Rank-Nullity Theorem in Linear Algebra

By Jose Divasón and Jesús Aransay*

September 19, 2015

Abstract

In this contribution, we present some formalizations based on the HOL-Multivariate-Analysis session of Isabelle. Firstly, a generalization of several theorems of such library are presented. Secondly, some definitions and proofs involving Linear Algebra and the four fundamental subspaces of a matrix are shown. Finally, we present a proof of the result known in Linear Algebra as the “Rank-Nullity Theorem”, which states that, given any linear map \( f \) from a finite dimensional vector space \( V \) to a vector space \( W \), then the dimension of \( V \) is equal to the dimension of the kernel of \( f \) (which is a subspace of \( V \)) and the dimension of the range of \( f \) (which is a subspace of \( W \)). The proof presented here is based on the one given in [1]. As a corollary of the previous theorem, and taking advantage of the relationship between linear maps and matrices, we prove that, for every matrix \( A \) (which has associated a linear map between finite dimensional vector spaces), the sum of its null space and its column space (which is equal to the range of the linear map) is equal to the number of columns of \( A \).

Contents

1 Generalizations 2
   1.1 Generalization of parts of the HMA library .............. 2

2 Dual Order 55
   2.1 Interpretation of dual order based on order .............. 55
   2.2 Computable greatest operator ............................. 56

3 Class for modular arithmetic 56
   3.1 Definition and properties ................................. 57
   3.2 Conversion between a modular class and the subset of natural numbers associated .................. 58
   3.3 Instantiations ............................................. 67

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

4.1 Definitions of number of rows and columns of a matrix . . . . 69
4.2 Basic properties about matrices . . . . . . . . . . . . . . . . 69
4.3 Theorems obtained from the AFP . . . . . . . . . . . . . . . 70
4.4 Basic properties involving span, linearity and dimensions . . 72
4.5 Basic properties about matrix multiplication . . . . . . . . . 74
4.6 Properties about invertibility . . . . . . . . . . . . . . . . . 75
4.7 Properties about the dimension of vectors . . . . . . . . . . . 77
4.8 Instantiations and interpretations . . . . . . . . . . . . . . . 78
4.9 Properties about lists . . . . . . . . . . . . . . . . . . . . . . 79

5 Fundamental Subspaces

5.1 The fundamental subspaces of a matrix . . . . . . . . . . . . 80
  5.1.1 Definitions . . . . . . . . . . . . . . . . . . . . . . . . . . 80
  5.1.2 Relationships among them . . . . . . . . . . . . . . . . . 80
5.2 Proving that they are subspaces . . . . . . . . . . . . . . . . . 81
5.3 More useful properties and equivalences . . . . . . . . . . . . 81

6 Rank Nullity Theorem of Linear Algebra

6.1 Previous results . . . . . . . . . . . . . . . . . . . . . . . . . . 84
6.2 The proof . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
6.3 The rank nullity theorem for matrices . . . . . . . . . . . . . 91

1 Generalizations

theory Generalizations
imports
  ~/src/HOL/Multivariate-Analysis/Multivariate-Analysis
begin

1.1 Generalization of parts of the HMA library

In this file, some parts of the Multivariate Analysis library required for our
formalizations of both the Rank Nullity Theorem and the Gauss-Jordan
algorithm are generalized.

Mainly, we have carried out four kinds of generalizations:

1. Lemmas involving real vector spaces (that is, lemmas that used the
   real-vector class) are now generalized to vector spaces over any field.

2. Some lemmas involving euclidean spaces (the euclidean-space class)
   have been generalized to finite dimensional vector spaces.

3. Lemmas involving real matrices have been generalized to matrices over
   any field.
4. Lemmas about determinants involving the class \textit{linordered-idom}, such as the lemma \textit{det-identical-columns}, are now proven using the class \textit{comm-ring-1}.

\textbf{hide-const (open)} \textit{span} \\
\textbf{hide-const (open)} \textit{dependent} \\
\textbf{hide-const (open)} \textit{independent} \\
\textbf{hide-const (open)} \textit{dim}

\textbf{interpretation} \textit{vec: vector-space op s :: 'a::field => 'a *b = > 'a *b} \\
by (unfold-locales, simp-all)

\textbf{locale} \textit{linear = B: vector-space scaleB + C: vector-space scaleC} \\
\textbf{for} scaleB :: ('a::field => 'b::ab-group-add => 'b) \textbf{(infixr \textit{*} 75)} \\
\textbf{and} scaleC :: ('a => 'c::ab-group-add => 'c) \textbf{(infixr \textit{*}c 75)} + \\
\textbf{fixes} \textit{f :: (scaleB)} \\
\textbf{assumes} cmult: \textit{f (r *b x) = r *c (f x)} \\
\textbf{and} add: \textit{f (a + b) = f a + f b}

begin

\textbf{lemma} \textit{linear-0: f 0 = 0} \\
by (metis add eq-add-iff)

\textbf{lemma} \textit{linear-cmul: f (c *b x) = c *c (f x)} \\
by (metis cmult)

\textbf{lemma} \textit{linear-neg: f (- x) = - f x} \\
using \textit{linear-cmul [where c=-1]} \\
by (metis add add-eq-0-iff linear-0)

\textbf{lemma} \textit{linear-add: f (x + y) = f x + f y} \\
by (metis add)

\textbf{lemma} \textit{linear-sub: f (x - y) = f x - f y} \\
by (metis diff-conv-add-uminus linear-add linear-neg)

\textbf{lemma} \textit{linear-setsum:} \\
\textbf{assumes} \textit{fin: finite S} \\
\textbf{shows} \textit{f (setsum g S) = setsum (f o g) S} \\
using \textit{fin}

\textbf{proof} \textit{induct}
case empty  
then show ?case 
  by (simp add: linear-0)
next 
case (insert x F) 
  have f (setsum g (insert x F)) = f (g x + setsum g F) 
    using insert.hyps by simp 
  also have ... = f (g x) + f (setsum g F) 
    using linear-add by simp 
  also have ... = setsum (f ∘ g) (insert x F) 
    using insert.hyps by simp 
finally show ?case .
qed

lemma linear-setsum-mul: 
  assumes fin: finite S 
  shows f (setsum (λ i. c i * b v i) S) = setsum (λ i. c i * c (f (v i))) S 
  using linear-setsum[OF fin] linear-cmul 
  by simp

lemma linear-injective-0: 
  shows inj f ←→ (∀ x. f x = 0 −→ x = 0) 
proof 
  have inj f ←→ (∀ x y. f x = f y −→ x = y) 
    by (simp add: inj-on-def) 
  also have ... ←→ (∀ x y. f x - f y = 0 −→ x - y = 0) 
    by simp 
  also have ... ←→ (∀ x y. f (x - y) = 0 −→ x - y = 0) 
    by (simp add: linear-sub) 
  also have ... ←→ (∀ x. f x = 0 −→ x = 0) 
    by auto 
finally show ?thesis .
qed

end

lemma linear-iff: 
  linear scaleB scaleC f ←→ (vector-space scaleB) ∧ (vector-space scaleC) 
  ∧ (∀ x y. f (x + y) = f x + f y) ∧ (∀ c x. f (scaleB c x) = scaleC c (f x)) 
(is linear scaleB scaleC f ←→ ?rhs)
proof 
  assume lf: linear scaleB scaleC f then interpret f: linear scaleB scaleC f . 
  have B: vector-space scaleB using lf unfolding linear-def by simp 
  moreover have C: vector-space scaleC using lf unfolding linear-def by simp 
  ultimately show ?rhs using f.linear-add f.linear-cmul by simp
next 
  assume ?rhs then show linear scaleB scaleC f
by (unfold-locales, auto simp add: vector-space.scale-right-distrib vector-space.scale-left-distrib vector-space.scale-scale vector-space.scale-one)

qed

lemma linear-iff2:
linear (op s) (op s) f ⟷ (∀ x y. f (x + y) = f x + f y) ∧ (∀ c. f (c * s x) = c * s (f x))
(is linear (op s) (op s) f ⟷ ?rhs)

proof
assume linear (op s) (op s) f then interpret f::linear (op s) (op s) f .
show ?rhs by (metis f.linear-add f.linear-cmul)

next
assume ?rhs then show linear (op s) (op s) f by (unfold-locales,auto)

qed

lemma linear-compose-sub: linear scale scaleC f ⇒ linear scale scaleC g ⇒ linear scale scaleC (λx. f x − g x)
unfolding linear-iff by (simp add: vector-space.scale-right-diff-distrib)

lemma linear-compose: linear scale scaleC f ⇒ linear scale scaleT g ⇒ linear scale scaleT (g o f)
unfolding linear-iff by auto

context vector-space
begin

lemma linear-id: linear scale scale id
by (simp add: linear-iff, unfold-locales)

lemma scale-minus1-left[simp]:
shows scale (−1) x = − x
using scale-minus-left [of 1 x] by simp

definition subspace :: 'b set ⇒ bool
  where subspace S ⟷ 0 ∈ S ∧ (∀ x ∈ S. ∀ y ∈ S. x + y ∈ S) ∧ (∀ c. ∀ x ∈ S. scale c x ∈ S)

definition span (S::'b set) = (subspace hull S)
definition dependent S ⟷ (∃ a ∈ S. a ∈ span (S − {a}))
definition independent s ≡ ¬ dependent s

Closure properties of subspaces.

lemma subspace-UNIV[simp]: subspace UNIV
  by (simp add: subspace-def)

lemma subspace-0: subspace S ⟷ 0 ∈ S
  by (metis subspace-def)
lemma subspace-add: subspace \( S \) \( \Rightarrow \) \( x \in S \Rightarrow y \in S \Rightarrow x + y \in S \)
by (metis subspace-def)

lemma subspace-mul: subspace \( S \) \( \Rightarrow \) \( x \in S \Rightarrow \text{scale } c \ x \in S \)
by (metis subspace-def)

lemma subspace-neg: subspace \( S \) \( \Rightarrow \) \( x \in S \Rightarrow -x \in S \)
by (metis scale-minus-left scale-one subspace-mul)

lemma subspace-sub: subspace \( S \) \( \Rightarrow \) \( x \in S \Rightarrow y \in S \Rightarrow x - y \in S \)
by (metis diff-conv-add-uminus subspace-add subspace-neg)

lemma subspace-setsum:
assumes \( sA: \text{subspace } A \) \and \( fB: \text{finite } B \) \and \( f: \forall x \in B. \ f \ x \in A \)
shows \( \text{setsum } f \ B \in A \)
using \( fB \ f sA \)
by (induct rule: finite-induct[\( \text{OF } fB \)])
(\text{simp add: subspace-def sA, auto simp add: sA subspace-add})

lemma subspace-linear-image:
assumes \( lf: \text{linear } \text{scale } \text{scaleC } f \) \and \( sS: \text{subspace } S \)
shows \( \text{vector-space. subspace scaleC } (f ' S) \)
proof –
interpret \( lf: \text{linear } \text{scale } \text{scaleC } f \)
using \( lf \)
by simp
have \( C: \text{vector-space scaleC } f \)
using \( lf \)
unfolding \( \text{linear-def} \)
by simp
show \( \text{thesis} \)
proof (\text{unfold vector-space. subspace-def[\( \text{OF } C \)], auto})
show \( 0 \in f ' S \)
by (metis \( \text{full-types} \) \text{image-eqI} \( \text{lf.linear-0} \ sS \text{ subspace-0} \))
fix \( x \ y \)
assume \( x: \ x \in S \) \and \( y: \ y \in S \)
show \( f \ x + f \ y \in f ' S \)
unfolding \text{image-iff}
apply (\text{rule-tac } x\text{=}x + y \text{ in}\text{ bexI})
using \( \text{lf.add subspace-add[\( \text{OF } sS \ x \ y \) by auto} \)
fix \( c \)
show \( \text{scaleC } c \ (f \ x) \in f ' S \)
by (metis \text{imageI subspace-mul} \( \text{lf.linear-cmul} \ sS \ x \))
qed
qed

lemma subspace-linear-vimage:
assumes \( lf: \text{linear } \text{scale } \text{scaleC } (f::'b::\text{ab-group-add}=>'c::\text{ab-group-add}) \) \and \( s: \text{vector-space. subspace scaleC } S \)
shows \( \text{subspace } (f ' S) \)
proof –
interpret \( lf: \text{linear } \text{scale } \text{scaleC } f \)
using \( lf \)
by simp
have \( C: \text{vector-space scaleC } f \)
using \( lf \)
(\text{unfold-locales})
show ?thesis
  unfolding subspace-def
  apply (auto)
  apply (metis if.C.subspace-0 if.linear-0 s)
  apply (metis (full-types) if.C.subspace-def if.linear-add s)
  by (metis if.C.subspace-mul if.linear-cmul s)
qed

lemma subspace-Times:
  assumes A: subspace A and B: subspace B
  shows vector-space.subspace (λx (a,b). (scale x a, scale x b)) (A × B)
proof –
  have v: vector-space (λx (a,b). (scale x a, scale x b))
    unfolding vector-space-def
    by (simp add: scale-left-distrib scale-right-distrib)
  show ?thesis
    using A B unfolding subspace-def
    unfolding vector-space.subspace-def[OF v] zero-prod-def by auto
qed

lemma vector-space-product: vector-space (λx (a, b). (scale x a, scale x b))
  by (unfold-locales, auto simp: scale-right-distrib scale-left-distrib)

Properties of span.

lemma span-mono: A ⊆ B ==> span A ⊆ span B
  by (metis span-def hull-mono)

lemma subspace-span: subspace (span S)
  unfolding span-def
  apply (rule hull-in)
  apply (simp only: subspace-def Inter-iff Int-iff subset-eq)
  apply auto
  done

lemma span-clauses:
  a ∈ S ==> a ∈ span S
  0 ∈ span S
  x ∈ span S ==> y ∈ span S ==> x + y ∈ span S
  x ∈ span S ==> scale c x ∈ span S
  by (metis span-def hull-subset subset-eq) (metis subspace-span subspace-def)

lemma span-unique:
  S ⊆ T ==> subspace T ==> (!!T'. S ⊆ T' ==> subspace T' ==> T ⊆ T')
  ==> span S = T
  unfolding span-def by (rule hull-unique)
lemma span-minimal: $S \subseteq T \implies$ subspace $T \implies$ span $S \subseteq T$
unfolding span-def by (rule hull-minimal)

lemma span-induct:
assumes $x: x \in \text{span } S$
and $P: \text{subspace } P$
and $SP: \forall x. x \in S \implies x \in P$
shows $x \in P$
proof -
  from $SP$ have $SP': S \subseteq P$
  by (simp add: subset-eq)
from $x$ hull-minimal[where $S=\text{subspace } OF SP' P$, unfolded span-def[symmetric]]
show $x \in P$
  by (metis subset-eq)
qed

lemma span-empty[simp]: span $\{\}$ = $\{0\}$
apply (simp add: span-def)
apply (rule hull-unique)
apply (auto simp add: subspace-def)
done

lemma independent-empty[intro]: independent $\{\}$
  by (simp add: dependent-def)

lemma dependent-single[simp]: dependent $\{x\} \iff x = 0$
unfolding dependent-def by auto

lemma independent-mono: independent $A \implies B \subseteq A \implies$ independent $B$
apply (clarsimp simp add: dependent-def span-mono)
apply (subgoal-tac span ($B - \{a\}$) $\leq$ span ($A - \{a\}$))
apply force
apply (rule span-mono)
apply auto
done

lemma span-subspace: $A \subseteq B \implies B \leq \text{span } A \implies$ subspace $B \implies$ span $A = B$
  by (metis order-antisym span-def hull-minimal)

lemma span-induct':
assumes $SP: \forall x \in S. P x$
  and $P: \text{subspace } \{x. P x\}$
shows $\forall x \in \text{span } S. P x$
using span-induct $SP P$ by blast

inductive-set span-induct-alt-help for $S:: 'b set$
where
  \textit{span-induct-alt-help-0: } 0 \in \textit{span-induct-alt-help } S
| \textit{span-induct-alt-help-S: }
  \begin{align*}
  x \in S & \implies z \in \textit{span-induct-alt-help } S \implies \scale c x + z \in \textit{span-induct-alt-help } S
  \end{align*}

\textbf{lemma} \textit{span-induct-alt'}.:
\begin{align*}
\text{assumes } h0 : & h 0 \\
\text{and } hS : & !!c \ x \ y, \ x \in S \implies y \implies h (\scale c x + y)
\end{align*}
\textbf{shows} \( \forall x \in \textit{span } S, \ h x \)
\textbf{proof} –
\begin{align*}
\{ & \\
\text{fix } x :: 'b \\
\text{assume } x : x \in \textit{span-induct-alt-help } S \\
\text{have } & h x \\
\text{apply } (\text{rule } \textit{span-induct-alt-help}.\text{induct}[OF x]) \\
\text{apply } (\text{rule } h0) \\
\text{apply } (\text{rule } hS) \\
\text{apply } & \text{assumption} \\
\text{apply } & \text{assumption} \\
\text{done}
\}
\text{note th0 = this}
\{ & \\
\text{fix } x \\
\text{assume } x : x \in \textit{span } S \\
\text{have } & x \in \textit{span-induct-alt-help } S \\
\text{proof } (\text{rule } \textit{span-induct}[\textbf{where } x=x \ \textbf{and } S=S]) \\
\text{show } & x \in \textit{span } S \ \textbf{by } (\text{rule } x)
\}
\text{next}
\{ & \\
\text{fix } x \\
\text{assume } xS : x \in S \\
\text{from } & \textit{span-induct-alt-help-S}[OF xS \ \textit{span-induct-alt-help-0}, \ \text{of } 1] \\
\text{show } & x \in \textit{span-induct-alt-help } S \\
\text{by } & \text{simp}
\}
\text{next}
\{ & \\
\text{have } & 0 \in \textit{span-induct-alt-help } S \ \textbf{by } (\text{rule } \textit{span-induct-alt-help-0}) \\
\text{moreover} \\
\{ & \\
\text{fix } x \ y \\
\text{assume } h : x \in \textit{span-induct-alt-help } S \ y \in \textit{span-induct-alt-help } S \\
\text{from } h & \text{have } (x + y) \in \textit{span-induct-alt-help } S \\
\text{apply } & \text{(induct rule: \textit{span-induct-alt-help}.\text{induct})} \\
\text{apply } & \text{simp} \\
\text{unfolding } & \text{add.assoc} \\
\text{apply } (\text{rule } \textit{span-induct-alt-help-S}) \\
\text{apply } & \text{assumption} \\
\text{apply } & \text{simp} \\
\text{done}
\}
\end{align*}
moreover

\begin{align*}
\text{fix } c, x \\
\text{assume } \forall x : x \in \text{span-induct-alt-help } S \\
\text{then have (} \text{scale } c, x \text{) } \in \text{span-induct-alt-help } S \\
\text{apply (} \text{induct rule: span-induct-alt-help.induct} \text{)} \\
\text{apply (} \text{simp add: span-induct-alt-help-0} \text{)} \\
\text{apply (} \text{simp add: scale-right-distrib} \text{)} \\
\text{apply (} \text{rule span-induct-alt-help-S} \text{)} \\
\text{apply assumption} \\
\text{apply simp} \\
\text{done}
\end{align*}

ultimately show \text{subspace (span-induct-alt-help } S \text{)}

unfolding \text{subspace-def Ball-def by blast}

qed

\begin{align*}
\text{with th0 show ?thesis by blast}
\end{align*}

qed

\begin{lemma}
\text{span-induct-alt:}
\text{assumes h0: h 0}
\text{and hS: } \forall c, x, y : x \in S \Longrightarrow h y \Longrightarrow h (\text{scale } c, x + y)
\text{and } x : x \in \text{span } S
\text{shows h x}
\text{using span-induct-alt"[of h S] h0 hS x by blast}
\end{lemma}

Individual closure properties.

\begin{lemma}
\text{span-span: span (span A) = span A}
\text{unfolding span-def hull-hull ..}
\end{lemma}

\begin{lemma}
\text{span-superset: x } \in S \Longrightarrow x \in \text{span } S
\text{by (metis span-clauses(1))}
\end{lemma}

\begin{lemma}
\text{span-0: 0 } \in \text{span } S
\text{by (metis span-clauses(2))}
\end{lemma}

\begin{lemma}
\text{span-inc: S } \subseteq \text{span } S
\text{by (metis subset-eq span-superset)}
\end{lemma}

\begin{lemma}
\text{dependent-0:}
\text{assumes 0 } \in A
\text{shows dependent A}
\text{unfolding dependent-def}
\text{apply (rule-tac } x=0 \text{ in bexI)}
\text{using assms span-0}
\text{apply auto}
\text{done}
\end{lemma}
lemma span-add: \( x \in \text{span } S \implies y \in \text{span } S \implies x + y \in \text{span } S \)
by (metis subspace-add subspace-span)

lemma span-mul: \( x \in \text{span } S \implies \text{scale } c x \in \text{span } S \)
by (metis span-clauses(4))

lemma span-neg: \( x \in \text{span } S \implies -x \in \text{span } S \)
by (metis subspace-neg subspace-span)

lemma span-sub: \( x \in \text{span } S \implies y \in \text{span } S \implies x - y \in \text{span } S \)
by (metis subspace-span subspace-sub)

lemma span-setsum: finite A \(\implies\) \( \forall x \in A. f x \in \text{span } S \implies \) \( \text{setsum } f A \in \text{span } S \)
by (rule subspace-setsum, rule subspace-span)

lemma span-add-eq: \( x \in \text{span } S \implies x + y \in \text{span } S \implies -y \in \text{span } S \)
apply (auto simp only: span-add span-sub)
apply (subgoal-tac (x + y) - x \in \text{span } S)
apply simp
apply (simp only: span-add span-sub)
done

lemma span-linear-image:
  assumes lf: linear scale scaleC \((f :: 'b::ab-group-add)\)
  shows vector-space.span scaleC \((f ' S)\) = \(f ' (\text{span } S)\)
proof
  interpret B: vector-space scale using lf by (metis linear-iff)
  interpret C: vector-space scaleC using lf by (metis linear-iff)
  interpret lf: linear scale scaleC f using lf by simp
  show ?thesis
  proof (rule C.span-unique)
    show \(f ' S \subseteq f ' (\text{span } S)\)
    by (rule image mono, rule span inc)
    show vector-space.subspace scaleC \((f ' S)\)
    using lf subspace-span by (rule subspace-linear-image)
  next
  fix T
  assume f '! S \subseteq T and vector-space.subspace scaleC T
  then show f '! \text{span } S \subseteq T
    unfolding image subset iff subset vimage
    by (metis subspace-linear vimage lf span minimal)
  qed
qed

lemma span-union: \( \text{span } (A \cup B) = (\lambda (a, b). a + b) ' (\text{span } A \times \text{span } B) \)
proof (rule span unique)
  show \(A \cup B \subseteq (\lambda (a, b). a + b) ' (\text{span } A \times \text{span } B)\)
by safe (force intro: span-clauses)+

**next**

**have** linear \((\lambda x \ (a, b), \ (\text{scale } x \ a, \ \text{scale } x \ b)) \text{ scale } (\lambda(a, b). \ a + b)\)

**proof** (unfold linear-def linear-axioms-def, auto)

**show** vector-space \((\lambda x \ (a, b), \ (\text{scale } x \ a, \ \text{scale } x \ b))\) using vector-space-product .

