

Rank-Nullity Theorem in Linear Algebra

By Jose Divasón and Jesús Aransay*

September 13, 2023

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	Dual Order	2
1.1	Interpretation of dual wellorder based on wellorder	2
1.2	Properties of the Greatest operator	3
2	Class for modular arithmetic	3
2.1	Definition and properties	3
2.2	Conversion between a modular class and the subset of natural numbers associated.	4
2.3	Instantiations	13

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

3	Miscellaneous	15
3.1	Definitions of number of rows and columns of a matrix	15
3.2	Basic properties about matrices	16
3.3	Theorems obtained from the AFP	16
3.4	Basic properties involving span, linearity and dimensions	18
3.5	Basic properties about matrix multiplication	19
3.6	Properties about invertibility	20
3.7	Properties about the dimension of vectors	21
3.8	Instantiations and interpretations	22
4	Fundamental Subspaces	23
4.1	The fundamental subspaces of a matrix	23
4.1.1	Definitions	23
4.1.2	Relationships among them	23
4.2	Proving that they are subspaces	24
4.3	More useful properties and equivalences	24
5	Rank Nullity Theorem of Linear Algebra	27
5.1	Previous results	27
5.2	The proof	29
5.3	The rank nullity theorem for matrices	33

1 Dual Order

```
theory Dual-Order
  imports Main
begin
```

1.1 Interpretation of dual wellorder based on wellorder

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 (λ(x::'a) y::'a. y ≤ x) (λ(x::'a) y::'a. y < x)
  using lin unfolding class.wellorder-def apply (rule conjI)
  apply (rule class.wellorder-axioms.intro) by (blast intro: wf-induct-rule [OF wf])
```

lemma (in *preorder*) *trancl-less'*: $(>)^{++} = (>)$

```
  by(auto simp add: fun-eq-iff intro: less-trans elim: tranclp.induct)
```

interpretation *dual-wellorder*: *wellorder* (\geq) : $(\geq) :: ('a :: \{linorder, finite\} \Rightarrow 'a \Rightarrow bool)$
 $(>)$

proof (rule *wf-wellorderI2*)

```
  show wf {(x :: 'a, y). y < x}
```

```
    by(auto simp add: trancl-def tranclp-less' intro!: finite-acyclic-wf acyclicI)
```

```
  show class.linorder (λ(x::'a) y::'a. y ≤ x) (λ(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

```

1.2 Properties of the Greatest operator

```

lemma dual-wellorder-Least-eq-Greatest[simp]: dual-wellorder.Least = Greatest
by (auto simp add: Greatest-def dual-wellorder.Least-def)

```

```

lemmas GreatestI = dual-wellorder.LeastI[unfolded dual-wellorder-Least-eq-Greatest]
lemmas GreatestI2-ex = dual-wellorder.LeastI2-ex[unfolded dual-wellorder-Least-eq-Greatest]
lemmas GreatestI2-wellorder = dual-wellorder.LeastI2-wellorder[unfolded dual-wellorder-Least-eq-Greatest]
lemmas GreatestI-ex = dual-wellorder.LeastI-ex[unfolded dual-wellorder-Least-eq-Greatest]
lemmas not-greater-Greatest = dual-wellorder.not-less-Least[unfolded dual-wellorder-Least-eq-Greatest]
lemmas GreatestI2 = dual-wellorder.LeastI2[unfolded dual-wellorder-Least-eq-Greatest]
lemmas Greatest-ge = dual-wellorder.Least-le[unfolded dual-wellorder-Least-eq-Greatest]

```

```

end

```

2 Class for modular arithmetic

```

theory Mod-Type
imports
  HOL-Library.Numeral-Type
  HOL-Analysis.Cartesian-Euclidean-Space
  Dual-Order
begin

```

2.1 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:  $\text{Rep } x < \text{int CARD } ('a)$ 
  by (rule type-definition.Rep [OF type, simplified, THEN conjunct2])

lemma Rep-le-n:  $\text{Rep } x \leq \text{int CARD } ('a)$ 
  by (rule Rep-less-n [THEN order-less-imp-le])

lemma Rep-inject-sym:  $x = y \longleftrightarrow \text{Rep } x = \text{Rep } y$ 
  by (rule type-definition.Rep-inject [OF type, symmetric])

lemma Rep-inverse:  $\text{Abs } (\text{Rep } x) = x$ 
  by (rule type-definition.Rep-inverse [OF type])

lemma Abs-inverse:  $m \in \{0..<\text{int CARD } ('a)\} \implies \text{Rep } (\text{Abs } m) = m$ 
  by (rule type-definition.Abs-inverse [OF type])

lemma Rep-Abs-mod:  $\text{Rep } (\text{Abs } (m \bmod \text{int CARD } ('a))) = m \bmod \text{int CARD } ('a)$ 
  using size0 by (auto simp add: Abs-inverse)

lemma Rep-Abs-0:  $\text{Rep } (\text{Abs } 0) = 0$ 
  apply (rule Abs-inverse [of 0])
  using size0 by simp

lemma Rep-0:  $\text{Rep } 0 = 0$ 
  by (simp add: zero-def Rep-Abs-0)

lemma Rep-Abs-1:  $\text{Rep } (\text{Abs } 1) = 1$ 
  by (simp add: Abs-inverse size1)

lemma Rep-1:  $\text{Rep } 1 = 1$ 
  by (simp add: one-def Rep-Abs-1)

lemma Rep-mod:  $\text{Rep } x \bmod \text{int CARD } ('a) = \text{Rep } x$ 
  apply (rule-tac x=x in type-definition.Abs-cases [OF type])
  apply (simp add: type-definition.Abs-inverse [OF type])
done

lemmas Rep-simps =
  Rep-inject-sym Rep-inverse Rep-Abs-mod Rep-mod Rep-Abs-0 Rep-Abs-1

```

2.2 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 $to\text{-}nat :: 'a \Rightarrow nat$
where $to\text{-}nat = nat \circ Rep$

definition $Abs' :: int \Rightarrow 'a$
where $Abs' x = Abs(x \bmod int \text{ CARD } ('a))$

definition $from\text{-}nat :: nat \Rightarrow 'a$
where $from\text{-}nat = (Abs' \circ int)$

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

lemma $mono\text{-}Rep$: $mono\ Rep$ **by** ($metis\ strict\text{-}mono\text{-}Rep\ strict\text{-}mono\text{-}mono$)

lemma $Rep\text{-}ge\text{-}0$: $0 \leq Rep\ x$ **using** $bij\text{-}Rep$ **unfolding** $bij\text{-}betw\text{-}def$ **by** $auto$

lemma $bij\text{-}Abs$: $bij\text{-}betw (Abs) \{0..<int \text{ CARD } ('a)\} (UNIV::'a \text{ set})$
proof ($unfold\ bij\text{-}betw\text{-}def, rule\ conjI$)
show $inj\text{-}on\ Abs \{0..<int \text{ CARD } ('a)\}$ **by** ($metis\ inj\text{-}on\text{-}inverseI\ type\ type\text{-}definition.Abs\text{-}inverse$)
show $Abs' \text{ ' } \{0..<int \text{ CARD } ('a)\} = (UNIV::'a \text{ set})$ **by** ($metis\ type\ type\text{-}definition.univ$)
qed

