Vector-Spaces

Jose Divasón Mallagaray

October 26, 2011

Contents

theory *Previous*
imports *Main*
begin

1 Previous general results

We present here some result and theorems which will be used in our developement. There are general properties, not centered in any section of our implementation.

lemma *less-than-Suc-union*:
  shows $\{i.\ i < Suc\ (n::nat)\} = \{i.\ i < n\} \cup \{n\}$
  unfolding *lessThan-def* [*symmetric*]
  unfolding *lessThan-Suc-atMost*
  using *atMost-Suc* **by** (*cases* n, *auto*)

Next two lemmas is a non-elegant trick which makes possible work with premisses that contains multiples *op* $\wedge$

lemma *conjI3*: $[\![A;\ B;\ C]\!] \Longrightarrow A \wedge B \wedge C$
  by *fast*

lemma *conjI4*: $[\![A;\ B;\ C;\ D]\!] \Longrightarrow A \wedge B \wedge C \wedge D$
  by *fast*

lemma *conjI5*: $[\![A;\ B;\ C;\ D;\ E]\!] \Longrightarrow A \wedge B \wedge C \wedge D \wedge E$
  by *fast*

lemma *conjI6*:

shows $[\![A;\ B;\ C;\ D;\ E;\ F]\!] \Longrightarrow A \wedge B \wedge C \wedge D \wedge E \wedge F$
by *fast*

Next lemmas prove some properties of the bijections between subsets of a given set.

lemma *bij-betw-subset*:
assumes *b*: *bij-betw f A B* **and** *sb*: $C \subseteq A$
shows *bij-betw f C* (f ‘ C)
using *b sb*
unfolding *bij-betw-def*
unfolding *image-def inj-on-def* **by** *auto*

lemma
bij-betw-image-minus:
assumes *b*: *bij-betw f A B* **and** *a*: $a \in A$
shows f ‘ $(A - \{a\}) = B - \{f\ a\}$
proof
show f ‘ $(A - \{a\}) \subseteq B - \{f\ a\}$
using *b*
unfolding *bij-betw-def*
using *a* **unfolding** *image-def* **unfolding** *inj-on-def* **by** *auto*
show $B - \{f\ a\} \subseteq f$ ‘ $(A - \{a\})$
using *b*
unfolding *bij-betw-def*
using *a* **unfolding** *image-def* **unfolding** *inj-on-def* **by** *auto*
qed

end
theory *Field2*
imports *Previous*
~~/src/HOL/Algebra/Ring
begin

2 Previous relations between algebraic structures.

We can create a lemma to check if one algebraic structure is a domain.

lemma *domainI*:
fixes *R* (**structure**)
assumes *cring*: *cring R*
and *one-not-zero*: $\mathbf{1}$ ~= $\mathbf{0}$
and *integral*: $\bigwedge a\ b.$ [| $a \otimes b = \mathbf{0}$; $a \in$ *carrier R*; $b \in$ *carrier R* |] ==> $a =$ $\mathbf{0}$ | $b = \mathbf{0}$
shows *domain R*
unfolding *domain-def*
apply (*rule conjI*)
using *cring* **apply** *fast*
unfolding *domain-axioms-def*
apply (*rule conjI*)

using *one-not-zero* **apply** *fast*
using *integral* **by** *fast*

Similarly with a field:

lemma *fieldI*:
fixes R (**structure**)
assumes *dom*: *domain R*
and *field-Units*: *Units R* = *carrier R* − {**0**}
shows *field R*
unfolding *field-def*
apply (*intro conjI*)
using *dom* **apply** *fast*
unfolding *field-axioms-def* **using** *field-Units* **.**

A field is an additive monoid

lemma (**in** *field*) *field-impl-monoid*:
monoid (| *carrier* = *carrier R*, *mult* = *add R*, *one* = *zero R* |)
using *abelian-monoid.a-monoid* [*of R*]
using *field-axioms*
unfolding *field-def*
unfolding *domain-def*
unfolding *cring-def*
unfolding *ring-def*
unfolding *abelian-group-def*
by *simp*

A field is a multiplicative monoid:

lemma *field-is-monoid*: **fixes** K (**structure**)
assumes *field-K*: *field K* **shows** *monoid K*
proof −
from *field-K* **show** *?thesis*
unfolding *field-def*
unfolding *domain-def*
unfolding *cring-def*
unfolding *ring-def*
by *best*
qed

Every field is a ring

lemma *field-is-ring*: **fixes** K (**structure**)
assumes *field-K*: *field K* **shows** *ring K*
proof −
from *field-K* **show** *?thesis*
unfolding *field-def*
unfolding *domain-def*
unfolding *cring-def*
by *best*
qed

2.1 Previous properties

First of all we are going to introduce some properties of fields. Most of them are also satisfied in rings and in previous algebraic structures, so they will be trivial for us.

This property is trivial and proved in the library:

lemma (**in** *field*) *r-zero*:
  $x \in carrier\ R \implies x \oplus \mathbf{0} = x$
 using *r-zero* [*of x*] **.**

However, we can make a long proof of the preceding fact.

lemma (**in** *field*) *r-zero2*: $x \in carrier\ R \implies x \oplus \mathbf{0} = x$
proof –
  assume *x-in-R*: $x \in carrier\ R$
  have *l-zero*: $\mathbf{0} \oplus x = x$
    — Using 'rule' we can give a lemma which goal is the same that the goal that we want to prove. Then, the 'rule' will convert my goal to the premisses of the theorem.
  proof (*rule abelian-monoid.l-zero* [*of R*])
    show *abelian-monoid R*
      print-facts
      using *field-axioms*
      unfolding *field-def*
      unfolding *domain-def*
      unfolding *cring-def*
      unfolding *ring-def*
      unfolding *abelian-group-def* **by** *fast*
  next
    — Using 'next' we closed the previous proof, so we would lose the local results of it. We open a new context for the second goal that we have. It is more or less than if we close a 'for' or a 'while' in C++ or Java: we will lose the local variables, but we will keep the global ones.
    show $x \in carrier\ R$
      using *x-in-R* **.**
  qed
  show *?thesis*
    find-theorems $?x \oplus ?y = ?y \oplus ?x$
    using *l-zero*
    using *a-comm* [*OF zero-closed x-in-R*] **by** *simp*
qed

This is also in the library (for commutative groups):

lemma (**in** *field*) *a-comm*:
  $!!\ x\ y.\ [\![x \in carrier\ R;\ y \in carrier\ R]\!] \Longrightarrow x \oplus y = y \oplus x$
  using *cring-simprules* (*10*) **.**

But we can prove it: we have that the property is satisfied in a commutative group. We will prove that a field is a commutative group and then we will use the property.

lemma (**in** *field*) *a-comm2*:
  !! $x\ y$. $[\![x \in \mathit{carrier}\ R;\ y \in \mathit{carrier}\ R]\!] \Longrightarrow x \oplus y = y \oplus x$
proof –
  fix $x\ y$
  assume *x-in-R*: $x \in \mathit{carrier}\ R$ **and** *y-in-R*: $y \in \mathit{carrier}\ R$
  — First we prove that the additive structure is a *comm-group*, the result is proved for *comm-monoid*.
  have *c-gr*: *comm-group* $(\!|\mathit{carrier} = \mathit{carrier}\ R,\ \mathit{mult} = \mathit{op}\ \oplus,\ \mathit{one} = \mathbf{0}|\!)$
  proof (*rule abelian-group.a-comm-group* [*of R*])
    show *abelian-group R*
      using *field-axioms*
      unfolding *field-def*
      unfolding *domain-def*
      unfolding *cring-def*
      unfolding *ring-def* **by** *fast*
  qed
  show $x \oplus y = y \oplus x$
    using *comm-monoid.m-comm* [*of* $(\!|\mathit{carrier} = \mathit{carrier}\ R,\ \mathit{mult} = \mathit{op}\ \oplus,\ \mathit{one} = \mathbf{0}|\!)$ $x\ y$]
    using *c-gr*
    unfolding *comm-group-def*
    using *x-in-R*
    using *y-in-R* **by** *simp*
qed

lemma (**in** *field*) *a-assoc*:
!! $x\ y\ z$. $[\![x \in \mathit{carrier}\ R;\ y \in \mathit{carrier}\ R;\ z \in \mathit{carrier}\ R]\!] \Longrightarrow (x \oplus y) \oplus z = x \oplus (y \oplus z)$
using *a-assoc* **.**

lemma (**in** *field*) *r-neg*:
$x \in \mathit{carrier}\ R \Longrightarrow x \oplus (\ominus\ x) = \mathbf{0}$
using *cring-simprules*(*17*) **.**

lemma (**in** *field*) *m-comm*:
!! $x\ y$. $[\![\ x \in \mathit{carrier}\ R;\ y \in \mathit{carrier}\ R]\!] \Longrightarrow x \otimes y = y \otimes x$
using *m-comm* **.**

lemma (**in** *field*) *m-assoc*:
!! $x\ y\ z$. $[\![x \in \mathit{carrier}\ R;\ y \in \mathit{carrier}\ R;\ z \in \mathit{carrier}\ R]\!] \Longrightarrow (x \otimes y) \otimes z = x \otimes (y \otimes z)$
using *m-assoc* **.**

lemma (**in** *field*) *r-one*:
$x \in \mathit{carrier}\ R \Longrightarrow x \otimes \mathbf{1} = x$
using *r-one* **.**

lemma (**in** *field*) *r-inv*:
$x\in$ *Units R* $\Longrightarrow x \otimes$ *inv* $x = \mathbf{1}$
using *Units-r-inv* **.**

lemma (**in** *field*) *r-distr*:
$[\![x\in$ *carrier R*; $y\in$ *carrier R*;$z\in$ *carrier R*$]\!] \Longrightarrow x \otimes (y \oplus z) = x \otimes y \oplus x \otimes z$
using *r-distr* **.**

lemma (**in** *field*) *l-one*:
$x \in$ *carrier R* $\Longrightarrow \mathbf{1} \otimes x{=}x$
  using *r-one*
  using *one-closed*
  using *m-comm* [*of x* **1**]
  by *simp*

2.2 Exercises in Halmos

Definition of field and some properties are already included in the library, so we don't make it.

Here we present some exercises proposed by Halmos. There are someone already solved in the library, so they will be trivial for us.

Exercise 1A

lemma (**in** *field*) *l-zero*:
  $x \in$ *carrier R* $\Longrightarrow \mathbf{0} \oplus x{=}x$
  using *r-zero* [*of x*]
  using *zero-closed*
  using *a-comm* [*of x* **0**] **by** *simp*

Exercise 1B

lemma (**in** *field*) *a-l-cancel*:
  $[\![x \in$ *carrier R*; $y\in$ *carrier R*;$z \in$ *carrier R*$]\!] \Longrightarrow (x \oplus y = x \oplus z) = (y = z)$
  using *a-l-cancel* **.**

Exercise 1C

lemma (**in** *field*) *plus-minus-cancel*:
  $[\![x \in$ *carrier R*; $y \in$ *carrier R*$]\!] \Longrightarrow x \oplus (y \ominus x) = y$
proof −
  assume *x-in-R*: $x \in$ *carrier R*
    and *y-in-R*: $y\in$ *carrier R*
  moreover have *minus-x-in-R*: $\ominus\ x \in$ *carrier R*
    using *a-inv-closed* [*OF x-in-R*] **.**
  have *prev-eq*: $(x \oplus \ominus\ x) \oplus y = y$
    using *x-in-R y-in-R*
    by (*simp add*: *r-neg l-zero*)
  show *?thesis*
    unfolding *minus-eq* [*OF y-in-R x-in-R*]

unfolding *a-comm* [*OF y-in-R minus-x-in-R*]
unfolding *a-assoc* [*symmetric*, *OF x-in-R minus-x-in-R y-in-R*]
using *prev-eq* **.**
qed

Corollary of 1C. It is in the library.

corollary (**in** *field*) *minus-eq*:
$[\![y \in carrier\ R;\ x \in carrier\ R]\!] \Longrightarrow y \ominus x = y \oplus (\ominus x)$
using *minus-eq* **by** *simp*

Exercise 1D

lemma (**in** *field*) *r-null*:
$x \in carrier\ R \Longrightarrow x \otimes \mathbf{0} = \mathbf{0}$
using *r-null* **.**

lemma (**in** *field*) *l-null*:
$x \in carrier\ R \Longrightarrow \mathbf{0} \otimes x = \mathbf{0}$
using *l-null* **.**

Exercise 1E

lemma (**in** *field*) *l-minus-one*:
$x \in carrier\ R \Longrightarrow (\ominus \mathbf{1}) \otimes x = \ominus x$
proof −
assume *x-in-R*: $x \in carrier\ R$
have $(\ominus \mathbf{1}) \otimes x = \ominus(\mathbf{1} \otimes x)$
using *l-minus*[*OF one-closed x-in-R*] **.**
also have ...$= \ominus x$ **using** *l-one*[*OF x-in-R*] **by** *presburger*
finally show *?thesis* **.**
qed

Exercise 1F

lemma (**in** *field*) *prod-minus*:
assumes *x-in-R*: $x \in carrier\ R$
and *y-in-R*: $y \in carrier\ R$
shows $(\ominus x) \otimes (\ominus y) = x \otimes y$
proof −
have *minus-x-in-R*: $\ominus\ x \in carrier\ R$
and *minus-y-in-R*: $\ominus\ y \in carrier\ R$
using *a-inv-closed* [*OF x-in-R*]
using *a-inv-closed* [*OF y-in-R*] **.**
show *?thesis*
unfolding *l-minus* [*OF x-in-R minus-y-in-R*]
unfolding *r-minus* [*OF x-in-R y-in-R*]
unfolding *minus-minus* [*OF m-closed* [*OF x-in-R y-in-R*]] **..**
qed

Exercise 1G

This exercise can be solved directly using integral property. However we will make it using $Units = carrier\ R - \{\mathbf{0}_R\}$. This is because field would not need to be derived from domain, the properties for domain follow from the assumptions of field (if we consider a field like a commutative ring in which $Units = carrier\ R - \{\mathbf{0}_R\}$

lemma (**in** *field*) *integral*:
  assumes *x-y-eq-0*: $x \otimes y = \mathbf{0}$
  and *x-in-R*: $x \in carrier\ R$
  and *y-in-R*: $y \in carrier\ R$
  shows $x = \mathbf{0} \mid y = \mathbf{0}$
proof ($cases\ x \neq \mathbf{0}$)

— We give as a parametrer to 'cases' a boolean (x not 0); this will make appear two cases: when the boolean is true (case True) and when the boolean is false (case False). For us, case False will be trivial.

  case *False* **show** *?thesis*
    using *False* — This is the negation of the boolean that I have written in 'cases'.

    by *fast* — Case False is trivial, it implies that x is zero and the lemma would be proved.

next

— We want to separate in cases and for that we must use next, if not in this case, we could apply the premise False in case True

  case *True* — Next case: case True
  note *x-neq-0* = *True*

— With this command we are assigning a pseudonym to True because we will separate in cases y not 0 and then we will meet with cases True and False, again.

  show *?thesis*
  proof ($cases\ y \neq \mathbf{0}$)
    case *False* **show** *?thesis* — Trivial case
      using *False* **by** *simp*
  next
    case *True*
    note *y-neq-0* = *True*

— Really here we will not need the pseudonym (we will not make more distinction between cases), but we will use it to clarify the premises and its names.

    show *?thesis*
    proof −
      have *y-un*: $y \in Units\ R$
        using *y-in-R*
        using *field-Units*
        using *y-neq-0* **by** *simp*
      have *inv-y-in-R*: $inv\ y \in carrier\ R$
        using *Units-inv-closed* [*OF y-un*] **.**

— Now we will begin with a 'calculation' in Isabelle. A calculation is a group of equalities which are linked amongst themselves. For that, we use the command 'also' and '...'

      have $\mathbf{0} = \mathbf{0} \otimes inv\ y$

— We can not use simp: left term of the equality is simpler than right one.

        using *l-null* [*symmetric*, *OF inv-y-in-R*] **.**

also have $\ldots = (x \otimes y) \otimes$ *inv* y
— Here we make use of the original premise of the lemma: x * y = 0
unfolding *x-y-eq-0* [*symmetric*] **..**
also have $\ldots = x \otimes (y \otimes$ *inv* $y)$
unfolding *m-assoc* [*OF x-in-R y-in-R inv-y-in-R*] **..**
also have $\ldots = x \otimes \mathbf{1}$
unfolding *Units-r-inv* [*OF y-un*] **..**
also have $\ldots = x$
unfolding *r-one* [*OF x-in-R*] **..**

— At the beginning of our 'calculation' we have started with 0, so we have proved that 0 = x (through some intermediate steps). To close a 'calculation' it is used the command 'finally' which makes equal the left term of the first 'have' before the 'also' with the right term of the last.

finally have $\mathbf{0} = x$ **.**
— Using 0 = x we can obtain a contradiction with our premises trivially.
then show *?thesis* **using** *x-neq-0* **by** *fast*
qed
qed
qed

end

theory *Vector-Space*
imports *Field2*
begin

3 Definition of Vector Space

Here the definition of a vector space using locales and inherit. We need to fix a field, an abelian group and the scalar product relating both structures (an abelian group together a field would be a vector space with one specific scalar product but not with another). A vector space is an algebraic structure composed of a field, an abelian monoid and a scalar product which satisfies some properties.

locale *vector-space* = K: *field* K + V: *abelian-group* V
for K (**structure**) **and** V (**structure**) +
fixes *scalar-product*:: $'a => 'b => 'b$ (**infixr** $\cdot$ *70*)
assumes *mult-closed*: $[\![x \in carrier\ V; a \in carrier\ K]\!]$
$\Longrightarrow a \cdot x \in carrier\ V$
and *mult-assoc*: $[\![x \in carrier\ V;\ a \in carrier\ K;\ b \in carrier\ K\ ]\!]$
$\Longrightarrow (a \otimes_K b) \cdot x = a \cdot (b \cdot x)$
and *mult-1*: $[\![x \in carrier\ V]\!] \Longrightarrow \mathbf{1}_K \cdot x = x$
and *add-mult-distrib1*:
$[\![x\in carrier\ V;\ y \in carrier\ V;\ a \in carrier\ K]\!]$
$\Longrightarrow a \cdot (x \oplus_V y)= a{\cdot}x \oplus_V a{\cdot}y$
and *add-mult-distrib2*:
$[\![x\in carrier\ V;\ a \in carrier\ K;\ b \in carrier\ K]\!]$

$\Longrightarrow (a \oplus_K b) \cdot x = a{\cdot}x \oplus_V b{\cdot}x$

Using this lemma we can check if an algebraic structure is a vector space

lemma *vector-spaceI*:
  fixes K (**structure**) **and** V (**structure**)
  and *scalar-product* $::'a => 'b => 'b$ (**infixr** $\cdot$ *70*)
  assumes *field-K*: *field K*
  and *abelian-group-V*: *abelian-group V*
  and *mult-closed*:
  $\bigwedge x\ a.\ [\![x \in carrier\ V; a \in carrier\ K]\!] \Longrightarrow a{\cdot}x \in carrier\ V$
  and *mult-assoc*:
  $\bigwedge x\ a\ b.\ [\![\ x \in carrier\ V;\ a \in carrier\ K;\ b \in carrier\ K\ ]\!]$
  $\Longrightarrow (a \otimes_K b){\cdot}\ x = a \cdot (b \cdot x)$
  and *mult-1*: $\bigwedge x.\ [\![x \in carrier\ V]\!] \Longrightarrow \mathbf{1}_K \cdot x = x$
  and *add-mult-distrib1*:
  $\bigwedge x\ y\ a.\ [\![x{\in}\ carrier\ V;\ y \in carrier\ V;\ a \in carrier\ K]\!]$
  $\Longrightarrow a{\cdot}(x \oplus_V y) = a{\cdot}x \oplus_V a{\cdot}\ y$
  and *add-mult-distrib2*:
  $\bigwedge x\ a\ b.\ [\![x{\in}\ carrier\ V;\ a \in carrier\ K;\ b \in carrier\ K]\!]$
  $\Longrightarrow (a \oplus_K b){\cdot}\ x = a{\cdot}x \oplus_V b{\cdot}x$
  shows *vector-space K V scalar-product*
proof (*unfold vector-space-def*, *intro conjI*)
  show *field K* **using** *field-K* **.**
  show *abelian-group V* **using** *abelian-group-V* **.**
next
  show *vector-space-axioms K V scalar-product*
    by (*auto intro*: *vector-space.intro abelian-group.intro*
    field.intro vector-space-axioms.intro assms)
qed

end

theory *Examples*
imports *Vector-Space RealDef*
begin

4 Examples

context *vector-space*
begin

Here we show that every field is a vector space over itself (we interpret the scalar product as the ordinary multiplication of the field. We use make use of *vector-spaceI*.

lemma *field-is-vector-space*:
  assumes *field-K*: *field K*

shows *vector-space K K op* $\otimes_K$
proof (*rule vector-spaceI*)
show *field K* **using** *field-K* **.**
show *abelian-group K* **using** *field-K*
unfolding *field-def*
unfolding *domain-def*
unfolding *cring-def*
unfolding *ring-def*
by *fast*
next
show $\bigwedge x\ a.\ [\![x \in carrier\ K;\ a \in carrier\ K]\!] \Longrightarrow a \otimes_K x \in carrier\ K$
using *monoid.m-closed* [*OF field-is-monoid* [*OF field-K*]] **by** *best*
next
show $\bigwedge x\ a\ b.\ [\![x \in carrier\ K;\ a \in carrier\ K;\ b \in carrier\ K]\!]$
$\Longrightarrow a \otimes_K b \otimes_K x = a \otimes_K (b \otimes_K x)$
using *monoid.m-assoc* [*OF field-is-monoid* [*OF field-K*]] **by** *best*
next
show $\bigwedge x.\ x \in carrier\ K \Longrightarrow \mathbf{1}_K \otimes_K x = x$
using *monoid.l-one* [*OF field-is-monoid* [*OF field-K*]] **by** *best*
next
show $\bigwedge x\ y\ a.\ [\![x \in carrier\ K;\ y \in carrier\ K;\ a \in carrier\ K]\!]$
$\Longrightarrow a \otimes_K (x \oplus_K y) = a \otimes_K x \oplus_K a \otimes_K y$
using *ring.r-distr* [*OF field-is-ring* [*OF field-K*]] **by** *best*
next
show $\bigwedge x\ a\ b.\ [\![x \in carrier\ K;\ a \in carrier\ K;\ b \in carrier\ K]\!]$
$\Longrightarrow (a \oplus_K b) \otimes_K x = a \otimes_K x \oplus_K b \otimes_K x$
proof −
fix x **and** a **and** b
assume *x-in-K*: $x \in carrier\ K$
and *a-in-K*: $a \in carrier\ K$ **and** *b-in-K*:$b \in carrier\ K$
show $(a \oplus_K b) \otimes_K x = a \otimes_K x \oplus_K b \otimes_K x$
using *ring.l-distr*
[*OF field-is-ring* [*OF field-K*] *a-in-K b-in-K x-in-K*] **.**
qed
qed (*auto*)

end
end
theory *Comments*
imports *Examples*
begin

5 Comments

context *vector-space*
begin

Now some properties of vector spaces.

Halmos proposes some exercises, but most of them are properties already proved in abelian groups, rings... so they are in the library and using the inheritance of properties provided by locales we obtain them for vector spaces. Lemmas in which the scalar product appears need to be proved and we make it here.