**show** vector-space scale by (unfold-locales)

next

**show** \(\land r \ a \ b. \ \text{scale } r \ a + \text{scale } r \ b = \text{scale } r \ (a + b)\) by (metis scale-right-distrib)

**qed**

moreover **have** vector-space.subspace \((\lambda x \ (a, b), \ (\text{scale } x \ a, \ \text{scale } x \ b))\) \((\text{span } A \times \text{span } B)\)

by (intro subspace-Times subspace-span)

ultimately **show** subspace \(((\lambda(a, b). \ a + b) \circ \text{span } A \times \text{span } B)\)

by (metis (lifting) linear-iff vector-space.subspace-linear-image)

**next**

**fix** \(T\)

**assume** \(A \cup B \subseteq T\) and **subspace** \(T\)

then **show** \((\lambda(a, b). \ a + b) \circ \text{span } A \times \text{span } B) \subseteq T\)

by (auto intro!: subspace-add elim: span-induct)

**qed**

**lemma** span-singleton: \(\text{span } \{x\} = \text{range } (\lambda k. \ \text{scale } k \ x)\)

**proof** (rule span-unique)

**show** \(\{x\} \subseteq \text{range } (\lambda k. \ \text{scale } k \ x)\)

by (fast intro: scale-one [symmetric])

**show** subspace \((\text{range } (\lambda k. \ \text{scale } k \ x))\)

unfolding subspace-def

by (auto intro: scale-left-distrib [symmetric])

**next**

**fix** \(T\)

**assume** \(\{x\} \subseteq T\) and **subspace** \(T\)

then **show** \(\text{range } (\lambda k. \ \text{scale } k \ x) \subseteq T\)

unfolding subspace-def by auto

**qed**

**lemma** span-insert: \(\text{span } (\text{insert } a \ S) = \{x. \ \exists k. (x - \text{scale } k \ a) \in \text{span } S\}\)

**proof** –

**have** \(\text{span } (\{a\} \cup S) = \{x. \ \exists k. (x - \text{scale } k \ a) \in \text{span } S\}\)

unfolding span-union span-singleton

apply safe

apply (rule-tac \(x=k\) in exI, simp)

apply (erule rev-image-eqI [OF SigmaI [OF rangeI]])

apply auto

done

then **show** \(\text{thesis by simp}\)

**qed**

12
lemma span-breakdown:
assumes bS: b ∈ S
and aS: a ∈ span S
shows ∃ k. a − scale k b ∈ span (S − {b})
using assms span-insert [of b S − {b}]
by (simp add: insert-absorb)

lemma span-breakdown-eq: x ∈ span (insert a S) ←→ (∃ k. x − scale k a ∈ span S)
by (simp add: span-insert)

lemma in-span-insert:
assumes a: a ∈ span (insert b S)
and na: a /∈ span S
shows b ∈ span (insert a S)

proof –
from span-breakdown[of b insert b S a, OF insertI1 a]
obtain k where k: a − scale k b ∈ span (S − {b}) by auto
show ?thesis
proof (cases k = 0)
  case True
  with k have a ∈ span S
  apply (simp)
  apply (rule set-rev-mp)
  apply assumption
  apply (rule span-mono)
  apply blast
  done
  with na show ?thesis by blast
next
  case False
  have eq: b = scale (1/k) a − (scale (1/k) a − b) by simp
  from False have eq': scale (1/k) (a − scale k b) = scale (1/k) a − b
  by (simp add: algebra-simps)
  from k have scale (1/k) (a − scale k b) ∈ span (S − {b})
  by (rule span-mul)
  then have th: scale (1/k) a − b ∈ span (S − {b})
    unfolding eq'.
  from k show ?thesis
    apply (subst eq)
    apply (rule span-sub)
    apply (rule span-mul)
    apply (rule span-superset)
    apply blast
    apply (rule set-rev-mp)
    apply (rule th)
    apply (rule span-mono)
    using na
apply blast
done
qed

lemma in-span-delete:
assumes a: a ∈ span S
and na: a /∈ span (S − {b})
shows b ∈ span (insert a (S − {b}))
apply (rule in-span-insert)
apply (rule set-rev-mp)
apply (rule a)
apply (rule span-mono)
apply blast
apply (rule na)
done

lemma span-redundant; x ∈ span S ⇒ span (insert x S) = span S
unfolding span-def by (rule hull-redundant)

lemma span-trans:
assumes x: x ∈ span S
and y: y ∈ span (insert x S)
shows y ∈ span S
using assms by (simp only: span-redundant)

lemma span-insert-0[simp]: span (insert 0 S) = span S
by (metis span-0 span-redundant)

lemma span-explicit:
span P = {y. ∃S u. finite S ∧ S ⊆ P ∧ setsum (λv. scale (u v) v) S = y}
(is - = ?E is - = {y. ?h y} is - = {y. ∃S u. ?Q S u y})
proof –
{ 
  fix x
  assume x: x ∈ ?E
  then obtain S u where fS: finite S and SP: S ⊆ P and u: setsum (λv. scale (u v) v) S = x
  by blast
  have x ∈ span P
    unfolding u[symmetric]
    apply (rule span-setsum[OF fS])
    using span-mono[OF SP]
    apply (auto intro: span-superset span-mult)
done
}
moreover

have ∀x ∈ span P, x ∈ ?E

proof (rule span-induct-alt')

show 0 ∈ Collect ?h

unfolding mem-Collect-eq

apply (rule exI[where x={}])

apply simp

done

next

fix c x y

assume x: x ∈ P

assume hy: y ∈ Collect ?h

from hy obtain S u where fS: finite S and SP: S ⊆ P

and u: setsum (λv. scale (u v) v) S = y by blast

let ?S = insert x S

let ?u = λy. if y = x then (if x ∈ S then u y + c else c) else u y

from fS SP x have th0: finite (insert x S) insert x S ⊆ P

by blast+

have ?Q ?S ?u (scale c x + y)

proof cases

assume xS: x ∈ S

have S1: S = (S − {x}) ∪ {x}

and Sss: finite (S − {x}) finite {x} (S − {x}) ∩ {x} = {}

using xS fS by auto

have setsum (λv. scale (?u v) v) ?S = (∑v∈S − {x}. scale (?u v) v) + scale (u x + c) x

using xS by (simp add: setsum.remove [OF fS xS] insert-absorb)

also have ... = (∑v∈S. scale (u v) v) + scale c x

by (simp add: setsum.remove [OF fS xS] algebra-simps)

also have ... = scale c x + y

by (simp add: add.commute u)

finally have setsum (λv. scale (?u v) v) ?S = scale c x + y.

then show ?thesis using th0 by blast

next

assume xS: x ∉ S

have th00: (∑v∈S. scale (if v = x then c else u v) v) = y

unfolding u[symmetric]

apply (rule setsum.cong)

using xS

apply auto

done

show ?thesis using fS xS th0

by (simp add: th00 setsum-clauses add.commute cong del: if-weak-cong)

qed

then show (scale c x + y) ∈ Collect ?h

unfolding mem-Collect-eq

apply –

apply (rule exI[where x=?S])

apply (rule exI[where x=?u])
apply metis
done
qed
ultimately show \textit{thesis} by blast
qed
unfolding scale-setsum-right[symmetric] u using uv by simp
finally have setsum (λv. scale (u v) v) S = a .
with th0 have ?lhs
  unfolding dependent-def span-explicit
  apply –
  apply (rule bexI[where x = a])
  apply (simp-all del: scale-minus-left)
  apply (rule exI[where x = S])
  apply (auto simp del: scale-minus-left)
done
}
ultimately show ?thesis by blast
qed

lemma span-finite:
  assumes fS: finite S
  shows span S = {y. ∃u. setsum (λv. scale (u v) v) S = y}
(is - = ?rhs)
proof –
  { fix y
    assume y: y ∈ span S
    from y obtain S' u where fS': finite S'
      and SS': S' ⊆ S
      and u: setsum (λv. scale (u v) v) S' = y
      unfolding span-explicit by blast
    let ?u = λx. if x ∈ S' then u x else 0
    have setsum (λv. scale (?u v) v) S' = setsum (λv. scale (u v) v) S'
      using SS' fS by (auto intro!: setsum.mono-neutral-cong-right)
    then have setsum (λv. scale (?u v) v) S = y by (metis u)
    then have y ∈ ?rhs by auto
  }
moreover
  { fix y u
    assume u: setsum (λv. scale (u v) v) S = y
    then have y ∈ span S using fS unfolding span-explicit by auto
  }
ultimately show ?thesis by blast
qed

lemma independent-insert:
independent (insert a S) ←→
  (if a ∈ S then independent S else independent S ∧ a ∉ span S)
(is ?lhs ←→ ?rhs)
proof (cases a ∈ S)
case True
then show \( \text{thesis} \)
  using insert-absorb[OF True] by simp

next
case False
show \( \text{thesis} \)
proof
  assume \( i : \text{lhs} \)
  then show \( \text{rhs} \)
    using False
    apply simp
    apply (rule conjI)
    apply (rule independent-mono)
    apply assumption
    apply blast
    apply (simp add: dependent-def)
  done
next
  assume \( i : \text{rhs} \)
  show \( \text{lhs} \)
    using \( i \) False
    apply simp
    apply (auto simp add: dependent-def)
    apply (case_tac \( aa = a \))
    apply auto
    apply (subgoal_tac \( a \in \text{span} (\text{insert} \( aa \) \( \text{S} - \{aa\} \)))
    apply simp
    apply (subgoal_tac \( a \in \text{span} \( \text{insert} \( aa \) \( \text{S} - \{aa\} \)))
    apply (subgoal_tac \( \text{insert} \( aa \) \( \text{S} - \{aa\} \)) = \( \text{S} \))
    apply simp
    apply blast
    apply (rule in-span-insert)
    apply assumption
    apply blast
    apply blast
  done
qed

lemma spanning-subset-independent:
assumes BA: \( B \subseteq A \)
  and iA: independent \( A \)
  and AsB: \( A \subseteq \text{span} B \)
shows \( A = B \)
proof
  show \( B \subseteq A \) by (rule BA)
from span-mono[OF BA] span-mono[OF AsB]
have \( s_{AB} \): \( \text{span} \; A = \text{span} \; B \) unfolding \( \text{span-span} \) by blast

\[
\begin{align*}
\{ & \\
& \text{fix } x \\
& \text{assume } x: x \in A \\
& \text{from } iA \text{ have th0: } x \notin \text{span } (A - \{x\}) \\
& \quad \text{unfolding dependent-def using } x \text{ by blast} \\
& \text{from } x \text{ have } xsA: x \in \text{span } A \\
& \quad \text{by } (\text{blast intro: span-superset}) \\
& \text{have } A - \{x\} \subseteq A \text{ by blast} \\
& \text{then have th1: span } (A - \{x\}) \subseteq \text{span } A \\
& \quad \text{by } (\text{metis span-mono}) \\
& \} \\
& \{ \\
& \quad \text{assume } xB: x \notin B \\
& \quad \text{from } xB \; BA \text{ have } B \subseteq A - \{x\} \\
& \quad \text{by blast} \\
& \quad \text{then have span } B \subseteq \text{span } (A - \{x\}) \\
& \quad \quad \text{by } (\text{metis span-mono}) \\
& \quad \quad \text{with th1 th0 } s_{AB} \text{ have } x \notin \text{span } A \\
& \quad \quad \text{by blast} \\
& \quad \quad \text{with } x \text{ have False} \\
& \quad \quad \quad \text{by } (\text{metis span-superset}) \\
& \} \\
& \text{then have } x \in B \text{ by blast} \\
& \} \\
& \text{then show } A \subseteq B \text{ by blast} \\
\text{qed}
\]

lemma exchange-lemma:

\[
\begin{align*}
\text{assumes } f: & \text{finite } t \\
& \text{and } i: \text{independent } s \\
& \text{and } sp: s \subseteq \text{span } t \\
\text{shows } & \exists t', \text{card } t' = \text{card } t \land \text{finite } t' \\
& \land s \subseteq t' \land t' \subseteq s \cup t \land s \subseteq \text{span } t' \\
\text{using } & f \; i \; sp \\
\text{proof } (\text{induct } \text{card } (t - s) \text{ arbitrary: } s \; t \text{ rule: less-induct}) \\
\text{case } & \text{less} \\
\text{note } & ft = (\text{finite } t) \text{ and } s = (\text{independent } s) \text{ and } sp = (s \subseteq \text{span } t) \\
\text{let } & ?P = \lambda t'. \text{card } t' = \text{card } t \land \text{finite } t' \land s \subseteq t' \land t' \subseteq s \cup t \land s \subseteq \text{span } t' \\
\text{let } & ?ths = \exists t'. \; ?P \; t' \\
& \{ \\
& \quad \text{assume } st: s \subseteq t \\
& \quad \text{from } st \; ft \text{ span-mono}[\text{OF } st] \\
& \quad \text{have } ?ths \\
& \quad \quad \text{apply -} \\
& \quad \quad \text{apply } (\text{rule exI}[\text{where } x=t]) \\
& \quad \quad \text{apply } (\text{auto intro: span-superset}) \\
& \quad \text{done} \\
& \} \\
\text{moreover} \\
\end{align*}
\]
{ 
  assume st: t ⊆ s
  from spanning-subset-independent[OF st s sp] st ft span-mono[OF st]
  have ?ths
    apply –
    apply (rule exI[where x=t])
    apply (auto intro: span-superset)
  done
}

moreover
{ 
  assume st: ¬ s ⊆ t − t ⊆ s
  from st(2) obtain b where b: b ∈ t b /∈ s
    by blast
  from b have t − {b} − s ⊂ t − s
    by blast
  then have cardlt: card (t − {b} − s) < card (t − s)
    using ft by (auto intro: psubset-card-mono)
  from b ft have ct0: card t ≠ 0
    by auto
  have ?ths
  proof cases
    assume stb: s ⊆ span (t − {b})
    from ft have ftb: finite (t − {b})
      by auto
    from less(1)[OF cardlt ftb s stb]
    obtain u where u: card u = card (t − {b}) s ⊆ u u ⊆ s ∪ (t − {b}) s ⊆ span u
      and fu: finite u by blast
    let ?w = insert b u
    have th0: s ⊆ insert b u
      using u by blast
    from u(3) b have u ⊆ s ∪ t
      by blast
    then have th1: insert b u ⊆ s ∪ t
      using u b by blast
    have bu: b /∈ u
      using b u by blast
    from u(1) ft b have card u = (card t − 1)
      by auto
    then have th2: card (insert b u) = card t
      using card-insert-disjoint[OF fu bu] ct0 by auto
    from u(4) have s ⊆ span u .
    also have ... ⊆ span (insert b u)
      by (rule span-mono) blast
    finally have th3: s ⊆ span (insert b u) .
    from th0 th1 th2 th3 fu have th: ?P ?w
      by blast
  from th show ?thesis by blast
}
next
  assume stb: ¬ s ⊆ span (t − {b})
  from stb obtain a where: a ∈ s a /∈ span (t − {b})
      by blast
  have ab: a ≠ b
      using a b by blast
  have at: a /∈ t
      using a ab span-superset[af a t− {b}] by auto
  have mlt: card ((insert a (t − {b}))) − s < card (t − s)
      using cardlt ft a b by auto
  have ft': finite (insert a (t − {b}))
      using ft by auto
  { fix x
    assume xS: x ∈ s
    have t: t ⊆ insert b (insert a (t − {b}))
        using b by auto
    from b(1) have b ∈ span t
        by (simp add: span-superset)
    have bs: b ∈ span (insert a (t − {b}))
        apply (rule in-span-delete)
        using a sp unfolding subset-eq
        apply auto
        done
    from xS sp have x ∈ span t
        by blast
    with span-mono[OF t] have x: x ∈ span (insert b (insert a (t − {b}))) ..
    from span-trans[OF bs x] have x ∈ span (insert a (t − {b})).
  }
  then have sp': s ⊆ span (insert a (t − {b}))
    by blast
  from less(1)(OF mlt ft' sp') obtain u where: u:
    card u = card (insert a (t − {b}))
    finite u s ⊆ u u ⊆ s ∪ insert a (t − {b})
    s ⊆ span u by blast
  from u a b ft at ct0 have ?P u
    by auto
  then show ?thesis by blast
qed

ultimately show ?ths by blast
qed

lemma independent-span-bound:
  assumes f: finite t
       and i: independent s
       and sp: s ⊆ span t
  shows finite s ∧ card s ≤ card t
  by (metis exchange-lemma[OF f i sp] finite-subset card-mono)
lemma independent-explicit:

\[ \text{independent } A = \]
\[ (\forall S \subseteq A. \text{finite } S \rightarrow (\forall a. (\sum_{v \in S.} (u \cdot v)) = 0 \rightarrow (\forall v \in S. u \cdot v = 0))) \]

unfolding dependent-explicit [of A] by (simp add: disj-not2)

A finite set \( A \) for which every of its linear combinations equal to zero requires every coefficient being zero, is independent:

lemma independent-if-scalars-zero:

assumes \( \text{fin-} A \): finite \( A \)
and \( \text{sum: } \forall f. (\sum x \in A. \text{scale} (f \cdot x)) = 0 \rightarrow (\forall x \in A. f \cdot x = 0) \)

shows \( \text{independent } A \)

proof (unfold independent-explicit, clarify)

fix \( S \) and \( a :: \ 'b \Rightarrow 'a \)
assume \( S \subseteq A \) and \( v : v \in S \)
let \( \lambda g. \forall x. \text{if } x \in S \text{ then } u \cdot x \text{ else } 0 \)
have \((\sum_{v \in A.} \text{scale} (\lambda g \cdot v)) = (\sum_{v \in S.} \text{scale} (u \cdot v))\)
using \( S \text{ fin-} A \) by (auto intro!: setsum.mono_neutral_cong_right)
also have \((\sum_{v \in S.} \text{scale} (u \cdot v)) = 0\)
finally have \( \lambda g \cdot v = 0 \) using \( v \cdot S \) sum by force
thus \( u \cdot v = 0 \) unfolding if-P[OF v].
qed

definition cart-basis = \{ axis i 1 | i. i \in UNIV \}

lemma finite-cart-basis: finite (cart-basis) unfolding cart-basis-def
using finite-Atleast-Atmost-nat by fastforce

lemma independent-cart-basis:

vec.independent \( \text{(cart-basis)} \)

proof (rule vec.independent-if-scalars-zero, auto)

show \( \text{finite } \) (cart-basis) using \( \text{finite-cart-basis} \).

fix \( f ::('a, 'b) \text{ vec} \Rightarrow 'a \) and \( x ::('a, 'b) \text{ vec} \)
assume \( \text{eq-} 0 \): \((\sum x \in \text{cart-basis}. f \cdot x \cdot s \cdot x) = 0 \) and \( \text{x-in}: x \in \text{cart-basis} \)

obtain \( i \) where \( x = \text{axis i 1} \) using \( \text{x-in} \) unfolding cart-basis-def by auto

have \( \text{setsum-} \text{eq-} 0 \): \((\sum x \in (\text{cart-basis} - \{ x \}). f \cdot x \cdot ($ x \cdot i ) = 0 \)
proof (rule setsum.neutral, rule ballI)
fix \( x a \) assume \( x a : x a \in \text{cart-basis} - \{ x \} \)

obtain \( a \) where \( a : x a = \text{axis i 1} \) and \( \text{a-not-i}: a \neq i \)
using \( x a \) unfolding cart-basis-def by auto

have \( x a \cdot \$ \cdot i = 0 \) unfolding \( \text{a-axis-def} \) using \( \text{a-not-i} \) by auto
thus \( f \cdot x a \cdot \$ \cdot i = 0 \) by simp
qed

have \( 0 = (\sum x \in \text{cart-basis}. f \cdot x \cdot s \cdot x) \cdot \$ \cdot i \) using \text{eq-0} by simp
also have \( ... = (\sum x \in \text{cart-basis}. (f \cdot x \cdot s \cdot x) \cdot \$ \cdot i) \) unfolding setsum-component ..
also have \( ... = (\sum x \in \text{cart-basis}. f \cdot x \cdot ($ x \cdot i )) \) unfolding vector-smult-component
also have ... = \( f \cdot x \star (x \star i) + (\sum x \in (\text{cart-basis}) - \{x\}) \cdot f \cdot x \star (x \star i) \)
by [rule setsum.remove[OF finite-cart-basis x-in]]
also have ... = \( f \cdot x \star (x \star i) \) unfolding setsum-eq-0 by simp
also have ... = \( f \cdot x \) unfolding \( x \cdot \text{axis-def} \) by auto
finally show \( f \cdot x = 0 \) ..

qed

lemma \text{span-cart-basis}:
\text{vec.span (cart-basis)} = \text{UNIV}

proof (auto)

fix \( x \cdot (a, b) \) \text{vec}

let \( \lambda v \cdot x \star (\text{THE } i. \ v = \text{axis } i \ 1) \)

show \( x \in \text{vec.span (cart-basis)} \)

proof (unfold \text{vec.span-finite[OF finite-cart-basis], auto, rule exI[of - ?f] , subst (2) \text{vec-eq-iff}], clarify)

fix \( i' : \text{'}b \)

let \( ?w = \text{axis } i \ (1 : \text{'}a) \)

have the-eq-i: \( (\text{THE } a. \ ?w = \text{axis } a \ 1) = i \)
by (rule the-equality, auto simp: axis-eq-axis)

have setsum-eq-0: \( (\sum v \in (\text{cart-basis}) - \{?w\}, x \star (\text{THE } i. \ v = \text{axis } i \ 1) \star v \star i) \)
= \( 0 \)

proof (rule setsum.neutral, rule ballI)

fix \( xa : (a, b) \) \text{vec}

assume \( xa: xa \in \text{cart-basis} - \{?w\} \)

obtain \( j \) where \( j: xa = \text{axis } j \ 1 \) and \( i \neq j \) using \( xa \) unfolding \text{cart-basis-def} by auto

have the-eq-j: \( (\text{THE } i. \ xa = \text{axis } i \ 1) = j \)

proof (rule the-equality)

show \( xa = \text{axis } j \ 1 \) using \( j \).

show \( \lambda i. \ xa = \text{axis } i \ 1 \implies i = j \) by (metis axis-eq-axis j zero-neq-one)

qed

show \( x \star (\text{THE } i. \ xa = \text{axis } i \ 1) \star xa \star i = 0 \)

apply (subst (2) \( j \))

unfolding the-eq-j unfolding \text{axis-def} using i-not-j by simp

qed

have \( (\sum v \in \text{cart-basis} \cdot x \star (\text{THE } i. \ v = \text{axis } i \ 1) \star s v) \star i = \)
(\( \sum v \in \text{cart-basis} \cdot (x \star (\text{THE } i. \ v = \text{axis } i \ 1) \star s v) \star i \)) unfolding setsum-component

.. also have ... = \( (\sum v \in \text{cart-basis} \cdot x \star (\text{THE } i. \ v = \text{axis } i \ 1) \star v \star i) \)

unfolding \text{vector-smult-component} ..

also have ... = \( x \star (\text{THE } a. \ ?w = \text{axis } a \ 1) \star ?w \star i + (\sum v \in (\text{cart-basis}) - \{?w\} \cdot x \star (\text{THE } i. \ v = \text{axis } i \ 1) \star v \star i) \)
by (rule setsum.remove[OF finite-cart-basis], auto simp add: \text{cart-basis-def})

also have ... = \( x \star (\text{THE } a. \ ?w = \text{axis } a \ 1) \star ?w \star i \) unfolding setsum-eq-0 by simp

also have ... = \( x \star i \) unfolding the-eq-i unfolding \text{axis-def} by auto

finally show \( (\sum v \in \text{cart-basis} \cdot x \star (\text{THE } i. \ v = \text{axis } i \ 1) \star s v) \star i = x \star i \).

qed

23
qed

locale finite-dimensional-vector-space = vector-space +
  fixes Basis :: 'b set
  assumes finite-Basis: finite (Basis)
  and independent-Basis: independent (Basis)
  and span-Basis: span (Basis) = UNIV
begin

definition dimension :: nat where
dimension ≡ card (Basis :: 'b set)

lemma independent-bound:
  shows independent S =⇒ finite S ∧ card S ≤ dimension
  using independent-span-bound[OF finite-Basis, of S]
  unfolding dimension-def span-Basis by auto

lemma maximal-independent-subset-extend:
  assumes sv: S ⊆ V
  and iS: independent S
  shows ∃ B. S ⊆ B ∧ B ⊆ V ∧ independent B ∧ V ⊆ span B
  using sv iS
proof (induct dimension − card S arbitrary: S rule: less-induct)
case less
  note sv = (S ⊆ V) and i = (independent S)
  let ?P = λB. S ⊆ B ∧ B ⊆ V ∧ independent B ∧ V ⊆ span B
  let ?ths = ∃ x. ?P x
  let ?d = dimension
  show ?ths
proof (cases V ⊆ span S)
  case True
  then show ?thesis
    using sv i by blast
next
case False
  then obtain a where a: a ∈ V a ∉ span S
    by blast
  from a have aS: a ∉ S
    by (auto simp add: span-superset)
  have th0: insert a S ⊆ V
    using a sv by blast
  from independent-insert[of a S] i a
  have th1: independent (insert a S)
    by auto
  have nlt: ?d − card (insert a S) < ?d − card S
    using aS a independent-bound[OF th1] by auto

24
from less(1)[OF mlt th0 th1]
obtain B where B: insert a S ⊆ B B ⊆ V independent B V ⊆ span B
by blast
from B have ?P B by auto
then show ?thesis by blast
qed

lemma maximal-independent-subset:
∃ B. B ⊆ V ∧ independent B ∧ V ⊆ span B
by (metis maximal-independent-subset-extend[of {}]
empty-subsetI independent-empty)
end

context vector-space
begin
definition dim V = (SOME n. ∃ B. B ⊆ V ∧ independent B ∧ V ⊆ span B ∧
card B = n)
end

context finite-dimensional-vector-space
begin
lemma basis-exists:
∃ B. B ⊆ V ∧ independent B ∧ V ⊆ span B ∧ (card B = dim V)
unfolding dim-def some-eq-ex[of λn. ∃ B. B ⊆ V ∧ independent B ∧ V ⊆ span B ∧ (card B = n)]
using maximal-independent-subset[of V] independent-bound
by auto

lemma independent-card-le-dim:
assumes B ⊆ V
and independent B
shows card B ≤ dim V
proof –
from basis-exists[of V] ⟨B ⊆ V⟩
obtain B' where independent B'
and B ⊆ span B'
and card B' = dim V
by blast
with independent-span-bound[OF - ⟨independent B⟩ ⟨B ⊆ span B'⟩] independent-bound[of B']
show ?thesis by auto
qed

lemma span-card-ge-dim:
shows B ⊆ V ⇒ V ⊆ span B ⇒ finite B ⇒ dim V ≤ card B
by (metis basis-exists[of V] independent-span-bound subset-trans)

25
lemma basis-card-eq-dim:
  shows $B \subseteq V \implies V \subseteq \text{span } B \implies \text{independent } B \implies \text{finite } B \land \text{card } B = \text{dim } V$
  by (metis order-iff independent-card-le-dim span-card-ge-dim independent-bound)

lemma dim-unique:
  shows $B \subseteq V \implies V \subseteq \text{span } B \implies \text{independent } B \implies \text{card } B = n \implies \text{dim } V = n$
  by (metis basis-card-eq-dim)

lemma dim-UNIV:
  shows dim UNIV = car(\text{Basis})
  by (metis basis-card-eq-dim independent-Basis span-Basis top-greatest)

lemma dim-subset:
  shows $S \subseteq T \implies \text{dim } S \leq \text{dim } T$
  using basis-exists[of T] basis-exists[of S]
  by (metis independent-card-le-dim subset-trans)

lemma dim-univ-eq-dimension:
  shows dim UNIV = dimension
  by (metis basis-card-eq-dim dimension-def independent-Basis span-Basis top-greatest)

lemma dim-subset-UNIV:
  shows dim S \leq \text{dimension}
  by (metis dimension-def dim-subset subset-UNIV dim-UNIV)

lemma card-ge-dim-independent:
  assumes BV: $B \subseteq V$
  and iB: \text{independent } B
  and dVB: \text{dim } V \leq \text{card } B
  shows $V \subseteq \text{span } B$

proof
  fix a
  assume aV: $a \in V$
  { assume aB: $a \notin \text{span } B$
    then have iaB: \text{independent } (\text{insert } a B)
      using iB aV BV by (simp add: independent-insert)
    from aV BV have th0: $\text{insert } a B \subseteq V$
      by blast
    from aB have a \notin B
      by (auto simp add: span-superset)
      with independent-card-le-dim[OF th0 iaB] dVB independent-bound[OF iB]
      have False by auto
  }
  then show $a \in \text{span } B$ by blast
qed

