Final Report for Deliverable Nr. 2.1

Basic Linear Algebra over a Field

Responsible: Yves Bertot, Yves.Bertot@sophia.inria.fr
Written by: Laurence Rideau, Laurence.Rideau@inria.fr
and Laurent Thery, Laurent.Thery@inria.fr
Site: INRIA, France

Deliverable Date: February 2011

Abstract

Linear algebra plays a central role in most of the other tasks of the Formath project. In this work package, we intend to formalise most of the linear algebra program as taught in first-year university curricula. This report covers specifically finite dimensional vector spaces.

1 Introduction

In this report, we detail our formalisation of finite vector spaces. It aims at providing all the very basic notions related to vector spaces. As this formalisation has been done within the SSReflect library, the reader needs to have a good familiarity with the library in order to understand how what has been developed fits in the library. In particular, this works is an encapsulation over the linear algebra described in [2]. It provides a more direct access to the usual notion of vector spaces of finite dimension. The presentation is organised as follows. We first present the new type `vectType` K that represents vector sets, then we present the different functions and predicates associated with this type.

1.1 The new type `vectType`

Following the methodology described in [1] to define this new types, vector spaces are defined in two stages. First, given an arbitrary field K, a new type `vectType` K is defined. An object VT of type `vectType` K represents a structure with a carrier (the set of vectors), the usual operations (addition `+` and scalar product `*:`) and their properties among which is finite dimension. As VT is coercible to its carrier, an object v of type VT is then a vector. More precisely, the `vectType` structure is an extension of the algebraic hierarchy developed in `ssralg.v`. It is defined as follows:

```
Structure type vectType (R : ringType} : Type :=
   Pack {sort : Type; _ : class_of sort}.

Structure class_of (V : Type) : Type := Class {
    base : Lmodule.class_of R V;
   mixin : mixin_of (Lmodule.Pack V base)
 }.
```

As the definition does not require for K any of the field properties, in the definition an arbitrary ring type R is used instead of. It is only later when proving properties that R is actually instantiated by K. The field `sort` is the carrier of the structure, it has to have a left-module structure (`lmodType`) associated. It is from this left-module structure that the vector type structure inherits the addition of two vectors (v_1 `+` v_2) and the scalar product (k `*:` v).

Now, the field `mixin` constains what is added with respect to the left module structure: the dimension finiteness. This is encoded as a bijection between elements of V and rows of elements of R.

```
Structure mixin_of (V : lmodType R) : Type := Mixin {
               dim : nat;
  v2rv_isomorphism : V-> 'rV[R]_dim;
                 _ : linear v2rv_isomorphism;
                 _ : bijective v2rv_isomorphism
}.
```

Note that the bijection is a linear application. This lets us import directly all the operations that are defined in `mxalgebra.v`.

Now that the new structure is defined, there are two separate uses. First, it can be used abstractly, stating the existence of a type of vectors:

```
Section Test.

Variable (K : fieldType) (VT : vectType K).

Check (fun (v1 v2 : VT) => v1 + (2%:R *: v2)).
  (* : VT -> VT -> VT *)

End Test.
```

Second, it can be used specifically. For this, the functions `VectMixin` and `VectType` let us build the mixin and the structure respectively. Let us take an example. Suppose we have a type V on which there is canonically a left module structure over a specific field K. We first need to define the bijection and its properties.

```
Definition v2rv (v : V) : 'rV[K]_ ... := ... .

Lemma v2rv_is_linear : linear v2rv.
Proof. ... Qed.

Canonical Structure v2rv_linear :=  Linear v2rv_linear_proof.

Lemma v2rv_bij : bijective v2rv.
Proof. ... Qed.
```

The actual `vectType` is then easily derived as:

```
Definition VT_mixin := VectMixin v2rv_linear v2rv_bij.
Canonical Structure VT := VectType K VT_mixin.
```

In the definition, we had to make the difference between V that has an associated left-module structure and VT the actual vector type structure. In the following, we do not refer anymore to left-module structures, so V is always denoting a vector type structure.

Two operations are defined directly on vector type structures:

`(`V_1 `:+:` V_2`)%VS` constructs the `vectType` composed of the pairs of one element of V_1 and one element of V_2.

`(`V `^` n`)%VS` constructs the `vectType` composed of the iterated pairs composed of n elements of V.

Note that the scope associated with vectors is `vspace_scope` and its associated key is `VS`.

2 Vector space

Now, that we have the `vectType` structure, we can derive the proper notion of vector space. If V is of type `vectType` K, the type {`vspace` V} defines the set of vector spaces on V. It is encoded as a row of vectors but this encoding should neither used directly. If we have a vector v of V, the vector space generated by the line of v is denoted by v`%:VS`. A special function `fullv` returns the full vector space, i.e the vector space composed of all the elements of V.

We can add vector spaces, intersect them, take the complement, and test for membership and inclusion. For example, here is one of the lemma about membership for addition:

```
Lemma memv_add : forall (v1 v2 : V) (vs1 vs2 : {vspace V}),
  v1 \in vs1 -> v2 \in vs2 -> v1 + v2 \in (vs1 + vs2)%VS.
```

and its equivalent for intersection

```
Lemma memv_cap : forall (v : V) (vs1 vs2 : {vspace V}),
  (v \in (vs1 :&: vs2)%VS) = (v \in vs1) && (v \in vs2).
```

This forms a modular lattice as stated by the following two theorems:

```
Lemma vspace_modl : forall vs1 vs2 vs3 : {vspace V},
  (vs1 <= vs3 -> vs1 + (vs2 :&: vs3) = (vs1 + vs2) :&: vs3)%VS.

Lemma vspace_modr : forall vs1 vs2 vs3: {vspace V},
  (vs3 <= vs1 -> (vs1 :&: vs2) + vs3 = vs1 :&: (vs2 + vs3))%VS.
```

Addition and intersection can be iterated. Here are two examples of lemmas that express the relation between summation and inclusion.

```
Lemma subv_sumP :
  forall (I : finType) (P : pred I) (vs_ : I -> {vspace V})
         (vs : {vspace V}),
  reflect (forall i, P i -> vs_ i <= vs)%VS
          (\sum_(i | P i) vs_ i <= vs)%VS.

Lemma subv_bigcapP :
  forall (I : finType) (P : pred I) (vs_ : I -> {vspace V})
         (vs : {vspace V}),
 reflect (forall i, P i -> vs <= vs_ i)%VS
         (vs <= \bigcap_(i | P i) vs_ i)%VS.
```

Associated to a vector space, there is a notion of dimension. Here is an example of a property of dimension:

```
Lemma dimv_leqif_sup : forall vs1 vs2,
  (vs1 <= vs2)%VS -> \dim vs1 <= \dim vs2 ?= iff (vs2 <= vs1)%VS.
```

where the notation `?=` indicates that the equality holds if and only `vs2 = vs1`. All this is standard. What is less standard is to provide an explicit function `_^C` that returns an arbitray complement to a vector space. This function has two main properties:

```
Lemma addv_complf : forall vs : {vspace V}, (vs + vs^C = fullv)%VS.

Lemma capv_compl : forall vs: {vspace V}, (vs :&: vs^C)%VS = 0%:VS.
```

With this function, the difference of two vector space (vs_1 `:\:` vs_2)`%VS` is defined as the intersection of vs_1 and the complement of the intersection of vs_1 and vs_2.