We have two zeros: $\mathbf{0}_V$ and $\mathbf{0}$. We need to define separately the closure property in order to avoid confusions. Alternatively, we could specify the structure writing *V.zero-closed* and *K.zero-closed*.

lemma *zeroV-closed*: $\mathbf{0}_V \in$ *carrier V*
using *V.zero-closed* **.**

lemma *zeroK-closed*: $\mathbf{0}_K \in$ *carrier K*
  using *K.zero-closed* **.**

A variation of *r-neg* ($x \in carrier\ V \Longrightarrow x \oplus_V \ominus_V x = \mathbf{0}_V$):

lemma *r-neg'*:
  assumes *x-in-V*: $x \in$ *carrier V*
  shows $x \ominus_V x = \mathbf{0}_V$
proof −
  have $\mathbf{0}_V = x \oplus_V \ominus_V x$
    using *V.r-neg* [*OF x-in-V*, *symmetric*] **.**
  also have $\ldots = x \ominus_V x$ **using** *a-minus-def* [*symmetric*, *OF x-in-V x-in-V*] **.**
  finally show *?thesis* **by** *simp*
qed

We want to prove that $a \cdot \mathbf{0}_V = \mathbf{0}_V$. First of all, we prove some auxiliary lemmas:

lemma *mult-zero-descomposition* [*simp*]:
  assumes *a-in-K*: $a \in$ *carrier K*
  shows $a \cdot \mathbf{0}_V \oplus_V a \cdot \mathbf{0}_V = a \cdot \mathbf{0}_V$
proof −
  have $a \cdot \mathbf{0}_V = a \cdot (\mathbf{0}_V \oplus_V \mathbf{0}_V)$
    using *V.r-zero* [*symmetric*, *OF V.zero-closed*] **by** *simp*
  also
  have $\ldots = a \cdot \mathbf{0}_V \oplus_V a \cdot \mathbf{0}_V$
    using *add-mult-distrib1* [*OF V.zero-closed V.zero-closed a-in-K*] **by** *simp*
  finally show *?thesis* **by** *rule*
qed

lemma *plus-minus-assoc*:
  assumes *x-in-V*: $x \in$ *carrier V*
  and *y-in-V*: $y \in$ *carrier V* **and** *z-in-V*: $z \in$ *carrier V*
  shows $x \oplus_V y \ominus_V z = x \oplus_V (y \ominus_V z)$
proof −
  have *minus-z-in-V*: $\ominus_V z \in$ *carrier V*
    using *V.a-inv-closed* [*OF z-in-V*] **.**
  have $x \oplus_V y \ominus_V z = x \oplus_V y \oplus_V \ominus_V z$

using *a-minus-def* [*of* $x \oplus_V y$, *OF* - *z-in-V*]
using *V.a-closed* [*OF x-in-V y-in-V*] **.**
also have $x \oplus_V y \oplus_V \ominus_V z = x \oplus_V (y \oplus_V \ominus_V z)$
using *V.a-assoc* [*OF x-in-V y-in-V minus-z-in-V*] **.**
also have $\ldots = x \oplus_V (y \ominus_V z)$
unfolding *a-minus-def* [*symmetric*, *OF y-in-V z-in-V*] **..**
finally show *?thesis* **by** *simp*
qed

Now we can complete theorem that we want to prove. It corresponds with exercise 1C in section 4 in Halmos.

lemma *scalar-mult-zeroV-is-zeroV*:
assumes *a-in-K*: $a \in$ *carrier K*
shows $a \cdot \mathbf{0}_V = \mathbf{0}_V$
proof −
have *mclosed*: $a \cdot \mathbf{0}_V \in$ *carrier V*
using *mult-closed* [*OF V.zero-closed a-in-K*] **.**
have $a \cdot \mathbf{0}_V = a \cdot \mathbf{0}_V \oplus_V a \cdot \mathbf{0}_V$
using *mult-zero-descomposition* [*OF a-in-K*] **by** *simp*
hence $a \cdot \mathbf{0}_V \ominus_V a \cdot \mathbf{0}_V = a \cdot \mathbf{0}_V \oplus_V a \cdot \mathbf{0}_V \ominus_V a \cdot \mathbf{0}_V$
using *mclosed* **by** *simp*
thus *?thesis*
unfolding *plus-minus-assoc* [*OF mclosed mclosed mclosed*]
unfolding *r-neg'* [*OF mclosed*]
using *V.r-zero* [*OF mclosed*] **by** *simp*
qed

We apply a similar reasoning to prove that $\mathbf{0} \cdot x = \mathbf{0}_V$ (this corresponds with exercise 1D in section 4 in Halmos):

lemma *mult-zero-descomposition2*:
assumes *x-in-V*: $x \in$ *carrier V*
shows $\mathbf{0}_K \cdot x \oplus_V \mathbf{0}_K \cdot x = \mathbf{0}_K \cdot x$
proof −
have $\mathbf{0}_K \cdot x = (\mathbf{0}_K \oplus_K \mathbf{0}_K) \cdot x$
using *zeroK-closed*
using *K.r-zero* [*OF zeroK-closed* ,*symmetric*] **by** *simp*
from *this* **show** *?thesis*
using *add-mult-distrib2* [*OF x-in-V zeroK-closed zeroK-closed*,*symmetric*]
by *simp*
qed

The exercise 1D in section 4 in Halmos is proved as follows:

lemma *zeroK-mult-V-is-zeroV*:
assumes *x-in-V*: $x \in$ *carrier V*
shows $\mathbf{0}_K \cdot x = \mathbf{0}_V$
proof −
have *mclosed*: $\mathbf{0}_K \cdot x \in$ *carrier V*
using *mult-closed* [*OF x-in-V zeroK-closed*] **.**

have $\mathbf{0}_K \cdot x = \mathbf{0}_K \cdot x \oplus_V \mathbf{0}_K \cdot x$
using *mult-zero-descomposition2* [*OF x-in-V*,*symmetric*] **.**
hence $\mathbf{0}_K \cdot x \ominus_V \mathbf{0}_K \cdot x = \mathbf{0}_K \cdot x \oplus_V \mathbf{0}_K \cdot x \ominus_V \mathbf{0}_K \cdot x$ **by** *simp*
thus *?thesis*
unfolding *plus-minus-assoc* [*OF mclosed mclosed mclosed*]
unfolding *r-neg'* [*OF mclosed*]
using *V.r-zero* [*OF mclosed*] **by** *simp*
qed

Another relevant property permit us to relate the additive inverse of the multiplicative unit with the additive inverse. It corresponds with exercise (1F) in section 4 in Halmos.

lemma *negate-eq*:
assumes *x-in-V*: $x \in$ *carrier* V
shows $(\ominus_K\ \mathbf{1}_K) \cdot x = \ominus_V x$
proof (*rule V.minus-equality* [*symmetric*, *of* $(\ominus_K\ \mathbf{1}_K) \cdot x\ x$])
show $x \in$ *carrier* V **using** *x-in-V* **.**
have *minus-oneK-closed*: $\ominus\ _K\ \mathbf{1}\ _K \in$ *carrier* K
using *K.a-inv-closed* [*OF K.one-closed*] **.**
show $\ominus\ \mathbf{1} \cdot x \in$ *carrier* V
using *mult-closed* [*OF x-in-V minus-oneK-closed*] **.**
show $\ominus\mathbf{1} \cdot x \oplus_V x = \mathbf{0}_V$
proof $-$
have $\mathbf{0}_V = \mathbf{0}_K \cdot x$
using *zeroK-mult-V-is-zeroV* [*symmetric*, *OF x-in-V*] **.**
also have $\ldots = (\ominus_K \mathbf{1}_K \oplus_K \mathbf{1}_K) \cdot x$
unfolding *K.l-neg* [*OF K.one-closed*] **..**
also have $\ldots = \ominus_K \mathbf{1}\ _K \cdot x \oplus_V \mathbf{1}\ _K \cdot x$
using *add-mult-distrib2* [*OF x-in-V minus-oneK-closed K.one-closed*] **.**
also have $\ldots = \ominus_K \mathbf{1}\ _K \cdot x \oplus_V x$
unfolding *mult-1* [*OF x-in-V*] **..**
finally show *?thesis* **by** *rule*
qed
qed

The previous property can be proved not only for the multiplicative unit of $\mathbb{K}$ but for every element in its carrier. We redo the demonstration (the previous lemma could be proved as a corollary of this):

lemma *negate-eq2*:
assumes *x-in-V*: $x \in$ *carrier* V
and *a-in-K*: $a \in$ *carrier* K
shows $(\ominus_K a) \cdot x = \ominus_V (a \cdot x)$
proof(*rule V.minus-equality* [*symmetric*, *of* $(\ominus_K\ a) \cdot x\ a \cdot x$])
show $a \cdot x \in$ *carrier* V **using** *mult-closed*[*OF x-in-V a-in-K*] **.**
show $\ominus\ a \cdot x \in$ *carrier* V
using *mult-closed* [*OF x-in-V K.a-inv-closed*[*OF a-in-K*]] **.**
show $\ominus a \cdot x \oplus_V a \cdot x = \mathbf{0}_V$
proof $-$
have $\mathbf{0}_V = \mathbf{0}_K \cdot x$

using *zeroK-mult-V-is-zeroV* [*symmetric*, *OF x-in-V*] **.**
also have ... = $(\ominus_K\ a \oplus_K\ a) \cdot\ x$
unfolding *K.l-neg* [*OF a-in-K*] **..**
also have ... = $\ominus_K\ a \cdot\ x \oplus_V\ a \cdot x$
using *add-mult-distrib2*
[*OF x-in-V K.a-inv-closed*[*OF a-in-K*] *a-in-K*] **.**
finally show *?thesis* **by** *rule*
qed
qed

The next two lemmas prove exercise 1E, which says that the scalar product also satisfies an integral property (if $a \cdot b = 0_V$, either $a = 0_{\mathbb{K}}$ or $b = 0_V$):

lemma *mult-zero-uniq*:
assumes *x-in-V*: $x \in$ *carrier V* **and** *x-not-zero*: $x \neq \mathbf{0}_V$
and *a-in-K*: $a \in$ *carrier K* **and** *m-ax-0*: $a \cdot x = \mathbf{0}_V$
shows $a = \mathbf{0}_K$
proof (*rule classical*)
assume *a-not-zero*: $a \neq \mathbf{0}_K$
have *a-un*: $a \in$ *Units K*
using *a-not-zero*
using *a-in-K*
using *K.field-Units* **by** *simp*
have *inv-a-in-K*: *inv a* $\in$ *carrier K*
using *K.Units-inv-closed* [*OF a-un*] **.**
have $x = (inv\ a \otimes a) \cdot x$
using *K.Units-l-inv* [*OF a-un*]
using *mult-1* [*OF x-in-V*]
by *simp*
also have ...= $inv\ a \cdot (a \cdot x)$
using *mult-assoc* [*OF x-in-V inv-a-in-K a-in-K*] **.**
also have ...= $inv\ a \cdot \mathbf{0}_V$ **using** *m-ax-0* **by** *simp*
also have ...= $\mathbf{0}_V$
using *scalar-mult-zeroV-is-zeroV* [*OF inv-a-in-K*] **.**
finally have $x = \mathbf{0}_V$ **.**
with *x-not-zero* **show** $a{=}\mathbf{0}_K$ **by** *contradiction*
qed

lemma *integral*:
assumes *x-in-V*: $x \in$ *carrier V*
and *a-in-K*: $a \in$ *carrier K*
and *m-ax-0*: $a \cdot x = \mathbf{0}_V$
shows $a = \mathbf{0}_K \mid x{=}\mathbf{0}_V$
proof (*cases* $x \neq \mathbf{0}_V$)
case *False* **show** *?thesis* **using** *False* **by** *simp*
next
case *True*
note *x-not-zero* = *True*
show *?thesis*

proof (*cases* $a \neq \mathbf{0}_K$)
  case *False* **show** *?thesis* **using** *False* **by** *simp*
next
  case *True*
  note *a-not-zero=True*
  show *?thesis*
    using *mult-zero-uniq* [*OF x-in-V x-not-zero a-in-K m-ax-0*]
    using *a-not-zero* **by** *contradiction*
qed
qed

We present here some other properties which don't appear in Halmos but that will be useful in our development. For instance, distributivity of substraction with respect to the scalar product:

lemma *diff-mult-distrib1*:
  assumes *x-in-V*: $x \in$ *carrier V*
  and *y-in-V*: $y \in$ *carrier V*
  and *a-in-K*: $a \in$ *carrier K*
  shows $a \cdot (x \ominus_V y) = a \cdot x \ominus_V a \cdot y$
proof −
  have *minus-y-in-V*: $\ominus_V\, y \in$ *carrier V*
    using *V.a-inv-closed* [*OF y-in-V*] **.**
  have *minus-one-in-K*: $\ominus_K\, \mathbf{1} \in$ *carrier K*
    using *K.a-inv-closed*[*OF K.one-closed*] **.**
  have *mclosed*: $a \cdot y \in$ *carrier V*
    using *mult-closed* [*OF y-in-V a-in-K*] **.**
  have *mclosed2*: $a \cdot x \in$ *carrier V*
    using *mult-closed* [*OF x-in-V a-in-K*] **.**
  have $a \cdot (x \ominus_V y) = a \cdot (x \oplus_V\ \ominus_V y)$
    using *a-minus-def*[*OF x-in-V y-in-V*] **by** *simp*
  also have $\ldots = a \cdot x \oplus_V a \cdot (\ominus_V y)$
    using *add-mult-distrib1* [*OF x-in-V minus-y-in-V a-in-K*] **.**
  also have $\ldots = a \cdot x \oplus_V a \cdot (\ominus_K \mathbf{1}_K \cdot y)$
    using *negate-eq* [*OF y-in-V*] **by** *simp*
  also have $\ldots = a \cdot x \oplus_V (a \otimes_K (\ominus_K \mathbf{1}_K)) \cdot y$
    using *mult-assoc* [*OF y-in-V a-in-K minus-one-in-K ,symmetric*]
    by *simp*
  also have $\ldots = a \cdot x \oplus_V ((\ominus_K \mathbf{1}_K) \otimes_K a) \cdot y$
    using *K.m-comm* [*OF minus-one-in-K a-in-K*] **by** *simp*
  also have $\ldots = a \cdot x \oplus_V (\ominus_K \mathbf{1}_K) \cdot a \cdot y$
    using *mult-assoc* [*OF y-in-V minus-one-in-K a-in-K*] **by** *simp*
  also have $\ldots = a \cdot x \oplus_V \ominus_V (a \cdot y)$
    using *negate-eq* [*OF mclosed*] **by** *simp*
  also have $\ldots = a \cdot x \ominus_V a \cdot y$
    using *a-minus-def* [*OF mclosed2 mclosed,symmetric*] **.**
  finally show *?thesis* **.**
qed

The following result proves distributivity of substraction (of $\mathbb{K}$) with respect to the scalar product:

lemma *diff-mult-distrib2*:
  assumes *x-in-V*: $x \in$ *carrier V*
  and *a-in-K*: $a \in$ *carrier K*
  and *b-in-K*: $b \in$ *carrier K*
  shows $(a \ominus_K b) \cdot x = a \cdot x \ominus_V b \cdot x$
proof −
  have *minus-b-in-K*: $\ominus_K b \in$ *carrier K*
    using *K.a-inv-closed* [*OF b-in-K*] **.**
  have *bx-in-V*: $b \cdot x \in$ *carrier V*
    using *mult-closed* [*OF x-in-V b-in-K*] **.**
  have $(a \ominus_K b) \cdot x = (a \oplus_K \ominus_K b) \cdot x$
    using *K.minus-eq* [*OF a-in-K b-in-K*] **by** *simp*
  also have $\ldots = a \cdot x \oplus_V (\ominus_K b) \cdot x$
    using *add-mult-distrib2* [*OF x-in-V a-in-K minus-b-in-K*] **.**
  also have $\ldots = a \cdot x \oplus_V (\ominus_K (\mathbf{1}_K \otimes_K b)) \cdot x$
    using *K.l-one* [*OF b-in-K*] **by** *simp*
  also have $\ldots = a \cdot x \oplus_V (\ominus_K \mathbf{1}_K \otimes_K b) \cdot x$
    using *K.l-minus* [*OF K.one-closed b-in-K,symmetric*] **by** *simp*
  also have $\ldots = a \cdot x \oplus_V (\ominus_K \mathbf{1}_K) \cdot b \cdot x$
    using *mult-assoc* [*OF x-in-V K.a-inv-closed*[*OF K.one-closed*] *b-in-K*]
    by *simp*
  also have $\ldots = a \cdot x \ \oplus_V \ \ominus_V (b \cdot x)$
    using *negate-eq* [*OF bx-in-V*] **by** *simp*
  also have $\ldots = a \cdot x \ominus_V b \cdot x$
    using *a-minus-def*[*OF mult-closed*[*OF x-in-V a-in-K*] *bx-in-V,symmetric*] **.**
  finally show *?thesis* **by** *simp*
qed

The following result proves that the unary substraction of $\mathbb{K}$ and V is a self-cancelling operation by means of the scalar product:

lemma *minus-mult-cancel*:
  assumes *x-in-V*: $x \in$ *carrier V* **and** *a-in-K*:$a \in$ *carrier K*
  shows $(\ominus_K a) \cdot (\ominus_V x) = a \cdot x$
proof −
  have $(\ominus_K a) \cdot (\ominus_V x) = (\ominus_K a \otimes (\ominus_K \mathbf{1}_K)) \cdot x$
    using *negate-eq*[*OF x-in-V*]
      mult-assoc[*OF x-in-V K.a-inv-closed*[*OF a-in-K*]
      K.a-inv-closed[*OF K.one-closed*]]
    by *auto*
  also have $\ldots = (a \otimes \mathbf{1}) \cdot x$
    using *K.prod-minus* [*OF a-in-K K.one-closed*] **by** *auto*
  finally show *?thesis* **using** *K.r-one* [*OF a-in-K*] **by** *auto*
qed

A result proving that the scalar product is commutative over the elements of $\mathbb{K}$:

lemma *mult-left-commute*:

assumes *x-in-V*: $x \in$ *carrier V*
and *a-in-K*: $a\in$ *carrier K*
and *b-in-K*:$b\in$ *carrier K*
shows $a \cdot b \cdot x = b \cdot a \cdot x$
proof −
have $a \cdot b \cdot x = (a \otimes b) \cdot x$
using *mult-assoc*[*OF x-in-V a-in-K b-in-K*, *symmetric*] **.**
also have $\ldots = (b \otimes a) \cdot x$ **using** *K.m-comm*[*OF a-in-K b-in-K*] **by** *simp*
finally show *?thesis*
using *mult-assoc*[*OF x-in-V b-in-K a-in-K*] **by** *simp*
qed

A result proving that the scalar product is left-cancelling for the elements of $\mathbb{K}$ different from 0:

lemma *mult-left-cancel*:
assumes *x-in-V*: $x \in$ *carrier V*
and *y-in-V*: $y\in$*carrier V*
and *a-in-K*: $a\in$*carrier K*
and *a-not-zero*: $a \neq \mathbf{0}_K$
shows $(a \cdot x = a \cdot y) = (x = y)$
proof
assume *ax-ay*:$a \cdot x = a \cdot y$
have *a-in-Units*: $a \in$ *Units K*
using *K.field-Units* **and** *a-in-K* **and** *a-not-zero* **by** *simp*
have $x = \mathbf{1}_K \cdot x$ **using** *mult-1*[*OF x-in-V*, *symmetric*] **.**
also have $\ldots = ((inv\ a) \otimes_K a) \cdot x$
using *K.Units-l-inv* [*OF a-in-Units*] **by** *simp*
also have $\ldots = (inv\ a) \cdot a \cdot x$
using *mult-assoc*[*OF x-in-V*
K.Units-inv-closed[*OF a-in-Units*] *a-in-K*]
by *simp*
also have $\ldots = (inv\ a) \cdot a \cdot y$ **using** *ax-ay* **by** *simp*
also have $\ldots = ((inv\ a) \otimes_K a) \cdot y$
using *mult-assoc*[*OF y-in-V K.Units-inv-closed*
[*OF a-in-Units*] *a-in-K*] **by** *simp*
also have $\ldots = \mathbf{1}_K \cdot y$
using *K.Units-l-inv* [*OF a-in-Units*, *symmetric*] **by** *simp*
finally show $x=y$ **using** *mult-1*[*OF y-in-V*] **by** *simp*
next
assume *x-y*: $x=y$
then show $a \cdot x = a \cdot y$ **by** *simp*
qed

A similar result to the previous one but proving that the element of V can be also cancelled:

lemma *mult-right-cancel*:
assumes *x-in-V*: $x \in$ *carrier V*
and *a-in-K*: $a \in$ *carrier K*
and *b-in-K*: $b\in$*carrier K*

```
  and x-not-zero: x≠0_V
  shows (a · x = b · x) = (a = b)
proof
  assume ax-by:a·x=b·x
  have (a ⊖ _K b)· x=a·x ⊖ _V b·x
    using diff-mult-distrib2[OF x-in-V a-in-K b-in-K] .
  also have ...=a·x ⊖ _V a·x using ax-by by simp
  also have ...=0_V
    using r-neg'[OF mult-closed[OF x-in-V a-in-K]] .
  finally have (a ⊖ _K b)· x=0_V by simp
  hence ab-zero: a ⊖ _K b=0_K
    using x-not-zero
    using integral[OF x-in-V K.minus-closed[OF a-in-K b-in-K]]
    by simp
  thus a=b
   proof −
     have a-min-b: a ⊕_K ⊖ _Kb=0_K
       using ab-zero and a-minus-def[OF a-in-K b-in-K] by simp
     have ⊖ _K(⊖ _K b)=a
       using K.minus-equality
       [OF a-min-b K.a-inv-closed[OF b-in-K] a-in-K] .
     thus ?thesis using K.minus-minus[OF b-in-K] by simp
   qed
next
  assume a=b
  then show a·x=b·x by simp
qed

end
end
theory Linear-dependence
imports Comments
begin
```

6 Linear dependence

```
context vector-space
begin
```

In this section we will present the definition of linearly dependent set and linearly independent set. First of all we will introduce the definition of *linear-combination.*

A linear combination is a finite sum of vectors of V multiplicated by scalars. However, how can we specify the scalars? In a linear combination each vector will be multiplicated by one specific scalar, so this scalar depends on the vector. For that reason, we introduce the notion of *coefficients-function.*

```
definition coefficients-function :: 'b set => ('b => 'a) set
```

where *coefficients-function* X
$= \{f.\ f \in X \to \mathit{carrier}\ K \wedge (\forall\, x.\ x \notin X \longrightarrow f\, x = \mathbf{0}_K)\}$

The explanation of the definition of coefficients function is as follows: given any set of vectors X, its coefficients functions will be every function which maps each of the vectors in X to scalars in $\mathbb{K}$. We impose an additional condition, in such a way that every element out of the set of vectors X is mapped to a distinguished element (in this case $\mathbf{0}$) of $\mathbb{K}$.

The first condition in the definition ($f \in X \to \mathit{carrier}\ K$) is clear. A coefficients function is a function which maps, as we have said before, the elements of a given set X to their corresponding scalars in $\mathbb{K}$. The second condition ($\forall\, x.\ x \notin X \longrightarrow f\, x = \mathbf{0}$) requires further explanation: the reason to map every element out of the set X to a distinguished point is that this allows us to compare coefficients functions through the extensional equality of functions $((f = g) = (\forall\, x.\ f\ x = g\ x))$. Thus, two coefficients function will be equal whenever they map every vector of X to the same scalar of $\mathbb{K}$ (this statement would not hold in the absence of the second condition).

Giving f a coefficients function and a certain x in *carrier* V then $f\, x$ (the scalar of the vector) will be in *carrier* K.

lemma *fx-in-K*:
  assumes *x-in-V*: $x \in \mathit{carrier}\ V$
  and *cf-f*: $f \in \mathit{coefficients\text{-}function}\ (\mathit{carrier}\ V)$
  shows $f(x) \in \mathit{carrier}\ K$
  using *assms* **unfolding** *coefficients-function-def* **by** *auto*

For every $x \in \mathit{carrier}\ V$, multiplication between the scalar and the vector ($f\ x \cdot x$) is in *carrier* V.

lemma *fx-x-in-V*:
  assumes *x-in-V*: $x \in \mathit{carrier}\ V$
  and *cf-f*: $f \in \mathit{coefficients\text{-}function}\ (\mathit{carrier}\ V)$
  shows $f(x) \cdot x \in \mathit{carrier}\ V$
  using *mult-closed*[*OF x-in-V fx-in-K*[*OF x-in-V cf-f*]] **.**

Now we are going to define a linear combination. In Halmos, next section is about linear combinations, however we have to introduce now the definition because we will use it to define the linear dependence of a set. We will use the definition of sums over a finite set (*finsum*) which already exists in the Isabelle library. Note that we are defining a *linear-combination* with two parameters: second is the set of elements of V and first is the coefficients function which assigns each vector to its scalar.