26
lemma card-le-dim-spanning:
assumes BV: B ⊆ V
  and VB: V ⊆ span B
  and fB: finite B
  and dVB: dim V ≥ card B
shows independent B
proof –
{  
  fix a
  assume a: a ∈ B a ∈ span (B − {a})
  from a fB have c0: card B ≠ 0
    by auto
  from a fB have cb: card (B − {a}) = card B − 1
    by auto
  from BV a have th0: B − {a} ⊆ V
    by blast
  }
  {  
    fix x
    assume x: x ∈ V
    from a have eq: insert a (B − {a}) = B
      by blast
    from x VB have x': x ∈ span B
      by blast
    from span-trans[OF a(2), unfolded eq, OF x']
    have x ∈ span (B − {a}) .
  }
  then have th1: V ⊆ span (B − {a})
    by blast
  have th2: finite (B − {a})
    using fB by auto
  from span-card-ge-dim[OF th0 th1 th2]
  have c: dim V ≤ card (B − {a}) .
  from c c0 dVB cb have False by simp
  }
then show ?thesis
  unfolding dependent-def by blast
qed

lemma card-eq-dim:
shows B ⊆ V ⇒ card B = dim V ⇒ finite B ⇒ independent B ↔ V ⊆ span B
by (metis order-eq-iff card-le-dim-spanning card-ge-dim-independent)

lemma independent-bound-general:
shows independent S ==⇒ finite S ∧ card S ≤ dim S
by (metis independent-card-le-dim independent-bound subset-refl)

lemma dim-span:
shows \( \dim (\text{span } S) = \dim S \)

proof –

have th0: \( \dim S \leq \dim (\text{span } S) \)
  by (auto simp add: subset-eq intro: dim-subset span-superset)
from basis-exists[of S]
obtain B where B: B \subseteq S independent B S \subseteq \text{span } B card B = \dim S
  by blast
from B have fB: finite B card B = \dim S
  using independent-bound by blast
have bSS: B \subseteq \text{span } S
  using B(1) by (metis subset-eq span-inc)
have sssB: \text{span } S \subseteq \text{span } B
  using span-mono[OF B(3)] by (simp add: span-span)
from span-card-ge-dim[OF bSS sssB fB(1)] th0 show ?thesis
  using fB(2) by arith
qed

lemma subset-le-dim:
  shows S \subseteq \text{span } T \Longrightarrow \dim S \leq \dim T
  by (metis dim-span dim-subset)

lemma span-eq-dim:
  shows \text{span } S = \text{span } T \Longrightarrow \dim S = \dim T
  by (metis dim-span)
end

context linear
begin

lemma independent-injective-image:
  assumes iS: B. independent S
  and fi: inj f
  shows C. independent (f ' S)
proof –
  have l: linear scaleB scaleC f by unfold-locales
  {  
    fix a
    assume a: a \in S \exists f a \in C.\text{span } (f ' S - \{f a\})
    have eq: f ' S - \{f a\} = f ' (S - \{a\})
      using fi by (auto simp add: inj-on-def)
    from a have f a \in f ' B.\text{span } (S - \{a\})
      unfolding eq B.span-linear-image[OF l, of S - \{a\}] by blast
    then have a \in B.\text{span } (S - \{a\})
      using fi by (auto simp add: inj-on-def)
    with a(1) iS have False
      by (simp add: B.dependent-def)
  }  
then show ?thesis
  unfolding dependent-def by blast
locale two-vector-spaces-over-same-field = B: vector-space scaleB + C: vector-space scaleC
  for scaleB :: (′a::field => ′b::ab-group-add => ′b) (infixr †b 75)
  and scaleC :: (′a => ′c::ab-group-add => ′c) (infixr †c 75)

context two-vector-spaces-over-same-field
begin

lemma linear-indep-image-lemma:
  assumes lf: linear (op †b) (op †c) f
  and fB: finite B
  and ifB: C.independent (f † B)
  and fi: inj-on f B
  and xsB: x ∈ B.span B
  and fx: f x = 0
  shows x = 0
  using fB ifB xsB fx
proof (induct arbitrary: x rule: finite-induct[OF fB])
  case 1
  then show ?case by auto
next
  case (2 a b x)
  have fb: finite b using 2.prems by simp
  have th0: f † b ⊆ f † (insert a b)
    apply (rule image mono)
    apply blast
  done
  from independent mono[ OF 2.prems(2) th0]
  have ifb: independent (f † b).
  have fib: inj-on f b
    apply (rule subset-inj-on [OF 2.prems(3)])
    apply blast
  done
  from B.span-breakdown[of a insert a b, simplified, OF 2.prems(4)]
  obtain k where k: x − k † b a ∈ B.span (b − {a})
    by blast
  have f (x − k † b a) ∈ C.span (f † b)
    unfolding B.span-linear-image[OF lf]
    apply (rule imageI)
    using k B.span-mono[of b − {a} b]
    apply blast
  done
  then have f x − k † c f a ∈ C.span (f † b)
    by (metis (full-types) lf linear.linear-cmul linear.linear-sub)

qed
end
then have \( -k \cdot c \cdot f \cdot a \in C.\text{span} \ (f \cdot b) \)
using 2.prems(5) by simp
have xsb: \( x \in B.\text{span} \ b \)

proof (cases \( k = 0 \))
  case True
  with \( k \) have \( x \in B.\text{span} \ (b - \{a\}) \) by simp
  then show \( \text{thesis} \) using B.span-mono[of \( b - \{a\} \) \( b \)]
    by blast
next
  case False
  with \( \text{span-mul} \ [\text{OF th}, \text{of} \ -1/k] \)
  have \( \text{th1}: f \cdot a \in \text{span} \ (f \cdot b) \)
    by auto
  from inj-on-image-set-diff[\text{OF 2.prems(3), of insert} \ a \ b \ \{a\}, \text{symmetric}]
  have \( \text{tha}: f \cdot \text{insert} \ a \ b - f \cdot \{a\} = f \cdot (\text{insert} \ a \ b - \{a\}) \) by blast
  from 2.prems(2) [\text{unfolded dependent-def bex-simps(8), rule-format, of f a}]
  have \( f \cdot a \notin \text{span} \ (f \cdot b) \) using tha
    using 2.hyps(2)
    2.prems(3) by auto
  with \( \text{th1} \) have \( \text{False} \) by blast
  then show \( \text{thesis} \) by blast
qed

lemma linear-independent-extend-lemma:
fixes \( f :: \ 'b \Rightarrow \ 'c \)
assumes fi: finite \( B \)
and ib: \( B.\text{independent} \ B \)
shows \( \exists \ g. \ 
(\forall x \in B.\text{span} \ B. \forall y \in B.\text{span} \ B. \ g \ (x + y) = g \cdot x + g \cdot y) \land
(\forall x \in B.\text{span} \ B. \forall c. \ g \ (c \cdot b \ x) = c \cdot (g \cdot x)) \land
(\forall x \in B. \ g \cdot x = f \cdot x) \)
using ib fi
proof (induct rule: finite-induct[\text{OF fi}])
  case 1
  then show \( \text{?case} \) by auto
next
  case (2 \ a \ b)
  from 2.prems 2.hyps have ibf: \( B.\text{independent} \ b \ \text{finite} \ b \)
    by (simp-all add: B.independent-insert)
  from 2.hyps(3)[\text{OF ibf}] obtain \( g \) where
    \( g: \forall x \in B.\text{span} \ b. \forall y \in B.\text{span} \ b. \ g \ (x + y) = g \cdot x + g \cdot y \)
    \( \forall x \in B.\text{span} \ b. \forall c. \ g \ (c \cdot b \ x) = c \cdot (g \cdot x) \forall x \in b. \ g \cdot x = f \cdot x \) by blast
  let \( ?h = \lambda z. \ \text{SOME} \ k. \ (z - k \cdot b \ a) \in B.\text{span} \ b \)
  { fix \( z \)
    assume \( z: z \in B.\text{span} \ (\text{insert} \ a \ b) \)
    have \( \text{th0} : z - ?h \cdot z \cdot b \ a \in B.\text{span} \ b \)
  }

  30
apply (rule someI-ex)
unfolding B.span-breakdown-eq[symmetric]
apply (rule z)
done

{ fix k
  assume k: z - ?h z * b a ∈ B.span b
  have eq: z - ?h z * b a - (z - k * b a) = (k - ?h z) * b a
    by (simp add: field-simps B.scale-left-distrib [symmetric])
  from B.span-sub[OF th0 k] have khz: (k - ?h z) * b a ∈ B.span b
    by (simp add: eq)
  }
  assume k ≠ ?h z
  then have k0: k - ?h z ≠ 0 by simp
  from k0 B.span-mul[OF khz, of 1/(k - ?h z)]
  have a ∈ B.span b by simp
  with 2.prems(1) 2.hyps(2) have False
    by (auto simp add: B.dependent-def)
  }
  then have k = ?h z by blast

with th0 have z - ?h z * b a ∈ B.span b ∧ (∀ k. z - k * b a ∈ B.span b → k = ?h z)
  by blast

note h = this
let ?g = λz. (?h z) * c (f a) + g (z - (?h z) * b a)

{ fix x y
  assume x: x ∈ B.span (insert a b)
  and y: y ∈ B.span (insert a b)
  have tha: ∀(x::'b) y a k. l. (x + y) - (k + l) * b a = (x - k * b a) + (y - l * b a)
    by (simp add: algebra-simps)
  have addh: ?h (x + y) = ?h x + ?h y
    apply (rule conjunct2[OF h, rule-format, symmetric])
    apply (rule B.span-add[OF x y])
    unfolding tha
    apply (metis B.span-add x y conjunct1[OF h, rule-format])
    done
  have ?g (x + y) = ?g x + ?g y
    unfolding addh tha
g(1)[rule-format,OF conjunct1[OF h, OF x] conjunct1[OF h, OF y]]
    by (simp add: C.scale-left-distrib)
  }
moreover
{ fix x :: 'b
  fix c :: 'a
  assume x: x ∈ B.span (insert a b)
have \( \text{tha: } (x::'b) \ c \ k a, c \ast b x - (c \ast k) \ast b a = c \ast b (x - k \ast b a) \)
by (simp add: algebra-simps)

have \( \text{hc: } \ ?h (c \ast b x) = c \ast ?h x \)
apply (rule conjunct2[OF h, rule-format, symmetric])
apply (metis B.span-mul x)
apply (metis tha B.span-mul x conjunct1[OF h])
done

have \( \ ?g (c \ast b x) = c \ast c \ ?g x \)
unfolding hc tha g(2)[rule-format, OF conjunct1[OF h, OF x]]
by (simp add: algebra-simps)

moreover
\{
fix x
assume x: \( x \in \text{insert } a \ b \)
\{
assume xa: \( x = a \)
have ha1: \( 1 = ?h a \)
apply (rule conjunct2[OF h, rule-format])
apply (metis B.span-superset insertI1)
using conjunct1[OF h, OF B.span-superset, OF insertI1]
apply (auto simp add: B.span-0)
done
from xa ha1[symmetric] have \( ?g x = f x \)
apply simp
using g(2)[rule-format, OF B.span-0, of 0]
apply simp
done
\}
moreover
\{
assume xb: \( x \in b \)
have h0: \( 0 = ?h x \)
apply (rule conjunct2[OF h, rule-format])
apply (metis B.span-superset x)
apply simp
apply (metis B.span-superset xb)
done
have \( ?g x = f x \)
by (simp add: h0[symmetric] g(3)[rule-format, OF xb])
\}
ultimately have \( ?g x = f x \)
using x by blast
\}
ultimately show \( ?case \)
apply -
apply (rule exI[where \( x = ?g \)])
apply blast
done

32
locale two-finite-dimensional-vector-spaces-over-same-field = B: finite-dimensional-vector-space scaleB BasisB + C: finite-dimensional-vector-space scaleC BasisC
for scaleB :: ('a::field => 'b::ab-group-add => 'b) (infixr '* b 75)
and scaleC :: ('a => 'c::ab-group-add => 'c) (infixr '* c 75)
and BasisB :: ('b set)
and BasisC :: ('c set)

context two-finite-dimensional-vector-spaces-over-same-field
begin
sublocale two-vector-spaces: two-vector-spaces-over-same-field by unfold-locales

lemma linear-independent-extend:
assumes iB: B.independent B
shows \exists g. linear (op '* b) (op '* c) g \land (\forall x \in B. g x = f x)
proof -
  have 1: vector-space (op '* b) and 2: vector-space (op '* c) by unfold-locales
  from B.maximal-independent-subset-extend[of B UNIV] iB
  obtain C where C: B \subseteq C B.independent C \land x. x \in B.span C
    by auto
  from C(2) B.independent-bound[of C] two-vector-spaces.linear-independent-extend-lemma[of C]
  obtain g where g:
    (\forall x \in B.span C. \forall y \in B.span C. g (x + y) = g x + g y) \land
    (\forall x \in B.span C. \forall c. g (c * b x) = c * c g x) \land
    (\forall x \in C. g x = f x) by blast
  from g show ?thesis
    unfolding linear-iff
    using C 1 2
    apply clarsimp
    apply blast
  done
qed
end

context vector-space
begin

lemma spans-image:
assumes lf: linear scaleC (f::'b=>'c::ab-group-add)
and VB: V \subseteq span B
shows f ' V \subseteq vector-space.span scaleC (f ' B)
unfolding span-linear-image[OF lf] by (metis VB image_mono)
lemma subspace-kernel:
  assumes lf: linear scale scaleC f
  shows subspace { x. f x = 0 }
  proof (unfold subspace-def, auto)
    interpret lf: linear scale scaleC f using lf by simp
    show f 0 = 0 using lf.linear-0
      fix x y assume fx: f x = 0 and fy: f y = 0
      show f (x + y) = 0 unfolding lf.linear-add fx fy by simp
    fix c::'a show f (scale c x) = 0 unfolding lf.linear-cmul lf.scale-zero-right
  qed

lemma linear-eq-0-span:
  assumes lf: linear scale scaleC f and f0: ∀ x∈B. f x = 0
  shows ∀ x∈span B. f x = 0
  using f0 subspace-kernel[OF lf]
  by (rule span-induct')

lemma linear-eq-0:
  assumes lf: linear scale scaleB f and SB: S ⊆ span B
  and f0: ∀ x∈B. f x = 0
  shows ∀ x∈S. f x = 0
  by (metis linear-eq-0-span[OF lf] subset-eq SB f0)

lemma linear-eq:
  assumes lf: linear scale scaleC f and lg: linear scale scaleC g
  and S: S ⊆ span B
  and fg: ∀ x∈B. f x = g x
  shows ∀ x∈S. f x = g x
  proof -
    let ?h = λx. f x - g x
    from fg have fg': ∀ x∈B. ?h x = 0 by simp
    from linear-eq-0[OF linear-compose-sub[OF lf lg]] S fg'
    show ?thesis by simp
  qed

locale linear-between-finite-dimensional-vector-spaces =
  l: linear scaleB scaleC f +
  B: finite-dimensional-vector-space scaleB BasisB +
  C: finite-dimensional-vector-space scaleC BasisC
  for scaleB :: ('a::field => 'b::ab-group-add => 'b) (infixr *b 75)
  and scaleC :: ('a => 'c::ab-group-add => 'c) (infixr *c 75)
  and BasisB :: ('b set)
  and BasisC :: ('c set)
and \( f :: (\text{'} b = \text{'} c) \)

context linear-between-finite-dimensional-vector-spaces
begin

lemma linear-eq-stdbasis:
  assumes \( \text{lg}: \text{linear \ (op \ *b) \ (op \ *c) \ g} \)
  and \( \text{fg}: \forall \text{b \in BasisB}. \ f \ b = g \ b \)
  shows \( f = g \)
proof
  have \( \text{l}: \text{linear \ (op \ *b) \ (op \ *c) \ f} \) by unfold-locales
  show \( \text{?thesis} \)
  using \( B.\text{linear-eq[OF l \ lg, of UNIV BasisB]} \) \( f g \) using \( B.\text{span-Basis} \) by auto
qed

lemma linear-injective-left-inverse:
  assumes \( \text{fi}: \text{inj \ f} \)
  shows \( \exists \ g. \text{linear \ (op \ *c) \ (op \ *b) \ g} \ \land \ g \circ f = \text{id} \)
proof
  interpret \( \text{fd}: \text{two-finite-dimensional-vector-spaces-over-same-field \ (op \ *c) \ (op \ *b) BasisC BasisB} \) by unfold-locales
  have \( \text{l}: \text{linear \ op \ *b \ op \ *c \ f} \) by unfold-locales
  from \( \text{fd.linear-independent-extend[OF independent-injective-image, OF B.independent-Basis, OF \ fi]} \)
  obtain \( \text{h :: \ 'c \Rightarrow \ 'b where \ h: linear \ (op \ *c) \ (op \ *b) \ h} \ \forall \text{x \in f \ ' BasisB}. \ h \ x = \text{inv f x} \)
  by blast
  from \( \text{h(2)} \) have \( \text{th: \ \forall i \in BasisB. \ (h \circ f) \ i = \text{id i} \)
  using \( \text{inv-o-cancel[OF \ fi, unfolded fun-eq-iff id-def o-def]} \)
  by auto
  interpret \( \text{l-hg: linear-between-finite-dimensional-vector-spaces \ op \ *b \ op \ *b BasisB BasisB} \) \( (h \circ f) \)
  apply (unfold-locales) using \( \text{linear-compose[OF l \ h(1)]}\) unfolding linear-iff by fast+
  show \( \text{?thesis} \)
  using \( \text{h(1) \ l-hg.linear-eq-stdbasis[OF B.linear-id \ th]} \) by blast
qed

sublocale two-finite-dimensional-vector-spaces: two-finite-dimensional-vector-spaces-over-same-field
by unfold-locales

lemma linear-surjective-right-inverse:
  assumes \( \text{sf}: \text{surj \ f} \)
  shows \( \exists g. \text{linear \ (op \ *c) \ (op \ *b) \ g} \ \land \ f \circ g = \text{id} \)
proof
  interpret \( \text{lh}: \text{two-finite-dimensional-vector-spaces-over-same-field \ op \ *c \ op \ *b BasisC BasisB} \)
by unfold-locales
have lf: linear (op *b) (op *c) f by unfold-locales
from lh.linear-independent-extend[OF independent-Basis] obtain h:: 'c ⇒ 'b where h: linear (op *c) (op *b) h ∀ x∈BasisC. h x = inv f x by blast
interpret l-fg: linear-between-finite-dimensional-vector-spaces op *c op *c BasisC BasisC (f o h)
  using linear-compose[OF h(1) lf] by (unfold-locales, auto simp add: linear-def linear-axioms-def)
from h(2) have th: ∀ i∈BasisC. (f o h) i = id i
  using sf by (metis comp-apply surj-iff)
from l-fg.linear-eq-stdbasis[OF linear-id th] have f o h = id .
then show ?thesis
  using h(1) by blast
qed

end

context finite-dimensional-vector-space
begin

lemma linear-injective-imp-surjective:
  assumes lf: linear scale scale f
  and fi: inj f
  shows surj f
proof –
  interpret lf: linear scale scale f using lf by auto
  let ?U = UNIV :: 'b set
  from basis-exists[of ?U] obtain B
    where B: B ⊆ ?U independent B ?U ⊆ span B card B = dim ?U
    by blast
  from B(4) have d: dim ?U = card B
    by simp
  have th: ?U ⊆ span (f ' B)
    apply (rule card-ge-dim-independent)
    apply blast
    apply (rule lf.independent-injective-image[OF B(2) fi])
    apply (rule order-eq-refl)
    apply (rule sym)
    unfolding d
    apply (rule card-image)
    apply (rule subset-inj-on[OF fi])
    apply blast
    done
  from th show ?thesis
    unfolding span-linear-image[OF lf] surj-def
    using B(3) by auto

36
lemma linear-surjective-imp-injective:
assumes lf: linear scale scale f
and sf: surj f
shows inj f
proof
interpret t: two-vector-spaces-over-same-field scale scale by unfold-locales
let ?U = UNIV :: 'b set
from basis-exists[of ?U] obtain B
by blast
{ fix x
assume x: x ∈ span B
assume fx: f x = 0
from B(2) have fB: finite B
using independent-bound by auto
have fBi: independent (f'B)
apply (rule card-le-dim-spanning[of f'B ?U])
apply blast
using sf B(3)
unfolding span-linear-image[OF lf] surj-def subset-eq image-iff
apply blast
using fB apply blast
unfolding d[symmetric]
apply (rule card-image-le)
apply (rule fB)
done
have th0: dim ?U ≤ card (f'B)
apply (rule span-card-ge-dim)
apply blast
unfolding span-linear-image[OF lf]
apply (rule subset-trans[where B = f'B UNIV])
using sf unfolding surj-def
apply blast
apply (rule image-mono)
apply (rule B(3))
apply (metis finite-imageI fB)
done
moreover have card (f'B) ≤ card B
by (rule card-image-le, rule fB)
ultimately have th1: card B = card (f'B)
unfolding d by arith
have fB: inj-on f B
unfolding surjective-iff-injective-gen[OF fB finite-imageI[OF fB] th1 subset-refl, symmetric]
by blast
from list:linear-indep-image-lemma[OF fB fBi fB i x] \ x
have \ x = 0 by blast
}
then show \ \thesis
  unfolding linear.linear-injective-0[OF f]
  using B(3)
  by blast
qed

lemma linear-injective-isomorphism:
  assumes \ f\!: linear scale scale \ f
  and \ f\!: inj \ f
  shows \ \exists f'. linear scale scale f' \ \land (\ \forall x. f' (f \ x) = \ x) \ \land (\ \forall f'. f' \ x = \ x)
proof
  interpret lbfdvs: linear-between-finite-dimensional-vector-spaces scale scale Basis \ f
  by (unfold-locales, simp add: f linear.linear-cmul linear.linear-add)
  show \ \thesis
    unfolding isomorphism-expand[symmetric]
    using lbfdvs.linear-surjective-right-inverse linear-linear-add
    by (metis left-right-inverse-eq)
qed

lemma linear-surjective-isomorphism:
  assumes \ f\!: linear scale scale \ f
  and \ f\!: surj \ f
  shows \ \exists f'. linear scale scale f' \ \land (\ \forall x. f' \ x = f \ x) \ \land (\ \forall f'. f' \ x = \ x)
proof
  interpret lbfdvs: linear-between-finite-dimensional-vector-spaces scale scale Basis \ f
  apply (unfold-locales) apply (simp add: f linear-linear-add)
  by (metis left-right-inverse-eq)
qed

lemma left-inverse-linear:
  assumes \ f\!: linear scale scale \ f
  and \ g\!: g \circ f = id
  shows \ linear scale scale \ g
proof
from \( gf \) have \( fi: \text{ inj } f \)
  by (metis inj-on-id inj-on-imageI2)
from linear-injective-isomorphism[OF \( fi \)]
obtain \( h: 'b \Rightarrow 'b \) where \( h: \text{ linear } \)
  \( scale \quad \forall x. \quad h \ (f x) = x \quad \forall x. \quad f \ (h x) = x \)
  by blast
have \( h = g \)
  apply (rule ext) using \( gf \ (2,3) \)
  by (metis comp-apply id-apply)
with \( h(1) \) show \(?thesis\) by blast
qed
end

interpretation vec: finite-dimensional-vector-space op *s (cart-basis)
by (unfold-locales, auto simp add: finite-cart-basis independent-cart-basis span-cart-basis)

lemma matrix-vector-mul-linear-between-finite-dimensional-vector-spaces:
linear-between-finite-dimensional-vector-spaces (op *s) (op *s)
  (cart-basis) (cart-basis) (\( \lambda x. A *v (x::'a::{field} ^-) \))
by (unfold-locales)
  (auto simp add: linear-iff2 matrix-vector-mult-def vec-eq-iff
  field-simps setsum-right-distrib setsum.distrib)

interpretation euclidean-space:
finite-dimensional-vector-space scaleR :: real => 'a => 'a::{euclidean-space}
proof
have \( v: \text{ vector-space } \) (scaleR :: real => 'a => 'a::{euclidean-space}) by (unfold-locales)
show finite \( \text{ (Basis::'a set)} \) by (metis finite-Basis)
show vector-space.independent op *R \( \text{ (Basis::'a set)} \)
  unfolding vector-space.dependent-def[OF \( v \)]
  apply (subst vector-space.span-finite[OF \( v \)])
  apply simp
  apply clarify
  apply (drule_tac f=inner a in arg-cong)
  apply (simp add: inner-Basis inner-setsum-right eq-commute)
  done
show vector-space.span op *R \( \text{ (Basis::'a set)} = \text{ UNIV} \)
  unfolding vector-space.span-finite [OF \( v \) finite-Basis]
  by (fast intro: euclidean-representation)
qed
lemma vector-mul-lcancel[simp]: \( a \cdot s \cdot x = a \cdot s \cdot y \iff a = (0::'a::{field}) \lor x = y \)
  by (metis eq-iff-diff-eq-0 vector-mul-eq-0 vector-sub-ldistrib)

lemma vector-mul-lcancel-imp: \( a \neq (0::'a::{field}) \implies a \cdot s \cdot x = a \cdot s \cdot y \implies (x = y) \)
  by (metis vector-mul-lcancel)

lemma linear-componentwise:
  fixes f :: 'a::{field} 'm :: 'n
  assumes lf: linear (op *s) (op *s) f
  shows \((f \cdot x) \cdot \bar{j} = \text{setsum} (\lambda i. (x \cdot i) \cdot (f \cdot (\text{axis} \cdot i) \cdot \bar{j})) \cdot (\text{UNIV} :: 'm set) \cdot (\text{is} \cdot ?lhs = \cdot ?rhs)\)
  proof –
    interpret lf: linear (op *s) (op *s) f
    using lf .
    let ?M = (\text{UNIV} :: 'm set)
    let ?N = (\text{UNIV} :: 'n set)
    have fM: finite ?M by simp
    have ?rhs = (\text{setsum} (\lambda i. (x \cdot i) \cdot (f \cdot (\text{axis} \cdot i)))) \cdot ?M \cdot \bar{j}
      unfolding setsum-component by simp
    then show ?thesis
      unfolding fM . linear-setsum-mul[OF fM, symmetric]
      unfolding basis-expansion by auto
  qed

lemma matrix-vector-mul-linear: linear (op *s) (op *s) (\lambda x. A * v (x :: 'a::{field} 'n))
  by (simp add: linear-iff2 matrix-vector-mult-def vec-eq-iff
    field-simps setsum-right-distrib setsum.distrib)

interpretation vec: linear op *s op *s (\lambda x. A * v (x :: 'a::{field} 'n))
  using matrix-vector-mul-linear .

