Rank-Nullity Theorem in Linear Algebra

By Jose Divasón and Jesús Aransay*

May 28, 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.
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 `linordered-idom`, such as the lemma `det-identical-columns`, are now proven using the class `comm-ring-1`.

```plaintext
hide-const (open) span
hide-const (open) dependent
hide-const (open) independent
hide-const (open) dim

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

locale linear = 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) +
fixes f :: ('b=>'c)
assumes cmult: f (r *b x) = r *c (f x)
and add: f (a + b) = f a + f b
begin

lemma linear-0: f 0 = 0
by (metis add eq-add-iff)

lemma linear-cmul: f (c *b x) = c *c (f x)
by (metis cmult)

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

lemma linear-add: f (x + y) = f x + f y
by (metis add)

lemma linear-sub: f (x - y) = f x - f y
by (metis diff-conv-add-uminus linear-add linear-neg)

lemma linear-setsum:
assumes fin: finite S
shows f (setsum g S) = setsum (f o g) S
using fin
proof induct
```
case empty
  then show \( ?\)case
    by (simp add: linear-0)
next
case (insert \( x F \))
  have \( f \left( \text{setsum} \ g \ (\ \text{insert} \ x \ \ F) \right) = f \left( g \ x + \text{setsum} \ g \ F \right) \)
    using insert.hyps by simp
  also have \( \ldots = f \ (g \ x) + f \ (\text{setsum} \ g \ F) \)
    using linear-add by simp
  also have \( \ldots = \text{setsum} \ (f \circ g) \ (\ \text{insert} \ x \ \ F) \)
    using insert.hyps by simp
finally show \( ?\)case .
qed

lemma linear-setsum-mul:
  assumes \( \text{fin} : \text{finite} \ S \)
  shows \( f \left( \text{setsum} \ (\lambda i. \ c \ i \ \ast \ b \ v \ i) \ S \right) = \text{setsum} \ (\lambda i. \ c \ i \ \ast \ (f \ v \ i)) \ S \)
  using linear-setsum[OF \( \text{fin} \)] linear-cmul
  by simp

lemma linear-injective-0:
  shows inj \( f \leftarrow \forall x. f \ x = 0 \rightarrow x = 0 \)
proof –
  have inj \( f \leftarrow \forall x y. f \ x = f \ y \rightarrow x = y \)
    by (simp add: inj-on-def)
  also have \( \ldots \leftarrow \forall x y. f \ x - f \ y = 0 \rightarrow x - y = 0 \)
    by simp
  also have \( \ldots \leftarrow \forall x y. f \ (x - y) = 0 \rightarrow x - y = 0 \)
    by (simp add: linear-sub)
  also have \( \ldots \leftarrow \forall x. f \ x = 0 \rightarrow x = 0 \)
    by auto
  finally show \( ?\)thesis .
qed
end

lemma linear-iff:
  linear scaleB scaleC \( f \leftarrow \text{vector-space} \ \text{scaleB} \) \& \text{vector-space} \ scaleC
  \& (\forall x y. f \ (x + y) = f \ x + f \ y) \& (\forall c x. f \ (\text{scaleB} \ c \ x) = \text{scaleC} \ c \ (f \ x))
  (is linear scaleB scaleC \( f \leftarrow ?\text{rhs} \))
proof
  assume \( \text{lf} : \text{linear} \ \text{scaleB} \ \text{scaleC} \ \text{f then interpret} \ f : \text{linear} \ \text{scaleB} \ \text{scaleC} \ \text{f} . \)
  have \( B : \text{vector-space} \ \text{scaleB} \) using \( \text{if unfolding} \ \text{linear-def} \) by simp
  moreover have \( C : \text{vector-space} \ \text{scaleC} \) using \( \text{if unfolding} \ \text{linear-def} \) by simp
  ultimately show \( ?\text{rhs} \) using \( f.\text{linear-add} f.\text{linear-cmul} \) by simp
next
  assume \( ?\text{rhs} \) 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 \iff (\forall x y. f (x + y) = f x + f y) \land (\forall c x. f (c *s x) = c *s (f x))
(is linear (op *s) (op *s) f \iff ?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 \implies linear scale scaleC g \implies linear scale scaleC (\lambda x. f x - g x)
unfolding linear-iff by (simp add: vector-space.scale-right-diff-distrib)

lemma linear-compose: linear scale scaleC f \implies linear scaleC scaleT g \implies 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 \Rightarrow bool
where subspace S \iff 0 \in S \land (\forall x\in S. \forall y\in S. x + y \in S) \land (\forall c. \forall x \in S. scale c x \in S )
definition span (S::'b set) = (subspace hull S)
definition dependent S \iff (\exists a \in S. a \in span (S - {a}))
abbreviation independent s \equiv \neg dependent s

Closure properties of subspaces.

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

lemma subspace-0: subspace S \implies 0 \in 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: subspace A
and fB: 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[of fB])
(simp add: subspace-def sA, auto simp add: sA subspace-add)

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

lemma subspace-linear-vimage:
assumes lf: linear scale scaleC (f::'b::ab-group-add=>'c::ab-group-add)
and s: vector-space.\text{subspace } scaleC S
shows subspace (f - ' S)
proof
- interpret lf: linear scale scaleC f using lf by simp
have C: vector-space scaleC using lf (unfold-locales)
show "thesis"
  unfolding subspace-def
  apply (auto)
  apply (metis C.subspace-0 C.linear-0 s)
  apply (metis (full-types) C.subspace-def C.linear-add s)
  by (metis full-types C.subspace-def C.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
  θ ∈ 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: $S \subseteq T \implies \text{subspace } T \implies \text{span } S \subseteq T$

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 $S=\text{subspace}$, $OF SP'$, unfolded span-def[symmetric]]
show $x \in P$
  by (metis subset-eq)
qed

lemma span-empty[simp]: $\text{span } \{\} = \{0\}$
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: $A \subseteq B \implies B \leq \text{span } A \implies \text{subspace } B \implies \text{span } A = B$
  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

\begin{align*}
\text{span-induct-alt-help-0: } 0 & \in \text{span-induct-alt-help } S \\
\text{span-induct-alt-help-S: } x & \in S \implies z \in \text{span-induct-alt-help } S \implies \\
& (\text{scale } c \ x + z) \in \text{span-induct-alt-help } S
\end{align*}

\textbf{lemma} \text{span-induct-alt'':}

\textbf{assumes} \ h0: h \ 0 \\
\text{and } \text{hS: } !! e \ x \ y. \ x \in S \implies y \implies h (\text{scale } c \ x + y)

\textbf{shows} \ \forall x \in \text{span } S. \ h x

\textbf{proof} –

\{ \\
\text{fix } x :: 'b \\
\text{assume } x: \ x \in \text{span-induct-alt-help } S \\
\text{have } h x \\
\text{apply } (\text{rule span-induct-alt-help.induct[OF } x]) \\
\text{apply } (\text{rule } h0) \\
\text{apply } (\text{rule } hS) \\
\text{apply assumption} \\
\text{apply assumption} \\
\text{done}
\}

\textbf{note } \text{th0 = this}

\{ \\
\text{fix } x \\
\text{assume } x: \ x \in \text{span } S \\
\text{have } x \in \text{span-induct-alt-help } S \\
\text{proof } (\text{rule span-induct[where } x=x \ \text{and } S=S]) \\
\text{show } x \in \text{span } S \ \text{by } (\text{rule } x)
\}

\textbf{next}

\text{fix } x \\
\text{assume } xS: \ x \in S \\
\text{from } \text{span-induct-alt-help-S[OF } xS \text{span-induct-alt-help-0, of } 1] \\
\text{show } x \in \text{span-induct-alt-help } S \\
\text{by } \text{simp}

\textbf{next}

\text{have } 0 \in \text{span-induct-alt-help } S \ \text{by } (\text{rule span-induct-alt-help-0})

\textbf{moreover}

\{ \\
\text{fix } x \ y \\
\text{assume } h: \ x \in \text{span-induct-alt-help } S \ y \in \text{span-induct-alt-help } S \\
\text{from } h \text{ have } (x + y) \in \text{span-induct-alt-help } S \\
\text{apply } (\text{induct rule: span-induct-alt-help.induct}) \\
\text{apply } \text{simp} \\
\text{unfolding } \text{add.assoc} \\
\text{apply } (\text{rule span-induct-alt-help-S}) \\
\text{apply assumption} \\
\text{apply } \text{simp} \\
\text{done}
\}

9
moreover

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

ultimately show subspace (span-induct-alt-help S) 
unfolding subspace-def Ball-def by blast
qed
\}

with \text{th0} show ?thesis by blast
qed

lemma \text{span-induct-alt}: 
assumes \text{h0: } h = 0 
and \text{hS: } \forall c \ y. \ x \in S \implies h \ y = h (scale c x + y) 
and \text{h: } x \in \text{span } S 
shows \text{h x}
using \text{span-induct-alt'[of h S] h0 hS x} by blast

Individual closure properties.


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

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

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

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

lemma \text{dependent-0:} 
assumes \text{0 } \in A 
shows \text{dependent } A 
unfolding \text{dependent-def} 
apply \text{(rule-tac } x=0 \text{ in bexI)} 
using \text{assms span-0} 
apply \text{auto} 
done
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 \( \text{scale} \ \text{scaleC} \ \text{f} ::\ 'b::ab-group-add \Rightarrow \ 'c::ab-group-add \)
  shows vector-space.\span \text{scaleC} \ f ' \ (\text{span} \ S) = f ' (\text{span} \ S)
proof
  interpret B: vector-space \text{scale} using lf by (metis linear-iff)
  interpret C: vector-space \text{scaleC} \ f using lf by (metis linear-iff)
  interpret lf: linear \text{scale} \ \text{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 \text{scaleC} \ f ' \ (\text{span} \ S)
      using lf \text{subspace-span} by (rule subspace-linear-image)
  next
  fix T
  assume \( f ' \ S \subseteq T \) and vector-space.\subspace \text{scaleC} \ T
  then show \( f ' \ (\text{span} \ S) \subseteq T \)
    unfolding image-subset-iff-subset-vimage
    by (metis \text{subspace-linear-vimage} lf \text{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)
    show \(\forall \ r \ a \ b. \ \text{scale} \ r \ a + \text{scale} \ r \ b = \text{scale} \ r \ (a + b)\) by (metis scale-right-distrib)
qed
moreover have vector-space.subspace \((\lambda x \ (a, b)). (\text{scale} \ x \ a, \ \text{scale} \ x \ b)\) (span \(A \times \text{span} \ B\))
by (intro subspace-Times subspace-span)
ultimately show subspace \(((\lambda (a, b). \ a + b) \cdot (\text{span} \ A \times \text{span} \ B))\)
  by (metis (lifting) linear-iff vector-space.subspace-linear-image)
next
fix \(T\)
assume \(A \cup B \subseteq T\) and subspace \(T\)
then show \((\lambda (a, b). \ a + b) \cdot (\text{span} \ A \times \text{span} \ B) \subseteq T\)
  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 \(\text{exI}, \text{simp}\))
  apply (erule rev-image-eqI [OF SigmaI [OF rangeI]])
  apply auto
  done
  then show ?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 \( \forall x \in \text{span } P, x \in \text{?E} \)

proof (rule span-induct-alt')
  show \( 0 \in \text{Collect ?h} \)
    unfolding mem-Collect-eq
    apply (rule \text{exI}[\text{where } x=\{\}])
    apply simp
    done
next
fix \( c \ x \ y \)
  assume \( x: x \in P \)
  assume \( hy: y \in \text{Collect ?h} \)
from \( hy \) obtain \( S \ u \) where \( fS: \text{finite } S \) and \( SP: S \subseteq P \)
and \( u: \text{setsum } (\lambda v. \text{scale } (u \ v) \ v) \ S = y \) by blast
let \( ?S = \text{insert } x S \)
let \( ?u = \lambda y. \text{if } y = x \text{ then } (\text{if } x \in S \text{ then } u + c \text{ else } c) \text{ else } u \ y \)
from \( fS \ SP \ x \) have \( \text{th0: } \text{finite } (\text{insert } x S) \text{ insert } x S \subseteq P \)
  by blast+
  have \( \text{?Q ?S ?u } (\text{scale } c \ x + y) \)
proof cases
  assume \( xS: x \in S \)
  have \( \text{th01: } S = (S - \{x\}) \cup \{x\} \)
  and \( Sss: \text{finite } (S - \{x\}) \text{ finite } \{x\} \text{ } (S - \{x\}) \cap \{x\} = \{\} \)
  using \( xS \ fS \) by auto
  have \( \text{setsum } (\lambda v. \text{scale } (\text{?u v} \ v)) \ ?S = (\sum v \in S - \{x\} \text{. } \text{scale } (\text{?u v} \ v) \text{ x} \)
    using \( xS \) by (simp add: setsum.remove [OF \( fS \ xS \] insert-absorb)
  also have \( \ldots = (\sum v \in S. \text{ scale } (u \ v) \ v) + \text{ scale } c \ x \)
    by (simp add: setsum.remove [OF \( fS \ xS \] algebra-simps)
  also have \( \ldots = \text{ scale } c \ x + y \)
    by (simp add: add.commute u)
  finally have \( \text{setsum } (\lambda v. \text{scale } (?u v) \ v) \ ?S = \text{ scale } c \ x + y \).
then show \( \text{?thesis using th0 by blast} \)
next
  assume \( xS: x \notin S \)
  have \( \text{th00: } (\sum v \in S. \text{ scale } (\text{if } v = x \text{ then } c \text{ else } u \ v) \ v) = y \)
    unfolding \( u[\text{symmetric}] \)
    apply (rule setsum.cong)
    using \( xS \)
    apply auto
    done
  show \( \text{?thesis using fS xS th0} \)
    by (simp add: th00 setsum-clauses add.commute cong del: if-weak-cong)
qed
then show \( \text{(scale } c \ x + y) \in \text{Collect ?h} \)
  unfolding mem-Collect-eq
  apply \( - \)
  apply (rule \text{exI}[\text{where } x=\{\}])
  apply (rule \text{exI}[\text{where } x=\{\}]})
apply metis
done

qed
ultimately show \textit{thesis by blast}
qed

lemma \textit{dependent-explicit}:
\[
\text{dependent } P \iff (\exists S \ u. \ \text{finite } S \land S \subseteq P \land (\exists v \in S. \ u \ v \neq 0 \land \text{setsum } (\lambda v. \ \text{scale } (u \ v) \ v) \ S = 0))
\]
(is \(\text{lhs} = \text{rhs}\))
proof –
{
 assume \(aP\): \text{dependent } P
 then obtain \(a \in P\) \text{ where } \(aP\): \(a \in P\) \text{ and } \(fS\): \text{finite } S
 and \(SP: S \subseteq P \setminus \{a\}\) \text{ and } \(ua: \text{setsum } (\lambda v. \ \text{scale } (u \ v) \ v) \ S = a\)
 unfolding \text{dependent-def span-explicit} by blast
 let \(S = \text{insert } a\ S\)
 let \(?u = \lambda y. \text{if } y = a \text{ then } -1 \text{ else } u \ y\)
 let \(?v = a\)
 from \(aP\) \(SP\) have \(aS: a \notin S\)
 by blast
 from \(fS\) \(SP\) \(aP\) have \(th0: \text{finite } ?S ?S \subseteq P \ ?v \in ?S \ ?a \ ?v \neq 0\) by auto
 have \(s0: \text{setsum } (\lambda v. \ \text{scale } (?a \ v) \ v) \ ?S = 0\)
 using \(fS\) \(aS\)
 apply \((\text{simp add: setsum-clauses field-simps})\)
 apply \((\text{subst (2) } ua[\text{symmetric}])\)
 apply \((\text{rule setsum.cong})\)
 apply auto
 done
 with \(th0\) have \(?rhs\) by fast
}
moreover
{
 fix \(S \ u \ v\)
 assume \(fS: \text{finite } S\)
 and \(SP: S \subseteq P\)
 and \(vS: v \in S\)
 and \(uv: u \ v \neq 0\)
 and \(u: \text{setsum } (\lambda v. \ \text{scale } (u \ v) \ v) \ S = 0\)
 let \(?a = v\)
 let \(?S = S \setminus \{v\}\)
 let \(?u = \lambda i. \ (\text{inverse } (u \ i)) \ / \ u \ v\)
 have \(th0: \ ?a \in P \ \text{finite } ?S \ ?S \subseteq P\)
 using \(fS\) \(SP\) \(?vS\) by auto
 have \(\text{setsum } (\lambda v. \ \text{scale } (?a \ v) \ v) \ ?S =
 \ \text{setsum } (\lambda v. \ \text{scale } (\text{inverse } (u \ ?a)) (\text{scale } (u \ v) \ v)) \ S \setminus \text{scale } (?u \ v) \ v\)
 using \(fS\) \(?vS\) \(uv\) by \((\text{simp add: setsum-diff1 field-simps})\)
 also have \(\ldots = ?a\)

16
\textbf{unfolding} scale-setsum-right[symmetric] \textbf{u using} uv by simp

\textbf{finally have} setsum (\lambda v. scale (?u v) v) ?S = ?a .