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

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

```

qed
show inj-on (from-nat::nat=>'a) {0::nat..<CARD('a)}
proof (unfold from-nat-def , rule comp-inj-on)
  show inj-on int {0::nat..<CARD('a)} by (metis inj-of-nat subset-inj-on top-greatest)
  show inj-on (Abs'::int=>'a) (int ' {0::nat..<CARD('a)})
    using bij-Abs unfolding bij-betw-def set-eq
    by (metis (opaque-lifting, no-types) Abs'-def Abs-inverse Rep-inverse Rep-mod
inj-on-def set-eq)
qed
show (from-nat::nat=>'a)' {0::nat..<CARD('a)} = UNIV
  unfolding from-nat-def using bij-Abs'
  unfolding bij-betw-def set-eq o-def by blast
qed

```

```

lemma to-nat-is-inv: the-inv-into {0..<CARD('a)} (from-nat::nat=>'a) = (to-nat::'a=>nat)
proof (unfold the-inv-into-def fun-eq-iff from-nat-def to-nat-def o-def, clarify)
  fix x::'a show (THE y::nat. y ∈ {0::nat..<CARD('a)} ∧ 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: ¬ (0::int) ≤ Rep x show (0::nat) < CARD('a) and Abs' (0::int)
= x
    using Rep-ge-0 x by auto
  next
  fix y::nat assume y: y < CARD('a)
  have (Rep(Abs'(int y)::'a)) = (Rep((Abs(int y mod int CARD('a))::'a)) un-
folding Abs'-def ..
  also have ... = (Rep (Abs (int y)::'a)) using 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
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 $(0 :: 'a) \leq \text{Abs} (\text{Rep } y \text{ mod int } \text{CARD}('a))$ **using** *strict-mono-Rep unfolding strict-mono-def*
by (*metis (opaque-lifting, mono-tags) Rep-0 Rep-ge-0 strict-mono-Rep strict-mono-less-eq*)
also have $\dots = y$ **by** (*metis Rep-inverse Rep-mod*)
finally show $(0 :: 'a) \leq y$.
qed

lemma *add-to-nat-def*: $x + y = \text{from-nat} (\text{to-nat } x + \text{to-nat } y)$
unfolding *from-nat-def to-nat-def o-def* **using** *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*: $\text{to-nat } 1 = 1$
by (*simp add: to-nat-def Rep-1*)

lemma *add-def'*:
shows $x + y = \text{Abs}' (\text{Rep } x + \text{Rep } y)$ **unfolding** *Abs'-def* **using** *add-def* **by** *simp*

lemma *Abs'-0*:
shows $\text{Abs}' (\text{CARD}('a)) = (0 :: 'a)$ **by** (*metis (opaque-lifting, mono-tags) Abs'-def mod-self zero-def*)

lemma *Rep-plus-one-le-card*:
assumes $a: a + 1 \neq 0$
shows $(\text{Rep } a) + 1 < \text{CARD} ('a)$
proof (*rule ccontr*)
assume $\neg \text{Rep } a + 1 < \text{CARD}('a)$ **hence** *to-nat-eq-card*: $\text{Rep } a + 1 = \text{CARD}('a)$
using *Rep-less-n*
by (*simp add: add1-zle-eq order-class.less-le*)
have $a+1 = \text{Abs}' (\text{Rep } a + \text{Rep } (1 :: 'a))$ **using** *add-def'* **by** *auto*
also have $\dots = \text{Abs}' ((\text{Rep } a) + 1)$ **using** *Rep-1* **by** *simp*
also have $\dots = \text{Abs}' (\text{CARD}('a))$ **unfolding** *to-nat-eq-card* ..
also have $\dots = 0$ **using** *Abs'-0* **by** *auto*
finally show *False* **using** a **by** *contradiction*
qed

lemma *to-nat-plus-one-less-card*: $\forall a. a+1 \neq 0 \longrightarrow \text{to-nat } a + 1 < \text{CARD}('a)$
proof (*clarify*)
fix a
assume $a: a + 1 \neq 0$
have $\text{Rep } a + 1 < \text{int } \text{CARD}('a)$ **using** *Rep-plus-one-le-card[OF a]* **by** *auto*
hence $\text{nat} (\text{Rep } a + 1) < \text{nat} (\text{int } \text{CARD}('a))$ **unfolding** *zless-nat-conj* **using** *size0* **by** *fast*
thus $\text{to-nat } a + 1 < \text{CARD}('a)$ **unfolding** *to-nat-def o-def* **using** *nat-add-distrib[OF Rep-ge-0]* **by** *simp*
qed

corollary *to-nat-plus-one-less-card'*:
assumes $a+1 \neq 0$
shows $\text{to-nat } a + 1 < \text{CARD}('a)$ **using** *to-nat-plus-one-less-card assms* **by** *simp*

lemma *strict-mono-to-nat*: *strict-mono 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*]: $\text{to-nat } x = \text{to-nat } y \longleftrightarrow x = y$
using *injD* [*OF bij-betw-imp-inj-on* [*OF bij-to-nat*]] **by** *blast*

lemma *mod-type-forall-eq* [*simp*]: $(\forall j::'a. (\text{to-nat } j) < \text{CARD}('a) \longrightarrow P j) = (\forall a. P a)$
proof (*auto*)
fix a **assume** $a: \forall j. (\text{to-nat}::'a \Rightarrow \text{nat}) j < \text{CARD}('a) \longrightarrow P j$
have $(\text{to-nat}::'a \Rightarrow \text{nat}) a < \text{CARD}('a)$ **using** *bij-to-nat* **unfolding** *bij-betw-def*
by *auto*
thus $P a$ **using** a **by** *auto*
qed

lemma *to-nat-from-nat*:
assumes $t: \text{to-nat } j = k$
shows $\text{from-nat } k = j$
proof –
have $\text{from-nat } k = \text{from-nat } (\text{to-nat } j)$ **unfolding** t ..
also have $\dots = \text{from-nat } (\text{the-inv-into } \{0.. < \text{CARD}('a)\} (\text{from-nat } j))$ **unfolding** *to-nat-is-inv* ..
also have $\dots = j$
proof (*rule f-the-inv-into-f*)
show $\text{inj-on from-nat } \{0.. < \text{CARD}('a)\}$ **by** (*metis bij-betw-imp-inj-on bij-from-nat*)
show $j \in \text{from-nat } \{0.. < \text{CARD}('a)\}$ **by** (*metis UNIV-I bij-betw-def bij-from-nat*)
qed
finally show $\text{from-nat } k = j$.
qed

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

lemma *to-nat-mono'*:
assumes $ab: a \leq b$
shows $\text{to-nat } a \leq \text{to-nat } b$
proof (*cases a=b*)
case *True* **thus** *?thesis* **by** *auto*
next
case *False*

hence $a < b$ using ab by $simp$
 thus $?thesis$ using $to-nat-mono$ by $fastforce$
 qed

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

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

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

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

lemma $from-nat-mono$:
 assumes $i-le-j: i < j$ and $j: j < CARD('a)$
 shows $(from-nat i::'a) < from-nat j$
 proof –