interpretation vec: linear-between-finite-dimensional-vector-spaces op *s op *s (cart-basis) (cart-basis) (op *v A)
  by unfold-locales

lemma matrix-works:
  assumes lf: linear (op *s) (op *s) f
  shows matrix f * v x = f (x :: 'a::field 'n)
  apply (simp add: matrix-def matrix-vector-mult-def vec-eq-iff mult.commute)
  apply clarify
  apply (rule linear-componentwise[OF lf, symmetric])
  done

lemma matrix-vector-mul: linear (op *s) (op *s) f =\(\gg f = (\lambda x. \text{matrix} f \cdot v)\)
\( (x::'a::{\text{field}} \to 'n)) \)

by (simp add: ext matrix-works)

lemma matrix-of-matrix-vector-mul: \( \text{matrix}(\lambda x. A * v) \ (x :: 'a::{\text{field}} \to 'n)) = A \)

by (simp add: matrix-eq matrix-vector-mul-linear matrix-works)

lemma matrix-compose: 
assumes \( f::'a::{\text{field}} \to 'm \) 
and \( g::'a\to 'm \)
shows \( \text{matrix}(g \circ f) = \text{matrix} g \circ \text{matrix} f \)
using \( \text{if lg linear-compose[OF If lg] matrix-works[OF linear-compose[OF If lg]]} \)
by (simp add: matrix-eq matrix-works matrix-vector-mul-assoc[symmetric] o-def)

lemma matrix-left-invertible-injective: 
(\( \exists B. (B::'a::{\text{field}} \to 'm \to 'n) \) ** \( A::'a::{\text{field}} \to 'n \to 'm) = \text{mat} \ 1 \) 
\( \iff (\forall x y. A * v x = A * v y \longrightarrow x = y) \) 

proof - 
{ fix \( B::'a\to 'm \) and \( x y \) assume \( B::B \to A = \text{mat} \ 1 \) and \( xy::A * v x = A * v y \)

from \( xy \) have \( B * v (A * v x) = B * v (A * v y) \) by simp

hence \( x = y \)

unfolding matrix-vector-mul-assoc B matrix-vector-mul-lid . } 

moreover 
{ assume \( A::\forall x y. A * v x = A * v y \longrightarrow x = y \)

hence \( i::\text{inj} (op * v A) \) unfolding inj-on-def by auto

from \( \text{vec.linear-injective-left-inverse[OF i]} \)
obtain \( g::\text{linear} (op * s) \) (op * s) \( g g o \circ op * v A = \text{id} \) by blast

have \( \text{matrix} g \) ** \( A = \text{mat} \ 1 \)

unfolding matrix-eq matrix-vector-mul-assoc[symmetric] 
matrix-works[OF g(1)] 
using \( g(2) \) by (metis comp-apply id-apply)

then have \( \exists B. (B::'a::{\text{field}} \to 'm \to 'n) \) ** \( A = \text{mat} \ 1 \) by blast }

ultimately show ?thesis by blast

qed

lemma matrix-left-invertible-ker: 
(\( \exists B. (B::'a::{\text{field}} \to 'm \to 'n) \) ** \( A::'a::{\text{field}} \to 'n \to 'm) = \text{mat} \ 1 \) \( \iff (\forall x. A * v x = 0 \longrightarrow x = 0) \) 

unfolding matrix-left-invertible-injective
using \( \text{vec.linear-injective-0[of A]} \)
by (simp add: inj-on-def)

lemma matrix-left-invertible-independent-columns: 
fixes \( A::'a::{\text{field}} \to 'n \)

shows \( (\exists B::'a \to 'm \to 'n). B \to A = \text{mat} \ 1 \) \( \iff 
(\forall c. \text{setsum} (\lambda i. c i * s \text{ column} i A) \ (UNIV :: 'n \text{ set}) = 0 \longrightarrow (\forall i. c i = 0)) \)

(is \( ?lhs \iff ?rhs \) 

proof –
let \(?U = UNIV :: 'n set\)
\{ assume \(k: \forall x. A \ast v x = 0 \longrightarrow x = 0\)
\{ fix \(c \ i\)
  assume \(c: \text{setsum} \ (\lambda i. c \ i \ast s \ \text{column} \ i A) \ ?U = 0\) and \(i: i \in \ ?U\)
  let \(?x = \chi \ i. c \ i\)
  have \(\text{th0} : A \ast v \ ?x = 0\)
  using \(c\)
  unfolding \(\text{matrix-mult-vsum vec-eq-iff}\)
  by \(\text{auto}\)
  from \(k[\text{rule-format, OF th0}]\)
  have \(?c = 0\) by \((\text{vector vec-eq-iff})\)
\}

hence \(?rhs\) by \(\text{blast}\)
}

moreover
\{ assume \(H: \ ?rhs\)

\{ fix \(x\) assume \(x: A \ast v x = 0\)
  let \(?c = \lambda i. ((x \ i)) :: 'a\)
  from \(H[\text{rule-format, of } \ ?c, \text{unfolded matrix-mult-vsum}[\text{symmetric}, \text{OF } x]\]
  have \(x = 0\) by \(\text{vector}\)
\}

ultimately show \(?thesis\) unfolding \(\text{matrix-left-invertible-ker}\) by \(\text{blast}\)

qed

lemma \(\text{matrix-right-invertible-independent-rows}:\)

fixes \(A :: 'a::{\text{field}} \ 'n \ 'm\)
shows \((\exists (B::'a \ 'm \ 'n). A ** B = mat 1) \longleftrightarrow \)
  \((\forall c. \text{setsum} \ (\lambda i. c \ i \ast s \ \text{row} \ i A) \ (UNIV :: 'm set) = 0 \longrightarrow (\forall i. c \ i = 0))\)

unfolding \(\text{left-invertible-transpose}[\text{symmetric}]\)
  \(\text{matrix-left-invertible-independent-columns}\)
by \((\text{simp add: column-transpose})\)

lemma \(\text{matrix-left-right-inverse}:\)

fixes \(A A' :: 'a::{\text{field}} \ 'n \ 'n\)
shows \(A ** A' = mat 1 \longleftrightarrow A' ** A = mat 1\)

proof –
\{ fix \(A A' :: 'a \ 'n \ 'n\)

assume \(AA' : A ** A' = mat 1\)

have \(sA: \text{surj} (op \ast v A)\)
  unfolding \(\text{surj-def}\)
  apply \(\text{clarify}\)
  apply \((\text{rule-tac } x=(A' \ast v y) \ \text{in } exI)\)
  apply \((\text{simp add: matrix-vector-mul-associative AA' matrix-vector-mul-lid})\)
  done

from \(\text{vec.linear-surjective-isomorphism}[OF matrix-vector-mul-linear sA]\)

obtain \(f' :: 'a \ 'n \Rightarrow 'a \ 'n\)
  where \(f' : \text{linear} (op \ast s) (op \ast s) \ f' \forall x. f' (A \ast v x) = x \forall x. A \ast v f' x = x\)
by \(\text{blast}\)

have \(?th: \text{matrix} f' ** A = mat 1\)

42
by (simp add: matrix-eq matrix-works[OF f'(1)])
matrix-vector-mul-assoc[symmetric] matrix-vector-mul-lid f'(2)[rule-format])

hence (matrix f' ** A) ** A' = mat 1 ** A' by simp
hence matrix f' = A'
by (simp add: matrix-mul-assoc[symmetric] AA' matrix-mul-rid matrix-mul-lid)

hence matrix f' ** A = A' ** A by simp
hence A' ** A = mat 1 by (simp add: th)
}

then show ?thesis by blast
qed

context vector-space
begin

lemma linear-injective-on-subspace-0:
assumes lf: linear scale scale f
and subspace S
shows inj-on f S ←→ (∀ x ∈ S. f x = 0 −→ x = 0)
proof –
have inj-on f S ←→ (∀ x ∈ S. ∀ y ∈ S. f x = f y −→ x = y)
  by (simp add: inj-on-def)
also have ... ←→ (∀ x ∈ S. ∀ y ∈ S. f x - f y = 0 −→ x - y = 0)
  by simp
also have ... ←→ (∀ x ∈ S. ∀ y ∈ S. f (x - y) = 0 −→ x - y = 0)
  using (subspace S) subspace-def[of S] subspace-sub[of S]
by auto
finally show ?thesis .
qed

end

lemma setsum-constant-scaleR:
shows (∑ x∈A. y) = of-nat (card A) *s y
apply (cases finite A)
apply (induct set: finite)
apply (simp-all add: algebra-simps)
done

context finite-dimensional-vector-space
begin

lemma indep-card-eq-dim-span:
assumes independent B
shows finite $B \land \text{card } B = \text{dim} (\text{span } B)$
using assms basis-card-eq-dim[of $B$ span $B$] span-inc by auto
end

context linear
begin

lemma independent-injective-on-span-image:
  assumes $iS$: $B$.independent $S$
  and $fi$: inj-on $f$ ($B$.span $S$)
  shows $C$.independent ($f \circ S$)
proof –
  have $l$: linear $(\text{op } \ast b) (\text{op } \ast c) f$
    by unfold-locales
  { fix $a$
    assume $a$: $a \in S$ $\Rightarrow$ $a \in C$.span ($f \circ S - \{f a\}$)
    have eq: $f \circ S - \{f a\} = f \circ (S - \{a\})$
      using $fi$ a $B$.span-inc by (auto simp add: inj-on-def)
    from $a$ have $f a \in f \circ B$.span ($S - \{a\}$)
      unfolding eq using $B$.span-linear-image[OF $l$] by auto
    moreover have $B$.span ($S - \{a\}$) $\subseteq B$.span $S$
      using $B$.span-mono[of $S - \{a\}$ $S$] by auto
    ultimately have $a \in B$.span ($S - \{a\}$)
      using $fi$ a $B$.span-inc by (auto simp add: inj-on-def)
    with $a(1) iS$ have False
      by (simp add: $B$.dependent-def)
  }
  then show ?thesis
    unfolding dependent-def by blast
qed
end

context vector-space
begin
lemma subspace-Inter: $\forall s \in f$. subspace $s \Rightarrow$ subspace ($\text{Inter } f$)
  unfolding subspace-def by auto

lemma span-eq[simp]: span $s = s \iff$ subspace $s$
  unfolding span-def by (rule hull-eq) (rule subspace-Inter)
end

context finite-dimensional-vector-space
begin
lemma subspace-dim-equal:
  assumes subspace $S$
  and subspace $T$
  and $S \subseteq T$
  and $\text{dim } S \geq \text{dim } T$

shows $S = T$

proof

obtain $B$ where $B: B \leq S$ independent $B \land S \subseteq \text{span } B$ card $B = \text{dim } S$
using basis-exists[of $S$] by auto
then have $\text{span } B \subseteq S$
using span-mono[of $B \leq S$] span-eq[of $S$] assms by metis
then have $\text{span } B = S$
using $B$ by auto
have $\text{dim } S = \text{dim } T$
using assms dim-subset[of $S \leq T$] by auto
then have $\text{span } B \subseteq S$
using card-eq-dim[of $B \leq T$] $B$ assms by (metis independent-bound-general subset-trans)
then show $?\text{thesis}$
using assms ⟨$\text{span } B = S$⟩ by auto
qed
end

lemma det-identical-columns:
fixes $A :: \'a::{\text{comm-ring-1}} \times \'n$
assumes $\text{jk}: j \neq k$
and $r$: column $j \ A = \text{column } k \ A$
shows $\det A = 0$
proof
let $?U = \text{UNIV} :: \'n$
let $?t-jk = \text{Fun}.\text{swap } j \ k \ \text{id}$
let $?PU = \{ p. p \text{ permutes } ?U \}$
let $?S1 = \{ p. p \in ?PU \land \text{evenperm } p \}$
let $?S2 = \{ (?t-jk \circ p) \mid p. p \in $?S1 \}$
let $?f = \lambda p. \text{of-int } (\text{sign } p) \times (\prod i \in \text{UNIV}. A \_ i \ p \ i)$
let $?g = \lambda p. (?t-jk \circ p)$
have $g-S1: \ ?S2 = ?g' \ ?S1$ by auto
have inj-$g$: inj-on $?g$ $?S1$
proof (unfold inj-on-def, auto)
fix $x \ y$ assume $x: x \text{ permutes } ?U$ and even-$x$: evenperm $x$
and $y: y \text{ permutes } ?U$ and even-$y$: evenperm $y$ and eq: $?t-jk \circ x = ?t-jk \circ y$
show $x = y$ by (metis hide-lams, no-types) comp-assoc eq$id$-comp swap-id-idempotent)
qed
have $tjk-permutes$: $?t-jk \text{ permutes } ?U$ unfolding permutes-def swap-id-eq by (auto,metis)
have $tjk$-eq: $\forall i \ l. A \_ i \ S \ ?t-jk \ l = A \_ i \ S \ l$
using $r \ jk$
unfolding column-def vec-eq-iff swap-id-eq by fastforce
have $\text{sign-}$ $tjk$: $\text{sign } ?t-jk = -1$ using $\text{sign}$-swap-id[of $j \ k$] $jk$ by auto
\{fix $x$
assume $x : x \in \mathcal{S}_1$

have $\text{sign } (\tau \cdot \text{jk} \circ x) = \text{sign } (\tau \cdot \text{jk}) \ast \text{sign } x$
by (metis (lifting) finite-class.finite-UNIV mem-Collect-eq
  permutation-permutes permutation-swap-id sign-compose x)
also have $\ldots = - \text{sign } x$ using sign-tjk by simp
also have $\ldots \neq \text{sign } x$ unfolding sign-def by simp
finally have $\text{sign } (\tau \cdot \text{jk} \circ x) \neq \text{sign } x$ and $(\tau \cdot \text{jk} \circ x) \in \mathcal{S}_2$
by (auto, metis (lifting, full-types) mem-Collect-eq x)

} hence disjoint: $\mathcal{S}_1 \cap \mathcal{S}_2 = \{\}$ by (auto, metis sign-def)

have PU-decomposition: $\mathcal{P} = \mathcal{S}_1 \cup \mathcal{S}_2$
proof (auto)
  fix $x$
  assume $x : x$ permutes $\mathcal{U}$ and $\forall p. p$ permutes $\mathcal{U} \longrightarrow x = \text{Fun.swap } j \ k \text{id } \circ p$
  and odd-p: $\neg$ evenperm $p$
  by (metis (no-types) comp-assoc id-comp inv-swap-id permutes-compose
    permutes-inv-o (1) tjk-permutes)

  thus evenperm $x$
  by (metis evenperm-comp evenperm-swap finite-class.finite-UNIV
    jk permutation-permutes permutation-swap-id)

next
  fix $p$
  assume $p : p$ permutes $\mathcal{U}$
  show $\text{Fun.swap } j \ k \text{id } \circ p$ permutes $\mathcal{U}$ by (metis $p$ permutes-compose
    tjk-permutes)
qed

have $\sum \mathcal{F} \mathcal{S}_2 = \sum ((\lambda p. \text{of-int } (\text{sign } p) \ast (\prod i \in \text{UNIV}. A \ \$ i \ \$ p \ i))
  \circ \text{op } \circ (\text{Fun.swap } j \ k \text{id})) \{p \in \{p \ p \text{ permutes } \text{UNIV}, \text{evenperm } p\}\}$
  unfolding $g \mathcal{S}_1$ by (rule setsum.reindex[OF inj-g])
also have $\ldots = \sum ((\lambda p. \text{of-int } (\text{sign } (\tau \cdot \text{jk} \circ p)) \ast (\prod i \in \text{UNIV}. A \ \$ i \ \$ p \ i))$
  $\mathcal{S}_2$
  unfolding o-def by (rule setsum.cong, auto simp add: tjk-eq)
also have $\ldots = \sum ((\lambda p. \text{\ldots } p) \mathcal{S}_1$
proof (rule setsum.cong, auto)
  fix $x$
  assume $x : x$ permutes $\mathcal{U}$
  and even-x: evenperm $x$

  hence perm-x: permutation $x$ and perm-tjk: permutation $\tau \cdot \text{jk}$
  using permutation-permutes[of $x$] permutation-permutes[of $\tau \cdot \text{jk}$] permutation-swap-id
  by (metis finite-code)+

  have $(\text{sign } (\tau \cdot \text{jk} \circ x)) = - (\text{sign } x)$
  unfolding sign-compose[of perm-tjk perm-x] sign-tjk by auto

  thus of-int (sign (\tau \cdot \text{jk} \circ x)) \ast (\prod i \in \text{UNIV}. A \ \$ i \ \$ x \ i)$
  $= - (\text{of-int } (\text{sign } x) \ast (\prod i \in \text{UNIV}. A \ \$ i \ \$ x \ i))$
  by auto

  qed
also have $\ldots = - \text{setsum } ?f \mathcal{S}_1$ unfolding setsum-negf ..
finally have $\ast: \text{setsum } ?f \mathcal{S}_2 = - \text{setsum } ?f \mathcal{S}_1$.
have \( \text{det } A = \left( \sum p \mid p \text{ permutes UNIV} \right) \cdot \text{of-int} (\text{sign } p) \ast \left( \prod i \in \text{UNIV}. \ A \$ \ i \$ \ p \ i \right) \)

unfolding \text{det-def}..
also have \( \ldots = \text{setsum } \ ?f \ ?S1 + \text{setsum } \ ?f \ ?S2 \)
by (\text{subst PU-decomposition, rule setsum.union-disjoint}[OF - - disjoint], auto)
also have \( \ldots = \text{setsum } \ ?f \ ?S1 - \text{setsum } \ ?f \ ?S1 \) unfolding * by auto
also have \( \ldots = 0 \) by simp
finally show \( \text{det } A = 0 \) by simp
qed

lemma \text{det-identical-rows}:
 fixes \( A :: 'a::\{\text{comm-ring-1}\} \rightarrow^{\cdot} 'n \cdot 'n \)
 assumes \( i:j \): \( i \neq j \) and \( r: \text{row } i \ A = \text{row } j \ A \)
 shows \( \text{det } A = 0 \)
 apply (\text{subst det-transpose}[\text{symmetric}])
 apply (\text{rule det-identical-columns}[OF \( ij \)])
 apply (\text{metis column-transpose } \ r)
done

lemma \text{det-zero-row}:
 fixes \( A :: 'a::\{\text{field}\} \rightarrow^{\cdot} 'n \cdot 'n \)
 assumes \( r: \text{row } i \ A = 0 \)
 shows \( \text{det } A = 0 \)
 using \( r \)
 apply (\text{simp add: row-def det-def vec-eq-iff})
 apply (\text{rule setsum.neutral})
 apply (\text{auto})
done

lemma \text{det-zero-column}:
 fixes \( A :: 'a::\{\text{field}\} \rightarrow^{\cdot} 'n \cdot 'n \)
 assumes \( r: \text{column } i \ A = 0 \)
 shows \( \text{det } A = 0 \)
 apply (\text{subst det-transpose}[\text{symmetric}])
 apply (\text{rule det-zero-row } [\text{of } i])
 apply (\text{metis row-transpose } \ r)
done

lemma \text{det-row-operation}:
 fixes \( A :: 'a::\{\text{comm-ring-1}\} \rightarrow^{\cdot} 'n \cdot 'n \)
 assumes \( i:j \): \( i \neq j \)
 shows \( \text{det } (\chi k. \text{if } k = i \text{ then row } i \ A + c * s \text{ row } j \ A \text{ else row } k \ A) = \text{det } A \)
proof --
let \( ?Z = (\chi k. \text{if } k = i \text{ then row } j \ A \text{ else row } k \ A) :: 'a \rightarrow^{\cdot} 'n \cdot 'n \)
have \( \text{th: row } i \triangleq Z = \text{ row } j \triangleq Z \) by (vector row-def)

have \( \text{th2: } ((\chi \ k. \ \text{if } k = i \ \text{then row } i \ A \ \text{else row } k \ A) :: a \cdot n \cdot' n) = A \)
by (vector row-def)

show \( \text{thesis} \)
by simp

qed

lemma \( \text{det-row-span:} \)
fixes \( A :: 'a::{field} \cdot' n \cdot' n \)
assumes \( x: x \in \text{vec.span } \{ \text{row } j \ A | j \neq i \} \)
shows \( \text{det } (\chi \ k. \ \text{if } k = i \ \text{then row } i \ A + x \ \text{else row } k \ A) = \text{det } A \)
proof –
let \( ?U = \text{UNIV } :: 'n \ \text{set} \)
let \( ?S = \{ \text{row } j \ A | j \neq i \} \)
let \( ?d = \lambda x. \text{ det } (\chi \ k. \ \text{if } k = i \ \text{then } x \ \text{else row } k \ A) \)
let \( ?P = \lambda x. ?d (\text{row } i \ A + x) = \text{det } A \)
\{
fix \( k \)
have \( (\text{if } k = i \ \text{then row } i \ A + 0 \ \text{else row } k \ A) = \text{row } k \ A \)
by simp
\}
then have \( P0: ?P \ 0 \)
apply –
apply (rule cong[of det, OF refl])
apply (vector row-def)
done
moreover
\{
fix \( c \ z \ y \)
assume \( zS: z \in ?S \) and \( Py: ?P \ y \)
from \( zS \) obtain \( j \) where \( j: z = \text{row } j \ A \ i \neq j \)
by blast
let \( ?w = \text{row } i \ A + y \)
have \( \text{th0: row } i \ A + (c * s \ z + y) = ?w + c * s \ z \)
by vector
have \( \text{thz: } ?d \ z = 0 \)
apply (rule det-identical-rows[OF j(2)])
using \( j \)
apply (vector row-def)
done
have \( ?d (\text{row } i \ A + (c * s \ z + y)) = ?d \ (?w + c * s \ z) \)
unfolding \( \text{th0} \) ...
then have \( ?P (c * s \ z + y) \)
unfolding \( \text{thz} \ Py \ \text{det-row-mul[of i]} \ \text{det-row-add[of i]} \)
by simp
\}
ultimately show \( \text{thesis} \)
apply –
apply (rule vec.span-induct-alt[of ?P ?S, OF P0, folded scalar-mult-eq-scaleR])
apply blast
apply (rule x)
done

qed

lemma det-dependent-rows:
  fixes A::'a::{field} 'n^'n
  assumes d: vec.dependent (rows A)
  shows det A = 0
proof
  let ?U = UNIV :: 'n set
  from d obtain i where i: row i A ∈ vec.span (rows A − {row i A})
  unfolding vec.dependent-def rows-def by blast
  { fix j k
    assume jk: j ≠ k and c: row j A = row k A
    from det-identical-rows[OF jk c] have thesis .
  }
  moreover
  { assume H: i∈'U. i ≠ j =⇒ row i A ≠ row j A
    have th0: − row i A ∈ vec.span {row j A| j ≠ i}
      apply (rule vec.span-neg)
      apply (rule set-rev-mp)
      apply (rule i)
      apply (rule vec.span-mono)
      using H i
      apply (auto simp add: rows-def)
    done
    from det-row-span[OF th0]
    have det A = det (χ k. if k = i then 0 * s 1 else row k A)
      unfolding right-minus vector-smult-lzero ..
    with det-row-mul[of i 0::'a λi. i]
    have det A = 0 by simp
  }
  ultimately show thesis by blast
qed