A special care has been taken in handling the direct sum of vector spaces. As we want the definition to work both for the binary case (`vs1` and `vs2` are in direct sum) and the n-ary case (the `vs_i` are in direct sums), the predicate uses the shape of its argument to compute the subcomponents. So `(directv (vs1 + vs2)%VS)` reads as `vs1` and `vs2` are in direct sum while `(directv (\sum_i vs_i)%VS)` reads as the `vs_i` are in direct sums. Not that, as the predicat acts as a macro, two applications of `directv` on two convertible terms may lead to different propositions.

The view `directvP` relates vector spaces in direct sum with their dimensions. Here are some examples.

The view `directvP` applied to a term of type (`directv (`vs_1 `+` vs_2`)`) leads to
`\dim (`vs_1 `+` vs_2`) = (\dim` vs_1 `+ \dim` vs_2`)`.

If the term is of type (`directv (`vs_1 `+` vs_2 `+` vs_3`)`), this leads to
`\dim (`vs_1 `+` vs_2 `+` vs_3`) = (\dim` vs_1 `+ \dim` vs_2 `+ \dim` vs_3`)`.

If the term is of type (`directv (\sum_(`i `| P` i`)` vs_i`)`), this leads to
`\dim (\sum_(`i `| P` i`)` vs_i`) = (\sum_(`i `| P` i`) \dim` vs_i`)%N`.

3 Bases and coordinates

We have seen how to manipulate vector spaces. In this section, we are concentrating on showing how to build vector spaces from vectors and some related notions such as the ones of bases and coordinates. The function `vpick` is a function that picks an element in a vector space with a special treatment for the trivial vector space composed of 0:

```
Lemma memv_pick : forall vs : {vspace V}, vpick vs \in vs.

Lemma vpick0 :
  forall vs : {vspace V}, (vpick vs == 0) = (vs == 0%:VS).
```

Conversely given a sequence of vectors l, (`span` l) constructs the vector space generated by the vectors in l. Furthermore, given a vector v, (`coord` l v) returns the coordinates of v in terms of a finite function from indices of l (0,1,..., (`size` l)`.-1`) to K. The relation between the spanning and the coordinates is given by the following lemma:

```
Lemma coord_span : forall (l : seq V) (v : V),
  v \in span l -> v = \sum_(i < size l) coord l v i *: l`_i.
```

A predicate `free` indicates if a sequence l is free.

```
Definition free l := \dim (span l) == size l.
```

We have now all the elements to talk about bases. Given a vector space vs and a sequence of vectors l, (`is_basis` vs l) indicates that l is a basis for vs, i.e. l generates vs and l is free. Conversely, given a vector space vs, (`vbasis` vs) returns a basis for vs. This is built iteratively by using the pick function and the complement.

4 Homomorphisms

Since it is always possible to extend homomorphism so that elements outside the domain are sent to `0`, homomorphisms are defined for all vectors, i.e their domains are always a `vectType` K. So we have:

> `'Hom(`V`,`W`)` represents the `vectType` that contains the homomorphisms from V to W where V and W are two (`vectType` K).
>
> `'End(`V`)` represents the `vectType` that contains the endomorphisms of V, i.e. `'Hom(`V`,`V`)`.

To build an homomorphism, the simplest way is to use the function `lapp_of_fun` that constructs an homomorphism out of an ordinary function from V to W. Its type is `(`$V \to W$`) → 'Hom(`V`,`W`)`. It takes the image by `f` of a basis of `(fullv V)` and builds the homomorphism by linearity. Of course, (`lapp_of_fun` f) and f coincides if only if f is linear:

```
Lemma lapp_of_funK :
  forall (R : ringType) (V W : vectType R) (f : V -> W),
    linear f -> lapp_of_fun f =1 f
```

A vector type structure is associated to `'Hom(`V`,`W`)`, so it is possible to add two homomorhisms or multiply an homomorphism by a scalar. Two other operations are provided, (f_1 `\o` f_2)`%VS` gives the composition and (f `\^-1`)`%VS` gives the inverse when it exists and an arbitrary function otherwise.

Homomorphisms interact with vector spaces. It is possible to compute the image (or the preimage) of a vector space. Kernel and image for homomorphisms are also defined. For example, here are the lemma that explicits the image for the composition:

```
Lemma limg_comp :
  forall  (f : 'Hom(V,W)) (g : 'Hom(W,Z)) (vs : {vspace V}),
    ((g \o f) @: vs = g @: (f @: vs))%VS.
```

the lemma that states the monotonicity of the preimage:

```
Lemma lpre_img_monotone :
  forall  (f : 'Hom(V,W)) (vs1 vs2 : {vspace W}),
    (vs1 <= vs2)%VS-> (f @^-1: vs1 <= f @^-1: vs2)%VS.
```

and the lemma that states when the inverse makes sense:

```
Lemma inv_lker0 : forall  (f : 'Hom(V,W)),
  lker f == 0%:VS -> (f \^-1 \o f = \1)%VS.
```

Projections are defined as homomorphisms. We have three of them:

- (`addv_pi1` V_1 V_2) projects elements of (V_1 `+` V_2)`%VS` onto V_1 along V_2.
- (`addv_pi2` V_1 V_2) projects elements of (V_1 `+` V_2)`%VS` onto V_2 along V_1.
- (`sumv_pi` V `_` P i) projects elements of (`\sum_(`j `|` P j`)` V_j) onto V_i along (`\sum_(`j `|` j `!=` i `&&` P j`)` V_j)

References

[1] F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging mathematical structures. In *TPHOLs*, volume 5674 of *LNCS*, pages 327–342. Springer, 2009.

[2] G. Gonthier. Point-free, set-free concrete linear algebra. In *ITP*, 2011. to appear.

