i. row i A = 0 ⟹ P A
  and diagonal: ⋀A. (⋀i j. i ≠ j ⟹ A$i$j = 0) ⟹ P A
  and swap-cols: ⋀A m n. ⟦P A; m ≠ n⟧ ⟹
    P (χ i j. A $ i $ Transposition.transpose m n j)
```

```
    and row-op: ⋀A m n c. ⟦P A; m ≠ n⟧ ⟹
      P (χ i. if i = m then row m A + c *s row n A else row i A)
  shows P A
proof −
  have P A if zeros:
    ⋀i j. ⟦j ∈ −K; i ≠ j⟧ ⟹ A$i$j = 0
    for A K
  proof −
    have finite K
      by simp
    then show ?thesis using zeros
    proof (induction arbitrary: A rule: finite-induct)
      case empty
      with diagonal show ?case
        by simp
    next
      case (insert k K)
      note insertK = insert
      have pivot: P A if nonzero: A$k$k ≠ 0
        and outside:
          ⋀i j. ⟦j ∈ − insert k K; i ≠ j⟧ ⟹
            A$i$j = 0
          ⋀i. ⟦i ∈ −L; i ≠ k⟧ ⟹ A$i$k = 0
        for A L
      proof −
        have finite L
          by simp
        then show ?thesis using outside nonzero
        proof (induction arbitrary: A rule: finite-induct)
          case (empty B)
          show ?case
          proof (rule insertK)
            fix i j
            assume i ∈ − K j ≠ i
            show B $ j $ i = 0
              using ‹j ≠ i› ‹i ∈ − K› empty
              by (metis ComplD ComplI Compl-eq-Diff-UNIV Diff-empty
                UNIV-I insert-iff)
          qed
        next
          case (insert l L B)
          show ?case
          proof (cases k = l)
            case True
            with insert show ?thesis
              by auto
          next
            case False
            let ?C = χ i.
```

```
      if i = l
      then row l B − (B $ l $ k / B $ k $ k) ∗s row k B
      else row i B
    have first-zero:
      ⟦j ∈ − insert k K; i ≠ j⟧ ⟹
        ?C $ i $ j = 0
      for j i
      by (auto simp: insert.prems(1) row-def)
    have column-zero: ?C $ i $ k = 0
      if i ∈ − L i ≠ k for i
    proof (cases i = l)
      case True
      with that insert.prems show ?thesis
        by (simp add: row-def)
    next
      case False
      with that show ?thesis
        by (simp add: insert.prems(2) row-def)
    qed
    have pivot-nonzero: ?C $ k $ k ≠ 0
      by (auto simp: insert.prems row-def ‹k ≠ l›)
    have P ?C
      using insert.IH[OF first-zero column-zero pivot-nonzero]
      by auto
    moreover have (χ i.
        if i = l
        then row l ?C + (B $ l $ k / B $ k $ k) ∗s row k ?C
        else row i ?C) = B
      using ‹k ≠ l›
      by (simp add: vec-eq-iff row-def)
    ultimately show ?thesis
      using row-op[of ?C l k B$l$k / B$k$k] ‹k ≠ l›
      by (simp add: cong: if-cong)
  qed
 qed
qed
then have nonzero-case: P A
 if A$k$k ≠ 0
   ⋀i j. j ∈ − insert k K ∧ i ≠ j ⟹
     A$i$j = 0
 for A
 using that by auto
show ?case
proof (cases row k A = 0)
 case True
 with zero-row show ?thesis
   by auto
next
 case False
```

```
      then obtain l where l: A$k$l ≠ 0
        by (auto simp: row-def zero-vec-def vec-eq-iff)
      show ?thesis
      proof (cases k = l)
        case True
        with l nonzero-case insert.prems show ?thesis
          by blast
      next
        case False
        have swapped-zero:
          A $ i $ Transposition.transpose k l j = 0
          if j ≠ k j ∉ K i ≠ j for i j
          using False l insert.prems that
          by (auto simp add: Transposition.transpose-def)
        have P (χ i j.
            (χ i j. A $ i $ Transposition.transpose k l j) $
              i $ Transposition.transpose k l j)
          by (rule swap-cols[OF nonzero-case False])
            (auto simp: l swapped-zero)
        moreover have (χ i j.
            (χ i j. A $ i $ Transposition.transpose k l j) $
              i $ Transposition.transpose k l j) = A
          by simp
        ultimately show ?thesis
          by simp
      qed
    qed
  qed
  then show ?thesis
    by blast
qed

lemma induct-matrix-elementary-field:
  fixes P :: 'a::field^'n^'n ⇒ bool
  assumes mult: ⋀A B. ⟦P A; P B⟧ ⟹ P (A ** B)
    and zero-row: ⋀A i. row i A = 0 ⟹ P A
    and diagonal: ⋀A. (⋀i j. i ≠ j ⟹ A$i$j = 0) ⟹ P A
    and swap: ⋀m n. m ≠ n ⟹
      P (χ i j. mat 1 $ i $ Transposition.transpose m n j)
    and add: ⋀m n c. m ≠ n ⟹
      P (χ i j.
        if i = m ∧ j = n then c else of-bool (i = j))
  shows P A
proof −
  have swap-columns:
    P (χ i j. A $ i $ Transposition.transpose m n j)
    if P A m ≠ n for A m n
  proof −
```

```
    have A ** (χ i j.
        mat 1 $ i $ Transposition.transpose m n j) =
      (χ i j. A $ i $ Transposition.transpose m n j)
      by (simp add: matrix-matrix-mult-def mat-def vec-eq-iff
        if-distrib sum.delta-remove)
    then show ?thesis
      using mult swap that by metis
  qed
  have add-row:
    P (χ i.
      if i = m then row m A + c *s row n A else row i A)
    if P A m ≠ n for A m n c
  proof −
    let ?B = χ i j.
      if i = m ∧ j = n then c else of-bool (i = j)
    have ?B ** A =
      (χ i. if i = m then row m A + c *s row n A else row i A)
      using ‹m ≠ n›
      unfolding matrix-matrix-mult-def row-def of-bool-def
      by (auto simp: vec-eq-iff
        if-distrib[of λx. x * y for y] sum.remove
        cong: if-cong)
    then show ?thesis
      by (rule subst) (auto simp: that mult add)
  qed
  show ?thesis
    by (rule induct-matrix-row-operations-field[
      OF zero-row diagonal swap-columns add-row])
qed

definition block-embed ::
    'j::finite ⇒ 'r::zero-neq-one^'i::finite^'i ⇒
      'r^('i × 'j)^('i × 'j)
  where block-embed j B =
    (χ ik hl.
      if snd ik = j ∧ snd hl = j
      then B $ fst ik $ fst hl
      else of-bool (ik = hl))

lemma sum-UNIV-prod:
  fixes f :: ('a::finite × 'b::finite) ⇒ 'c::comm-monoid-add
  shows (∑ x∈UNIV. f x) =
    (∑ a∈UNIV. ∑ b∈UNIV. f (a, b))
  by (simp only: UNIV-Times-UNIV[symmetric] sum.cartesian-product')

lemma cartesian-matrix-mult-component:
  fixes B C :: 'r::comm-ring-1^'i::finite^'i
  shows (B ** C) $ i $ h = (∑ a∈UNIV. B $ i $ a * C $ a $ h)
  by (simp only: matrix-matrix-mult-def vec-lambda-beta)
```

lemma *block-embed-component* [*simp*]:
fixes $j :: 'j$::*finite*
and $B :: 'r$::*zero-neq-one*$\hat{}'i$::*finite*$\hat{}'i$
shows *block-embed* j B \$ (i, k) \$ $(h, l) =$
$(if\ k = j \wedge l = j\ then\ B\ \$\ i\ \$\ h$
else of-bool $((i, k) = (h, l)))$
by (*simp add*: *block-embed-def*)

lemma *block-embed-one* [*simp*]:
fixes $j :: 'j$::*finite*
shows *block-embed* j $(mat\ 1 :: 'r$::*comm-ring-1*$\hat{}'i\hat{}'i) =$
$(mat\ 1 :: 'r\hat{}('i \times 'j)\hat{}('i \times 'j))$
by (*vector block-embed-def mat-def of-bool-def*)

lemma *sum-of-bool-left*:
fixes $f :: 'a$::*finite* $\Rightarrow$ $'r$::*comm-ring-1*
shows $(\sum x{\in}UNIV.\ of\text{-}bool\ (x = p) * f\ x) = f\ p$
by (*subst sum.remove*[*of UNIV p*]) (*auto simp*: *of-bool-def*)

lemma *sum-of-bool-right*:
fixes $f :: 'a$::*finite* $\Rightarrow$ $'r$::*comm-ring-1*
shows $(\sum x{\in}UNIV.\ f\ x * of\text{-}bool\ (x = p)) = f\ p$
by (*subst sum.remove*[*of UNIV p*]) (*auto simp*: *of-bool-def*)

lemma *prod-if-eq*:
fixes $j :: 'a$::*finite*
and $x :: 'r$::*comm-monoid-mult*
shows $(\prod k{\in}UNIV.\ if\ k = j\ then\ x\ else\ 1) = x$
by (*subst prod.remove*[*of UNIV j*]) *simp-all*

lemma *block-embed-row-outside*:
fixes $j :: 'j$::*finite*
and $B :: 'r$::*zero-neq-one*$\hat{}'i$::*finite*$\hat{}'i$
assumes $k \neq j$
shows *block-embed* j B \$ (i, k) \$ $(h, l) =$
of-bool $((i, k) = (h, l))$
proof −
have *outside*: $\neg\ (k = j \wedge l = j)$
using *assms* **by** *blast*
show *?thesis*
apply (*subst block-embed-component*)
apply (*subst if-not-P*[*OF outside*])
apply (*rule refl*)
done
qed

lemma *block-embed-column-outside*:
fixes $j :: 'j$::*finite*

```
    and B :: 'r::zero-neq-one^'i::finite^'i
  assumes l ≠ j
  shows block-embed j B $ (i, k) $ (h, l) =
    of-bool ((i, k) = (h, l))
proof −
  have outside: ¬ (k = j ∧ l = j)
    using assms by blast
  show ?thesis
    apply (subst block-embed-component)
    apply (subst if-not-P[OF outside])
    apply (rule refl)
    done
qed

lemma block-embed-mult:
  fixes j :: 'j::finite
    and B C :: 'r::comm-ring-1^'i::finite^'i
  shows block-embed j (B ** C) = block-embed j B ** block-embed j C
proof −
  have entry:
    block-embed j (B ** C) $ ik $ hl =
      (block-embed j B ** block-embed j C) $ ik $ hl
    for ik hl
  proof −
    obtain i k where ik-eq: ik = (i, k)
      by (cases ik) blast
    obtain h l where hl-eq: hl = (h, l)
      by (cases hl) blast
    have equal-case:
      block-embed j (B ** C) $ (i, k) $ (h, l) =
        (block-embed j B ** block-embed j C) $ (i, k) $ (h, l)
      if k-eq: k = j
    proof (cases l = j)
      case True
      have inner:
        (∑ b∈UNIV.
            block-embed j B $ (i, j) $ (a, b) *
            block-embed j C $ (a, b) $ (h, j)) =
          B $ i $ a * C $ a $ h
        for a
      proof −
        have selected:
          block-embed j B $ (i, j) $ (a, j) *
              block-embed j C $ (a, j) $ (h, j) =
            B $ i $ a * C $ a $ h
        proof −
          have first:
            block-embed j B $ (i, j) $ (a, j) = B $ i $ a
            apply (subst block-embed-component)
```