lemma det-mul:
  fixes A B :: 'a::{comm-ring-1} 'n^'n
  shows det (A ** B) = det A * det B
proof
  let ?U = UNIV :: 'n set
  let ?F = {f. (∀ i∈'?U. f i ∈ ?U) ∧ (∀ i. i ∉ ?U −→ f i = i)}
  let ?PU = {p. p permutes ?U}
  have fU: finite ?U
    by simp
  have fF: finite ?F
    by simp
by (rule finite)
{
  fix p
  assume p: p permutes ?U
  have p ∈ ?F unfolding mem-Collect-eq permutes-in-image[OF p]
    using p[unfolded permutes-def] by simp
}
then have PUF: ?PU ⊆ ?F by blast
{
  fix f
  assume fPU: f ∈ ?F − ?PU
  have fUU: f · ?U ⊆ ?U
    using fPU by auto
  from fPU have f: ∀i ∈ ?U. f i ∈ ?U ∀i. i /∈ ?U −→ f i = i −(∀y. ∃!x. f x = y)
    unfolding permutes-def by auto
  let ?A = (χ i. A$(i/s) f i * s B$ f i) :: 'a"n"n
  let ?B = (χ i. B$ f i) :: 'a"n"n
  assume fni: ¬ inj-on f ?U
  then obtain i j where ij: f i = f j i ≠ j
    unfolding inj-on-def by blast
  from ij have rth: row i ?B = row j ?B
    by (vector row-def)
  from det-identical-rows[OF ij(2) rth]
  have det (χ i. A$(i/s) f i * s B$ f i) = 0
    unfolding det-rows-mult by simp
}
moreover
{
  assume fi: inj-on f ?U
  from f fi have fith: ∀i j. f i = f j −→ i = j
    unfolding inj-on-def by metis
  note fs = f[unfolded surjective-iff-injective-gen][OF fU fU refl fUU, symmetric]]
  {
    fix y
    from fs f have ∃x. f x = y
      by blast
    then obtain x where x: f x = y
      by blast
    {
      fix z
      assume z: f z = y
      from fith x z have z = x
        by metis
    }
    with x have ∃!x. f x = y
}
by blast
}

with \( f(\varepsilon) \) have \( \det (\chi \ i. \ A\$i\$f \ i \ * \ B\$f \ i) = 0 \)
by blast
}

ultimately have \( \det (\chi \ i. \ A\$i\$f \ i \ * \ B\$f \ i) = 0 \)
by blast
}

then have \( \forall \ f \in ?F - ?PU. \ det (\chi \ i. \ A\$i\$f \ i \ * \ B\$f \ i) = 0 \)
by simp
{
fix \( p \)
assume \( pU: \ p \in ?PU \)
from \( pU \) have \( p: \ p \ \text{permutes} \ ?U \)
by blast
let \( ?s = \lambda p. \ \text{of-int} \ \text{(sign} \ p \))
let \( \forall f = \lambda q. \ ?s \ p \ * (\prod \ i \in ?U. \ A \$ i \$ p \ i) * (\forall q * (\prod \ i \in ?U. \ B \$ i \$ q \ i)) \)
have \( \text{setsum} (\lambda q. \ ?s \ p \ * (\prod \ i \in ?U. \ A \$ i \$ p \ i \ * \ \text{('a -'n -'n)} \$ i \$ q \ i)) \ ?PU \)
unfolding \text{sum-permutations-compose-right}[\text{OF permutes-inv}[\text{OF} \ p], \ \text{of} \ ?f] \)
proof (rule setsum.comm)
fix \( q \)
assume \( qU: \ q \in ?PU \)
then have \( q: \ q \ \text{permutes} \ ?U \)
by blast
from \( p \ q \) have \( pp: \ \text{permutation} \ p \ \text{and} \ pq: \ \text{permutation} \ q \)
unfolding \text{permutation-permutes} by auto
have \( \text{th00:} \ \text{of-int} \ \text{(sign} \ p \) \ * \ \text{of-int} \ \text{(sign} \ p \) \ = \ (1::'a) \)
unfolding \text{mult.assoc}[\text{symmetric}] \)
unfolding \text{of-int-mul}[\text{symmetric}] \)
by (simp-all add: \text{sign-idempotent})
have \( \text{ths:} \ \forall s \ q = \ ?s \ p \ * \ ?s \ (q \ o \ \text{inv} \ p) \)
using \( pp \ pq \ \text{permutation-inverse}[\text{OF} \ pp], \sign-inverse[\text{OF} \ pp] \)
by (simp add: \text{th00 ac-simps sign-idempotent sign-compose})
have \( \text{th001:} \ \text{setprod} (\lambda i. \ B\$i\$ q \ (\text{inv} \ p \ i)) \ ?U \ = \ \text{setprod} ((\lambda i. \ B\$i\$ q \ (\text{inv} \ p \ i))) \ o \ p) \ ?U \)
by (rule setprod-permute[\text{OF} \ p])
have \( \text{thp:} \ \text{setprod} (\lambda i. (\chi \ i. \ A\$i\$p \ i \ * \ B\$p \ i) :: \ \text{('a -'n -'n)} \$ i \$ q \ i) \ ?U \ = \ \text{setprod} (\lambda i. \ A\$i\$p \ i) \ ?U \ * \ \text{setprod} (\lambda i. \ B\$i\$ q \ (\text{inv} \ p \ i)) \ ?U \)
unfolding \( \text{th001 setprod.distrib}[\text{symmetric}] \ o-def \ \text{permutes-inverses}[\text{OF} \ p] \)
apply (rule setprod.cong[\text{OF refl}])
using \text{permutes-in-image}[\text{OF} \ q] \)
apply vector
done
show \( \forall s \ q \ * \ \text{setprod} (\lambda i. ( (\chi \ i. \ A\$i\$p \ i \ * \ B\$p \ i) :: \ \text{('a -'n -'n)}$i$s \ q \ i)) \ ?U \ = \ ?s \ p \ * \ (\text{setprod} (\lambda i. \ A\$i\$p \ i) \ ?U) * (\forall s \ (q \ o \ \text{inv} \ p) \ * \ \text{setprod} (\lambda i. \ B\$i\$ (q \ o \ \text{inv} \ p) \)) \)

51
lemma invertible-left-inverse:
  fixes A :: 'a::{field} ^'n'*'n
  shows invertible A \iff (\exists B::'a^'n*'n). B ** A = mat 1)
  by (metis invertible-def matrix-left-right-inverse)

lemma invertible-right-inverse:
  fixes A :: 'a::{field} ^'n*'^'n
  shows invertible A \iff (\exists B::'a*'^'n*'. B** B = mat 1)
  by (metis invertible-def matrix-left-right-inverse)

lemma invertible-det-nz:
  fixes A::'a::{field} ^'^'n
  shows invertible A \iff det A \neq 0
proof –
  
| { assume invertible A
| then obtain B :: 'a*'^'n*'. where B: A ** B = mat 1
| unfolding invertible-right-inverse by blast
| then have det (A ** B) = det (mat 1 :: 'a*'^'n*'.
| by simp
| then have det A \neq 0
| by (simp add: det-mul det-I) algebra
  }
moreover
| { assume H: \neg invertible A
| let ?U = UNIV :: 'n set
| have fU: finite ?U
| by simp
| from H obtain c i where c: setsum (\lambda i. c i * s row i A) ?U = 0
| and iU: i \in ?U
| and ci: c i \neq 0

52
unfolding invertible-right-inverse
unfolding matrix-right-invertible-independent-rows
by blast
have ∗: \((a::'a"n") b. a + b = 0 \implies -a = b\)
  apply (drule_tac f=op + (\- a) in cong[OF refl])
  apply (simp only: ab-left-minus add.assoc[symmetric])
  apply simp
  done
from c ci
have thr0: \(-row i A = setsum {λ j. (1/ c i) *s (c j *s row j A)} (U - \{i\})\)
  unfolding setsum.remove[OF fU iU] setsum-cmul
  apply –
  apply (rule vector-mul-lecancel-imp[OF ci])
  apply (auto simp add: field-simps)
  unfolding ∗
  apply rule
  done
have thr: \(-row i A \in vec.span \{row j A| j. j \neq i\}\)
  unfolding thr0
  apply (rule vec.span-setsum)
  apply simp
  apply (rule ballI)
  apply (rule vec.span-superset)
  apply auto
  done
let ?B = \((χ k. if k = i then 0 else row k A) :: 'aˆnˆn\)
have thrb: \(row i ?B = 0\) using iU by (vector row-def)
have det A = 0
  unfolding det-row-span[OF thr, symmetric] right-minus
  unfolding det-zero-row[OF thrb] ..
ultimately show ?thesis
  by blast
qed

locale linear-first-finite-dimensional-vector-space = 
l: linear scaleB scaleC f +
B: finite-dimensional-vector-space scaleB BasisB +
C: vector-space scaleC
for scaleB :: ('a::field => 'b::ab-group-add => 'b) (infixr *b 75)
and scaleC :: ('a => 'c::ab-group-add => 'c) (infixr *c 75)
and BasisB :: ('b set)
and f :: ('b=>c)
context linear-between-finite-dimensional-vector-spaces
begin
sublocale lbf: linear-first-finite-dimensional-vector-space by unfold-locales
end

lemma vec-dim-card: vec.dim (UNIV::('a::{field} ^'n) set) = CARD ('n)
proof
  let 'f=λi::'n. axis i (1::'a)
have vec.dim (UNIV::('a::{field} ^'n) set) = card (cart-basis::('a ^'n) set)
    unfolding vec.dim-UNIV ..
also have ... = card (i. i ∈ UNIV)::('n) set
proof (rule bij-betw-same-card[of 'f, symmetric], unfold bij-betw-def, auto)
  show inj (λi::'n. axis i (1::'a)) by (simp add: inj-on-def axis-eq-axis)
  fix i::'n
  show axis i 1 ∈ cart-basis unfolding cart-basis-def by auto
  fix x::'a ^'n
  assume x ∈ cart-basis
  thus x ∈ range (λi. axis i 1) unfolding cart-basis-def by auto
qed
also have ... = CARD('n) by auto
finally show ?thesis .
qed

interpretation vector-space-over-itself: vector-space op * :: 'a::{field} = 'a
  by unfold-locales (simp-all add: algebra-simps)

interpretation vector-space-over-itself: finite-dimensional-vector-space
  op * :: 'a::{field} = 'a
proof (unfold-locales, auto)
  have v: vector-space (op * :: 'a::{field} = 'a) by unfold-locales
  fix x::'a
  show x ∈ vector-space.span (op *) {1::'a} unfolding vector-space.span-singleton[OF v] by auto
qed

lemma dimension-eq-1[code-unfold]: vector-space-over-itself.dimension TYPE('a::{field})= 1
  unfolding vector-space-over-itself.dimension-def by simp

interpretation complex-over-reals: finite-dimensional-vector-space (op *R)::real=>complex=>complex
  {1, i}
proof unfold-locales
  show finite {1, i} by auto
  show vector-space.independent (op *R) {1, i}
    by (metis Basis-complex-def euclidean-space.independent-Basis)
  show vector-space.span (op *R) {1, i} = UNIV
    by (metis Basis-complex-def euclidean-space.span-Basis)
qed
qed

lemma complex-over-reals-dimension[code-unfold]:
  complex-over-reals.dimension = 2 unfolding complex-over-reals.dimension-def
by auto

term op *s

term op *R

end

2  Dual Order

theory Dual-Order
  imports Main
begin

2.1 Interpretation of dual order based on order

Computable Greatest value operator for finite linorder classes. Based on
Least ?P = ( THE x. ?P x ∧ (∀ y. ?P y → x ≤ y))

interpretation dual-order: order (op ≥)::('a::{order}=>'a=>bool) (op >)
proof
  fix x y::'a::{order} show (y < x) = (y ≤ x ∧ ¬ x ≤ y) using less-le-not-le .
  show x ≤ x using order-refl .
  fix z show y ≤ x → z ≤ y → z ≤ x using order-trans .
next
  fix x y::'a::{order} show y ≤ x → x ≤ y → x = y by (metis eq-iff)
qed

interpretation dual-linorder: linorder (op ≥)::('a::{linorder}=>'a=>bool) (op >)
proof
  fix x y::'a show y ≤ x ∨ x ≤ y using linear .
qed

lemma wf-wellorderI2:
  assumes wf: wf {(x::'a::ord, y). y < x}
  assumes lin: class.linorder (λ(x::'a) y::'a. y ≤ x) (λ(x::'a) y::'a. y < x)
shows class.wellorder \((\lambda x::'a\ y::'a.\ y \leq x)\) \((\lambda x::'a\ y::'a.\ y < x)\)

using lin unfolding class.wellorder-def apply (rule conjI)
apply (rule class.wellorder-axioms.intro) by (blast intro: wf-induct-rule [OF wf])

lemma (in preorder) tranclp-less': \(op \gg\gg = op >\)
by(auto simp add: fun-eq-iff intro: less-trns elim: tranclp.induct)

interpretation dual-wellorder: wellorder \((\gg\gg)\) ::('a::{linorder, finite}=>'a==bool)
(op >)
proof (rule wf-wellorderI2)
show wf \{\(x::'a,\ y.\ y < x\}\}
by(auto simp add: trancl-def tranclp-less' intro: finite-acyclic-wf acyclic1)
show class.linorder \((\lambda(x::'a\ y::'a.\ y \leq x)\) \((\lambda(x::'a\ y::'a.\ y < x)\)
unfolding class.linorder-def unfolding class.linorder-axioms-def unfolding class.order-def
unfolding class.preorder-def unfolding class.order-axioms-def by auto
qed

2.2 Computable greatest operator

definition Greatest' :: ('a::order => bool) => 'a::order (binder GREATEST' 10)
where Greatest' \(P = dual-order.Least P\)

The following THE operator will be computable when the underlying type belongs to a suitable class (for example, Enum).

lemma [code]: Greatest' \(P = (THE x::'a::order.\ P x \land (\forall y::'a::order.\ P y \rightarrow y \leq x))\)
unfolding Greatest'-def ord.Least-def by fastforce

lemmas Greatest'12-order = dual-order.Least12-order[folded Greatest'-def]
lemmas Greatest'-equality = dual-order.Least-equality[folded Greatest'-def]
lemmas Greatest'1 = dual-wellorder.Least1[folded Greatest'-def]
lemmas Greatest'12-ex = dual-wellorder.Least12-ex[folded Greatest'-def]
lemmas Greatest'12-wellorder = dual-wellorder.Least12-wellorder[folded Greatest'-def]
lemmas Greatest'1-ex = dual-wellorder.Least1-ex[folded Greatest'-def]
lemmas not-greater-Greatest' = dual-wellorder.not-less-Least[folded Greatest'-def]
lemmas Greatest'12 = dual-wellorder.Least12[folded Greatest'-def]
lemmas Greatest'-ge = dual-wellorder.Least-le[folded Greatest'-def]

end

3 Class for modular arithmetic

theory Mod-Type
imports
$ISABELLE-HOME/src/HOL/Library/Numeral-Type
$ISABELLE-HOME/src/HOL/Multivariate-Analysis/Cartesian-Euclidean-Space
Dual-Order

56
3.1 Definition and properties

Class for modular arithmetic. It is inspired by the locale mod\_type.

class mod\_type = times + wellorder + neg\_numeral +

fixes Rep :: 'a => int
and Abs :: int => 'a

assumes type: type-definition Rep Abs {0..<int CARD ('a)}
and size1: 1 < int CARD ('a)
and zero-def: 0 = Abs 0
and one-def: 1 = Abs 1
and add-def: x + y = Abs ((Rep x + Rep y) mod (int CARD ('a)))
and mult-def: x * y = Abs ((Rep x * Rep y) mod (int CARD ('a)))
and diff-def: x - y = Abs ((Rep x - Rep y) mod (int CARD ('a)))
and minus-def: - x = Abs ((- Rep x) mod (int CARD ('a)))
and strict-mono-Rep: strict-mono Rep

begin

lemma size0: 0 < int CARD ('a)
using size1 by simp

lemmas definitions =
zero-def one-def add-def mult-def minus-def diff-def

lemma Rep-less-n: Rep x < int CARD ('a)
by (rule type-definition.Rep [OF type, simplified, THEN conjunct2])

lemma Rep-le-n: Rep x ≤ int CARD ('a)
by (rule Rep-less-n [THEN order-less-imp-le])

lemma Rep-inject-sym: x = y <-> Rep x = Rep y
by (rule type-definition.Rep-inject [OF type, symmetric])

lemma Rep-inverse: Abs (Rep x) = x
by (rule type-definition.Rep-inverse [OF type])

lemma Abs-inverse: m ∈ {0..<int CARD ('a)} ==> Rep (Abs m) = m
by (rule type-definition.Abs-inverse [OF type])

lemma Rep-Abs-mod: Rep (Abs (m mod int CARD ('a))) = m mod int CARD ('a)
by (simp add: Abs-inverse pos-mod-conj [OF size0])

lemma Rep-Abs-0: Rep (Abs 0) = 0
apply (rule Abs-inverse [of 0])
using size0 by simp

lemma Rep-0: Rep 0 = 0
by (simp add: zero-def Rep-Abs-0)

lemma Rep-Abs-1: Rep (Abs 1) = 1
by (simp add: Abs-inverse size1)

lemma Rep-1: Rep 1 = 1
by (simp add: one-def Rep-Abs-1)

lemma Rep-mod: Rep x mod int CARD ('a) = Rep x
  apply (rule-tac x=x in type-definition.Abs-cases [OF type])
  apply (simp add: type-definition.Abs-inverse [OF type])
  apply (simp add: mod-pos-pos-trivial)
done


3.2 Conversion between a modular class and the subset of natural numbers associated.

Definitions to make transformations among elements of a modular class and naturals

definition to-nat :: 'a => nat
  where to-nat = nat ◦ Rep

definition Abs' :: int => 'a
  where Abs' x = Abs(x mod int CARD ('a))

definition from-nat :: nat => 'a
  where from-nat = (Abs' ◦ int)

lemma bij-Rep: bij-betw (Rep) (UNIV::'a set) {0..<int CARD('a)}
proof (unfold bij-betw-def , rule conjI)
  show inj Rep by (metis strict-mono-imp-inj-on strict-mono-Rep)
  show range Rep = {0..<int CARD('a)} using Typedef.type-definition.Rep-range[OF type].
qed

lemma mono-Rep: mono Rep by (metis strict-mono-Rep strict-mono-mono)

lemma Rep-ge-0: 0 ≤ Rep x using bij-Rep unfolding bij-betw-def by auto

lemma bij-Abs: bij-betw (Abs) {0..<int CARD('a)} (UNIV::'a set)
proof (unfold bij-betw-def , rule conjI)
  show inj-on Abs {0..<int CARD('a)} by (metis inj-on-inverse1 type type-definition.Abs-inverse)
  show Abs ' {0..<int CARD('a)} = (UNIV::'a set) by (metis type type-definition.univ)
qed
corollary bij-Abs': bij-btw (Abs') {0..<CARD('a')} (UNIV::'a set)
proof (unfold bij-btw-def, rule conjI)
  show inj-on Abs' {0..<CARD('a)}
    unfolding inj-on-def Abs'-def
    by (auto, metis Rep-Abs-mod mod-pos-pos-trivial)
show Abs' ' {0..<CARD('a)} = (UNIV::'a set)
proof (unfold image-def Abs'-def, auto)
  fix x show \exists x\in\{0..<CARD('a)\}. x = Abs (xa mod int CARD('a))
    by (rule bexI[of - Rep x], auto simp add: Rep-less-n[of x] Rep-ge-0[of x], metis Rep-inverse Rep-mod)
qed
qed

lemma bij-from-nat: bij-btw (from-nat) {0..<CARD('a)} (UNIV::'a set)
proof (unfold bij-btw-def, rule conjI)
  have set-eq: \{0::int..<CARD('a)\} = int' \{0..<CARD('a)\} apply (auto)
  proof
    fix x::int assume x1: (0::int) \leq x and x2: x < int CARD('a)
    show x \in int' \{0::nat..<CARD('a)\}
      proof (unfold image-def, auto, rule bexI[of - nat x])
        show x = int (nat x) using x1 by auto
      qed
    show nat x \in \{0::nat..<CARD('a)\} using x1 x2 by auto
  qed
qed

show inj-on (from-nat::nat\Rightarrow'a) {0::nat..<CARD('a)}
proof (unfold from-nat-def, rule comp-inj-on)
  show inj-on int {0::nat..<CARD('a)} by (metis inj-of-nat subset-inj-on top-greatest)
  show inj-on (Abs'::int\Rightarrow'a) (int' \{0::nat..<CARD('a)\})
    using bij-Abs unfolding bij-btw-def set-eq
    by (metis (hide-lams, no-types) Abs'-def Abs-inverse Rep-inverse Rep-mod inj-on-def set-eq)
  qed

show (from-nat::nat\Rightarrow'a) ' {0::nat..<CARD('a)} = UNIV
proof (unfold from-nat-def using bij-Abs')
  unfolding bij-btw-def set-eq o-def by blast
qed

lemma to-nat-is-inv: the-inv-into \{0..<CARD('a)\} (from-nat::nat\Rightarrow'a) = (to-nat::'a\Rightarrow>nat)
proof (unfold the-inv-into-def fun-eq-iff from-nat-def to-nat-def o-def, clarify)
  fix x::'a show \(THE y::nat. y \in \{0::nat..<CARD('a)\} \land Abs' (int y) = x\) = nat (Rep x)
proof (rule the-equality, auto)
  show Abs' (Rep x) = x by (metis Abs'-def Rep-inverse Rep-mod)
  show nat (Rep x) < CARD('a) by (metis (full-types) Rep-less-n nat-int size0 zless-nat-conj)
  assume x: \neg (0::int) \leq Rep x show \(0::nat) < CARD('a) and Abs' (0::int) = x
    using Rep-ge-0 x by auto
next

59
\[
\begin{align*}
\text{fix } y \colon \text{nat} &\quad \text{assume } y \colon y < \text{CARD}(\text{'}a\text{')} \\
\text{have } (\text{Rep}(\text{Abs}(\text{int } y) \colon \text{'}a\text{'))) &\quad = \quad (\text{Rep}((\text{Abs}(\text{int } y \mod \text{ int } \text{CARD}(\text{'}a\text{'))} \colon \text{'}a\text{'})) \quad \text{unfolding Abs'-def} \\
&\quad \text{also have } \ldots &= \quad (\text{Rep} (\text{Abs } (\text{int } y) \colon \text{'}a\text{'))) \quad \text{using zmod-int[of } y \text{ CARD}(\text{'}a\text{'))} \\
&\quad \text{using } y \mod \text{ less by auto} \\
&\quad \text{also have } \ldots &= \quad (\text{int } y) \quad \text{proof (rule Abs-inverse) show } \text{int } y \in \{0::\text{int}..<\text{int CARD}(\text{'}a\text{'))} \\
&\quad \text{using } y \by \text{auto} \quad \text{qed} \\
\text{finally show } y &= \quad \text{nat } (\text{Rep} (\text{Abs } (\text{int } y) \colon \text{'}a\text{'))} \quad \text{by (metis nat-int)} \quad \text{qed} \\
\text{qed} \\
\text{lemma bij-to-nat: bij-betw } (\text{to-nat }) \quad (\text{UNIV} \colon \text{'}a \text{ set}) \quad \{0..<\text{CARD}(\text{'}a\text{'))} \\
&\quad \text{using bij-betw-the-inv-into[OF bij-from-nat]} \quad \text{unfolding to-nat-is-inv} . \\
\text{lemma finite-mod-type: finite } (\text{UNIV} \colon \text{'}a \text{ set}) \\
&\quad \text{using finite-imageD[of to-nat UNIV :\text{'}a \text{ set}] \quad \text{using bij-to-nat} \quad \text{unfolding bij-betw-def} \\
&\quad \by \text{auto} \\
\text{subclass (in mod-type) finite by (intro-classes, rule finite-mod-type)} \\
\text{lemma least-0: (LEAST } n \colon n \in (\text{UNIV} \colon \text{'}a \text{ set}) = 0 \\
&\quad \text{proof (rule Least-equality, auto)} \\
&\quad \text{fix } y : \text{'}a \\
&\quad \text{have } (0:='a) &\quad \leq \quad \text{Abs } (\text{Rep } y \mod \text{ int } \text{CARD}(\text{'}a\text{'))} \quad \text{using strict-mono-Rep \quad unfolding} \\
&\quad \text{strict-mono-def} \\
&\quad \text{by (metis (hide-lams, mono-tags) Rep-0 Rep-ge-0 strict-mono-Rep strict-mono-less-eq)} \\
&\quad \text{also have } \ldots &= \quad y \quad \by \quad \text{metis \ Rep-inverse \ Rep-mod} \\
&\quad \text{finally show } (0::'a) \ &\leq \ y . \quad \text{qed} \\
\text{qed} \\
\text{lemma add-to-nat-def: } x + y &= \quad \text{from-nat } (\text{to-nat } x + \text{ to-nat } y) \\
&\quad \text{unfolding from-nat-def to-nat-def o-def using Rep-ge-0[of } x \text{]} \quad \text{using Rep-ge-0[of } y \text{]} \\
&\quad \text{using Rep-less-n[of } x \text{]} \quad \text{Rep-less-n[of } y \text{]} \\
&\quad \text{unfolding Abs'-def \quad unfolding add-def[of } x \ y \text{]} \quad \text{by auto} \\
\text{lemma to-nat-1: to-nat } 1 &= 1 \\
&\quad \by \quad \text{metis (hide-lams, mono-tags) Rep-1 comp-apply to-nat-def transfer-nat-int-numerals(2)} \\
\text{lemma add-def':} \\
&\quad \text{shows } x + y = \quad \text{Abs'} (\text{Rep } x + \text{ Rep } y) \quad \text{unfolding Abs'-def using add-def by simp} \\
\text{lemma Abs'-0:} \\
&\quad \text{shows } \text{Abs'} (\text{CARD}(\text{'}a\text'))=(0::'a) \quad \text{by (metis (hide-lams, mono-tags) Abs'-def mod-self zero-def)} \\
\text{lemma Rep-plus-one-le-card:} \\
\end{align*}
\]
assumes $a$: $a + 1 \neq 0$
shows $(\text{Rep } a) + 1 < \text{CARD} \ ('a')$

proof (rule ccontr)

assume $\neg \text{Rep } a + 1 < \text{CARD} ('a')$ hence to-nat-eq-card: $\text{Rep } a + 1 = \text{CARD} ('a')$

by (metis (hide-lams, mono-tags) Rep-less-n add1-zle-eq dual-order.le-less)

have $a + 1 = \text{Abs} ('a') (\text{Rep } a + \text{Rep} (1::'a))$ using add-def by auto

also have $... = \text{Abs} ('a') ((\text{Rep } a) + 1)$ using Rep-1 simp

also have $... = \text{Abs} ('a') (\text{CARD} ('a'))$ unfolding to-nat-eq-card ..