 have $i: i < CARD('a)$ using $i-le-j j$ 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 $i-le-j: i \leq j$ and $j < CARD ('a)$
 shows $(from-nat i::'a) \leq from-nat j$
 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 $to\text{-nat } (x)+1 < CARD ('a)$
shows $to\text{-nat } (x + 1::'a) = (to\text{-nat } x) + 1$
proof –
have $(x::'a) + 1 = from\text{-nat } (to\text{-nat } x + to\text{-nat } (1::'a))$ **unfolding** *add-to-nat-def*
..
hence $to\text{-nat } ((x::'a) + 1) = to\text{-nat } (from\text{-nat } (to\text{-nat } x + to\text{-nat } (1::'a))::'a)$
by *presburger*
also have $\dots = to\text{-nat } (from\text{-nat } (to\text{-nat } x + 1)::'a)$ **unfolding** *to-nat-1* **..**
also have $\dots = (to\text{-nat } x + 1)$ **by** (*metis* *assms to-nat-from-nat-id*)
finally show *?thesis* .
qed

lemma *to-nat-le*:
assumes $y < from\text{-nat } k$
shows $to\text{-nat } y < k$
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** $to\text{-nat } y < CARD ('a)$ **using** *bij-to-nat* **unfolding** *bij-betw-def*
by *auto*
thus *?thesis* **using** *False* **by** *auto*
qed

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

lemma *le-Suc'*:
assumes $ab: a + 1 \leq b$
and *less-card*: $(to\text{-nat } a) + 1 < CARD ('a)$
shows $a < b$
proof –

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

lemma 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** $(\text{metis less-card to-nat-suc})$
 hence $(\text{from-nat } (\text{to-nat } a)::'a) < \text{from-nat } (\text{to-nat } (a + 1))$
 by $(\text{rule from-nat-mono, metis less-card to-nat-suc})$
 thus $a < a + 1$ **by** $(\text{metis to-nat-from-nat})$
qed

lemma $\text{Suc-le}'$:
 fixes $a::'a$
 assumes $a + 1 \neq 0$
 shows $a < a + 1$ **using** $\text{Suc-le to-nat-plus-one-less-card assms}$ **by** blast

lemma from-nat-not-eq :
 assumes $a\text{-eq-to-nat: } a \neq \text{to-nat } b$
 and $a\text{-less-card: } a < \text{CARD}('a)$
 shows $\text{from-nat } a \neq b$
proof (rule ccontr)
 assume $\neg \text{from-nat } a \neq b$ **hence** $\text{from-nat } a = b$ **by** simp
 hence $\text{to-nat } ((\text{from-nat } a)::'a) = \text{to-nat } b$ **by** auto
 thus False **by** $(\text{metis a-eq-to-nat a-less-card to-nat-from-nat-id})$
qed

lemma Suc-less :
 fixes $i::'a$
 assumes $i < j$
 and $i + 1 \neq j$
 shows $i + 1 < j$ **by** $(\text{metis assms le-Suc le-neq-trans})$

lemma $\text{Greatest-is-minus-1: } \forall a::'a. a \leq -1$
proof (clarify)
 fix $a::'a$
 have $\text{zero-ge-card-1: } 0 \leq \text{int } \text{CARD}('a) - 1$ **using** size1 **by** auto
 have $\text{card-less: } \text{int } \text{CARD}('a) - 1 < \text{int } \text{CARD}('a)$ **by** auto

have *not-zero*: $1 \bmod \text{int } \text{CARD}('a) \neq 0$
by (*metis* (*opaque-lifting*, *mono-tags*) *Rep-Abs-1 Rep-mod zero-neq-one*)
have *int-card*: $\text{int } (\text{CARD}('a) - 1) = \text{int } \text{CARD}('a) - 1$ **using** *of-nat-diff*[*of 1 CARD ('a)*]
using *size1* **by** *simp*
have $a = \text{Abs}'(\text{Rep } a)$ **by** (*metis* (*opaque-lifting*, *mono-tags*) *Rep-0 add-0-right add-def'*
monoid-add-class.add.right-neutral)
also have $\dots = \text{Abs}'(\text{int } (\text{nat } (\text{Rep } a)))$ **by** (*metis* *Rep-ge-0 int-nat-eq*)
also have $\dots \leq \text{Abs}'(\text{int } (\text{CARD}('a) - 1))$
proof (*rule from-nat-mono'*[*unfolded from-nat-def o-def, of nat (Rep a) CARD('a) - 1*])
show $\text{nat } (\text{Rep } a) \leq \text{CARD}('a) - 1$ **using** *Rep-less-n*
using *int-card nat-le-iff* **by** *auto*
show $\text{CARD}('a) - 1 < \text{CARD}('a)$ **using** *finite-UNIV-card-ge-0 finite-mod-type*
by *fastforce*
qed
also have $\dots = -1$
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*: $\forall a::'a. a+1 = 0 \longrightarrow a = -1$
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 $\text{from-nat } y \in (\text{UNIV}::'a \text{ set})$ **by** *auto*
from this obtain x **where** $x1: \text{from-nat } y = (\text{from-nat } x::'a)$ **and** $x2: x \in \{0..<\text{CARD}('a)\}$
using *bij-from-nat* **unfolding** *bij-betw-def*
by (*metis from-nat-to-nat-id rangeI 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 $to\text{-}nat\ j < CARD\ ('a)$
using *bij-to-nat* **unfolding** *bij-betw-def* **by** *auto*

lemma *from-nat-0*: $from\text{-}nat\ 0 = 0$
unfolding *from-nat-def o-def of-nat-0 Abs'-def mod-0 zero-def ..*
lemma *to-nat-0*: $to\text{-}nat\ 0 = 0$ **unfolding** *to-nat-def o-def Rep-0 nat-0 ..*
lemma *to-nat-eq-0*: $(to\text{-}nat\ x = 0) = (x = 0)$
by (*auto simp add: to-nat-0 from-nat-0 dest: to-nat-from-nat*)

lemma *suc-not-zero*:
assumes $to\text{-}nat\ a + 1 \neq CARD('a)$
shows $a+1 \neq 0$
proof (*rule ccontr, simp*)
assume *a-plus-one-zero*: $a + 1 = 0$
hence *rep-eq-card*: $Rep\ a + 1 = CARD('a)$
using *assms to-nat-0 Suc-eq-plus1 Suc-lessI Zero-not-Suc to-nat-less-card to-nat-suc*
by (*metis (opaque-lifting, mono-tags)*)
moreover have $Rep\ a + 1 < CARD('a)$
using *Abs'-0 Rep-1 Suc-eq-plus1 Suc-lessI Suc-neq-Zero add-def' assms*
rep-eq-card to-nat-0 to-nat-less-card to-nat-suc **by** (*metis (opaque-lifting, mono-tags)*)
ultimately show *False* **by** *fastforce*
qed

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

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

end

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

2.3 Instantiations

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

definition (*Rep::'a bit0 => int*) $x = Rep\text{-}bit0\ x$
definition (*Abs::int => 'a bit0*) $x = Abs\text{-}bit0'\ x$

definition (*Rep::'a bit1 => int*) $x = Rep\text{-}bit1\ x$
definition (*Abs::int => 'a bit1*) $x = Abs\text{-}bit1'\ x$