A Appendix: Header Comment of vector.v

```
(*****************************************************************************)
(*  * Finite dimensional vector spaces                                       *)
(*      vectType K      == type for findim vector space structure.           *)
(*      vectMixin K     == builds the mixins containing the definition       *)
(*                         of a vector space over K.                         *)
(*      VectType T M    == packs the mixin M to build a vector space of type *)
(*                         vectType K. The field K will be inferred from the *)
(*                         mixin M. The underlying type T should have a      *)
(*                         lmodType canonical structure.                     *)
(*      {vspace V}      == a vector space over V                             *)
(*     (V :+: W)%VS     == the product of two vecTypes                       *)
(*     (V ^ n)%VS       == the iterated product of a vecType                 *)
(*     'Hom(V,W)        == homomorphisms from V to W                         *)
(*     'End(V)          == endomorphisms of V                                *)
(*                                                                           *)
(*  Functions:                                                               *)
(*         v%:VS        == a vector space of dimension 1 composed of v       *)
(*         0%:VS        == the singleton vector space                        *)
(*         fullv        == the complete vector space                         *)
(*      (V1 + V2)%VS    == union of two vector spaces                        *)
(*     (V1 :&: V2)%VS   == intersection of two vector spaces                 *)
(*       (V^C)%VS       == complement of a vector space                      *)
(*     (V1 :\: V2)%VS   == V1 :&: ((V1 :&: V2)^C)                            *)
(*       \dim V         == dimension of a vector space                       *)
(*      coord l v       == coordinates of v in the vector space generated    *)
(*                         by the sequence l                                 *)
(*     vpick vs         == pick an element of vs (0 iff vs is 0%:VS)         *)
(*       span l         == the linear span generated by the sequence l       *)
(*     vbasis vs        == a basis of vs                                     *)
(*     \1               == the unit linear function                          *)
(*     (f \o g)%VS      == the composite of two linear functions             *)
(*     (f \^-1)%VS      == the inverse linear function                       *)
(*     (f @: vs)%VS     == the image of vs by the linear function f          *)
(*     (f @^-1: vs)%VS  == the pre-image of vs by the linear function f      *)
(*     lker f           == the kernel of the linear function f               *)
(*     limg f           == the image of the linear function f                *)
(* (addv_pi1 V1 V2)     == projection of (V1 + V2)%VS onto V1 along V2       *)
(* (addv_pi2 V1 V2)     == projection of (V1 + V2)%VS onto V2 along V1       *)
(* (sumv_pi V_ P i)     == projection of (\sum_(j | P j) V_ j)%VS onto V_ i   *)
(*                         along  (\sum_(j | j != i && P j) V_ j)            *)
(*                                                                           *)
(*  Predicates:                                                              *)
(*     v \in vs         == v belongs to vs                                   *)
(*    (vs1 <= vs2)%VS   == vs1 is a subspace of vs2                          *)
(*      free l          == l is a list of free vectors                       *)
(*     is_span vs l     == l generates vs                                    *)
(*    is_basis vs l     == l is a basis of vs                                *)
(*     directv vexpr    == the vector spaces in the expression vexpr are     *)
(*                         in direct sum                                     *)
(*****************************************************************************)
```

B Appendix: Proved Lemmas

Lemma subv_refl : reflexive subv.

Lemma subv_trans : transitive subv.

Lemma subv_anti : antisymmetric subv.

Lemma injvP : $\forall$ *v1 v2*, reflect ($\exists$ *k*, *v1* = *k* *: *v2*) (*v1* “in *v2*%:VS).

Lemma mem0v : $\forall$ *vs*, 0 “in *vs*.

Lemma memv_inj : $\forall$ *v*, *v* “in *v*%:VS.

Lemma memvD : $\forall$ *vs v1 v2*, *v1* “in *vs* $\rightarrow$ *v2* “in *vs* $\rightarrow$ *v1* + *v2* “in *vs*.

Lemma memvMn : $\forall$ *vs v m*, *v* “in *vs* $\rightarrow$ *v* *+ *m* “in *vs*.

Lemma memvZl : $\forall$ *vs k v*, *v* “in *vs* $\rightarrow$ *k* *: *v* “in *vs*.

Lemma memvZ : $\forall$ *vs k v*, (*k* *: *v* “in *vs*) = (*k* == 0) —— (*v* “in *vs*).

Lemma memvNr : $\forall$ *vs v*, *v* “in *vs* $\rightarrow$ -v “in *vs*.

Lemma memvNl : $\forall$ *vs v*, -v “in *vs* $\rightarrow$ *v* “in *vs*.

Lemma memv_sub : $\forall$ *vs v1 v2*, *v1* “in *vs* $\rightarrow$ *v2* “in *vs* $\rightarrow$ *v1* - *v2* “in *vs*.

Lemma memvDl : $\forall$ *vs v1 v2*, *v1* “in *vs* $\rightarrow$ (*v1* + *v2* “in *vs*) = (*v2* “in *vs*).

Lemma memvDr : $\forall$ *vs v1 v2*, *v1* “in *vs* $\rightarrow$ (*v2* + *v1* “in *vs*) = (*v2* “in *vs*).

Lemma memvN : $\forall$ *vs v*, (-v “in *vs*) = (*v* “in *vs*).

Lemma memv_subl : $\forall$ *vs v1 v2*, *v1* “in *vs* $\rightarrow$ (*v1* - *v2* “in *vs*) = (*v2* “in *vs*).

Lemma memv_subr : $\forall$ *vs v1 v2*, *v1* “in *vs* $\rightarrow$ (*v2* - *v1* “in *vs*) = (*v2* “in *vs*).

Lemma memv_suml : $\forall$ (*I* : finType) (*P* : pred *I*) (*vs_* : *I* $\rightarrow$ *V*) *vs*,
(∀ *i*, *P i* $\rightarrow$ *vs_ i* “in *vs*) $\rightarrow$ “sum_(i | *P i*) *vs_ i* “in *vs*.

Lemma subvP : $\forall$ *vs1 vs2*,
reflect (∀ *v*, *v* “in *vs1* $\rightarrow$ *v* “in *vs2*) (*vs1* $\leq$ *vs2*)%VS.

Lemma vspaceP : $\forall$ *vs1 vs2*, reflect (*vs1* =i *vs2*) (*vs1* == *vs2*)%VS.

Lemma subset0v : $\forall$ *vs*, (0%:VS $\leq$ *vs*)%VS.

Lemma subv0 : $\forall$ *vs*, (*vs* $\leq$ 0%:VS)%VS = (*vs* == 0%:VS).

Lemma memv0 : $\forall$ *v*, *v* “in 0%:VS = (*v* == 0).

Lemma subvf : $\forall$ *vs*, (*vs* $\leq$ fullv)%VS.

Lemma memvf : $\forall$ *v*, *v* “in fullv.

Lemma memv_pick : $\forall$ *vs*, vpick *vs* “in *vs*.

Lemma vpick0 : $\forall$ *vs*, (vpick *vs* == 0) = (*vs* == 0%:VS).

Lemma subv_add : $\forall$ *vs1 vs2 vs3*,
(*vs1* + *vs2* $\leq$ *vs3*)%VS = (*vs1* $\leq$ *vs3*)%VS && (*vs2* $\leq$ *vs3*)%VS.

Lemma addvSl : $\forall$ *vs1 vs2*, (*vs1* $\leq$ *vs1* + *vs2*)%VS.