Due to the definition of *finsum-def* we are only considering the case of a finite linear combination. The case of infinite linear combinations is undefined. This is not a problem for us, because we will work with finite vector spaces and in our development we will only need linear combinations over finite sets. In addition, the sums in an infinite vector space are all finite because without additional structure the axioms of a vector space do not permit us to meaningfully speak about an infinite sum of vectors.

definition *linear-combination* :: $('b \Rightarrow 'a) \Rightarrow 'b\ set \Rightarrow 'b$
  where *linear-combination* $f\ X$ = *finsum* V $(\lambda y.\ f(y) \cdot y)$ X

In order to define the notion of linear dependence of a set we need to demand that this set be finite and a subset of the carrier. To abbreviate notation we will define these two premises as *good-set*.

definition *good-set* :: $'b\ set$ => *bool*
  where *good-set* X = (*finite* $X \wedge X \subseteq$ *carrier* V)

Next two lemmas show both properties:

lemma *good-set-finite*:
  assumes *good-set-X*: *good-set* X
  shows *finite* X
  using *good-set-X*
  unfolding *good-set-def* **by** *simp*

lemma *good-set-in-carrier*:
  assumes *good-set-X*: *good-set* X
  shows $X \subseteq$ *carrier* V
  using *good-set-X*
  unfolding *good-set-def* **by** *simp*

Empty set is a *good-set*.

lemma [*simp*]: *good-set* {}
  unfolding *good-set-def* **by** *simp*

Now, we can present the definition of linearly dependent set. A set will be dependent if there exists a linear combination equal to zero in which not all scalars are zero.

definition *linear-dependent* :: $'b\ set \Rightarrow$ *bool*
  where *linear-dependent* X = (*good-set* X
  $\wedge$ $(\exists f.\ f \in$ *coefficients-function* (*carrier* V) $\wedge$ *linear-combination* $f\ X = \mathbf{0}_V$
  $\wedge \neg(\forall x \in X.\ f\ x = \mathbf{0}_K)))$

This definition is equivalent to the previous one:

definition *linear-dependent-2* :: $'b\ set \Rightarrow$ *bool*
  where *linear-dependent-2* X =
  $(\exists f.\ f \in$ *coefficients-function* (*carrier* V) $\wedge$ *good-set* X
  $\wedge$ *linear-combination* $f\ X = \mathbf{0}_V \wedge \neg\ (\forall x \in X.\ f\ x = \mathbf{0}_K))$

Next lemma, which is in the library, proves that are equivalent

lemma $(\exists f.\ X \wedge Y f) = (X \wedge (\exists f.\ Y f))$
  using *ex-simps* (*2*) [*of* X Y] **.**

lemma *linear-dependent-eq-def*:
shows *linear-dependent* X = *linear-dependent-2* X
unfolding *linear-dependent-def*
unfolding *linear-dependent-2-def* **by** *blast*

We introduce now the notion of a linearly independent set. We will prove later that linear dependence and independence are complementary notions (every set will be either dependent or independent).

definition *linear-independent* :: *'b set* ⇒ *bool*
where *linear-independent* X =
(*good-set* X
∧ (∀ f. (f∈ *coefficients-function* (*carrier* V) ∧ *linear-combination* f X = $\mathbf{0}_V$)
⟶ (∀ x ∈ X. $f(x)$=$\mathbf{0}_K$)))

Next lemmas prove that if we have a linear (in)dependent set hence we have a *good-set* (finite and in the carrier).

lemma *l-ind-good-set*: *linear-independent* X ⟹ *good-set* X
unfolding *linear-independent-def* **by** *simp*

lemma *l-dep-good-set*: *linear-dependent* X ⟹ *good-set* X
unfolding *linear-dependent-def* **by** *simp*

The empty set is linearly independent.

lemma *empty-set-is-linearly-independent* [*simp*]:
shows *linear-independent* {}
unfolding *linear-independent-def*
by *simp*

We can prove that linear independence is the opposite of linear dependence. For that, we first prove that every set which is not linearly independent must be linearly dependent:

lemma *not-independent-implies-dependent*:
assumes *good-set*: *good-set* X
shows ¬ *linear-independent* X ⟹ *linear-dependent* X
proof (*unfold linear-dependent-def*)
assume *not-linear-independent*: ¬ *linear-independent* X
from *not-linear-independent* **obtain** f
where *f-in-coefficients*: f ∈ *coefficients-function* (*carrier* V)
and *sum-zero*: *linear-combination* f X = $\mathbf{0}_V$
and *not-all-zero*: ¬(∀ x ∈ X. $f(x)$=$\mathbf{0}_K$)
unfolding *linear-independent-def* **using** *good-set* **by** *best*
have f ∈ *coefficients-function* (*carrier* V)
∧ *linear-combination* f X = $\mathbf{0}_V$ ∧ ¬ (∀ x∈X. f x = $\mathbf{0}$)
using *f-in-coefficients* **and** *good-set* **and** *sum-zero* **and** *not-all-zero*
by *simp*
hence ∃ f. f ∈ *coefficients-function* (*carrier* V)
∧ *linear-combination* f X = $\mathbf{0}_V$ ∧ ¬ (∀ x∈X. f x = $\mathbf{0}$)

by (*rule exI* [*of - f*])
thus *good-set X* ∧ (∃ *f*. *f* ∈ *coefficients-function* (*carrier V*)
∧ *linear-combination f X* = $\mathbf{0}_V$ ∧ ¬ (∀ *x*∈*X*. *f x* = **0**))
using *good-set* **by** *simp*
qed

Now we prove that every set which is linearly dependent is not linearly independent:

lemma *dependent-implies-not-independent*:
shows *linear-dependent X* ⟹ ¬ *linear-independent X*
proof (*rule impE*)
assume *ld*: *linear-dependent X*
show ¬ *linear-independent X*
proof (*unfold linear-independent-def*)
from *ld* **obtain** *f* **where** *good-set*: *good-set X*
and *cf-f*: *f* ∈ *coefficients-function* (*carrier V*)
and *lc-f-X-zero*: *linear-combination f X* = $\mathbf{0}_V$
and *not-all-zero*: ¬(∀ *x* ∈ *X*. *f x* = $\mathbf{0}_K$)
unfolding *linear-dependent-def* **by** *auto*
have ¬ (∀ *f*. *f* ∈ *coefficients-function* (*carrier V*)
∧ *linear-combination f X* = $\mathbf{0}_V$ ⟶ (∀ *x*∈*X*. *f x* = **0**))
using *cf-f* **and** *lc-f-X-zero* **and** *not-all-zero* **by** *auto*
thus ¬ (*good-set X*
∧ (∀ *f*. *f* ∈ *coefficients-function* (*carrier V*)
∧ *linear-combination f X* = $\mathbf{0}_V$ ⟶ (∀ *x*∈*X*. *f x* = **0**)))
using *good-set* **by** *auto*
qed
qed (*auto*)

Hence the result:

lemma *dependent-if-only-if-not-independent*:
assumes *good-set*: *good-set X*
shows *linear-dependent X* ⟷ ¬ *linear-independent X*
using *dependent-implies-not-independent*
and *not-independent-implies-dependent* [*OF good-set*] **by** *auto*

Analogously, we can prove that a set is not linearly dependent if and only if it is linearly independent. We use ⟦¬ *P*; ¬ *R* ⟹ *P*⟧ ⟹ *R* and the previous lemma:

lemma *not-dependent-implies-independent*:
assumes *good-set*: *good-set X*
shows ¬ *linear-dependent X* ⟹ *linear-independent X*
proof −
assume *not-linear-dependent*: ¬ *linear-dependent X*
have *imp*: ¬ *linear-independent X* ⟹ *linear-dependent X*
using *not-independent-implies-dependent* [*OF good-set*] **.**
show *linear-independent X*
apply (*rule swap* [*OF not-linear-dependent imp*]) **.**

qed

lemma *independent-implies-not-dependent*:
  shows *linear-independent X* $\Longrightarrow$ $\neg$ *linear-dependent X*
proof –
  assume *li*: *linear-independent X*
  have *imp*: *linear-dependent X* $\Longrightarrow$ $\neg$ *linear-independent X*
    using *dependent-implies-not-independent* **.**
  show $\neg$ *linear-dependent X* **apply** (*rule swap*[*OF - imp*])
    using *li* **by** *simp+*
qed

Finally, we obtain the equivalence of definitions:

lemma *independent-if-only-if-not-dependent*:
  assumes *good-set*: *good-set X*
  shows *linear-independent X* $\longleftrightarrow$ $\neg$ *linear-dependent X*
  using *independent-implies-not-dependent*
    and *not-dependent-implies-independent* [*OF good-set*]
  by *fast*

Every good set will be either dependent or independent (but not both at the same time). Note: the operator OR of this proof is not an exclusive OR, so really here we are proving that every set is either dependent or independent or both.

lemma *li-or-ld*:
  assumes *good-set*:*good-set X*
  shows *linear-dependent X* | *linear-independent X*
proof (*cases linear-dependent X*)
  case *False* **show** *?thesis*
    using *not-dependent-implies-independent* [*OF good-set*] **by** *fast*
next
  case *True* **thus** *?thesis* **by** *fast*
qed

In order to avoid that problem, we need to implement the operator exclusive OR:

definition *xor* :: *bool* $\Rightarrow$ *bool* $\Rightarrow$ *bool*
  where *xor A B* $\equiv$ $(A \wedge \neg B) \vee (\neg A \wedge B)$

Now we can prove that every good set will be either dependent or independent (but not both at the same time):

lemma *li-xor-ld*:
  assumes *good-set*:*good-set X*
  shows *xor* (*linear-dependent X*) (*linear-independent X*)
proof (*unfold xor-def*,*auto*)
  assume *ld-X*: *linear-dependent X*
    and *li-X*: *linear-independent X*
  have $\neg$ *linear-independent X*

using *dependent-implies-not-independent*[*OF ld-X*] **.**
thus *False* **using** *li-X* **by** *contradiction*
next
assume ¬ *linear-independent X* **thus** *linear-dependent X*
using *not-independent-implies-dependent*[*OF good-set* -]
by *simp*
qed

A corollary of these theorems using that the empty set is linearly independent: if we have a linearly dependent set, then it isn't the empty set:

lemma *dependent-not-empty*:
assumes *ld-A*: *linear-dependent A*
shows $A \neq \{\}$
using *dependent-implies-not-independent*[*OF ld-A*] *empty-set-is-linearly-independent*
by *auto*

Now we prove that every set X containing a linearly dependent subset Y is itself linearly dependent. This property is stated in Halmos but not proved, he says that the fact is clear.

The proof is easy but long. We want to achieve a linear combination of the elements of X equal to zero and where not all scalars are zero. We know that a subset Y of X is dependent, so there exists a linear combination of the elements of Y equal to zero where not all scalars are zero (we will denote its coefficients funcion as f). If we define a coefficients function for the set X where the scalars of the elements $y \in Y$ are $f(y)$ and 0_K for the rest of elements in X, then we will obtain a linear combination of elements of X equal to zero where not all scalars are zero (because not for all $x \in Y$ $f(x)$ is 0_K).

lemma *linear-dependent-subset-implies-linear-dependent-set*:
assumes *Y-subset-X*: $Y \subseteq X$ **and** *good-set*: *good-set X*
and *linear-dependent-Y*: *linear-dependent Y*
shows *linear-dependent X*
proof (*unfold linear-dependent-def*)
— Using that Y is dependent, we can obtain a linear combination equal to zero where not all scalars are zero.
from *linear-dependent-Y*
obtain *f* **where** *sum-zero-f-Y*:*linear-combination f Y* = $\mathbf{0}_V$
and *not-all-zero-f*: ¬ ($\forall x \in Y.\ f\ x = \mathbf{0}$)
and *coefficients-function-f*:
$f \in$ *coefficients-function* (*carrier V*)
unfolding *linear-dependent-def*
by *best*
— Now we define the function and prove that is a coefficients function:
let *?g*= ($\lambda x.$ *if* $x \in Y$ *then* $f(x)$ *else* $\mathbf{0}_K$)
have *coefficients-function-g*:
?g $\in$ *coefficients-function* (*carrier V*)
using *coefficients-function-f*

unfolding *coefficients-function-def*
by *auto*

— We want to prove another two things: that the linear combination is zero and not all scalars are zero.

— First:

have *sum-zero-g-X*: *linear-combination ?g X* = $\mathbf{0}_V$
proof −

— We will separate the linear combination into two ones, in the set Y and in the set $X - Y$. We can do it thanks to the theorem *finsum-Un-disjoint*: ⟦*finite A*; *finite B*; $A \cap B = \{\}$; $g \in A \to$ *carrier V*; $g \in B \to$ *carrier V*⟧ $\Longrightarrow$ *finsum V g* $(A \cup B)$ = *finsum V g A* $\oplus_V$ *finsum V g B* and that the descomposition of the sets is disjoint.

— Some properties which we will need for the proof:

have *descomposicion-conjuntos*:$X=Y\cup(X-Y)$
using *Y-subset-X* **by** *auto*
have *disjuntos*: $Y \cap (X-Y)=\{\}$
by *simp*
have *finite-X*: *finite X*
using *good-set*
unfolding *good-set-def* **by** *simp*
have *finite-Y*: *finite Y*
using *linear-dependent-Y*
unfolding *linear-dependent-def*
unfolding *good-set-def* **by** *auto*
have *finite-X-minus-Y*: *finite* $(X-Y)$
using *finite-X* **by** *simp*
have *g1*:*?g* $\in Y \to$ *carrier K*
using *coefficients-function-g*
unfolding *coefficients-function-def*
using *good-set*
unfolding *good-set-def*
using *Y-subset-X*
by *auto*
have *g2*:*?g* $\in (X-Y) \to$ *carrier K*
using *coefficients-function-g*
unfolding *coefficients-function-def*
using *good-set*
unfolding *good-set-def*
by *auto*
let *?h*=$(\lambda x.$ *?g*$(x)\cdot x)$
have *h1*: *?h* $\in Y \to$ *carrier V*
proof
fix *x*
assume *x-in-Y*: $x\in Y$
have *x-in-V*: $x\in$ *carrier V*
proof
have *Y-subset-V*: $Y\subseteq$ *carrier V*
using *good-set*
unfolding *good-set-def*

using *Y-subset-X*
by *auto*
show *?thesis* **using** *Y-subset-V* **and** *x-in-Y* **by** *auto*
qed (*auto*)
have *gx-in-K*: *?g*$(x)\in$ *carrier K*
using *g1*
using *x-in-Y*
unfolding *Pi-def* **by** *auto*
have *gx-x-in-V*: *?g*$(x)\cdot x \in$ *carrier V*
using *mult-closed* [*OF x-in-V gx-in-K*] **by** *auto*
show (*if* $x \in Y$ *then f x else* **0**) $\cdot\ x \in$ *carrier V*
using *gx-x-in-V* **by** *auto*
qed
have *h2*: *?h* $\in (X-Y) \to$ *carrier V*
proof
fix *x*
assume *x-in-X-minus-Y*: $x\in (X-Y)$
have *x-in-V*: $x\in$ *carrier V*
proof
have *X-minus-Y-subset-V*: $(X-Y) \subseteq$ *carrier V*
using *good-set*
unfolding *good-set-def*
using *Y-subset-X*
by *auto*
show *?thesis*
using *X-minus-Y-subset-V*
using *x-in-X-minus-Y* **by** *auto*
qed (*auto*)
have *gx-in-K*: *?g*$(x)\in$ *carrier K*
using *x-in-X-minus-Y*
by *auto*
have *gx-x-in-V*: *?g*$(x)\cdot x \in$ *carrier V*
using *mult-closed* [*OF x-in-V gx-in-K*] **by** *auto*
show (*if* $x \in Y$ *then f x else* **0**) $\cdot\ x \in$ *carrier V*
using *gx-x-in-V* **by** *auto*
qed

— And now the decomposition. We will make a calculation until we achieve the thesis.

have *linear-combination ?g X*
= *linear-combination ?g* $(Y\cup(X-Y))$
using *descomposicion-conjuntos* **by** *simp*
also have *descomposicion*:
...=*linear-combination ?g Y* $\oplus_V$ *linear-combination ?g* $(X-Y)$
unfolding *linear-combination-def*
using *finsum-Un-disjoint* [*OF finite-Y finite-X-minus-Y*
disjuntos h1 h2]
by *auto*

— First linear combination of right term is the same linear combination of the elements of Y where it was equal to zero.

also have ...=$\mathbf{0}_V \oplus_V$ *linear-combination ?g* $(X-Y)$
proof –
have *linear-combination ?g Y=linear-combination f Y*
proof (*unfold linear-combination-def*)
have *iguales*: $Y=Y$ **..**
show $(\bigoplus_V y{\in}Y.\ (\textit{if } y \in Y \textit{ then } f\ y \textit{ else } \mathbf{0}) \cdot y)$
$= (\bigoplus_V y{\in}Y.\ f\ y \cdot y)$
using *finsum-cong* [*OF iguales*] **using** *h1* **by** *auto*
qed
also have ...=$\mathbf{0}_V$ **using** *sum-zero-f-Y* **.**
finally show *?thesis* **by** *simp*
qed
also have ...=$\mathbf{0}_V \oplus_V \mathbf{0}_V$
proof –

— Thanks to the definition of ?g, the linear combination in $(X - Y)$ is also zero (because all scalars are zero).

— As each scalar is zero, the multiplication between it and its vector is zero (*zeroK-mult-V-is-zeroV*: $x \in \textit{carrier } V \Longrightarrow \mathbf{0} \cdot x = \mathbf{0}_V$). Then we are adding a finite sum of zeros, so it will be zero using *finsum-zero*: $\textit{finite } A \Longrightarrow (\bigoplus_V i{\in}A.\ \mathbf{0}_V) = \mathbf{0}_V$.

have *sum-g-X-minus-Y*:*linear-combination ?g* $(X-Y)=\mathbf{0}_V$
proof –
have *X-subset-V*: $X \subseteq \textit{carrier } V$
using *good-set*
unfolding *good-set-def* **by** *auto*
hence *X-minus-Y-subset-V*:$(X-Y) \subseteq \textit{carrier } V$ **by** *auto*
have *not-in-Y*: $x{\in}\ (X-Y){\Longrightarrow}\ x{\notin}\ Y$ **by** *auto*
have *linear-combination ?g* $(X-Y)=(\bigoplus_V y{\in}X - Y.\ \mathbf{0} \cdot y)$
proof (*unfold linear-combination-def*)
have *igualesX-minus-Y*: $X-Y=X-Y$ **..**
show $(\bigoplus_V y{\in}X - Y.\ (\textit{if } y \in Y \textit{ then } f\ y \textit{ else } \mathbf{0}) \cdot y)$
$= \textit{finsum } V\ (\textit{op} \cdot \mathbf{0})\ (X - Y)$
using *finsum-cong* [*OF igualesX-minus-Y eqTrueI* [*OF h2*]]
by *auto*
qed
also have ...=$(\bigoplus_V y{\in}X - Y.\ \mathbf{0}_V)$
proof (*rule finsum-cong'*)
show $X - Y = X - Y$ **..**
show $(\lambda y.\ \mathbf{0}_V) \in X - Y \rightarrow \textit{carrier } V$ **by** *simp*
show $\bigwedge i.\ i \in X - Y \Longrightarrow \mathbf{0} \cdot i = \mathbf{0}_V$
using *zeroK-mult-V-is-zeroV*
using *X-minus-Y-subset-V* **by** *auto*
qed
also have ...=$\mathbf{0}_V$
using *finsum-zero* [*OF finite-X-minus-Y*] **.**
finally show *?thesis* **.**
qed
thus *?thesis* **by** *simp*
qed

also have ...$=\mathbf{0}_V$ **by** *simp*
finally show *?thesis* **.**
qed
— Second property is easy:
have *not-all-zero-g*: $\neg$ ($\forall x \in X.$ *?g* $x = \mathbf{0}$)
using *Y-subset-X*
using *not-all-zero-f* **by** *auto*
have *?g* $\in$ *coefficients-function* (*carrier V*)
$\wedge$ *linear-combination ?g X* $= \mathbf{0}_V \wedge \neg$ ($\forall x \in X.$ *?g* $x = \mathbf{0}$)
using *coefficients-function-g* **and** *good-set*
and *sum-zero-g-X* **and** *not-all-zero-g* **by** *fast*
hence
$\exists f.$ $f \in$ *coefficients-function* (*carrier V*)
$\wedge$ *linear-combination f X* $= \mathbf{0}_V \wedge \neg$ ($\forall x \in X.$ $f\ x = \mathbf{0}$)
by (*rule exI*[*of - ?g*])
thus *good-set X* $\wedge$ ($\exists f.$ $f \in$ *coefficients-function* (*carrier V*)
$\wedge$ *linear-combination f X* $= \mathbf{0}_V \wedge \neg$ ($\forall x \in X.$ $f\ x = \mathbf{0}$))
using *good-set* **by** *simp*
qed

More properties and facts:

lemma *exists-subset-ld*:
assumes *ld-X*: *linear-dependent X*
shows $\exists Y.$ $Y \subseteq X \wedge$ *linear-dependent Y*
using *ld-X* **by** *auto*

lemma *exists-subset-li*:
assumes *ld-X*: *linear-dependent X*
shows $\exists Y.$ $Y \subseteq X \wedge$ *linear-independent Y*
proof (*rule exI*[*of -* {}])
show $\{\} \subseteq X \wedge$ *linear-independent* {}
using *empty-set-is-linearly-independent* **by** *auto*
qed

A set containing $\mathbf{0}_V$ is not an independent set:

lemma *zero-not-in-linear-independent-set*:
assumes *li-A*: *linear-independent A*
shows $\mathbf{0}_V \notin A$
proof (*cases* $\mathbf{0}_V \notin A$)
case *True* **thus** *?thesis* **.**
next
case *False* **show** *?thesis*
proof −
have *cb-A*: *good-set A* **using** *l-ind-good-set*[*OF li-A*] **.**
have *zero-in-A*: $\mathbf{0}_V \in A$ **using** *False* **by** *simp*
let *?g*=($\lambda x.$ *if* $x = \mathbf{0}_V$ *then* $\mathbf{1}_K$ *else* $\mathbf{0}_K$)
have *cf-g*: *?g* $\in$ *coefficients-function* (*carrier V*)
unfolding *coefficients-function-def* **by** *auto*
have *lc-zero*: *linear-combination ?g A*$=\mathbf{0}_V$

proof (*unfold linear-combination-def*)
have ($\bigoplus_V y\in A.$ (*if* $y = \mathbf{0}_V$ *then* $\mathbf{1}$ *else* $\mathbf{0}$) $\cdot$ y)
=($\bigoplus_V y\in A.$ $\mathbf{0}_V$)
proof (*rule finsum-cong'*,*auto*)
show $\mathbf{1} \cdot \mathbf{0}_V = \mathbf{0}_V$
using *scalar-mult-zeroV-is-zeroV* **by** *auto*
fix *i*
assume *i-in-A*: $i \in A$ **and** *i-not-zero*: $i \neq \mathbf{0}_V$
show $\mathbf{0} \cdot i = \mathbf{0}_V$
using *zeroK-mult-V-is-zeroV* **and** *i-in-A* **and** *cb-A*
unfolding *good-set-def* **by** *auto*
qed
also have ...=$\mathbf{0}_V$
using *finsum-zero* **using** *good-set-finite*[*OF cb-A*] **by** *auto*
finally show
($\bigoplus_V y\in A.$ (*if* $y = \mathbf{0}_V$ *then* $\mathbf{1}$ *else* $\mathbf{0}$) $\cdot$ y) = $\mathbf{0}_V$ **.**
qed
have *not-all-zero*: $\neg(\forall x\in A.$ *?g* $x = \mathbf{0})$
using *zero-in-A* **by** *auto*
— Contradiction with *linear-independent*
show *?thesis*
using *cf-g lc-zero not-all-zero li-A*
unfolding *linear-independent-def* **by** *auto*
qed
qed

Every subset of an independent set is also independent. This property has been proved using *sledgehammer*.

lemma *independent-set-implies-independent-subset*:
assumes *A-in-B*: $A \subseteq B$
and *li-B*: *linear-independent B*
shows *linear-independent A*
by (*metis A-in-B good-set-def good-set-finite good-set-in-carrier dependent-implies-not-independent finite-subset l-ind-good-set li-B linear-dependent-subset-implies-linear-dependent-set not-independent-implies-dependent subset-trans*)

We can even extend the notions of linearly dependent and independent sets to infinite sets in the following way. We shall say that a set is linearly independent if every finite subset of it is such.

definition *linear-independent-ext*:: *'b set* $\Rightarrow$ *bool*
where *linear-independent-ext X*
= ($\forall A.$ *finite* $A \wedge A \subseteq X \longrightarrow$ *linear-independent* A)

Otherwise, it is linearly dependent.

definition *linear-dependent-ext*:: *'b set* $\Rightarrow$ *bool*
where *linear-dependent-ext X*
= ($\exists A.$ $A \subseteq X \wedge$ *linear-dependent* A)

As expected, if we have a linearly independent set it will be also *linear-independent-ext* set.

lemma *independent-imp-independent-ext*:
  assumes *li-X*: *linear-independent X*
  shows *linear-independent-ext X*
proof −
  have *fin-X*: *finite X* **and** *X-in-V*: $X \subseteq$ *carrier V*
    using *l-ind-good-set*[*OF li-X*] **unfolding** *good-set-def* **by** *simp+*
  show *?thesis* **unfolding** *linear-independent-ext-def*
  proof (*auto*)
    fix *A*
    assume *A-in-X*: $A \subseteq X$
    show *linear-independent A*
      using *independent-set-implies-independent-subset*
      [*OF A-in-X li-X*] **.**
  qed
qed

The same property holds for dependent sets:

lemma *dependent-imp-dependent-ext*:
  assumes *ld-X*: *linear-dependent X*
  shows *linear-dependent-ext X*
  unfolding *linear-dependent-ext-def*
  using *l-dep-good-set*[*OF ld-X*]
  unfolding *good-set-def*
  using *ld-X*
  by *fast*

Every finite set which is *linear-independent-ext* will also be *linear-independent*:

lemma *fin-ind-ext-impl-ind*:
  assumes *li-ext-X*: *linear-independent-ext X*
  and *finite-X*: *finite X*
  shows *linear-independent X*
  by (*metis finite-X li-ext-X linear-independent-ext-def subset-refl*)

Similarly with the notion of linear dependence:

lemma *fin-dep-ext-impl-dep*:
  assumes *ld-ext-X*: *linear-dependent-ext X*
  and *gs-X*: *good-set X*
  shows *linear-dependent X*
  by (*metis gs-X ld-ext-X linear-dependent-ext-def*
    linear-dependent-subset-implies-linear-dependent-set)