```
    apply (subst if-P)
     apply blast
    apply (rule refl)
    done
  have second:
    block-embed j C $ (a, j) $ (h, j) = C $ a $ h
    apply (subst block-embed-component)
    apply (subst if-P)
     apply blast
    apply (rule refl)
    done
  show ?thesis
    apply (subst first)
    apply (subst second)
    apply (rule refl)
    done
qed
have off:
  block-embed j B $ (i, j) $ (a, b) *
      block-embed j C $ (a, b) $ (h, j) = 0
  if b ∈ UNIV − {j} for b
proof −
  have b-ne: b ≠ j
    using that by blast
  have outside: ¬ (j = j ∧ b = j)
    using b-ne by blast
  have unequal: (i, j) ≠ (a, b)
    using b-ne by blast
  have first-zero:
    block-embed j B $ (i, j) $ (a, b) = 0
    apply (subst block-embed-component)
    apply (subst if-not-P[OF outside])
    apply (simp only: of-bool-def unequal if-False)
    done
  show ?thesis
    apply (subst first-zero)
    apply simp
    done
qed
have off-sum:
  (∑ b∈UNIV − {j}.
      block-embed j B $ (i, j) $ (a, b) *
      block-embed j C $ (a, b) $ (h, j)) = 0
  by (intro sum.neutral ballI; rule off)
have split:
  (∑ b∈UNIV.
      block-embed j B $ (i, j) $ (a, b) *
      block-embed j C $ (a, b) $ (h, j)) =
    block-embed j B $ (i, j) $ (a, j) *
```

```
      block-embed j C $ (a, j) $ (h, j) +
    (∑ b∈UNIV − {j}.
      block-embed j B $ (i, j) $ (a, b) ∗
      block-embed j C $ (a, b) $ (h, j))
  by (subst sum.remove[of UNIV j])
    simp-all
 have reduced:
  block-embed j B $ (i, j) $ (a, j) ∗
      block-embed j C $ (a, j) $ (h, j) +
    (∑ b∈UNIV − {j}.
      block-embed j B $ (i, j) $ (a, b) ∗
      block-embed j C $ (a, b) $ (h, j)) =
    B $ i $ a ∗ C $ a $ h
  apply (subst selected)
  apply (subst off-sum)
  apply simp
  done
 show ?thesis
  by (rule trans[OF split reduced])
qed
have lhs:
 block-embed j (B ∗∗ C) $ (i, k) $ (h, l) =
  (∑ a∈UNIV. B $ i $ a ∗ C $ a $ h)
proof −
 have block:
  block-embed j (B ∗∗ C) $ (i, k) $ (h, l) =
    (B ∗∗ C) $ i $ h
  apply (subst block-embed-component)
  apply (subst if-P)
   using True k-eq apply blast
  apply (rule refl)
  done
 show ?thesis
  by (rule trans[OF block cartesian-matrix-mult-component])
qed
have rhs:
 (block-embed j B ∗∗ block-embed j C) $ (i, k) $ (h, l) =
  (∑ a∈UNIV. B $ i $ a ∗ C $ a $ h)
proof −
 have (block-embed j B ∗∗ block-embed j C) $ (i, k) $ (h, l) =
    (∑ x∈UNIV.
      block-embed j B $ (i, k) $ x ∗
      block-embed j C $ x $ (h, l))
  by (rule cartesian-matrix-mult-component)
 also have ... =
    (∑ a∈UNIV. ∑ b∈UNIV.
      block-embed j B $ (i, j) $ (a, b) ∗
      block-embed j C $ (a, b) $ (h, j))
  apply (subst sum-UNIV-prod)
```

```
      apply (subst k-eq)
      apply (subst True)
      apply (rule refl)
      done
    also have ... =
        (∑ a∈UNIV. B $ i $ a * C $ a $ h)
    proof (rule sum.cong)
      show (UNIV :: 'i set) = UNIV
        by simp
      fix a
      assume a ∈ (UNIV :: 'i set)
      show (∑ b∈UNIV.
          block-embed j B $ (i, j) $ (a, b) *
          block-embed j C $ (a, b) $ (h, j)) =
        B $ i $ a * C $ a $ h
        by (rule inner)
    qed
    finally show ?thesis .
  qed
  show ?thesis
    apply (rule trans[OF lhs])
    apply (rule rhs[symmetric])
    done
next
  case False
  have column:
    block-embed j C $ x $ (h, l) =
      of-bool (x = (h, l))
    for x
  proof −
    obtain a b where x-eq: x = (a, b)
      by (cases x) blast
    have pair:
      block-embed j C $ (a, b) $ (h, l) =
        of-bool ((a, b) = (h, l))
      by (rule block-embed-column-outside[OF False])
    show ?thesis
      using pair by (simp only: x-eq)
  qed
  have rhs:
    (block-embed j B ** block-embed j C) $ (i, k) $ (h, l) =
      block-embed j B $ (i, k) $ (h, l)
  proof −
    have (block-embed j B ** block-embed j C) $ (i, k) $ (h, l) =
        (∑ x∈UNIV.
          block-embed j B $ (i, k) $ x *
          block-embed j C $ x $ (h, l))
      by (rule cartesian-matrix-mult-component)
    also have ... =
```

```
        (∑ x∈UNIV.
          block-embed j B $ (i, k) $ x *
          of-bool (x = (h, l)))
      proof (rule sum.cong)
        show (UNIV :: ('i × 'j) set) = UNIV
          by simp
        fix x
        assume x ∈ (UNIV :: ('i × 'j) set)
        show block-embed j B $ (i, k) $ x *
            block-embed j C $ x $ (h, l) =
          block-embed j B $ (i, k) $ x *
            of-bool (x = (h, l))
          by (subst column) (rule refl)
      qed
      also have ... = block-embed j B $ (i, k) $ (h, l)
        by (rule sum-of-bool-right)
      finally show ?thesis .
    qed
    have lhs-delta:
      block-embed j (B ** C) $ (i, k) $ (h, l) =
        of-bool ((i, k) = (h, l))
      by (rule block-embed-column-outside[OF False])
    have rhs-delta:
      block-embed j B $ (i, k) $ (h, l) =
        of-bool ((i, k) = (h, l))
      by (rule block-embed-column-outside[OF False])
    have outside:
      block-embed j (B ** C) $ (i, k) $ (h, l) =
        block-embed j B $ (i, k) $ (h, l)
      by (rule trans[OF lhs-delta rhs-delta[symmetric]])
    show ?thesis
      by (rule trans[OF outside rhs[symmetric]])
qed
have unequal-case:
  block-embed j (B ** C) $ (i, k) $ (h, l) =
    (block-embed j B ** block-embed j C) $ (i, k) $ (h, l)
  if k-ne: k ≠ j
proof −
  have row:
    block-embed j B $ (i, k) $ x = of-bool (x = (i, k))
    for x
  proof −
    obtain a b where x-eq: x = (a, b)
      by (cases x) blast
    have block-embed j B $ (i, k) $ (a, b) =
        of-bool ((i, k) = (a, b))
      by (rule block-embed-row-outside[OF k-ne])
    also have ... = of-bool ((a, b) = (i, k))
      by (simp only: eq-commute)
```

```
    finally have pair:
      block-embed j B $ (i, k) $ (a, b) =
        of-bool ((a, b) = (i, k)) .
    show ?thesis
      using pair by (simp only: x-eq)
  qed
  have rhs:
    (block-embed j B ** block-embed j C) $ (i, k) $ (h, l) =
      block-embed j C $ (i, k) $ (h, l)
  proof −
    have (block-embed j B ** block-embed j C) $ (i, k) $ (h, l) =
        (∑ x∈UNIV.
          block-embed j B $ (i, k) $ x *
          block-embed j C $ x $ (h, l))
      by (rule cartesian-matrix-mult-component)
    also have ... =
        (∑ x∈UNIV.
          of-bool (x = (i, k)) *
          block-embed j C $ x $ (h, l))
    proof (rule sum.cong)
      show (UNIV :: ('i × 'j) set) = UNIV
        by simp
      fix x
      assume x ∈ (UNIV :: ('i × 'j) set)
      show block-embed j B $ (i, k) $ x *
            block-embed j C $ x $ (h, l) =
          of-bool (x = (i, k)) *
            block-embed j C $ x $ (h, l)
        by (subst row) (rule refl)
    qed
    also have ... = block-embed j C $ (i, k) $ (h, l)
      by (rule sum-of-bool-left)
    finally show ?thesis .
  qed
  have lhs-delta:
    block-embed j (B ** C) $ (i, k) $ (h, l) =
      of-bool ((i, k) = (h, l))
    by (rule block-embed-row-outside[OF k-ne])
  have rhs-delta:
    block-embed j C $ (i, k) $ (h, l) =
      of-bool ((i, k) = (h, l))
    by (rule block-embed-row-outside[OF k-ne])
  have outside:
    block-embed j (B ** C) $ (i, k) $ (h, l) =
      block-embed j C $ (i, k) $ (h, l)
    by (rule trans[OF lhs-delta rhs-delta[symmetric]])
  show ?thesis
    by (rule trans[OF outside rhs[symmetric]])
qed
```

```
    have pair-entry:
      block-embed j (B ** C) $ (i, k) $ (h, l) =
      (block-embed j B ** block-embed j C) $ (i, k) $ (h, l)
    proof (cases k = j)
      case True
      then show ?thesis by (rule equal-case)
    next
      case False
      then show ?thesis by (rule unequal-case)
    qed
    show ?thesis
      using pair-entry by (simp only: ik-eq hl-eq)
  qed
  show ?thesis
    apply (subst vec-eq-iff)
    apply (rule allI)
    apply (subst vec-eq-iff)
    apply (rule allI)
    apply (rule entry)
    done
qed

theorem det-block-embed:
  fixes j :: 'j::finite
    and B :: 'r::field^'i::finite^'i
  shows det (block-embed j B) = det B
proof (rule induct-matrix-elementary-field)
  fix C D :: 'r^'i^'i
  assume C: det (block-embed j C) = det C
    and D: det (block-embed j D) = det D
  show det (block-embed j (C ** D)) = det (C ** D)
    by (simp only: block-embed-mult det-mul C D)
next
  fix C :: 'r^'i^'i and i
  assume zero: row i C = 0
  have embedded-zero: row (i, j) (block-embed j C) = 0
    using zero
    by (auto simp: row-def vec-eq-iff block-embed-def of-bool-def
      split: prod.splits)
  have det (block-embed j C) = 0
    by (rule det-zero-row(2)[OF embedded-zero])
  moreover have det C = 0
    by (rule det-zero-row(2)[OF zero])
  ultimately show det (block-embed j C) = det C
    by simp
next
  fix C :: 'r^'i^'i
  assume diagonal: ⋀i h. i ≠ h ⟹ C $ i $ h = 0
  have embedded-diagonal:
```