```

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

end

```

3 Miscellaneous

```

theory Miscellaneous
  imports
    HOL-Analysis.Determinants
    Mod-Type
    HOL-Library.Function-Algebras
begin

context Vector-Spaces.linear begin
sublocale vector-space-pair by unfold-locales— TODO: (re)move?
end

hide-const (open) Real-Vector-Spaces.linear
abbreviation linear  $\equiv$  Vector-Spaces.linear

```

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

3.1 Definitions of number of rows and columns of a matrix

```

definition nrows ::  $'a \hat{\ } \text{columns} \hat{\ } \text{rows} \Rightarrow \text{nat}$ 
  where nrows  $A = \text{CARD}('rows)$ 

```

```

definition ncols ::  $'a \hat{\ } \text{columns} \hat{\ } \text{rows} \Rightarrow \text{nat}$ 
  where ncols  $A = \text{CARD}('columns)$ 

```

```

definition matrix-scalar-mult ::  $'a::\text{ab-semigroup-mult} \Rightarrow 'a \hat{\ } n \hat{\ } m \Rightarrow 'a \hat{\ } n \hat{\ } m$ 
  (infixl  $*k$  70)

```

where $k * k A \equiv (\chi \ i \ j. k * A \ \$ \ i \ \$ \ j)$

3.2 Basic properties about matrices

lemma *nrows-not-0*[simp]:
shows $0 \neq \text{nrows } A$ **unfolding** *nrows-def* **by** *simp*

lemma *ncols-not-0*[simp]:
shows $0 \neq \text{ncols } A$ **unfolding** *ncols-def* **by** *simp*

lemma *nrows-transpose*: $\text{nrows } (\text{transpose } A) = \text{ncols } A$
unfolding *nrows-def* *ncols-def* ..

lemma *ncols-transpose*: $\text{ncols } (\text{transpose } A) = \text{nrows } A$
unfolding *nrows-def* *ncols-def* ..

lemma *finite-rows*: *finite* (*rows* A)
using *finite-Atleast-Atmost-nat*[of $\lambda i. \text{row } i \ A$] **unfolding** *rows-def* .

lemma *finite-columns*: *finite* (*columns* A)
using *finite-Atleast-Atmost-nat*[of $\lambda i. \text{column } i \ A$] **unfolding** *columns-def* .

lemma *transpose-vector*: $x \ v * A = \text{transpose } A \ * v \ x$
by *simp*

lemma *transpose-zero*[simp]: $(\text{transpose } A = 0) = (A = 0)$
unfolding *transpose-def* *zero-vec-def* *vec-eq-iff* **by** *auto*

3.3 Theorems obtained from the AFP

The following theorems and definitions have been obtained from the AFP http://isa-afp.org/browser_info/current/HOL/Tarskis_Geometry/Linear_Algebra2.html. I have removed some restrictions over the type classes.

lemma *vector-scalar-matrix-ac*:
fixes $k :: 'a::\{\text{field}\}$ **and** $x :: 'a::\{\text{field}\}^n$ **and** $A :: 'a^m^n$
shows $x \ v * (k * k \ A) = k * s \ (x \ v * \ A)$
using *scalar-vector-matrix-assoc*
unfolding *vector-matrix-mult-def* *matrix-scalar-mult-def* *vec-eq-iff*
by (*auto simp add: sum-distrib-left vector-space-over-itself.scale-scale*)

lemma *transpose-scalar*: $\text{transpose } (k * k \ A) = k * k \ \text{transpose } A$
unfolding *transpose-def*
by (*vector, simp add: matrix-scalar-mult-def*)

lemma *scalar-matrix-vector-assoc*:
fixes $A :: 'a::\{\text{field}\}^m^n$
shows $k * s \ (A * v \ v) = k * k \ A * v \ v$
by (*metis transpose-scalar vector-scalar-matrix-ac vector-transpose-matrix*)

lemma *matrix-scalar-vector-ac*:
fixes $A :: 'a::\{\text{field}\}^{\wedge m \wedge n}$
shows $A * v (k * s v) = k * k A * v v$
by (*simp add: Miscellaneous.scalar-matrix-vector-assoc vec.scale*)

definition
is-basis :: $('a::\{\text{field}\}^{\wedge n}) \text{ set} \Rightarrow \text{bool}$ **where**
is-basis $S \equiv \text{vec.independent } S \wedge \text{vec.span } S = \text{UNIV}$

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

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

qed

lemma *basis-finite*:

fixes $B :: ('a::\{field\})^n$ set

assumes *is-basis* B

shows *finite* B

proof –

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

by *simp*

with *card-finite* [of B , **where** $n = n$] **show** *finite* B **by** *simp*

qed

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

3.4 Basic properties involving span, linearity and dimensions

context *finite-dimensional-vector-space*

begin

This theorem is the reciprocal theorem of *local.independent* $?B \implies \text{finite } ?B \wedge \text{card } ?B = \text{local.dim } (\text{local.span } ?B)$

lemma *card-eq-dim-span-indep*:

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

shows *independent* A

by (*metis* *assms* *card-le-dim-spanning* *dim-subset* *equalityE* *span-superset*)

lemma *dim-zero-eq*:

assumes *dim-A*: $\text{dim } A = 0$

shows $A = \{\}$ \vee $A = \{0\}$

using *dim-A* *local.card-ge-dim-independent* *local.independent-empty* **by** *force*

lemma *dim-zero-eq'*:

assumes $A: A = \{\}$ \vee $A = \{0\}$

shows $\text{dim } A = 0$

using *assms* *local.dim-span* *local.indep-card-eq-dim-span* *local.independent-empty* **by** *fastforce*

lemma *dim-zero-subspace-eq*:

assumes *subs-A*: *subspace* A

shows $(\text{dim } A = 0) = (A = \{0\})$

by (*metis* *dim-zero-eq* *dim-zero-eq'* *subspace-0[OF subs-A]* *empty-iff*)

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

assumes *span* $A = \{0\}$

shows $A = \{\}$ \vee $A = \{0\}$ **by** (*metis* *assms* *span-superset* *subset-singletonD*)

end

context *Vector-Spaces.linear*
begin

lemma *linear-injective-ker-0*:
shows $\text{inj } f = (\{x. f \ x = 0\} = \{0\})$
using *inj-iff-eq-0* **by** *auto*

end

lemma *snd-if-conv*:
shows $\text{snd } (if \ P \ \text{then } (A,B) \ \text{else } (C,D)) = (if \ P \ \text{then } B \ \text{else } D)$ **by** *simp*

3.5 Basic properties about matrix multiplication

lemma *row-matrix-matrix-mult*:
fixes $A::'a::\{comm-ring-1\}^{\wedge n \wedge m}$
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!: sum.cong)*

corollary *row-matrix-matrix-mult'*:
fixes $A::'a::\{comm-ring-1\}^{\wedge n \wedge m}$
shows $(\text{row } i \ P) \ v * A = \text{row } i \ (P ** A)$
using *row-matrix-matrix-mult* **unfolding** *row-def* *vec-nth-inverse* .

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

lemma *matrix-matrix-mult-inner-mult*:
shows $(A ** B) \ \$ \ i \ \$ \ j = \text{row } i \ A \ \cdot \ \text{column } j \ B$
unfolding *inner-vec-def* *matrix-matrix-mult-def* *row-def* *column-def* **by** *auto*

lemma *matrix-vmult-column-sum*:
fixes $A::'a::\{field\}^{\wedge n \wedge m}$
shows $\exists f. A * v \ x = \text{sum } (\lambda y. f \ y * s \ y) \ (\text{columns } A)$
proof *(rule exI[of - $\lambda y. \text{sum } (\lambda i. x \ \$ \ i) \ \{i. y = \text{column } i \ A\}$])*
let $?f = \lambda y. \text{sum } (\lambda i. x \ \$ \ i) \ \{i. y = \text{column } i \ A\}$
let $?g = (\lambda y. \ \{i. y = \text{column } i \ (A)\})$
have *inj*: $\text{inj-on } ?g \ (\text{columns } (A))$ **unfolding** *inj-on-def* **unfolding** *columns-def*
by *auto*
have *union-univ*: $\bigcup \ (?g \ (\text{columns } (A))) = \text{UNIV}$ **unfolding** *columns-def* **by** *auto*
have $A * v \ x = (\sum \ i \in \text{UNIV}. x \ \$ \ i * s \ \text{column } i \ A)$ **unfolding** *matrix-mult-sum*
..
also have $\dots = \text{sum } (\lambda i. x \ \$ \ i * s \ \text{column } i \ A) \ (\bigcup \ (?g \ (\text{columns } A)))$ **unfolding**

union-univ ..
also have ... = $\text{sum } (\text{sum } ((\lambda i. x \$ i *s \text{column } i A)))$ ($?g$ '(*columns* *A*))
by (*rule sum.Union-disjoint[unfolded o-def]*, *auto*)
also have ... = $\text{sum } ((\text{sum } ((\lambda i. x \$ i *s \text{column } i A))) \circ ?g)$ (*columns* *A*)
by (*rule sum.reindex, simp add: inj*)
also have ... = $\text{sum } (\lambda y. ?f y *s y)$ (*columns* *A*)
proof (*rule sum.cong, unfold o-def*)
fix *xa*
have $\text{sum } (\lambda i. x \$ i *s \text{column } i A) \{i. xa = \text{column } i A\}$
= $\text{sum } (\lambda i. x \$ i *s xa) \{i. xa = \text{column } i A\}$ **by** *simp*
also have ... = $\text{sum } (\lambda i. x \$ i) \{i. xa = \text{column } i A\} *s xa$
using *vec.scale-sum-left[of (\lambda i. x \\$ i) \{i. xa = \text{column } i A\} xa]* ..
finally show $(\sum i \mid xa = \text{column } i A. x \$ i *s \text{column } i A) = (\sum i \mid xa =$
*column } i A. x \\$ i) *s xa .
qed *rule*
finally show $A *v x = (\sum y \in \text{columns } A. (\sum i \mid y = \text{column } i A. x \$ i) *s y)$.
qed*

3.6 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 $\langle \text{invertible } M \rangle$ **and** *someI-ex [of $\lambda N. M ** N = \text{mat } 1 \wedge N ** M = \text{mat } 1$]*
unfolding *invertible-def and matrix-inv-def*
by *simp-all*

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

lemma *matrix-inv-unique*:
fixes $A::'a::\{\text{semiring-1}\}^{\wedge n} \wedge n$
assumes *AB: A ** B = mat 1 and BA: B ** A = mat 1*
shows *matrix-inv A = B*
by (*metis AB BA invertible-def matrix-inv-right matrix-mul-assoc matrix-mul-lid*)

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