We can prove that also in the infinite case, the definitions of *linear-independent-ext* and *linear-dependent-ext* are complementary (every set will be of one type or the other). Let's see it:

lemma *not-independent-ext-implies-dependent-ext*:
  assumes *X-in-V*: $X \subseteq$ *carrier V*

shows ¬ *linear-independent-ext X* ⟹ *linear-dependent-ext X*
unfolding *linear-independent-ext-def* **and** *linear-dependent-ext-def*
using *not-independent-implies-dependent* **and** *X-in-V*
unfolding *good-set-def*
by *auto*

lemma *not-dependent-ext-implies-independent-ext*:
assumes *X-in-V*: *X* ⊆ *carrier V*
shows ¬ *linear-dependent-ext X* ⟹ *linear-independent-ext X*
by (*metis X-in-V not-independent-ext-implies-dependent-ext*)

lemma *independent-ext-implies-not-dependent-ext*:
shows *linear-independent-ext X* ⟹ ¬ *linear-dependent-ext X*
by (*metis good-set-finite independent-implies-not-dependent l-dep-good-set linear-dependent-ext-def linear-independent-ext-def*)

lemma *dependent-ext-implies-not-independent-ext*:
shows *linear-dependent-ext X* ⟹ ¬ *linear-independent-ext X*
by (*metis independent-ext-implies-not-dependent-ext*)

corollary *dependent-ext-if-only-if-not-indepentent-ext*:
assumes *X-in-V*: *X* ⊆ *carrier V*
shows *linear-dependent-ext X* ⟷ ¬ *linear-independent-ext X*
using *assms not-independent-ext-implies-dependent-ext dependent-ext-implies-not-independent-ext*
by *blast*

corollary *independent-ext-if-only-if-not-depentent-ext*:
assumes *X-in-V*: *X* ⊆ *carrier V*
shows *linear-independent-ext X* ⟷ ¬ *linear-dependent-ext X*
using *assms not-dependent-ext-implies-independent-ext independent-ext-implies-not-dependent-ext*
by *blast*

end
end
theory *Indexed-Set*
imports *Main FuncSet Previous*
begin

7 Indexed sets

The next type definition, *iset*, represents the notion of an indexed set, which is a pair: a set and a function that goes from naturals to the set.

type-synonym ('*a*) *iset* = '*a set* × (*nat* => '*a*)

Now we define functions which make possible to separate an indexed set into the set and the function and we add them to the simplifier, since they are only meant to be abbreviations of the "fst" and "snd" operations:

definition *iset-to-set* :: *'a iset* => *'a set*
 where *iset-to-set A* = *fst A*

definition *iset-to-index* :: *'a iset* => (*nat* => *'a*)
 where *iset-to-index A* = *snd A*

lemmas [*simp*] = *iset-to-set-def iset-to-index-def*

An indexing of a set will be any bijection between the set of the natural numbers less than its cardinality (because we start counting from *0*) and the set. Note: we will always work with finite sets. By default, the definition of *card* assigns to an infinite set cardinality equal to *0*.

definition *indexing* :: (*'a iset*) => *bool*
 where *indexing A* = *bij-betw* (*iset-to-index A*)
 {..<*card* (*iset-to-set A*)} (*iset-to-set A*)

Once we have the definition of *indexing*, we are going to prove some properties of it:

We introduce some lemmas presenting properties and alternative definitions of "indexing". For instance, whenever we have an indexing $A = (iset\_to\_set\ A, iset\_to\_index\ A)$ the index function will map naturals in the range $\{.. < card(A)\}$ to elements of $iset\_to\_set\ A$ and, moreover, the image set of the indexing function in such range will be whole set $iset\_to\_set\ A$.

lemma *indexing-equiv-img*:
 assumes *ob*: *indexing A*
 shows (*iset-to-index A*)
 ∈ {..<(*card* (*iset-to-set A*))} → (*iset-to-set A*)
 ∧ (*iset-to-index A*) ‘ {..<(*card* (*iset-to-set A*))}
 = (*iset-to-set A*)
 using *ob*
 unfolding *indexing-def*
 unfolding *bij-betw-def* **by** *auto*

The implication is also satisfied in the opposite direction:

lemma *img-equiv-indexing*:
 assumes *f*: (*iset-to-index A*)
 ∈ {..<(*card* (*iset-to-set A*))} → (*iset-to-set A*)
 ∧ (*iset-to-index A*) ‘ {..<(*card* (*iset-to-set A*))}
 = (*iset-to-set A*)
 shows *indexing A*
 proof −
 have *inj-on* (*iset-to-index A*) {..<*card* (*iset-to-set A*)}
  proof −
  have *card* ((*iset-to-index A*) ‘ {..<(*card* (*iset-to-set A*))})

=*card* (*iset-to-set A*) **using** *f* **by** *auto*
also have ...= *card* ({..<*card* (*iset-to-set A*)})
using *card-lessThan* **by** *auto*
finally have *1*:
card ((*iset-to-index A*) ‘ {..<(*card* (*iset-to-set A*))})
= *card* ({..<*card* (*iset-to-set A*)}) **.**
have *2*: *finite* {..<*card* (*iset-to-set A*)}
by (*metis finite-lessThan*)
show *?thesis* **using** *eq-card-imp-inj-on* [*OF 2 1*] **.**
qed
moreover have *iset-to-index A* ‘ {..<*card* (*iset-to-set A*)}
= *iset-to-set A* **using** *f* **by** *auto*
ultimately show *?thesis*
unfolding *indexing-def* **unfolding** *bij-betw-def*
by *simp*
qed

Now we present another alternative definition of indexing linking it with the notions of injectivity and surjectivity:

lemma *indexing-inj-surj*:
assumes *ob*: *indexing A*
shows *inj-on* (*iset-to-index A*) {..<(*card* (*iset-to-set A*))}
∧ (*iset-to-index A*) ‘ {..<(*card* (*iset-to-set A*))}
= (*iset-to-set A*)
using *ob*
unfolding *indexing-def*
unfolding *bij-betw-def* **.**

lemma *indexing-inj-surj-inv*:
assumes *inj-on* (*iset-to-index A*) {..<(*card* (*iset-to-set A*))}
∧ (*iset-to-index A*) ‘ {..<(*card* (*iset-to-set A*))} = (*iset-to-set A*)
shows *indexing A*
unfolding *indexing-def*
unfolding *bij-betw-def* **by** *fact*

One basic property is that the empty set with any function of appropriate type is an *indexing*:

lemma *indexing-empty*:
indexing ({}, *f*)
unfolding *indexing-def*
unfolding *bij-betw-def* **by** *simp*

We can obtain an equivalent notion of previous lemma writing the property in the unfolded definition of *indexing*.

lemma *indexing-empty-inv*:
shows *inj-on* (*iset-to-index* ({}, *f*)) {..<*card* (*iset-to-set* ({}, *f*))}
∧ *iset-to-index* ({}, *f*) ‘ {..<*card* (*iset-to-set* ({}, *f*))} = *iset-to-set* ({}, *f*) **by** *simp*

Now we are proving a basic but useful lemma: if we have an *indexing* of a set, then the image of a natural less than the cardinality of the set is an element of the set.

lemma *indexing-in-set*:
  assumes *indexing* (A,f)
  and $n <$ *card* A
  shows $f\ n \in A$
  using *assms* **unfolding** *indexing-def bij-betw-def* **by** *auto*

We present two auxiliary lemmas about indexings and their behaviours as injective functions. The first one claims that if we have an *indexing* and two naturals (less than the cardinality of the set) with the same image, then the naturals are equal (which is a consequence of injectivity):.

lemma
  indexing-impl-eq-preimage:
  assumes i: *indexing* (A, f)
  and x: $x \in \{..<$*card* $A\}$ **and** y: $y \in \{..<$*card* $A\}$
  and f: $f\ x = f\ y$
  shows $x = y$
  apply (*rule inj-onD* [*of* f $\{..<$*card* $A\}$])
  using i
  unfolding *indexing-def bij-betw-def*
  by *simp fact+*

On the contrary, if we have the same assumptions than before but we consider that the image of both naturals are different, then the numbers are distinct.

lemma
  indexing-impl-ndiff-image:
  assumes i: *indexing* (A, f)
  and x: $x \in \{..<$*card* $A\}$ **and** y: $y \in \{..<$*card* $A\}$
  and f: $x \neq y$
  shows $f\ x \neq f\ y$
proof (*rule ccontr*, *simp*)
  assume $f\ x = f\ y$
  hence $x = y$
    using i
    unfolding *indexing-def bij-betw-def inj-on-def*
    using $x\ y$ **by** *auto*
  thus *False* **using** f **by** *contradiction*
qed

The following lemma proves that for any finite set A, there exist a natural number n and a function f such that f is an index function of A with $\{.. < n\}$ the collection of indexes. The prof is no constructive, is based on a lemma in the Isabelle library proving that every finite set is a mapping of a range of the naturals.

lemma *finite-imp-nat-seg-image-inj-on-Pi*:
  assumes f: *finite* A
  shows $(\exists\, n{::}nat.\ \exists f{\in}\{i.\ i < n\} \rightarrow A.$
  $((f\ `\ \{i.\ i < n\} = A) \wedge inj\text{-}on\ f\ \{i.\ i < n\}))$
proof $-$
  obtain f **and** n
    where $a1$: $f\ `\ \{i.\ i < (n{::}nat)\} = A \wedge inj\text{-}on\ f\ \{i.\ i < n\}$
    and $a2$: $f \in \{i.\ i < n\} \rightarrow A$
    using *finite-imp-nat-seg-image-inj-on* [*OF f*] **by** *auto*
  thus *?thesis* **by** *auto*
qed

The bijection is between the naturals up to card A and the set. Thanks to that we are giving to the set an indexation, we are representing a set more or less like a vector in C++: a structure with $card(A)$ components (from position 0 to $(card(A) - 1)$). Each component $f(i)$ tallies with one element of the set.

The following lemma extends the previous one, since we prove that n in the previous lemma is actually $card(A)$. The proof is carried out by induction on the finite set A, and the indexing function is explicitly given ($?f$ in the proof below):

lemma *finite-imp-nat-seg-image-inj-on-Pi-card*:
  assumes f: *finite* A
  shows $(\exists f \in \{i.\ i < (card\ A)\} \rightarrow A.\ ((f\ `\ \{i.\ i < (card\ A)\} = A)$
  $\wedge\ inj\text{-}on\ f\ \{i.\ i < (card\ A)\}))$
  using f **proof** (*induct*)
  case *empty*
  show *?case* **by** *auto*
next
  case (*insert b B*)
  show $\exists f{\in}\{i{::}nat.\ i <$
    $card\ (insert\ b\ B)\} \rightarrow insert\ b\ B.$
    $f\ `\ \{i{::}nat.\ i < card\ (insert\ b\ B)\} = insert\ b\ B\ \wedge$
    $inj\text{-}on\ f\ \{i{::}nat.\ i < card\ (insert\ b\ B)\}$
  proof $-$
    obtain g
      where $g1$: $g \in \{i.\ i < (card\ B)\} \rightarrow B$
      and $g2$: $g\ `\ \{i{::}nat.\ i < card\ B\} = B \wedge inj\text{-}on\ g$
      $\{i{::}nat.\ i < card\ B\}$
      using *insert.hyps* (*3*) **by** *auto*
    let $?f = (\lambda n{::}nat.\ if\ n \in \{i.\ i < card\ B\}\ then\ g\ n$
      $else\ if\ n = card\ B\ then\ b\ else\ g\ n)$
    have $f1$: $?f\ \in\{i{::}nat.\ i < card\ (insert\ b\ B)\}$
      $\rightarrow insert\ b\ B$
    proof
      fix x
      assume *x-bounded*: $x \in \{i{::}nat.\ i < card\ (insert\ b\ B)\}$
      show $(if\ x \in \{i{::}nat.\ i < card\ B\}\ then\ g\ x\ else\ if\ x$

= *card B then b else g x*) ∈ *insert b B*
proof (*cases x* ∈ {*i*::*nat*. *i* < *card B*})
case *True* **then show** *?thesis* **using** *g1* **unfolding** *Pi-def* **by** *simp*
next
case *False*
have *x* = *card B*
proof −
have *card* (*insert b B*) = *Suc* (*card B*) — To prove this we need that *b* won't be in *B* and that the set be finite
using *insert.hyps* (*2*)
using *insert.hyps* (*1*) **by** *simp*
thus *?thesis*
using *False*
using *x-bounded* **by** *simp*
qed
thus *?thesis* **by** *simp*
qed
qed
have *f2*: *?f* ' {*i*::*nat*. *i* < *card* (*insert b B*)}
= (*insert b B*) ∧ *inj-on ?f* {*i*::*nat*. *i* < *card* (*insert b B*)}
proof
show *?f* ' {*i*::*nat*. *i* < *card* (*insert b B*)} = *insert b B*
proof −
have *?f* ' {*i*::*nat*. *i* < *card* (*insert b B*)} = *?f* '
({*i*::*nat*. *i* < *card B*} ∪ {*i*. *i* = *card B*})
using *insert.hyps* (*2*)
using *insert.hyps* (*1*) **by** *auto*
also have ... = *?f* ' {*i*::*nat*. *i* < *card B*} ∪
?f ' {*i*. *i* = *card B*}
by (*rule image-Un*)
also have ... = *B* ∪ *?f* ' {*i*. *i* = *card B*}
using *g2* **by** *auto*
also have ... = *B* ∪ {*b*} **by** *simp*
finally show *?thesis* **by** *simp*
qed
show *inj-on ?f* {*i*::*nat*. *i* < *card* (*insert b B*)}
proof −
have *inj-on ?f* {*i*::*nat*. *i* < *card* (*insert b B*)} = *inj-on*
?f (*insert* (*card B*) {*i*. *i* < (*card B*)})
proof −
have {*i*::*nat*. *i* < *card* (*insert b B*)} = *insert* (*card*
B) {*i*. *i* < (*card B*)}
using *insert.hyps* (*2*)
using *insert.hyps* (*1*) **by** *auto*
thus *?thesis* **by** *simp*
qed
also have ... = (*inj-on ?f* {*i*. *i* < (*card B*)} ∧ *?f*
(*card B*) ∉ *?f* ' ({*i*. *i* < (*card B*)} − {*card B*}))
by (*rule inj-on-insert*)

also have $\ldots = (True \wedge\ ?f\ (card\ B) \notin\ ?f$ ‘
  $(\{i.\ i < (card\ B)\} - \{card\ B\}))$
  using *g2* **unfolding** *inj-on-def* **by** *auto*
also have $\ldots = (True \wedge True)$
  using *insert.hyps* (*2*) **using** *g2* **by** *auto*
also have $\ldots = True$ **by** *fast*
finally show *?thesis* **by** *fast*
qed
qed
show *?thesis*
  using *f1 f2* **by** *auto*
qed
qed

As a corollary, we prove that for each finite set there exists an indexing of it. This is the main theorem of this section and it will be very useful in the future to assign an order to a finite set (we will need it in future proofs).

corollary *obtain-indexing*:
  assumes *finite-A*: *finite A*
  shows $\exists f.\ indexing\ (A,f)$
proof (*unfold indexing-def*,*unfold bij-betw-def*,*auto*)
  from *finite-A* **obtain** *f* **where** *surj*: f ‘ $\{i.\ i < (card\ A)\} = A$ **and** *inj-on*: *inj-on* $f\ \{i.\ i < (card\ A)\}$
    using *finite-imp-nat-seg-image-inj-on-Pi-card*[*OF finite-A*] **by** *auto*
  show $\exists f.\ inj\text{-}on\ f\ \{..<card\ A\} \wedge f$ ‘ $\{..<card\ A\} = A$ **using** *surj* **and** *inj-on* **and** *lessThan-def*[*of card A*]
    by *auto*
qed

In addition, if we have an indexing we will know that the set is finite. This lemma will allow us to remove the premise $finite\ A$ whenever we have indexings. This is because Isabelle assigns 0 as the cardinality of an infinite set. Suppose that A is infinite. If we have an $indexing(A, f)$, hence f is a bijection between the set of naturals less than the cardinality of A (0 due to the implementation) and A. Then, $A = f‘\{.. < card(A)\} = f‘\{.. < 0\} = f‘\{\} = \{\}$. However, we have supposed that A was infinite and $\{\}$ is not, so we have a contradiction and A is always finite.

lemma *indexing-finite*[*simp*]:
  assumes *indexing-A*: *indexing* (*A*,*f*)
  shows *finite A*
  by (*metis bij-betw-finite finite-lessThan*
    fst-conv indexing-def iset-to-set-def indexing-A)

After introducing the notion of indexed set, we need to introduce two basic operations over indexed sets: insert and remove. They will be generic with respect to the position where an element can be inserted or removed. For instance, given an indexed set $\{(a, 0), (b, 1), (c, 2)\}$ if we are to insert an element d, we will admit indexing $\{(d, 0), (a, 1), (b, 2), (c, 3)\}, \{(a, 0), (d, 1), (b, 2), (c, 3)\}$

and so on. In other words, inserting an element in a sorted set preserves the order of the elements, but maybe not their positions.

First we define the function which, for a given indexing A and an element a gives all possible indexings for the set *insert a (iset_to_set A)* preserving *(iset_to_index A)*:

n is the position where 'a' will be inserted. It shoul be a natural number between 0 (first position) and card A (last position).

```
definition indexing-ext :: ('a iset) => 'a => (nat => nat => 'a)
  where
  indexing-ext A a =
  (%n. %k. if k < n then (iset-to-index A) k
  else if k = n then a
  else (iset-to-index A) (k − 1))
```

Now we present a basic property (it will be useful to be applied in induction proofs): If one *indexing-ext* generated from an indexation F and from one element $a \notin$ *index-to-set* F is good (is an indexing), then the indexation of F is also good (an indexing).

It is a long lemma (about 300 lines). The proof of injectivity must be separated in several different cases, depending on the position where we insert the element (after, before or exactly in the *nth* position):

```
lemma indexing-indexing-ext:
  assumes ob:
  indexing ((insert x (iset-to-set F)), (indexing-ext F x n))
  and n1: 0 ≤ n
  and n2: n ≤ card (iset-to-set F)
  and x-notin-F: x ∉ (iset-to-set F)
  shows indexing F
proof (unfold indexing-def bij-betw-def, intro conjI)
  let ?h = iset-to-index F
  let ?F = iset-to-set F
  show inj-on-h:inj-on ?h {..<card ?F}
  proof (unfold inj-on-def, rule ballI, rule ballI, rule impI)
    fix xa y
    assume xa: xa ∈ {..<card ?F}
      and y: y ∈ {..<card ?F} and h: ?h xa = ?h y
    show xa = y
    proof (rule inj-onD
        [of (indexing-ext F x n) {..<card (insert x ?F)}])
      show xa ∈ {..<card (insert x ?F)}
         using xa
        by (metis card-infinite card-insert-le gr-implies-not0 le-neq-implies-less
          lessThan-iff less-or-eq-imp-le order-le-less-trans)
      show y ∈ {..<card (insert x ?F)}
        using y
```

```
    by (metis card-infinite card-insert-le gr-implies-not0 le-neq-implies-less
      lessThan-iff less-or-eq-imp-le order-le-less-trans)
   show inj-on (indexing-ext F x n) {..<card (insert x ?F)}
    using ob unfolding indexing-def bij-betw-def
    by auto
  next
   show indexing-ext F x n xa = indexing-ext F x n y
   proof (cases xa < n)
    case True note xa-l-n = True
    show ?thesis
    proof (cases y < n)
     case True
     show ?thesis
      unfolding indexing-ext-def
      using xa-l-n True using h by simp
    next
     case False hence n-le-y: n ≤ y and xa-l-y: xa < y
      using xa-l-n by simp-all
     have ?h xa = (indexing-ext F x n) xa
      unfolding indexing-ext-def
      using xa-l-n by simp
     moreover have ?h y = (indexing-ext F x n) (Suc y)
      using n-le-y
      unfolding indexing-ext-def by simp
     ultimately
     have eq: (indexing-ext F x n) xa = (indexing-ext F x n) (Suc y)
      using h by simp
     have xa = Suc y
     proof (rule inj-onD [of indexing-ext F x n {..<card (insert x ?F)}])
      show inj-on (indexing-ext F x n) {..<card (insert x ?F)}
       using ob
       unfolding indexing-def
       unfolding bij-betw-def by auto
      show indexing-ext F x n xa = indexing-ext F x n (Suc y)
       using eq .
      show xa ∈ {..<card (insert x ?F)}
       using xa
     by (metis card-infinite card-insert-disjoint lessThan-iff less-SucI less-zeroE
x-notin-F)
      show Suc y ∈ {..<card (insert x ?F)}
       using x-notin-F using y
          by (metis Suc-mono card-infinite card-insert-disjoint lessThan-iff
less-zeroE)
     qed
     hence False using xa-l-y by simp
     thus ?thesis by simp
    qed
   next
    case False
```

```
      hence n-le-xa: n ≤ xa using False by simp
      show ?thesis
      proof (cases n = xa)
        case True note n-eq-xa = True
        show ?thesis
        proof (cases y < n)
          case True
          have x-eq: ?h xa = indexing-ext F x n (Suc xa)
            unfolding indexing-ext-def
            using n-eq-xa by simp
          moreover have y-eq: ?h y = indexing-ext F x n y
            unfolding indexing-def
            using True unfolding indexing-ext-def by simp
          ultimately
          have eq: (indexing-ext F x n) y = (indexing-ext F x n) (Suc xa)
            using h by simp
          have y = Suc xa
          proof (rule inj-onD [of indexing-ext F x n {..<card (insert x ?F)}])
            show inj-on (indexing-ext F x n) {..<card (insert x ?F)}
              using ob
              unfolding indexing-def
              unfolding bij-betw-def by auto
            show indexing-ext F x n y = indexing-ext F x n (Suc xa)
              using eq .
            show y ∈ {..<card (insert x ?F)}
              using y
                by (metis card-infinite card-insert-disjoint lessThan-iff less-SucI
less-zeroE x-notin-F)
            show Suc xa ∈ {..<card (insert x ?F)}
              using x-notin-F using xa
                by (metis Suc-mono card-infinite card-insert-disjoint lessThan-iff
less-zeroE)
          qed
          hence False using n-eq-xa True by simp
          thus ?thesis by simp
        next
          case False
          hence n-le-y: n ≤ y by simp
          show ?thesis
          proof (cases n = y)
            case True note n-eq-y = True
            show ?thesis
              unfolding indexing-ext-def
              using n-eq-xa n-eq-y by simp
          next
            case False hence n-l-y: n < y using n-le-y by simp
            have x-eq: ?h xa = indexing-ext F x n (Suc xa)
              unfolding indexing-ext-def
              using n-eq-xa by simp
```

```
          moreover have y-eq: ?h y = indexing-ext F x n (Suc y)
            unfolding indexing-ext-def
            using n-l-y by simp
          ultimately
          have eq: (indexing-ext F x n) (Suc y) = (indexing-ext F x n) (Suc xa)
            using h by simp
          have Suc y = Suc xa
          proof (rule inj-onD [of indexing-ext F x n {..<card (insert x ?F)}])
            show inj-on (indexing-ext F x n) {..<card (insert x ?F)}
              using ob
              unfolding indexing-def
              unfolding bij-betw-def by auto
            show indexing-ext F x n (Suc y) = indexing-ext F x n (Suc xa)
              using eq .
            show Suc y ∈ {..<card (insert x ?F)}
              using y
                by (metis Suc-mono card-infinite card-insert-disjoint lessThan-iff
less-zeroE x-notin-F)
            show Suc xa ∈ {..<card (insert x ?F)}
              using x-notin-F using xa
                by (metis Suc-mono card-infinite card-insert-disjoint lessThan-iff
less-zeroE)
          qed
          hence False using n-eq-xa n-l-y by simp
          thus ?thesis by simp
        qed
      qed
    next
      case False
      hence n-l-xa: n < xa using n-le-xa by simp
      show ?thesis
      proof (cases y < n)
        case True note y-l-n = True
        have x-eq: ?h xa = indexing-ext F x n (Suc xa)
          unfolding indexing-ext-def
          using n-l-xa by simp
        moreover have y-eq: ?h y = indexing-ext F x n y
          unfolding indexing-ext-def
          using True by simp
        ultimately
        have eq: (indexing-ext F x n) y =  (indexing-ext F x n) (Suc xa)
          using h by simp
        have y = Suc xa
        proof (rule inj-onD [of indexing-ext F x n {..<card (insert x ?F)}])
          show inj-on (indexing-ext F x n) {..<card (insert x ?F)}
            using ob
            unfolding indexing-def
            unfolding bij-betw-def by auto
          show indexing-ext F x n y = indexing-ext F x n (Suc xa)
```

```
            using eq .
          show y ∈ {..<card (insert x ?F)}
            using y
              by (metis card-infinite card-insert-disjoint lessThan-iff less-SucI
less-zeroE x-notin-F)
          show Suc xa ∈ {..<card (insert x ?F)}
            using x-notin-F using xa
              by (metis Suc-mono card-infinite card-insert-disjoint lessThan-iff
less-zeroE)
        qed
        hence False using n-l-xa True by simp
        thus ?thesis by simp
      next
        case False
        hence n-le-y: n ≤ y by simp
        show ?thesis
        proof (cases n = y)
          case True note n-eq-y = True
          have ?h xa = (indexing-ext F x n) (Suc xa)
            unfolding indexing-ext-def
            using n-l-xa by simp
          moreover have ?h y = (indexing-ext F x n) (Suc y)
            using n-le-y
            unfolding indexing-ext-def by simp
          ultimately
          have eq: (indexing-ext F x n) (Suc xa) = (indexing-ext F x n) (Suc y)
            using h by simp
          have Suc xa = Suc y
          proof (rule inj-onD [of indexing-ext F x n {..<card (insert x ?F)}])
            show inj-on (indexing-ext F x n) {..<card (insert x ?F)}
              using ob
              unfolding indexing-def
              unfolding bij-betw-def by auto
            show indexing-ext F x n (Suc xa) = indexing-ext F x n (Suc y)
              using eq .
            show Suc xa ∈ {..<card (insert x ?F)}
              using xa
                by (metis Suc-mono card-infinite card-insert-disjoint lessThan-iff
less-zeroE x-notin-F)
            show Suc y ∈ {..<card (insert x ?F)}
              using x-notin-F using y
                by (metis Suc-mono card-infinite card-insert-disjoint lessThan-iff
less-zeroE)
          qed
          hence False using n-l-xa n-eq-y by simp
          thus ?thesis by simp
        next
          case False
          hence n-l-y: n < y using n-le-y by simp
```

```
              have ?h xa = (indexing-ext F x n) (Suc xa)
                unfolding indexing-ext-def
                using n-l-xa by simp
              moreover have ?h y = (indexing-ext F x n) (Suc y)
                using n-l-y
                unfolding indexing-ext-def by simp
              ultimately
             have eq: (indexing-ext F x n) (Suc xa) =  (indexing-ext F x n) (Suc y)
                using h by simp
              have Suc xa = Suc y
              proof (rule inj-onD [of indexing-ext F x n {..<card (insert x ?F)}])
                show inj-on (indexing-ext F x n) {..<card (insert x ?F)}
                  using ob
                  unfolding indexing-def
                  unfolding bij-betw-def by auto
                show indexing-ext F x n (Suc xa) = indexing-ext F x n (Suc y)
                  using eq .
                show Suc xa ∈ {..<card (insert x ?F)}
                  using xa
                    by (metis Suc-mono card-infinite card-insert-disjoint lessThan-iff
less-zeroE x-notin-F)
                show Suc y ∈ {..<card (insert x ?F)}
                  using x-notin-F using y
                    by (metis Suc-mono card-infinite card-insert-disjoint lessThan-iff
less-zeroE)
              qed
              thus ?thesis by simp
            qed
          qed
        qed
      qed
    qed
  qed
  show ?h ‘ {..<card ?F} = ?F
  proof −
    have finite-iset-to-set-F: finite (iset-to-set F)
         by (metis bij-betw-finite finite-insert finite-lessThan fst-conv indexing-def
iset-to-set-def ob)
     have surj-indexing: (indexing-ext F x n) ‘ {..<card (insert x ?F)}=(insert x
(?F))
      using ob unfolding indexing-def and bij-betw-def by auto
    have inj-on-indexing: inj-on (indexing-ext F x n) {..<card (insert x ?F)}
      using ob unfolding indexing-def bij-betw-def by auto
   have descomposicion-conjunto: {..<card (insert x ?F)}={..<n}∪{n}∪{n<..<card
(insert x ?F)}
      using n1 and n2 and x-notin-F and card-insert-if and finite-iset-to-set-F
      unfolding iset-to-set-def by auto
     have F-indexing-ext-desc:(?F)=(indexing-ext F x n) ‘ {..<n} ∪ (indexing-ext
F x n) ‘ {n<..<card (insert x ?F)}
```