\textbf{with} th0 \textbf{have} ?lhs

\textbf{unfolding} dependent-def span-explicit

\textbf{apply} –

\textbf{apply} (rule bexI[where \textbf{x} = ?a])

\textbf{apply} (simp-all del: scale-minus-left)

\textbf{apply} (rule exI[where \textbf{x} = ?S])

\textbf{apply} (auto simp del: scale-minus-left)

\textbf{done}

\textbf{ultimately show} ?thesis \textbf{by} blast

\textbf{qed}

\textbf{lemma} span-finite:

\textbf{assumes} \textbf{fS: finite} \(S\)

\textbf{shows} \(span \ S = \{ y \, | \, \exists u. \text{setsum} (\lambda v. \text{scale} \ (u v) \ v) \ S = y \}\)

(is - = ?rhs)

\textbf{proof} –

\{ 
  \textbf{fix} \(y\)
  \textbf{assume} \(y \in span \ S\)
  \textbf{from} \(y\) \textbf{obtain} \(S' \ u\) \textbf{where} \textbf{fS': finite} \(S'\)
  \textbf{and} \(SS': S' \subseteq S\)
  \textbf{and} \(u\): \text{setsum} (\lambda v. \text{scale} \ (u v) v) \ S' = y\)

  \textbf{unfolding} span-explicit \textbf{by} blast

  \textbf{let} \(?u = \lambda x. \text{if} \ x \in S' \text{then} u \ x \text{else} 0\)

  \textbf{have} \text{setsum} (\lambda v. \text{scale} \ (?u v) v) \ S = \text{setsum} (\lambda v. \text{scale} \ (u v) v) \ S'

  \textbf{using} \text{SS' fS by} (auto intro!: setsum.mono-neutral-cong-right)

  \textbf{then have} \text{setsum} (\lambda v. \text{scale} \ (?u v) v) \ S = y \textbf{by} (metis u)

  \textbf{then have} \(y \in ?rhs\) \textbf{by auto}

\}

\textbf{moreover}

\{ 
  \textbf{fix} \(y\ \ u\)
  \textbf{assume} \(u\): \text{setsum} (\lambda v. \text{scale} \ (u v) v) \ S = y\)

  \textbf{then have} \(y \in span \ S\) \textbf{using} \textbf{fS} \textbf{unfolding} span-explicit \textbf{by} auto

\}

\textbf{ultimately show} ?thesis \textbf{by} blast

\textbf{qed}

\textbf{lemma} independent-insert:

\textbf{independent} (insert \(a\) \(S\)) \leftrightarrow

  (\text{if} \(a \in S\) \text{then} \text{independent} \ S \text{ else} \text{independent} \ S \land a \notin \text{span} \ S)

(is ?lhs \leftrightarrow ?rhs)

\textbf{proof} (cases \(a \in S\))
case True
then show ?thesis
  using insert-absorb[OF True] by simp
next
case False
show ?thesis
proof
  assume i: ?lhs
  then show ?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: ?rhs
  show ?lhs
    using i False
    apply simp
    apply (auto simp add: dependent-def)
    apply (case-tac aa = a)
    apply auto
    apply (subgoal-tac a S - {aa} = insert a (S - {aa}))
    apply simp
    apply (subgoal-tac a ∈ span (insert aa (S - {aa})))
    apply (subgoal-tac insert aa (S - {aa}) = S)
    apply simp
    apply blast
    apply (rule in-span-insert)
    apply assumption
    apply blast
    apply blast
    done
qed
qed

lemma spanning-subset-independent:
  assumes BA: B ⊆ A
    and iA: independent A
    and AsB: A ⊆ span B
  shows A = B
proof
  show B ⊆ A by (rule BA)

  from span-mono[OF BA] span-mono[OF AsB]
have \( sAB \): \( \text{span} \ A = \text{span} \ B \) unfolding \( \text{span-span} \) by blast

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

qed

lemma exchange-lemma:
assumes \( f: \text{finite} \ t \)
and \( i: \text{independent} \ s \)
and \( sp: s \subseteq \text{span} \ t \)
shows \( \exists t', \text{card} \ t' = \text{card} \ t \land \text{finite} \ t' \land s \subseteq t' \land t' \subseteq s \cup t \land s \subseteq \text{span} \ t' \)
using \( f \ i \ sp \)
proof (induct \( \text{card} \ (t - s) \) arbitrary: \( s \ t \) rule: less-induct)

case less
note \( ft = (\text{finite} \ t) \) and \( s = (\text{independent} \ s) \) and \( sp = (s \subseteq \text{span} \ t) \)
let \( ?P = \lambda t'. \text{card} \ t' = \text{card} \ t \land \text{finite} \ t' \land s \subseteq t' \land t' \subseteq s \cup t \land s \subseteq \text{span} \ t' \)
let \( ?ths = \exists t'. ?P \ t' \)

\{
& \text{assume } st: s \subseteq t \\
& \text{from } st \ ft \text{ span-mono[OF st]} \\
& \text{have } ?ths \\
& \text{apply } - \\
& \text{apply } (\text{rule exI[where } x=t]) \\
& \text{apply } (\text{auto intro: span-superset}) \\
& \text{done} \\
\}

moreover
{ 
  assume st: \( t \subseteq s \)
  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: \( \neg s \subseteq t \land \neg t \subseteq s \)
  from st(2) obtain b where b: \( b \in t \land b \notin s \)
    by blast
  from b have t - \{b\} - s \subseteq 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 \neq 0
    by auto
  have ?ths
  proof cases
    assume stb: s \subseteq 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 \subseteq u u \subseteq s \cup (t - \{b\}) s \subseteq span u
      and fu: finite u by blast
    let ?w = insert b u
    have th0: s \subseteq insert b u
      using u by blast
    from u(3) b have u \subseteq s \cup t
      by blast
    then have th1: insert b u \subseteq s \cup t
      using u b by blast
    have bu: b \notin 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 \subseteq span u.
    also have \ldots \subseteq span (insert b u)
      by (rule span mono) blast
    finally have th3: s \subseteq 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: \(~\subseteq span (t - \{b\})
  from stb obtain a where a: a \in s a \notin span (t - \{b\})
    by blast
  have ab: a \neq b
    using a b by blast
  have at: a \notin t
    using a ab span-superset[of 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 \in s
    have t: t \subseteq insert b (insert a (t - \{b\}))
      using b by auto
    from b(1) have b \in span t
      by (simp add: span-superset)
    have bs: b \in span (insert a (t - \{b\}))
      apply (rule in-span-delete)
      using a sp unfolding subset-eq
      apply auto
    done
    from xs sp have x \in span t
      by blast
    with span-mono[of t] have x: x \in span (insert b (insert a (t - \{b\}))) ..
    from span-trans[of bs x] have x \in span (insert a (t - \{b\})) .
  }
  then have sp': s \subseteq 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 \subseteq u u \subseteq s \cup insert a (t - \{b\})
    s \subseteq span u by blast
  from u a b ft at ct0 have ?P a
    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 \subseteq span t
  shows finite s \land card s \leq card t
  by (metis exchange-lemma[OF f i sp] finite-subset card-mono)
lemma independent-explicit:

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

unfolding dependent-explicit \([A] \) 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 fin-\(A\): finite \(A\)

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

shows independent \(A\)

proof (unfold independent-explicit, clarify)

fix \(S\) \(v\) and \(u :: 'b \Rightarrow 'a\)

assume \(S : S \subseteq A\) \(\text{and} v : v \in S\)

let \(?g\) = \(\lambda x. \text{if } x \in S \text{ then } u \cdot x \text{ else } 0\)

have \((\sum_{v \in S.} \text{scale } (?g \cdot v)) = (\sum_{v \in S} \text{scale } (u \cdot v))\)

using \(S\).fin-\(A\) by (auto intro!: setsum.mono_neutral_cong_right)

also assume \((\sum_{v \in S.} \text{scale } (u \cdot v)) = 0\)

finally have \(?g \cdot v = 0\) using \(v \in S\) sum by force

thus \(u \cdot v = 0\) unfolding if-[P[OF \(v\)]].

qed

definition cart-basis = \(\{\text{axis } i \ 1 \mid i. \ i \in \text{UNIV}\}\)

lemma finite-cart-basis: finite (cart-basis) unfolding cart-basis-def

using finite-Atleast-Atmost-nat by fastforce

lemma independent-cart-basis:

vec.independent (cart-basis)

proof (rule vec.independent_if-scalars-zero, auto)

show finite (cart-basis) using finite-cart-basis .

fix \(f :: (\text{vec } 'a, 'b) \text{ vec} \Rightarrow 'a \text{ and } u :: (\text{vec } 'a, 'b) \text{ vec}\)

assume eq-\(0\): \((\sum_{x \in \text{cart-basis}.} f \cdot x \cdot s \cdot x) = 0\) \(\text{and} x \cdot \text{in}: x \in \text{cart-basis}\)

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

have setsum-eq-\(0\): \((\sum_{x \in (\text{cart-basis})} - \{x\}. f \cdot x \cdot (x \cdot s \cdot i)) = 0\)

proof (rule setsum.neutral, rule ballI)

fix \(axa\) assume \(axa: xa \in \text{cart-basis} - \{x\}\)

obtain \(a\) where \(a = \text{axis } a \ 1\) \(\text{and} a \cdot \text{not-i}: a \neq i\)

using \(xa \cdot \text{unfolding}\) cart-basis-def by auto

have \(xa \cdot s \cdot i = 0\) unfolding a axis-def using a-not-i by auto

thus \(f \cdot xa \cdot s \cdot i = 0\) by simp

qed

have \(0 = (\sum_{x \in \text{cart-basis}.} f \cdot x \cdot s \cdot x) \cdot s \cdot i\) using eq-\(0\) by simp

also have \(\ldots = (\sum_{x \in \text{cart-basis}.} (f \cdot x \cdot s \cdot x) \cdot s \cdot i)\) unfolding setsum-component ..

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

**lemma** span-cart-basis: 
vec.span (cart-basis) = UNIV
proof (auto)
fix \( x \)::('a, 'b) vec 
let \( \lambda v. x \mod (\text{THE } i. v = \text{axis } i \ 1) \)
show \( x \in \text{vec.span (cart-basis)} \)
proof (unfold vec.span-finite[OF finite-cart-basis], auto rule ex1[of - \( ?f \)], subst (2) vec-eq-iff, clarify)
fix \( i::'b \)
let \( \lambda w. \text{axis } i \ (1::'a) \)
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: \( \sum_{v \in \{\text{cart-basis}\}} - \{?w\}. x \mod (\text{THE } i. v = \text{axis } i \ 1) \cdot v \mod i \) 
= 0
proof (rule setsum.neutral, rule ballI)
fix \( xa::('a, 'b) vec \)
assume \( xa: xa \in \text{cart-basis} - \{?w\} \)
obtain \( j \) where \( j: xa = \text{axis } j \ 1 \) and \( i\not= j \) using \( xa \) unfolding 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 \( \lambda i. xa = \text{axis } i \ 1 \Rightarrow i = j \) by (metis axis-eq-axis \( j \) zero-neq-one)
qed 
show \( x \mod (\text{THE } i. xa = \text{axis } i \ 1) \cdot xa \mod i = 0 \)
apply (subst (2) \( j \))
unfolding the-eq-j unfolding axis-def using \( i\not= j \) by simp 
qed 
have \( \sum_{v \in \text{cart-basis}.} x \mod (\text{THE } i. v = \text{axis } i \ 1) \mod s \ v \) \( \mod i = \) 
\( \sum_{v \in \text{cart-basis}.} (x \mod (\text{THE } i. v = \text{axis } i \ 1) \cdot s \ v) \mod i \) unfolding setsum-component ..
also have ... = (\( \sum_{v \in \text{cart-basis}.} x \mod (\text{THE } i. v = \text{axis } i \ 1) \cdot v \mod i \)) unfolding vec-smult-component ..
also have ... = \( x \mod (\text{THE } a. \ ?w = \text{axis } a \ 1) \cdot ?w \ mod i + (\sum_{v \in \{\text{cart-basis}\}} - \{?w\}. x \mod (\text{THE } i. v = \text{axis } i \ 1) \cdot v \ mod i) \) 
  by (rule setsum.remove[OF finite-cart-basis], auto simp add: cart-basis-def)
also have ... = \( x \mod (\text{THE } a. \ ?w = \text{axis } a \ 1) \cdot ?w \ mod i \) unfolding setsum-eq-0 by simp 
also have ... = \( x \mod i \) unfolding the-eq-i unfolding axis-def by auto 
finally show \( \sum_{v \in \text{cart-basis}.} x \mod (\text{THE } i. v = \text{axis } i \ 1) \mod s \ v \) \( \mod i = x \ mod i \) . 
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 mlt: ?d − card (insert a S) < ?d − card S
      using aS a independent-bound[OF th1] by auto
from less1[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
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
  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)
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-eq-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 \( \text{dim UNIV} = \text{card } (\text{Basis}) \)
  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 \( \text{dim UNIV} = \text{dimension} \)
  by (metis basis-card-eq-dim dimension-def independent-Basis span-Basis top-greatest)

lemma dim-subset-UNIV:
  shows \( \text{dim } S \leq \text{dimension} \)
  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 \( iB: \text{independent } (\text{insert } a \text{ } B) \)
      using \( iB: aV BV \) by (simp add: independent-insert)
    from \( aV BV \) have \( \text{th0: insert } a \subseteq V \)
      by blast
    from \( aB \) have \( a \notin B \)
      by (auto simp add: span-superset)
    with \( \text{independent-card-le-dim[OF th0 iaB]} \)
      have \( \text{False} \) by auto
  }
  then show \( a \in \text{span } B \) by blast
qed
lemma card-le-dim-spanning:
assumes BV: B \subseteq V
  and VB: V \subseteq \text{span} B
  and fB: finite B
  and dVB: dim V \geq \text{card} B
shows independent B
proof –
{ 
  fix a
  assume a: a \in B a \in \text{span} (B - \{a\})
  from a fB have c0: card B \neq 0
    by auto
  from a fB have cb: card (B - \{a\}) = card B - 1
    by auto
  from BV a have th0: B - \{a\} \subseteq V
    by blast 
  
  fix x
  assume x: x \in V
  from a have eq: insert a (B - \{a\}) = B
    by blast 
  from x VB have x': x \in \text{span} B
    by blast 
  from span-trans[OF a(2), unfolded eq, OF x']
  have x \in \text{span} (B - \{a\}) .
  
  then have th1: V \subseteq \text{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 \leq 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 \subseteq V \Rightarrow \text{card} B = \text{dim} V \Rightarrow \text{finite} B \Rightarrow \text{independent} B \leftarrow\rightarrow V \subseteq \text{span} B
by (metis order-eq-iff card-le-dim-spanning card-ge-dim-independent)

lemma independent-bound-general:
shows independent S ==\Rightarrow \text{finite} S \land \text{card} S \leq \text{dim} S
by (metis independent-card-le-dim independent-bound subset-refl)