```
    block-embed j C $ ik $ hl = 0 if ik ≠ hl for ik hl
  proof −
    obtain i k where ik: ik = (i, k)
      by (cases ik) blast
    obtain h l where hl: hl = (h, l)
      by (cases hl) blast
    show ?thesis
      using that diagonal
      by (auto simp: ik hl block-embed-def of-bool-def)
  qed
  have pair-product:
    (∏ ik∈(UNIV :: ('i × 'j) set).
        block-embed j C $ ik $ ik) =
      (∏ i∈UNIV. C $ i $ i)
  proof −
    have diagonal-entry:
      block-embed j C $ (i, k) $ (i, k) =
        (if k = j then C $ i $ i else 1) for i k
      by (cases k = j)
        (simp-all add: of-bool-def)
    have (∏ ik∈(UNIV :: ('i × 'j) set).
        block-embed j C $ ik $ ik) =
      (∏ i∈UNIV. ∏ k∈UNIV.
        block-embed j C $ (i, k) $ (i, k))
      by (simp only: UNIV-Times-UNIV[symmetric]
        prod.cartesian-product')
    also have ... =
      (∏ i∈UNIV. ∏ k∈UNIV.
        if k = j then C $ i $ i else 1)
      by (intro prod.cong refl; rule diagonal-entry)
    also have ... = (∏ i∈UNIV. C $ i $ i)
      by (simp only: prod-if-eq)
    finally show ?thesis .
  qed
  have det (block-embed j C) =
      (∏ ik∈UNIV. block-embed j C $ ik $ ik)
    by (rule det-diagonal) (rule embedded-diagonal)
  also have ... = (∏ i∈UNIV. C $ i $ i)
    by (rule pair-product)
  also have ... = det C
    by (rule det-diagonal[OF diagonal, symmetric])
  finally show det (block-embed j C) = det C .
next
  fix m n :: 'i
  assume mn: m ≠ n
  let ?C = χ i h.
    mat 1 $ i $ Transposition.transpose m n h :: 'r^'i^'i
  let ?D = χ ik hl.
    mat 1 $ ik $ Transposition.transpose (m, j) (n, j) hl ::
```

```
    'r^('i × 'j)^('i × 'j)
  have embedded: block-embed j ?C = ?D
    using mn
    by (auto simp: vec-eq-iff block-embed-def mat-def of-bool-def
      Transposition.transpose-def split: prod.splits)
  have permutes-C:
    Transposition.transpose m n permutes (UNIV :: 'i set)
    by (rule permutes-swap-id) simp-all
  have sign-C: sign (Transposition.transpose m n) = -1
    by (simp add: sign-swap-id mn)
  have det-C: det ?C = -1
    using det-permute-columns[OF permutes-C,
      of mat 1 :: 'r^'i^'i]
    by (simp add: sign-C)
  have pair-ne: (m, j) ≠ (n, j)
    using mn by simp
  have permutes-D:
    Transposition.transpose (m, j) (n, j) permutes
      (UNIV :: ('i × 'j) set)
    by (rule permutes-swap-id) simp-all
  have sign-D:
    sign (Transposition.transpose (m, j) (n, j)) = -1
  proof -
    have sign (Transposition.transpose (m, j) (n, j)) =
      (if (m, j) = (n, j) then 1 else -1)
      by (rule sign-swap-id)
    then show ?thesis
      using pair-ne by simp
  qed
  have det-D: det ?D = -1
    using det-permute-columns[OF permutes-D,
      of mat 1 :: 'r^('i × 'j)^('i × 'j)]
    by (simp add: sign-D)
  show det (block-embed j ?C) = det ?C
    by (simp only: embedded det-C det-D)
next
  fix m n :: 'i and c :: 'r
  assume mn: m ≠ n
  let ?C = χ i h.
    if i = m ∧ h = n then c else of-bool (i = h) :: 'r^'i^'i
  let ?D = χ ik hl.
    if ik = (m, j) ∧ hl = (n, j)
    then c else of-bool (ik = hl) ::
      'r^('i × 'j)^('i × 'j)
  have embedded: block-embed j ?C = ?D
    using mn
    by (auto simp: vec-eq-iff block-embed-def of-bool-def
      split: prod.splits)
  have C-row:
```

```
    ?C = (χ i.
      if i = m
      then row m (mat 1) + c *s row n (mat 1)
      else row i (mat 1))
    using mn
    by (auto simp: vec-eq-iff row-def mat-def of-bool-def)
  have D-row:
    ?D = (χ ik.
      if ik = (m, j)
      then row (m, j) (mat 1) + c *s row (n, j) (mat 1)
      else row ik (mat 1))
    using mn
    by (auto simp: vec-eq-iff row-def mat-def of-bool-def
      split: prod.splits)
  have det-C: det ?C = 1
    unfolding C-row
    by (subst det-row-operation[OF mn]) simp
  have det-D: det ?D = 1
    unfolding D-row
    by (subst det-row-operation)
      (simp-all add: mn)
  show det (block-embed j ?C) = det ?C
    by (simp only: embedded det-C det-D)
qed

definition block-swap-entries ::
    'i::finite set ⇒ 'j::finite ⇒ 'j ⇒
      'r::comm-ring-1^('i × 'j)^('i × 'j)
  where block-swap-entries X m n =
    (χ ik hl. mat 1 $ ik $
      (if fst hl ∈ X
       then (fst hl, Transposition.transpose m n (snd hl))
       else hl))

definition block-swap ::
    'j::finite ⇒ 'j ⇒
      'r::comm-ring-1^('i::finite × 'j)^('i × 'j)
  where block-swap m n = block-swap-entries UNIV m n

lemma block-swap-entries-empty [simp]:
  block-swap-entries {} m n =
    (mat 1 :: 'r::comm-ring-1^('i::finite × 'j::finite)^
      ('i × 'j))
  by (vector block-swap-entries-def)

lemma block-swap-entries-insert:
  fixes X :: 'i::finite set
    and m n :: 'j::finite
  assumes a ∉ X
```

```
  shows block-swap-entries (insert a X) m n =
    (χ ik hl. block-swap-entries X m n $ ik $
      Transposition.transpose (a, m) (a, n) hl)
  using assms
  by (auto simp: vec-eq-iff block-swap-entries-def mat-def
    Transposition.transpose-def split: prod.splits)

lemma det-block-swap-entries:
  fixes X :: 'i::finite set
    and m n :: 'j::finite
  assumes m ≠ n
  shows det (block-swap-entries X m n ::
      'r::field^('i × 'j)^('i × 'j)) =
    (-1) ^ card X
proof -
  have finite X
    by simp
  then show ?thesis
  proof (induction X rule: finite-induct)
    case empty
    show ?case
      by simp
  next
    case (insert a X)
    have pair-ne: (a, m) ≠ (a, n)
      using assms by simp
    have permutes:
      Transposition.transpose (a, m) (a, n) permutes
        (UNIV :: ('i × 'j) set)
      by (rule permutes-swap-id) simp-all
    have sign:
      sign (Transposition.transpose (a, m) (a, n)) = -1
    proof -
      have sign (Transposition.transpose (a, m) (a, n)) =
        (if (a, m) = (a, n) then 1 else -1)
        by (rule sign-swap-id)
      then show ?thesis
        using pair-ne by simp
    qed
    have det (block-swap-entries (insert a X) m n ::
        'r^('i × 'j)^('i × 'j)) =
      det (χ ik hl. block-swap-entries X m n $ ik $
        Transposition.transpose (a, m) (a, n) hl)
      by (simp only: block-swap-entries-insert[OF insert.hyps(2)])
    also have ... =
      of-int (sign (Transposition.transpose (a, m) (a, n))) *
        det (block-swap-entries X m n)
      by (rule det-permute-columns[OF permutes])
    also have ... = (-1) ^ card (insert a X)
```

```
    by (simp add: sign insert.hyps insert.IH)
   finally show ?case .
 qed
qed

theorem det-block-swap:
 fixes m n :: 'j::finite
 assumes m ≠ n
 shows det (block-swap m n ::
    'r::field^('i::finite × 'j)^('i × 'j)) =
  (-1) ^ CARD('i)
 unfolding block-swap-def
 by (simp add: det-block-swap-entries[OF assms])

definition block-shear-entries ::
   ('i::finite × 'i) set ⇒ 'j::finite ⇒ 'j ⇒
    'r::comm-ring-1^'i^'i ⇒
    'r^('i × 'j)^('i × 'j)
 where block-shear-entries X m n B =
   (χ ik hl.
     if snd ik = m ∧ snd hl = n ∧ (fst ik, fst hl) ∈ X
     then B $ fst ik $ fst hl
     else of-bool (ik = hl))

definition block-shear ::
   'j::finite ⇒ 'j ⇒ 'r::comm-ring-1^'i::finite^'i ⇒
    'r^('i × 'j)^('i × 'j)
 where block-shear m n B = block-shear-entries UNIV m n B

lemma block-shear-entries-empty [simp]:
 block-shear-entries {} m n B =
   (mat 1 :: 'r::comm-ring-1^('i::finite × 'j::finite)^
     ('i × 'j))
 by (vector block-shear-entries-def mat-def of-bool-def)

lemma block-shear-entries-insert:
 fixes X :: ('i::finite × 'i) set
   and m n :: 'j::finite
   and B :: 'r::comm-ring-1^'i^'i
 assumes pair-out: (i, h) ∉ X
   and mn: m ≠ n
 shows block-shear-entries (insert (i, h) X) m n B =
   (χ ik.
     if ik = (i, m)
     then row (i, m) (block-shear-entries X m n B) +
       (B $ i $ h) *s row (h, n) (block-shear-entries X m n B)
     else row ik (block-shear-entries X m n B))
 using pair-out mn
 by (auto simp: vec-eq-iff row-def block-shear-entries-def
```

```
    of-bool-def split: prod.splits)

lemma det-block-shear-entries:
  fixes X :: ('i::finite × 'i) set
    and m n :: 'j::finite
    and B :: 'r::field^'i^'i
  assumes m ≠ n
  shows det (block-shear-entries X m n B) = 1
proof −
  have finite X
    by simp
  then show ?thesis
  proof (induction X rule: finite-induct)
    case empty
    show ?case
      by simp
  next
    case (insert ih X)
    obtain i h where ih: ih = (i, h)
      by (cases ih) blast
    have pair-out: (i, h) ∉ X
      using insert.hyps(2) ih by simp
    have row-form:
      block-shear-entries (insert ih X) m n B =
        (χ ik.
          if ik = (i, m)
          then row (i, m) (block-shear-entries X m n B) +
            (B $ i $ h) *s row (h, n)
              (block-shear-entries X m n B)
          else row ik (block-shear-entries X m n B))
      unfolding ih
      by (rule block-shear-entries-insert[OF pair-out assms])
    have pair-ne: (i, m) ≠ (h, n)
      using assms by auto
    have det (block-shear-entries (insert ih X) m n B) =
      det (block-shear-entries X m n B)
      unfolding row-form
      by (rule det-row-operation[OF pair-ne])
    also have ... = 1
      by (rule insert.IH)
    finally show ?case .
  qed
qed