proof −
have *descomposicion-conjuntos2*: $\{..<n\} \cup \{n<..<card\ (insert\ x\ ?F)\}=\{..<card\ (insert\ x\ ?F)\}-\{n\}$
using *n2* **and** *descomposicion-conjunto* **by** *auto*
have $(indexing\text{-}ext\ F\ x\ n)$ ‘ $\{..<n\} \cup (indexing\text{-}ext\ F\ x\ n)$ ‘ $\{n<..<card\ (insert\ x\ ?F)\}$
$=(indexing\text{-}ext\ F\ x\ n)$ ‘ $(\{..<n\} \cup \{n<..<card\ (insert\ x\ ?F)\})$
by *auto*
also have $...=indexing\text{-}ext\ F\ x\ n$ ‘ $(\{..<card\ (insert\ x\ ?F)\}-\{n\})$
using *descomposicion-conjuntos2* **by** *auto*
also have $...=indexing\text{-}ext\ F\ x\ n$ ‘ $\{..<card\ (insert\ x\ ?F)\}-indexing\text{-}ext\ F\ x\ n$ ‘$\{n\}$
proof (*rule inj-on-image-set-diff* [*OF inj-on-indexing*])
show $\{..<card\ (insert\ x\ (iset\text{-}to\text{-}set\ F))\} \subseteq \{..<card\ (insert\ x\ (iset\text{-}to\text{-}set\ F))\}$ **..**
show $\{n\} \subseteq \{..<card\ (insert\ x\ (iset\text{-}to\text{-}set\ F))\}$ **using** *descomposicion-conjunto* **by** *auto*
qed
also have $...= (insert\ x\ ?F) - \{x\}$ **using** *surj-indexing* **unfolding** *indexing-ext-def* **by** *auto*
also have $...=?F$ **using** *x-notin-F* **by** *auto*
finally show $?F= (indexing\text{-}ext\ F\ x\ n)$ ‘ $\{..<n\} \cup (indexing\text{-}ext\ F\ x\ n)$ ‘ $\{n<..<card\ (insert\ x\ ?F)\}$ **by** *auto*
qed
have *card-insert-suc-eq*: $card\ (insert\ x\ (?F))-Suc\ 0=card\ (?F)$
using *card-insert-if* **and** *x-notin-F* **and** *finite-iset-to-set-F* **by** *auto*
have *desc1*: $(indexing\text{-}ext\ F\ x\ n)$ ‘ $\{..<n\}=?h$‘ $\{..<n\}$ **unfolding** *indexing-ext-def* **by** *auto*
have *desc2*: $(indexing\text{-}ext\ F\ x\ n)$ ‘ $\{n<..<card\ (insert\ x\ ?F)\}= ?h$ ‘$\{i.\ n\leq i \wedge i<card\ (insert\ x\ (?F))-Suc\ 0\}$
unfolding *indexing-ext-def image-def Pi-def* **apply** *auto*
proof −
show $\bigwedge xa.\ [\![n < xa; xa < card\ (insert\ x\ (fst\ F))]\!]$
$\Longrightarrow \exists xb\geq n.\ xb < card\ (insert\ x\ (fst\ F)) - Suc\ 0 \wedge snd\ F\ (xa - Suc\ 0) = snd\ F\ xb$
proof −
fix *xa*
assume *n-l-xa*: $n < xa$ **and** *xa-l-card-xF*: $xa < card\ (insert\ x\ (fst\ F))$
show $\exists xb\geq n.\ xb < card\ (insert\ x\ (fst\ F)) - Suc\ 0 \wedge snd\ F\ (xa - Suc\ 0) = snd\ F\ xb$
proof −
let $?xb=xa-Suc\ 0$
have $xa>0$ **using** *n1* **and** *n-l-xa* **by** *auto*
hence *1*:$?xb < card\ (insert\ x\ (fst\ F)) - Suc\ 0$ **using** *xa-l-card-xF* **by** *auto*
have *2*:$snd\ F\ (xa - Suc\ 0) = snd\ F\ ?xb$ **..**
have *3*: $?xb\geq n$ **using** *n-l-xa* **by** *auto*
show *?thesis* **using** *1* **and** *2* **and** *3* **by** *auto*
qed qed

show $\bigwedge xa.\ [\![n \leq xa;\ xa < card\ (insert\ x\ (fst\ F)) - Suc\ 0]\!]$
$\Longrightarrow \exists x \in \{n<..<card\ (insert\ x\ (fst\ F))\}.\ snd\ F\ xa = snd\ F\ (x - Suc\ 0)$
proof −
fix *xa*
assume *n-le-xa*: $n \leq xa$ **and** *xa-l-card-xF-suc*: $xa < card\ (insert\ x\ (fst\ F)) - Suc\ 0$
show $\exists x \in \{n<..<card\ (insert\ x\ (fst\ F))\}.\ snd\ F\ xa = snd\ F\ (x - Suc\ 0)$
proof (*rule bexI*[*of - xa+Suc 0*])
show $snd\ F\ xa = snd\ F\ (xa + Suc\ 0 - Suc\ 0)$ **by** *auto*
show $xa + Suc\ 0 \in \{n<..<card\ (insert\ x\ (fst\ F))\}$ **using** *n-le-xa* **and** *xa-l-card-xF-suc* **by** *auto*
qed qed qed
have $?h$ ‘ $\{..<card\ ?F\} = ?h`\{..<n\} \cup ?h`\{i.\ n \leq i \wedge i < card\ ?F\}$ **using** *n2* **by** *force*
also have ...=(*indexing-ext F x n*) ‘ $\{..<n\}$ ∪ (*indexing-ext F x n*) ‘ $\{n<..<card\ (insert\ x\ ?F)\}$
using *desc1* **and** *desc2* **and** *card-insert-suc-eq* **by** *auto*
also have ...=?*F* **using** *F-indexing-ext-desc* **by** *simp*
finally show ?*thesis* **.**
qed
qed

From the above definitions we can define the operation insert for indexed sets. We don't assume that the new element (which is going to be inserted in the set) is not in the set, this will appear as a premise in the corresponding results.

Given any indexed set A, an element a and a position n, the operation *insert_iset* will introduce a in *iset_to_set* A in the position n (modifying accordingly the original indexation *iset_to_index* A).

definition *insert-iset* :: *'a iset => 'a => nat => 'a iset*
where
insert-iset A a n
= (*insert a* (*iset-to-set A*), *indexing-ext A a n*)

Next lemma claims that if we insert an element in an *indexing*, we are increasing the cardinality of the set in a unit. Logically, we need to assume that the element which is going to be inserted is not in the set.

lemma *insert-iset-increase-card*:
assumes *indexing-A*: *indexing* (*A,f*)
and *a-notin-A*: $a \notin A$
shows *card* (*iset-to-set* (*insert-iset* (*A,f*) *a n*)) = *card A* + *1*
by (*metis a-notin-A card.insert fst-conv indexing-A indexing-finite insert-iset-def iset-to-set-def nat-add-commute*)

Given an indexing (A, f), an element $a \notin A$ and a position $n \leq card(A)$, the result of inserting a in A in position n will be an indexing:

lemma *insert-iset-indexing*:

```
  assumes indexing-A: indexing (A,f)
  and a-notin-A: a ∉ A
  and n2: n ≤ (card A)
  shows indexing (insert-iset (A,f) a n)
proof (unfold indexing-def,unfold bij-betw-def, rule conjI)
  have finite-A: finite A using indexing-finite[OF indexing-A] .
  have card-insert: card (insert a A)=card A + 1
    using a-notin-A card-insert-if[OF finite-A] by force
  have descomposicion-conjunto:
    {..< card (insert a A)}={..<n}∪{n}∪{n<..<card (insert a A)}
    using n2
    by (metis Suc-eq-plus1 Un-commute Un-empty-right Un-insert-right
      atLeastLessThanSuc-atLeastAtMost
      atLeastSucAtMost-greaterThanAtMost
      atLeastSucLessThan-greaterThanLessThan card-insert
      ivl-disj-un(9) lessThan-Suc lessThan-Suc-atMost)
  show surj: iset-to-index (insert-iset (A, f) a n) `
    {..<card (iset-to-set (insert-iset (A, f) a n))}
    = iset-to-set (insert-iset (A, f) a n)
  proof (unfold insert-iset-def, simp)
    have ∀x∈{..<n}. indexing-ext (A, f) a n x = f x unfolding indexing-ext-def
by auto
   hence ind-1: indexing-ext (A, f) a n ` {..<n}= f `{..<n} unfolding image-def
by auto
    have ∀x∈{n<..< card (insert a A)}. indexing-ext (A, f) a n x = f (x − Suc
0) unfolding indexing-ext-def by auto
   hence ind-2: indexing-ext (A, f) a n ` {n<..< card (insert a A)} = f ` {i. n≤i
∧ i < card A} unfolding image-def
    proof (auto)
      show ⋀xa. ⟦∀x∈{n<..<card (insert a A)}. indexing-ext (A, f) a n x = f (x
− Suc 0); n < xa; xa < card (insert a A)⟧
        ⟹ ∃x≥n. x < card A ∧ f (xa − Suc 0) = f x
      proof −
        fix xa
        assume n-l-xa: n < xa and xa-l-cardAa: xa < card (insert a A)
        show ∃x≥n. x < card A ∧ f (xa − Suc 0) = f x
        proof −
          let ?x=xa − Suc 0
          have 1: ?x < card A using xa-l-cardAa using card-insert
            by (metis One-nat-def Suc-diff-1 Suc-eq-plus1 gr0I gr-implies-not0
              less-diff-conv less-irrefl-nat linorder-neqE-nat n-l-xa xt1(9))
          have 2: f (xa − Suc 0) = f ?x by simp
          have 3: ?x≥n using n-l-xa by simp
          show ?thesis using 1 and 2 and 3 by auto
        qed
      qed
      show ⋀xa. ⟦∀x∈{n<..<card (insert a A)}. indexing-ext (A, f) a n x = f (x
− Suc 0); n ≤ xa; xa < card A⟧
        ⟹ ∃x∈{n<..<card (insert a A)}. f xa = f (x − Suc 0)
```

proof −
fix *xa*
assume *n-le-xa*: $n \leq xa$ **and** *xa-l-cardA*: $xa < card\ A$
show $\exists x \in \{n<..<card\ (insert\ a\ A)\}.\ f\ xa = f\ (x - Suc\ 0)$
proof −
let $?x = xa + Suc\ 0$
have *1*: $f\ xa = f\ (?x - Suc\ 0)$ **by** *simp*
have *2*: $?x \in \{n<..<card\ (insert\ a\ A)\}$ **using** *n-le-xa* **and** *xa-l-cardA* *card-insert* **by** *auto*
show *?thesis* **using** *1* **and** *2* **by** *fast*
qed
qed
qed
have *desc-indexing*: *indexing-ext* (A, f) *a n* ‘ $\{..<n\} \cup$ *indexing-ext* (A, f) *a n* ‘ $\{n<..<card\ (insert\ a\ A)\}$
$= f$‘$\{..<card\ A\}$
using *ind-1* **and** *ind-2* **and** *n2* **by** *force*
show *indexing-ext* (A, f) *a n* ‘ $\{..<card\ (insert\ a\ A)\} = insert\ a\ A$
proof −
have *indexing-ext* (A, f) *a n* ‘ $\{..<card\ (insert\ a\ A)\}$
$=$ *indexing-ext* (A, f) *a n* ‘ $\{..<n\}$ $\cup$ *indexing-ext* (A, f) *a n* ‘ $\{n\}$
$\cup$ *indexing-ext* (A, f) *a n* ‘ $\{n<..<card\ (insert\ a\ A)\}$ **using** *descomposicion-conjunto* **by** *blast*
also have ...$= f$‘$\{..<card\ A\} \cup \{a\}$ **using** *desc-indexing* **unfolding** *indexing-ext-def* **by** *simp*
also have ...$=insert\ a\ A$ **using** *indexing-A* **unfolding** *indexing-def bij-betw-def a-notin-A* **by** *force*
finally show *?thesis* **.**
qed
qed
show *inj-on* (*iset-to-index* (*insert-iset* (A, f) *a n*))
$\{..<card$ (*iset-to-set* (*insert-iset* (A, f) *a n*))$\}$
proof (*rule eq-card-imp-inj-on*) — We need to have proved previously the injectivity
show *finite* $\{..<card$ (*iset-to-set* (*insert-iset* (A, f) *a n*))$\}$
unfolding *insert-iset-def* **by** *simp*
show *card* (*iset-to-index* (*insert-iset* (A, f) *a n*) ‘ $\{..<card$ (*iset-to-set* (*insert-iset* (A, f) *a n*))$\}$)
$=$ *card* $\{..<card$ (*iset-to-set* (*insert-iset* (A, f) *a n*))$\}$
using *surj* **by** *simp*
qed
qed

We introduce the definition of a generic function *remove-iset* which removes the *nth* element of an indexed set. Logically, the position of the element which is going to be removed must be less than the cardinality of the set. The indexing must be also modified in such a way that every element above n will decrease its position in one unit. For instance, if we have the indexed set $\{(a, 0), (b, 1), (c, 2)\}$ and we remove the position 0, we will obtain

$\{(b,0),(c,1)\}$.

definition *remove-iset* :: *'a iset* => *nat* => *'a iset*
  where *remove-iset A n* = (*fst A* − {(*snd A*) *n*},
  (λ*k*. *if k* < *n then* (*snd A*) *k* *else* (*snd A*) (*Suc k*)))

Here an equivalent definition to *remove-iset ?A ?n* = (*fst ?A* − {*snd ?A ?n*}, λ*k*. *if k* < *?n then snd ?A k else snd ?A* (*Suc k*)):

lemma *remove-iset-def'*:
  remove-iset (*A*, *f*) *n* = (*A* − {*f n*}, (λ*k*. *if k* < *n then f k* *else f* (*Suc k*)))
  unfolding *remove-iset-def* **by** (*auto simp add*: *fun-eq-iff*)

The following lemma proves that, for any indexing, the result of removing an element in a valid position will be again an indexing. This is a long lemma (about 150 lines).

lemma
  indexing-remove-iset:
  assumes *i*: *indexing* (*B*, *h*)
  and *n*: *n* < *card B*
  shows *indexing* (*remove-iset* (*B*, *h*) *n*)
proof (*unfold indexing-def bij-betw-def*, *intro conjI*, *simp*)
  have *fin-B*: *finite B* **using** *indexing-finite*[*OF i*] **.**
  have *h-n-in-B*: *h n* ∈ *B*
    using *n i* **unfolding** *indexing-def bij-betw-def* **by** *auto*
  have *eq-i*: ⋀*x y*. ⟦*x* ∈ {..<*card B*}; *y* ∈ {..<*card B*}; *h x* = *h y*⟧
    ⟹ *x* = *y*
    using *i* **unfolding** *indexing-def bij-betw-def inj-on-def*
    by *auto*
  show *inj-on* (*snd* (*remove-iset* (*B*, *h*) *n*))
    {..<*card* (*fst* (*remove-iset* (*B*, *h*) *n*))}
    unfolding *remove-iset-def*
    unfolding *inj-on-def*
  proof (*rule ballI*, *rule ballI*, *rule impI*, *unfold fst-conv snd-conv*)
    fix *x y*
    assume *x*: *x* ∈ {..<*card* (*B* − {*h n*})}
      and *y*: *y* ∈ {..<*card* (*B* − {*h n*})}
      and *eq*: (*if x* < *n then h x else h* (*Suc x*)) = (*if y* < *n then h y else h* (*Suc y*))
    show *x* = *y*
    proof (*cases x* < *n*)
      case *True* **note** *x-l-n* = *True*
      show *x* = *y*
      proof (*cases y* < *n*)
        case *True*
        show *x* = *y*
        proof (*rule eq-i*)
          show *x* ∈ {..<*card B*} **using** *x*
            by (*metis lessThan-iff less-or-eq-imp-le n order-le-less-trans x-l-n*)
          show *y* ∈ {..<*card B*} **using** *y*
            by (*metis lessThan-iff less-or-eq-imp-le n order-le-less-trans True*)

show *h x* = *h y* **using** *eq x-l-n True* **by** *simp*
qed
next
case *False*
have *x* ≠ (*Suc y*) **using** *x-l-n False* **by** *auto*
moreover have *x* = (*Suc y*)
proof (*rule eq-i*)
show *x* ∈ {..<*card B*} **using** *x*
by (*metis lessThan-iff less-or-eq-imp-le n order-le-less-trans x-l-n*)
show (*Suc y*) ∈ {..<*card B*} **using** *y* **using** *h-n-in-B*
by (*metis Suc-eq-plus1* ‹*x* ∈ {..<*card B*}› *card-Diff-singleton card-infinite*

emptyE lessThan-0 lessThan-iff less-diff-conv)
show *h x* = *h* (*Suc y*) **using** *eq x-l-n False* **by** *simp*
qed
ultimately have *False* **by** *contradiction*
thus *?thesis* **by** *fast*
qed
next
case *False* **hence** *n-le-x*: *n* ≤ *x* **by** *arith*
show *x* = *y*
proof (*cases y* < *n*)
case *True*
have *x-ne-y*: (*Suc x*) ≠ *y* **using** *n-le-x True* **by** *auto*
moreover have (*Suc x*) = *y*
proof (*rule eq-i*)
show *y* ∈ {..<*card B*} **using** *y*
by (*metis lessThan-iff less-or-eq-imp-le n order-le-less-trans True*)
show (*Suc x*) ∈ {..<*card B*} **using** *x* **using** *h-n-in-B*
by (*metis Suc-eq-plus1* ‹*y* ∈ {..<*card B*}› *card-Diff-singleton card-infinite*

emptyE lessThan-0 lessThan-iff less-diff-conv)
show *h* (*Suc x*) = *h y* **using** *eq True n-le-x* **by** *simp*
qed
ultimately have *False* **by** *contradiction*
thus *x* = *y* **by** *fast*
next
case *False*
have *Suc x* = *Suc y*
proof (*rule eq-i*)
show *Suc x* ∈ {..<*card B*}
using *x* **using** *card-Diff1-less* [*OF fin-B h-n-in-B*] **using** *h-n-in-B*
by (*metis Suc-eq-plus1 card-Diff-singleton fin-B lessThan-iff less-diff-conv*)
show *Suc y* ∈ {..<*card B*}
using *y* **using** *card-Diff1-less* [*OF fin-B h-n-in-B*] **using** *h-n-in-B*
by (*metis Suc-eq-plus1 card-Diff-singleton fin-B lessThan-iff less-diff-conv*)
show *h* (*Suc x*) = *h* (*Suc y*)
using *eq* **using** *False* **using** *n-le-x* **by** *simp*
qed

thus $x = y$ **by** *simp*
qed
qed
qed
have *h-im*: h ‘ $\{..<card\ B\} = B$ **using** *i* **unfolding** *indexing-def bij-betw-def* **by** *auto*
show *iset-to-index* (*remove-iset* (B, h) n) ‘
$\{..<card$ (*iset-to-set* (*remove-iset* (B, h) n))$\}$
= *iset-to-set* (*remove-iset* (B, h) n)
proof (*unfold remove-iset-def iset-to-index-def iset-to-set-def fst-conv snd-conv*)
show (λk. *if* $k < n$ *then* $h\ k$ *else* h (*Suc* k)) ‘ $\{..<card\ (B - \{h\ n\})\} = B - \{h\ n\}$
(**is** $?h'$ ‘ $\{..<card\ (B - \{h\ n\})\} = B - \{h\ n\}$)
proof –
have $B - \{h\ n\} = h$ ‘ $(\{..<card\ B\} - \{n\})$
using *bij-betw-image-minus* [*symmetric*, *of* h $\{..<card\ B\}$ B n]
using *n* **using** *i* **unfolding** *indexing-def bij-betw-def* **by** *simp*
also have ... $= h$ ‘ $(\{..<n\} \cup \{n<..<card\ B\})$ **using** *n* **by** *auto*
also have ... $= h$ ‘ $\{..<n\} \cup h$ ‘ $\{n<..<card\ B\}$ **unfolding** *image-Un* **..**
also have ... $= ?h'$ ‘ $\{..<n\} \cup h$ ‘ $\{n<..<card\ B\}$ **by** *auto*
also have ... $= ?h'$ ‘ $\{..<n\} \cup ?h'$ ‘ $\{n..<card\ (B - \{h\ n\})\}$
proof –
have $?h'$ ‘ $\{n..<card\ (B - \{h\ n\})\} = h$ ‘ $\{n<..<card\ B\}$
unfolding *image-def* **using** *fin-B h-n-in-B*
proof (*auto*, *force*)
fix *xa*
assume *n*: $n < xa$ **and** *xa*: $xa < card\ B$
hence *xa-n-0*: $0 < xa$ **by** *simp*
show $\exists x \in \{n..<card\ B - Suc\ 0\}$. $h\ xa = h\ (Suc\ x)$
apply (*rule bexI* [*of* - $xa - 1$])
apply (*metis Suc-diff-1 xa-n-0*)
using *n xa xa-n-0* **by** *force*
qed
thus *?thesis* **by** *fast*
qed
also have ... $= ?h'$ ‘ $(\{..<n\} \cup \{n..<card\ (B - \{h\ n\})\})$
by (*rule image-Un* [*symmetric*, *of* $?h'$ $\{..<n\}$ $\{n..<card\ (B - \{h\ n\})\}$])
also have ... $= ?h'$ ‘ $\{..<card\ (B - \{h\ n\})\}$ **using** *n* **using** *fin-B h-n-in-B* **by** *auto*
finally show *?thesis* **by** *simp*
qed
qed
qed

The result of inserting an element in an indexed set in position n and then removing the element in position n is the original indexed set.

lemma
remove-iset-insert-iset-id:
assumes *x-notin-A*: $x \notin A$

and *n-l-c*: $n < card\ A$
shows *remove-iset* $(insert\text{-}iset\ (A, f)\ x\ n)\ n = (A, f)$
unfolding *insert-iset-def*
using *x-notin-A*
unfolding *indexing-ext-def*
unfolding *remove-iset-def* **by** (*auto simp add*: *fun-eq-iff n-l-c*)

Next lemma is a good example of proof by acumulation of facts, and it is ideal to structure it using *moreover* and finish it with *ultimately*. However, we can use $[\![A;\ B;\ C;\ D]\!] \Longrightarrow A \wedge B \wedge C \wedge D$ to abridge it:

The lemma claims that given an indexing (X, f), there exists an indexing $(insert\ x\ X, h)$ which places x in the last position (and keeps the elements of X in their original places).

lemma *indexation-x-union-X*:
assumes *finite*: *finite X* **and** *x-not-in-X*: $x \notin X$
and *f-buena*:$f \in \{i.\ i < (card\ X)\} \rightarrow X$ **and** *ordenFX*: $f\ `\ \{i.\ i < (card\ X)\} =$
X
shows $\exists h.\ (h \in \{i.\ i < (card\ (insert\ x\ X))\} \rightarrow (insert\ x\ X)$
$\wedge\ h`\{i.\ i < (card\ (insert\ x\ X))\} = (insert\ x\ X)$
$\wedge\ h\ (card\ X) = x \wedge (\forall i.\ i<card(X) \longrightarrow h\ i = f\ i))$
proof (*rule exI* [*of* - (λi::*nat*. *if* $i<(card\ X)$ *then* $f(i)$ *else* x)], *rule conjI4*)
let *?h* = (λi::*nat*. *if* $i<(card\ X)$ *then* $f(i)$ *else* x)
show *?h* $\in \{i.\ i < card\ (insert\ x\ X)\} \rightarrow insert\ x\ X$
using *f-buena* **unfolding** *Pi-def* **by** *auto*
show *?h* ` $\{i.\ i < card\ (insert\ x\ X)\} = insert\ x\ X$
using *ordenFX*
unfolding *card-insert-disjoint* [*OF finite x-not-in-X*]
unfolding *less-than-Suc-union*
unfolding *image-Un* **by** *auto*
show (*if* $card\ X < card\ X$ *then* $f\ (card\ X)$ *else* x) $= x$ **by** *simp*
show ($\forall i<card\ X.$ (*if* $i < card\ X$ *then* $f\ i$ *else* x) $= f\ i$) **by** *simp*
qed

This is an indispensable lemma to prove the theorem that claims that an independent set can be completed to a basis. Given any pair of (disjoint) sets A and B, there exists an indexing function h which places the elements of A in the first $card(A)$ positions and then the elements of B. In the proof, the indexing function is explicitly provided:

lemma *indexing-union*:
assumes *disjuntos*: $A \cap B = \{\}$
and *finite-A*: *finite A*
and *A-not-empty*: $A \neq \{\}$ — If not the result is trivial.
and *finite-B*: *finite B*
shows $\exists h.\ indexing\ (A \cup B, h) \wedge h`\ \{..<card(A)\} = A$
$\wedge\ h`\ (\{..<(card(A)+card(B))\} - \{..<card(A)\}) = B$
proof −
have $\exists f.\ indexing\ (A, f)$ **using** *obtain-indexing*[*OF finite-A*] **.**

```
  from this obtain f where indexing-A-f: indexing (A,f) by auto
  have ∃ g. indexing (B,g) using obtain-indexing[OF finite-B] .
  from this obtain g where indexing-B-g:  indexing (B,g) by auto
  show ?thesis
  proof (rule exI[of - (λx. if x∈ {..<card(A)}
      then f(x) else g(x−card(A)))])
    let ?h=(λx. if x∈ {..<card(A)} then f(x) else g(x−card(A)))
    have ∀ x∈{..<card(A)}. f(x)= ?h (x) by simp
    hence surj-h-A: ?h‘ {..<card(A)}=A
      using indexing-A-f unfolding indexing-def bij-betw-def by auto
      have ∀ x∈{..<(card(A)+card(B))}−{..<card(A)}. g(x−card(A))=?h(x) by
auto
    hence ?h‘ ({..<(card(A)+card(B))}−{..<card(A)}) = g‘{..<card(B)}
      unfolding image-def
    proof (auto)
      fix xa
      assume xa-le-cardB: xa < card B
      show ∃ x∈{..<card A + card B} − {..<card A}. g xa = g (x − card A)
      proof (rule bexI[of - xa+card(A)])
        have cardA-not-zero: card A ≠ 0 using A-not-empty finite-A by auto
        thus g xa = g (xa + card A − card A) by auto
        show  xa + card A ∈ {..<card A + card B} − {..<card A}
            by (metis DiffI diff-add-inverse lessThan-iff less-diff-conv not-add-less2
xa-le-cardB)
      qed
    qed
    hence surj-h-B: ?h‘ ({..<(card(A)+card(B))}−{..<card(A)})=B
      using indexing-B-g unfolding indexing-def and bij-betw-def by auto
    have indexing: indexing (A∪B,?h)
    proof (unfold indexing-def, simp,unfold  bij-betw-def)
      have 1: ?h ‘ {..<card (A ∪ B)} = A ∪ B
      proof −
        have card (A∪B)=card(A)+card(B) using disjuntos and finite-A finite-B
card-Un-disjoint by auto
        hence {..<card (A ∪ B)}={..<card(A)+card(B)} by simp
        also have ...={..<card(A)} ∪ ({..<(card(A)+card(B))}−{..<card(A)}) by
auto
      finally have ?h‘ {..<card (A ∪ B)} = ?h ‘ {..<card(A)} ∪ ?h ‘ ({..<(card(A)+card(B))}−{..<card(A)})
by force
        thus ?thesis using surj-h-B and surj-h-A by auto
      qed
      have 2: inj-on ?h {..<card (A ∪ B)}
      proof (rule eq-card-imp-inj-on)
        show finite {..<card (A ∪ B)} using finite-A and finite-B by auto
        show  card (?h ‘ {..<card (A ∪ B)}) = card {..<card (A ∪ B)}
          using 1 and card-lessThan by auto
      qed
      show inj-on (λx. if x < card A then f x else g (x − card A))
        {..<card (A ∪ B)} ∧
```

$(\lambda x.\ \mathit{if}\ x < \mathit{card}\ A\ \mathit{then}\ f\ x\ \mathit{else}\ g\ (x - \mathit{card}\ A))$ ‘ $\{..<\mathit{card}\ (A \cup B)\} =$
$A \cup B$
using *1* **and** *2* **by** *auto*
qed
show $\mathit{indexing}(A \cup B, ?h) \wedge$
$?h$ ‘$\{..<\mathit{card}\ A\} = A \wedge$
$?h$ ‘$(\{..<\mathit{card}\ A + \mathit{card}\ B\} - \{..<\mathit{card}\ A\}) =$
B **using** *indexing surj-h-A surj-h-B* **by** *auto*
qed
qed

Now we are going to define a new function which returns the position where an element a is in a set A. When we use this function it is very important to assume that $a \in A$, since functions are total in HOL, and without the premise $a \in A$ we would obtain an undefined value of the righ type. An alternative definition could be made writing LEAST instead of THE and then we could remove $n < \mathit{card}\ A$. Note that both THE and LEAST are based on the Hilbert's ϵ operator, which, in general, places us out of a constructive setting.

This function will be very important for the proof that each basis of a vector space has the same cardinality.

definition *obtain-position* :: $'c \Rightarrow 'c\ \mathit{iset} \Rightarrow \mathit{nat}$
where *obtain-position* $a\ A = (\mathit{THE}\ n.\ (\mathit{snd}\ A)\ n = a$
$\wedge\ n < \mathit{card}\ (\mathit{fst}\ A))$

Under the right premises, this natural number exists and is smaller than $card(A)$ which ensures that *obtain-position* is well-defined.

lemma *exists-n-obtain-position*:
assumes *a-in-A*: $a \in A$
and *indexing-A*: *indexing* (A,f)
shows $\exists\, n::\mathit{nat}.\ f\ n = a$
proof −
have $A \neq \{\}$ **using** *a-in-A* **by** *blast*
hence *cardA-g-0*: $\mathit{card}\ A > 0$ **using** *card-gt-0-iff* **and** *indexing-finite*[*OF indexing-A*]
by *blast*
thus *?thesis* **using** *a-in-A indexing-A* **unfolding** *indexing-def bij-betw-def* **by** *force*
qed

We proof that exists someone that also verifies $n < \mathit{card}\ A$

lemma *exists-n-and-less-card-obtain-position*:
assumes *a-in-A*: $a \in A$
and *indexing-A*: *indexing* (A,f)
shows $\exists\, n::\mathit{nat}.\ f\ n = a \wedge n < (\mathit{card}\ A)$
proof −
have $A \neq \{\}$ **using** *a-in-A* **by** *blast*
hence *cardA-g-0*: $\mathit{card}\ A > 0$

using *card-gt-0-iff* **and** *indexing-finite*[*OF indexing-A*] **by** *blast*
thus *?thesis* **using** *a-in-A indexing-A* **unfolding** *indexing-def bij-betw-def* **by** *force*
qed

Thanks to the previous lemma and the injectivity of indexing functions, we can prove the existence and the unicity of *obtain-position*:

```
lemma exists-n-and-is-unique-obtain-position:
  assumes a-in-A: a∈ A
  and indexing-A: indexing (A,f)
  shows ∃!n::nat. f n = a ∧ n < (card A)
proof (rule ex-ex1I)
  show ∃ n. f n = a ∧ n < card A
    using exists-n-and-less-card-obtain-position
    [OF a-in-A indexing-A] .
  show ⋀n y. ⟦f n = a ∧ n < card A; f y = a ∧ y < card A⟧
    ⟹ n = y
  proof −
    fix n and y
    assume hip-n: f n = a ∧ n < card A
      and hip-y: f y = a ∧ y < card A
    show n=y
    proof −
      have inj-on: inj-on f {..<card A}
        using indexing-A unfolding indexing-def bij-betw-def by simp
      show ?thesis using inj-on-eq-iff[OF inj-on - - ] using hip-n hip-y by auto
    qed
  qed
qed
```

Now that we have proved that *obtain-position* is well defined, we prove that its result satisfies the required properties. The number which is returned by *obtain-position* is less than the cardinal of the set:

```
lemma obtain-position-less-card:
  assumes a-in-A: a ∈ A
  and indexing-A: indexing (A,f)
  shows (obtain-position a (A,f)) < card A
proof (unfold obtain-position-def)
  let ?P = (λn. f n = a ∧ n < card A)
  have exK: (∃!k. ?P k)
    using exists-n-and-is-unique-obtain-position[OF a-in-A indexing-A] .
  have ex-THE: ?P (THE k. ?P k)
    using theI' [OF exK] .
  def n≡(THE k. ?P k)
  have n < card A unfolding n-def
    by (metis ex-THE)
  thus (THE n. snd (A, f) n = a ∧ n < card (fst (A, f))) < card A
    by (metis ex-THE fst-conv n-def snd-conv)
qed
```

The function really returns the position of the element.

lemma *obtain-position-element*:
  assumes *a-in-A*: $a \in A$
  and *indexing-A*: *indexing* (A,f)
  shows f (*obtain-position* a $(A,f)) = a$
proof (*unfold obtain-position-def*)
  let $?P = (\lambda n.\ f\ n = a \wedge n < card\ A)$
  have *exK*: $(\exists!k.\ ?P\ k)$
    using *exists-n-and-is-unique-obtain-position*[*OF a-in-A indexing-A*] **.**
  have *ex-THE*: $?P$ (*THE* $k.\ ?P\ k$)
    using *theI'* [*OF exK*] **.**
  def $n \equiv$(*THE* $k.\ ?P\ k$)
  have $f\ n = a$ **unfolding** *n-def*
    by (*metis ex-THE*)
  thus f (*THE* $n.$ *snd* $(A, f)\ n = a \wedge n < card$ (*fst* $(A, f))) = a$
    by (*metis ex-THE fst-conv n-def snd-conv*)
qed

An element will not be in the set returned by the function *remove-iset* called with the position of that element.

lemma *a-notin-remove-iset*:
  assumes *a-in-A*: $a \in A$
  and *indexing-A*: *indexing* (A,f)
  shows $a \notin$ *fst* (*remove-iset* (A,f) (*obtain-position* a (A,f)))
  unfolding *remove-iset-def*
  using *obtain-position-element*[*OF a-in-A indexing-A*] **by** *simp*

Finally some important theorems to prove future properties of indexed sets. Isabelle has an induction rule to prove properties of finite sets. Unfortunately, this rule is of little help for proving properties of indexed sets, since the set and the indexing function must behave accordingly in the induction rule, and their inherent properties. Consequently, we have to introduce a special induction rule for indexed sets.

First an auxiliary lemma:

lemma *exists-indexing-ext*:
  assumes *i*: *indexing* (*insert* x A, f)
  shows $\exists h.\ \exists n \in \{..card\ A\}.\ (f = (indexing\text{-}ext\ (A,\ h)\ x)\ n)$
proof –
  have *x-in-insert*: $x \in$ (*insert* x A) **by** *simp*
  from *i* **obtain** n **where** *n-less-card-insert*: $n < card$ (*insert* x A)
    and *fn-x*: $f\ n = x$ **using** *obtain-position-less-card*[*OF x-in-insert i*]
    and *obtain-position-element*[*OF x-in-insert i*]
    by *blast*
  show *?thesis*
  proof (*rule exI*, *rule bexI*[*of - n*])
    have *finite* (*insert* x A) **using** *indexing-finite*[*OF i*] **.**
    thus $n \in \{..card\ A\}$ **using** *n-less-card-insert* **and** *x-in-insert*

by (*metis atMost-iff card-insert-disjoint*
finite-insert insert-absorb less-or-eq-imp-le linorder-not-le not-less-eq-eq)
def $h \equiv (\lambda x.\ \textit{if}\ x < n\ \textit{then}\ f\ x\ \textit{else}\ f\ (x + \mathit{1}))$
show $f = \textit{indexing-ext}\ (A, h)\ x\ n$
unfolding *indexing-ext-def* **unfolding** *h-def fun-eq-iff*
using *n-less-card-insert fn-x*
by *fastsimp*
qed
qed

The first one induction rule:

theorem
indexed-set-induct:
assumes *indexing* (A, f)
and *finite* A
and $!!f.$ *indexing* $(\{\}, f) ==> P\ \{\}\ f$
and *step*: $!!a\ A\ f\ n.$ $[|a \notin A;$ *finite* A; *indexing* (A, f);
$\mathit{0} \leq n;\ n \leq$ *card* $A|] ==> P$ (*insert* a A) ((*indexing-ext* (A, f) a) n)
shows $P\ A\ f$
using ‹*finite* A› **and** ‹*indexing* (A, f)›
proof (*induct arbitrary*: f)
case *empty*
show *?case* **using** *empty* (*1*) **by** *fact*
next
case (*insert* x F h')
show *?case*
proof $-$
obtain $h\ n$
where h'-*def*: $h' =$ (*indexing-ext* (F, h) x) n
and *n1*: $\mathit{0} \leq n$
and *n2*: $n \leq$ *card* F
using *exists-indexing-ext*[*OF insert.prems*] **by** *blast*
show *?case*
unfolding h'-*def*
proof (*rule step*)
show $x \notin F$ **by** *fact*
show *finite* F **by** *fact*
show *indexing* (F, h)
apply (*rule indexing-indexing-ext* [*of* x - n])
using *insert.prems* **unfolding** h'-*def* **apply** *simp*
unfolding *iset-to-set-def fst-conv* **by** *fact+*
show $\mathit{0} \leq n$ **using** *n1* **.**
show $n \leq$ *card* F **using** *n2* **.**
qed
qed
qed

This induction rule is similar to the proper of finite sets, $[\![$*finite* F; $P\ \{\}$; $\bigwedge x\ F.\ [\![$*finite* F; $x \notin F$; $P\ F]\!] \Longrightarrow P$ (*insert* x F)$]\!] \Longrightarrow P\ F$, but taking into account the indexing. Thus, if a property P holds for the empty set and one of its indexing functions, and when it holds for a given set A and an indexing function f, we now how to prove it for the pair *insert a A* (with $a \notin A$) and any of the extensions of f, then P holds for every indexing (A, f). The proof of the property is completed by induction over the set A, but keeping f free for later instantiation with the right indexing functions.

```
lemma
  indexed-set-induct2 [case-names indexing finite empty insert]:
  assumes indexing (A, f)
  and finite A
  and !!f. indexing ({}, f) ==> P {} f
  and step: !!a A f n. [|a ∉ A;
          [| indexing (A, f) |] ==> P A f;
          finite (insert a A);
          indexing ((insert a A), (indexing-ext (A, f) a n));
          0 ≤ n; n ≤ card A |] ==>
          P (insert a A) (indexing-ext (A, f) a n)
  shows P A f
  using ‹finite A› and ‹indexing (A, f)›
proof (induct arbitrary: f)
  case empty
  show ?case using empty (1) by fact
next
  case (insert x F h')
  show ?case
  proof -
    obtain n h
      where h'-def: h' = (indexing-ext (F, h) x) n
      and n1: 0 ≤ n
      and n2: n ≤ card F using exists-indexing-ext[OF insert.prems] by blast
    show ?case
      unfolding h'-def
    proof (rule step)
      show x ∉ F by fact
      have i-F-h: indexing (F, h)
        apply (rule indexing-indexing-ext [of x (F, h) n])
        using insert.prems unfolding h'-def
        using n1 n2 insert.hyps (2) by simp-all
      show P F h by (rule insert.hyps (3)) (rule i-F-h)
      show 0 ≤ n using n1 .
      show n ≤ card F using n2 .
      show finite (insert x F) using insert.hyps (1) by simp
      show indexing (insert x F, indexing-ext (F, h) x n)
        using insert.prems unfolding h'-def .
    qed
  qed
qed
```

end

theory *Linear-combinations*
imports *Linear-dependence Indexed-Set*
begin

8 Linear combinations

context *vector-space*
begin

To define the notion of linear dependence and independence we already introduced the definition of linear combination. Nevertheless, here we present some properties of linear combinations. We could have used them to simplify the proofs of some theorems in the previous section, but we have decided to keep the order of the sections in Halmos.

A *linear-combination* is closed, when considering a set $X \subseteq$ *carrier V* and a proper coefficients function f:

lemma *linear-combination-closed*:
  assumes *good-set*: *good-set X*
  and *f*: $f \in$ *coefficients-function* (*carrier V*)
  shows *linear-combination f X* $\in$ *carrier V*
proof (*unfold linear-combination-def*, *rule finsum-closed*)
  show *finite X* **using** *good-set* **unfolding** *good-set-def* **by** *auto*
  show $(\lambda y.\ f\ y \cdot y) \in X \to$ *carrier V*
  proof (*unfold Pi-def*, *auto*)
    fix *y*
    assume *y-in-X*: $y \in X$
    hence *y-in-V*: $y \in$ *carrier V* **using** *good-set* **unfolding** *good-set-def* **by** *fast*
    show $f\ y \cdot y \in$ *carrier V* **using** *fx-x-in-V*[*OF y-in-V f*] **.**
  qed
qed

A *linear-combination* over the empty set is equal to $\mathbf{0}_V$

lemma *linear-combination-of-zero*:
  shows *linear-combination f* $\{\} = x \longleftrightarrow x = \mathbf{0}_V$
proof
  assume *l-combination-x*: *linear-combination f* $\{\} = x$
  have *l-combination-zero*: *linear-combination f* $\{\}=\mathbf{0}_V$
    unfolding *linear-combination-def*
    using *finsum-empty* **by** *auto*
  show $x = \mathbf{0}_V$
    using *l-combination-x* **and** *l-combination-zero* **by** *auto*
next
  assume *x-zero*: $x = \mathbf{0}_V$
  have *l-combination-x*: *linear-combination f* $\{\} = \mathbf{0}_V$
    unfolding *linear-combination-def*

    using *finsum-empty* **by** *auto*
  show *linear-combination f* {}=*x*
    using *l-combination-x* **and** *x-zero* **by** *simp*
qed

From previous lemma we can obtain a corollary which will be useful as a simplification rule.

corollary *linear-combination-empty-set* [*simp*]:
  shows *linear-combination f* {} = $\mathbf{0}_V$
  using *linear-combination-of-zero* **by** *simp*

The computation of the linear combination of a unipuntual set is direct:

lemma *linear-combination-singleton*:
  assumes *cf-f*: $f \in$ *coefficients-function* (*carrier V*)
  and *x-in-V*: $x \in$ *carrier V*
  shows *linear-combination f* $\{x\} = f\ x \cdot x$
proof −
  have *linear-combination f* (*insert x* {})
    = $(f\ x) \cdot x \oplus_V$ *linear-combination f* {}
  proof (*unfold linear-combination-def*, *rule finsum-insert*)
    show *finite* {} **by** *simp*
    show $x \notin$ {} **by** *simp*
    show $(\lambda y.\ f\ y \cdot y) \in \{\} \to$ *carrier V* **by** *simp*
    show $f\ x \cdot x \in$ *carrier V*
    proof (*rule mult-closed*)
      show $x \in$ *carrier V* **using** *x-in-V* **.**
      show $f\ x \in$ *carrier K* **using** *cf-f*
        unfolding *coefficients-function-def* **using** *x-in-V* **by** *auto*
    qed
  qed
  also have ... = $(f\ x) \cdot x \oplus_V \mathbf{0}_V$
    using *linear-combination-empty-set* **by** *auto*
  also have ... = $(f\ x) \cdot x$
  proof (*rule V.r-zero*)
    show $f\ x \cdot x \in$ *carrier V*
    proof (*rule mult-closed*)
      show $x \in$ *carrier V* **using** *x-in-V* **.**
      show $f\ x \in$ *carrier K*
        using *cf-f*
        unfolding *coefficients-function-def* **using** *x-in-V* **by** *auto*
    qed
  qed
  finally show *?thesis* **by** *auto*
qed

A *linear-combination* of *insert x X* is equal to $f\ x \cdot x \oplus_V$ *linear-combination f X*

lemma *linear-combination-insert*:

assumes *good-set-X*: *good-set X*
and *x-in-V*: $x \in$ *carrier V*
and *x-not-in-X*: $x \notin X$
and *cf-f*: $f \in$ *coefficients-function* (*carrier V*)
shows *linear-combination f* (*insert x X*)
$= f\ x \cdot x \oplus_V$ *linear-combination f X*
proof (*unfold linear-combination-def*, *rule finsum-insert*)
show *finite X* **using** *good-set-X*
unfolding *good-set-def* **by** *simp*
show $x \notin X$ **using** *x-not-in-X* **.**
show $(\lambda y.\ f\ y \cdot y) \in X \to$ *carrier V*
proof (*unfold Pi-def*,*auto*)
show $\bigwedge x.\ x \in X \Longrightarrow f\ x \cdot x \in$ *carrier V*
proof (*rule fx-x-in-V*)
fix *y*
assume *y-in-X*: $y \in X$
show $y \in$ *carrier V*
using *good-set-X*
unfolding *good-set-def* **using** *y-in-X* **by** *auto*
show $f \in$ *coefficients-function* (*carrier V*) **using** *cf-f* **.**
qed
qed
show $f\ x \cdot x \in$ *carrier V* **using** *fx-x-in-V*[*OF x-in-V cf-f*] **.**
qed

If each term of the linear combination is zero, then the sum is zero.

lemma *linear-combination-zero*:
assumes *good-set-X*: *good-set X*
and *cf-f*: $f \in$ *coefficients-function* (*carrier V*)
and *all-zero*: $\bigwedge x.\ x \in X \Longrightarrow f\ (x) \cdot x = \mathbf{0}_V$
shows *linear-combination f X* $= \mathbf{0}_V$
proof −
have *linear-combination f X* $= (\bigoplus{}_V y{\in}X.\ f\ y \cdot y)$
unfolding *linear-combination-def* **..**
also have ...$=(\bigoplus{}_V y{\in}X.\ \mathbf{0}_V)$
proof (*rule finsum-cong′*,*auto*)
fix *x*
assume *x-in-X*: $x{\in}X$
show $f\ x \cdot x = \mathbf{0}_V$
using *all-zero*[*OF x-in-X*] **.**
qed
also have ...$=\mathbf{0}_V$ **using** *finsum-zero good-set-X*
unfolding *good-set-def* **by** *blast*
finally show *?thesis* **.**
qed

This is an auxiliary lemma which we will use later to prove that $a \cdot$ *linear-combination f X* = *linear-combination* $(\lambda i.\ a \otimes f\ i)\ X$. We prove it doing induction over the finite set X. Firstly, we have to prove the property in case that the set is empty. After that, we suppose that the result is true for a set X and then we have to prove it for a set *insert* x X where $x \notin X$.