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

proof –

have th0: \( \dim S \leq \dim (\operatorname{span} S) \)
  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 \operatorname{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 \operatorname{span} S
  using B(1) by (metis subset-eq span-inc)
have sssB: \operatorname{span} S \subseteq \operatorname{span} B
  using span-mono[OF B(3)] by (simp add: span-span)
from span-card-ge-dim[OF bSS sssB fb] th0 show ?thesis
  using fb(2) by arith
qed

lemma subset-le-dim:
  shows S \subseteq \operatorname{span} T ==> \dim S \leq \dim T
  by (metis dim-span dim-subset)

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

class 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 \setminus f\ a \in \operatorname{C.span} (f \ ' S \setminus \{f\ a\})
    have eq: f \ ' S \setminus \{f\ a\} = f \ ' (S \setminus \{a\})
      using fi by (auto simp add: inj-on-def)
    from a have f\ a \in f \ ' B.\operatorname{span} (S \setminus \{a\})
      unfolding eq B.\operatorname{span-linear-image}[OF l, of S \setminus \{a\}] by blast
    then have a \in B.\operatorname{span} (S \setminus \{a\})
      using fi by (auto simp add: inj-on-def)
    with a(1) iS have False
      by (simp add: B.\operatorname{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 fibr:: 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 fx − k *c f a ∈ C.span (f ' b)
    by (metis (full-types) lf linear.linear-cmul linear.linear-sub)

qed
then have \( th : -k \cdot c f a \in C.\text{span} \ (f \cdot b) \)
  using \( 2.\text{prems}(5) \) 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 \( \textit{thesis} \) using \( B.\text{span-mono}[of \ b - \{a\} \ b] \)
    by blast
next
  case \( False \)
  with \( \text{span-mul}[OF th] \)
    have \( \textit{th1} : f a \in \text{span} \ (f \cdot b) \)
      by auto
  from \( \text{inj-on-image-set-diff}[OF 2.\text{prems}(3), \ of \ insert \ a \ b \ \{a\}, \ symmetric] \)
    have \( \textit{tha} : f (\text{insert} \ a \ b - \{a\}) = f (\text{insert} \ a \ b - \{a\}) \) by blast
  from \( 2.\text{prems}(2) \) [unfolded dependent-def \( \text{bex-simps}(8), \ rule-format, \ of \ f a \)]
    have \( f a \notin \text{span} \ (f \cdot b) \) using \( \text{tha} \)
      using \( 2.\text{hyps}(2) \)
      \( 2.\text{prems}(3) \) by auto
    with \( \textit{th1} \) have \( \textit{False} \) by blast
  then show \( \textit{thesis} \) by blast
qed

lemma \( \text{linear-independent-extend-lemma} : \)
  fixes \( f :: 'b \Rightarrow 'c \)
  assumes \( \textit{fi} : \text{finite} \ B \)
  and \( \textit{ib} : B.\text{independent} B \)
  shows \( \exists g. \)
    \( (\forall x \in B.\text{span} B. \ \forall y \in B.\text{span} B. \ g (x + y) = g x + g y) \) \land
    \( (\forall x \in B.\text{span} B. \ \forall c. \ g (c \cdot b x) = c \cdot g (g x)) \) \land
    \( (\forall x \in B. \ g x = f x) \)
  using \( \textit{ib} \ f i \)
proof (induct rule: \( \text{finite-induct}[OF \ f i] \))
  case \( 1 \)
  then show \( \textit{case} \) by auto
next
  case \( \{2 \ a \ b\} \)
  from \( 2.\text{prems} \) \( 2.\text{hyps} \) have \( \textit{ibf} : B.\text{independent} b \ \text{finite} \ b \)
    by \( \text{simp-all add: B.independent-insert} \)
  from \( 2.\text{hyps}(3)[OF \textit{ibf}] \) obtain \( g \) where
    \( g : \forall x \in B.\text{span} b. \ \forall y \in B.\text{span} b. \ g (x + y) = g x + g y \) \land
    \( \forall x \in B.\text{span} b. \ \forall c. \ g (c \cdot b x) = c \cdot g x \ \forall x \in b. \ g x = f x \) by blast
  let \( \textit{?h} = \lambda z. \ \text{SOME} \ k. \ (z - k \cdot b a) \in B.\text{span} b \)
{ fix \( z \)
  assume \( z : z \in B.\text{span} (\text{insert} \ a \ b) \)
  have \( \textit{th0} : z - \textit{?h} z \cdot b a \in B.\text{span} b \)
apply (rule someI-ex)
unfolding B.span-breakdown-eq[symmetric]
apply (rule z)
done
{
fix k
assume \( k: z - k \cdot b \ a \in B.\text{span} \ b \)
have eq: \( z - ?h \ z \cdot b \ a = (z - k \cdot b \ a) = (k - ?h \ z) \cdot b \ a \)
  by (simp add: field-simps B.scale-left-distrib [symmetric])
from B.span-sub[OF th0 k] have khz: \( (k - ?h \ z) \cdot b \ a \in B.\text{span} \ b \)
  by (simp add: eq)
{
  assume \( k \neq ?h \ z \)
  then have \( k0: k - ?h \ z \neq 0 \) by simp
  from k0 B.span-mul[OF khz, of 1/(k - ?h \ z)]
  have a \( \in B.\text{span} \ b \) 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 \( z - ?h \ z \cdot b \ a \in B.\text{span} \ b \land (\forall k. z - k \cdot b \ a \in B.\text{span} \ b \rightarrow k = ?h \ z) \)
  by blast
}
note \( h = \text{this} \)
let ?g = \( \lambda z. (?h \ z) \cdot c \ (f \ a) + g \ (z - (?h \ z) \cdot b \ a) \)
{
fix \( x \ y \)
assume \( x: x \in B.\text{span} \ (\text{insert} \ a \ b) \)
and \( \ y: y \in B.\text{span} \ (\text{insert} \ a \ b) \)
have tha: \( \forall (x::'b) \ y a k l. (x + y) - (k + l) \cdot b \ a = (x - k \cdot b \ a) + (y - l \cdot b \ a) \)
  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 \( x: x \in B.\text{span} \ (\text{insert} \ a \ b) \)
have \( \forall (x::'b) \ c \ k \ a, \ c \ast b \ x - (c \ast k) \ast b \ a = c \ast b \ (x - k \ast b \ a) \) by (simp add: algebra-simps)

have \( hc: \ ?h \ (c \ast b \ x) = c \ast \ ?h \ x \)

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
}
m 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 \( ?\text{case} \)
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
  C: finite-dimensional-vector-space
  scaleB BasisB +
  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 ∃ g. linear (op *b) (op *c) g ∧ (∀ x∈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 ⊆ C B.independent C ∧ x. x ∈ 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:
    (∀ x∈B. span C. ∀ y∈B. span C. g (x + y) = g x + g y) ∧
    (∀ x∈B. span C. ∀ c. g (c *b x) = c *c g x) ∧
    (∀ x∈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 scale scaleC (f::'b=>'c::ab-group-add)
and VB: V ⊆ span B
shows f ' V ⊆ vector-space.span scaleC (f ' B)
unfolding span-linear-image[OF lf] by (metis VB image_mono)

qed
end
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: \( \forall x \in B. f x = 0 \)
  shows \( \forall x \in \text{span } B. f x = 0 \)
  using f0 subspace-kernel[OF lf]
  by (rule span-induct')

lemma linear-eq-0:
  assumes lf: linear scale scaleB f
  and SB: S \subseteq \text{span } B
  and f0: \( \forall x \in B. f x = 0 \)
  shows \( \forall x \in S. f x = 0 \)
  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 \subseteq \text{span } B
  and fg: \( \forall x \in B. f x = g x \)
  shows \( \forall x \in S. f x = g x \)
  proof -
    let \( ?h = \lambda x. f x - g x \)
    from fg have fg': \( \forall x \in B. ?h x = 0 \) 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)

34
and $f :: (\'b\Rightarrow\'c)$

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

begin

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

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

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 ' UNIV])
      using sf unfolding surj-def
      apply blast
      apply (rule image-mono)
      apply (rule B(3))
      apply (rule 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
  }

qed
from linear-indep-image-lemma[OF lf fB fBi fB x] have x = 0 by blast 

then show ?thesis
unfolding linear-linear-injective-0[OF lf]
using B(3)
by blast
qed

lemma linear-injective-isomorphism:
assumes lf: linear scale scale f
and fi: inj f
shows \( \exists f'\cdot \text{linear scale scale } f' \wedge (\forall x. f' (f x) = x) \wedge (\forall x. f (f' x) = x) \)
proof
interpret lbfdvs: linear-between-finite-dimensional-vector-spaces scale scale Basis f
by (unfold-locales, simp add: lf linear.linear-cmul linear.linear-add)
show ?thesis
unfolding isomorphism-expand[symmetric]
using lbfdvs.linear-surjective-right-inverse
using linear-injective-imp-surjective
by (metis comp-assoc comp-id fi lbfdvs.linear-injective-left-inverse lf)
qed

lemma linear-surjective-isomorphism:
assumes lf: linear scale scale f
and sf: surj f
shows \( \exists f'\cdot \text{linear scale scale } f' \wedge (\forall x. f' (f x) = x) \wedge (\forall x. f (f' x) = x) \)
proof
interpret lbfdvs: linear-between-finite-dimensional-vector-spaces scale scale Basis f
apply (unfold-locales) apply (simp add: lf linear-linear-add)
by (metis lf linear-linear-add)
show ?thesis
unfolding isomorphism-expand[symmetric]
using lbfdvs.linear-surjective-right-inverse[OF sf]
using lbfdvs.linear-injective-left-inverse[OF linear-surjective-imp-injective[OF lf sf]]
by (metis left-right-inverse-eq)
qed

lemma left-inverse-linear:
assumes lf: linear scale scale f
and gf: \( g \circ f = \text{id} \)
shows linear scale scale g
proof

from \( gf \) have \( fi: \text{inj} f \)
  by (metis inj-on-id inj-on-imageI2)

from \( \text{linear-injective-isomorphism} \) [OF \( fi \)]
obtain \( h :: \'b \Rightarrow \'b \) where \( h: \text{linear scale scale h} \)
  \( \forall x. h (f \ x) = x \)
  \( \forall x. f (h \ x) = x \)
  by blast

have \( h = g \)
  apply (rule ext) using \( gf \ h \) (2,3)
  by (metis comp-apply id-apply)

with \( h(1) \) show \( ?\text{thesis} \) by blast
qed

end

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

lemma matrix-vector-mult-linear-between-finite-dimensional-vector-spaces:
linear-between-finite-dimensional-vector-spaces (op **s) (op **s)
  (cart-basis) (cart-basis) \( (\lambda x. A \ *v \ (x::'a::{field} ^ n)) \)
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}
Basis
proof
have \( v: \text{vector-space (scaleR :: real => 'a => 'a::{euclidean-space})} \)
  by (unfold-locales)
show \( \text{finite (Basis::'a set)} \) by (metis finite_Basis)
show \( \text{vector-space.independent op *R (Basis::'a set)} \)
  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 \( \text{vector-space.span op *R (Basis::'a set)} = UNIV} \)
  unfolding vector-space.span_finite [OF \( v \) finite_Basis]
  by (fast intro: euclidean_representation)
qed
lemma vector-mul-lcancel[simp]: \( a \ast s x = a \ast s y \iff a = (\theta::'a::{\text{field}}) \vee x = y \)
  by (metis eq-iff-diff-eq-0 vector-mul-eq-0 vector-ssub-ldistrib)

lemma vector-mul-lcancel-imp: \( a \neq (\theta::'a::{\text{field}}) \Longrightarrow a \ast s x = a \ast s y \iff (x = y) \)
  by (metis vector-mul-lcancel)

lemma linear-componentwise:
  fixes f::'a::field "'m => 'a "'n
  assumes lf: linear (op *\ s) (op *\ s) f
  shows \((f x)\$j = \text{setsum} (\lambda i. (x\$i) \ast (f (axis i 1)\$j)) \text{ (UNIV :: 'm set)} \text{ (is ?lhs = ?rhs)}\)
  proof –
  interpret lf: linear (op *\ s) (op *\ s) f using lf.
  let ?M = (UNIV :: 'm set)
  let ?N = (UNIV :: 'n set)
  have fM: finite ?M by simp
  have ?rhs = \text{setsum} (\lambda i. (x\$i) \ast (f (axis i 1))) \text{ ?M}\$j
    unfolding setsum-component by simp
  then show ?thesis
    unfolding \text{basis-expansion by auto}
  qed

lemma matrix-vector-mul-linear: linear (op *\ s) (op *\ s) \((\lambda x. A \ast v (x::'a::{\text{field}} \^ -))\)
  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 \((\lambda x. A \ast v (x::'a::{\text{field}} \^ -))\)
  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 \(\text{matrix f *v x = f (x::'a::field \^ 'n)}\)
  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 \Longrightarrow f = (\lambda x. matrix f *v
(x::'a::{field} "'n))
by (simp add: ext matrix-works)

lemma matrix-of-matrix-vector-mul: matrix(λx. A *v (x :: 'a::{field} "'n)) = A
by (simp add: matrix-eq matrix-vector-mul-linear matrix-works)

lemma matrix-compose:
assumes lf: linear (op *s) (op *s) (f::'a::{field} "'n ⇒ 'a"'m)
and lg: linear (op *s) (op *s) (g::'a"'m ⇒ 'a")
shows matrix (g oo f) = matrix g ** matrix f
using lf lg linear-compose[OF lf lg] matrix-works[OF linear-compose[OF lf lg]]
by (simp add: matrix-equiv matrix-works matrix-vector-mul-assoc[ symmetric] o_def)

lemma matrix-left-invertible-injective:
(∃B. (B::'a::{field} "'m"'n) ** (A::'a::{field} "'n"'m) = mat 1)
    --- (∀x y. A *v x = A *v y → x = y)
proof -
{ fix B::'a"'m"'n and x y assume B: B ** A = 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: ∀x y. A *v x = A *v y → x = y
hence i: inj (op *v A) unfolding inj-on-def by auto
from vec.linear-injective-left-inverse[OF i]
obtain g where g: linear (op *s) (op *s) g g oo op *v A = id by blast
have matrix g ** A = mat 1
unfolding matrix-equiv matrix-vector-mul-ld matrix-vector-mul-assoc[ symmetric]
matrix-works[OF g(1)]
using g(2) by (metis comp-apply id-apply)
then have ∃B. (B::'a::{field} "'m"'n) ** A = mat 1 by blast }
ultimately show thesis by blast
qed

lemma matrix-left-invertible-ker:
(∃B. (B::'a::{field} "'m"'n) ** (A::'a::{field} "'n"'m) = mat 1) --- (∀x. A *v x = 0 → x = 0)
unfolding matrix-left-invertible-injective
using vec.linear-injective-0[OF A]
by (simp add: inj-on-def)

lemma matrix-left-invertible-independent-columns:
fixes A :: 'a::{field} "'n"'m
shows (∃B::'a "'m"'n). B ** A = mat 1) --- (∀c. setsum (λi. c i *s column i A) (UNIV :: 'n set) = 0 → (∀i. c i = 0))
(is ?lhs ----> ?rhs)
proof -
let ?U = UNIV :: 'n set
{ assume k: \( \forall x. A \ast v x = 0 \longrightarrow x = 0 \)
  { fix c i
    assume c: setsum (\( \lambda i. c \ast s column i A \)) ?U = 0 and i: i \in ?U
    let \( \chi_i \) = c i
    have th0: \( A \ast v \chi_i = 0 \)
      unfolding matrix-mult-vsum vec-eq-iff
      by auto
    from k [rule-format, OF th0 i]
    have c i = 0 by (vector vec-eq-iff)
  }
  hence ?rhs by blast }
moreover
{ assume H: ?rhs
  { fix x assume x: A \ast v x = 0
    let ?c = \( \lambda i. ((x \ast i) :: 'a) \)
    from H [rule-format, of ?c, unfolded matrix-mult-vsum[symmetric], OF x]
    have x = 0 by vector
  }
}
ultimately show ?thesis unfolding matrix-left-invertible-ker by blast
qed