theorem det-block-shear:
  fixes m n :: 'j::finite
    and B :: 'r::field^'i::finite^'i
  assumes m ≠ n
  shows det (block-shear m n B) = 1
```

unfolding *block-shear-def*
by (*rule det-block-shear-entries*[*OF assms*])

context *indexed-algebra*
begin

lemma *regular-matrix-sum*:
regular-matrix ($\sum x{\in}X.\ f\ x$) =
($\sum x{\in}X.$ *regular-matrix* ($f\ x$))
by (*induction X rule*: *infinite-finite-induct*)
(*simp-all add*: *regular-matrix-add*)

definition *algebra-norm* :: $'a \Rightarrow 'r$
where *algebra-norm a* = *det* (*regular-matrix a*)

lemma *algebra-norm-zero* [*simp*]:
algebra-norm 0 = *0*
proof −
have *zero-matrix*: (*0* :: $'r\hat{}'i\hat{}'i$) = *mat 0*
by (*vector mat-def*)
have *regular-matrix 0* = (*mat 0* :: $'r\hat{}'i\hat{}'i$)
by (*simp only*: *regular-matrix-zero zero-matrix*)
then show *?thesis*
by (*simp only*: *algebra-norm-def det-0*)
qed

lemma *algebra-norm-one* [*simp*]:
algebra-norm 1 = *1*
by (*simp add*: *algebra-norm-def*)

theorem *algebra-norm-mult*:
algebra-norm (*a* ∗ *b*) = *algebra-norm a* ∗ *algebra-norm b*
by (*simp add*: *algebra-norm-def regular-matrix-mult det-mul*)

lemma *algebra-norm-scalar*:
algebra-norm (*scale r 1*) = $r\ \hat{}\ CARD('i)$
proof −
have *det* (*mat r* :: $'r\hat{}'i\hat{}'i$) =
($\prod i{\in}UNIV.$ (*mat r* :: $'r\hat{}'i\hat{}'i$) $ *i* $ *i*)
by (*rule det-diagonal*) (*simp add*: *mat-def*)
also have ... = $r\ \hat{}\ CARD('i)$
by (*simp add*: *mat-def*)
finally show *?thesis*
by (*simp add*: *algebra-norm-def regular-matrix-scalar-embed*)
qed

end

context *indexed-algebra-reindex*

begin

theorem *algebra-norm-reindexed*:
reindexed.algebra-norm a = original.algebra-norm a
proof −
have *matrix-reindexed*:
reindexed.regular-matrix a =
(χ i k. original.regular-matrix a $
index-iso i $ index-iso k)
by (*vector regular-matrix-reindexed-component*)
show *?thesis*
unfolding *reindexed.algebra-norm-def original.algebra-norm-def*
unfolding *matrix-reindexed*
by (*rule det-reindex-bij*[*OF index-iso-bij*])
qed

end

context *indexed-basis-pair*
begin

theorem *det-matrix-change-basis*:
assumes *module-hom scale scale f*
shows *det* (*second.matrix-of f*) = *det* (*first.matrix-of f*)
proof −
have *inverse-determinants*:
det to-first ∗ det to-second = 1
proof −
have *det* (*to-first ∗∗ to-second*) =
det (*mat 1* :: *'r^'i^'i*)
by (*simp only*: *to-first-to-second*)
then show *?thesis*
by (*simp add*: *det-mul*)
qed
have *det* (*second.matrix-of f*) =
det ((*to-second ∗∗ first.matrix-of f*) ∗∗ *to-first*)
by (*simp only*: *matrix-change-basis*[*OF assms*])
also have ... =
det to-second ∗ det (*first.matrix-of f*) ∗ *det to-first*
by (*simp add*: *det-mul*)
also have ... =
det (*first.matrix-of f*) ∗ (*det to-first ∗ det to-second*)
by (*simp only*: *mult.assoc mult.left-commute mult.commute*)
also have ... = *det* (*first.matrix-of f*)
by (*simp add*: *inverse-determinants*)
finally show *?thesis* **.**
qed

end

context *indexed-algebra*
begin

theorem *algebra-norm-basis-independent*:
fixes *other-basis* :: *'i* ⇒ *'a*
assumes *other-inj*: *inj other-basis*
and *other-independent*: *independent* (*range other-basis*)
and *other-span*: *span* (*range other-basis*) = *UNIV*
shows *algebra-norm a* =
indexed-algebra.algebra-norm scale other-basis a
proof −
interpret *other*: *indexed-algebra scale other-basis*
by *unfold-locales*
(*fact other-inj other-independent other-span*)+
interpret *change*: *indexed-basis-pair scale basis other-basis*
by *unfold-locales*
(*fact basis-inj basis-independent basis-span*
other-inj other-independent other-span)+
have *det* (*other.regular-matrix a*) = *det* (*regular-matrix a*)
unfolding *other.regular-matrix-def regular-matrix-def*
by (*rule change.det-matrix-change-basis*)
(*rule left-multiplication-module-hom*)
then show *?thesis*
by (*simp add*: *algebra-norm-def other.algebra-norm-def*)
qed

theorem *algebra-norm-basis-independent-bij*:
fixes *other-basis* :: *'j*::*finite* ⇒ *'a*
and *index-iso* :: *'i* ⇒ *'j*
assumes *other-inj*: *inj other-basis*
and *other-independent*: *independent* (*range other-basis*)
and *other-span*: *span* (*range other-basis*) = *UNIV*
and *index-iso-bij*: *bij index-iso*
shows *algebra-norm a* =
indexed-algebra.algebra-norm scale other-basis a
proof −
interpret *other*: *indexed-algebra scale other-basis*
by *unfold-locales*
(*fact other-inj other-independent other-span*)+
interpret *reindex*:
indexed-algebra-reindex scale other-basis index-iso
by *unfold-locales*
(*fact other-inj other-independent other-span index-iso-bij*)+
have *algebra-norm a* = *reindex.reindexed.algebra-norm a*
by (*rule algebra-norm-basis-independent*)
(*fact reindex.reindexed.basis-inj*
reindex.reindexed.basis-independent
reindex.reindexed.basis-span)+

```
  also have ... = other.algebra-norm a
    by (rule reindex.algebra-norm-reindexed)
  finally show ?thesis .
qed

end

context indexed-algebra-isomorphism
begin

theorem algebra-norm-iso:
  target.algebra-norm (iso a) = source.algebra-norm a
  by (simp add: target.algebra-norm-def source.algebra-norm-def
    regular-matrix-iso)

end

context indexed-algebra-isomorphism-general
begin

theorem algebra-norm-iso:
  target.algebra-norm (iso a) = source.algebra-norm a
proof −
  have basis-change:
    transported.algebra-norm (iso a) =
      target.algebra-norm (iso a)
    by (rule transported.algebra-norm-basis-independent-bij)
      (fact target.basis-inj target.basis-independent
        target.basis-span index-iso-bij)+
  have transported:
    transported.algebra-norm (iso a) =
      source.algebra-norm a
    by (rule basis-iso.algebra-norm-iso)
  show ?thesis
    using basis-change transported by simp
qed

end

locale indexed-field-algebra-tower =
  indexed-algebra-tower rS sA rA lower-basis upper-basis
  for rS :: 'r::field ⇒ 's::field ⇒ 's
    and sA :: 's ⇒ 'a::comm-ring-1 ⇒ 'a
    and rA :: 'r ⇒ 'a ⇒ 'a
    and lower-basis :: 'i::finite ⇒ 's
    and upper-basis :: 'j::finite ⇒ 'a
begin

definition restricted-matrix ::
```

```
    's^'j^'j ⇒ 'r^('i × 'j)^('i × 'j)
  where restricted-matrix M =
    (χ ik hl.
      lower-algebra.regular-matrix (M $ snd ik $ snd hl) $
        fst ik $ fst hl)

lemma restricted-matrix-component [simp]:
  restricted-matrix M $ (i, j) $ (h, l) =
    lower-algebra.regular-matrix (M $ j $ l) $ i $ h
  by (simp add: restricted-matrix-def)

lemma restricted-matrix-zero [simp]:
  restricted-matrix 0 = 0
  by (auto simp: vec-eq-iff restricted-matrix-def split: prod.splits)

lemma restricted-matrix-one [simp]:
  restricted-matrix (mat 1) = mat 1
  by (auto simp: vec-eq-iff restricted-matrix-def mat-def of-bool-def
    split: prod.splits)

lemma restricted-matrix-mult:
  restricted-matrix (B ** C) =
    restricted-matrix B ** restricted-matrix C
proof −
  have entry:
    restricted-matrix (B ** C) $ (i, j) $ (h, l) =
      (restricted-matrix B ** restricted-matrix C) $ (i, j) $ (h, l)
    for i j h l
  proof −
    have lhs:
      restricted-matrix (B ** C) $ (i, j) $ (h, l) =
        (∑ k∈UNIV. ∑ a∈UNIV.
          lower-algebra.regular-matrix (B $ j $ k) $ i $ a *
          lower-algebra.regular-matrix (C $ k $ l) $ a $ h)
    proof −
      have restricted-matrix (B ** C) $ (i, j) $ (h, l) =
        lower-algebra.regular-matrix ((B ** C) $ j $ l) $ i $ h
        by (rule restricted-matrix-component)
      also have ... =
        lower-algebra.regular-matrix
          (∑ k∈UNIV. B $ j $ k * C $ k $ l) $ i $ h
        by (simp only: matrix-matrix-mult-def vec-lambda-beta)
      also have ... =
        (∑ k∈UNIV.
          lower-algebra.regular-matrix (B $ j $ k * C $ k $ l)) $ i $ h
        by (simp only: lower-algebra.regular-matrix-sum)
      also have ... =
        (∑ k∈UNIV.
          lower-algebra.regular-matrix (B $ j $ k * C $ k $ l) $
```

```
      i $ h)
    by simp
  also have ... =
    (∑k∈UNIV.
      (lower-algebra.regular-matrix (B $ j $ k) **
        lower-algebra.regular-matrix (C $ k $ l)) $ i $ h)
  proof (rule sum.cong)
    show (UNIV :: 'j set) = UNIV
      by simp
    fix k
    assume k ∈ (UNIV :: 'j set)
    show lower-algebra.regular-matrix
          (B $ j $ k * C $ k $ l) $ i $ h =
        (lower-algebra.regular-matrix (B $ j $ k) **
          lower-algebra.regular-matrix (C $ k $ l)) $ i $ h
      by (simp only: lower-algebra.regular-matrix-mult)
  qed
  also have ... =
    (∑k∈UNIV. ∑a∈UNIV.
      lower-algebra.regular-matrix (B $ j $ k) $ i $ a *
      lower-algebra.regular-matrix (C $ k $ l) $ a $ h)
    by (simp only: matrix-matrix-mult-def vec-lambda-beta)
  finally show ?thesis .
qed
have rhs:
  (restricted-matrix B ** restricted-matrix C) $ (i, j) $ (h, l) =
    (∑a∈UNIV. ∑k∈UNIV.
      lower-algebra.regular-matrix (B $ j $ k) $ i $ a *
      lower-algebra.regular-matrix (C $ k $ l) $ a $ h)
proof −
  have (restricted-matrix B ** restricted-matrix C) $
      (i, j) $ (h, l) =
    (∑x∈UNIV.
      restricted-matrix B $ (i, j) $ x *
      restricted-matrix C $ x $ (h, l))
    by (simp add: matrix-matrix-mult-def)
  also have ... =
    (∑a∈UNIV. ∑k∈UNIV.
      restricted-matrix B $ (i, j) $ (a, k) *
      restricted-matrix C $ (a, k) $ (h, l))
    by (simp only: sum-UNIV-prod)
  also have ... =
    (∑a∈UNIV. ∑k∈UNIV.
      lower-algebra.regular-matrix (B $ j $ k) $ i $ a *
      lower-algebra.regular-matrix (C $ k $ l) $ a $ h)
    by (simp only: restricted-matrix-component)
  finally show ?thesis .
qed
have swapped:
```