lemma *finsum-aux*:
  $[\![$*finite* X; $X \subseteq$ *carrier* V; $a \in$ *carrier* K; $f \in X \to$ *carrier* $K]\!]$
  $\Longrightarrow a \cdot (\bigoplus_V y \in X.\ f\ y \cdot y) = (\bigoplus_V y \in X.\ a \cdot (f\ y \cdot y))$
proof (*induct set*: *finite*)
  case *empty* **then show** *?case*
    using *scalar-mult-zeroV-is-zeroV* **by** *auto*
next
  case (*insert* x X) **then show** *?case*
  proof −
    have *sum-closed*: $(\bigoplus_V y \in X.\ f\ y \cdot y) \in$ *carrier* V
    proof (*rule finsum-closed*)
      show *finite* X **using** *insert.hyps* (*1*) **.**
      show $(\lambda y.\ f\ y \cdot y) \in X \to$ *carrier* V
        using *insert.prems* (*1*) **and** *insert.prems* (*3*) **and** *mult-closed*
        by *auto*
    qed
    have *fx-x-in-V*: $f\ x \cdot x \in$ *carrier* V
      using *insert.prems* (*1*) **and** *insert.prems* (*3*) **and** *mult-closed*
      by *auto*
    have $(\bigoplus_V y \in$ *insert* x $X.\ f\ y \cdot y) = f(x) \cdot x \oplus_V (\bigoplus_V y \in X.\ f\ y \cdot y)$
    proof (*rule finsum-insert*)
      show *finite* X **using** *insert.hyps* (*1*) **.**
      show $x \notin X$ **using** *insert.hyps* (*2*) **.**
      show $f\ x \cdot x \in$ *carrier* V **using** *fx-x-in-V* **.**
      show $(\lambda y.\ f\ y \cdot y) \in X \to$ *carrier* V
        using *insert.prems* (*1*) **and** *insert.prems* (*3*) **and** *mult-closed*
        by *auto*
    qed
    hence $a \cdot (\bigoplus_V y \in$ *insert* x $X.\ f\ y \cdot y) = a \cdot f(x) \cdot x \oplus_V a \cdot (\bigoplus_V y \in X.\ f\ y \cdot y)$
      using *add-mult-distrib1*[*OF fx-x-in-V*
        sum-closed insert.prems(*2*)] **by** *auto*
    also have $\ldots = a \cdot f(x) \cdot x \oplus_V (\bigoplus_V y \in X.\ a \cdot f\ y \cdot y)$
    proof −
      have *X-subset-V*: $X \subseteq$ *carrier* V**using** *insert.prems*(*1*) **by** *auto*
      have *f1*: $f \in X \to$*carrier* K **using** *insert.prems*(*3*) **by** *auto*
       show *?thesis* **using** *insert.hyps*(*3*)[*OF X-subset-V insert.prems*(*2*) *f1*] **by** *auto*
    qed
    also have $\ldots = (\bigoplus_V y \in$ *insert* x $X.\ a \cdot f\ y \cdot y)$
    proof (*rule finsum-insert*[*symmetric*])
      show *finite* X **using** *insert.hyps*(*1*) **.**
      show $x \notin X$ **using** *insert.hyps*(*2*) **.**
      show $(\lambda y.\ a \cdot f\ y \cdot y) \in X \to$ *carrier* V
      proof (*unfold Pi-def*, *auto*)
        fix y
        assume *y-in-X*: $y \in X$
        show $a \cdot f\ y \cdot y \in$ *carrier* V

```
        proof (rule mult-closed)
            show f(y)·y ∈ carrier V using y-in-X and insert.prems(1) and in-
sert.prems(3) and mult-closed by auto
          show a∈ carrier K using insert.prems(2) .
        qed
      qed
      show a · f x · x ∈ carrier V
      proof (rule mult-closed)
       show  f x · x ∈ carrier V using insert.prems (1) and insert.prems(3) and
mult-closed by auto
        show a ∈ carrier K using insert.prems(2) .
      qed
    qed
    finally show ?thesis by auto
  qed
qed
```

To multiply a linear combination by a scalar a is the same that multiplying each term of the linear combination by a.

```
lemma linear-combination-rdistrib:
  ⟦good-set X; f ∈ coefficients-function (carrier V);
  a ∈ carrier K⟧ ⟹ a · (linear-combination f X)
  = linear-combination (%i. a ⊗ f(i)) X
proof −
  assume good-set: good-set X
    and coefficients-function-f:
    f ∈ coefficients-function (carrier V)
    and a-in-K:a ∈ carrier K
  have X-subset-V: X⊆ carrier V
    using good-set unfolding good-set-def by auto
  have finite-X: finite X
    using good-set unfolding good-set-def by auto
  have f: f ∈ X→carrier K
  proof (unfold Pi-def, auto)
    fix x
    assume x-in-X: x ∈ X
    show f x∈ carrier K
      using x-in-X and X-subset-V and coefficients-function-f
      unfolding coefficients-function-def  by auto
  qed
  show a · linear-combination f X
    = linear-combination (λi. a ⊗ f i) X
  proof (unfold linear-combination-def)
    have (⨁Vy∈X. (a ⊗ f y) · y)=(⨁Vy∈X. a · f y · y)
    proof (rule finsum-cong')
      show X=X ..
      show (λy. a · f y · y) ∈ X → carrier V
      proof (unfold Pi-def, auto)
        fix y
```

```
      assume y-in-X: y∈ X
      show a · f y · y ∈ carrier V
      proof (rule mult-closed)
        show f y · y ∈ carrier V using y-in-X and X-subset-V and f and
mult-closed by auto
        show a∈ carrier K using a-in-K .
      qed
    qed
    show ⋀i. i ∈ X ⟹ (a ⊗ f i) · i = a · f i · i
    proof (rule impE, auto)
      fix i
      assume i-in-X: i∈ X
      show (a ⊗ f i) · i = a · f i · i
      proof (rule mult-assoc)
        show i∈ carrier V using i-in-X and X-subset-V by auto
        show a∈ carrier K using a-in-K .
        show f i ∈ carrier K using i-in-X and f by auto
      qed
    qed
  qed
  also have ...=a·(⨁ Vy∈X. f y · y)
    using finsum-aux [OF finite-X X-subset-V a-in-K f, symmetric] .
  finally show a · (⨁ Vy∈X. f y · y) = (⨁ Vy∈X. (a ⊗ f y) · y)
    by auto
 qed
qed
```

Now some useful lemmas which will be helpful to prove other ones.

```
lemma coefficients-function-g-f-null:
  assumes cf-f: f ∈ coefficients-function (carrier V)
  shows (λx. if x ∈ Y then f(x) else 0K)
  ∈ coefficients-function (carrier V) using cf-f
  unfolding coefficients-function-def by auto
```

This lemma is a generalization of the idea through we have proved *linear-dependent-subset-implies-line* $[\![Y \subseteq X;$ *good-set* X; *linear-dependent* $Y]\!] \Longrightarrow$ *linear-dependent* X. Using it we could reduce its proof, but in Halmos the section of linear dependence goes before the one about linear combinations. The proof is based on dividing the linear combination into two sums, from which one of them is equal to 0_V. This lemma takes up about 130 code lines.

```
lemma eq-lc-when-out-of-set-is-zero:
  assumes good-set-A: good-set A and good-set-Y: good-set Y
  and cf-f: f∈ coefficients-function (carrier V)
  shows linear-combination (λx. if x ∈ Y then f(x) else 0K)
  (Y∪A) = linear-combination f Y
proof -
  let ?g=(λx. if x ∈ Y then f(x) else 0K)
  have descomposicion-conjuntos:Y∪A=Y∪(A−Y) by auto
```

```
have disjuntos: Y Int (A− Y)={}
  by simp
have finite-A: finite A
  using good-set-A
  unfolding good-set-def by simp
have finite-Y: finite Y
  using good-set-Y
  unfolding good-set-def by auto
have finite-A-minus-Y: finite (A− Y)
  using finite-A  by simp
have g1:?g ∈ Y → carrier K
  using coefficients-function-g-f-null[OF cf-f, of Y]
  unfolding coefficients-function-def
  using good-set-Y
  unfolding good-set-def
  by auto
have g2:?g ∈ (A− Y) → carrier K
  using coefficients-function-g-f-null[OF cf-f, of (A− Y)]
  unfolding coefficients-function-def
  by auto
let ?h=(λx. ?g(x)·x)
have h1: ?h ∈ Y → carrier V
proof
  fix x
  assume x-in-Y: x∈Y
  have x-in-V: x∈ carrier V
  proof
    have Y-subset-V: Y⊆ carrier V
      using good-set-Y
      unfolding good-set-def
      by auto
    show ?thesis using Y-subset-V and x-in-Y by auto
  qed (auto)
  have gx-in-K: ?g(x)∈ carrier K
    using g1
    using x-in-Y
    unfolding Pi-def by auto
  have gx-x-in-V: ?g(x)·x ∈ carrier V
    using mult-closed [OF x-in-V gx-in-K] by auto
  show (if x ∈ Y then f x else 0) · x ∈ carrier V
    using gx-x-in-V by auto
qed
have h2: ?h ∈ (A− Y) → carrier V
proof
  fix x
  assume x-in-A-minus-Y: x∈ (A− Y)
  have x-in-V: x∈ carrier V
  proof
    have A-minus-Y-subset-V: (A− Y)⊆carrier V
```

using *good-set-Y* **and** *good-set-A*
unfolding *good-set-def*
by *auto*
show *?thesis*
using *A-minus-Y-subset-V*
using *x-in-A-minus-Y* **by** *auto*
qed (*auto*)
have *gx-in-K*: *?g(x)*$\in$ *carrier K*
using *x-in-A-minus-Y*
by *auto*
have *gx-x-in-V*: *?g(x)$\cdot$x* $\in$ *carrier V*
using *mult-closed* [*OF x-in-V gx-in-K*] **by** *auto*
show (*if x* $\in$ *Y then f x else* **0**) $\cdot$ *x* $\in$ *carrier V*
using *gx-x-in-V* **by** *auto*
qed
have *descomposicion*: *linear-combination ?g* (*Y*$\cup$(*A*$-$*Y*))=*linear-combination ?g Y* $\oplus_V$ *linear-combination ?g* (*A*$-$*Y*)
unfolding *linear-combination-def*
using *finsum-Un-disjoint* [*OF finite-Y finite-A-minus-Y disjuntos h1 h2*]
by *auto*
have *sum-g-Y-equal-sum-f-Y*: *linear-combination ?g Y*=*linear-combination f Y*
proof (*unfold linear-combination-def*)
have *iguales*: *Y*=*Y* **..**
show ($\bigoplus_V y\in Y$. (*if y* $\in$ *Y then f y else* **0**) $\cdot$ *y*) = ($\bigoplus_V y\in Y$. *f y* $\cdot$ *y*)
using *finsum-cong* [*OF iguales*] **using** *h1* **by** *auto*
qed
have *sum-g-A-minus-Y*:*linear-combination ?g* (*A*$-$*Y*)=$\mathbf{0}_V$
proof $-$
have *X-subset-V*: *A* $\subseteq$ *carrier V*
using *good-set-A*
unfolding *good-set-def* **by** *auto*
hence *A-minus-Y-subset-V*:(*A*$-$*Y*) $\subseteq$ *carrier V* **by** *auto*
have *not-in-Y*: *x* $\in$ (*A*$-$*Y*)$\Longrightarrow$ *x* $\notin$ *Y* **by** *auto*
have *linear-combination ?g* (*A*$-$*Y*)=($\bigoplus_V y\in A - Y$. **0** $\cdot$ *y*)
proof (*unfold linear-combination-def*)
have *igualesA-minus-Y*: *A*$-$*Y*=*A*$-$*Y* **..**
show ($\bigoplus_V y\in A - Y$. (*if y* $\in$ *Y then f y else* **0**) $\cdot$ *y*) = *finsum V* (*op* $\cdot$ **0**) (*A* $-$ *Y*)
using *finsum-cong* [*OF igualesA-minus-Y eqTrueI* [*OF h2*]] **by** *auto*
qed
also have ...=($\bigoplus_V y\in A - Y$. $\mathbf{0}_V$)
proof (*rule finsum-cong'*)
show *A* $-$ *Y* = *A* $-$ *Y* **..**
show (λy. $\mathbf{0}_V$) $\in$ *A* $-$ *Y* $\rightarrow$ *carrier V* **by** *simp*
show $\bigwedge i$. *i* $\in$ *A* $-$ *Y* $\Longrightarrow$ **0** $\cdot$ *i* = $\mathbf{0}_V$
using *zeroK-mult-V-is-zeroV*
using *A-minus-Y-subset-V* **by** *auto*
qed
also have ...=$\mathbf{0}_V$

using *finsum-zero* [*OF finite-A-minus-Y*] **.**
finally show *?thesis* **by** *auto*
qed
have *aux*: *linear-combination ?g* $(Y \cup (A - Y))$=*linear-combination ?g* $(Y \cup A)$
using *descomposicion-conjuntos* **by** *auto*
show *?thesis*
using *descomposicion*
using *aux*
using *sum-g-Y-equal-sum-f-Y*
using *sum-g-A-minus-Y*
using *V.r-zero*[*OF linear-combination-closed*[*OF good-set-Y cf-f*]]
by *auto*
qed

Another auxiliary lemma. It will be very useful to prove properties in future sections. If we have an equality of the form $\mathbf{0}_V = g\ x \cdot x \oplus_V$ *linear-combination* $g\ X$, then we can work out the value of x (there exists a coefficients function f such that $x = linear\_combination\ f\ X$. This coefficients function is explicitly defined by dividing each of the values $g(y)$ by $g(x)$).

lemma *work-out-the-value-of-x*:
assumes *good-set*: *good-set X*
and *coefficients-function-g*:
$g \in$ *coefficients-function* (*carrier V*)
and *x-in-V*: $x \in$ *carrier V*
and *gx-not-zero*: $g\ x \neq \mathbf{0}_K$
and *lc-descomposicion*: $\mathbf{0}_V = g(x) \cdot x \oplus_V$ *linear-combination g X*
shows $\exists f.\ f \in$ *coefficients-function* (*carrier V*)
$\wedge$ *linear-combination f X* $= x$
proof $-$
have *gx-in-K*: $g(x) \in$ *carrier K*
using *coefficients-function-g* **using** *x-in-V*
unfolding *coefficients-function-def* **by** *auto*
hence *gx-in-Units*: $g(x) \in$ *Units K*
using *gx-not-zero* **using** *field-Units* **by** *auto*
hence *inv-gx-in-K*: *inv* $g(x) \in$ *carrier K*
using *Units-inv-closed* **by** *auto*
hence *minv-gx-in-K*: $\ominus$ (*inv* $g(x)$) $\in$ *carrier K*
using *a-inv-closed* **by** *auto*
have $(\ominus_K$ (*inv g x*)$)\cdot$ $\mathbf{0}_V = \ominus_K$ (*inv g x*) $\cdot$ $(g(x) \cdot x \oplus_V$ *linear-combination g X*$)$
using *lc-descomposicion* **by** *auto*
hence $\mathbf{0}_V = \ominus_K$ (*inv g x*)$\cdot g(x) \cdot x \oplus_V \ominus_K$ (*inv g x*)$\cdot$ (*linear-combination g X*)
using *scalar-mult-zeroV-is-zeroV*[*OF minv-gx-in-K*]
using *add-mult-distrib1*[*OF mult-closed*[*OF x-in-V gx-in-K*]
linear-combination-closed[*OF good-set coefficients-function-g*] *minv-gx-in-K*]
by *auto*
also have ...=$(\ominus_K$ (*inv g x*)$\otimes$ $g(x))\cdot x \oplus_V \ominus_K$ (*inv g x*)$\cdot$ (*linear-combination g X*)
using *mult-assoc*[*OF x-in-V minv-gx-in-K gx-in-K*] **by** *auto*

also have ...=($\ominus_K$ ((*inv g x*)$\otimes$ *g*(*x*)))$\cdot$*x* $\oplus_V$ $\ominus_K$ (*inv g x*)$\cdot$ (*linear-combination g X*)
  using *l-minus*[*OF inv-gx-in-K gx-in-K*] **by** *auto*
also have ...=$\ominus_V$*x* $\oplus_V$ $\ominus_K$ (*inv g x*)$\cdot$ (*linear-combination g X*)
  using *Units-l-inv*[*OF gx-in-Units*] **using** *negate-eq*[*OF x-in-V*] **by** *auto*
also have ...=$\ominus_V$*x* $\oplus_V$ *linear-combination* (%*i*. ($\ominus_K$ (*inv g x*)) $\otimes$ *g*(*i*)) *X*
  using *linear-combination-rdistrib* [*OF good-set coefficients-function-g minv-gx-in-K*] **by** *auto*
finally have *igualdad*: $\mathbf{0}_V$=$\ominus_V$*x* $\oplus_V$ *linear-combination* (%*i*. ($\ominus_K$ (*inv g x*)) $\otimes$ *g*(*i*)) *X* **.**
let *?f*=(λ*y*.($\ominus_K$ (*inv g x*))$\otimes$ *g*(*y*))
have *coefficients-function-f*: *?f* $\in$ *coefficients-function* (*carrier V*)
proof (*unfold coefficients-function-def*, *unfold Pi-def*, *auto*)
  fix *y*
  assume *y-in-V*: *y*$\in$ *carrier V*
  show $\ominus$ (*inv g x*) $\otimes$ *g y* $\in$ *carrier K*
  using *minv-gx-in-K y-in-V coefficients-function-g* **unfolding** *coefficients-function-def* **by** *auto*
  next
    fix *xa*
    assume *xa-notin-V*: *xa* $\notin$ *carrier V*
    have $\ominus$ (*inv g x*) $\otimes$ *g xa* = $\ominus$ (*inv g x*) $\otimes$ $\mathbf{0}$
      using *xa-notin-V coefficients-function-g* **unfolding** *coefficients-function-def* **by** *simp*
    also have ...= $\mathbf{0}$ **using** *K.r-null*[*OF minv-gx-in-K*] **.**
    finally show $\ominus$ (*inv g x*) $\otimes$ *g xa* = $\mathbf{0}$ **.**
qed
hence *x* $\oplus_V$ $\mathbf{0}_V$= *x* $\oplus_V$ $\ominus_V$ *x* $\oplus_V$ *linear-combination ?f X*
  using *igualdad*
  using *V.a-assoc* [*OF x-in-V a-inv-closed*[*OF x-in-V*] *linear-combination-closed*[*OF good-set* -], *symmetric*]
  by *auto*
hence *x* = *linear-combination ?f X*
  using *V.r-zero* [*OF x-in-V*]
  using *a-minus-def*[*OF x-in-V x-in-V*,*symmetric*] *r-neg* [*OF x-in-V*]
  using *V.l-zero* [*OF linear-combination-closed*[*OF good-set coefficients-function-f*]]

  by *auto*
thus *?thesis* **using** *coefficients-function-f* **by** *fastsimp*
qed

Now we are going to prove a property presented in Halmos, section 6: if $\{x_i\}_{i\in\mathbb{N}}$ is linearly independent, then a necessary an sufficient condition that x be a linear combination of $\{x_i\}_{i\in\mathbb{N}}$ is that the enlarged set, obtained by adjoining x to $\{x_i\}_{i\in\mathbb{N}}$, be linearly dependent.

Here the first implication. The proof is based on definig a linear combination of the set *insert x X* equal to 0_V. As long as we know that *linear_combination f X* = *x* we define a coefficients function for *insert x*

X where the coefficients of $y \in X$ are $f(y)$ and the coefficient of x is -1. A detail that is omitted in Halmos is that not every coefficient is zero since the coefficient of x is -1. The complete proof requires 102 lines of Isabelle code.

```
lemma lc1:
  assumes linear-independent-X: linear-independent X
  and x-in-V:x ∈ carrier V and x-not-in-X:x ∉ X
  shows (∃f. f∈ coefficients-function (carrier V) ∧ linear-combination f X = x)
⟹ linear-dependent (insert x X)
proof −
  assume (∃f. f∈ coefficients-function (carrier V) ∧ linear-combination f X = x)
  from this obtain f
    where coefficients-function-f: f∈ coefficients-function (carrier V)
    and linear-combination-x: linear-combination f X = x by auto
  show linear-dependent (insert x X)
  proof (unfold linear-dependent-def)
    have good-set: good-set X using l-ind-good-set [OF linear-independent-X] .
    have finite-X-union-x: finite (insert x X)
      using good-set unfolding good-set-def by auto
    have X-union-x-in-V: (insert x X) ⊆ carrier V
    proof −
      have X-subset-V: X⊆ carrier V using good-set unfolding good-set-def by
auto
      from this show ?thesis using x-in-V by auto
    qed
    have good-set-X-union-x: good-set (insert x X)
      unfolding good-set-def using finite-X-union-x X-union-x-in-V by auto

    let ?g=(λy. if y = x then ⊖K 1 K else f(y))
    have g: ?g ∈ (insert x X)→ carrier K
      using X-union-x-in-V
      using coefficients-function-f
      unfolding coefficients-function-def by auto
    have coefficients-function-g: ?g∈ coefficients-function (carrier V)
      proof (unfold coefficients-function-def, auto)
        fix x
        assume x ∈ carrier V
        thus f x ∈ carrier K using fx-in-K[OF - coefficients-function-f] by simp
        next
        assume x-notin-carrier-V: x ∉ carrier V
        thus ⊖ 1 = 0 using x-in-V by contradiction
        next
        fix xa
        assume xa-not-x: xa ≠ x and xa-notin-V: xa ∉ carrier V
      thus f xa = 0 using coefficients-function-f unfolding coefficients-function-def
by blast
      qed
    have sum-zero: linear-combination ?g (insert x X)=0V
    proof −
```

have *linear-combination ?g* (*insert x X*)=*?g x* · *x* $\oplus_V$ *linear-combination ?g X*
proof (*rule linear-combination-insert*)
show *good-set X* **using** *good-set* **.**
show $x \in$ *carrier V* **using** *x-in-V* **.**
show $x \notin X$ **using** *x-not-in-X* **.**
show *?g* $\in$ *coefficients-function* (*carrier V*) **using** *coefficients-function-g* **.**
qed
also have ...=$\ominus$ **1** · *x* $\oplus_V$ *linear-combination ?g X* **using** *x-not-in-X* **by** *auto*
also have ...=$\ominus_V x \oplus_V x$
proof −
have *linear-combination ?g X*=*linear-combination f X* **unfolding** *linear-combination-def*

proof (*rule finsum-cong′*, *auto*)
assume *x-in-X*: $x \in X$
thus $\ominus$ **1** · *x* = *f x* · *x* **using** *x-not-in-X* **by** *contradiction*
next
fix *y*
assume *y-in-X*: $y \in X$
hence *y-in-V*: $y \in$ *carrier V* **using** *good-set* **unfolding** *good-set-def* **by** *fast*
show *f y* · *y* $\in$ *carrier V* **using** *fx-x-in-V*[*OF y-in-V coefficients-function-f*] **.**
qed
thus *?thesis*
using *negate-eq*
using *linear-combination-x*
using *x-in-V* **unfolding** *linear-combination-def* **by** *auto*
qed
also have ...=$\mathbf{0}_V$ **using** *V.l-neg*[*OF x-in-V*] **.**
finally show *?thesis* **by** *simp*
qed
have *not-all-zero*: ¬ (∀ *x*::′*b*∈*insert x X*. *?g x* = **0**)
proof −
have *minus-one-not-zero*: $\ominus$**1** ≠ **0**
— We know that 1 is not 0, but not that - 1 is not 0. We have to prove it.
proof (*rule notI*)
assume *minus-one-eq-zero*: $\ominus$ **1** = **0**
hence $\ominus$ **1** $\oplus$ **1** = **0** $\oplus$ **1** **by** *simp*
hence **0**=**1** **using** *K.l-neg* **using** *K.one-closed* **using** *l-zero* **by** *simp*
thus *False* **using** *K.one-not-zero* **by** *simp*
qed
thus *?thesis*
using *x-not-in-X* **by** *auto*
qed
have *?g* $\in$ *coefficients-function* (*carrier V*)
∧ *linear-combination ?g* (*insert x X*) = $\mathbf{0}_V$ ∧ ¬ (∀ *x*::′*b*∈*insert x X*. *?g x* =

$\mathbf{0}$)
using *coefficients-function-g* **and** *sum-zero* **and** *not-all-zero* **by** *auto*
hence $\exists f::'b \Rightarrow 'a.$
$f \in$ *coefficients-function* (*carrier V*) $\wedge$ *linear-combination f* (*insert x X*) $=$ $\mathbf{0}_V$
$\wedge \neg (\forall x::'b \in insert\ x\ X.\ f\ x = \mathbf{0})$
apply (*rule exI* [*of - ?g*]) **.**
thus
good-set (*insert x X*) $\wedge$
($\exists f.\ f \in$ *coefficients-function* (*carrier V*) $\wedge$ *linear-combination f* (*insert x X*) $= \mathbf{0}_V$
$\wedge \neg (\forall x \in insert\ x\ X.\ f\ x = \mathbf{0})$)
using *good-set-X-union-x* **by** *simp*
qed
qed

And now we present the second implication. The proof is based on obtaining a linear combination of *insert x X* in which not all scalars are zero (we can do it since X is linearly dependent). Hence we prove that the scalar of x is not zero (if it is, hence X would be dependent and independent so a contradiction). Then, we can express x as a linear combination of the elements of X.