lemma matrix-right-invertible-independent-rows:
fixes A :: 'a::{field}'n'\cdot'm
shows \((\exists B::'m'\cdot'n). A ** B = mat 1) \longleftrightarrow
(\forall c. setsum (\( \lambda i. c \ast s row i A \)) (UNIV :: 'm set) = 0 \longrightarrow (\forall i. c i = 0))
unfolding left-invertible-transpose[symmetric]
matrix-left-invertible-independent-columns
by (simp add: column-transpose)

lemma matrix-left-right-inverse:
fixes A A' :: 'a::{field}'n'\cdot'n
shows A ** A' = mat 1 \longleftrightarrow A' ** A = mat 1
proof –
{ fix A A' :: 'a'\cdot'n'\cdot'n
  assume AA': A ** A' = mat 1
  have sA: surj (op \ast v A)
    unfolding surj-def
  apply clarify
  apply (rule-tac x=(A' \ast v y) in exI)
  apply (simp add: matrix-vector-mul-assoc AA' matrix-vector-mul-lid)
  done
from vec.linear-surjective-isomorphism[OF matrix-vector-mul-linear sA]
obtain f' :: 'a'\cdot'n \Rightarrow 'a'\cdot'n
  where f' linear (op **s) (op **s) f' \( \forall x. f' (A \ast v x) = x \forall x. A \ast v f' x = x 
by blast
have th: 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 \rightleftharpoons \ (\forall x \in S. f x = 0 \longrightarrow x = 0)
proof –
have inj-on \ f S \rightleftharpoons \ (\forall x \in S. \forall y \in S. f x = f y \longrightarrow x = y)
by (simp add: inj-on-def)
also have \dots \rightleftharpoons \ (\forall x \in S. \forall y \in S. f x - f y = 0 \longrightarrow x - y = 0)
by simp
also have \dots \rightleftharpoons \ (\forall x \in S. \forall y \in S. f (x - y) = 0 \longrightarrow x - y = 0)
by (simp add: linear.linear-sub[of \ f[\lf]])
also have \dots \rightleftharpoons \ (\forall x \in S. f x = 0 \longrightarrow x = 0)
using (subspace S) subspace-def[of S] subspace-sub[of S] by auto
finally show ?thesis .
qed

end

lemma setsum-constant-scaleR:
shows \((\sum x \in A. \ y) = of-nat \ (card A) ** \ y\)
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

43
shows finite $B$ $\land$ $\text{card } B = \dim (\text{span } B)$

using assms basis-card-eq-dim[of $B$ $\text{span } B$] 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 $(\text{op } \ast b) (\text{op } \ast c)$
by unfold-locales
{ fix $a$
  assume $a$: $a \in S$ $f a \in C$ 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 $(S - \{\})$
    unfolding eq using $B$ span-linear-image[OF $l$] by auto
  moreover have $B$ span $(S - \{\}) \subseteq B$ 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 $s$ $\Longrightarrow$ subspace $(\text{Inter } f)$
unfolding subspace-def by auto

lemma span-eq[simp]: span $s = s$ $\Longleftrightarrow$ 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 $\dim S \geq \dim T$

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

lemma det-identical-columns:
  fixes $A : \text{'}a\text{'} \cdot \text{'}n\text{'}$
  assumes $j k : j \neq k$
  and $r : \text{column } j A = \text{column } k A$
  shows $\det A = 0$
proof -
  let $?U = \text{UNIV} : \text{'}n\text{'}$
  let $?t-jk = \text{Fun}.\text{swap } j \ k \ id$
  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 $?f = \lambda p . \text{of-int } (\text{sign } p) \ast (\prod i \in \text{UNIV} . A \$_{i} \ p i)$
  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 $\text{even-x: evenperm } x$
        and $y : y \text{ permutes } ?U$ and $\text{even-y: evenperm } y$ 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 $tjk-eq : \forall i \ l . A \$_{i} \ ?t-jk l = A \$_{i} \ l$
    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 \( x : x \in ?S1 \)

have \( \text{sign} (\?t-jk \circ x) = \text{sign} (\?t-jk) \ast \text{sign} x \)
by \( \text{(metis (lifting) finite-class.finite-UNIV mem-Collect-eq} \)
  \( \text{permutation-permutes permutation-swap-id sign-compose x}) \)
also have \( ... = - \text{sign} x \) using \( \text{sign-tjk by simp} \)
also have \( ... \neq \text{sign} x \) unfolding \( \text{sign-def} \) by simp
finally have \( \text{sign} (\?t-jk \circ x) \neq \text{sign} x \) and \( (\?t-jk \circ x) \in ?S2 \)
by \( \text{(auto, metis (lifting, full-types) mem-Collect-eq x}) \)

hence \( \text{disjoint:} ?S1 \cap ?S2 = \{\} \) by \( \text{(auto, metis sign-def}) \)

have \( \text{PU-decomposition:} ?PU = ?S1 \cup ?S2 \)
proof \( \text{(auto}) \)
  fix \( x \)
  assume \( x : x \text{ permutes ?U} \) and \( \forall p. \ p \text{ permutes ?U} \longrightarrow x = \text{Fun.swap } j \ k \ id \circ \)
  \( p \longrightarrow \neg \text{evenperm } p \)
  from this obtain \( p \) where \( p : p \text{ permutes UNIV } \) and \( x-eq : x = \text{Fun.swap } j \ k \ id \circ \)
  \( p \)
  and \( \text{odd-p} : \neg \text{evenperm } p \)
  by \( \text{(metis no-types comp-assoc id-comp inv-swap-id permutes-compose} \)
    \( \text{permutes-inv-o(1) tjk-permutes}) \)
thus \( \text{evenperm } x \)
  by \( \text{(metis evenperm-comp evenperm-swap finite-class.finite-UNIV} \)
    \( jk \text{ permutation-permutes permutation-swap-id}) \)
next
fix \( p \) assume \( p : p \text{ permutes ?U} \)
  show \( \text{Fun.swap } j \ k \ id \circ p \text{ permutes UNIV} \) by \( \text{(metis p permutes-compose} \)
    \( \text{tjk-permutes}) \)
qed

have \( \text{setsum } ?f \ ?S2 = \text{setsum } ((\lambda p. \ \text{of-int} (\text{sign} p) \ast (\prod i \in \text{UNIV}. \ A \$ i \$ p) i)) \)
  \( \circ op \circ (\text{Fun.swap } j \ k \ id) \) \( \{p \in \{p. \ p \text{ permutes UNIV}\}. \ \text{evenperm } p\} \)
  unfolding \( g-S1 \) by \( \text{(rule setsum.reindex [OF inj-g])} \)
also have \( ... = \text{setsum } ((\lambda p. \ \text{of-int} (\text{sign} (\?t-jk \circ p)). \ \prod i \in \text{UNIV}. \ A \$ i \$ p) i)) \)
  \( ?S2 \)
  unfolding \( o-def \) by \( \text{(rule setsum.cong, auto simp add: tjk-eq}) \)
also have \( ... = \text{setsum } ((\lambda p. \ - \ ?f p) \ ?S1 \)
proof \( \text{(rule setsum.cong, auto}) \)
  fix \( x \)
  assume \( x : x \text{ permutes ?U} \)
  and \( \text{even-x: evenperm } x \)
  hence \( \text{perm-x: permutation } x \) and \( \text{perm-tjk: permutation } ?t-jk \)
  using \( \text{permutation-permutes[of x] permutation-permutes[of ?t-jk] permutation-swap-id} \)
  by \( \text{(metis finite-code}) \)
  have \( (\text{sign} (\?t-jk \circ x)) = - (\text{sign} x) \)
  unfolding \( \text{sign-compose[OF perm-tjk perm-x] sign-tjk by auto} \)
thus \( \text{of-int} (\text{sign} (\?t-jk \circ x)). \ \prod i \in \text{UNIV}. \ A \$ i \$ x i) \)
  \( = - (\text{of-int} (\text{sign} x). \ \prod i \in \text{UNIV}. \ A \$ i \$ x i) \)
  by \( \text{auto} \)
qed
also have \( ... = - \text{setsum } ?f \ ?S1 \) unfolding \( \text{setsum-negf ..} \)
finally have \( *: \text{setsum } ?f \ ?S2 = - \text{setsum } ?f \ ?S1 \).
have \( \det A = (\sum p \mid p \text{ permutes } \text{UNIV} \cdot \text{of-int} \cdot \text{sign } p) \cdot (\prod i \in \text{UNIV}. \ A \$ i \$ p i) \)

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

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

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

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

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

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

show \(?\text{thesis}\)

by simp

qed

lemma det-row-span:

fixes \( A :: \langle a::\langle \text{field} \rangle \cdot' n \cdot' n \rangle \)
assumes \( x :: x \in \text{vec.span } \{\text{row j A } | j \neq i}\) 
shows \( \det (\chi, k. \text{ if } k = i \text{ then row i A + x else row k A}) = \det A \)

proof —

let \(?U = \text{UNIV } :: \langle n \rangle \text{ set }\)
let \(?S = \{\text{row j A } | j \neq i}\) 
let \(?d = \lambda x. \det (\chi, k. \text{ if } k = i \text{ then } x \text{ else row k A})\)
let \(?P = \lambda x. \det (\text{row i A + x}) = \det A\)

{ 
  fix \( k \)
  have \( \text{if } k = i \text{ then row i A + 0 else row k A} = \text{row k A} \)
  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 \( zS: z \in ?S \text{ and } Py: ?P y \)
  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 \( \text{thz Py det-row-mul[of i] det-row-add[of i]} \)
  by simp
}

ultimately show \(?\text{thesis}\)
  apply —

48
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 j. j ≠ i → row i A ≠ row j A
  have th0: ¬ row i A ∈ vec.span {row j A| j ≠ i}
    (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 λ i. 1]
  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

49
by \textit{(rule finite)}
\begin{itemize}
\item fix \(p\)
\item assume \(p: p\ \text{permutes} \ ?U\)
\item have \(p \in \ ?F\) unfolding \textsf{mem-Collect-eq \ permutes-in-image(OF \ p)}
\item using \(p[\text{unfolded \ permutes-def}]\) by \textit{simp}
\end{itemize}
then have \(PUF: \ ?PU \subseteq \ ?F\) by \textit{blast}
\begin{itemize}
\item fix \(f\)
\item assume \(fPU: f \in \ ?F - \ ?PU\)
\item have \(fUU: f \cdot \ ?U \subseteq \ ?U\)
\item using \(fPU\) by \textit{auto}
\item from \(fPU\) have \(f: \forall i \in \ ?U. \ f\ i \in \ ?U \ \forall \ i \notin \ ?U \ \longrightarrow \ f\ i = i \ \neg(\forall y. \ \exists!x. \ f\ x = y)\)
\item unfolding \textsf{permutes-def} by \textit{auto}
\end{itemize}
\begin{itemize}
\item let \(?A = (\chi \ i. \ A \cdot \ f \ i \cdot s \ B \cdot f \ i)\ :: \ 'a^{\cdot \ 'n}\')\)
\item let \(?B = (\chi \ i. \ B \cdot f \ i)\ :: \ 'a^{\cdot \ 'n}\')\)
\item assume \(fni: \neg \ \text{inj-on} \ f \ ?U\)
\item then obtain \(i\ j\) where \(ij: f\ i = f\ j\ i \neq j\)
\item unfolding \textsf{inj-on-def} by \textit{blast}
\item from \(ij\)
\item have \(rth: \text{row} \ i \ ?B = \text{row} \ j \ ?B\)
\item by \textit{(vector \ row-def)}
\item from \textsf{det-identical-rows(OF \ ij(2) \ rth)}
\item have \(\text{det} \ (\chi \ i. \ A \cdot \ f \ i \cdot s \ B \cdot f \ i) = 0\)
\item unfolding \textsf{det-rows-mult} by \textit{simp}
\end{itemize}
moreover
\begin{itemize}
\item assume \(fi: \ \text{inj-on} \ f \ ?U\)
\item from \(fi\) have \(fith: \ \forall i\ j. \ f\ i = f\ j \ \Longrightarrow \ i = j\)
\item unfolding \textsf{inj-on-def} by \textit{metis}
\item note \(fs = fi[\text{unfolded \ surjective-iff-injective-gen(OF \ fU \ fU \ refl \ fUU, \ symmetric)}]\)
\item fix \(y\)
\item from \(fs\ f\) have \(\exists x. \ f\ x = y\)
\item by \textit{blast}
\item then obtain \(x\) where \(x: f\ x = y\)
\item by \textit{blast}
\item fix \(z\)
\item assume \(z: f\ z = y\)
\item from \(fith \ x\ z\) have \(z = x\)
\item by \textit{metis}
\item with \(x\) have \(\exists!x. \ f\ x = y\)
\end{itemize}

50
ultimately have \( \det (\chi i. A \# i \# f i \ast \ast \# B \# f i) = 0 \)

by blast

}

with \( f(\beta) \) have \( \det (\chi i. A \# i \# f i \ast \ast \# B \# f i) = 0 \)

by blast

}

then have \( z\text{th: } \forall f \in \# F - \# PU. \det (\chi i. A \# i \# f i \ast \ast \# B \# f i) = 0 \)

by simp

{ 
  fix \( p \)
  assume \( pU: p \in \# PU \)
  from \( pU \) have \( p: p \) permutes \( ?U \)
  by blast
  let \( ?s = \lambda p. \# \text{of-int } (\# \text{sign } p) \)
  let \( \# f = \lambda q. ?s p \ast (\prod i \in \# ?U. A \# i \# p i) \ast (\# q \ast (\prod i \in \# ?U. B \# i \# q i)) \)
  have (setsum (\# \text{sign } q \ast

\prod i \in \# ?U. (\chi i. A \# i \# s B \# p i :: 'a \# \text{of-int } (\# \text{sign } i) :: \# q i)) \# \text{permutes-inv} (\# \text{permutes-inv} [\# \text{OF } p], \# \text{OF } \# f])

proof (rule setsum.cong)

fix \( q \)

assume \( qU: q \in \# PU \)
then have \( q: q \) permutes \( ?U \)
  by blast

from \( p q \) have \( pp: \# \text{permutation } p \) and \( \# pq: \# \text{permutation } q \)

unfolding \# \text{permutation-permutes } by auto

have \( \# \text{th00: } \# \text{of-int } (\# \text{sign } p) \ast (\# \text{of-int } (\# \text{sign } p) \ast \text{a}) \) = \( \langle \# \text{a} \rangle \)