```
      (∑ k∈UNIV. ∑ a∈UNIV.
          lower-algebra.regular-matrix (B $ j $ k) $ i $ a *
          lower-algebra.regular-matrix (C $ k $ l) $ a $ h) =
        (∑ a∈UNIV. ∑ k∈UNIV.
          lower-algebra.regular-matrix (B $ j $ k) $ i $ a *
          lower-algebra.regular-matrix (C $ k $ l) $ a $ h)
      by (rule sum.swap)
    show ?thesis
      by (rule trans[OF lhs trans[OF swapped rhs[symmetric]]])
  qed
  show ?thesis
  proof (subst vec-eq-iff, intro allI)
    fix ik :: 'i × 'j
    obtain i j where ik: ik = (i, j)
      by (cases ik) blast
    show restricted-matrix (B ** C) $ ik =
      (restricted-matrix B ** restricted-matrix C) $ ik
    proof (subst vec-eq-iff, intro allI)
      fix hl :: 'i × 'j
      obtain h l where hl: hl = (h, l)
        by (cases hl) blast
      show restricted-matrix (B ** C) $ ik $ hl =
        (restricted-matrix B ** restricted-matrix C) $ ik $ hl
        unfolding ik hl
        by (rule entry)
    qed
  qed
qed

definition diagonal-factor :: 'j ⇒ 's ⇒ 's^'j^'j
  where diagonal-factor j x =
    (χ k l. if k = j ∧ l = j then x else of-bool (k = l))

definition partial-diagonal :: 'j set ⇒ 's^'j^'j ⇒ 's^'j^'j
  where partial-diagonal X M =
    (χ k l.
      if k = l
      then if k ∈ X then M $ k $ k else 1
      else 0)

lemma det-diagonal-factor:
  det (diagonal-factor j x) = x
proof −
  have diagonal:
    diagonal-factor j x $ k $ l = 0 if k ≠ l for k l
    using that
    by (auto simp: diagonal-factor-def of-bool-def)
  have entry:
    diagonal-factor j x $ k $ k =
```

```
      (if k = j then x else 1) for k
    by (cases k = j)
      (simp-all add: diagonal-factor-def of-bool-def)
  have det (diagonal-factor j x) =
    (∏ k∈UNIV. diagonal-factor j x $ k $ k)
    by (rule det-diagonal[OF diagonal])
  also have ... =
    (∏ k∈UNIV. if k = j then x else 1)
    by (intro prod.cong refl; rule entry)
  also have ... = x
    by (rule prod-if-eq)
  finally show ?thesis .
qed

lemma diagonal-factor-component-alt:
  diagonal-factor j x $ k $ l =
    of-bool (k = l) * (if k = j then x else 1)
  by (auto simp: diagonal-factor-def of-bool-def)

lemma restricted-diagonal-factor:
  restricted-matrix (diagonal-factor j x) =
    block-embed j (lower-algebra.regular-matrix x)
  by (auto simp: vec-eq-iff restricted-matrix-def diagonal-factor-def
    block-embed-def of-bool-def mat-def split: prod.splits)

lemma partial-diagonal-empty [simp]:
  partial-diagonal {} M = mat 1
  by (vector partial-diagonal-def mat-def)

lemma partial-diagonal-UNIV:
  assumes diagonal: ⋀k l. k ≠ l ⟹ M $ k $ l = 0
  shows partial-diagonal UNIV M = M
  using diagonal
  by (auto simp: vec-eq-iff partial-diagonal-def)

lemma partial-diagonal-insert:
  assumes j ∉ X
  shows partial-diagonal (insert j X) M =
    diagonal-factor j (M $ j $ j) ** partial-diagonal X M
proof −
  have mult-entry:
    (diagonal-factor j (M $ j $ j) ** partial-diagonal X M) $
        k $ l =
      (if k = j then M $ j $ j else 1) *
        partial-diagonal X M $ k $ l for k l
  proof −
    have (diagonal-factor j (M $ j $ j) ** partial-diagonal X M) $
        k $ l =
      (∑ a∈UNIV.
```

```
      diagonal-factor j (M $ j $ j) $ k $ a *
      partial-diagonal X M $ a $ l)
    by (simp only: matrix-matrix-mult-def vec-lambda-beta)
  also have ... =
    diagonal-factor j (M $ j $ j) $ k $ k *
      partial-diagonal X M $ k $ l
    by (subst sum.remove[of UNIV k])
      (auto simp: diagonal-factor-component-alt of-bool-def)
  also have ... =
    (if k = j then M $ j $ j else 1) *
      partial-diagonal X M $ k $ l
    by (simp add: diagonal-factor-component-alt of-bool-def)
  finally show ?thesis .
 qed
 show ?thesis
 proof (subst vec-eq-iff, intro allI)
  fix k
  show partial-diagonal (insert j X) M $ k =
    (diagonal-factor j (M $ j $ j) ** partial-diagonal X M) $ k
  proof (subst vec-eq-iff, intro allI)
   fix l
   show partial-diagonal (insert j X) M $ k $ l =
     (diagonal-factor j (M $ j $ j) ** partial-diagonal X M) $ k $ l
     using assms
     by (subst mult-entry)
       (auto simp: partial-diagonal-def)
  qed
 qed
qed

lemma restricted-swap:
 assumes m ≠ n
 shows restricted-matrix
     (χ j l. mat 1 $ j $ Transposition.transpose m n l) =
   block-swap m n
 using assms
 by (auto simp: vec-eq-iff restricted-matrix-def block-swap-def
   block-swap-entries-def mat-def of-bool-def
   Transposition.transpose-def split: prod.splits)

lemma restricted-shear:
 assumes m ≠ n
 shows restricted-matrix
     (χ j l. if j = m ∧ l = n then c else of-bool (j = l)) =
   block-shear m n (lower-algebra.regular-matrix c)
 using assms
 by (auto simp: vec-eq-iff restricted-matrix-def block-shear-def
   block-shear-entries-def of-bool-def mat-def split: prod.splits)
```

```
theorem det-restrict-scalars:
  det (restricted-matrix M) =
    lower-algebra.algebra-norm (det M)
proof −
  let ?P = λM. det (restricted-matrix M) =
    lower-algebra.algebra-norm (det M)
  have mult-P: ?P (B ** C) if ?P B ?P C for B C
    using that
    by (simp only: restricted-matrix-mult det-mul
      lower-algebra.algebra-norm-mult)
  have one-P: ?P (mat 1)
    by simp
  show ?P M
  proof (rule induct-matrix-elementary-field[where P = ?P])
    fix B C :: 's^'j^'j
    assume ?P B ?P C
    then show ?P (B ** C)
      by (rule mult-P)
  next
    fix B :: 's^'j^'j and j
    assume zero: row j B = 0
    have zero-entry: B $ j $ l = 0 for l
      using zero
      by (simp add: row-def vec-eq-iff)
    have restricted-zero:
      row (undefined, j) (restricted-matrix B) = 0
      using zero-entry
      by (auto simp: row-def vec-eq-iff restricted-matrix-def split: prod.splits)
    have det-restricted: det (restricted-matrix B) = 0
      by (rule det-zero-row(2)[OF restricted-zero])
    have det-B: det B = 0
      by (rule det-zero-row(2)[OF zero])
    show ?P B
      by (simp only: det-restricted det-B
        lower-algebra.algebra-norm-zero)
  next
    fix B :: 's^'j^'j
    assume diagonal: ⋀j l. j ≠ l ⟹ B $ j $ l = 0
    have factor-P:
      ?P (diagonal-factor j (B $ j $ j)) for j
      unfolding lower-algebra.algebra-norm-def
      by (simp only: restricted-diagonal-factor det-block-embed
        det-diagonal-factor)
    have partial-P: ?P (partial-diagonal X B) for X
    proof −
      have finite X
        by simp
      then show ?thesis
      proof (induction X rule: finite-induct)
```

```
      case empty
      show ?case
        by (simp only: partial-diagonal-empty one-P)
    next
      case (insert j X)
      have ?P
        (diagonal-factor j (B $ j $ j) ** partial-diagonal X B)
        by (rule mult-P[OF factor-P insert.IH])
      then show ?case
        by (simp only: partial-diagonal-insert[OF insert.hyps(2)])
    qed
  qed
  have partial-eq:
    partial-diagonal (UNIV :: 'j set) B = B
    by (rule partial-diagonal-UNIV[OF diagonal])
  have partial-result:
    ?P (partial-diagonal (UNIV :: 'j set) B)
    by (rule partial-P)
  from partial-result show ?P B
    unfolding partial-eq .
next
  fix m n :: 'j
  assume mn: m ≠ n
  let ?B = χ j l.
    mat 1 $ j $ Transposition.transpose m n l :: 's^'j^'j
  have permutes:
    Transposition.transpose m n permutes (UNIV :: 'j set)
    by (rule permutes-swap-id) simp-all
  have sign: sign (Transposition.transpose m n) = -1
    by (simp add: sign-swap-id mn)
  have det-B: det ?B = -1
    using det-permute-columns[OF permutes,
      of mat 1 :: 's^'j^'j]
    by (simp add: sign)
  have norm-neg-one:
    lower-algebra.algebra-norm (-1) =
      (-1 :: 'r) ^ CARD('i)
  proof -
    have lower-algebra.algebra-norm (rS (-1) 1) =
      (-1 :: 'r) ^ CARD('i)
      by (rule lower-algebra.algebra-norm-scalar)
    then show ?thesis
      by simp
  qed
  have det-restricted:
    det (restricted-matrix ?B) = (-1 :: 'r) ^ CARD('i)
    by (simp only: restricted-swap[OF mn] det-block-swap[OF mn])
  show ?P ?B
    by (simp only: det-restricted det-B norm-neg-one)
```

```
  next
    fix m n :: 'j and c :: 's
    assume mn: m ≠ n
    let ?B = χ j l.
      if j = m ∧ l = n then c else of-bool (j = l) :: 's^'j^'j
    have B-row:
      ?B = (χ j.
        if j = m
        then row m (mat 1) + c *s row n (mat 1)
        else row j (mat 1))
      using mn
      by (auto simp: vec-eq-iff row-def mat-def of-bool-def)
    have det-B: det ?B = 1
      unfolding B-row
      by (subst det-row-operation[OF mn]) simp
    have det-restricted: det (restricted-matrix ?B) = 1
      by (simp only: restricted-shear[OF mn]
        det-block-shear[OF mn])
    show ?P ?B
      by (simp only: det-restricted det-B
        lower-algebra.algebra-norm-one)
  qed
qed

theorem matrix-restrict-scalars:
  assumes f: module-hom sA sA f
  shows composed.matrix-of f =
    restricted-matrix (upper.matrix-of f)
proof −
  interpret f: module-hom sA sA f
    by fact
  have entry:
    composed.matrix-of f $ (i, j) $ (h, l) =
      restricted-matrix (upper.matrix-of f) $ (i, j) $ (h, l)
    for i j h l
  proof −
    have composed.matrix-of f $ (i, j) $ (h, l) =
      composed.coordinate (f (tower-basis (h, l))) (i, j)
      by simp
    also have ... =
      tower-coordinate (f (tower-basis (h, l))) (i, j)
      by (rule composed-coordinate)
    also have ... =
      tower-coordinate
        (f (sA (lower-basis h) (upper-basis l))) (i, j)
      by (simp only: tower-basis-def fst-conv snd-conv)
    also have ... =
      tower-coordinate
        (sA (lower-basis h) (f (upper-basis l))) (i, j)
```