Lemma addvSr : $\forall$ *vs1 vs2*, (*vs2* $\leq$ *vs1* + *vs2*)%VS.

Lemma addvKl : $\forall$ *vs1 vs2*, (*vs1* $\leq$ *vs2*)%VS = (*vs1* + *vs2* == *vs2*)%VS.

Lemma addvKr: $\forall$ *vs1 vs2*, (*vs2* $\leq$ *vs1*)%VS = (*vs1* + *vs2* == *vs1*)%VS.

Lemma addvC : commutative addv.

Lemma addvA : associative addv.

Lemma addvS : $\forall$ *vs1 vs2 vs3 vs4*,
  $(vs1 \leq vs2 \to vs3 \leq vs4 \to vs1 + vs3 \leq vs2 + vs4)\%\mathrm{VS}$.

Lemma addvv : idempotent addv.

Lemma sum0v : left_id $0\%:\mathrm{VS}$ addv.

Lemma addv0 : right_id $0\%:\mathrm{VS}$ addv.

Lemma sumfv : left_zero fullv addv.

Lemma addvf : right_zero fullv addv.

Lemma memv_add : $\forall$ *v1 v2 vs1 vs2*,
  v1 "in *vs1* $\to$ *v2* "in *vs2* $\to$ *v1 + v2* "in $(vs1 + vs2)\%\mathrm{VS}$.

Lemma memv_addP : $\forall$ *v vs1 vs2*,
  reflect ($\exists$ *v1*, $\exists$ *v2*, [$\wedge$ *v1* "in *vs1*, *v2* "in *vs2* & $v = v1 + v2$])
    (v "in $(vs1 + vs2)\%\mathrm{VS}$).

Section BigSum.

Variable I : finType.

Lemma sumv_sup : $\forall$ *i0 P vs* $(vs\_ : I \to \{vspace\ V\})$,
  $P\ i0 \to (vs \leq vs\_\ i0)\%\mathrm{VS} \to$
   $(vs \leq$ "sum_$(\mathrm{i} \mid P\ i)\ vs_\ i)\%\mathrm{VS}$.

Lemma subv_sumP : $\forall$ P $(vs\_ : I \to \{vspace\ V\})$ *vs*,
  reflect ($\forall$ *i*, $P\ i \to vs\_\ i \leq vs)\%\mathrm{VS}$
    ("sum_$(\mathrm{i} \mid P\ i)\ vs_\ i \leq vs)\%\mathrm{VS}$.

Lemma memv_sumP : $\forall$ P $(vs\_ : I \to \{vspace\ V\})$ v,
  reflect ($\exists$ $v\_ : I \to V$,
    ($\forall$ *i*, $P\ i \to v\_\ i$ "in $vs\_\ i$) $\wedge$
     $v =$ "sum_$(\mathrm{i} \mid P\ i)\ v_\ i$)
    (v "in ("sum_$(\mathrm{i} \mid P\ i)\ vs_\ i)\%\mathrm{VS}$).

Lemma memv_sumr : $\forall$ P $(vs\_ : I \to V)$ $(Vs\_ : I \to \{vspace\ V\})$,
  ($\forall$ *i*, $P\ i \to vs\_\ i$ "in $Vs\_\ i$) $\to$
  "sum_$(\mathrm{i} \mid P\ i)\ vs_\ i$ "in ("sum_$(\mathrm{i} \mid P\ i)\ Vs_\ i)\%\mathrm{VS}$.

End BigSum.

Lemma subv_cap : $\forall$ *vs1 vs2 vs3*,
  $(vs1 \leq vs2 :\&: vs3)\%\mathrm{VS} = (vs1 \leq vs2)\%\mathrm{VS}$ && $(vs1 \leq vs3)\%\mathrm{VS}$.

Lemma capvSl : $\forall$ *vs1 vs2*, $(vs1 :\&: vs2 \leq vs1)\%\mathrm{VS}$.

Lemma capvSr : $\forall$ *vs1 vs2*, $(vs1 :\&: vs2 \leq vs2)\%\mathrm{VS}$.

Lemma capvKl : $\forall$ *vs1 vs2*, $(vs1 \leq vs2)\%\mathrm{VS} = (vs1 :\&: vs2 == vs1)\%\mathrm{VS}$.

Lemma capvKr : $\forall$ *vs1 vs2*, $((vs2 \leq vs1) = (vs1 :\&: vs2 == vs2))\%\mathrm{VS}$.

Lemma capvv : idempotent capv.

Lemma capvC : commutative capv.

Lemma capvA : associative capv.

Lemma capvS : $\forall$ *vs1 vs2 vs3 vs4*,
$(vs1 \leq vs2 \to vs3 \leq vs4 \to vs1 :\&: vs3 \leq vs2 :\&: vs4)\%\mathrm{VS}$.

Lemma cap0v : left_zero 0%:VS capv.

Lemma capv0 : right_zero 0%:VS capv.

Lemma capfv : left_id fullv capv.

Lemma capvf : right_id fullv capv.

Lemma memv_cap : $\forall$ *v vs1 vs2*,
$(v \text{ “in } (vs1 :\&: vs2)\%\mathrm{VS}) = (v \text{ “in } vs1) \mathbin{\&\&} (v \text{ “in } vs2)$.

Lemma vspace_modl : $\forall$ *vs1 vs2 vs3*,
$(vs1 \leq vs3 \to vs1 + (vs2 :\&: vs3) = (vs1 + vs2) :\&: vs3)\%\mathrm{VS}$.

Lemma vspace_modr : $\forall$ *vs1 vs2 vs3*,
$(vs3 \leq vs1 \to (vs1 :\&: vs2) + vs3 = vs1 :\&: (vs2 + vs3))\%\mathrm{VS}$.

Section BigCap.
Variable I : finType.
Implicit Type P : pred I.

Lemma bigcapv_inf : $\forall$ *i0* P $(vs\_ : I \to \{vspace\ V\})$ *vs*,
$P\ i0 \to (vs\_\ i0 \leq vs \to \text{“bigcap\_(i} \mid P\ i)\ vs\_\ i \leq vs)\%\mathrm{VS}$.

Lemma subv_bigcapP : $\forall$ P *vs* $(vs\_ : I \to \{vspace\ V\})$,
reflect $(\forall\ i,\ P\ i \to vs \leq vs\_\ i)\%\mathrm{VS}$
$(vs \leq \text{“bigcap\_(i} \mid P\ i)\ vs\_\ i)\%\mathrm{VS}$.

End BigCap.

Lemma addv_complf : $\forall$ *vs*, $(vs + vs\hat{}\mathrm{C} = \mathsf{fullv})\%\mathrm{VS}$.

Lemma capv_compl : $\forall$ *vs*, $(vs :\&: vs\hat{}\mathrm{C})\%\mathrm{VS} = 0\%\mathrm{:VS}$.