\text{of-int } (\# \text{sign } p) \ast \# \text{of-int } (\# \text{sign } p) \ast \# a = \# a

unfolding \# \text{mult.assoc}[\# \text{symmetric}]

unfolding \# \text{of-int-mult}[\# \text{symmetric}]

by (simp-all add: \# \text{sign-idempotent})

have \( \# \text{ths: } ?s q = ?s p \ast ?s (q \circ \# \text{inv } p) \)

using \# \text{pp pq permutation-inverse}[\# \text{OF } pp] \# \text{sign-inverse}[\# \text{OF } pp]

by (simp add: \# \text{th00 ac-simps sign-idempotent sign-compose})

have \( \# \text{th001: setprod } (\lambda i. B \# s \# q (inv p i)) \# \text{inv } p i) \# \text{permutes-inv}\) \( ((\lambda i. B \# s \# q (inv p i)) \circ \# p) \# ?U \)

by (rule setprod.permute[\# \text{OF } p])

have \( \# \text{thp: setprod } (\lambda i. (\chi i. A \# s \# p i \ast \# s B \# p i :: \# a \# \text{of-int } (\# \text{sign } i) :: \# s i \# q i) ?U = \# \text{setprod } (\lambda i. A \# s \# p i) \# \text{setprod } (\lambda i. B \# s \# q (inv p i)) (\# ?U \ast \# \text{p}) ?U \)

unfolding \# \text{th001 setprod.distrib}[\# \text{symmetric}] \# \text{of-def permutes-inverses}[\# \text{OF } p]

apply (rule setprod.cong[\# \text{OF refl}])

using \# \text{permutes-in-image}[\# \text{OF } q]

apply \# \text{vector}

done

show \( ?s q \ast \# \text{setprod } (\lambda i. (((\chi i. A \# s \# p i \ast \# s B \# p i) :: \# a \# \text{of-int } (\# \text{sign } i) :: \# s i \# q i)) ?U = ?s p \ast (\# \text{setprod } (\lambda i. A \# s \# p i) ?U) \ast (\# q \circ \# \text{inv } p) \ast \# \text{setprod } \# (\lambda i. B \# s \# (q \circ
inv p) i) ?U)
  
  by (simp add: sign-nz th00 field-simps sign-idempotent sign-compose)
  qed

then have th2: setsum (λf. det (χ i. A$i*f i *s B$f i)) ?PU = det A * det B
  unfolding det-def setsum-product
  by (rule setsum.cong[OF refl])

have det (A*B) = setsum (λf. det (χ i. A $ i * f $ i *s B $ f i)) ?F
  unfolding matrix-mul-setsum-alt det-linear-rows-setsum[OF fU]
  by simp
also have ... = setsum (λf. det (χ i. A$i*f i *s B$f i)) ?PU
  using setsum mono-neutral-cong-left[OF fF PUF zth, symmetric]
  unfolding det-rows-mul by auto
finally show ?thesis unfolding th2 .
qed

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

lemma invertible-det-nz:
  fixes A::'a::{field} 'n*'n
  shows invertible A ↔ det A ≠ 0

proof –
  
  assume invertible A
  then obtain B :: 'a*'n*'n where B: A ** B = mat 1
    unfolding invertible-right-inverse by blast
  then have det (A ** B) = det (mat 1 :: 'a*'n*'n)
    by simp
  then have det A ≠ 0
    by (simp add: det-mul det-I) algebra

  moreover
  
  assume H: ¬ invertible A
  let ?U = UNIV :: 'n set
  have fU: finite ?U
    by simp
  from H obtain c i where c: setsum (λi. c i *s row i A) ?U = 0
    and iU: i ∈ ?U
    and ci: c i ≠ 0

  52
unfolding invertible-right-inverse
unfolding matrix-right-invertible-independent-rows
by blast
have *: \( (a':\cdot':n') \) \( b \). \( a + b = 0 \implies -a = b \)
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 (\( \lambda j \). (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-cancel-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 = (\( \chi k \). if k = i then 0 else row k A) :: 'a'=':n'=':n'
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 lbfl: linear-first-finite-dimensional-vector-space by unfold-locales
end

lemma vec-dim-card: vec.dim (UNIV::{':field'='n'} set) = CARD ('n)
proof
  let ?f=λi::':n. axis i (1::':a)
  have vec.dim (UNIV::{':field'='n'} set) = card (cart-basis::{':field'='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 axis (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
      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 => 'a
  by unfold-locales (simp-all add: algebra-simps)
interpretation vector-space-over-itself: finite-dimensional-vector-space
  op * :: ':a::field => 'a => 'a {1}
proof (unfold-locales, auto)
  have v: vector-space (op * :: ':a::field => 'a => '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
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 \( ?P = ( \text{THE } x. \ ?P x \land (\forall y. \ ?P y \longrightarrow x \leq y)) \)

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 cong)
apply (rule class.wellorder-axioms.intro) by (blast intro: wf-induct-rule [OF wf])

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

interpretation dual-wellorder: wellorder (op $\geq$)::('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 acneyclic1)
  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 $\Rightarrow$ bool) $\Rightarrow$ '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 $\&\&$ (\forall y::'a::order. P y $\Rightarrow$ y $\leq$ x))
  unfolding Greatest'-def ord.Least-def by fastforce

lemmas Greatest'I2-order = dual-order.LeastI2-order[folded Greatest'-def]
lemmas Greatest'-equality = dual-order.Least-equality[folded Greatest'-def]
lemmas Greatest'I = dual-wellorder.LeastI[folded Greatest'-def]
lemmas Greatest'I2-ex = dual-wellorder.LeastI2-ex[folded Greatest'-def]
lemmas Greatest'I2-wellorder = dual-wellorder.LeastI2-wellorder[folded Greatest'-def]
lemmas Greatest'I-ex = dual-wellorder.LeastI-ex[folded Greatest'-def]
lemmas not-greater-Greatest' = dual-wellorder.not-less-Least[folded Greatest'-def]
lemmas Greatest'I2 = dual-wellorder.LeastI2[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/Carlson-Euclidean-Space
  Dual-Order
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

57
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 o Rep

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

definition from-nat :: nat => 'a
  where from-nat = (Abs' o 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') \{\theta..<\text{CARD}'(a)\} (UNIV::'a set)
proof (unfold bij-btw-def, rule conjI)
  show inj-on Abs' \{\theta..<\text{CARD}'(a)\}
  unfolding inj-on-def Abs'-def
   by (auto, metis Rep-Abs-mod mod-pos-pos-trivial)
show Abs'\' \{\theta..<\text{CARD}'(a)\} = (UNIV::'a set)
proof (unfold image-def Abs'-def, auto)
  fix x show \exists xa\in\{\theta..<\text{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

lemma bij-from-nat: bij-btw (from-nat) \{\theta..<\text{CARD}'(a)\} (UNIV::'a set)
proof (unfold bij-btw-def, rule conjI)
  have set-eq: \{\theta::int..<\text{CARD}'(a)\} = int\' \{\theta..<\text{CARD}'(a)\} apply (auto)
  proof
    fix x::int assume x1: (\theta::int) \leq x and x2: x < int CARD'(a) show x \in int
      \{\theta::nat..<\text{CARD}'(a)\}
    proof (unfold image-def, auto, rule bexI[of - nat x])
      show x = int (nat x) using x1 by auto
      show nat x \in \{\theta::nat..<\text{CARD}'(a)\} using x1 x2 by auto
    qed
      qed
  show inj-on (from-nat::nat\rightarrow'a) \{\theta::nat..<\text{CARD}'(a)\}
proof (unfold from-nat-def , rule comp-inj-on)
  show inj-on int \{\theta::nat..<\text{CARD}'(a)\} by (metis inj-of-nat subset-inj-on top-greatest)
  show inj-on (Abs':\'int\rightarrow'a) (int\' \{\theta::nat..<\text{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)* \{\theta::nat..<\text{CARD}'(a)\} = UNIV
proof 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 \{\theta..<\text{CARD}'(a)\} (from-nat::nat\rightarrow'a) = (to-nat::'a\rightarrow
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 \{\theta::nat..<\text{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 (\theta::int) \leq Rep x show (\theta::nat) < CARD'(a) and Abs' (\theta::int) = x
    using Rep-ge-0 x by auto
next

59
fix y :: nat assume y : y < CARD('a)

have (Rep(Abs'(int y)::'a)) = (Abs'(int y mod int CARD('a))::'a)) unfolding Abs'-'def

also have ... = (Rep (Abs (int y)::'a)) unfolding zmod-int[of y CARD('a)] using y mod-less by auto

also have ... = (int y) proof (rule Abs-inverse) show int y ∈ {0::int..<int CARD('a)}

using y by auto qed

finally show y = nat (Rep (Abs (int y)::'a)) by (metis nat-int) qed

lemma bij-to-nat: bij-betw (to-nat) (UNIV::'a set) {0..<CARD('a)}

using bij-betw-the-inv-into[OF bij-from-nat]

unfolding to-nat-is-inv.

lemma finite-mod-type: finite (UNIV::'a set)

using finite-imageD[of to-nat UNIV::'a set] using bij-to-nat unfolding bij-betw-def

by auto

subclass (in mod-type) finite by (intro-classes, rule finite-mod-type)

lemma least-0: (LEAST n. n ∈ (UNIV::'a set)) = 0

proof (rule Least-equality, auto)

fix y :: 'a

have (0::'a) ≤ Abs (Rep y mod int CARD('a)) unfolding strict-mono-Rep

strict-mono-def

by (metis (hide-lams, mono-tags) Rep-0 Rep-ge-0 strict-mono-Rep strict-mono-less-eq)

also have ... = y by (metis Rep-inverse Rep-mod)

finally show (0::'a) ≤ y .

qed

lemma add-to-nat-def: x + y = from-nat (to-nat x + to-nat y)

unfolding from-nat-def to-nat-def o-def unfolding Rep-ge-0[of x] using Rep-ge-0[of y]

using Rep-less-n[of x] Rep-less-n[of y]

unfolding Abs'-'def unfolding add-def[of x y] by auto

lemma to-nat-1: to-nat 1 = 1

by (metis (hide-lams, mono-tags) Rep-1 comp-apply to-nat-def transfer-nat-int-numerals(2))

lemma add-def':

shows x + y = Abs' (Rep x + Rep y) unfolding Abs'-'def using add-def by simp

lemma Abs'-'0:

shows Abs' (CARD('a)) = (0::'a) by (metis (hide-lams, mono-tags) Abs'-'def mod-self zero-def)

lemma Rep-plus-one-le-card:
assumes $a \colon a + 1 \neq 0$

shows $(\Rep a) + 1 < \CARD('a)$

proof (rule ccontr)

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

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

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

also have $\ldots = \Abs 'a ((\Rep a) + 1)$ using Rep-1 simp

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

also have $\ldots = 0$ using Abs'-0 by auto

finally show False using a by contradiction

qed

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

proof (clarify)

fix $a$

assume $a + 1 \neq 0$

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

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

thus $to-nat a + 1 < \CARD('a)$ 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 $to-nat a + 1 < \CARD('a)$ using to-nat-plus-one-less-card assms simp

lemma strict-mono-to-nat: strict-mono 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]: $\forall x. x = to-nat y \leftrightarrow x = y$

using injD [OF bij_betw_imp_inj_on[OF bij-to-nat]] by blast

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

proof (auto)

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

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

thus $P a$ using a by auto

qed

lemma to-nat-from-nat:

assumes $t::to-nat j = k$

shows from-nat $k = j$

proof –
have from-nat \( k = \text{from-nat} \ (\text{to-nat} \ j) \) unfolding \( t \) ..
also have \( \ldots = \text{from-nat} \ (\text{the-inv-into} \ \{0..<\text{CARD}(\alpha)\} \ (\text{from-nat}) \ j) \) unfolding to-nat-is-inv ..
also have \( \ldots = j \)
proof (rule f-the-inv-into-f)
  show inj-on from-nat \( \{0..<\text{CARD}(\alpha)\} \) by (metis bij-betw-imp-inj-on bij-from-nat)
  show \( j \in \text{from-nat} \ (\{0..<\text{CARD}(\alpha)\}) \) by (metis 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 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 to-nat \( a \leq \text{to-nat} \ b \)
proof (cases \( a=b \))
case \( \text{True} \) thus \( ?\text{thesis} \) by auto
next
case \( \text{False} \)
hence \( a< b \) using \( ab \) by simp
thus \( ?\text{thesis} \) using to-nat-mono by fastforce
qed

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

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

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

**lemma** from-nat-to-nat:
assumes \( \text{i-le-j} : i < j \) and \( j < \text{CARD}(\alpha) \)
shows to-nat \( k = j \) by (metis \( j \) \( t \) to-nat-from-nat-id)

**lemma** from-nat-mono:
assumes \( i\leq j \) : \( i < j \) and \( j < \text{CARD}(\alpha) \)
shows (from-nat i::'a) < from-nat j
proof –
have i: i<CARD('a) using i-le-j by simp
obtain a where a: i=to-nat a
  using bij-to-nat unfolding bij-betw-def using i to-nat-from-nat-id by metis
obtain b where b: j=to-nat b
  using bij-to-nat unfolding bij-betw-def using j to-nat-from-nat-id by metis
show ?thesis by (metis a b from-nat-to-nat-id i-le-j strict-mono-less strict-mono-to-nat)
qed

lemma from-nat-mono':
  assumes i-le-j: i ≤ j and j<CARD ('a)
  shows (from-nat i::'a) ≤ from-nat j
proof (cases i=j)
  case True
  have (from-nat i::'a) = from-nat j using True by simp
  thus ?thesis by simp
next
  case False
  hence i< j using i-le-j by simp
  thus ?thesis by (metis assms (2) from-nat-mono less-imp-le)
qed

lemma to-nat-suc:
  assumes to-nat (x)+1 < CARD ('a)
  shows to-nat (x + 1::'a) = (to-nat x) + 1
proof –
  have (x::'a) + 1 = from-nat (to-nat x + to-nat (1::'a)) unfolding add-to-nat-def
  ..
  hence to-nat ((x::'a) + 1) = to-nat (from-nat (to-nat x + to-nat (1::'a))::'a)
  by presburger
  also have ... = to-nat (from-nat (to-nat x + 1)::'a) unfolding to-nat-1 ..
  also have ... = (to-nat x + 1) by (metis assms to-nat-from-nat-id)
  finally show ?thesis .
qed

lemma to-nat-le:
  assumes y < from-nat k
  shows to-nat y < k
proof (cases k<CARD('a))
  case True show ?thesis by (metis (full-types) True assms to-nat-from-nat-id to-nat-mono)
next
  case False have to-nat y < CARD ('a) using bij-to-nat unfolding bij-betw-def by auto
  thus ?thesis using False by auto
qed

lemma 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 \( ... \leq (\text{from-nat} \ (\text{to-nat} \ (b :: 'a)) :: 'a) \)
  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 \( \text{to-nat} \ a + 1 < \text{CARD} \ ('a) \) 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 \( \text{to-nat} \ a + 1 \leq \text{to-nat} \ b \)
  to-nat-suc)
  
  qed

finally show \( a + (1 :: 'a) \leq b \).

qed

lemma \( \text{le-Suc} : \)
assumes \( ab : a + 1 \leq b \)
and \( \text{less-card} : (\text{to-nat} \ a) + 1 < \text{CARD} \ ('a) \)
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 \( ... < (\text{from-nat} \ (\text{to-nat} \ b)) :: 'a \)
  proof (rule from-nat-mono)
    show \( \text{to-nat} \ b < \text{CARD} ('a) \) 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 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 \( \text{Suc-le} : \)
assumes \( \text{less-card} : (\text{to-nat} \ a) + 1 < \text{CARD} ('a) \)
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 \( \text{Suc-le}' : \)
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-monono[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 $a \leq -1$ by fastforce
qed

lemma a-eq-minus-1: $\forall a :: 'a. a + 1 = 0 \rightarrow a = -1$
by (metis eq-neg-iff-add-eq-0)

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

lemma from-nat-eq-imp-eq:
assumes f-eq: $\text{from-nat} x = \text{from-nat} xa :: 'a$
and x: $x < \text{CARD}('a)$ and xa: $xa < \text{CARD}('a)$
shows $x = xa$ 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: $\text{from-nat} 0 = 0$
unfolding from-nat-def o-def of-nat-0 Abs' - def mod-0 zero-def ..
lemma to-nat-0: $\text{to-nat} 0 = 0$ unfolding to-nat-def o-def Rep-0 nat-0 ..
lemma to-nat-eq-0: $(\text{to-nat} x = 0) = (x = 0)$ using to-nat-0 to-nat-from-nat by auto

lemma suc-not-zero:
assumes to-nat a + 1 \neq \text{CARD}('a)
shows a + 1 \neq 0
proof (rule ccontr, simp)
  assume a-plus-one-zero: $a + 1 = 0$
  hence rep-eq-card: $\text{Rep} a + 1 = \text{CARD}('a)$
  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 $\text{Rep} a + 1 < \text{CARD}('a)$
  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