```
    by (simp only: f.scale)
  also have ... =
    lower.coordinate
      (lower-basis h *
        upper.coordinate (f (upper-basis l)) j) i
    by (simp add: tower-coordinate-def)
  also have ... =
    lower.coordinate
      (upper.coordinate (f (upper-basis l)) j *
        lower-basis h) i
    by (simp only: mult.commute)
  also have ... =
    lower-algebra.regular-matrix
      (upper.matrix-of f $ j $ l) $ i $ h
    by (simp add: lower-algebra.regular-matrix-def
      RS.left-multiplication-def)
  also have ... =
    restricted-matrix (upper.matrix-of f) $ (i, j) $ (h, l)
    by simp
  finally show ?thesis .
 qed
 show ?thesis
 proof (subst vec-eq-iff, intro allI)
  fix ik :: 'i × 'j
  obtain i j where ik: ik = (i, j)
    by (cases ik) blast
  show composed.matrix-of f $ ik =
    restricted-matrix (upper.matrix-of f) $ ik
  proof (subst vec-eq-iff, intro allI)
    fix hl :: 'i × 'j
    obtain h l where hl: hl = (h, l)
      by (cases hl) blast
    show composed.matrix-of f $ ik $ hl =
      restricted-matrix (upper.matrix-of f) $ ik $ hl
      unfolding ik hl
      by (rule entry)
  qed
 qed
qed

theorem algebra-norm-trans:
 tower-algebra.algebra-norm a =
  lower-algebra.algebra-norm (upper-algebra.algebra-norm a)
 by (simp add: SA.left-multiplication-module-hom
    det-restrict-scalars indexed-algebra.algebra-norm-def
    matrix-restrict-scalars tower-algebra.algebra-norm-def
    tower-algebra.regular-matrix-def
    upper-algebra.indexed-algebra-axioms
    upper-algebra.regular-matrix-def)
```

end

end
theory *Algebra-Towers-Examples*
imports
Algebra-Norm
HOL−Computational-Algebra.Primes
begin

10 Examples

10.1 Finite-field towers

datatype *gf4* = *GF4 bool bool*

instantiation *gf4* :: *field*
begin

definition *zero-gf4* :: *gf4*
where *zero-gf4* = *GF4 False False*

definition *one-gf4* :: *gf4*
where *one-gf4* = *GF4 True False*

fun *plus-gf4* :: *gf4* ⇒ *gf4* ⇒ *gf4*
where *plus-gf4* (*GF4 a b*) (*GF4 c d*) =
GF4 (*a* ≠ *c*) (*b* ≠ *d*)

fun *uminus-gf4* :: *gf4* ⇒ *gf4*
where *uminus-gf4* (*GF4 a b*) = *GF4 a b*

definition *minus-gf4* :: *gf4* ⇒ *gf4* ⇒ *gf4*
where *minus-gf4 x y* = *x* + *y*

fun *times-gf4* :: *gf4* ⇒ *gf4* ⇒ *gf4*
where *times-gf4* (*GF4 a b*) (*GF4 c d*) =
GF4 ((*a* ∧ *c*) ≠ (*b* ∧ *d*))
(((*a* ∧ *d*) ≠ (*b* ∧ *c*)) ≠ (*b* ∧ *d*))

fun *inverse-gf4* :: *gf4* ⇒ *gf4*
where
inverse-gf4 (*GF4 False False*) = *GF4 False False*
| *inverse-gf4* (*GF4 True False*) = *GF4 True False*
| *inverse-gf4* (*GF4 False True*) = *GF4 True True*
| *inverse-gf4* (*GF4 True True*) = *GF4 False True*

definition *divide-gf4* :: *gf4* ⇒ *gf4* ⇒ *gf4*
where *divide-gf4 x y* = *x* ∗ *inverse y*

```
lemma inverse-times-gf4:
  assumes x ≠ (0 :: gf4)
  shows inverse x * x = 1
proof (cases x)
  case (GF4 a b)
  then show ?thesis
    using assms
    by (cases a; cases b;
        simp add: zero-gf4-def one-gf4-def)
qed

instance
  apply standard
  subgoal for a b c
    by (cases a; cases b; cases c; auto)
  subgoal for a b
    by (cases a; cases b; auto)
  subgoal for a
    by (cases a; auto simp: one-gf4-def)
  subgoal for a b c
    by (cases a; cases b; cases c; auto)
  subgoal for a b
    by (cases a; cases b; auto)
  subgoal for a
    by (cases a; auto simp: zero-gf4-def)
  subgoal for a
    by (cases a; auto simp: zero-gf4-def)
  subgoal for a b
    by (cases a; cases b; auto simp: minus-gf4-def)
  subgoal for a b c
    by (cases a; cases b; cases c; auto)
  apply (simp-all add: inverse-times-gf4 divide-gf4-def
      zero-gf4-def one-gf4-def)
  done

end

instance gf4 :: finite-field
proof
  show finite (UNIV :: gf4 set)
  proof (rule finite-subset)
    show finite
      {GF4 False False, GF4 True False, GF4 False True, GF4 True True}
      by simp
    show UNIV ⊆
      {GF4 False False, GF4 True False, GF4 False True, GF4 True True}
    proof
      fix x :: gf4
```

```
      assume x ∈ UNIV
      obtain a b where x = GF4 a b
        by (cases x) simp
      show x ∈
        {GF4 False False, GF4 True False, GF4 False True, GF4 True True}
        using ‹x = GF4 a b›
        by (cases a; cases b; simp)
    qed
  qed
qed

locale finite-field-tower-example =
  indexed-field-algebra-tower rS sA rA lower-basis upper-basis
  for rS :: 'r::finite-field ⇒ 's::finite-field ⇒ 's
    and sA :: 's ⇒ 'a::finite-field ⇒ 'a
    and rA :: 'r ⇒ 'a ⇒ 'a
    and lower-basis :: 'i::finite ⇒ 's
    and upper-basis :: 'j::finite ⇒ 'a
begin

sublocale dimensions: indexed-field-tower
  rS sA rA lower-basis upper-basis
  by unfold-locales

theorem finite-field-tower-dimension:
  dimensions.direct-space.dim UNIV =
    dimensions.lower-space.dim UNIV *
      dimensions.upper-space.dim UNIV
  by (rule dimensions.finrank-tower)

theorem finite-field-tower-trace:
  tower-algebra.algebra-trace a =
    lower-algebra.algebra-trace (upper-algebra.algebra-trace a)
  by (rule algebra-trace-trans)

theorem finite-field-tower-norm:
  tower-algebra.algebra-norm a =
    lower-algebra.algebra-norm (upper-algebra.algebra-norm a)
  by (rule algebra-norm-trans)

end
```

The preceding locale covers arbitrary nested finite fields without imposing a separability assumption. The next example supplies a concrete nontrivial extension: the four-element field is represented as *GF4 a b*, corresponding to $a + b\alpha$ over $\mathbb{F}_2$, with $\alpha^2 = \alpha + 1$.

10.2 The four-element field over $\mathbb{F}_2$

lemma *gf2-cases* [*case-names zero one*]:
  fixes *r* :: *gf2*
  obtains *r* = *0* | *r* = *1*
  by *transfer auto*

lemma *gf2-one-add-one* [*simp*]:
  (*1* :: *gf2*) + *1* = *0*
  by *transfer simp*

lemma *gf2-two* [*simp*]:
  (*2* :: *gf2*) = *0*
  using *gf2-one-add-one*
  by (*simp only*: *one-add-one*)

definition *gf2-gf4-embed* :: *gf2* ⇒ *gf4*
  where *gf2-gf4-embed r* = (*if r* = *0 then 0 else 1*)

definition *gf2-gf4-scale* :: *gf2* ⇒ *gf4* ⇒ *gf4*
  where *gf2-gf4-scale r x* = *gf2-gf4-embed r* ∗ *x*

definition *gf4-self-scale* :: *gf4* ⇒ *gf4* ⇒ *gf4*
  where *gf4-self-scale r x* = *r* ∗ *x*

lemma *gf2-gf4-embed-zero* [*simp*]:
  gf2-gf4-embed 0 = *0*
  by (*simp add*: *gf2-gf4-embed-def*)

lemma *gf2-gf4-embed-one* [*simp*]:
  gf2-gf4-embed 1 = *1*
  by (*simp add*: *gf2-gf4-embed-def*)

lemma *gf2-gf4-embed-add* [*simp*]:
  gf2-gf4-embed (*r* + *s*) =
    gf2-gf4-embed r + *gf2-gf4-embed s*
  by (*cases r rule*: *gf2-cases*; *cases s rule*: *gf2-cases*;
      simp add: *gf2-gf4-embed-def zero-gf4-def one-gf4-def*)

lemma *gf2-gf4-embed-mult* [*simp*]:
  gf2-gf4-embed (*r* ∗ *s*) =
    gf2-gf4-embed r ∗ *gf2-gf4-embed s*
  by (*cases r rule*: *gf2-cases*; *cases s rule*: *gf2-cases*;
      simp add: *gf2-gf4-embed-def*)

interpretation *gf2-gf4*: *commutative-algebra gf2-gf4-scale*
  by *unfold-locales*
    (*simp-all add*: *gf2-gf4-scale-def algebra-simps*)

interpretation *gf4-self*: *commutative-algebra gf4-self-scale*

```
  by unfold-locales
    (simp-all add: gf4-self-scale-def algebra-simps)

abbreviation gf4-alpha :: gf4
  where gf4-alpha ≡ GF4 False True

lemma gf4-one-ne-alpha [simp]:
  (1 :: gf4) ≠ gf4-alpha
  by (simp add: one-gf4-def)

definition gf4-basis :: 2 ⇒ gf4
  where gf4-basis i = (if i = 1 then 1 else gf4-alpha)

definition gf4-unit-basis :: 1 ⇒ gf4
  where gf4-unit-basis i = 1

definition gf4-tower-basis :: (2 × 1) ⇒ gf4
  where gf4-tower-basis ij =
    gf4-self-scale (gf4-basis (fst ij))
      (gf4-unit-basis (snd ij))

lemma gf4-basis-range:
  range gf4-basis = {1, gf4-alpha}
proof −
  have index-cases: i = 1 ∨ i = 2 for i :: 2
    using UNIV-2 by auto
  have subset: range gf4-basis ⊆ {1, gf4-alpha}
  proof
    fix x
    assume x ∈ range gf4-basis
    then obtain i where x = gf4-basis i
      by auto
    with index-cases[of i] show x ∈ {1, gf4-alpha}
      by (auto simp: gf4-basis-def)
  qed
  have 1 = gf4-basis (1 :: 2)
    by (simp add: gf4-basis-def)
  moreover have gf4-alpha = gf4-basis (2 :: 2)
    by (simp add: gf4-basis-def one-gf4-def)
  ultimately have {1, gf4-alpha} ⊆ range gf4-basis
    by blast
  with subset show ?thesis
    by blast
qed

lemma gf4-basis-inj:
  inj gf4-basis
proof (rule injI)
  fix i j :: 2
```

```
  have i-cases: i = 1 ∨ i = 2
    using UNIV-2 by auto
  have j-cases: j = 1 ∨ j = 2
    using UNIV-2 by auto
  show gf4-basis i = gf4-basis j ⟹ i = j
    using i-cases j-cases
    by (auto simp: gf4-basis-def one-gf4-def)
qed

lemma gf2-gf4-scale-one-zero-iff [simp]:
  gf2-gf4-scale r 1 = 0 ⟷ r = 0
  by (cases r rule: gf2-cases;
      simp add: gf2-gf4-scale-def gf2-gf4-embed-def)

lemma gf2-gf4-scale-alpha-zero-iff [simp]:
  gf2-gf4-scale r gf4-alpha = 0 ⟷ r = 0
  by (cases r rule: gf2-cases;
      simp add: gf2-gf4-scale-def gf2-gf4-embed-def
        zero-gf4-def one-gf4-def)

lemma gf4-linear-combination-zero-iff [simp]:
  gf2-gf4-scale r 1 +
      gf2-gf4-scale s gf4-alpha = 0 ⟷
    r = 0 ∧ s = 0
  by (cases r rule: gf2-cases; cases s rule: gf2-cases;
      simp add: gf2-gf4-scale-def gf2-gf4-embed-def
        zero-gf4-def one-gf4-def)

lemma gf4-basis-independent:
  gf2-gf4.independent (range gf4-basis)
  unfolding gf2-gf4.independent-explicit-module
proof (intro allI impI)
  fix T c x
  assume subset: T ⊆ range gf4-basis
    and sum-zero:
      (∑ v∈T. gf2-gf4-scale (c v) v) = 0
    and x: x ∈ T
  have T-cases:
    T = {} ∨ T = {1} ∨ T = {gf4-alpha} ∨
      T = {1, gf4-alpha}
    using subset
    by (auto simp: gf4-basis-range)
  then consider
      (empty) T = {}
    | (one) T = {1}
    | (alpha) T = {gf4-alpha}
    | (both) T = {1, gf4-alpha}
    by blast
  then show c x = 0
```