```

next
  assume  $A * v x = 0$ 
  thus  $P ** A * v x = 0$  by (metis matrix-vector-mul-assoc matrix-vector-mult-0-right)
qed

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

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

```

3.7 Properties about the dimension of vectors

```

lemma dimension-vector[code-unfold]:  $vec.dimension\ TYPE('a::\{field\})\ TYPE('rows::\{mod-type\}) = CARD('r$ 
proof -
let  $?f = \lambda x. axis\ (from-nat\ x)\ 1::'a^rows::\{mod-type\}$ 
have  $vec.dimension\ TYPE('a::\{field\})\ TYPE('rows::\{mod-type\}) = card\ (cart-basis::('a^rows::\{mod-type\})$ 
set)
  unfolding  $vec.dimension-def$  ..
also have  $\dots = card\ \{.. < CARD('rows)\}$  unfolding  $cart-basis-def$ 
  proof (rule bij-betw-same-card[symmetric, of ?f], unfold  $bij-betw-def$ , unfold
 $inj-on-def\ axis-eq-axis$ , auto)

```

```

fix x y assume x: x < CARD('rows) and y: y < CARD('rows) and eq:
from-nat x = (from-nat y::'rows)
show x = y using from-nat-eq-imp-eq[OF eq x y] .
next
fix i show axis i 1 ∈ (λx. axis (from-nat x::'rows) 1) ‘ {..<CARD('rows)}
unfolding image-def
by (auto, metis lessThan-iff to-nat-from-nat to-nat-less-card)
qed
also have ... = CARD('rows) by (metis card-lessThan)
finally show ?thesis .
qed

```

3.8 Instantiations and interpretations

Functions between two real vector spaces form a real vector

```

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

```

```

definition scaleR-fun a f = (λi. a *R f i )

```

```

instance

```

```

by (intro-classes, auto simp add: fun-eq-iff scaleR-fun-def scaleR-left.add scaleR-right.add)
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

```

```

interpretation matrix: vector-space ((*k))::'a::{field}=>'a ^ cols ^ rows=>'a ^ cols ^ rows

```

```

proof (unfold-locales)

```

```

fix a::'a and x y::'a ^ cols ^ rows

```

```

show a *k (x + y) = a *k x + a *k y

```

```

unfolding matrix-scalar-mult-def vec-eq-iff

```

```

by (simp add: vector-space-over-itself.scale-right-distrib)

```

```

next

```

```

fix a b::'a and x::'a ^ cols ^ rows

```

```

show (a + b) *k x = a *k x + b *k x

```

```

unfolding matrix-scalar-mult-def vec-eq-iff

```

```

by (simp add: comm-semiring-class.distrib)

```

```

show a *k (b *k x) = a * b *k x

```

```

  unfolding matrix-scalar-mult-def vec-eq-iff by auto
show1 *k x = x unfolding matrix-scalar-mult-def vec-eq-iff by auto
qed

end

```

4 Fundamental Subspaces

```

theory Fundamental-Subspaces
imports
  Miscellaneous
begin

```

4.1 The fundamental subspaces of a matrix

4.1.1 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)

```

4.1.2 Relationships among them

```

lemma left-null-space-eq-null-space-transpose: left-null-space A = null-space (transpose A)

```

```

  unfolding null-space-def left-null-space-def transpose-vector ..

```

```

lemma null-space-eq-left-null-space-transpose: null-space A = left-null-space (transpose A)

```

```

  using left-null-space-eq-null-space-transpose[of transpose A]

```

```

  unfolding transpose-transpose ..

```

```

lemma row-space-eq-col-space-transpose:

```

```

  fixes A::'a::{field} ^columns ^rows

```

```

  shows row-space A = col-space (transpose A)

```

```

  unfolding col-space-def row-space-def columns-transpose[of A] ..

```

```

lemma col-space-eq-row-space-transpose:

```

```

  fixes A::'a::{field} ^n ^m

```

```

  shows col-space A = row-space (transpose A)

```

```

  unfolding col-space-def row-space-def unfolding rows-transpose[of A] ..

```

4.2 Proving that they are subspaces

lemma *subspace-null-space*:

fixes $A::'a::\{\text{field}\}^{\wedge n}{}^{\wedge m}$
shows $\text{vec.subspace (null-space } A)$
by (*auto simp: vec.subspace-def null-space-def vec.scale vec.add*)

lemma *subspace-left-null-space*:

fixes $A::'a::\{\text{field}\}^{\wedge n}{}^{\wedge m}$
shows $\text{vec.subspace (left-null-space } A)$
unfolding *left-null-space-eq-null-space-transpose* **using** *subspace-null-space* .

lemma *subspace-row-space*:

shows $\text{vec.subspace (row-space } A)$ **by** (*metis row-space-def vec.subspace-span*)

lemma *subspace-col-space*:

shows $\text{vec.subspace (col-space } A)$ **by** (*metis col-space-def vec.subspace-span*)

4.3 More useful properties and equivalences

lemma *col-space-eq*:

fixes $A::'a::\{\text{field}\}^{\wedge m}::\{\text{finite, wellorder}\}^{\wedge n}$
shows $\text{col-space } A = \{y. \exists x. A * v x = y\}$

proof (*unfold col-space-def vec.span-finite[OF finite-columns], auto*)

fix x

show $A * v x \in \text{range } (\lambda u. \sum_{v \in \text{columns } A} u v * s v)$ **using** *matrix-vmult-column-sum[of A x]* **by** *auto*

next

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

let $?g = \lambda y. \{i. y = \text{column } i \text{ } A\}$

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

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

proof (*unfold matrix-mult-sum, rule exI[of - ?x], auto*)

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

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

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

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

unfolding *union-univ* ..