also have $... = 0$ using Abs-0 by auto

finally show False using a by contradiction

qed

lemma to-nat-plus-one-less-card: $\forall a. a + 1 \neq 0 \rightarrow \text{to-nat } a + 1 < \text{CARD} ('a')$

proof (clarify)

fix $a$

assume $a + 1 \neq 0$

have $\text{Rep } a + 1 < \text{int} \ \text{CARD} ('a)$ using Rep-plus-one-le-card[OF $a$] by auto

hence $\text{nat} (\text{Rep } a + 1) < \text{nat} (\text{int} \ \text{CARD} ('a))$ unfolding zless-nat-conj using size0 by fast

thus $\text{to-nat } a + 1 < \text{CARD} ('a)$ unfolding to-nat-def o-def using nat-add-distrib[OF Rep-ge-0] by simp

qed

corollary to-nat-plus-one-less-card':

assumes $a+1 \neq 0$

shows $\text{to-nat } a + 1 < \text{CARD} ('a)$ using to-nat-plus-one-less-card assms by simp

lemma strict-mono-to-nat: strict-mono $\text{to-nat}$

using strict-mono-Rep

unfolding strict-mono-def to-nat-def using Rep-ge-0 by (metis comp-apply nat-less-eq-zless)

lemma to-nat-eq [simp]: $\forall x. \text{to-nat } x = \text{to-nat } y \leftrightarrow x = y$

using injD [OF bij-betw-imp-inj-on[OF bij-to-nat]] by blast

lemma mod-type-forall-eq [simp]: $(\forall j::'a. (\text{to-nat}::'a=\text{nat}) j < \text{CARD} ('a) \rightarrow P j) = (\forall a. P a)$

proof (auto)

fix $a$ assume $a: \forall j. (\text{to-nat}::'a=\text{nat}) j < \text{CARD} ('a) \rightarrow P j$

have $(\text{to-nat}::'a=\text{nat}) a < \text{CARD} ('a)$ using bij-to-nat unfolding bij-betw-def by auto

thus $P a$ using $a$ by auto

qed

lemma to-nat-from-nat:

assumes $t::\text{to-nat } j = k$

shows $\text{from-nat } k = j$

proof –
have from-nat \( k = \text{from-nat} \ (\text{to-nat} \ j) \) unfolding \( t \) ..
also have \( \ldots = \text{from-nat} \ (\text{the-inv-into} \ \{0..<\text{CARD}('a)\} \ (\text{from-nat}) \ j) \) unfolding to-nat-is-inv ..
also have \( \ldots = j \)

proof (rule f-the-inv-into-f)
  show inj-on from-nat \( \{0..<\text{CARD}('a)\} \) by (metis bij-betw-imp-inj-on bij-from-nat)
  show \( j \in \text{from-nat} \ \{0..<\text{CARD}('a)\} \) by (metis \text{UNIV-I} bij-betw-def bij-from-nat)
qed
finally show from-nat \( k = j \).
qed

lemma to-nat-mono:
  assumes \( ab : a < b \)
  shows \( \text{to-nat} \ a < \text{to-nat} \ b \)
  using strict-mono-to-nat unfolding strict-mono-def using assms by fast

lemma to-nat-mono':
  assumes \( ab : a \leq b \)
  shows \( \text{to-nat} \ a \leq \text{to-nat} \ b \)
  proof (cases \( a = b \))
    case True thus \( \text{thesis} \) by auto
  next
    case False
    hence \( a < b \) using \( ab \) by simp
    thus \( \text{thesis} \) using to-nat-mono by fastforce
  qed

lemma least-mod-type:
  shows \( 0 \leq (\text{n}::'a) \)
  using least-0 by (metis \text{full-types} Least-le \text{UNIV-I})

lemma to-nat-from-nat-id:
  assumes \( x : x < \text{CARD}('a) \)
  shows \( \text{to-nat} \ ((\text{from-nat} \ x)::'a) = x \)
  unfolding to-nat-is-inv[symmetric] proof (rule the-inv-into-f.f)
  show inj-on (from-nat::nat\Rightarrow>'a) \( \{0..<\text{CARD}('a)\} \) using bij-from-nat unfolding bij-betw-def by auto
  show \( x \in \{0..<\text{CARD}('a)\} \) using \( x \) by simp
  qed

lemma from-nat-to-nat-id[simp]:
  shows from-nat (to-nat \( x) = x \) by (metis to-nat-from-nat)

lemma from-nat-to-nat:
  assumes \( t:\text{from-nat} \ j = k \) and \( j : j < \text{CARD}('a) \)
  shows \( \text{to-nat} \ k = j \) by (metis \( j \ t \) to-nat-from-nat-id)

lemma from-nat-mono:
  assumes \( i\leq j : i < j \) and \( j : j < \text{CARD}('a) \)
shows \((\text{from-nat } i::'a) < \text{from-nat } j\)

proof –

have \(i < \text{CARD}('a)\) using \(\text{i-le-}j\) by simp

obtain \(a\) where \(a = \text{to-nat } i\)

using \(\text{bij-to-nat}\) unfolding \(\text{bij-betw-def}\) using \(\text{to-nat-from-nat-id}\) by metis

obtain \(b\) where \(b = \text{to-nat } j\)

using \(\text{bij-to-nat}\) unfolding \(\text{bij-betw-def}\) using \(\text{to-nat-from-nat-id}\) by metis

show ?thesis by (metis \(a\) \(b\) \(\text{from-nat-to-nat-id}\) \(\text{i-le-}j\) \(\text{strict-mono-less}\) \(\text{strict-mono-to-nat}\))

qed

lemma \(\text{from-nat-mono}'\):

assumes \(\text{i-le-}j\): \(i \leq j\) and \(j < \text{CARD} ('a)\)

shows \((\text{from-nat } i::'a) \leq \text{from-nat } j\)

proof (cases \(i = j\))

  case True
  have \((\text{from-nat } i::'a) = \text{from-nat } j\) using True by simp
  thus ?thesis by simp

next

  case False
  hence \(i < j\) using \(\text{i-le-}j\) by simp
  thus ?thesis by (metis assms \(2\) \(\text{from-nat-mono}\) \(\text{less-imp-le}\))

qed

lemma \(\text{to-nat-suc}\):

assumes \(\text{to-nat} (x) + 1 < \text{CARD} ('a)\)

shows \(\text{to-nat} (x + 1::'a) = (\text{to-nat } x) + 1\)

proof –

  have \((x::'a) + 1 = \text{from-nat} (\text{to-nat } x + \text{to-nat} (1::'a))\) unfolding \(\text{add-to-nat-def}\)
  ..
  hence \(\text{to-nat} ((x::'a) + 1) = \text{to-nat} (\text{from-nat} (\text{to-nat } x + \text{to-nat} (1::'a))::'a)\)
  by presburger

  also have \(...) = \text{to-nat} (\text{from-nat} (\text{to-nat } x + 1)::'a)\) unfolding \(\text{to-nat-}1\) ..

  also have \(...) = \(\text{to-nat} x + 1\) by (metis assms \text{to-nat-from-nat-id})

  finally show ?thesis.

qed

lemma \(\text{to-nat-le}\):

assumes \(y < \text{from-nat } k\)

shows \(\text{to-nat } y < k\)

proof (cases \(k < \text{CARD} ('a)\))

  case True show ?thesis by (metis \text{full-types} assms \(\text{to-nat-from-nat-id}\) \(\text{to-nat-mono}\))

next

  case False have \(\text{to-nat } y < \text{CARD} ('a)\) using \(\text{bij-to-nat}\) unfolding \(\text{bij-betw-def}\)
  by auto

  thus ?thesis using False by auto

qed

lemma \(\text{le-Suc}\):

\[63\]
assumes \( ab : a < (b::'a) \)
shows \( a + 1 \leq b \)

proof –
have \( a + 1 = (\text{from-nat } (\text{to-nat } (a + 1))::'a) \) using from-nat-to-nat-id [of \( a+1 \), symmetric].
also have \( \ldots \leq (\text{from-nat } (\text{to-nat } (b::'a))::'a) \)
proof (rule from-nat-mono')
  have \( \text{to-nat } a < \text{to-nat } b \) using \( ab \) by (metis to-nat-mono)
  hence \( \text{to-nat } a + 1 \leq \text{to-nat } b \) by simp
  thus \( \text{to-nat } b < \text{CARD} (a) \) using bij-to-nat unfolding bij-betw-def by auto
  hence \( \text{to-nat } a + 1 \leq \text{CARD} (a) \) by (metis to-nat a + 1 \leq to-nat b)
preorder-class.le-less-trans)
  thus \( \text{to-nat } (a + 1) \leq \text{to-nat } b \) by (metis \( \text{to-nat } a + 1 \leq \text{to-nat } b \)
preorder-class.le-less-trans)
qed

also have \( \ldots = b \) by (metis from-nat-to-nat-id)
finally show \( a + (1::'a) \leq b \).
qed

lemma le-Suc':
assumes \( ab : a + 1 \leq b \)
and less-card: \( (\text{to-nat } a) + 1 < \text{CARD} (a) \)
shows \( a < b \)
proof –
have \( a = (\text{from-nat } (\text{to-nat } a))::'a \) using from-nat-to-nat-id [of \( a \), symmetric].
also have \( \ldots < (\text{from-nat } (\text{to-nat } b))::'a \)
proof (rule from-nat-mono)
  show \((\text{to-nat } a) < (\text{to-nat } b)\) using bij-to-nat unfolding bij-betw-def by auto
  have \(\text{to-nat } (a + 1) \leq \text{to-nat } b \) using \( ab \) by (metis to-nat-mono')
  hence \( \text{to-nat } a + 1 \leq \text{to-nat } b \) using \( \text{to-nat-suc}[OF less-card] \) by auto
  thus \( \text{to-nat } a < \text{to-nat } b \) by simp
qed

finally show \( a < b \) by (metis to-nat-from-nat)
qed

lemma Suc-le:
assumes less-card: \( (\text{to-nat } a) + 1 < \text{CARD} (a) \)
shows \( a < a + 1 \)
proof –
have \( (\text{to-nat } a) < (\text{to-nat } a) + 1 \) by simp
hence \( (\text{to-nat } a) < \text{to-nat } (a + 1) \) by (metis less-card to-nat-suc)
  hence \( (\text{from-nat } (\text{to-nat } a))::'a < \text{from-nat } (\text{to-nat } (a + 1)) \)
by (rule from-nat-mono, metis less-card to-nat-suc)
thus \( a < a + 1 \) by (metis to-nat-from-nat)
qed

lemma Suc-le':
fixes a::'a
assumes \( a + 1 \neq 0 \)
shows \( a < a + 1 \) using Suc-le to-nat-plus-one-less-card assms by blast
lemma from-nat-not-eq:
  assumes a-eq-to-nat: a ≠ to-nat b
  and a-less-card: a < CARD('a)
  shows from-nat a ≠ b
proof (rule ccontr)
  assume ¬ from-nat a ≠ b hence from-nat a = b by simp
  hence to-nat ((from-nat a)::'a) = to-nat b by auto
  thus False by (metis a-eq-to-nat a-less-card to-nat-from-nat-id)
qed

lemma Suc-less:
  fixes i::'a
  assumes i < j and i+1 ≠ j
  shows i+1 < j by (metis assms le-Suc le-neq-trans)

lemma Greatest-is-minus-1:
  ∀ a::'a. a ≤ −1
proof (clarify)
  fix a::'a
  have zero-ge-card-1: 0 ≤ int CARD('a) − 1 using size1 by auto
  have card-less: int CARD('a) − 1 < int CARD('a) by auto
  have not-zero: 1 mod int CARD('a) ≠ 0
    by (metis (hide-lams, mono-tags) Rep-Abs-1 Rep-mod zero-neq-one)
  have int-card: int (CARD('a) − 1) = int CARD('a) − 1 using zdiff-int[of 1 CARD ('a)]
    using size1 by simp
  have a = Abs' (Rep a) by (metis (hide-lams, mono-tags) Rep-0 add-0-right add-def'
    monoid-add-class.add.right-neutral)
  also have ... = Abs' (int (nat (Rep a))) by (metis Rep-ge-0 int-nat-eq)
  also have ... ≤ Abs' (int (CARD('a) − 1))
proof (rule from-nat-mono[of unfolded from-nat-def o-def, of nat (Rep a) CARD ('a) − 1])
  show nat (Rep a) ≤ CARD('a) − 1 using Rep-less-n
    by (metis (hide-lams, mono-tags) Rep-1 Rep-le-n dual-linorder.leD dual-linorder.le-less-linear
      of-nat-1 of-nat-diff zle-diff1-eq zle-int zless-nat-eq-int-zless)
  show CARD('a) − 1 < CARD('a) using finite-UNIV-card-ge-0 finite-mod-type
    by fastforce
  qed
  also have ... = −1
  unfolding Abs' -def unfolding minus-def zmod-zminus1-eq-if unfolding Rep-1
  apply (rule cong [of Abs], rule refl)
  unfolding if-not-P [OF not-zero]
  unfolding int-card
  unfolding mod-pos-pos-trivial[OF zero-ge-card-1 card-less]
using mod-pos-pos-trivial[OF size1] by presburger

finally show a ≤ −1 by fastforce

qed

lemma a-eq-minus-1: ∀ a::'a. a + 1 = 0 −→ a = −1
by (metis eq-neg-iff-add-eq-0)

lemma forall-from-nat-rw:
shows (∀ x ∈ {0..<CARD('a)}.{ P (from-nat x::'a)}. P (from-nat x))
proof (auto)
fix y assume *: ∀ x ∈ {0..<CARD('a)}. P (from-nat x)
have from-nat y ∈ (UNIV::'a set) by auto
from this obtain x where x1: from-nat y = (from-nat x::'a) and x2: x ∈ {0..<CARD('a)}
using bij-from-nat unfolding bij-betw-def
by (metis from-nat-to-nat-id rangeI the-inv-into-onto to-nat-is-inv)
show P (from-nat y::'a) unfolding x1 using * x2 by simp
qed

lemma from-nat-eq-imp-eq:
assumes f-eq: from-nat x = (from-nat xa::'a)
and x: x < CARD('a) and xa: xa < CARD('a)
shows x=x a using assms from-nat-not-eq by metis

lemma to-nat-less-card:
fixes j::'a
shows to-nat j < CARD ('a)
using bij-to-nat unfolding bij-betw-def by auto

lemma from-nat-0: from-nat 0 = 0
unfolding from-nat-def o-def of-nat-0 Abs'-def mod-0 zero-def ..
lemma to-nat-0: to-nat 0 = 0 unfolding to-nat-def o-def Rep-0 nat-0 ..
lemma to-nat-eq-0: (to-nat x = 0) = (x = 0) using to-nat-0 to-nat-from-nat by auto

lemma suc-not-zero:
assumes to-nat a + 1 ≠ CARD('a)
shows a+1 ≠ 0
proof (rule contr, simp)
assume a-plus-one-zero: a + 1 = 0
hence rep-eq-card: Rep a + 1 = CARD('a)
using assms to-nat-0 Suc-eq-plus1 Suc-lessI Zero-not-Suc to-nat-less-card to-nat-suc

by (metis (hide-lams, mono-tags))
moreover have Rep a + 1 < CARD('a)
using Abs'-0 Rep-1 Suc-eq-plus1 Suc-lessI Suc-neq-Zero add-def' assms
rep-eq-card to-nat-0 to-nat-less-card to-nat-suc by (metis (hide-lams, mono-tags))
ultimately show False by fastforce

lemma from-nat-suc:
shows from-nat \((j + 1)\) = from-nat \(j + 1\)
unfolding from-nat-def o-def Abs'-def add-def' Rep-1 Rep-Abs-mod
unfolding of-nat-add apply (subst mod-add-left-eq) unfolding int-1 ..

lemma to-nat-plus-1-set:
shows to-nat \(a\) + 1 \(\in\) \(\{1..<\)CARD\((\'a\)\)+1\}\)
using to-nat-less-card by simp

end

lemma from-nat-CARD:
shows from-nat (CARD\((\'a\)) = (\(0::\'a::\{mod-type\}\))
unfolding from-nat-def o-def Abs'-def by (simp add: zero-def)

3.3 Instantiations

instantiation bit0 and bit1:: (finite) mod-type
begin

definition (Rep::\'a bit0 => int) \(x\) = Rep-bit0 \(x\)
definition (Abs::int => \'a bit0) \(x\) = Abs-bit0' \(x\)

definition (Rep::\'a bit1 => int) \(x\) = Rep-bit1 \(x\)
definition (Abs::int => \'a bit1) \(x\) = Abs-bit1' \(x\)

instance

proof

show \((0::\'a bit0) = Abs (0::int)\) unfolding Abs-bit0-def Abs-bit0'-def zero-bit0-def
by auto

show \((1::int) < int\) CARD\((\'a bit0\)\) by (metis bit0.size1)

show type-definition (Rep::\'a bit0 => int) (Abs:: int => \'a bit0) \((0::int..<int\) CARD\((\'a bit0\))\)

proof (unfold type-definition-def Rep-bit0-def [abs-def]
Abs-bit0-def [abs-def] Abs-bit0'-def', intro conjI)

show \(\forall x::\'a bit0.\) Rep-bit0 \(x\) \(\in\) \(\{0::int..<int\) CARD\(('a bit0)\}\)

unfolding card-bit0 unfolding int-mult

using Rep-bit0 [where ?'a = 'a] by simp

show \(\forall x::\'a bit0.\) Abs-bit0 (Rep-bit0 \(x\) mod int \(\)CARD\(('a bit0)\)) = \(x\)

by (metis Rep-bit0-inverse bit0.Rep-mod)

show \(\forall y::int.\) \(y\) \(\in\) \(\{0::int..<int\) CARD\(('a bit0)\}\)

\(\rightarrow\) Rep-bit0 \((\)Abs-bit0::int => \'a bit0\) \((y\) mod int \(\)CARD\(('a bit0)\))\) = \(y\)

by (metis bit0.Abs-inverse bit0.Rep-mod)

qed

show \((1::\'a bit0) = Abs (1::int)\) unfolding Abs-bit0-def Abs-bit0'-def one-bit0-def

qed
by (metis bit0.of-nat-eq of-nat-1 one-bit0-def)
fix x y :: 'a bit0
show \(x + y = \text{Abs}\ ((\text{Rep}\ x + \text{Rep}\ y) \bmod \text{int CARD}'(\text{a bit0}))\)
  unfolding Abs-bit0-def Rep-bit0-def plus-bit0-def Abs-bit0'-def by fastforce
show \(x \cdot y = \text{Abs}\ ((\text{Rep}\ x \cdot \text{Rep}\ y) \bmod \text{int CARD}'(\text{a bit0}))\)
  unfolding Abs-bit0-def Rep-bit0-def times-bit0-def Abs-bit0'-def by fastforce
show \(x - y = \text{Abs}\ ((\text{Rep}\ x - \text{Rep}\ y) \bmod \text{int CARD}'(\text{a bit0}))\)
  unfolding Abs-bit0-def Rep-bit0-def minus-bit0-def Abs-bit0'-def by fastforce
show \(- x = \text{Abs}\ (<-\ \text{Rep}\ x \bmod \text{int CARD}'(\text{a bit0}))\)
  unfolding Abs-bit0-def Rep-bit0-def uminus-bit0-def Abs-bit0'-def by fastforce
show \((0::'a bit1) = \text{Abs}\ (0::\text{int})\) unfolding Abs-bit1-def Abs-bit1'-def zero-bit1-def
by auto
show \((1::\text{int}) < \text{int CARD}'(\text{a bit1})\) by (metis bit1.size1)
show \((1::'a bit1) = \text{Abs}\ (1::\text{int})\) unfolding Abs-bit1-def Abs-bit1'-def one-bit1-def
  by (metis bit1.of-nat-eq of-nat-1 one-bit1-def)
fix x y :: 'a bit1
show \(x + y = \text{Abs}\ ((\text{Rep}\ x + \text{Rep}\ y) \bmod \text{int CARD}'(\text{a bit1}))\)
  unfolding Abs-bit1-def Rep-bit1-def Abs-bit1'-def Rep-bit1-def plus-bit1-def by fastforce
show \(x \cdot y = \text{Abs}\ ((\text{Rep}\ x \cdot \text{Rep}\ y) \bmod \text{int CARD}'(\text{a bit1}))\)
  unfolding Abs-bit1-def Rep-bit1-def times-bit1-def Abs-bit1'-def by fastforce
show \(x - y = \text{Abs}\ ((\text{Rep}\ x - \text{Rep}\ y) \bmod \text{int CARD}'(\text{a bit1}))\)
  unfolding Abs-bit1-def Rep-bit1-def minus-bit1-def Abs-bit1'-def by fastforce
show \(- x = \text{Abs}\ (<-\ \text{Rep}\ x \bmod \text{int CARD}'(\text{a bit1}))\)
  unfolding Abs-bit1-def Rep-bit1-def uminus-bit1-def Abs-bit1'-def by fastforce
show type-definition (Rep::'a bit1 => int) (Abs::\text{int} => 'a bit1) \{0::\text{int}..<\text{int CARD}'(\text{a bit1})\}
proof (unfold type-definition-def Rep-bit1-def [abs-def]
  Abs-bit1-def [abs-def] Abs-bit1'-def, intro conj)
have int-2: \(\text{int 2} = 2\) by auto
show \(\forall x::'a bit1. \text{Rep-bit1} x \in \{0::\text{int}..<\text{int CARD}'(\text{a bit1})\}\)
  unfolding card-bit1
  unfolding int-Suc int-mult
  using \text{Rep-bit1} [where '?a = 'a] unfolding int-2 unfolding add.commute
  ...
  show \(\forall x::'a bit1. \text{Abs-bit1} (\text{Rep-bit1} x \bmod \text{int CARD}'(\text{a bit1})) = x\)
    by (metis \text{Rep-bit1}-inverse bit1.Rep-mod)
show \(\forall y::\text{int. } y \in \{0::\text{int}..<\text{int CARD}'(\text{a bit1})\}\)
  \(\Rightarrow\ \text{Rep-bit1} ((\text{Abs-bit1}::\text{int} => 'a bit1) (y \bmod \text{int CARD}'(\text{a bit1}))) = y\)
  by \(\text{metis bit1.\text{Abs-inverse bit1.Rep-mod}}\)
qed
show strict mono (Rep::'a bit0 => int) unfolding strict mono-def
  by (metis Rep-bit0-def less-bit0-def)
show strict mono (Rep::'a bit1 => int) unfolding strict mono-def
  by (metis Rep-bit1-def less-bit1-def)
qed
end
end

68
4 Miscellaneous

theory Miscellaneous
imports Generalizations
$ISABELLE-HOME/src/HOL/Library/Bit
begin

In this file, we present some basic definitions and lemmas about linear algebra and matrices.