Lemma diffvSl : $\forall$ *vs1 vs2*, $(vs1 :\text{“}: vs2 \leq vs1)\%\mathrm{VS}$.

Lemma capv_diff : $\forall$ *vs1 vs2*, $((vs1 :\text{“}: vs2) :\&: vs2 = 0\%\mathrm{:VS})\%\mathrm{VS}$.

Lemma addv_diff_cap_eq : $\forall$ *vs1 vs2*, $(vs1 :\text{“}: vs2 + vs1 :\&: vs2 = vs1)\%\mathrm{VS}$.

Lemma dimv0 : $\text{“dim } (0\ \%\mathrm{:VS}) = 0\%\mathrm{N}$.

Lemma dimv_eq0 : $\forall$ *vs*, $(\text{“dim } vs == 0\%\mathrm{N}) = (vs == 0\%\mathrm{:VS})$.

Lemma dimvf : $\text{“dim } \mathsf{fullv} = \mathsf{vdim}\ V$.

Lemma dim_injv : $\forall$ *v*, $\text{“dim } (v\ \%\mathrm{:VS}) = (v\ != 0)$.

Lemma dimvS : $\forall$ *vs1 vs2*, $(vs1 \leq vs2)\%\mathrm{VS} \to \text{“dim } vs1 \leq \text{“dim } vs2$.

Lemma dimv_leqif_sup : $\forall$ *vs1 vs2*,
$(vs1 \leq vs2)\%\mathrm{VS} \to \text{“dim } vs1 \leq \text{“dim } vs2\ ?= \mathit{iff}\ (vs2 \leq vs1)\%\mathrm{VS}$.

Lemma dimv_leqif_eq : $\forall$ *vs1 vs2*,
$(vs1 \leq vs2)\%\mathrm{VS} \to \text{“dim } vs1 \leq \text{“dim } vs2\ ?= \mathit{iff}\ (vs1 == vs2)\%\mathrm{VS}$.

```
Lemma dimv_compl : ∀ vs, “dim vs^C = (n - “dim vs)%N.

Lemma dimv_disjoint_sum : ∀ vs1 vs2,
  (vs1 :&: vs2)%VS = 0%:VS → “dim (vs1 + vs2)%VS = (“dim vs1 + “dim vs2)%N.

Lemma dimv_cap_compl : ∀ vs1 vs2,
  (“dim (vs1 :&: vs2) + “dim (vs1 :“: vs2))%N = “dim vs1.

Lemma dimv_sum_cap : ∀ vs1 vs2,
  (“dim (vs1 + vs2) + “dim (vs1 :&: vs2) = “dim vs1 + “dim vs2)%N.

Lemma dimv_sum_leqif : ∀ vs1 vs2,
  “dim (vs1 + vs2) ≤ “dim vs1 + “dim vs2 ?= iff (vs1 :&: vs2 ≤ 0 %:VS)%VS.

Lemma subv_diffv0 : ∀ vs1 vs2,
  ((vs1 ≤ vs2) = (vs1 :“: vs2 == 0%:VS))%VS.

Section BigDim.
Variable I : finType.
Implicit Type P : pred I.

Lemma dimv_leq_sum : ∀ P (vs_ : I → {vspace V}),
  “dim (“sum_(i | P i) vs_ i)%VS ≤ (“sum_(i | P i) “dim (vs_ i))%N.

End BigDim.

Lemma directvP : ∀ (S : proper_addv_expr),
  reflect (“dim S = proper_addv_dim S) (directv S).
Lemma directv_trivial : ∀ vs, directv (unwrap (@trivial_addv vs)).

Lemma dimv_sumw_leqif : ∀ (S : addv_expr),
  “dim (unwrap S) ≤ unwrap (addv_dim S) ?= iff directv (unwrap S).

Lemma directvE : ∀ (S : addv_expr),
  directv (unwrap S) = (“dim (unwrap S) == unwrap (addv_dim S)).

Lemma directvEgeq : ∀ (S : addv_expr),
  directv (unwrap S) = (“dim (unwrap S) ≥ unwrap (addv_dim S)).

Section BinaryDirect.

Lemma directv_addE : ∀ (S1 S2 : addv_expr),
    directv (unwrap S1 + unwrap S2)
     = [&& directv (unwrap S1), directv (unwrap S2)
         & unwrap S1 :&: unwrap S2 == 0%:VS]%VS.

Lemma directv_addP vs1 vs2 :
    reflect (vs1 :&: vs2 = 0%:VS)%VS (directv (vs1 + vs2)).

Lemma directv_add_unique : ∀ vs1 vs2,
    directv (vs1 + vs2) ↔
    (∀ u1 v1 u2 v2, u1 “in vs1 → v1 “in vs1 →
                    u2 “in vs2 → v2 “in vs2 →
                    u1 + u2 == v1 + v2 = ((u1,u2) == (v1,v2))).

End BinaryDirect.
```

```
Section NaryDirect.

Variables (I : finType) (P : pred I).

Lemma directv_sumP : ∀ vs_ : I → {vspace V},
  reflect (∀ i,
             P i → vs_ i :&: (“sum_(j | P j && (j != i)) vs_ j) = 0%:VS)%VS
          (directv (“sum_(i | P i) vs_ i)).

Lemma directv_sumE : ∀ (S_ : I → addv_expr) (xunwrap := unwrap),
  reflect (and (∀ i, P i → directv (unwrap (S_ i)))
               (directv (“sum_(i | P i) (xunwrap (S_ i)))))
          (directv (“sum_(i | P i) (unwrap (S_ i)))).

Lemma directv_sum_unique : ∀ vs_ : I → {vspace V},
    directv (“sum_(i | P i) vs_ i) ↔
    (∀ (u_ v_ : I → V), (∀ i : I, P i → u_ i “in vs_ i) →
                                  (∀ i : I, P i → v_ i “in vs_ i) →
                                  (“sum_(i | P i) u_ i == “sum_(i | P i) v_ i) =
                                  (forallb i : I, P i ==¿ (u_ i == v_ i))).

End NaryDirect.

Lemma span_nil : span [::] = 0%:VS.

Lemma span_seq1: ∀ v, span [::v] = v%:VS.

Lemma span_cons : ∀ v l, span (v::l) = (v%:VS + span l)%VS.

Lemma span_cat : ∀ l1 l2, span (l1 ++ l2) = (span l1 + span l2)%VS.

Lemma memv_span : ∀ l v, v “in l → v “in span l.

Lemma span_subset : ∀ (l1 l2 : seq V),
  {subset l1 ≤ l2} → (span l1 ≤ span l2)%VS.

Lemma span_eq : ∀ (l1 l2 : seq V), l1 =i l2 → span l1 = span l2.

Lemma dim_span : ∀ l, “dim (span l) ≤ size l.

Lemma coord_is_linear : ∀ l, linear (coord l).
Lemma coord_sumE : ∀ l I r (P : pred I) (F : I → V) i,
 coord l (“sum_(j ← r | P j) F j) i = “sum_(j ← r | P j) coord l (F j) i.

Lemma coord_span :
  ∀ l v, v “in span l → v = “sum_(i ¡ size l) coord l v i *: l‘_i.

Lemma span_subsetl : ∀ l vs,
  all (fun v ⇒ v “in vs) l = (span l ≤ vs)%VS.

Lemma free_span_mx : ∀ l, free l = row_free (span_mx l).

Lemma free_nil : free [::].

Lemma free_seq1 : ∀ v, v != 0 → free [::v].

Lemma free_perm_eq : ∀ (l1 l2 : seq V),
  perm_eq l1 l2 → free l1 = free l2.
```

Lemma free_notin0 : ∀ v l, free l → v “in l → v != 0.