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

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

also have $\dots = \text{sum } ((\text{sum } (\lambda i. (\text{if } i = (\text{LEAST } a. \text{column } i \text{ } A = \text{column } a \text{ } A) \text{ then } u (\text{column } i \text{ } A) \text{ else } 0) * s \text{column } i \text{ } A)) \circ ?g)$

$(\text{columns } A)$ **by** (*rule sum.reindex, simp add: inj*)

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

proof (*rule sum.cong, auto*)

fix x
assume x -in-cols: $x \in \text{columns } A$
obtain b **where** b : $x = \text{column } b \ A$ **using** x -in-cols **unfolding** columns-def **by**
blast
let $?f = (\lambda i. (\text{if } i = (\text{LEAST } a. \text{column } i \ A = \text{column } a \ A) \text{ then } u (\text{column } i \ A) \text{ else } 0) *s \text{column } i \ A)$
have sum-rw : $\text{sum } ?f (\{i. x = \text{column } i \ A\} - \{\text{LEAST } a. x = \text{column } a \ A\}) = 0$
by (*rule sum.neutral, auto*)
have $\text{sum } ?f \{i. x = \text{column } i \ A\} = ?f (\text{LEAST } a. x = \text{column } a \ A) + \text{sum } ?f (\{i. x = \text{column } i \ A\} - \{\text{LEAST } a. x = \text{column } a \ A\})$
apply (*rule sum.remove, auto, rule LeastI-ex*)
using x -in-cols **unfolding** columns-def **by** *auto*
also have $\dots = ?f (\text{LEAST } a. x = \text{column } a \ A)$ **unfolding** sum-rw **by** *simp*
also have $\dots = u \ x *s \ x$
proof (*auto, rule LeastI2*)
show $x = \text{column } b \ A$ **using** b .
fix xa
assume x : $x = \text{column } xa \ A$
show $u (\text{column } xa \ A) *s \text{column } xa \ A = u \ x *s \ x$ **unfolding** x ..
next
assume $(\text{LEAST } a. x = \text{column } a \ A) \neq (\text{LEAST } a. \text{column } (\text{LEAST } c. x = \text{column } c \ A) \ A = \text{column } a \ A)$
moreover have $(\text{LEAST } a. x = \text{column } a \ A) = (\text{LEAST } a. \text{column } (\text{LEAST } c. x = \text{column } c \ A) \ A = \text{column } a \ A)$
by (*rule Least-equality[symmetric], rule LeastI2, simp-all add: b, rule Least-le, metis (lifting, full-types) LeastI*)
ultimately show $u \ x = 0$ **by** *contradiction*
qed
finally show $(\sum i \mid x = \text{column } i \ A. (\text{if } i = (\text{LEAST } a. \text{column } i \ A = \text{column } a \ A) \text{ then } u (\text{column } i \ A) \text{ else } 0) *s \text{column } i \ A) = u \ x *s \ x$.
qed
finally show $(\sum i \in \text{UNIV}. (\text{if } i = (\text{LEAST } a. \text{column } i \ A = \text{column } a \ A) \text{ then } u (\text{column } i \ A) \text{ else } 0) *s \text{column } i \ A) = (\sum y \in \text{columns } A. u \ y *s \ y)$.
qed
qed

corollary *col-space-eq'*:

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

lemma *row-space-eq*:

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

lemma *null-space-eq-ker*:

```

fixes f::('a::fieldn) => ('am)
assumes lf: Vector-Spaces.linear (*s) (*s) f
shows null-space (matrix f) = {x. f x = 0}
unfolding null-space-def using matrix-works [OF lf] by auto

lemma col-space-eq-range:
fixes f::('a::fieldn::{finite, wellorder}) => ('am)
assumes lf: Vector-Spaces.linear (*s) (*s) f
shows col-space (matrix f) = range f
unfolding col-space-eq unfolding matrix-works[OF lf] by blast

lemma null-space-is-preserved:
fixes A::'a::{field}colsrows
assumes P: invertible P
shows null-space (P**A) = null-space A
unfolding null-space-def
using P matrix-inv-left matrix-left-invertible-ker matrix-vector-mul-assoc ma-
trix-vector-mult-0-right
by metis

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

```

qed
end

5 Rank Nullity Theorem of Linear Algebra

theory *Dim-Formula*
 imports *Fundamental-Subspaces*
begin

context *vector-space*
begin

5.1 Previous results

Linear dependency is a monotone property, based on the monotonicity of linear independence:

lemma *dependent-mono*:
 assumes *d:dependent A*
 and *A-in-B: A ⊆ B*
 shows *dependent B*
 using *independent-mono [OF - A-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 *fin-A: finite A*
 and *ind: independent A*
 and *sum: (∑ x∈A. scale (f x) x) = 0*
 shows $\forall x \in A. f x = 0$
 using *fin-A ind local.dependent-finite sum by blast*

end

context *finite-dimensional-vector-space*
begin

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

$\llbracket \text{finite } A; \text{ local.independent } A; (\sum x \in A. f x * s x) = (0 :: 'b) \rrbracket$
 $\implies \forall x \in A. f x = (0 :: 'a)$

holds:

corollary *scalars-zero-if-independent-euclidean*:
 assumes *ind: independent A*
 and *sum: (∑ x∈A. scale (f x) x) = 0*
 shows $\forall x \in A. f x = 0$

using *finiteI-independent ind scalars-zero-if-independent sum* **by** *blast*

end

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

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

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

context *vector-space*

begin

lemma *inj-on-extended*:

assumes *lf: Vector-Spaces.linear scaleB scaleC f*

and *f: finite C*

and *ind-C: independent C*

and *C-eq: C = B \cup W*

and *disj-set: B \cap W = {}*

and *span-B: {x. f x = 0} \subseteq span B*

shows *inj-on f W*

— The proof is carried out by reductio ad absurdum

proof (*unfold inj-on-def, rule+, rule ccontr*)

interpret *lf: Vector-Spaces.linear scaleB scaleC f* **using** *lf* **by** *simp*

— Some previous consequences of the premises that are used later:

have *fin-B: finite B* **using** *finite-subset [OF - f] C-eq* **by** *simp*

have *ind-B: independent B* **and** *ind-W: independent W*

using *independent-mono[OF ind-C] C-eq* **by** *simp-all*

— The proof starts here; we assume that there exist two different elements

— with the same image:

fix *x::'b* **and** *y::'b*

assume *x: x \in W* **and** *y: y \in W* **and** *f-eq: f x = f y* **and** *x-not-y: x \neq y*

have *fin-yB: finite (insert y B)* **using** *fin-B* **by** *simp*

have *f (x - y) = 0* **by** (*metis diff-self f-eq lf.diff*)

hence *x - y \in {x. f x = 0}* **by** *simp*

hence $\exists g. (\sum v \in B. \text{scale } (g v) v) = (x - y)$ **using** *span-B*

unfolding *span-finite [OF fin-B]* **by** *force*

then obtain *g* **where** *sum: (\sum v \in B. \text{scale } (g v) v) = (x - y)* **by** *blast*