4.1 Definitions of number of rows and columns of a matrix

definition nrows :: 'a::{ab_semigroup_mult}rows => nat
  where nrows A = CARD(rows)

definition ncols :: 'a::{ab_semigroup_mult}rows => nat
  where ncols A = CARD(columns)

definition matrix-scalar-mult :: 'a::{ab_semigroup_mult}n*m => 'a::{ab_semigroup_mult}n*m
  (infixl ∗ 70)
  where k ∗ k A ≡ (χ i j. k ∗ A $ i $ j)

4.2 Basic properties about matrices

lemma nrows-not-0[simp]:
  shows 0 ≠ nrows A unfolding nrows-def by simp

lemma ncols-not-0[simp]:
  shows 0 ≠ ncols A unfolding ncols-def by simp

lemma nrows-transpose: nrows (transpose A) = ncols A
  unfolding nrows-def ncols-def ..

lemma ncols-transpose: ncols (transpose A) = nrows A
  unfolding nrows-def ncols-def ..

lemma finite-rows: finite (rows A)

lemma finite-columns: finite (columns A)
  using finite-Atleast-Atmost-nat[λ i. column i A] unfolding columns-def .

lemma matrix-vector-zero: A ∗ v 0 = 0
  unfolding matrix-vector-mult-def by (simp add: zero_vec_def)

lemma vector-matrix-zero: 0 v* A = 0
  unfolding vector-matrix-mult-def by (simp add: zero_vec_def)
lemma vector-matrix-zero': x v* 0 = 0
  unfolding vector-matrix-mult-def by (simp add: zero-vec-def)

lemma transpose-vector: x v* A = transpose A * v x
  by (unfold matrix-vector-mult-def vector-matrix-mult-def transpose-def, auto)

lemma transpose-zero[simp]: (transpose A = 0) = (A = 0)
  unfolding transpose-def zero-vec-def vec-eq-iff by auto

4.3 Theorems obtained from the AFP

The following theorems and definitions have been obtained from the AFP
http://afp.sourceforge.net/browser_info/current/HOL/Tarski's_Geometry/Linear_Algebra2.html. I have removed some restrictions over the type classes.

lemma vector-matrix-left-distrib:

  shows (x + y) v* A = x v* A + y v* A
  unfolding vector-matrix-mult-def
  by (simp add: algebra-simps setsum.distrib vec-eq-iff)

lemma matrix-vector-right-distrib:

  shows M * v (v + w) = M * v v + M * v w
  proof -
    have M * v (v + w) = (v + w) v* transpose M by (metis transpose-transpose
      transpose-vector)
    also have ... = v v* transpose M + w v* transpose M
      by (rule vector-matrix-left-distrib [of v w transpose M])
    finally show M * v (v + w) = M * v v + M * v w by (metis transpose-transpose
      transpose-vector)
  qed

lemma scalar-vector-matrix-assoc:

  fixes k :: 'a::{field} and x :: 'a::{field} ^'n and A :: 'a ^'m ^'n
  shows (k *s x) v* A = k *s (x v* A)
  unfolding vector-matrix-mult-def unfolding vec-eq-iff
  by (auto simp add: setsum-right-distrib, rule setsum.cong, simp-all)

lemma vector-scalar-matrix-ac:

  fixes k :: 'a::{field} and x :: 'a::{field} ^'n and A :: 'a ^'m ^'n
  shows x v* (k * A) = k *s (x v* A)
  using scalar-vector-matrix-assoc
  unfolding vector-matrix-mult-def matrix-scalar-mult-def vec-eq-iff
  by (auto simp add: setsum-right-distrib)
lemma transpose-scalar: \( (k \cdot k A) = k \cdot k \text{transpose } A \)
unfolding transpose-def
by (vector, simp add: matrix-scalar-mult-def)

lemma scalar-matrix-vector-assoc:
fixes \( A :: 'a::{field}\ 'm\ 'n \)
sows \( k \cdot s (A \cdot v \cdot v) = k \cdot k A \cdot v \cdot v \)
proof –
  have \( k \cdot s (A \cdot v \cdot v) = k \cdot s (v \cdot v \cdot \text{transpose } A) \) by (metis transpose-transpose transpose-vector)
  also have \( \ldots = v \cdot v \cdot (k \cdot k \text{transpose } A) \)
    by (rule vector-scalar-matrix-ac [symmetric])
  also have \( \ldots = v \cdot v \cdot k \cdot k \text{transpose } A \) unfolding transpose-scalar ..
  finally show \( k \cdot s (A \cdot v \cdot v) = k \cdot k A \cdot v \cdot v \) by (metis transpose-transpose transpose-vector)
qed

lemma matrix-scalar-vector-ac:
fixes \( A :: 'a::{field}\ 'm\ 'n \)
sows \( A \cdot v \cdot (k \cdot s \cdot v) = k \cdot k A \cdot v \cdot v \)
proof –
  have \( A \cdot v \cdot (k \cdot s \cdot v) = k \cdot s (v \cdot v \cdot \text{transpose } A) \)
    by (metis transpose-transpose scalar-vector-matrix-assoc transpose-vector)
  also have \( \ldots = v \cdot v \cdot (k \cdot k \text{transpose } A) \)
    by (rule subst vector-scalar-matrix-ac simp)
  also have \( \ldots = k \cdot k A \cdot v \cdot v \) by (metis transpose-transpose transpose-vector)
  finally show \( A \cdot v \cdot (k \cdot s \cdot v) = k \cdot k A \cdot v \cdot v \).
qed

definition is-basis :: ('a::{field}\ 'm\ 'n) set => bool where
  is-basis S ≡ vec.independent S ∧ vec.span S = UNIV

lemma card-finite:
  assumes card S = CARD('n::finite)
  shows finite S
proof –
  from \( \text{card } S = \text{CARD('n)} \) have \( \text{card } S \neq 0 \) by simp
  with \( \text{card-eq-0-iff } [\text{of } S] \) show finite S by simp
qed

lemma independent-is-basis:
  fixes \( B :: 'a::{field}\ 'm\ 'n \) set
  shows vec.independent B ∧ card B = CARD('n) ⇔ is-basis B
proof
  assume vec.independent B ∧ card B = CARD('n)
  hence vec.independent B and card B = CARD('n) by simp
from card-finite [of B, where 'n = 'n] and (card B = CARD('n))
have finite B by simp
from (card B = CARD('n))
have card B = vec.dim (UNIV :: ('a ^ 'n) set) unfolding vec-dim-card .
with vec.card-eq-dim [of B UNIV] and (finite B) and (vec.independent B)
have vec.span B = UNIV by auto
with (vec.independent B) show is-basis B unfolding is-basis-def ..
next
assume is-basis B
hence vec.independent B unfolding is-basis-def ..
moreover have card B = CARD('n)
proof –
  have B ⊆ UNIV by simp
  moreover
    { from (is-basis B) have UNIV ⊆ vec.span B and vec.independent B
      unfolding is-basis-def
      by simp+ }
    ultimately have card B = vec.dim (UNIV::{real ^ 'n} set))
    using vec.basis-card-eq-dim [of B UNIV]
    unfolding vec-dim-card
    by simp
    then show card B = CARD('n)
    by (metis vec-dim-card)
  qed
ultimately show vec.independent B ∧ card B = CARD('n) ..
qing

lemma basis-finite:
  fixes B :: ('a::{field} ^ 'n) set
  assumes is-basis B
  shows finite B
proof –
  from independent-is-basis [of B] and (is-basis B) have card B = CARD('n)
  by simp
  with card-finite [of B, where 'n = 'n] show finite B by simp
qed

Here ends the statements obtained from AFP: http://afp.sourceforge.net/
browser_info/current/HOL/Tarskis_Geometry/Linear_Algebra2.html which
have been generalized.

4.4 Basic properties involving span, linearity and dimensions

context finite-dimensional-vector-space
begin
This theorem is the reciprocal theorem of local.independent ?B ⇒ finite
?B ∧ card ?B = local.dim (local.span ?B)

lemma card-eq-dim-span-indep:
assumes \( \text{dim} (\text{span} \ A) = \text{card} \ A \) and finite \( A \)
shows independent \( A \)
by (metis assms card-le-dim-spanning dim-subset equalityE span-inc)

lemma \( \text{dim-zero-eq} \):

assumes \( \text{dim-A: dim} \ A = 0 \)
shows \( A = \{\} \lor A = \{0\} \)
proof –
obtain \( B \) where ind-\( B \): independent \( B \) and A-in-span-\( B \): \( A \subseteq \text{span} \ B \)
and card-\( B \): card \( B = 0 \) using basis-exists[of \( A \)] unfolding dim-A by blast
have finite-\( B \): finite \( B \) using indep-card-eq-dim-span[of \( B \)] by simp
hence B-eq-empty: \( B = \{\} \) using card-B unfolding card-eq-0-iff by simp
have \( A \subseteq \{\} \) using A-in-span-B unfolding B-eq-empty span-empty.
thus \( \text{?thesis} \) by blast
qed

lemma \( \text{dim-zero-eq}' \):

assumes \( A: A = \{\} \lor A = \{0\} \)
shows \( \text{dim} \ A = 0 \)
proof –
have card \( \{\}\::'b set \) = dim \( A \)
proof (rule basis-card-eq-dim[THEN conjunct2, of \( \{\}\::'b set \ A \)])
  show \( \{\} \subseteq A \) by simp
  show \( A \subseteq \text{span} \ \{\} \) using A by fastforce
  show independent \( \{\} \) by (rule independent-empty)
qed
thus \( \text{?thesis} \) by simp
qed

lemma \( \text{dim-zero-subspace-eq} \):

assumes subs-\( A \): subspace \( A \)
shows \( (\text{dim} \ A = 0) = (A = \{0\}) \) using \( \text{dim-zero-eq} \) \( \text{dim-zero-eq}' \) subspace-0[of \( \text{subs-}A \)] by auto

lemma span-0-imp-set-empty-or-0:
assumes span \( A = \{0\} \)
shows \( A = \{\} \lor A = \{0\} \) by (metis assms span-inc subset-singletonD)
end

context linear
begin
lemma linear-injective-ker-0:
shows inj \( f = (\{x. f \ x = 0\} = \{0\}) \)
unfolding linear-injective-0
using linear-0 by blast

end

lemma snd-if-conv:
shows snd (if P then (A,B) else (C,D)) = (if P then B else D) by simp

4.5 Basic properties about matrix multiplication

lemma row-matrix-matrix-mult:
fixes A :: 'a::{comm-ring-1}' 'n' 'm
shows (P $ i) v* A = (P ** A) $ i
unfolding vec-eq-iff unfolding vector-matrix-mult-def unfolding matrix-matrix-mult-def
by (auto intro: setsum.cong)
corollary row-matrix-matrix-mult':
fixes A :: 'a::{comm-ring-1}' 'n' 'm
shows (row i P) v* A = row i (P ** A)
using row-matrix-matrix-mult unfolding row-def vec-nth-inverse.

lemma column-matrix-matrix-mult:
shows column i (P**A) = P * v (column i A)
unfolding column-def matrix-vector-mult-def matrix-matrix-mult-def by fastforce

lemma matrix-matrix-mult-inner-mult:
shows (A**B) $ i $ j = row i A · column j B
unfolding inner-vec-def matrix-matrix-mult-def row-def column-def by auto

lemma matrix-vmult-column-sum:
fixes A :: 'a::{field}' 'n' 'm
shows ∃ f. (A * v x) = setsum (λ y. f y * s y) (columns A)
proof (rule exI[of - λ y. setsum (λ i. x $ i) {i. y = column i A}])
let ?f = λ y. setsum (λ i. x $ i) {i. y = column i A}
let ?g = λ y. {i. y = column i A}
have inj: inj-on ?g (columns A) unfolding inj-on-def unfolding columns-def
by auto
have union-univ: ⋃ {?g'(columns A)) = UNIV unfolding columns-def by auto
have A * v x = (∑ i∈UNIV. x $ i * s column i A) unfolding matrix-mult-vmult
also have ... = setsum (λ i. x $ i * s column i A) (⋃ (?g'(columns A))) unfolding union-univ ...
also have ... = setsum (setsum ((λ i. x $ i * s column i A))) (?g'(columns A)) by (rule setsum.Union-disjoint[unfolded o-def], auto)
also have ... = setsum ((setsum ((λ i. x $ i * s column i A))) o ?g) (columns A)
by (rule setsum.reindex, simp add: inj)
also have ... = setsum (λy. ?f y * y) (columns A)
proof (rule setsum.cong, unfold o-def)
  fix xa
  have setsum (λi. x $ i * s column i A) {i. xa = column i A} 
    = setsum (λi. x $ i * xa) {i. xa = column i A} by simp
  also have ... = setsum (λi. x $ i) {i. xa = column i A} * s xa
    using eec.scale-setsum-left[of (λi. x $ i) {i. xa = column i A} xa] ..
  finally show (∑i | xa = column i A. x $ i * s column i A) = (∑i | xa = 
    column i A. x $ i) * s xa .
qed rule
finally show A ** x = (∑y∈columns A. (∑i | y = column i A. x $ i) * s y) .
qed

4.6 Properties about invertibility

lemma matrix-inv:
  assumes invertible M
  shows matrix-inv-left: matrix-inv M ** M = mat 1 
  and matrix-inv-right: M ** matrix-inv M = mat 1 
  using (invertible M) and someI-ex [of λ N. M ** N = mat 1 ∧ N ** M = 
    mat 1] 
  unfolding invertible-def and matrix-inv-def
  by simp-all

lemma invertible-mult:
  assumes inv-A: invertible A 
  and inv-B: invertible B 
  shows invertible (A+B)
proof –
  obtain A’ where AA’: A ** A’ = mat 1 
    and A’A: A’ ** A = mat 1 
    using inv-A unfolding invertible-def by blast
  obtain B’ where BB’: B ** B’ = mat 1 
    and B’B: B’ ** B = mat 1 
    using inv-B unfolding invertible-def by blast
  show ?thesis
proof (unfold invertible-def, rule exI[of - B’**A’], rule conjI)
  have A ** B ** (B’ ** A’) = A ** (B ** (B’ ** A’))
    using matrix-mul-assoc[of A B (B’ ** A’), symmetric] .
  also have ... = A ** (B ** B’ ** A’) unfolding matrix-mul-assoc[of B B’ A’] ..
  also have ... = A ** (mat 1 ** A’) unfolding BB’ ..
  also have ... = A ** A’ unfolding matrix-mul-lid ..
  also have ... = mat 1 unfolding AA’ ..
  finally show A ** B ** (B’ ** A’) = mat (1::'a) .
  have B’**A’ ** (A ** B) = B’ ** (A’ ** (A ** B)) using matrix-mul-assoc[of 
    B’ A’ (A ** B), symmetric] .
  also have ... = B’ ** (A’ ** A ** B) unfolding matrix-mul-assoc[of A’ A B] ..
  also have ... = B’ ** (mat 1 ** B) unfolding A’A ..
also have ... = B′ ** B unfolding matrix-mul-lid ..
also have ... = mat 1 unfolding B'B ..
finally show B′ ** A′ ** (A ** B) = mat 1 .
qed
qed

In the library, matrix-inv ?A = (SOME A′. ?A ** A' = mat (1::'?a) ∧ A' ** ?A = mat (1::'?a)) allows the use of non square matrices. The following lemma can be also proved fixing A

lemma matrix-inv-unique:
  fixes A::'a::{semiring-1}°n°n
  assumes AB: A ** B = mat 1 and BA: B ** A = mat 1
  shows matrix-inv A = B
proof (unfold matrix-inv-def, rule some-equality)
  show A ** B = mat (1::'a) ∧ B ** A = mat (1::'a) using AB BA by simp
  fix C assume A ** C = mat (1::'a) ∧ C ** A = mat (1::'a)
  hence AC: A ** C = mat (1::'a) ∧ CA: C ** A = mat (1::'a) by auto
  have B = B ** (mat 1) unfolding matrix-mul-rid ..
  also have ... = B ** (A*C) unfolding AC ..
  also have ... = B ** A ** C unfolding matrix-mul-assoc ..
  also have ... = C unfolding BA matrix-mul-lid ..
  finally show C = B ..
qed

lemma matrix-vector-mult-zero-eq:
  assumes P: invertible P
  shows ((P**A)*v x = 0) = (A *v x = 0)
proof (rule iffI)
  assume P**A*v x = 0
  hence matrix-inv P *v (P ** A *v x) = matrix-inv P *v 0 by simp
  hence matrix-inv P *v (P ** A *v x) = 0 by (metis matrix-vector-zero)
  hence (matrix-inv P ** P ** A) *v x = 0 by (metis matrix-vector-mul-assoc)
  thus A *v x = 0 by (metis assms matrix-inv-left matrix-mul-lid)
next
  assume A *v x = 0
  thus P ** A *v x = 0 by (metis matrix-vector-mul-assoc matrix-vector-zero)
qed

lemma inj-matrix-vector-mult:
  fixes P::'a::{field}°n°m
  assumes P: invertible P
  shows inj (op *v P)
unfolding vec.linear-injective-0
using matrix-left-invertible-ker[of P] P unfolding invertible-def by blast

lemma independent-image-matrix-vector-mult:
  fixes P::'a::{field}°n°m
  assumes ind-B: vec.independent B and inv-P: invertible P
shows vec.independent ((op *v P) ' B)
proof (rule vec.independent-injective-on-span-image)
  show vec.independent B using ind-B .
  show inj-on (op *v P) (vec.span B)
    using inj-matrix-vector-mult[of inv-P] unfolding inj-on-def by simp
qed

lemma independent-preimage-matrix-vector-mult:
fixes P :: 'a::{field} 'n 
assumes ind-B: vec.independent ((op *v P) ' B) and inv-P: invertible P
shows vec.independent B
proof -
  have vec.independent ((op *v (matrix-inv P)) ' ((op *v P) ' B))
    proof (rule independent-image-matrix-vector-mult)
      show vec.independent (op *v P ' B) using ind-B .
      show invertible (matrix-inv P)
        by (metis matrix-inv-left matrix-vector-mul-assoc matrix-vector-mul-lid)
    qed
  moreover have (op *v (matrix-inv P)) ' ((op *v P) ' B) = B
    proof (auto)
      fix x assume x: x ∈ B show matrix-inv P *v (P *v x) ∈ B
        by (metis (full-types) x inv-P matrix-inv-left matrix-vector-mul-assoc matrix-vector-mul-lid)
      thus x ∈ op *v (matrix-inv P) ' op *v P ' B
    unfolding image-def
    by (auto, metis inv-P matrix-inv-left matrix-vector-mul-assoc matrix-vector-mul-lid)
  qed
ultimately show ?thesis by simp
qed

4.7 Properties about the dimension of vectors

lemma dimension-vector[code-unfold]: vec.dimension TYPE('a::{field}) TYPE('rows::{mod-type})=CARD('rows)
  proof -
  let ?f = λx. axis (from-nat x) 1::'a::{rows::{mod-type}}
  have vec.dimension TYPE('a::{field}) TYPE('rows::{mod-type}) = card (cart-basis::('a::{rows::{mod-type}})
    unfolding vec.dimension-def..
  also have ... = card[..<CARD('rows)]) unfolding cart-basis-def
    proof (rule bij-betw-same-card[symmetric, of ?f], unfold bij-betw-def, unfold inj-on-def axis-eq-axis, auto)
      fix x y assume x: x < CARD('rows) and y: y < CARD('rows) and eq: from-nat x = (from-nat y::'rows)
      show x = y using from-nat-eq-imp-eq[OF eq x y] .
      next
      fix i show axis i 1 ∈ (λx. axis (from-nat x::'rows) 1) ' \(..<CARD('rows))
    unfolding image-def
      by (auto, metis lessThan-iff to-nat-from-nat to-nat-less-card)
  qed
  also have ... = CARD('rows) by (metis card-lessThan)
finally show \(\text{thesis} \).
qed

### 4.8 Instantiations and interpretations

Functions between two real vector spaces form a real vector

**instantiation** fun :: \(\text{real-vector, real-vector}\) \(\text{real-vector}\)

begin

**definition** plus-fun \(f \, g = (\lambda \, i. \, f \, i + g \, i)\)

**definition** zero-fun = \((\lambda \, i. \, 0)\)

**definition** scaleR-fun \(a \, f = (\lambda \, i. \, a \ast_R f \, i)\)

instance proof
- fix \(a :: a \Rightarrow 'b \) and \(b :: a \Rightarrow 'b \) and \(c :: a \Rightarrow 'b \)
  - show \(a + b + c = a + (b + c)\) unfolding fun-eq-iff unfolding plus-fun-def by auto

- show \((\theta :: a \Rightarrow 'b) + a = a\) unfolding fun-eq-iff unfolding plus-fun-def zero-fun-def by auto
  - show \(- a + a = (\theta :: a \Rightarrow 'b)\) unfolding fun-eq-iff unfolding plus-fun-def zero-fun-def by auto
  - show \(a - b = a + - b\) unfolding fun-eq-iff unfolding plus-fun-def zero-fun-def by auto

next
- fix \(a :: \text{real}\) and \(x :: 'a \Rightarrow 'b\) and \(y :: 'a \Rightarrow 'b\)
  - show \(a \ast_R (x + y) = a \ast_R x + a \ast_R y\) unfolding fun-eq-iff plus-fun-def scaleR-fun-def scaleR-right.add by auto

next
- fix \(a :: \text{real}\) and \(b :: \text{real}\) and \(x :: 'a \Rightarrow 'b\)
  - show \((a + b) \ast_R x = a \ast_R x + b \ast_R x\) unfolding fun-eq-iff unfolding plus-fun-def scaleR-fun-def unfolding scaleR-left.add by auto
  - show \(a \ast_R b \ast_R x = (a \ast_R b) \ast_R x\) unfolding fun-eq-iff unfolding scaleR-fun-def by auto
  - show \((1 :: \text{real}) \ast_R x = x\) unfolding fun-eq-iff unfolding scaleR-fun-def by auto
qed

end

**instantiation** vec :: \((\text{type, finite})\) equal

begin

**definition** equal-vec :: \(\langle 'a, 'b::\text{finite}\rangle\) vec \(\Rightarrow\) \(\langle 'a, 'b::\text{finite}\rangle\) vec \(\Rightarrow\) \(\text{bool}\)
- where equal-vec \(x \, y = (\forall \, i. \, x
\, i = y \, i)\)

instance proof \((\text{intro-classes})\)
- fix \(x, y :: 'a, 'b::\text{finite}\) vec
  - show \(\text{equal-class.equal} \, x \, y = (x = y)\) unfolding equal-vec-def using vec-eq-iff by auto

78
qed

end

instantiation bit :: linorder

begin

definition less-eq-bit :: bit ⇒ bit ⇒ bool
  where less-eq-bit x y = (y=1 ∨ x=0)

definition less-bit :: bit ⇒ bit ⇒ bool
  where less-bit x y = (y=1 ∧ x=0)

instance
proof (intro-classes, auto simp add: less-eq-bit-def less-bit-def)
qed

end

interpretation matrix: vector-space (op *k):'a::{field}=>'a^'cols^'rows=>'a^'cols^'rows
proof (unfold-locales)
fix a::'a and x::'a^'cols^'rows
show a *k (x + y) = a *k x + a *k y
  unfolding matrix-scalar-mult-def vec-eq-iff
  by (simp add: vector-space-over-itself.scale-right-distrib)
next
fix a b::'a and x::'a^'cols^'rows
show (a + b) *k x = a *k x + b *k x
  unfolding matrix-scalar-mult-def vec-eq-iff
  by (simp add: comm-semiring-class.distrib)
show a *k (b *k x) = a * b *k x
  unfolding matrix-scalar-mult-def vec-eq-iff by auto
show 1 *k x = x unfolding matrix-scalar-mult-def vec-eq-iff by auto
qed


4.9 Properties about lists

The following definitions and theorems are developed in order to compute
setprods. More theorems and properties can be demonstrated in a similar
way to the ones about listsum.

definition (in monoid-mult) listprod :: 'a list ⇒ 'a where
  listprod xs = foldr times xs 1

lemma (in monoid-mult) listprod-simps [simp]:
  listprod [] = 1
  listprod (x # xs) = x * listprod xs
  by (simp-all add: listprod-def)

lemma (in monoid-mult) listprod-append [simp]:
  listprod (xs @ ys) = listprod xs * listprod ys
  by (induct xs) (simp-all add: mult.assoc)
lemma (in comm-monoid-mult) listprod-rev [simp]:
  listprod (rev xs) = listprod xs
by (simp add: listprod-def foldr-fold fold-rev fun-eq-iff ac-simps)

lemma (in monoid-mult) listprod-distinct-conv-setprod-set:
  distinct xs ==> listprod (map f xs) = setprod f (set xs)
by (induct xs) simp-all

lemma setprod-code [code]:
  setprod f (set xs) = listprod (map f (remdups xs))
by (simp add: listprod-distinct-conv-setprod-set)

end

5 Fundamental Subspaces

theory Fundamental-Subspaces
imports
  ~~/src/HOL/Multivariate-Analysis/Multivariate-Analysis
  Miscellaneous
begin

5.1 The fundamental subspaces of a matrix

5.1.1 Definitions

definition left-null-space :: 'a::{semiring-1} ^'n^'m => ('a ^'m) set
  where left-null-space A = {x. x v* A = 0}
definition null-space :: 'a::{semiring-1} ^'n^'m => ('a ^'n) set
  where null-space A = {x. A *v x = 0}
definition row-space :: 'a::{field} ^'n^'m => ('a ^'n) set
  where row-space A = vec.span (rows A)
definition col-space :: 'a::{field} ^'n^'m => ('a ^'m) set
  where col-space A = vec.span (columns A)

5.1.2 Relationships among them

lemma left-null-space-eq-null-space-transpose: left-null-space A = null-space (transpose A)
  unfolding null-space-def left-null-space-def transpose-vector ..

lemma null-space-eq-left-null-space-transpose: null-space A = left-null-space (transpose A)
  using left-null-space-eq-null-space-transpose[of transpose A]
  unfolding transpose-transpose ..
lemma row-space-eq-col-space-transpose:
fixes A :: 'a::{field} "columns" "rows"
shows row-space A = col-space (transpose A)
unfolding col-space-def row-space-def columns-transpose[of A] ..

lemma col-space-eq-row-space-transpose:
fixes A :: 'a::{field} "n" "m"
shows col-space A = row-space (transpose A)
unfolding col-space-def row-space-def unfolding rows-transpose[of A] ..