Lemma *freeP* : ∀ l,
  reflect
    (∀ s, “sum_(i ¡ *size* l) s i *: l‘_i = 0 → s =1 `fun _ ⇒ 0`)
     (*free* l).

Lemma *free_coord* : ∀ l i,
  free l → *coord* l (l‘_i) = [*ffun* j : 'I_(*size* l) ⇒ (i == j)%:R].

Lemma *free_coordE* : ∀ l i j, *free* l → *coord* l (l‘_i) j = (i == j)%:R.

Lemma *free_catl* : ∀ *l2* *l1*, *free* (*l1* ++ *l2*) → *free l1*.

Lemma *free_catr* : ∀ *l1* *l2*, *free* (*l1* ++ *l2*) → *free l2*.

Lemma *free_filter* : ∀ P (l: *seq* V),
  free l → *free* (*filter* P l).

Lemma *addv_free* : ∀ *l1* *l2*,
  directv (*span l1* + *span l2*) → *free* (*l1* ++ *l2*) = *free l1* && *free l2*.

Lemma *is_span_nil* : *is_span* 0%:VS [::].

Lemma *is_span_seq1* : ∀ v, v != 0 → *is_span* v%:VS [::v].

Lemma *memv_is_span* : ∀ v *vs* l, *is_span vs* l → v “in l → v “in *vs*.

Lemma *is_span_span* : ∀ *vs* l v,
  is_span vs l → v “in *vs* → v = “sum_i *coord* l v i *: l‘_i.

Lemma *addv_is_span* : ∀ *vs1* *vs2* *l1* *l2*,
  is_span vs1 *l1* → *is_span vs2* *l2* → *is_span* (*vs1* + *vs2*)%VS (*l1* ++ *l2*).

Lemma *is_basis_span* : ∀ *vs* l, *is_basis vs* l → *is_span vs* l.

Lemma *is_basis_free* : ∀ *vs* l, *is_basis vs* l → *free* l.

Lemma *is_basis_nil* : *is_basis* 0%:VS [::].

Lemma *is_basis_seq1* : ∀ v, v != 0 → *is_basis* v%:VS [::v].

Lemma *is_basis_notin0* : ∀ v *vs* l, *is_basis vs* l → v “in l → v != 0.

Lemma *memv_is_basis* : ∀ v *vs* l, *is_basis vs* l → v “in l → v “in *vs*.

Lemma *addv_is_basis* : ∀ *vs1* *vs2* *l1* *l2*,
  directv (*vs1* + *vs2*) → *is_basis vs1* *l1* → *is_basis vs2* *l2* →
  is_basis (*vs1* + *vs2*)%VS (*l1* ++ *l2*).

Lemma *size_is_basis* : ∀ *vs* l, *is_basis vs* l → *size* l = “dim *vs*.

Lemma *vbasis0* : *vbasis* 0%:VS = [::].

Lemma *is_basis_vbasis* : ∀ *vs*, *is_basis vs* (*vbasis vs*).

Lemma *memv_basis* : ∀ v *vs*, v “in (*vbasis vs*) → v “in *vs*.

Lemma *coord_basis* : ∀ v *vs*, v “in *vs* →
  v = “sum_(i ¡ *size* (*vbasis vs*)) *coord* (*vbasis vs*) v i *: (*vbasis vs*)‘_i.

```
Lemma size_basis : ∀ vs, size (vbasis vs) = “dim vs.

Section BigSumBasis.

Variable I : finType.
Implicit Type P : pred I.

Lemma span_bigcat : ∀ P (l_ : I → seq V),
  span (“big[cat/[::]]_(i | P i) l_ i) = (“sum_(i | P i) span (l_ i))%VS.

Lemma bigaddv_free : ∀ P (l_ : I → seq V),
  (directv (“sum_(i | P i) span (l_ i)))%VS →
  (∀ i, P i → free (l_ i)) → free (“big[cat/[::]]_(i | P i) l_ i).

Lemma bigaddv_is_span : ∀ P l_ (vs_ : I → {vspace V}),
  (∀ i, P i → is_span (vs_ i) (l_ i)) →
    is_span (“sum_(i | P i) vs_ i)%VS (“big[cat/[::]]_(i | P i) l_ i).

Lemma bigaddv_is_basis : ∀ P l_ (vs_ : I → {vspace V}),
  let vs := (“sum_(i | P i) vs_ i)%VS in
  directv vs →
  (∀ i, P i → is_basis (vs_ i) (l_ i)) →
    is_basis vs (“big[cat/[::]]_(i | P i) l_ i).

End BigSumBasis.

End VSpaceDef.

Section LinearApp.

Variable (R : ringType) (V W : vectType R).

Lemma eq_lapp : ∀ (f g : linearApp), reflect (f =1 g) (f == g).

Lemma lapp_addA : associative add_lapp.

Lemma lapp_addC : commutative add_lapp.

Lemma lapp_add0 : left_id zero_lapp add_lapp.

Lemma lapp_addN : left_inverse zero_lapp opp_lapp add_lapp.

Lemma zero_lappE : ∀ x, (“0 x = 0)%VS.

Lemma add_lappE : ∀ f g x, (f “+ g)%VS x = f x + g x.

Lemma opp_lappE : ∀ f x, (opp_lapp f) x = - f x.

Lemma sum_lappE : ∀ I (r : seq I) (P : pred I) (F : I → linearApp) x,
  (“big[+%R/0]_(i ← r | P i) F i) x = “big[+%R/0]_(i ← r | P i) (F i x).

Lemma lapp_scaleA : ∀ k1 k2 f, (k1 “*: (k2 “*: f) = (k1 × k2) “*: f)%VS.
Lemma lapp_scale1 : ∀ f, (1 “*: f = f)%VS.

Lemma lapp_addr : ∀ k f1 f2,
  (k “*: (f1 “+ f2) = (k “*: f1) “+ (k “*: f2))%VS.

Lemma lapp_addl : ∀ f k1 k2,
  ((k1 + k2) “*: f = (k1 “*: f) “+ (k2 “*: f))%VS.
```

```
Variables (phV : phant V) (phW : phant W).

Lemma hom_is_linear : ∀ (f : linearVect phV phW), linear f.
Lemma fun_of_lappK : cancel fun_of_lapp lapp_of_fun.

Lemma lapp_of_funK : ∀ (f : V → W), linear f → lapp_of_fun f =1 f.

End LinearApp.
Section ScaleLapp.

Variable (R : comRingType) (V W : vectType R).
Variable f : linearApp V W.

Lemma scale_lappE : ∀ a x, ((a “*: f) x)%VS = a *: f x.

End ScaleLapp.

Section UnitApp.

Variable (R : ringType) (V : vectType R).
Hypothesis vdim_nz : (vdim V != 0)%N.
Lemma unit_lappE : ∀ x, (“1 x = x)%VS.

Lemma unit_nonzero_lapp : (“1 != “0)%VS.

End UnitApp.
Section CompLinearApp.

Variable (R : ringType) (V W Z : vectType R).
Implicit Type f : linearApp W Z.
Implicit Type g : linearApp V W.

Lemma comp_lappE : ∀ f g, (f “o g)%VS =1 f “o g.

End CompLinearApp.

Section InvLinearApp.

Variable (K : fieldType) (V W : vectType K).
Implicit Type f : linearApp V W.

Lemma inv_lapp_def : ∀ f, (f “o (f “^-1 “o f) = f)%VS.

End InvLinearApp.
Section LAlgLinearApp.

Variable (R : ringType) (V W Z T : vectType R).
Implicit Type h : linearApp V W.
Implicit Type g : linearApp W Z.
Implicit Type f : linearApp Z T.

Lemma comp_lappA : ∀ f g h, (f “o (g “o h) = (f “o g) “o h)%VS.

Lemma comp_lapp_addl : ∀ f1 f2 g,
  ((f1 “+ f2) “o g = (f1 “o g) “+ (f2 “o g))%VS.

Lemma comp_lapp_addr : ∀ f g1 g2,
  (f “o (g1 “+ g2) = (f “o g1) “+ (f “o g2))%VS.
```