```
  proof cases
    case empty
    with x show ?thesis
      by simp
  next
    case one
    have c 1 = 0
      using sum-zero
      by (simp add: one)
    with one x show ?thesis
      by simp
  next
    case alpha
    have c gf4-alpha = 0
      using sum-zero
      by (simp add: alpha)
    with alpha x show ?thesis
      by simp
  next
    case both
    have coefficients:
      c 1 = 0 ∧ c gf4-alpha = 0
      using sum-zero
      by (simp add: both)
    with both x show ?thesis
      by auto
  qed
qed

lemma gf4-basis-span:
  gf2-gf4.span (range gf4-basis) = UNIV
proof −
  have one-mem:
      (1 :: gf4) ∈ gf2-gf4.span (range gf4-basis)
    by (rule gf2-gf4.span-base)
      (auto simp: gf4-basis-range)
  have alpha-mem:
      gf4-alpha ∈ gf2-gf4.span (range gf4-basis)
    by (rule gf2-gf4.span-base)
      (auto simp: gf4-basis-range)
  have sum-mem:
      1 + gf4-alpha ∈ gf2-gf4.span (range gf4-basis)
    by (rule gf2-gf4.span-add[OF one-mem alpha-mem])
  have x ∈ gf2-gf4.span (range gf4-basis) for x
  proof (cases x)
    case (GF4 a b)
    then show ?thesis
      using gf2-gf4.span-zero one-mem alpha-mem sum-mem
      by (cases a; cases b;
```

```
        simp add: zero-gf4-def one-gf4-def)
  qed
  then show ?thesis
    by auto
qed

lemma gf4-unit-basis-inj:
  inj gf4-unit-basis
  by (auto simp: inj-def)

lemma gf4-unit-basis-range:
  range gf4-unit-basis = {1}
  by (auto simp: gf4-unit-basis-def)

lemma gf4-unit-basis-independent:
  gf4-self.independent (range gf4-unit-basis)
  unfolding gf4-self.independent-explicit-module
proof (intro allI impI)
  fix T c x
  assume subset: T ⊆ range gf4-unit-basis
    and sum-zero:
      (∑ v∈T. gf4-self-scale (c v) v) = 0
    and x: x ∈ T
  have T: T = {1}
    using subset x
    by (auto simp: gf4-unit-basis-range gf4-unit-basis-def)
  have c 1 = 0
    using sum-zero
    by (simp add: T gf4-self-scale-def)
  with T x show c x = 0
    by simp
qed

lemma gf4-unit-basis-span:
  gf4-self.span (range gf4-unit-basis) = UNIV
proof −
  have one-mem:
      (1 :: gf4) ∈ gf4-self.span (range gf4-unit-basis)
    by (rule gf4-self.span-base)
      (auto simp: gf4-unit-basis-def)
  have x ∈ gf4-self.span (range gf4-unit-basis) for x
  proof −
    have gf4-self-scale x 1 ∈
        gf4-self.span (range gf4-unit-basis)
      by (rule gf4-self.span-scale[OF one-mem])
    then show ?thesis
      by (simp add: gf4-self-scale-def)
  qed
  then show ?thesis
```

by *auto*
qed

lemma *gf4-unit-basis-independent-single*:
gf4-self.independent {*gf4-unit-basis* (*1* :: *1*)}
using *gf4-unit-basis-independent*
by (*simp add*: *gf4-unit-basis-range gf4-unit-basis-def*)

lemma *gf4-unit-basis-span-single*:
gf4-self.span {*gf4-unit-basis* (*1* :: *1*)} = *UNIV*
using *gf4-unit-basis-span*
by (*simp add*: *gf4-unit-basis-range gf4-unit-basis-def*)

interpretation *gf4-over-gf2*:
indexed-algebra gf2-gf4-scale gf4-basis
by *unfold-locales*
(*fact gf4-basis-inj gf4-basis-independent*
gf4-basis-span)+

interpretation *gf4-over-itself*:
indexed-algebra gf4-self-scale gf4-unit-basis
by *unfold-locales*
(*fact gf4-unit-basis-inj gf4-unit-basis-independent*
gf4-unit-basis-span)+

interpretation *gf4-over-gf2-space*: *vector-space gf2-gf4-scale*
using *gf2-gf4.module-axioms*
by (*simp add*: *module-iff-vector-space*)

definition *gf2-of-bool* :: *bool* ⇒ *gf2*
where *gf2-of-bool b* = (*if b then 1 else 0*)

The Boolean coordinates of an element determine its coefficients in this basis of the four-element field over the two-element field.

lemma *gf4-expansion-coefficients*:
gf2-gf4-scale r 1 +
gf2-gf4-scale s gf4-alpha = *GF4 a b* ⟷
r = *gf2-of-bool a* ∧ *s* = *gf2-of-bool b*
by (*cases r rule*: *gf2-cases*; *cases s rule*: *gf2-cases*)
(*cases a*; *cases b*; *simp add*: *gf2-of-bool-def gf2-gf4-scale-def*
gf2-gf4-embed-def zero-gf4-def one-gf4-def)+

lemma *gf4-coordinate-one* [*simp*]:
gf4-over-gf2.coordinates.coordinate (*GF4 a b*) (*1* :: *2*) =
gf2-of-bool a
proof −
have *expansion*:
gf2-gf4-scale
(*gf4-over-gf2.coordinates.coordinate* (*GF4 a b*) (*1* :: *2*)) *1* +

```
    gf2-gf4-scale
      (gf4-over-gf2.coordinates.coordinate (GF4 a b) (2 :: 2))
        gf4-alpha =
    GF4 a b
  using gf4-over-gf2.coordinates.basis-expansion[of GF4 a b]
  by (simp add: UNIV-2 gf4-basis-def one-gf4-def)
show ?thesis
  using gf4-expansion-coefficients[THEN iffD1, OF expansion]
  by simp
qed

lemma gf4-coordinate-two [simp]:
  gf4-over-gf2.coordinates.coordinate (GF4 a b) (2 :: 2) =
    gf2-of-bool b
proof −
  have expansion:
    gf2-gf4-scale
       (gf4-over-gf2.coordinates.coordinate (GF4 a b) (1 :: 2)) 1 +
    gf2-gf4-scale
      (gf4-over-gf2.coordinates.coordinate (GF4 a b) (2 :: 2))
        gf4-alpha =
    GF4 a b
  using gf4-over-gf2.coordinates.basis-expansion[of GF4 a b]
  by (simp add: UNIV-2 gf4-basis-def one-gf4-def)
show ?thesis
  using gf4-expansion-coefficients[THEN iffD1, OF expansion]
  by simp
qed

lemma gf4-regular-matrix-11 [simp]:
  gf4-over-gf2.regular-matrix (GF4 a b) $
     (1 :: 2) $ (1 :: 2) = gf2-of-bool a
  by (cases a; cases b;
     simp add: gf4-over-gf2.regular-matrix-def
       gf2-gf4.left-multiplication-def gf4-basis-def
       gf2-of-bool-def one-gf4-def)

lemma gf4-regular-matrix-12 [simp]:
  gf4-over-gf2.regular-matrix (GF4 a b) $
     (1 :: 2) $ (2 :: 2) = gf2-of-bool b
  by (cases a; cases b;
     simp add: gf4-over-gf2.regular-matrix-def
       gf2-gf4.left-multiplication-def gf4-basis-def
       gf2-of-bool-def one-gf4-def)

lemma gf4-regular-matrix-21 [simp]:
  gf4-over-gf2.regular-matrix (GF4 a b) $
     (2 :: 2) $ (1 :: 2) = gf2-of-bool b
  by (cases a; cases b;
```

```
    simp add: gf4-over-gf2.regular-matrix-def
      gf2-gf4.left-multiplication-def gf4-basis-def
      gf2-of-bool-def one-gf4-def)

lemma gf4-regular-matrix-22 [simp]:
  gf4-over-gf2.regular-matrix (GF4 a b) $
    (2 :: 2) $ (2 :: 2) = gf2-of-bool (a ≠ b)
  by (cases a; cases b;
    simp add: gf4-over-gf2.regular-matrix-def
      gf2-gf4.left-multiplication-def gf4-basis-def
      gf2-of-bool-def one-gf4-def)

lemma gf4-algebra-trace:
  gf4-over-gf2.algebra-trace (GF4 a b) = gf2-of-bool b
  by (cases a; cases b;
    simp add: gf4-over-gf2.algebra-trace-def trace-def
      UNIV-2 gf2-of-bool-def)

lemma gf4-algebra-norm:
  gf4-over-gf2.algebra-norm (GF4 a b) =
    gf2-of-bool (a ∨ b)
  unfolding gf4-over-gf2.algebra-norm-def det-2
  by (cases a; cases b;
    simp add: gf2-of-bool-def)

theorem gf4-over-gf2-trace-norm:
  gf4-over-gf2.algebra-trace gf4-alpha = 1 ∧
    gf4-over-gf2.algebra-norm gf4-alpha = 1
  by (simp add: gf4-algebra-trace gf4-algebra-norm
    gf2-of-bool-def)

theorem gf4-over-gf2-dimension:
  gf4-over-gf2-space.dim UNIV = 2
proof −
  have dimension:
    card (range gf4-basis) = gf4-over-gf2-space.dim UNIV
    by (rule gf4-over-gf2-space.basis-card-eq-dim)
      (simp-all add: gf4-basis-span gf4-basis-independent)
  show ?thesis
    using dimension
    by (simp add: gf4-basis-range)
qed

theorem concrete-finite-field-tower-laws:
  indexed-algebra.algebra-trace
      gf2-gf4-scale gf4-tower-basis x =
    gf4-over-gf2.algebra-trace
      (gf4-over-itself.algebra-trace x) ∧
   indexed-algebra.algebra-norm
```

```
    gf2-gf4-scale gf4-tower-basis x =
   gf4-over-gf2.algebra-norm
    (gf4-over-itself.algebra-norm x)
proof −
 interpret tower: indexed-field-algebra-tower
  gf2-gf4-scale gf4-self-scale gf2-gf4-scale
  gf4-basis gf4-unit-basis
  by unfold-locales
   (simp-all add: gf2-gf4-scale-def gf4-self-scale-def
     algebra-simps gf4-basis-inj gf4-basis-independent
     gf4-basis-span gf4-unit-basis-inj gf4-unit-basis-range
     gf4-unit-basis-independent gf4-unit-basis-span
     gf4-unit-basis-independent-single gf4-unit-basis-span-single)
 note trace-law = tower.algebra-trace-trans[of x]
 note norm-law = tower.algebra-norm-trans[of x]
 have basis-eq: tower.tower-basis = gf4-tower-basis
  by (rule ext)
   (simp add: tower.tower-basis-def gf4-tower-basis-def)
 show ?thesis
  using trace-law norm-law
  by (simp only: basis-eq)
qed
```

10.3 The quadratic extension of the reals by the imaginary unit