— We define one of the elements as a linear combination of the second element and the ones in B

```

define  $h :: 'b \Rightarrow 'a$  where  $h\ a = (if\ a = y\ then\ 1\ else\ g\ a)$  for  $a$ 
have  $x = y + (\sum\ v \in B.\ scale\ (g\ v)\ v)$  using  $sum$  by  $auto$ 
also have  $\dots = scale\ (h\ y)\ y + (\sum\ v \in B.\ scale\ (g\ v)\ v)$  unfolding  $h\text{-def}$  by  $simp$ 
also have  $\dots = scale\ (h\ y)\ y + (\sum\ v \in B.\ scale\ (h\ v)\ v)$ 
  apply  $(unfold\ add\text{-left}\text{-cancel},\ rule\ sum.cong)$ 
  using  $y\ h\text{-def}\ empty\text{-iff}\ disj\text{-set}$  by  $auto$ 
also have  $\dots = (\sum\ v \in (insert\ y\ B).\ scale\ (h\ v)\ v)$ 
  by  $(rule\ sum.insert[symmetric],\ rule\ fin\text{-}B)$ 
   $(metis\ (lifting)\ IntI\ disj\text{-set}\ empty\text{-iff}\ y)$ 
finally have  $x\text{-in}\text{-span}\text{-}yB: x \in span\ (insert\ y\ B)$ 
  unfolding  $span\text{-finite}[OF\ fin\text{-}yB]$  by  $auto$ 
— We have that a subset of elements of  $C$  is linearly dependent
have  $dep: dependent\ (insert\ x\ (insert\ y\ B))$ 
  by  $(unfold\ dependent\text{-def},\ rule\ bezI\ [of\ -\ x])$ 
   $(metis\ Diff\text{-insert}\text{-absorb}\ Int\text{-iff}\ disj\text{-set}\ empty\text{-iff}\ insert\text{-iff}\ x\ x\text{-in}\text{-span}\text{-}yB\ x\text{-not}\text{-}y,\ simp)$ 
— Therefore, the set  $C$  is also dependent:
hence  $dependent\ C$  using  $C\text{-eq}\ x\ y$ 
  by  $(metis\ Un\text{-commute}\ Un\text{-upper}2\ dependent\text{-mono}\ insert\text{-absorb}\ insert\text{-subset})$ 
— This yields the contradiction, since  $C$  is independent:
thus  $False$  using  $ind\text{-}C$  by  $contradiction$ 
qed
end

```

5.2 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  $finite\text{-dimensional}\text{-vector}\text{-space}$ 
begin

```

```

theorem  $rank\text{-nullity}\text{-theorem}$ :

```

```

  assumes  $l: Vector\text{-Spaces.linear}\ scale\ scaleC\ f$ 

```

```

  shows  $dimension = dim\ \{x.\ f\ x = 0\} + vector\text{-space.dim}\ scaleC\ (range\ f)$ 

```

```

proof —

```

— For convenience we define abbreviations for the universe set, V , and the kernel of f

```

  interpret  $l: Vector\text{-Spaces.linear}\ scale\ scaleC\ f$  by  $fact$ 

```

```

  define  $V :: 'b\ set$  where  $V = UNIV$ 

```

```

  define  $ker\text{-}f$  where  $ker\text{-}f = \{x.\ f\ x = 0\}$ 

```

— The kernel is a proper subspace:

```

  have  $sub\text{-}ker: subspace\ \{x.\ f\ x = 0\}$  using  $l.subspace\text{-kernel}$  .

```

— The kernel has its proper basis, B :
obtain B **where** B -in-ker: $B \subseteq \{x. f x = 0\}$
and $independent$ - B : $independent\ B$
and ker -in-span: $\{x. f x = 0\} \subseteq span\ B$
and $card$ - B : $card\ B = dim\ \{x. f x = 0\}$ **using** $basis$ -exists **by** $blast$

— The space V has a (finite dimensional) basis, C :
obtain C **where** B -in- C : $B \subseteq C$ **and** C -in- V : $C \subseteq V$
and $independent$ - C : $independent\ C$
and $span$ - C : $V = span\ C$
unfolding V -def
by ($metis\ independent$ - $B\ extend$ -basis-superset $independent$ -extend-basis $span$ -extend-basis $span$ -superset)

— The basis of V , C , can be decomposed in the disjoint union of the basis of the kernel, B , and its complementary set, $C - B$
have C -eq: $C = B \cup (C - B)$ **by** ($rule\ Diff$ -partition [$OF\ B$ -in- C , $symmetric$])
have eq - fC : $f\ ' C = f\ ' B \cup f\ ' (C - B)$
by ($subst\ C$ -eq, $unfold\ image$ -Un, $simp$)

— The basis C , and its image, are finite, since V is finite-dimensional
have $finite$ - C : $finite\ C$
using $finiteI$ -independent[$OF\ independent$ - C].
have $finite$ - fC : $finite\ (f\ ' C)$ **by** ($rule\ finite$ -imageI [$OF\ finite$ - C])

— The basis B of the kernel of f , and its image, are also finite
have $finite$ - B : $finite\ B$ **by** ($rule\ rev$ -finite-subset [$OF\ finite$ - $C\ B$ -in- C])
have $finite$ - fB : $finite\ (f\ ' B)$ **by** ($rule\ finite$ -imageI [$OF\ finite$ - B])

— The set $C - B$ is also finite
have $finite$ - CB : $finite\ (C - B)$ **by** ($rule\ finite$ -Diff [$OF\ finite$ - C , of B])
have dim -ker-le- dim - V : $dim\ (ker$ - $f) \leq dim\ V$
using 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 $dimension = dim\ V$ **unfolding** V -def dim -UNIV $dimension$ -def
by ($metis\ basis$ -card-eq-dim $dimension$ -def $independent$ -Basis $span$ -Basis top -greatest)
also **have** $dim\ V = dim\ C$ **unfolding** $span$ - $C\ dim$ -span ..
also **have** $\dots = card\ C$
using $basis$ -card-eq-dim [of $C\ C$, OF - $span$ -superset $independent$ - C] **by** $simp$
also **have** $\dots = card\ (B \cup (C - B))$ **using** C -eq **by** $simp$
also **have** $\dots = card\ B + card\ (C - B)$
by ($rule\ card$ -Un-disjoint[$OF\ finite$ - $B\ finite$ - CB], $fast$)
also **have** $\dots = dim\ ker$ - $f + card\ (C - B)$ **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** $\dots = dim\ ker$ - $f + lvs2$. $dim\ (range\ f)$
proof ($unfold\ add$ -left-cancel)
define W **where** $W = C - B$
have $finite$ - W : $finite\ W$ **unfolding** W -def **using** $finite$ - CB .
have $finite$ - fW : $finite\ (f\ ' W)$ **using** $finite$ -imageI [$OF\ finite$ - W] .

have $\text{card } W = \text{card } (f \text{ ' } W)$
by (rule *card-image* [*symmetric*], rule *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 $\dots = \text{l.vs2.dim } (\text{range } f)$
— The image set of W is independent and its span contains the range of f , so it is a basis of the range:
proof (rule *l.vs2.basis-card-eq-dim*)
— 1. The image set of W generates the range of f :
show $\text{range } f \subseteq \text{l.vs2.span } (f \text{ ' } W)$
proof (*unfold l.vs2.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 $v :: 'b$
have $fV\text{-span}: f \text{ ' } V \subseteq \text{l.vs2.span } (f \text{ ' } C)$
by (*simp add: span-C l.span-image*)
have $\exists g. (\sum x \in f \text{ ' } C. \text{scaleC } (g \ x) \ x) = f \ v$
using $fV\text{-span}$ **unfolding** $V\text{-def}$
using l.vs2.span-finite [*OF finite-fC*]
by (*metis (no-types, lifting) V-def rangeE rangeI span-C l.span-image*)
then obtain g **where** $fv: f \ v = (\sum x \in f \text{ ' } C. \text{scaleC } (g \ x) \ x)$ **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 $\text{zero-fB}: (\sum x \in f \text{ ' } B. \text{scaleC } (g \ x) \ x) = 0$
using $B\text{-in-ker}$ **by** (*auto intro!: sum.neutral*)
have $\text{zero-inter}: (\sum x \in (f \text{ ' } B \cap f \text{ ' } W). \text{scaleC } (g \ x) \ x) = 0$
using $B\text{-in-ker}$ **by** (*auto intro!: sum.neutral*)
have $f \ v = (\sum x \in f \text{ ' } C. \text{scaleC } (g \ x) \ x)$ **using** fv .
also have $\dots = (\sum x \in (f \text{ ' } B \cup f \text{ ' } W). \text{scaleC } (g \ x) \ x)$
using $eq-fC$ $W\text{-def}$ **by** *simp*
also have $\dots =$
 $(\sum x \in f \text{ ' } B. \text{scaleC } (g \ x) \ x) + (\sum x \in f \text{ ' } W. \text{scaleC } (g \ x) \ x)$
 $- (\sum x \in (f \text{ ' } B \cap f \text{ ' } W). \text{scaleC } (g \ x) \ x)$
using sum-Un [*OF finite-fB finite-fW*] **by** *simp*
also have $\dots = (\sum x \in f \text{ ' } W. \text{scaleC } (g \ x) \ x)$
unfolding zero-fB zero-inter **by** *simp*
— We have proved that the image set of W is a generating set of the range of f
finally show $f \ v \in \text{range } (\lambda u. \sum v \in f \text{ ' } W. \text{scaleC } (u \ v) \ v)$ **by** *auto*
qed
— 2. The image set of W is linearly independent:
show $\text{l.vs2.independent } (f \text{ ' } W)$
using finite-fW
proof (rule *l.vs2.independent-if-scalars-zero*)
— Every linear combination (given by g) of the elements of the image set of W equal to zero, requires every coefficient to be zero:
fix $g :: 'c \Rightarrow 'a$ **and** $w :: 'c$
assume $\text{sum}: (\sum x \in f \text{ ' } W. \text{scaleC } (g \ x) \ x) = 0$ **and** $w: w \in f \text{ ' } W$

have $0 = (\sum x \in f^{-1} W. \text{scale}_C (g x) x)$ **using** *sum* **by** *simp*
also have $\dots = \text{sum } ((\lambda x. \text{scale}_C (g x) x) \circ f) W$
by (*rule sum.reindex*, *rule inj-on-extended*[*OF l*, *of C B*])
(unfold W-def, rule finite-C, rule independent-C, rule C-eq, simp,
rule ker-in-span)
also have $\dots = (\sum x \in W. \text{scale}_C ((g \circ f) x) (f x))$ **unfolding** *o-def* ..
also have $\dots = f (\sum x \in W. \text{scale } ((g \circ f) x) x)$
unfolding *l.sum*[*symmetric*] *l.scale*[*symmetric*] **by** *simp*
finally have *f-sum-zero*: $f (\sum x \in W. \text{scale } ((g \circ f) x) x) = 0$ **by** (*rule sym*)
hence $(\sum x \in W. \text{scale } ((g \circ f) x) x) \in \text{ker-}f$ **unfolding** *ker-f-def* **by** *simp*
hence $\exists h. (\sum v \in B. \text{scale } (h v) v) = (\sum x \in W. \text{scale } ((g \circ f) x) x)$
using *span-finite*[*OF finite-B*] **using** *ker-in-span*
unfolding *ker-f-def* **by** *force*
then obtain *h* **where**
sum-h: $(\sum v \in B. \text{scale } (h v) v) = (\sum x \in W. \text{scale } ((g \circ f) x) x)$ **by** *blast*
define *t* **where** $t a = (\text{if } a \in B \text{ then } h a \text{ else } -((g \circ f) a))$ **for** *a*
have $0 = (\sum v \in B. \text{scale } (h v) v) + - (\sum x \in W. \text{scale } ((g \circ f) x) x)$
using *sum-h* **by** *simp*
also have $\dots = (\sum v \in B. \text{scale } (h v) v) + (\sum x \in W. -(\text{scale } ((g \circ f) x) x))$
x))
unfolding *sum-negf* ..
also have $\dots = (\sum v \in B. \text{scale } (t v) v) + (\sum x \in W. -(\text{scale } ((g \circ f) x) x))$
unfolding *add-right-cancel* **unfolding** *t-def* **by** *simp*
also have $\dots = (\sum v \in B. \text{scale } (t v) v) + (\sum x \in W. \text{scale } (t x) x)$
by (*unfold add-left-cancel t-def W-def, rule sum.cong*) *simp*+
also have $\dots = (\sum v \in B \cup W. \text{scale } (t v) v)$
by (*rule sum.union-inter-neutral* [*symmetric*], *rule finite-B*, *rule finite-W*)

(simp add: W-def)
finally have $(\sum v \in B \cup W. \text{scale } (t v) v) = 0$ **by** *simp*
hence *coef-zero*: $\forall x \in B \cup W. t x = 0$
using *C-eq scalars-zero-if-independent* [*OF finite-C independent-C*]
unfolding *W-def* **by** *simp*
obtain *y* **where** *w-fy*: $w = f y$ **and** *y-in-W*: $y \in W$ **using** *w* **by** *fast*
have $-g w = t y$
unfolding *t-def w-fy* **using** *y-in-W* **unfolding** *W-def* **by** *simp*
also have $\dots = 0$ **using** *coef-zero y-in-W* **unfolding** *W-def* **by** *simp*
finally show $g w = 0$ **by** *simp*
qed
qed *auto*
finally show $\text{card } (C - B) = l.\text{vs2}.\text{dim } (\text{range } f)$ **unfolding** *W-def* .
qed
finally show *?thesis* **unfolding** *V-def ker-f-def* **unfolding** *dim-UNIV* .
qed
qed
end

5.3 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 $A::'a::\{\text{field}\}^{\wedge}\text{cols}::\{\text{finite}, \text{wellorder}\}^{\wedge}\text{rows}$

shows $\text{ncols } A = \text{vec.dim } (\text{null-space } A) + \text{vec.dim } (\text{col-space } A)$

using *vec.rank-nullity-theorem*[*OF matrix-vector-mul-linear-gen*, *of A*]

apply (*subst* (2 3) *matrix-of-matrix-vector-mul* [*of A*, *symmetric*])

unfolding *null-space-eq-ker*[*OF matrix-vector-mul-linear-gen*]

unfolding *col-space-eq-range* [*OF matrix-vector-mul-linear-gen*]

unfolding *vec.dimension-def* *ncols-def* *card-cart-basis*

by *simp*

end

References

- [1] S. Axler. *Linear Algebra Done Right*. Springer, 2nd edition, 1997.
- [2] M. S. Gockenbach. *Finite Dimensional Linear Algebra*. CRC Press, 2010.