5.2 Proving that they are subspaces

lemma subspace-null-space:
fixes A :: 'a::{field} "n" "m"
shows vec.subspace (null-space A)
proof (unfold vec.subspace-def null-space-def, auto)
show A *v 0 = 0 by (metis add-diff-cancel eq_iff_diff_eq_0 matrix-vector-right-distrib)

fix x y
assume Ax: A *v x = 0 and Ay: A *v y = 0
have A *v (x + y) = (A *v x) + (A *v y) unfolding matrix-vector-right-distrib ..
also have ... = 0 unfolding Ax Ay by simp
finally show A *v (x + y) = 0 .
fix c
have A *v (c *s x) = c *s (A *v x)
  unfolding scalar-matrix-vector-assoc matrix-scalar-vector-ac by auto
also have ... = 0 unfolding Ax by simp
finally show A *v (c *s x) = 0 .
qed

lemma subspace-left-null-space:
fixes A :: 'a::{field} "n" "m"
shows vec.subspace (left-null-space A)
unfolding left-null-space-eq-null-space-transpose unfolding subspace-null-space .

lemma subspace-row-space:
shows vec.subspace (row-space A) by (metis row-space-def vec.subspace-span)

lemma subspace-col-space:
shows vec.subspace (col-space A) by (metis col-space-def vec.subspace-span)

5.3 More useful properties and equivalences

lemma col-space-eq:
fixes A :: 'a::{field} "m:: {finite, wellorder}" "n"
shows col-space A = { y, \exists x. A *v x = y }
proof (unfold col-space-def vec.span-finite[of finite-columns], auto)
fix x
show \( \exists u. (\sum v \in \text{columns } A. \ u \ v \ * s \ v) = A \ * v \ x \) using \( \text{matrix-vmult-column-sum[of A x]} \) by auto

next

fix \( a::('a, 'n) \text{ vec} \Rightarrow 'a \)

let \( \Rightarrow g=\lambda y. \{ i. \ y=\text{column } i \ A \} \)

let \( \Rightarrow x=x(i. \ i=(\text{LEAST } a. \ a \in \Rightarrow g \ (\text{column } i \ A)) \ then \ u \ (\text{column } i \ A) \ else \ 0) \)

show \( \exists x. A \ * v \ x = (\sum v \in \text{columns } A. \ u \ v \ * s \ v) \)

proof (unfold \( \text{matrix-vmult-vec, rule exI[of - ?x]} \), auto)

have inj: inj-on \( \Rightarrow g \ (\text{columns } A) \) unfolding inj-on-def unfolding columns-def by auto

have union-univ: \( \bigcup(\Rightarrow g'(\text{columns } A)) = \text{UNIV} \) unfolding columns-def by auto

have setsum (\( \lambda i. \) (if \( i = (\text{LEAST } a. \ \text{column } i \ A = \text{column } a \ A) \) then \( u \ (\text{column } i \ A) \ else \ 0) \) \* s \ column i A) \UNIV

= setsum (\( \lambda i. \) (if \( i = (\text{LEAST } a. \ \text{column } i \ A = \text{column } a \ A) \) then \( u \ (\text{column } i \ A) \ else \ 0) \) \* s \ column i A) \( \bigcup(\Rightarrow g'(\text{columns } A)) \)

unfolding union-univ ..

also have ... = setsum (setsum (\( \lambda i. \) (if \( i = (\text{LEAST } a. \ \text{column } i \ A = \text{column } a \ A) \) then \( u \ (\text{column } i \ A) \ else \ 0) \) \* s \ column i A) (\( \Rightarrow g'(\text{columns } A)) \)

by (rule setsum.Union-disjoint[unfolded o-def], auto)

also have ... = setsum ((setsum (\( \lambda i. \) (if \( i = (\text{LEAST } a. \ \text{column } i \ A = \text{column } a \ A) \) then \( u \ (\text{column } i \ A) \ else \ 0) \) \* s \ column i A)) o \( \Rightarrow g) \)

(columns A) by (rule setsum.reindex, simp add: inj)

also have ... = setsum (\( \lambda y. \ u \ y \ * s \ y) \ (\text{columns } A) \)

proof (rule setsum.cong, auto)

fix \( x \)

assume x-in-cols: \( x \in \text{columns } A \)

obtain \( b \) where \( b:=\text{column } b \ A \) using x-in-cols unfolding columns-def by blast

let \( \Rightarrow f=(\lambda i. \) (if \( i = (\text{LEAST } a. \ \text{column } i \ A = \text{column } a \ A) \) then \( u \ (\text{column } i \ A) \ else \ 0) \) \* s \ column i A) \)

have setsum-rw: setsum \( \Rightarrow f \ (\{ i. \ x = \text{column } i \ A \} - \{ \text{LEAST } a. \ x = \text{column } a \ A \}) = 0 \)

by (rule setsum.neutral, auto)

have setsum \( \Rightarrow f \ (\{ i. \ x = \text{column } i \ A \} - \{ \text{LEAST } a. \ x = \text{column } a \ A \}) + \)

setsum \( \Rightarrow f \ (\{ i. \ x = \text{column } i \ A \} - \{ \text{LEAST } a. \ x = \text{column } a \ A \}) \)

apply (rule setsum.remove, auto, rule LeastI-ex)

using x-in-cols unfolding columns-def by auto

also have ... = \( \Rightarrow f \ (\text{LEAST } a. \ x = \text{column } a \ A) \) unfolding setsum-rw by simp

also have ... = \( u \ x \ * s \ x \)

proof (auto, rule LeastI2)

show \( x = \text{column } b \ A \) using \( b \).

fix \( x a \)

assume \( x: x = \text{column } xa \ A \)

show \( u \ (\text{column } xa \ A) \ * s \ column xa A = u \ x \ * s \ x \) unfolding x ..

next

assume \( (\text{LEAST } a. \ x = \text{column } a \ A) \neq (\text{LEAST } a. \ \text{column } (\text{LEAST } c. \ x = \text{column } c \ A) \ A = \text{column } a \ A) \)

82
moreover have \((\text{LEAST } a \cdot x = \text{column } a \ A) = (\text{LEAST } a \cdot \text{column } c \ A = \text{column } a \ A)\)

by (rule Least-equality[symmetric], rule LeastI2, simp-all add: b, rule Least-le, metis (lifting, full-types) LeastI)

ultimately show \(u \ x = 0\) by contradiction

qed

finally show \(\left(\sum_{i \in\text{UNIV}} (\text{if } i = (\text{LEAST } a \cdot \text{column } i \ A = \text{column } a \ A) \text{ then } u \ (\text{column } i \ A) \text{ else } 0) \ast s \text{ column } i \ A\right) = \left(\sum y \in\text{columns } A \ u \ y \ast s \ y\right)\).

qed

qed

corollary col-space-eq':
fixes A::'a::{field} ⇒ 'm::{finite, wellorder} ⇒ 'n
shows col-space A = range (λx. A *v x)
unfolding col-space-eq by auto

lemma row-space-eq:
fixes A::'a::{field} ⇒ 'm::'n
shows row-space A = {w. ∃y. (transpose A) *v y = w}
unfolding row-space-eq-col-space-transpose col-space-eq ..

lemma null-space-eq-ker:
fixes f::('a::{field} ⇒ 'm) ⇒ 'n
assumes lf: linear op *s op *s f
shows null-space (matrix f) = {x. f x = 0}
unfolding null-space-def using matrix-works [OF lf] by auto

lemma col-space-eq-range:
fixes f::('a::{field} ⇒ 'm::{finite, wellorder}) ⇒ 'n
assumes lf: linear op *s op *s f
shows col-space (matrix f) = range f
unfolding col-space-eq unfolding matrix-works[OF lf] by blast

lemma null-space-is-preserved:
fixes A::'a::{field} ⇒ 'cols⇒'rows
assumes P: invertible P
shows null-space (P**A) = null-space A
unfolding null-space-def
using P matrix-inv-left matrix-left-invertible-ker matrix-vector-mul-assoc matrix-vector-zero
by metis

lemma row-space-is-preserved:
fixes A::'a::{field} ⇒ 'cols⇒'rows::{finite, wellorder}
and P::'a::{field} ⇒ 'rows::{finite, wellorder} ⇒ 'rows::{finite, wellorder}
assumes P: invertible P
shows row-space \((P^{**}A) = row-space A\)
proof (auto)
fix \(w\)
assume \(w \in row-space (P^{**}A)\)
from this obtain \(y\) where \(w-By\): \(w = (\text{transpose} (P^{**}A)) \ast v y\)
  unfolding row-space-eq[of \(P \ast P\)] by fast
have \(w = (\text{transpose} (P^{**}A)) \ast v y\) using \(w-By\).
also have \(\ldots = (\text{transpose} A) \ast (\text{transpose} (P)) \ast v y\) unfolding matrix-transpose-mul
.. also have \(\ldots = (\text{transpose} A) \ast v ((\text{transpose} P) \ast v y)\) unfolding matrix-vector-mul-assoc
.. finally show \(w \in row-space A\) unfolding row-space-eq by blast
next
fix \(w\)
assume \(w \in row-space A\)
from this obtain \(y\) where \(w-Ay\): \(w = (\text{transpose} A) \ast v y\)
  unfolding row-space-eq by fast
have \(w = (\text{transpose} A) \ast v y\) using \(w-Ay\).
also have \(\ldots = (\text{transpose} ((\text{matrix-inv} P) \ast P^{**}A)) \ast v y\)
  by (metis \(P\) matrix-inv-left matrix-mul-assoc matrix-mul-id)
also have \(\ldots = (\text{transpose} (P^{**}A)) \ast (\text{transpose} (\text{matrix-inv} P)) \ast v y\)
  unfolding matrix-transpose-mul ..
also have \(\ldots = \text{transpose} (P^{**}A) \ast v (\text{transpose} (\text{matrix-inv} P) \ast v y)\)
  unfolding matrix-vector-mul-assoc ..
finally show \(w \in row-space (P^{**}A)\) unfolding row-space-eq by blast
qed
end

6 Rank Nullity Theorem of Linear Algebra

theory Dim-Formula
imports Fundamental-Subspaces
begin

context vector-space
begin

6.1 Previous results

Linear dependency is a monotone property, based on the monotonocity of linear independence:
lemma dependent-mono:
  assumes \(d: \text{dependent} A\)
  and \(A \subseteq B\)
  shows \(d: \text{dependent} B\)
  using dependent-mono [OF - \(A \subseteq B\)] \(d\) by auto

Given a finite independent set, a linear combination of its elements equal to
zero is possible only if every coefficient is zero:

**lemma scalars-zero-if-independent:**
- **assumes** fin-$A$: finite $A$
- and ind: independent $A$
- and sum: $(\sum x \in A. \text{scale}(f \times x)) = 0$
- **shows** $\forall x \in A. f \times x = 0$
- using assms unfolding independent-explicit by auto

**end**

**context finite-dimensional-vector-space**

**begin**

In an finite dimensional vector space, every independent set is finite, and thus

\[
[[\text{finite } A; \text{local.independent } A; (\sum x \in A. \text{scale}(f \times x)) = (0::'b)] = \Rightarrow \forall x \in A. f \times x = (0::'a)]
\]

holds:

**corollary scalars-zero-if-independent-euclidean:**
- **assumes** ind: independent $A$
- and sum: $(\sum x \in A. \text{scale}(f \times x)) = 0$
- **shows** $\forall x \in A. f \times x = 0$
- **by** (rule scalars-zero-if-independent,
  rule conjunct1 [OF independent-bound [OF ind]])
  (rule ind, rule sum)

**end**

The following lemma states that every linear form is injective over the elements which define the basis of the range of the linear form. This property is applied later over the elements of an arbitrary basis which are not in the basis of the nullifier or kernel set (i.e., the candidates to be the basis of the range space of the linear form).

Thanks to this result, it can be concluded that the cardinal of the elements of a basis which do not belong to the kernel of a linear form $f$ is equal to the cardinal of the set obtained when applying $f$ to such elements.

The application of this lemma is not usually found in the pencil and paper proofs of the “rank nullity theorem”, but will be crucial to know that, being $f$ a linear form from a finite dimensional vector space $V$ to a vector space $V'$, and given a basis $B$ of $\ker f$, when $B$ is completed up to a basis of $V$ with a set $W$, the cardinal of this set is equal to the cardinal of its range set:

**context vector-space**

**begin**

**lemma inj-on-extended:**
assumes If: linear scaleB scaleC f
and f: finite C
and ind-C: independent C
and C-eq: C = B ∪ W
and disj-set: B ∩ W = {} 
and span-B: \{x. f x = 0\} ⊆ span B
shows inj-on f W
— The proof is carried out by reductio ad absurdum

proof (unfold inj-on-def, rule+, rule ccontr)
interpret If: linear scaleB scaleC f using If by simp
— Some previous consequences of the premises that are used later:

have fin-B: finite B using finite-subset [OF - f] C-eq by simp
have ind-B: independent B and ind-W: independent W

using independent-mono [OF ind-C] C-eq by simp-all
— The proof starts here; we assume that there exist two different elements
— with the same image:

fix x::'b and y::'b
assume x: x ∈ W and y: y ∈ W and f-eq: f x = f y and x-not-y: x ≠ y
have fin-yB: finite (insert y B) using fin-B by simp
have f (x - y) = 0 by (metis diff-self f-eq If.linear-sub)
hence x - y ∈ {x. f x = 0} by simp
hence ∃ g. (∑ v∈B. scale (g v) v) = (x - y) using span-B

unfolding span-finite [OF fin-B] by auto
then obtain g where \(\sum v∈B. \text{scale} (g v) v = (x - y)\) by blast
— We define one of the elements as a linear combination of the second element
and the ones in B

def h ≡ (λa. if a = y then (1::'a) else g a)

have x = y + (∑ v∈B. scale (g v) v) using sum by auto
also have \(\ldots = \text{scale} (h y) y + (\sum v∈B. \text{scale} (g v) v)\) unfolding h-def by simp
also have \(\ldots = \text{scale} (h y) y + (\sum v∈B. \text{scale} (h v) v)\)

apply (unfold add-left-cancel, rule setsum.cong)
using y h-def empty-iff disj-set by auto
also have \(\ldots = (∑ v∈\text{insert} y B). \text{scale} (h v) v\)
by (rule setsum.insert[symmetric], rule fin-B)

(metis (lifting) IntI disj-set empty-iff y)
finally have x-in-span-yB: x ∈ span (insert y B)

unfolding span-finite [OF fin-yB] by auto
— We have that a subset of elements of C is linearly dependent

have dep: dependent (insert x (insert y B))

by (unfold dependent-def, rule bexI [of - x])

(metis Diff-insert-absorb Int-iff disj-set empty-iff insert-iff
x x-in-span-yB x-not-y, simp)
— Therefore, the set C is also dependent:

hence dependent C using C-eq x y
by (metis Un-commute Un-upper2 dependent-mono insert-absorb insert-subset)
— This yields the contradiction, since C is independent:
thus False using ind-C by contradiction
qed
6.2 The proof

Now the rank nullity theorem can be proved; given any linear form \( f \), the sum of the dimensions of its kernel and range subspaces is equal to the dimension of the source vector space.

The statement of the “rank nullity theorem for linear algebra”, as well as its proof, follow the ones on [1]. The proof is the traditional one found in the literature. The theorem is also named “fundamental theorem of linear algebra” in some texts (for instance, in [2]).

**context linear-first-finite-dimensional-vector-space**

begin

**theorem rank-nullity-theorem:**
- **shows** \( B.\text{dimension} = B.\text{dim} \{ x. \ f x = 0 \} + C.\text{dim} (\text{range } f) \)
- **proof**
  - **have** \( l: \text{linear scale } B \text{ scale } C \ f \) by unfold-locales
    - For convenience we define abbreviations for the universe set, \( V \):
      - **def** \( V == \text{UNIV}::'b \text{ set} \)
      - **def** \( \ker f == \{ x. \ f x = 0 \} \)
        - The kernel is a proper subspace:
          - **have** \( \ker \text{subspace} \{ x. \ f x = 0 \} \) using \( B.\text{subspace-kernel}[OF \ l] \).
        - The kernel has its proper basis, \( B \):
          - **obtain** \( B \) where \( B\text{-in-ker}: B \subseteq \{ x. \ f x = 0 \} \) and \( B.\text{independent } B \)
          - **obtain** \( C \) where \( B\text{-in-C}: B \subseteq C \) and \( C\text{-in-V}: C \subseteq V \)
          - **using** \( B.\text{maximal-independent-subset-extend} [OF - independent-B, of V] \)
          - The basis of \( V, C \), can be decomposed in the disjoint union of the basis of the kernel, \( B \), and its complementary set, \( C - B \):
            - **have** \( C\text{-eq}: C = B \cup (C - B) \) by (rule \( \text{Diff-partition} [OF B\text{-in-C}, \text{symmetric}] \))
            - **have** \( eq\text{-fC}: f \ C = f \ B \cup f \ (C - B) \) by (subst \( C\text{-eq}, \text{unfold image-Un, simp} \))
          - The basis \( C \), and its image, are finite, since \( V \) is finite-dimensional
            - **have** \( \text{finite-C}: \text{finite } C \)
              - **using** \( B.\text{independent-bound-general} [OF \text{independent-C}] \) by fast
            - **have** \( \text{finite-fC}: \text{finite } (f \ C) \) by (rule \( \text{finite-imageI} [OF \text{finite-C}] \))
              - The basis \( B \) of the kernel of \( f \), and its image, are also finite
            - **have** \( \text{finite-B}: \text{finite } B \) by (rule \( \text{rev-finite-subset} [OF \text{finite-C} B\text{-in-C}] \))
            - **have** \( \text{finite-fB}: \text{finite } (f \ B) \) by (rule \( \text{finite-imageI}[OF \text{finite-B}] \))
— The set $C - B$ is also finite

**have** finite-CB: finite $(C - B)$ **by** (rule finite-Diff [OF finite-C, of B])

**have** dim-ker-le-dim-V.B.dim (ker-f) $\leq$ B.dim V **using** B.dim-subset [of ker-f V] unfolding V-def **by** simp

— Here it starts the proof of the theorem: the sets $B$ and $C - B$ must be proven to be bases, respectively, of the kernel of $f$ and its range

**show** ?thesis

**proof**

**have** B.dimension = B.dim V unfolding V-def dim-UNIV dimension-def **by** (metis B.basis-card-eq-dim B.dimension-def B.independent-Basis B.span-Basis top-greatest)

**also have** B.dim V = B.dim C unfolding span-C B.dim-span ...

**also have** ... = card C **using** B.basis-card-eq-dim [of C C, OF - B.span-inc independent-C] **by** simp

**also have** ...

— Now it has to be proved that the elements of $C - B$ are a basis of the range of $f$

**also have** ...

**proof** (unfold add-left-cancel)

**def** W ::= C' - B

**have** finite-W: finite W unfolding W-def using finite-CB .

**have** finite-fW: finite $(f \cdot W)$ **using** finite-imageI [OF finite-W] .

**have** card W = card $(f \cdot W)$ **by** (rule card-Un-disjoint[OF finite-B finite-CB], fast)

**also have** ...

— Given any element $v$ in $V$, its image can be expressed as a linear combination of elements of the image by $f$ of $C$:

**fix** v :: 'b

**have** fV-span: $f \cdot V \subseteq C.span (f \cdot C)$ **using** B.spans-image [OF l] span-C **by** simp

**have** $\exists g. (\sum x \in f \cdot C. scaleC (g x) x) = f v$ **using** fV-span unfolding V-def **using** l.C.span-finite [OF finite-fW], auto

**then obtain** g where $f u = (\sum x \in f \cdot C. scaleC (g x) x)$ **by** metis

— We recall that $C$ is equal to $B$ union $(C - B)$, and $B$ is the basis of the kernel; thus, the image of the elements of $B$ will be equal to zero:

**have** zero-fB: $(\sum x \in f \cdot B. scaleC (g x) x) = 0$
using B-in-ker by (auto intro!: setsum.neutral)
have zero-inter: \((\sum x \in (f \cdot B \cap f \cdot W). \ scaleC (g x) x) = 0\)
using B-in-ker by (auto intro!: setsum.neutral)
have \(f v = (\sum x \in f \cdot C. \ scaleC (g x) x)\) using \(f v\).
also have \(\vdash (\sum x \in (f \cdot B \cup f \cdot W). \ scaleC (g x) x)\)
using \(eq-fC \ W-def\) by simp
also have \(\vdash (\sum x \in f \cdot B. \ scaleC (g x) x) + (\sum x \in f \cdot W. \ scaleC (g x) x)\)
using \(setsum-Un\) [OF finite-fB finite-fW] by simp
also have \(\vdash (\sum x \in f \cdot W. \ scaleC (g x) x)\)
unfolding zero-fB zero-inter by simp

— We have proved that the image set of \(W\) is a generating set of the
range of \(f\)
finally show \(\exists s. \ (\sum x \in f \cdot W. \ scaleC (s x) x) = f v\) by auto
qed

— 2. The image set of \(W\) is linearly independent:
have independent-fW: \(l.C.independent (f \cdot W)\)
proof (rule \(l.C.independent-if-scalars-zero\) [OF finite-fW], rule+)
— Every linear combination (given by \(g x\)) of the elements of the image set
of \(W\) equal to zero, requires every coefficient to be zero:
fix \(g :: 'c => 'a and w :: 'c\)
assume sum: \((\sum x \in f \cdot W. \ scaleC (g x) x) = 0\) and \(w: w \in f \cdot W\)
have \(\theta = (\sum x \in f \cdot W. \ scaleC (g x) x)\) using sum by simp
also have \(\vdash \setsum ((\lambda x. \ scaleC (g x) x) \cdot f) \ W\)
by (rule setsum.reindex, rule B.inj-on-extended[OF \(l\), of \(C B\)])
(unfold W-def, rule \(finite-C\), rule independent-C, rule C-eq, simp,
rule ker-in-span)
also have \(\vdash (\sum x \in W. \ scaleC ((g \circ f) x) (f x))\) unfolding o-def ..
also have \(\vdash f (\sum x \in W. \ scaleB ((g \circ f) x) x)\)
using l.linear-setsum-mail [symmetric, OF finite-fW] .
finally have \(f\)-sum-zero-f: \((\sum x \in W. \ scaleB ((g \circ f) x) x) = 0\) by (rule sym)

hence \((\sum x \in W. \ scaleB ((g \circ f) x) x) \in \ker-f\) unfolding ker-f-def by simp
hence \(\exists h. \ (\sum x \in W. \ scaleB ((g \circ f) x) x) = (\sum x \in W. \ scaleB ((g \circ f) x) x)\)
using B.span-finite[OF finite-B] unfolding ker-in-span
unfolding ker-f-def by auto
then obtain \(h\) where
sum-h: \((\sum v \in B. \ scaleB (h v) v) = (\sum x \in W. \ scaleB ((g \circ f) x) x)\) by blast

def \(t \equiv (\lambda a. \ if a \in B \ then h a \ else -((g \circ f) a))\)
have \(0 = (\sum v \in B. \ scaleB (h v) v) + - (\sum x \in W. \ scaleB ((g \circ f) x) x)\)
using sum-h by simp
also have \(\vdash (\sum v \in B. \ scaleB (h v) v) + (\sum x \in W. - (scaleB ((g \circ f) x) x))\)
unfolding setsum-negf ..
also have \(\vdash (\sum v \in B. \ scaleB (t v) v) + (\sum x \in W. -(scaleB((g \circ f) x) x))\)
unfolding add-right-cancel unfolding t-def by simp
also have \( \ldots = (\sum_{v \in B}. \scaleB (t \, v)) + (\sum_{x \in W}. \scaleB (t \, x)) \)
by (unfold add-left-cancel t-def W-def, rule setsum.cong) simp+
also have \( \ldots = (\sum_{v \in B \cup W}. \scaleB (t \, v)) \)
by (rule setsum.union-inter-neutral [symmetric], rule finite-B, rule finite-W)

\[
(\text{simp add: W-def})
\]
finally have \( (\sum_{v \in B \cup W}. \scaleB (t \, v)) = 0 \) by simp
hence \( \text{coef-zero: } \forall x \in B \cup W. \, t \, x = 0 \)
using C-eq B.scalars-zero-if-independent [OF finite-C independent-C]
unfolding W-def by simp
obtain y where w-fy: \( w = f \, y \) and y-in-W: \( y \in W \) using w
by fast
unfolding t-def w-fy using y-in-W unfolding W-def by simp
also have \( \ldots = 0 \) using coef-zero y-in-W
unfolding W-def by simp
finally show \( g \, w = 0 \) by simp
qed

— The image set of \( W \) is independent and its span contains the range of \( f \), so it is a basis of the range:
show \( \exists B \subseteq \text{range } f. \neg \text{vector-space. dependent scaleC } B \)
\( \land \text{range } f \subseteq \text{vector-space. span scaleC } B \lor \text{card } B = \text{card } (f' \, W) \)
by (rule exI [of -(f' \, W)],
  simp add: range-in-span-fW independent-fW image-mono)
— Now, it has to be proved that any other basis of the subspace range of \( f \) has equal cardinality:
show \( \forall n :: \text{nat}. \exists B \subseteq \text{range } f. \text{l.C.independent } B \land \text{range } f \subseteq \text{l.C.span } B \land \text{card } B = n \)
\( \Rightarrow \text{card } (f' \, W) = n \)
proof (clarify)
fix \( S :: 'c \ \text{set} \)
assume S-in-range: \( S \subseteq \text{range } f \) and independent-S: \( \text{vector-space.independent } S \) scaleC S
  and range-in-spanS: \( \text{range } f \subseteq \text{vector-space.span scaleC } S \)
  have S-le: \( \text{finite } S \land \text{card } S \leq \text{card } (f' \, W) \)
by (rule l.C.independent-span-bound [OF finite-fW independent-S])
(rule subset-trans [OF S-in-range range-in-span-fW])
show \( \text{card } (f' \, W) = \text{card } S \)
by (rule le-antisym, rule conjunct2, rule l.C.independent-span-bound)
(rule conjunct1 [OF S-le], rule independent-fW,
  rule subset-trans [OF - range-in-spanS], auto simp add: S-le)
qed

finally show \( \text{card } (C - B) = C.\dim (\text{range } f) \) unfolding W-def .
qed
finally show \( \text{thesis unfolding V-def ker-f-def unfolding dim-UNIV} \).
qed

end
6.3 The rank nullity theorem for matrices

The proof of the theorem for matrices is direct, as a consequence of the “rank nullity theorem”.

**lemma** rank-nullity-theorem-matrices:
- **fixes** \( A::\{\text{field}\} \times\{\text{finite, wellorder}\} \times\{\text{rows}\} \)
- **shows** \( \text{ncols} \ A = \text{vec.dim} \ (\text{null-space} \ A) + \text{vec.dim} \ (\text{col-space} \ A) \)
- **proof** —
  - **show** \( \text{thesis} \)
    - **apply** \( \text{subst} \ (1 \ 2) \ \text{matrix-of-matrix-vector-mul} \ [\text{of} \ A, \text{symmetric}] \)
    - **unfolding** \( \text{null-space-eq-ker} \ [\text{OF} \ \text{matrix-vector-mul-linear}] \)
    - **unfolding** \( \text{col-space-eq-range} \ [\text{OF} \ \text{matrix-vector-mul-linear}] \)
    - **using** \( \text{vec.rank-nullity-theorem} \)
    - **by** (\( \text{metis} \ \text{col-space-eq} \ \text{ncols-def} \ \text{vec.dim-UNIV} \ \text{vec.dimension-def} \ \text{vec-dim-card} \))
  - **qed**
- **end**

**References**