```
definition real-complex-scale :: real ⇒ complex ⇒ complex
 where real-complex-scale r z = of-real r ∗ z

definition real-complex-direct-scale :: real ⇒ complex ⇒ complex
 where real-complex-direct-scale r z = of-real r ∗ z

definition complex-complex-scale :: complex ⇒ complex ⇒ complex
 where complex-complex-scale z w = z ∗ w

definition complex-basis :: 2 ⇒ complex
 where complex-basis i = (if i = 1 then 1 else i)

definition complex-unit-basis :: 1 ⇒ complex
 where complex-unit-basis i = 1

definition complex-tower-basis :: (2 × 1) ⇒ complex
 where complex-tower-basis ij =
  complex-complex-scale (complex-basis (fst ij))
   (complex-unit-basis (snd ij))

interpretation real-complex: commutative-algebra real-complex-scale
 by unfold-locales
  (simp-all add: real-complex-scale-def algebra-simps)
```

interpretation *complex-complex*: *commutative-algebra complex-complex-scale*
by *unfold-locales*
(*simp-all add*: *complex-complex-scale-def algebra-simps*)

lemma *complex-basis-range*:
range complex-basis = {*1*, i}
proof −
have *index-cases*: *i* = *1* ∨ *i* = *2* **for** *i* :: *2*
using *UNIV-2* **by** *auto*
have *subset*: *range complex-basis* ⊆ {*1*, i}
proof
fix *z*
assume *z* ∈ *range complex-basis*
then obtain *i* **where** *z* = *complex-basis i*
by *auto*
with *index-cases*[*of i*] **show** *z* ∈ {*1*, i}
by (*auto simp*: *complex-basis-def*)
qed
have *1* = *complex-basis* (*1* :: *2*)
by (*simp add*: *complex-basis-def*)
moreover have i = *complex-basis* (*2* :: *2*)
by (*simp add*: *complex-basis-def*)
ultimately have {*1*, i} ⊆ *range complex-basis*
by *blast*
with *subset* **show** *?thesis*
by *blast*
qed

lemma *complex-basis-inj*:
inj complex-basis
proof (*rule injI*)
fix *i j* :: *2*
have *i-cases*: *i* = *1* ∨ *i* = *2*
using *UNIV-2* **by** *auto*
have *j-cases*: *j* = *1* ∨ *j* = *2*
using *UNIV-2* **by** *auto*
show *complex-basis i* = *complex-basis j* ⟹ *i* = *j*
using *i-cases j-cases*
by (*auto simp*: *complex-basis-def complex-eq-iff*)
qed

lemma *complex-basis-independent*:
real-complex.independent (*range complex-basis*)
unfolding *real-complex.independent-explicit-module*
proof (*intro allI impI*)
fix *T c w*
assume *finite-T*: *finite T*
and *subset*: *T* ⊆ *range complex-basis*

```
    and sum-zero:
      (∑ z∈T. real-complex-scale (c z) z) = 0
    and w: w ∈ T
  have T-cases:
    T = {} ∨ T = {1} ∨ T = {i} ∨ T = {1, i}
    using subset
    by (auto simp: complex-basis-range)
  then consider
      (empty) T = {}
    | (one) T = {1}
    | (imaginary) T = {i}
    | (both) T = {1, i}
    by blast
  then show c w = 0
  proof cases
    case empty
    with w show ?thesis
      by simp
  next
    case one
    have coefficient: c 1 = 0
      using sum-zero
      by (simp add: one real-complex-scale-def)
    with one w show ?thesis
      by simp
  next
    case imaginary
    have coefficient: c i = 0
      using sum-zero
      by (simp add: imaginary real-complex-scale-def)
    with imaginary w show ?thesis
      by simp
  next
    case both
    have coefficient-one: c 1 = 0
      using sum-zero
      by (simp add: both real-complex-scale-def complex-eq-iff)
    have coefficient-imaginary: c i = 0
      using sum-zero
      by (simp add: both real-complex-scale-def complex-eq-iff)
    with coefficient-one both w show ?thesis
      by auto
  qed
qed

lemma complex-basis-span:
  real-complex.span (range complex-basis) = UNIV
proof −
  have one-mem: (1 :: complex) ∈
```

```
    real-complex.span (range complex-basis)
   by (rule real-complex.span-base) (auto simp: complex-basis-range)
  have imaginary-mem: i ∈
    real-complex.span (range complex-basis)
   by (rule real-complex.span-base) (auto simp: complex-basis-range)
  have expansion:
   real-complex-scale (Re z) 1 +
    real-complex-scale (Im z) i = z for z
  proof −
   have z = of-real (Re z) + i ∗ of-real (Im z)
    by (rule complex-eq)
   then show ?thesis
    by (simp add: real-complex-scale-def mult.commute)
  qed
  have z ∈ real-complex.span (range complex-basis) for z
  proof −
   have real-complex-scale (Re z) 1 +
      real-complex-scale (Im z) i ∈
     real-complex.span (range complex-basis)
     by (rule real-complex.span-add)
      (rule real-complex.span-scale[OF one-mem],
       rule real-complex.span-scale[OF imaginary-mem])
   then show ?thesis
    by (simp only: expansion)
  qed
  then show ?thesis
   by auto
qed

lemma complex-unit-basis-inj:
  inj complex-unit-basis
  by (auto simp: inj-def)

lemma complex-unit-basis-range:
  range complex-unit-basis = {1}
  by (auto simp: complex-unit-basis-def)

lemma complex-unit-basis-independent:
  complex-complex.independent (range complex-unit-basis)
  unfolding complex-complex.independent-explicit-module
proof (intro allI impI)
  fix T c w
  assume subset: T ⊆ range complex-unit-basis
   and sum-zero:
    (∑ z∈T. complex-complex-scale (c z) z) = 0
   and w: w ∈ T
  have T: T = {1}
   using subset w
   by (auto simp: complex-unit-basis-range complex-unit-basis-def)
```

```
  have c 1 = 0
    using sum-zero
    by (simp add: T complex-complex-scale-def)
  with T w show c w = 0
    by simp
qed

lemma complex-unit-basis-span:
  complex-complex.span (range complex-unit-basis) = UNIV
proof −
  have one-mem: (1 :: complex) ∈
      complex-complex.span (range complex-unit-basis)
    by (rule complex-complex.span-base)
      (auto simp: complex-unit-basis-def)
  have z ∈ complex-complex.span (range complex-unit-basis) for z
  proof −
    have complex-complex-scale z 1 ∈
      complex-complex.span (range complex-unit-basis)
      by (rule complex-complex.span-scale[OF one-mem])
    then show ?thesis
      by (simp add: complex-complex-scale-def)
  qed
  then show ?thesis
    by auto
qed

lemma complex-unit-basis-independent-single:
  complex-complex.independent {complex-unit-basis (1 :: 1)}
  using complex-unit-basis-independent
  by (simp add: complex-unit-basis-range complex-unit-basis-def)

lemma complex-unit-basis-span-single:
  complex-complex.span {complex-unit-basis (1 :: 1)} = UNIV
  using complex-unit-basis-span
  by (simp add: complex-unit-basis-range complex-unit-basis-def)

interpretation complex-real-algebra:
    indexed-algebra real-complex-scale complex-basis
  by unfold-locales
    (fact complex-basis-inj complex-basis-independent
      complex-basis-span)+

interpretation complex-scalar-algebra:
    indexed-algebra complex-complex-scale complex-unit-basis
  by unfold-locales
    (fact complex-unit-basis-inj complex-unit-basis-independent
      complex-unit-basis-span)+

lemma complex-coordinate-one [simp]:
```

complex-real-algebra.coordinates.coordinate z (1 :: 2) = Re z
using *complex-real-algebra.coordinates.basis-expansion*[*of z*]
by (*simp add*: *UNIV-2 complex-basis-def real-complex-scale-def complex-eq-iff*)

lemma *complex-coordinate-two* [*simp*]:
complex-real-algebra.coordinates.coordinate z (2 :: 2) = Im z
using *complex-real-algebra.coordinates.basis-expansion*[*of z*]
by (*simp add*: *UNIV-2 complex-basis-def real-complex-scale-def complex-eq-iff*)

lemma *complex-regular-matrix-11* [*simp*]:
complex-real-algebra.regular-matrix z $ (1 :: 2) $ (1 :: 2) = Re z
by (*simp add*: *complex-real-algebra.regular-matrix-def real-complex.left-multiplication-def complex-basis-def*)

lemma *complex-regular-matrix-12* [*simp*]:
complex-real-algebra.regular-matrix z $ (1 :: 2) $ (2 :: 2) = − Im z
by (*simp add*: *complex-real-algebra.regular-matrix-def real-complex.left-multiplication-def complex-basis-def*)

lemma *complex-regular-matrix-21* [*simp*]:
complex-real-algebra.regular-matrix z $ (2 :: 2) $ (1 :: 2) = Im z
by (*simp add*: *complex-real-algebra.regular-matrix-def real-complex.left-multiplication-def complex-basis-def*)

lemma *complex-regular-matrix-22* [*simp*]:
complex-real-algebra.regular-matrix z $ (2 :: 2) $ (2 :: 2) = Re z
by (*simp add*: *complex-real-algebra.regular-matrix-def real-complex.left-multiplication-def complex-basis-def*)

lemma *complex-algebra-trace*:
complex-real-algebra.algebra-trace z = 2 ∗ Re z
by (*simp add*: *complex-real-algebra.algebra-trace-def trace-def UNIV-2*)

lemma *complex-algebra-norm*:
complex-real-algebra.algebra-norm z = (Re z)² + (Im z)²
unfolding *complex-real-algebra.algebra-norm-def det-2*
by (*simp add*: *power2-eq-square*)

theorem *quadratic-extension-trace-norm*:
complex-real-algebra.algebra-trace z = 2 ∗ Re z ∧
complex-real-algebra.algebra-norm z = (Re z)² + (Im z)²

```
  by (simp add: complex-algebra-trace complex-algebra-norm)

theorem complex-quadratic-tower-laws:
  indexed-algebra.algebra-trace
      real-complex-direct-scale complex-tower-basis z =
    complex-real-algebra.algebra-trace
      (complex-scalar-algebra.algebra-trace z) ∧
   indexed-algebra.algebra-norm
      real-complex-direct-scale complex-tower-basis z =
    complex-real-algebra.algebra-norm
      (complex-scalar-algebra.algebra-norm z)
proof −
  interpret tower: indexed-field-algebra-tower
    real-complex-scale complex-complex-scale
    real-complex-direct-scale complex-basis complex-unit-basis
    by unfold-locales
      (simp-all add: real-complex-scale-def
        real-complex-direct-scale-def complex-complex-scale-def
        algebra-simps complex-basis-inj complex-basis-independent
        complex-basis-span complex-unit-basis-inj
        complex-unit-basis-range complex-unit-basis-independent
        complex-unit-basis-span complex-unit-basis-independent-single
        complex-unit-basis-span-single)
  have tower.tower-basis = complex-tower-basis
    by (force simp: tower.tower-basis-def complex-tower-basis-def)
  then show ?thesis
    using tower.algebra-norm-trans tower.algebra-trace-trans by force
qed

end
```

Acknowledgments

AI assistance was used for proof engineering.

References

[1] S. Lang. *Algebra*. Springer, New York, third edition, 2002. DOI: `https://doi.org/10.1007/978-1-4613-0041-0`.