66
lemma from-nat-suc:
shows from-nat \((j + 1) = from-nat j + 1\)
unfolding from-nat-def a-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

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
by (metis bit0.of-nat-eq of-nat-1 one-bit0-def)
fix x y :: 'a bit0
show \(x + y = Abs ((Rep x + Rep y) mod int CARD('a bit0))\)
unfolding Abs-bit0-def Rep-bit0-def plus-bit0-def Abs-bit0'-def by fastforce

qed
show $x \ast y = \text{Abs}\ (\text{Rep}\ x \ast \text{Rep}\ y \mod\ \text{int}\ \text{CARD}(\text{abs1}))$
  unfolding \text{Abs-bit1-def Rep-bit1-def times-bit1-def Abs-bit1'-def} by fastforce
show $x = y = \text{Abs}\ ((\text{Rep}\ x - \text{Rep}\ y) \mod\ \text{int}\ \text{CARD}(\text{abs1}))$
  unfolding \text{Abs-bit1-def Rep-bit1-def minus-bit1-def Abs-bit1'-def} by fastforce
show $-x = \text{Abs}\ (-\text{Rep}\ x \mod\ \text{int}\ \text{CARD}(\text{abs1}))$
  unfolding \text{Abs-bit1-def Rep-bit1-def uminus-bit1-def Abs-bit1'-def} by fastforce
show $(0::\text{a\ bit1}) = \text{Abs}\ (0::\text{int})$ unfolding \text{Abs-bit1-def Abs-bit1'-def zero-bit1-def}
by auto
show $(1::\text{int}) < \text{int}\ \text{CARD}(\text{abs1})$ by (metis abs1.size1)
show $(1::\text{a\ bit1}) = \text{Abs}\ (1::\text{int})$ unfolding \text{Abs-bit1-def Abs-bit1'-def one-bit1-def}
  by (metis abs1.of-nat-eq of-nat-1 one-bit1-def)
fix $x\ y::\text{a\ bit1}$
show $x + y = \text{Abs}\ ((\text{Rep}\ x + \text{Rep}\ y) \mod\ \text{int}\ \text{CARD}(\text{abs1}))$
  unfolding \text{Abs-bit1-def Abs-bit1'-def Rep-bit1-def plus-bit1-def} by fastforce
show $x \ast y = \text{Abs}\ (\text{Rep}\ x \ast \text{Rep}\ y \mod\ \text{int}\ \text{CARD}(\text{abs1}))$
  unfolding \text{Abs-bit1-def Rep-bit1-def times-bit1-def Abs-bit1'-def} by fastforce
show $x - y = \text{Abs}\ ((\text{Rep}\ x - \text{Rep}\ y) \mod\ \text{int}\ \text{CARD}(\text{abs1}))$
  unfolding \text{Abs-bit1-def Rep-bit1-def minus-bit1-def Abs-bit1'-def} by fastforce
show $-x = \text{Abs}\ (-\text{Rep}\ x \mod\ \text{int}\ \text{CARD}(\text{abs1}))$
  unfolding \text{Abs-bit1-def Rep-bit1-def uminus-bit1-def Abs-bit1'-def} by fastforce
show type-definition $(\text{Rep}::\text{a\ bit1} => \text{int})\ (\text{Abs}::\text{int} => 'a\ \text{bit1})\ \{0::\text{int}..<\text{int}\ \text{CARD}(\text{abs1})\}$
proof (unfold type-definition-def Rep-bit1-def [abs-def]
  Abs-bit1-def [abs-def] Abs-bit1'-def, intro conjI)
  have int-2: $\text{int}\ 2 = 2$ by auto
show $\forall x::\text{a\ bit1}.\ \text{Rep-bit1}\ x \in \{0::\text{int}..<\text{int}\ \text{CARD}(\text{abs1})\}$
  unfolding card-bit1
  unfolding int-Suc int-mult
  using Rep-bit1 [where ?'a = 'a] unfolding int-2 unfolding add.commute
.. show $\forall x::\text{a\ bit1}.\ \text{Abs-bit1}\ (\text{Rep-bit1}\ x \mod\ \text{int}\ \text{CARD}(\text{abs1})) = x$
  by (metis Rep-bit1-inverse bit1.Rep-mod)
show $\forall y::\text{int}.\ y \in \{0::\text{int}..<\text{int}\ \text{CARD}(\text{abs1})\}$
  $\to\ \text{Rep-bit1}\ ((\text{Abs-bit1}::\text{int} => 'a\ \text{bit1})\ (y \mod\ \text{int}\ \text{CARD}(\text{abs1}))) = y$
  by (metis bit1.Abs-inverse bit1.Rep-mod)
qed
show strict-mono $(\text{Rep}::\text{a\ bit0} => \text{int})$ unfolding strict-mono-def
  by (metis Rep-bit0-def less-bit0-def)
show strict-mono $(\text{Rep}::\text{a\ bit1} => \text{int})$ unfolding strict-mono-def
  by (metis Rep-bit1-def less-bit1-def)
qed
end

4 Miscellaneous

theory Miscellaneous
imports
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

\textbf{definition} \texttt{nrows} :: 'a::{columns, rows} => \texttt{nat}
\textbf{where} \texttt{nrows A = CARD(rows)}

\textbf{definition} \texttt{ncols} :: 'a::{columns, rows} => \texttt{nat}
\textbf{where} \texttt{ncols A = CARD(columns)}

\textbf{definition} \texttt{matrix-scalar-mult} :: 'a => ('a::{semiring-1, m}) => ('a::{semiring-1, n}) => ('a::{semiring-1, m})
\textbf{where} \texttt{k * k A ≡ (χ i j. k * A[i][j])}

4.2 Basic properties about matrices

\textbf{lemma} \texttt{nrows-not-0[simp]}
\textbf{shows} \texttt{0 ≠ nrows A unfolding nrows-def by simp}

\textbf{lemma} \texttt{ncols-not-0[simp]}
\textbf{shows} \texttt{0 ≠ ncols A unfolding ncols-def by simp}

\textbf{lemma} \texttt{nrows-transpose: nrows (transpose A) = ncols A}
\textbf{unfolding} \texttt{nrows-def ncols-def ..}

\textbf{lemma} \texttt{ncols-transpose: ncols (transpose A) = nrows A}
\textbf{unfolding} \texttt{nrows-def ncols-def ..}

\textbf{lemma} \texttt{finite-rows: finite (rows A)}
\textbf{using} \texttt{finite-Atleast-Atmost-nat[of λi. row i A] unfolding rows-def .}

\textbf{lemma} \texttt{finite-columns: finite (columns A)}
\textbf{using} \texttt{finite-Atleast-Atmost-nat[of λi. column i A] unfolding columns-def .}

\textbf{lemma} \texttt{matrix-vector-zero: A * v 0 = 0}
\textbf{unfolding} \texttt{matrix-vector-mult-def by (simp add: zero-vec-def)}

\textbf{lemma} \texttt{vector-matrix-zero: 0 * v A = 0}
\textbf{unfolding} \texttt{vector-matrix-mult-def by (simp add: zero-vec-def)}

\textbf{lemma} \texttt{vector-matrix-zero': x * v 0 = 0}
\textbf{unfolding} \texttt{vector-matrix-mult-def by (simp add: zero-vec-def)}
lemma \texttt{transpose-vector}: \(x \cdot v \cdot A = \text{transpose } A \cdot v \cdot x\)

\textbf{by} (unfold matrix-vector-mult-def vector-matrix-mult-def transpose-def, auto)

lemma \texttt{transpose-zero}[simp]: (transpose \(A = 0\)) = (\(A = 0\))

\textbf{unfolding} transpose-def zero-vec-def vec-eq-iff \textbf{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/Tarskis_Geometry/Linear_Algebra2.html. I have removed some restrictions over the type classes.

\textbf{lemma} \texttt{vector-matrix-left-distrib}:

\textbf{shows} \((x + y) \cdot v \cdot A = x \cdot v \cdot A + y \cdot v \cdot A\)

\textbf{unfolding} vector-matrix-mult-def

\textbf{by} (simp add: algebra-simps setsum.distrib vec-eq-iff)

\textbf{lemma} \texttt{matrix-vector-right-distrib}:

\textbf{shows} \(M \cdot v \cdot (v + w) = M \cdot v \cdot v + M \cdot v \cdot w\)

\textbf{proof} –

\textbf{have} \(M \cdot v \cdot (v + w) = (v + w) \cdot v \cdot \text{transpose } M\) \textbf{by} (metis transpose-transpose transpose-vector)

\textbf{also have} \(= v \cdot \text{transpose } M + w \cdot v \cdot \text{transpose } M\)

\textbf{by} (rule vector-matrix-left-distrib [of v w transpose M])

\textbf{finally show} \(M \cdot v \cdot (v + w) = M \cdot v \cdot v + M \cdot v \cdot w\) \textbf{by} (metis transpose-transpose transpose-vector)

\textbf{qed}

\textbf{lemma} \texttt{scalar-vector-matrix-assoc}:

\textbf{fixes} \(k :: 'a::{field} \text{ and } x :: 'a::{field} \cdot 'n \text{ and } A :: 'a \cdot 'm \cdot 'n\)

\textbf{shows} \((k \cdot s \cdot x) \cdot v \cdot A = k \cdot s \cdot (x \cdot v \cdot A)\)

\textbf{unfolding} vector-matrix-mult-def \texttt{unfolding} vec-eq-iff

\textbf{by} (auto simp add: setsum-right-distrib, rule setsum.cong, simp-all)

\textbf{lemma} \texttt{vector-scalar-matrix-ac}:

\textbf{fixes} \(k :: 'a::{field} \text{ and } x :: 'a::{field} \cdot 'n \text{ and } A :: 'a \cdot 'm \cdot 'n\)

\textbf{shows} \(x \cdot v \cdot (k \cdot s \cdot A) = k \cdot s \cdot (x \cdot v \cdot A)\)

\textbf{using} scalar-vector-matrix-assoc

\textbf{unfolding} vector-matrix-mult-def matrix-scalar-mult-def vec-eq-iff

\textbf{by} (auto simp add: setsum-right-distrib)

\textbf{lemma} \texttt{transpose-scalar}:

\textbf{transpose} \((k \cdot k \cdot A) = k \cdot k \cdot \text{ transpose } A\)

\textbf{unfolding} transpose-def

\textbf{by} (vector, simp add: matrix-scalar-mult-def)
lemma scalar-matrix-vector-assoc:
  fixes A :: 'a::{field} ⇒ m ⇒ n
  shows k * s (A * v v) = k * k A * v v
proof
  have k * s (A * v v) = k * s (v v * transpose A) by (metis transpose-transpose transpose-vector)
  also have ... = v v * (k * k transpose A)
    by (rule vector-scalar-matrix-ac [symmetric])
  also have ... = v v * transpose (k * k A) unfolding transpose-vector ..
  finally show k * s (A * v v) = k * k A * v v by (metis transpose-transpose)
qed

lemma matrix-scalar-vector-ac:
  fixes A :: 'a::{field} ⇒ m ⇒ n
  shows A * v (k * s v) = k * k A * v v
proof
  have A * v (k * s v) = k * s (v v * transpose A)
    by (metis transpose-transpose scalar-vector-matrix-assoc)
  also have ... = v v * (k * k transpose A)
    by (subst vector-scalar-matrix-ac)
  also have ... = v v * transpose (k * k A)
    by (subst transpose-scalar)
  finally show A * v (k * s v) = k * k A * v 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 ⟨card S = CARD('n)⟩ have card S ≠ 0 by simp
  with card-eq-0-iff [of S] show finite S by simp
qed

lemma independent-is-basis:
  fixes B :: ('a::{field} ⇒ 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 \( \text{card } B = \text{vec.dim } (\text{UNIV} :: ((\text{’a}^\cdot \text{’n}) \text{ set})) \) 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 \( \text{card } B = \text{CARD}(\text{’n}) \)

proof –

have \( B \subseteq \text{UNIV} \) by simp

moreover

\{ from (is-basis B) have UNIV \subseteq \text{vec.span } B \) and (vec.independent B)

unfolding is-basis-def

by simp+ \}

ultimately have \( \text{card } B = \text{vec.dim } (\text{UNIV} :: ((\text{real}^\cdot \text{’n}) \text{ set})) \)

using vec.basis-card-eq-dim [of B UNIV]

unfolding vec-dim-card

by simp

then show \( \text{card } B = \text{CARD}(\text{’n}) \)

by (metis vec-dim-card)

qed

ultimately show vec.independent B \( \land \) \( \text{card } B = \text{CARD}(\text{’n}) \)

qed

lemma basis-finite:

fixes \( B :: (\text{’a} :: \{\text{field}\}^\cdot \text{’n}) \text{ set} \)

assumes is-basis B

shows finite \( B \)

proof –

from independent-is-basis [of B] and (is-basis B) have \( \text{card } B = \text{CARD}(\text{’n}) \)

by simp

with \( \text{card-finite } \) [of \( B \), where \( \text{’n} = \text{’n} \)] show \( \text{finite } B \) 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 \( \text{local.independent } ?B \implies \text{finite } ?B \land \text{ card } ?B = \text{local.dim } (\text{local.span } ?B) \)

lemma card-eq-dim-span-indep:

assumes \( \text{dim } (\text{span } A) = \text{card } A \) and finite A

shows independent A

72
by (metis assms card-le-dim-spanning dim-subset equalityE span-inc)

lemma dim-zero-eq:

assumes dim-A: dim A = 0
shows A = {} ∨ A = {0}
proof –
obtain B where ind-B: independent B and A-in-span-B: A ⊆ 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 ind-B] by simp
hence B-eq-empty: B = {} using card-B unfolding card-eq-0-iff by simp
have A ⊆ {0} using A-in-span-B unfolding B-eq-empty span-empty .
thus ?thesis by blast
qed

lemma dim-zero-eq':

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

lemma dim-zero-subspace-eq:

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

lemma span-0-imp-set-empty-or-0:

assumes span: span A = {0}
shows A = {} ∨ 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


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 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
also have ... = setsum (λ i. x $ i * s column i A) (UNIV - (?g' (columns A))) unfolding union-univ ..
also have ... = setsum ((λ i. x $ i * s column i A)) (?g' (columns A)) unfolding setsum.setsum_univ
by (rule setsum.Union-disjoint[unfolded o-def], auto)
also have ... = setsum ((λ i. x $ i * s column i A)) o ?g (columns A)
by (rule setsum.reindex, simp add: inj)
also have ... = setsum (λ y. if y * s y) (columns A)
proof (rule setsum.cong, unfold o-def)
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 some-l-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.
In the library, \( \text{matrix-inv } ?A = (\text{SOME } A') \) \( ?A \iff A' = \text{mat } (1::'?a) \land A' \) \( \iff ?A = \text{mat } (1::'?a) \) allows the use of non square matrices. The following lemma can be also proved fixing \( A \):

**lemma** matrix-inv-unique:

**fixes** \( A::'?a::\{\text{semiring-1}\} ^\langle n \cdot n \rangle \)

**assumes** \( AB: A \iff B = \text{mat } 1 \) \text{ and } BA: \( B \iff A = \text{mat } 1 \)

**shows** matrix-inv \( A = B \)

**proof** (unfold matrix-inv-def, rule some-equality)

show \( A \iff B = \text{mat } 1 \) \text{ and } \( B \iff A = \text{mat } 1 \)

hence \( AC: A \iff C = \text{mat } 1 \) \text{ and } CA: \( C \iff A = \text{mat } 1 \) by auto

have \( B = B \iff (\text{mat } 1) \) unfolding matrix-mul-rid ..

also have \( ... = B \iff (A**C) \) unfolding AC ..

also have \( ... = B \iff A \iff C \) unfolding matrix-mul-assoc ..

also have \( ... = C \) unfolding BA matrix-mul-lid ..

finally show \( C = B .. \)

qed

**lemma** matrix-vector-mult-zero-eq:

**assumes** \( P: \text{invertible } P \)

**shows** \((P**A) \cdot v x = 0\) \iff \((A \cdot v x = 0)\)

**proof** (rule iffI)

assume \( P \iff A \cdot v x = 0 \)

hence matrix-inv \( P \cdot v \) \( (P \iff A \cdot v x) = \text{matrix-inv } P \cdot v \) \( 0 \) by simp

hence matrix-inv \( P \cdot v \) \( (P \iff A \cdot v x) = 0 \) by (metis matrix-vector-zero)

hence \( (\text{matrix-inv } P \iff P \iff A) \cdot v x \iff 0 \) by (metis matrix-vector-mul-assoc)

thus \( A \cdot v x = 0 \) by (metis assms matrix-inv-left matrix-mul-lid)

next

assume \( A \cdot v x = 0 \)

thus \( P \iff A \cdot v x = 0 \) by (metis matrix-vector-mul-assoc matrix-vector-zero)

qed

**lemma** inj-matrix-vector-mult:

**fixes** \( P::'?a::\{\text{field}\} ^\langle n \cdot m \rangle \)

**assumes** \( P: \text{invertible } P \)

**shows** \( \text{inj } (\text{op } \cdot v P) \)

unfolding \( \text{vec.linear-injective-0} \)

using \( \text{matrix-left-invertible-ker[of } P] \) \( \text{unfolding } \text{invertible-def by blast} \)

**lemma** independent-image-matrix-vector-mult:

**fixes** \( P::'?a::\{\text{field}\} ^\langle n \cdot m \rangle \)

**assumes** \( \text{ind-B: vec.independent } B \) \text{ and } \( \text{inv-P: invertible } P \)

**shows** \( \text{vec.independent } ((\text{op } \cdot v P) \cdot B) \)

**proof** (rule vec.independent-injective-on-span-image)

show \( \text{vec.independent } B \) using \( \text{ind-B} \).