Lemma *comp_1lapp* : ∀ *f*, (1 ∘ *f* = *f*)%VS.

Lemma *comp_lapp1* : ∀ *f*, (*f* ∘ 1)%VS = *f*.

Lemma *scale_lapp_Ar* : ∀ *k f g*, (*k* ·: (*f* ∘ *g*) = *f* ∘ (*k* ·: *g*))%VS.

End *LAlgLinearApp*.

Section *AlgLinearApp*.

Variable (*R* : *comRingType*) (*V W Z* : *vectType R*).
Implicit Type *g* : *linearApp V W*.
Implicit Type *f* : *linearApp W Z*.

Lemma *scale_lapp_Al* : ∀ *k f g*, (*k* ·: (*f* ∘ *g*) = (*k* ·: *f*) ∘ *g*)%VS.

End *AlgLinearApp*.

Section *LinearImage*.

Variable (*K* : *fieldType*) (*V W* : *vectType K*).

Implicit Type *f* : *'Hom*(*V*,W).
Implicit Type *vs* : {*vspace V*}.

Lemma *lkerE* : ∀ *f vs*, (*f* @: *vs* == 0%:VS)%VS = (*vs* ≤ *lker f*)%VS.

Lemma *limgE* : ∀ *f*, *limg f* = (*f* @: (*fullv V*))%VS.

Lemma *limg_monotone* : ∀ *f vs1 vs2*,
(*vs1* ≤ *vs2*)%VS → (*f* @: *vs1* ≤ *f* @: *vs2*)%VS.

Lemma *limg_inj* : ∀ *f v*, (*f* @: (*v* %:VS))%VS = (*f v*)%:VS.

Lemma *lpre_img0* : ∀ *f*, (*f* @^-1: 0%:VS)%VS = *lker f*.

Lemma *lpre_img_full* : ∀ *f* (*vs* : {*vspace W*}),
(*f* @^-1: (*vs* :&: *limg f*))%VS = (*f* @^-1: *vs*)%VS.

Lemma *lpre_imgK* : ∀ *f* (*vs* : {*vspace W*}),
(*vs* ≤ *limg f*)%VS → (*f* @: (*f* @^-1: *vs*))%VS = *vs*.

Lemma *limg0* : ∀ *f*, (*f* @: (0 %:VS))%VS = 0%:VS.

Lemma *lim0g* : ∀ *vs*, (0 @: *vs*)%VS = 0%:VS.

Lemma *memv_img* : ∀ *f v vs*, *v* ∈ *vs* → *f v* ∈ (*f* @: *vs*)%VS.

Lemma *memv_ker* : ∀ *f v*, (*v* ∈ *lker f*)%VS = (*f v* == 0).

Lemma *limg_add* : ∀ *f*, {*morph* (*fun_of_limg f*) : *u v* / (*u* + *v*)%VS}.

Lemma *limg_sum* : ∀ *f* (*I* : *finType*) (*P* : *pred I*) (*vs_* : *I* → {*vspace V*}),
(*f* @: (∑_(i | *P i*) *vs_ i*) = ∑_(i | *P i*) (*f* @: (*vs_ i*)))%VS.

Lemma *limg_cap* : ∀ *f vs1 vs2*,
(*f* @: (*vs1* :&: *vs2*)) ≤ *f* @: *vs1* :&: *f* @: *vs2*)%VS.

Lemma *limg_bigcap* :
∀ *f* (*I* : *finType*) (*P* : *pred I*) (*vs_* : *I* → {*vspace V*}),
(*f* @: (⋂_(i | *P i*) *vs_ i*) ≤ ⋂_(i | *P i*) (*f* @: (*vs_ i*)))%VS.

Lemma *limg_span* : ∀ *f l*, (*f* @: *span l*)%VS = *span* (*map f l*).

Lemma *memv_imgP* : ∀ *f v vs*,
reflect (∃ *v1*, *v1* “in *vs* ∧ *v* = *f v1*) (*v* “in *f* @: *vs*)%VS.

Lemma *limg_ker_compl* : ∀ *f vs*, (*f* @: (*vs* :“: *lker f*))%VS = (*f* @: *vs*)%VS.

Lemma *limg_dim_eq* : ∀ *f vs*,
(*vs* :&: *lker f*)%VS = 0%:VS → “dim (*f* @: *vs*) = “dim *vs*.

Lemma *limg_is_basis* : ∀ *f vs l*, (*vs* :&: *lker f*)%VS = 0%:VS →
(*is_basis vs l* → *is_basis* (*f* @: *vs*)%VS (*map f l*)).

Lemma *limg_ker_dim* : ∀ *f vs*,
(“dim (*vs* :&: *lker f*) + “dim (*f* @: *vs*) = “dim *vs*)%N.

Lemma *lker0P* : ∀ *f*, *reflect* (*injective f*) (*lker f* == 0%:VS).

Lemma *limg_ker0* : ∀ *f vs ws*,
lker f == 0%:VS → (*f* @: *vs* ≤ *f* @: *ws*)%VS = (*vs* ≤ *ws*)%VS.

Lemma *eq_limg_ker0* : ∀ *f vs ws*,
lker f == 0%:VS → (*f* @: *vs* == *f* @: *ws*)%VS = (*vs* == *ws*)%VS.

End *LinearImage*.

Section *UnitImage*.

Variable (*K* : *fieldType*) (*V* : *vectType K*).

Lemma *lim1g* : ∀ *vs*, ((“1 : ’End(*V*)) @: *vs* = *vs*)%VS.

End *UnitImage*.

Section *CompImage*.

Variable (*K* : *fieldType*) (*V W Z* : *vectType K*).

Lemma *limg_comp* : ∀ (*f* : ’*Hom*(*V*,W)) (*g* : ’*Hom*(*W*,Z)) *vs*,
((*g* “o *f*) @: *vs* = *g* @: (*f* @: *vs*))%VS.

End *CompImage*.

Section *LinearPreImage*.

Variable (*K* : *fieldType*) (*V W* : *vectType K*).

Implicit Type *f* : ’*Hom*(*V*,W).
Implicit Type *vs* : {*vspace V*}.

Lemma *lpre_imgE* : ∀ *f* (*vs* : {*vspace W*}),
(*f* @^-1: *vs*)%VS = (((*f* “^-1) @: (*vs* :&: *limg f*)%VS) + *lker f*)%VS.

Lemma *lpre_img_monotone* : ∀ *f* (*vs1 vs2* : {*vspace W*}),
(*vs1* ≤ *vs2*)%VS-¿ (*f* @^-1: *vs1* ≤ *f* @^-1: *vs2*)%VS.

Lemma *memv_pre_img* : ∀ *f v* (*vs* : {*vspace W*}), (*f v* “in *vs*) = (*v* “in *f* @^-1: *vs*)%VS.

Lemma *inv_lker0* : ∀ *f*, *lker f* == 0%:VS → (*f* “^-1 “o *f* = “1)%VS.

End *LinearPreImage*.

```
Section Projection.

Variable (K : fieldType) (V : vectType K).
Implicit Types vs : {vspace V}.

Lemma projv_id : ∀ vs v, v “in vs → projv vs v = v.

Lemma lker_proj : ∀ vs, lker (projv vs) = (vs^C)%VS.

Lemma limg_proj : ∀ vs, limg (projv vs) = vs.

Lemma memv_proj : ∀ vs v, projv vs v “in vs.

Lemma memv_projC : ∀ vs v, v - projv vs v “in (vs^C)%VS.

Lemma memv_pi1 : ∀ vs1 vs2 v, (addv_pi1 vs1 vs2) v “in vs1.

Lemma memv_pi2 : ∀ vs1 vs2 v, (addv_pi2 vs1 vs2) v “in vs2.

Lemma addv_pi1_pi2 : ∀ vs1 vs2 v, v “in (vs1 + vs2)%VS →
  v = ((addv_pi1 vs1 vs2) + (addv_pi2 vs1 vs2)) v.

Section Sumv_Pi.

Variable I : finType.
Variable V_ : I → {vspace V}.
Variable P : pred I.

Lemma memv_sum_pi : ∀ i v, sumv_pi i v “in V_ i.

Lemma sumv_sum_pi : ∀ v, v “in (“sum_(i | P i) V_ i)%VS →
  v = (“sum_(i | P i) sumv_pi i) v.

End Sumv_Pi.

End Projection.

Section SubVectorType.

Variable (K : fieldType) (V : vectType K) (vs : {vspace V}).

Lemma subvP : ∀ u, sv_val u “in vs.
Lemma subvect_inj : injective sv_val.
Lemma congr_subv : ∀ u v, u = v → sv_val u = sv_val v.

Lemma subvect_addA : associative subvect_add.
Lemma subvect_addC : commutative subvect_add.
Lemma subvect_add0 : left_id subvect_zero subvect_add.
Lemma subvect_addN : left_inverse subvect_zero subvect_opp subvect_add.

Lemma subvect_scaleA : ∀ k1 k2 u,
   subvect_scale k1 (subvect_scale k2 u) = subvect_scale (k1 × k2) u.
Lemma subvect_scale1 : left_id 1 subvect_scale.
Lemma subvect_scale_addr : ∀ k, {morph (subvect_scale k) : x y / x + y}.
Lemma subvect_scale_addl : ∀ u,
   {morph (subvect_scale)^~ u : k1 k2 / k1 + k2}.

Lemma subvect_is_linear : linear subvect_v2rv.
```

```
Lemma subvect_bij : bijective subvect_v2rv.

End SubVectorType.

Section ProdVector.

Variable (K : fieldType) (V W : vectType K).

Lemma p2pvK : cancel p2pv pv2p.

Lemma pv2pK : cancel pv2p p2pv.

Lemma pvaddE : ∀ p q : prodVector,
  pv2p (p + q) = ((pv2p p).1+(pv2p q).1,(pv2p p).2+(pv2p q).2).

Lemma pvoppE : ∀ p : prodVector,
  pv2p (-p) = (-(pv2p p).1,-(pv2p p).2).

Lemma pvscaleE : ∀ (a: K) (p : prodVector),
  pv2p (a *: p) = (a *: (pv2p p).1, a *: (pv2p p).2).

Lemma pvfK : ∀ (f : 'End(V)) (g : 'End(W)),
    pvf f g =1 (fun x : prodVector ⇒ p2pv (f (pv2p x).1 , g (pv2p x).2)).

End ProdVector.

Section ExpVector.

Variable (K : fieldType) (V : vectType K).

Lemma l2ev_cons : ∀ x l, l2ev (x :: l) = p2pv (x, (l2ev l)).

Lemma tuple_of_evK : ∀ n, cancel (@tuple_of_ev n) (@ev_of_tuple n).

Lemma ev_of_tupleK : ∀ n, cancel (@ev_of_tuple n) (@tuple_of_ev n).

Lemma evfK : ∀ n (ft : n.-tuple (V → V)),
  (∀ i, linear (tnth ft i)) →
  evf ft =1 (fun p ⇒ ev_of_tuple (map_tuple (fun x ⇒ x p) ft)).

End ExpVector.

Section Solver.

Variable (K : fieldType) (V : vectType K).

Variable n : nat.
Variable feq : n.-tuple (V → V).
Variable veq : n.-tuple V.
Variable feq_linear : ∀ i, linear (tnth feq i).

Lemma vsolve_eqP : ∀ (ms : {vspace V}),
  reflect
      (∃ u, u "in ms ∧
          ∀ i : 'I_n, tnth feq i u = tnth veq i)
      (vsolve_eq ms).

End Solver.

Export VectorType.Exports.
```