76
show \textit{inj-on} (\textit{op} *v \textit{P}) (\textit{vec.span} \textit{B})

using \textit{inj-matrix-vector-mult}\[\textit{OF inv-P}\] unfolding \textit{inj-on-def} by simp

qed

\textbf{lemma} \textit{independent-preimage-matrix-vector-mult}: 
\textbf{fixes} \textit{P}::'\textit{a::field}''\textit{n}''\textit{n} 
\textbf{assumes} \textit{ind-B}: \textit{vec.independent} ((\textit{op} *v \textit{P})' \textit{B}) \textbf{and} \textit{inv-P}: invertible \textit{P} 
\textbf{shows} \textit{vec.independent} \textit{B} 
\textbf{proof} –
\textbf{have} \textit{vec.independent} ((\textit{op} *v (\textit{matrix-inv} \textit{P})){'} ((\textit{op} *v \textit{P})' \textit{B}))
\textbf{proof} (rule independent-image-matrix-vector-mult)
\textbf{show} \textit{vec.independent} (\textit{op} *v \textit{P} B) using \textit{ind-B}.
\textbf{show} invertible (\textit{matrix-inv} \textit{P})
\textbf{by} (metis \textit{matrix-inv-left} \textit{matrix-inv-right} \textit{inv-P} \textit{invertible-def})
\textbf{qed}

moreover \textbf{have} (\textit{op} *v (\textit{matrix-inv} \textit{P})){'} ((\textit{op} *v \textit{P})' \textit{B}) = \textit{B} 
\textbf{proof} (auto)
\textbf{fix} \textit{x} assume \textit{x}: \textit{x} \in \textit{B} show \textit{matrix-inv} \textit{P} *v \textit{P} *v \textit{x} \in \textit{B}
\textbf{by} (metis \textit{full-types} \textit{x} \textit{inv-P} \textit{matrix-inv-left} \textit{matrix-vector-mul-assoc} \textit{matrix-vector-mul-lid})
\textbf{thus} \textit{x} \in \textit{op} *v (\textit{matrix-inv} \textit{P})' \textit{op} *v \textit{P}' \textit{B}
unfolding \textit{image-def}
\textbf{by} (auto, metis \textit{inv-P} \textit{matrix-inv-left} \textit{matrix-vector-mul-assoc} \textit{matrix-vector-mul-lid})
\textbf{qed}

ultimately \textbf{show} \textbf{?thesis} by simp

qed

\textbf{4.7 Properties about the dimension of vectors}

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

\textbf{qed}
4.8 Instantiations and interpretations

Functions between two real vector spaces form a real vector

**instantiation** fun :: (real-vector, real-vector) real-vector

**begin**

**definition** plus-fun f g = (λ i. f i + g i)
**definition** zero-fun = (λ i. 0)
**definition** scaleR-fun a f = (λ i. a *R f i)

**instance** proof

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

fix a::'a ⇒ 'b and x::('a ⇒ 'b) and y::('a ⇒ 'b)
unfolding fun-eq-iff plus-fun-def scaleR-fun-def scaleR-right.add by auto

fix a::real and x::('a ⇒ 'b) and y::('a ⇒ 'b)
show a + b + x = a + y unfolding fun-eq-iff plus-fun-def scaleR-fun-def scaleR-left.add by auto

fix a::real and b::real and x::('a ⇒ 'b)
show (a + b) *R x = a *R (b *R x)
unfolding fun-eq-iff plus-fun-def scaleR-fun-def unfolding scaleR-fun-def by auto

show (1::real) *R x = x unfolding fun-eq-iff unfolding scaleR-fun-def by auto

qed

end

**instantiation** vec :: (type, finite) equal

**begin**

**definition** equal-vec :: ('a, 'b::finite) vec => ('a, 'b::finite) vec => bool

where equal-vec x y = (∀ i. x$i = y$i)

**instance** proof (intro-classes)

fix x y::('a, 'b::finite) vec

show equal-class.equal x y = (x = y) unfolding equal-vec-def using vec-eq-iff by auto

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⇒'a⇒'a⇒'a⇒'a rows
proof (unfold-locales)
fix a::'a and x y::'a⇒'a⇒'a 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-itsel.scale-right-distrib)
next
fix a b::'a and x::'a⇒'a⇒'a 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
~/sr/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} 'n columns 'rows
shows row-space $A = \text{col-space (transpose } A)$
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 $A = \text{row-space (transpose } A)$
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 using 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 $A :: 'a::{field} ^'n :: {finite, wellorder} ^'m$
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 matrix-vmult-column-sum[of $A x$] by auto
next
\textbf{fix } \( w \colon (\text{', } n) \text{ vec } \Rightarrow \text{'a} \)
\textbf{let } \( \lambda g \colon \{ i. \ y = \text{column } i \ A \} \)
\textbf{let } \( \lambda x \colon (\chi i. \ if \ i = (\text{LEAST } a. \ a \in ?g \ (\text{column } i \ A)) \ then \ u \ (\text{column } i \ A) \ else \ \theta) \)
\textbf{show } \( \exists x. \ A \ast v \ x = (\sum v \in \text{columns } A. \ u \ v \ast s \ v) \)
\textbf{proof } (\text{unfold matrix-mull-vsum, rule exI[af - ?x], auto})
\begin{align*}
& \text{have inj: } \text{inj-on } ?y \ (\text{columns } A) \ \text{unfolding in-j-on-def unfolding columns-def by auto} \\
& \quad \text{have union-univ: } \bigcup(?g'(\text{columns } A)) = \text{UNIV unfolding columns-def by auto} \\
& \quad \text{have setsum } (\lambda i. (if \ i = (\text{LEAST } a. \ \text{column } i \ A = \text{column } a \ A) \ then \ u \ (\text{column } i \ A) \ else \ \theta) \ast s \ \text{column } i \ A) \ UNIV \\
& \quad \quad = \text{setsum } (\lambda i. (if \ i = (\text{LEAST } a. \ \text{column } i \ A = \text{column } a \ A) \ then \ u \ (\text{column } i \ A) \ else \ \theta) \ast s \ \text{column } i \ A) \ (\bigcup(?g'(\text{columns } A))) \\
& \quad \text{unfolding union-univ ..} \\
& \quad \text{also have } ... = \text{setsum } (\text{setsum } (\lambda i. (if \ i = (\text{LEAST } a. \ \text{column } i \ A = \text{column } a \ A) \ then \ u \ (\text{column } i \ A) \ else \ \theta) \ast s \ \text{column } i \ A)) \ (?g'(\text{columns } A)) \\
& \quad \quad \text{by } (\text{rule setsum.Union-disjoint[unfolded o-def], auto}) \\
& \quad \text{also have } ... = \text{setsum } ((\text{setsum } (\lambda i. (if \ i = (\text{LEAST } a. \ \text{column } i \ A = \text{column } a \ A) \ then \ u \ (\text{column } i \ A) \ else \ \theta) \ast s \ \text{column } i \ A)) \circ \ ?g) \\
& \quad \quad \text{(columns } A) \ \text{by } (\text{rule setsum.reindex, simp add: inj}) \\
& \quad \text{also have } ... = \text{setsum } (\lambda y. \ u \ y \ast s \ y) \ (\text{columns } A) \\
& \quad \text{proof } (\text{rule setsum.cong, auto}) \\
& \text{fix } x \\
& \text{assume } x\text{-in-cols: } x \in \text{columns } A \\
& \text{obtain } b \ \text{where } b \colon \text{column } b \ A \ \text{using } x\text{-in-cols unfolding columns-def by blast} \\
& \quad \text{let } ?f = (\lambda i. (if \ i = (\text{LEAST } a. \ \text{column } i \ A = \text{column } a \ A) \ then \ u \ (\text{column } i \ A) \ else \ \theta) \ast s \ \text{column } i \ A) \\
& \quad \text{have setsum-\text{rw: setsum } ?f \ ((\{i. \ x = \text{column } i \ A\} - \{\text{LEAST } a. \ x = \text{column } a \ A\}) = 0} \\
& \quad \quad \text{by } (\text{rule setsum.neutral, auto}) \\
& \quad \text{have setsum } ?f \ (i. \ x = \text{column } i \ A) = ?f \ (\text{LEAST } a. \ x = \text{column } a \ A) + \\
& \quad \quad \text{setsum } ?f \ ((i. \ x = \text{column } i \ A) - \{\text{LEAST } a. \ x = \text{column } a \ A\}) \\
& \quad \quad \text{apply } (\text{rule setsum.remove, auto, rule LeastI-ex}) \\
& \quad \text{using } x\text{-in-cols unfolding columns-def by auto} \\
& \quad \text{also have } ... = ?f \ (\text{LEAST } a. \ x = \text{column } a \ A) \ \text{unfolding setsum-\text{rw by simp} ..} \\
& \quad \text{also have } ... = u \ x \ast s \ x \\
& \quad \text{proof } (\text{auto, rule LeastI2}) \\
& \quad \text{show } x = \text{column } b \ A \ \text{using } b . \\
& \quad \text{fix } xa \\
& \quad \text{assume } x: \ x = \text{column } xa \ A \\
& \quad \text{show } u \ (\text{column } xa \ A) \ast s \ \text{column } xa \ A = u \ x \ast s \ x \ \text{unfolding } x .. \\
& \text{next} \\
& \text{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) \\
& \text{moreover have } (\text{LEAST } a. \ x = \text{column } a \ A) = (\text{LEAST } a. \ \text{column } \ (\text{LEAST } c. \ x = \text{column } c \ A) \ A = \text{column } a \ A) \\
& \quad \text{by } (\text{rule Least-equality[symmetric], rule LeastI2, simp-all add: b, rule} \ )
Least-le, metis (lifting, full-types) LeastI
ultimately show \( u \times 0 \) by contradiction
qed
finally show \( \sum_{i \mid x = \text{column } i A. (i = \text{LEAST } a. \text{column } i A = \text{column } a A)} u (\text{column } i A) \times s \text{column } i A = u \times s x \).
qed
finally show \( \sum_{i \in \text{UNIV. } (i = \text{LEAST } a. \text{column } i A = \text{column } a A)} u (\text{column } i A) \times s \text{column } i A = (\sum_{y \in \text{columns } A. u \times s y}) \).
qed

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

**lemma** row-space-eq:
fixes \( A :: 'a :: \{\text{field}\} \cdot m :: \{\text{finite, wellorder}\} \cdot n \)
shows row-space \( A = \{w. \exists y. (\text{transpose } A) \times v y = w\} \)
unfolding row-space-eq-col-space-transpose col-space-eq ..

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

**lemma** col-space-eq-range:
fixes \( f :: ('a :: \text{field}) \cdot m => ('a \cdot m) \)
assumes \( \text{lf: linear op } \times s \text{ op } \times s f \)
shows col-space (matrix \( f \)) = \text{range } f
unfolding col-space-eq unfolding matrix-works\([\text{OF } \text{lf}]\) by blast

**lemma** null-space-is-preserved:
fixes \( A :: 'a :: \{\text{field}\} \cdot \text{cols} \cdot \text{rows} \)
assumes \( \text{P: invertible } P \)
shows null-space \( (P \cdot \times A) = \text{null-space } A\)
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 \( A :: 'a :: \{\text{field}\} \cdot \text{cols} \cdot \text{rows} :: \{\text{finite, wellorder}\} \)
and \( P :: 'a :: \{\text{field}\} \cdot \text{rows} :: \{\text{finite, wellorder}\} \cdot \text{rows} :: \{\text{finite, wellorder}\} \)
assumes \( \text{P: invertible } P \)
shows row-space \( (P \cdot \times A) = \text{row-space } A\)
proof (auto)
fix \( w \)

83
assume \( w: w \in \text{row-space } (P\ast^\ast A) \)

from this obtain \( y \) where \( w-By: w=(\text{transpose } (P\ast^\ast A)) \ast v y \)

unfolding row-space-eq[of \( P \ast^\ast A \)] by fast

have \( w = (\text{transpose } (P\ast^\ast A)) \ast v y \)

using w-By.

also have \( ... = ((\text{transpose } A) \ast (\text{transpose } P)) \ast v y \)

unfolding matrix-transpose-mul ..

also have \( ... = (\text{transpose } A) \ast v ((\text{transpose } P) \ast v y) \)

unfolding matrix-vector-mul-assoc ..

finally show \( w \in \text{row-space } A \)

unfolding row-space-eq by blast

next

fix \( w \)

assume \( w: w \in \text{row-space } A \)

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 \( ... = (\text{transpose } ((\text{matrix-inv } P) \ast (P\ast^\ast A))) \ast v y \)

by (metis P matrix-inv-left matrix-mul-assoc matrix-mul-lid)

also have \( ... = (\text{transpose } (P\ast^\ast A) \ast (\text{transpose } (\text{matrix-inv } P))) \ast v y \)

unfolding matrix-transpose-mul ..

also have \( ... = \text{transpose } (P\ast^\ast A) \ast v ((\text{transpose } (\text{matrix-inv } P)) \ast v y) \)

unfolding matrix-vector-mul-assoc ..

finally show \( w \in \text{row-space } (P\ast^\ast A) \)

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 \( A\text{-in-}B: A \subseteq B \)

shows \( \text{dependent } B \)

using independent-mono [OF - A\text{-in-}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 \( \text{fin-}A: \text{finite } A \)
and  \( \text{ind}: \text{independent } A \)
and  \( \text{sum}: (\sum_{x \in A} \text{scale}(f x) x) = 0 \)
shows  \( \forall x \in A. f x = 0 \)
using  \text{assms unfolding independent-explicit by auto}

end

context  \text{finite-dimensional-vector-space}
begin
In an finite dimensional vector space, every independent set is finite, and thus

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

holds:
corollary  \text{scalars-zero-if-independent-euclidean}:
assumes  \text{ind: independent } A
and  \text{sum}: (\sum_{x \in A} \text{scale}(f x) x) = 0
shows  \( \forall x \in A. f x = 0 \)
by  \text{(rule scalars-zero-if-independent,}
    \text{rule conjunct1 [OF independent-bound [OF ind]])}
    \text{(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 (\text{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 \( \text{ker} f \), 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  \text{vector-space}
begin
lemma  \text{inj-on-extended}:
assumes  \text{if: linear scaleB scaleC f}
and  \text{f: finite } C
and  \text{ind-C: independent } C

end
and \( C\text{-eq} \): \( C = B \cup W \)
and \( \text{disj-set} \): \( B \cap W = \{\} \)
and \( \text{span-B} \): \( \{x. f x = 0\} \subseteq \text{span B} \)
shows \( \text{inj-on} f W \)
— The proof is carried out by reductio ad absurdum

**proof** (unfold \( \text{inj-on-def} \), \( \text{rule+} \), \( \text{rule contr} \))

**interpret** \( \text{if: linear scaleB scaleC f using if by simp} \)
— Some previous consequences of the premises that are used later:
**have** \( \text{fin-B} \): \( \text{finite B} \) using \( \text{finite-subset [OF f eq C-eq by simp} \)
**have** \( \text{ind-B} \): \( \text{independent B} \) and \( \text{ind-W} \): \( \text{independent W} \) using \( \text{independent-mono [OF ind-C C-eq by simp-all} \)
— The proof starts here; we assume that there exist two different elements
— with the same image:
**fix** \( x::\prime b \) and \( y::\prime b \)
assume \( x\in W \) and \( y\in W \) and \( f\text{-eq} \): \( f x = f y \) and \( x\not= y \)
**have** \( \text{fin-yB} \): \( \text{finite (insert y B)} \) using \( \text{fin-B by simp} \)
**have** \( f (x - y) = 0 \) by \( \text{(metis diff-self f eq if.linear-sub} \)
**hence** \( x - y \in \{x. f x = 0\} \) by \( \text{simp} \)
**hence** \( \exists g. (\sum v \in B. \text{scale (g v) v}) = (x - y) \) using \( \text{span-B} \)
unfolding \( \text{span-finite [OF fin-B by auto} \)
then obtain \( g \) where \( \text{sum: (\sum v \in B. \text{scale (g v) v}) = (x - y)} \) by \( \text{blast} \)
— We define one of the elements as a linear combination of the second element
and the ones in \( B \)
**def** \( h \equiv (\lambda a. \text{if } a = y \text{ then } (1::\prime a) \text{ else } g a) \)
**have** \( x = y + (\sum v \in B. \text{scale (g v) v}) \) using \( \text{sum by auto} \)
**also have** \( \ldots \) = \( \text{scale (h y) y + (\sum v \in B. \text{scale (g v) v})} \) unfolding \( \text{h-def by simp} \)
**also have** \( \ldots \) = \( \text{scale (h y) y + (\sum v \in B. \text{scale (g v) v})} \) unfolding \( \text{h-def by simp} \)
apply (unfold \( \text{add-left-cancel, rule setsum.cong} \)
using \( y \) h-def \( \text{empty-iff disj-set by auto} \)
**also have** \( \ldots \) = \( (\sum v \in (\text{insert y B) . scale (g v) v}) \)
by (rule setsum.insert[\text{symmetric}], rule \( \text{fin-B} \)
(\text{metis (lifting) IntI disj-set empty-iff y} \)
finally have \( x\text{-in-span-yB: } x \in \text{span (insert y B)} \)
unfolding \( \text{span-finite [OF fin-yB] by auto} \)
— We have that a subset of elements of \( C \) is linearly dependent
**have** \( \text{dep: dependent (insert x (insert y B))} \)
by (unfold dependent-def, \( \text{rule bexI [of - x]} \)
(\text{metis Diff-insert-absorb Int-iff disj-set empty-iff insert-iff} 
\text{x x\text{-in-span-yB x\not= y, simp} \)
— Therefore, the set \( C \) is also dependent:
**hence** \( \text{dependent C using C-eq x y} \)
by (metis \( \text{Un-commute Un-upper2 dependent-mono insert-absorb insert-subset} \)
— This yields the contradiction, since \( C \) is independent:
**thus** \( \text{False using ind-C by contradiction} \)
qed
end
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** \( B.\dim \{ x. \ f x = 0 \} + C.\dim (\text{range } f) \)

**proof**

- **have** \( l \) : linear scale \( B \) scale \( C \) \( f \) by unfold-locales
  - For convenience we define abbreviations for the universe set, \( V \), and the kernel of \( f \)
    
    **def** \( V == \text{UNIV} :: 'b \text{ set} \)
    **def** \( \ker f == \{ x. \ f x = 0 \} \)
  - The kernel is a proper subspace:
    **have** \( \text{sub-ker} \):
      - **B.\subs space \{ x. \ f x = 0 \} \text{ using } \text{B.subspace-kernel[OF } l] \text{.} \)
    - The kernel has its proper basis, \( B \):
      **obtain** \( B \) where
        - \( \text{B-in-ker} \):
          - \( B \subseteq \{ x. \ f x = 0 \} \)
        - \( \text{independent-B} \):
          - \( B.\independent B \)
      and
        - \( \text{card-B} \):
          - \( B.\dim \{ x. \ f x = 0 \} \text{ using } \text{B.basis-exists by blast} \)
  - The space \( V \) has a (finite dimensional) basis, \( C \):
    **obtain** \( C \) where
      - \( \text{B-in-C} \):
        - \( B \subseteq C \)
      and
        - \( \text{C-in-V} \):
          - \( C \subseteq V \)
      and
        - \( \text{independent-C} \):
          - \( B.\independent C \)
      and
        - \( \text{span-C} \):
          - \( V = B.\text{span } C \)
    using \( \text{B.maximal-independent-subset-extend [OF - independent-B, of } V] \)
- **unfolding** \( \text{V-def by auto} \)
  - 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** \( \text{C-eq} \): \( C = B \cup (C - B) \text{ by (rule Diff-partition [OF B-in-C, symmetric])} \)
    **have** \( \text{eq-fC} \): \( f \cdot C = f \cdot B \cup f \cdot (C - B) \)
    - **by** \( \text{subst C-eq, unfold image-Un, simp} \)
      - The basis \( C \), and its image, are finite, since \( V \) is finite-dimensional
    **have** \( \text{finite-C} : \text{finite } C \)
      - **using** \( \text{B.independent-bound-general [OF independent-C] by fast} \)
    **have** \( \text{finite-fC} : \text{finite } (f \cdot C) \text{ by (rule finite-imageI [OF finite-C])} \)
      - The basis \( B \) of the kernel of \( f \), and its image, are also finite
    **have** \( \text{finite-B} : \text{finite } B \) by (rule rev-finite-subset [OF finite-C B-in-C])
    **have** \( \text{finite-fB} : \text{finite } (f \cdot B) \text{ by (rule finite-imageI[OF finite-B])} \)
      - The set \( C - B \) is also finite
    **have** \( \text{finite-CB} : \text{finite } (C - B) \text{ by (rule finite-Diff [OF finite-C, of } B]) \)
    **have** \( \text{dim-ker-le-dim-V: }B.\dim (\ker f) \leq B.\dim V \)

87
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

also have ... = card ($B$ $C$) using C-eq by simp
also have ... = card $B$ + card ($C - B$) by (rule card-Un-disjoint[OF finite-$B$ finite-CB], fast)
also have ... = $B$.dim ker-f + card ($C - B$) unfolding ker-f-def card-$B$ ..
— Now it has to be proved that the elements of $C - B$ are a basis of the range of $f$
also have ... = $B$.dim ker-f + C.dim (range f)
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 ' W) using finite-imageI[OF finite-W] .
have card W = card (f ' W)
by (rule card-image [symmetric], rule $B$.inj-on-extended[OF l, of C B], rule finite-C)
(rule independent-C, unfold W-def, subst C-eq, rule refl, simp, rule ker-in-span)
also have ... = C.dim (range f)
unfolding C.dim-def
proof (rule someI2)
— 1. The image set of $W$ generates the range of $f$:
have range-in-span-fW: range f $\subseteq$ C.span (f ' W)
proof (unfold l.C.span-finite [OF finite-fW], auto)
— 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 $v :: l$
have fV-span: f ' V $\subseteq$ C.span (f ' C)
using $B$.spans-image [OF l] span-C by simp
have $\exists$ g. ($\sum x \in f' C$. scaleC (g x) x) = f v
using fV-span unfolding V-def
using l.C.span-finite[OF finite-fC] by auto
then obtain g where fg: f v = ($\sum x \in f' C$. 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: ($\sum x \in f' B$. scaleC (g x) x) = 0
using $B$.in-ker by (auto intro!: setsum.neutral)
have zero-inter: ($\sum x \in (f ' B \cap f ' W)$. scaleC (g x) x) = 0
using $B$.in-ker by (auto intro!: setsum.neutral)
have \( f \ v = (\sum x \in f \ W. \ \text{scale} \ C \ (g \ x) \ x) \) using \( f \ v \).
also have \( ... = (\sum x \in (f \ B \cup f \ W). \ \text{scale} \ C \ (g \ x) \ x) \)
using \( \text{eq-f}C \ \text{W-def by simp} \)
also have \( ... = (\sum x \in f \ B. \ \text{scale} \ C \ (g \ x) \ x) + (\sum x \in f \ W. \ \text{scale} \ C \ (g \ x) \ x) \)
also have \( ... = (\sum x \in (f \ B \cap f \ W). \ \text{scale} \ C \ (g \ x) \ x) \)
using \( \text{setsum-Un} \ [\text{OF finite-fB} \ \text{finite-fW}] \) by \( \text{simp} \)
also have \( ... = (\sum x \in f \ W. \ \text{scale} \ C \ (g \ x) \ x) \)

— We have proved that the image set of \( W \) is a generating set of \( f \) range of \( f \)

finally show \( \exists \ s. \ (\sum x \in f \ W. \ \text{scale} \ C \ (s \ x) \ x) = f \ v \) by \( \text{auto} \)
qed

— 2. The image set of \( W \) is linearly independent:

have \( \text{independent-fW}: \ \text{l.c.independent} \ (f \ W) \)

proof (rule \( \text{l.c.independent-if-scalars-zero} \ [\text{OF finite-fW}], \ \text{rule+} \))
— Every linear combination (given by \( gx \)) of the elements of the image set of \( W \) equal to zero, requires every coefficient to be zero:

fix \( g :: \ 'c = \rightarrow \ 'a \ \text{and} \ w :: \ 'c \)
assume \( \text{sum}: \ (\sum x \in f \ W. \ \text{scale} \ C \ (g \ x) \ x) = 0 \) and \( w: \ w \in f \ W \)

have \( 0 = (\sum x \in f \ W. \ \text{scale} \ C \ (g \ x) \ x) \) using \( \text{sum by simp} \)
also have \( ... = (\sum x \in f \ W. \ \text{scale} \ C \ ((g \ o \ f) \ x) \ x) \) unfolding \( o-def .. \)
also have \( ... = f (\sum x \in f \ W. \ \text{scale} \ B \ ((g \ o \ f) \ x) \ x) \)

finally have \( f-\text{sum-zero-f} \) (\( \sum x \in f \ W. \ \text{scale} \ B \ ((g \ o \ f) \ x) \ x) = 0 \) by \( \text{rule \ symmetric} \)

hence \( (\sum x \in W. \ \text{scale} \ B \ ((g \ o \ f) \ x) \ x) \in \text{ker-f} \) unfolding \( \text{ker-f-def by simp} \)

hence \( \exists h. \ (\sum v \in B. \ \text{scale} \ B \ (h \ v) \ v) = (\sum x \in W. \ \text{scale} \ B \ ((g \ o \ f) \ x) \ x) \)
using \( \text{B.span-finite} [\text{OF finite-B}] \) using \( \text{ker-in-span} \)

unfolding \( \text{ker-f-def by auto} \)
then obtain \( h \) where

\( \text{sum-h}: \ (\sum v \in B. \ \text{scale} \ B \ (h \ v) \ v) = (\sum x \in W. \ \text{scale} \ B \ ((g \ o \ f) \ x) \ x) \) by \( \text{blast} \)

def \( t \equiv (\lambda a. \ \text{if} \ a \in B \ \text{then} \ h \ a \ \text{else} \ -(g \ o \ f) \ a) \)

have \( 0 = (\sum v \in B. \ \text{scale} \ B \ (h \ v) \ v) + (\sum x \in W. \ \text{scale} \ B \ ((g \ o \ f) \ x) \ x) \)
using \( \text{sum-h by simp} \)
also have \( ... = (\sum v \in B. \ \text{scale} \ B \ (t \ v) \ v) + (\sum x \in W. \ -(\text{scaleB} ((g \ o \ f) \ x) \ x)) \)
unfolding \( \text{setsum-negf ..} \)
also have \( ... = (\sum v \in B. \ \text{scaleB} \ (t \ v) \ v) + (\sum x \in W. \ -(\text{scaleB} ((g \ o \ f) \ x) \ x)) \)

unfolding \( \text{add-right-cancel unfolding t-def by simp} \)
also have \( ... = (\sum v \in B. \ \text{scaleB} \ (t \ v) \ v) + (\sum x \in W. \ \text{scaleB} \ (t \ x) \ x) \)
by \( \text{(unfold add-left-cancel t-def W-def, rule setsum.cong)} \) \( \text{simp+} \)
also have \( ... = (\sum v \in B \cup W. \ \text{scaleB} \ (t \ v) \ v) \)

89
by (rule setsum.union-inter-neutral [symmetric], rule finite-B, rule finite-W)

(simp add: W-def)
finally have \((\sum v \in B \cup W. \text{scale}_B (t \cdot v)) = 0\) by simp
hence \(\text{coef-zero}: \forall x \in B \cup W. t \cdot x = 0\)
  using C-eq B.scalars-zero-if-independent [OF finite-C independent-C]
  unfolding W-def by simp
obtain \(y\) where \(w-fy\): \(w = f \cdot y\) and \(y-in-W\): \(y \in W\) using \(w\) by fast
have \(-g \cdot w = t \cdot y\)
  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 \(g \cdot w = 0\) 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 \(\exists B \subseteq \text{range } f. \neg \text{vector-space.dependent scale}_{C} B\)
  \& \(\text{range } f \subseteq \text{vector-space.span scale}_{C} B \land \text{card } B = \text{card } (f \cdot W)\)
by (rule \text{exI} [of \(f \cdot W\)],
  simp add: range-in-span-fW independent-fW image-mono)
show \(\forall n :: \text{nat}. \exists B \subseteq \text{range } f. l.C.independent B \land \text{range } f \subseteq l.C.span B \land \text{card } B = n\)
  \(\Rightarrow \text{card } (f \cdot W) = n\)
proof (clarify)
fix \(S :: \{\text{c set}\}\)
assume S-in-range: \(S \subseteq \text{range } f\) and independent-S: \(\text{vector-space.independent scale}_{C} S\)
and range-in-spanS: \(\text{range } f \subseteq \text{vector-space.span scale}_{C} S\)
have S-le: \(\text{finite } S \land \text{card } S \leq \text{card } (f \cdot W)\)
by (rule l.C.independent-span-bound[OF finite-fW independent-S])
(rule subset-trans [OF S-in-range range-in-span-fW])
show \(\text{card } (f \cdot W) = \text{card } S\)
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 \(\text{card } (C - B) = C.\text{dim } (\text{range } f)\) unfolding W-def .

finally show \(?\text{thesis unfolding } V-def \text{ ker-f-def unfolding dim-UNIV }\) .

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::'a::{field} ´cols::{finite, wellorder} ´rows
shows ncols A = vec.dim (null-space A) + vec.dim (col-space A)
proof –
show ?thesis
  apply (subst (1 2) matrix-of-matrix-vector-mul [of A, symmetric])
  unfolding null-space-eq-ker [OF matrix-vector-mul-linear]
  unfolding col-space-eq-range [OF matrix-vector-mul-linear]
  using vec.rank-nullity-theorem
  by (metis col-space-eq ncols-def vec.dim-UNIV vec.dimension-def vec-dim-card)
qed

end

References