lemma *lc2*:
assumes *linear-independent-X*: *linear-independent X*
and *x-in-V*: $x \in$ *carrier V*
and *x-not-in-X*: $x \notin X$
shows *linear-dependent* (*insert x X*)
$\Longrightarrow$ ($\exists f.\ f \in$ *coefficients-function* (*carrier V*)
$\wedge$ *linear-combination f X* $= x$)
proof −
assume *linear-dependent-X-union-x*: *linear-dependent* (*insert x X*)
show ($\exists f.\ f \in$ *coefficients-function* (*carrier V*) $\wedge$ *linear-combination f X* $= x$)
proof −
have *good-set*: *good-set X* **using** *l-ind-good-set*[*OF linear-independent-X*] **.**
have *X-subset-V*: $X \subseteq$ *carrier V* **using** *good-set* **unfolding** *good-set-def* **by** *auto*
have *finite-X*: *finite X* **using** *good-set* **unfolding** *good-set-def* **by** *auto*
from *linear-dependent-X-union-x* **obtain** *g*
where *coefficients-function-g*: $g \in$ *coefficients-function* (*carrier V*)
and *sum-zero-g-X-union-x*: *linear-combination g* (*insert x X*) $= \mathbf{0}_V$
and *not-all-zero-g-X-union-x*: $\neg (\forall x \in insert\ x\ X.\ g\ x = \mathbf{0})$
unfolding *linear-dependent-def* **unfolding** *coefficients-function-def* **unfolding** *linear-combination-def* **by** *auto*
have *lc-descomposicion*: *linear-combination g* (*insert x X*) $= g(x) \cdot x \oplus_V$ *linear-combination g X*
proof (*unfold linear-combination-def*, *rule finsum-insert*)
show *finite X* **using** *finite-X* **.**
show $x \notin X$ **using** *x-not-in-X* **.**
show $(\lambda y.\ g\ y \cdot y) \in X \rightarrow$ *carrier V*

using *coefficients-function-g* **unfolding** *coefficients-function-def* **using** *X-subset-V* **using** *mult-closed* **by** *auto*
show $g\ x \cdot x \in carrier\ V$
using *coefficients-function-g* **unfolding** *coefficients-function-def* **using** *x-in-V* **using** *mult-closed* **by** *auto*
qed
have *gx-not-zero*: $g\ x \neq \mathbf{0}_K$
proof (*rule notI*)
assume *gx-zero*: $g\ x = \mathbf{0}_K$
have *sum-zero-g-X*: *linear-combination* $g\ X=\mathbf{0}_V$
proof −
have *gx-x-zero*: $g(x)\cdot\ x=\mathbf{0}_V$ **using** *gx-zero* **using** *zeroK-mult-V-is-zeroV* [*OF x-in-V*] **by** *auto*
have $\mathbf{0}_V= \mathbf{0}_V\oplus_V$ *linear-combination* $g\ X$
using *lc-descomposicion* **using** *gx-x-zero* **using** *sum-zero-g-X-union-x* **by** *auto*
also have ...=*linear-combination* $g\ X$
proof (*rule V.l-zero*)
show *linear-combination* $g\ X \in carrier\ V$
proof (*unfold linear-combination-def*, *rule finsum-closed*)
show *finite* X **using** *good-set* **unfolding** *good-set-def* **by** *auto*
show $(\lambda y.\ g\ y \cdot y) \in X \rightarrow carrier\ V$
using *coefficients-function-g* **unfolding** *coefficients-function-def*
using *X-subset-V* **using** *mult-closed* **by** *auto*
qed
qed
finally show *?thesis* **by** *simp*
qed
have *not-all-zero-g-X*: $\neg\ (\forall x{\in}X.\ g\ x = \mathbf{0})$ **using** *not-all-zero-g-X-union-x* **and** *gx-zero* **by** *auto*
have $g \in$ *coefficients-function* (*carrier V*) $\wedge$ *good-set* X $\wedge$ *linear-combination* $g\ X = \mathbf{0}_V$
using *coefficients-function-g* **and** *good-set* **and** *sum-zero-g-X* **by** *simp*
thus *False* **using** *linear-independent-X* **and** *not-all-zero-g-X* **unfolding** *linear-independent-def* **by** *auto*
qed
have $\exists f.\ f \in$ *coefficients-function* (*carrier V*) $\wedge$ *linear-combination* $f\ X = x$
proof (*rule work-out-the-value-of-x*)
show *good-set* X **using** *good-set* **.**
show $g \in$ *coefficients-function* (*carrier V*) **using** *coefficients-function-g* **.**
show $x \in carrier\ V$ **using** *x-in-V* **.**
show $g\ x \neq \mathbf{0}$ **using** *gx-not-zero* **.**
show $\mathbf{0}_V = g\ x \cdot x \oplus_V$ *linear-combination* $g\ X$ **using** *lc-descomposicion* **using** *sum-zero-g-X-union-x* **by** *auto*
qed
thus *?thesis* **by** *fast*
qed
qed

Finally, the theorem proving the equivalence of both definitions.

lemma *lc1-eq-lc2*:
assumes *linear-independent-X*: *linear-independent X*
and *x-in-V*:$x \in$ *carrier V* **and** *x-not-in-X*:$x \notin X$
shows *linear-dependent* (*insert x X*) $\longleftrightarrow$
($\exists$ *f*. *f*$\in$ *coefficients-function* (*carrier V*)
$\wedge$ *linear-combination f X* = *x*)
using *assms lc1 lc2* **by** *blast*

This lemma doesn't appears in Halmos (but it seems to be a similar result to the theorem **??**). The proof is based on obtaining a linear combination of the elements of $X \cup Y$ equal to 0_V where not all scalars are equal to $0_{\mathbb{K}}$. Hence we can express an element $y \in (X \cup Y)$ such that its scalar is not zero as a linear combination of the rest elements of $X \cup Y$. This is a long proof of 180 lines.

lemma *exists-x-linear-combination*:
assumes *li-X*: *linear-independent X*
and *ld-XY*: *linear-dependent* ($X \cup Y$)
shows $\exists y \in Y$. $\exists g$. $g \in$ *coefficients-function* (*carrier V*)
$\wedge$ y = *linear-combination g* ($X \cup (Y - \{y\})$)
proof −
from *ld-XY* **obtain** *f* **where** *coefficients-function-f*: $f \in$ *coefficients-function* (*carrier V*)
and *sum-zero-XY*: *linear-combination f* ($X \cup Y$)=$\mathbf{0}_V$
and *not-all-zero*: $\neg$ ($\forall x \in X \cup Y$. $f\,x = \mathbf{0}_K$)
and *good-set-XY*: *good-set* ($X \cup Y$)
unfolding *linear-dependent-def* **by** *auto*
have $X \cup Y = X \cup (Y - X)$ **by** *simp*
hence *linear-combination f* ($X \cup Y$)= *linear-combination f* ($X \cup (Y - X)$) **by** *simp*
also have ...=*linear-combination f X* $\oplus_V$ *linear-combination f* ($Y - X$)
proof (*unfold linear-combination-def*, *rule finsum-Un-disjoint*)
show *finite X* **using** *good-set-XY* **unfolding** *good-set-def* **by** *auto*
show *finite* ($Y - X$) **using** *good-set-XY* **unfolding** *good-set-def* **by** *auto*
show $X \cap (Y - X) = \{\}$ **by** *simp*
show (λy. $f\,y \cdot y$) $\in X \rightarrow$ *carrier V*
proof (*unfold Pi-def*, *auto*)
fix *x*
assume *x-in-X*: $x \in X$
hence *x-in-V*: $x \in$ *carrier V* **using** *good-set-XY* **unfolding** *good-set-def* **by** *fast*
show $f\,x \cdot x \in$ *carrier V* **using** *fx-x-in-V*[*OF x-in-V coefficients-function-f*] **.**
qed
show (λy. $f\,y \cdot y$) $\in Y - X \rightarrow$ *carrier V*
proof −
have $Y - X \subseteq$ *carrier V* **using** *good-set-XY* **unfolding** *good-set-def* **by** *auto*
thus *?thesis* **using** *coefficients-function-f* **unfolding** *coefficients-function-def* **using** *mult-closed* **by** *auto*
qed
qed
finally have *descomposicion*: *linear-combination f X* $\oplus_V$ *linear-combination f*

$(Y - X)=\mathbf{0}_V$ **using** *sum-zero-XY* **by** *simp*
have $\neg(\forall x \in (Y-X).\ f\ x = \mathbf{0}_K)$
proof (*rule notI*)
assume *all-zero-YX*: $(\forall x \in (Y-X).\ f\ x = \mathbf{0}_K)$
have *good-set-X*:*good-set X* **using** *good-set-XY* **unfolding** *good-set-def* **by** *auto*
have *linear-combination f* $(Y-X)=\mathbf{0}_V$
proof −
have *YX-in-V*: $Y-X \subseteq$ *carrier V* **using** *good-set-XY* **unfolding** *good-set-def* **by** *auto*
have *finite-YX*:*finite* $(Y-X)$ **using** *good-set-XY* **unfolding** *good-set-def* **by** *auto*
have *good-set-X*:*good-set X* **using** *good-set-XY* **unfolding** *good-set-def* **by** *auto*
have $(\bigoplus_V y\in Y - X.\ f\ y \cdot y)=(\bigoplus_V y\in Y - X.\ \mathbf{0}_V)$
proof (*rule finsum-cong'*)
show $Y - X = Y - X$ **by** *simp*
show $(\lambda y.\ \mathbf{0}_V) \in Y - X \to$ *carrier V* **by** *simp*
show $\bigwedge i.\ i \in Y - X \Longrightarrow f\ i \cdot i = \mathbf{0}_V$ **using** *YX-in-V* **using** *all-zero-YX* **using** *zeroK-mult-V-is-zeroV* **by** *auto*
qed
also have $\ldots=\mathbf{0}_V$ **using** *finsum-zero*[*OF finite-YX*] **.**
finally show *?thesis* **unfolding** *linear-combination-def* **by** *simp*
qed
hence *linear-combination f* $X=\mathbf{0}_V$
using *descomposicion* **and** *good-set-X*
and *V.r-zero*[*OF linear-combination-closed*[*OF good-set-X coefficients-function-f*]] **by** *auto*
hence $(\forall x\in X.\ f\ x = \mathbf{0})$
using *coefficients-function-f* **and** *good-set-X* **and** *li-X* **unfolding** *linear-independent-def* **by** *auto*
hence $(\forall x\in X\cup(Y-X).\ f\ x = \mathbf{0})$ **using** *all-zero-YX* **by** *auto*
hence $(\forall x\in X\cup Y.\ f\ x = \mathbf{0})$ **by** *auto*
thus *False* **using** *not-all-zero* **by** *contradiction*
qed
then obtain y **where** *y-in-Y*: $y \in Y$ **and** *y-notin-X*: $y \notin X$ **and** *fy-not-zero*: $f(y)\neq\mathbf{0}_K$ **by** *auto*
hence *igualdad-conjuntos*: *insert y* $((Y-X)-\{y\})=Y-X$ **by** *auto*
have *linear-combination f* (*insert y* $((Y-X)-\{y\})$)$=f(y)\cdot y \oplus_V$ *linear-combination f* $((Y-X)-\{y\})$
proof (*unfold linear-combination-def*, *rule finsum-insert*)
show *finite* $(Y - X - \{y\})$ **using** *good-set-XY* **unfolding** *good-set-def* **by** *auto*
show $y \notin Y - X - \{y\}$ **by** *simp*
show $(\lambda x.\ f\ x \cdot x) \in Y - X - \{y\} \to$ *carrier V*
proof −
have $(Y - X - \{y\}) \subseteq$ *carrier V* **using** *good-set-XY* **unfolding** *good-set-def* **by** *auto*
thus *?thesis*

using *coefficients-function-f* **unfolding** *coefficients-function-def*
using *mult-closed* **by** *auto*
qed
show $f\ y \cdot y \in carrier\ V$
proof (*rule mult-closed*)
show $y \in carrier\ V$ **using** *y-in-Y* **and** *good-set-XY* **unfolding** *good-set-def* **by** *auto*
show $f(y) \in carrier\ K$
using *coefficients-function-f* **unfolding** *coefficients-function-def*
using *good-set-XY* **unfolding** *good-set-def* **using** *y-in-Y* **by** *auto*
qed
qed
hence *eq-lc-when-out-of-set-is-zero*: *linear-combination* $f\ (Y - X) = f(y) \cdot y \oplus_V$ *linear-combination* $f\ ((Y - X) - \{y\})$
using *igualdad-conjuntos* **by** *auto*
have *good-set-X*: *good-set* X **using** *good-set-XY* **unfolding** *good-set-def* **by** *simp*
have *cb-YXy*: *good-set* $(Y - X - \{y\})$**using** *good-set-XY* **unfolding** *good-set-def* **by** *auto*
have *cb-XYy*: *good-set* $(X \cup (Y - \{y\}))$ **using** *good-set-XY* **unfolding** *good-set-def* **by** *auto*
have *fy-in-K*:$f(y) \in carrier\ K$
using *coefficients-function-f* **unfolding** *coefficients-function-def*
using *y-in-Y good-set-XY* **unfolding** *good-set-def* **by** *auto*
hence *mfy-in-K*: $\ominus_K f(y) \in carrier\ K$ **using** *K.a-inv-closed* **by** *auto*
have $\ominus_K f(y) \neq \mathbf{0}_K$
proof (*rule notI*)
assume $\ominus\ f\ y = \mathbf{0}$
hence $\ominus(\ominus\ f(y)) = \ominus\mathbf{0}$ **by** *auto*
hence $f(y) = \mathbf{0}$ **using** *fy-in-K* **and** *K.minus-minus* **and** *K.minus-zero* **by** *auto*
thus *False* **using** *fy-not-zero* **by** *contradiction*
qed
hence *mfy-in-Units-K*: $\ominus_K f(y) \in Units\ K$ **using** *mfy-in-K* **and** *K.field-Units* **by** *auto*
hence *inv-mfy-in-K*: $inv(\ominus_K f(y)) \in carrier\ K$ **by** *auto*
have *fy-y-in-V*: $f(y) \cdot y \in carrier\ V$
proof (*rule mult-closed*)
show $y \in carrier\ V$ **using** *y-in-Y good-set-XY* **unfolding** *good-set-def* **by** *auto*
show $f(y) \in carrier\ K$ **using** *fy-in-K* **.**
qed
have *linear-combination* $f\ (Y - X) =$ *linear-combination* $f\ ((Y - X) - \{y\}) \oplus_V f(y) \cdot y$
using *eq-lc-when-out-of-set-is-zero V.a-comm*
[*OF linear-combination-closed*[*OF cb-YXy coefficients-function-f*] *fy-y-in-V*] **by** *auto*
hence *linear-combination* $f\ X \oplus_V$ (*linear-combination* $f\ ((Y - X) - \{y\}) \oplus_V f(y) \cdot y) = \mathbf{0}_V$
using *descomposicion* **by** *auto*
hence *descomposicion2*: *linear-combination* $f\ X \oplus_V$ *linear-combination* $f\ ((Y - X) - \{y\}) \oplus_V f(y) \cdot y = \mathbf{0}_V$

using $V.a\text{-}assoc$
[*OF linear-combination-closed*[*OF good-set-X coefficients-function-f*] *linear-combination-closed*
[*OF cb-YXy coefficients-function-f*] *fy-y-in-V*] **by** *auto*
hence *linear-combination f X* $\oplus_V$ *linear-combination f* $((Y-X)-\{y\}) \oplus_V f(y)\cdot y$
$\oplus_V \ominus_V (f(y)\cdot y) = \mathbf{0}_V \oplus_V \ominus_V (f(y)\cdot y)$ **by** *simp*
have *igualdad-conjuntos2*: $X\cup((Y-X)-\{y\}) = X\cup(Y-\{y\})$ **using** *y-in-Y y-notin-X*
by *auto*
have *linear-combination f* $(X\cup((Y-X)-\{y\}))$ = *linear-combination f X* $\oplus_V$
linear-combination f $((Y-X)-\{y\})$
proof (*unfold linear-combination-def*, *rule finsum-Un-disjoint*)
show *finite X* **using** *good-set-X* **unfolding** *good-set-def* **by** *auto*
show *finite* $(Y - X - \{y\})$ **using** *good-set-XY* **unfolding** *good-set-def* **by**
auto
show $X \cap (Y - X - \{y\}) = \{\}$ **using** *y-in-Y y-notin-X* **by** *auto*
show $(\lambda x.\ f\,x \cdot x) \in X \to$ *carrier V*
using *good-set-X* **unfolding** *good-set-def*
using *coefficients-function-f* **unfolding** *coefficients-function-def* **using** *mult-closed*
by *auto*
show $(\lambda x.\ f\,x \cdot x) \in Y - X - \{y\} \to$ *carrier V*
proof –
have $(Y - X - \{y\}) \subseteq$ *carrier V* **using** *good-set-XY* **unfolding** *good-set-def*
by *auto*
thus *?thesis*
using *coefficients-function-f* **unfolding** *coefficients-function-def*
using *mult-closed* **by** *auto*
qed
qed
hence *linear-combination f* $(X\cup(Y-\{y\}))$ = *linear-combination f X* $\oplus_V$ *linear-combination*
f $((Y-X)-\{y\})$
using *igualdad-conjuntos2* **by** *auto*
hence *linear-combination f* $(X\cup(Y-\{y\})) \oplus_V f(y)\cdot y \oplus_V \ominus_V (f(y)\cdot y) = \mathbf{0}_V \oplus_V$
$\ominus_V (f(y)\cdot y)$ **using** *descomposicion2* **by** *auto*
hence *linear-combination f* $(X\cup(Y-\{y\})) \oplus_V (f(y)\cdot y \oplus_V \ominus_V (f(y)\cdot y)) = \mathbf{0}_V$
$\oplus_V \ominus_V (f(y)\cdot y)$
using $V.a\text{-}assoc$[*OF linear-combination-closed* [*OF cb-XYy coefficients-function-f*]
fy-y-in-V $V.a\text{-}inv\text{-}closed$[*OF fy-y-in-V*]]
by *auto*
hence *linear-combination f* $(X\cup(Y-\{y\})) \oplus_V \mathbf{0}_V = \mathbf{0}_V \oplus_V \ominus_V (f(y)\cdot y)$ **using**
$V.r\text{-}neg$[*OF fy-y-in-V*] **by** *auto*
hence *linear-combination f* $(X\cup(Y-\{y\})) = \ominus_V(f(y)\cdot y)$
using $V.r\text{-}zero$[*OF linear-combination-closed* [*OF cb-XYy coefficients-function-f*]]
using $V.l\text{-}zero$[*OF* $V.a\text{-}inv\text{-}closed$[*OF fy-y-in-V*]]
by *auto*
hence *linear-combination f* $(X\cup(Y-\{y\})) = (\ominus_K f(y)\cdot y)$ **using** *good-set-XY* **un-
folding** *good-set-def* **using** *y-in-Y*
using *negate-eq2*[*OF - fy-in-K*] **by** *auto*
hence $inv(\ominus_K f(y)) \cdot$ *linear-combination f* $(X\cup(Y-\{y\})) = inv(\ominus_K f(y))\cdot(\ominus_K f(y)\cdot y)$
by *simp*
hence $inv(\ominus_K f(y)) \cdot$ *linear-combination f* $(X\cup(Y-\{y\})) = ((inv(\ominus_K f(y))) \otimes_K (\ominus_K f(y)))\cdot$

y
  using *y-in-Y* **using** *good-set-XY* **unfolding** *good-set-def* **using** *mult-assoc*[*OF - inv-mfy-in-K mfy-in-K*,*symmetric*]
  by *auto*
hence $inv(\ominus_K f(y)) \cdot$ *linear-combination* f $(X \cup (Y - \{y\}))= \mathbf{1}_K \cdot y$ **using** *K.Units-l-inv*[*OF mfy-in-Units-K*] **by** *auto*
hence $inv(\ominus_K f(y)) \cdot$ *linear-combination* f $(X \cup (Y - \{y\}))= y$
  using *y-in-Y* **using** *good-set-XY* **unfolding** *good-set-def* **using** *mult-1* **by** *auto*
hence *descomposicion3*: *linear-combination* $(\%i.\ inv(\ominus_K f(y)) \otimes f(i))$ $(X \cup (Y - \{y\}))= y$
  using *linear-combination-rdistrib*[*OF cb-XYy coefficients-function-f inv-mfy-in-K*] **by** *auto*
let $?g=(\%i.\ inv(\ominus_K f(y)) \otimes f(i))$
have *coefficients-function-g*: $?g \in$ *coefficients-function* (*carrier V*)
proof (*unfold coefficients-function-def*, *unfold Pi-def*, *auto*)
  fix x
  assume *x-in-V*: $x \in$ *carrier V*
  hence *fx-in-K*: $f\,x \in$ *carrier K* **using** *coefficients-function-f* **unfolding** *coefficients-function-def* **by** *auto*
  show $inv\ (\ominus\ f\ y) \otimes f\ x \in$ *carrier K* **using** *K.m-closed* [*OF inv-mfy-in-K fx-in-K*] **.**
  next
  fix x
  assume *x-notin-V*: $x \notin$ *carrier V*
  have $inv\ (\ominus\ f\ y) \otimes f\ x = inv\ (\ominus\ f\ y) \otimes \mathbf{0}$
    using *x-notin-V coefficients-function-f* **unfolding** *coefficients-function-def* **by** *simp*
  also have ...= **0** **using** *K.r-null*[*OF inv-mfy-in-K*] **.**
  finally show $inv\ (\ominus\ f\ y) \otimes f\ x = \mathbf{0}$ **.**
qed
have *linear-combination* $?g$ $(X \cup (Y - \{y\}))= y$ **using** *descomposicion3* **by** *auto*
hence $?g \in$ *coefficients-function* (*carrier V*) $\wedge$ $y=$*linear-combination* $?g$ $(X \cup (Y - \{y\}))$
  using *coefficients-function-g* **by** *auto*
hence $\exists\, g.\ g \in$ *coefficients-function* (*carrier V*) $\wedge$ $y=$*linear-combination* g $(X \cup (Y - \{y\}))$
  apply (*rule exI*[*of - ?g*]) **.**
thus *?thesis* **using** *y-in-Y* **by** *auto*
qed

A corollary of the previous lemma claims that if we have a linearly dependent set, then there exists one element which can be expressed as a linear combination of the other elements of the set.

corollary *exists-x-linear-combination2*:
assumes *ld-Y*: *linear-dependent Y*
shows $\exists\, y \in Y.\ \exists\, g.\ g \in$ *coefficients-function* (*carrier V*)
$\wedge$ $y =$ *linear-combination* g $(Y - \{y\})$
proof −
have *ld-empty-Y*: *linear-dependent*$(\{\} \cup Y)$ **using** *ld-Y* **by** *simp*
have $\exists\, y \in Y.\ \exists\, g.\ g \in$ *coefficients-function* (*carrier V*)

$\wedge$ y = *linear-combination* g $(\{\} \cup (Y - \{y\}))$
using *exists-x-linear-combination*
[*OF empty-set-is-linearly-independent ld-empty-Y*] **.**
thus *?thesis* **by** *simp*
qed

Every singleton set is linearly independent. This lemma could be in previous section, however we have to make use of some properties of linear combinations. We can repeat the proof without these properties, but it would be longer. We will use that $a \cdot x = \mathbf{0} \Longrightarrow a = \mathbf{0}$ because $x \neq \mathbf{0}$.

lemma *unipuntual-is-li*:
assumes *x-in-V*: $x \in$ *carrier* V **and** *x-not-zero*: $x \neq \mathbf{0}_V$
shows *linear-independent* $\{x\}$
proof (*cases linear-independent* $\{x\}$)
case *True* **show** *?thesis* **using** *True* **.**
next
case *False* **show** *?thesis*
proof −
have *cb*: *good-set* $\{x\}$
using *x-in-V* **unfolding** *good-set-def* **by** *simp*
have *linear-dependent* $\{x\}$
using *False*
using *not-independent-implies-dependent*[*OF cb False*]
by *auto*
from *this* **obtain** *f*
where *cf-f*: $f \in$ *coefficients-function* (*carrier* V)
and *lc*: *linear-combination* f $\{x\} = \mathbf{0}_V$
and *not-all-zero*: $\neg (\forall x \in \{x\}.\ f\ x = \mathbf{0})$
unfolding *linear-dependent-def* **by** *auto*
have *fx-not-zero*: $f\ x \neq \mathbf{0}$ **using** *not-all-zero* **by** *auto*
have $(f\ x) \cdot x = \mathbf{0}_V$ **thm** *finsum-insert*
proof −
— We could have used $[\![fa \in$ *coefficients-function* (*carrier* V); $xa \in$ *carrier* $V]\!] \Longrightarrow$ *linear-combination* fa $\{xa\} = fa\ xa \cdot xa$ directly or next calculation:
have *linear-combination* f (*insert* x $\{\}$)
$= (f\ x) \cdot x \oplus_V$ *linear-combination* f $\{\}$
using *linear-combination-insert*[*OF - x-in-V - cf-f*]
by *auto*
also have $\ldots = (f\ x) \cdot x \oplus_V \mathbf{0}_V$
using *linear-combination-of-zero* **by** *auto*
also have $\ldots = (f\ x) \cdot x$
using *V.r-zero*[*OF fx-x-in-V*[*OF x-in-V cf-f*]] **.**
finally show *?thesis* **using** *lc* **by** *auto*
qed
hence $f\ x = \mathbf{0}_K$
using *mult-zero-uniq* **and** *x-in-V* **and** *x-not-zero* **and** *cf-f*
unfolding *coefficients-function-def* **by** *auto*
thus *?thesis* **using** *fx-not-zero* **by** *contradiction*
qed

qed

Now we are ready to prove the theorem 1 in section 6 in Halmos. It will be useful (really indispensable) in future proofs and it is basic in our developement. The theorem claims that in a linear dependent set exists an element which is a linear combination of the preceding ones.

NOTE: As we are assuming that $\mathbf{0}_V$ is not in the set, the element which is a linear combination of the preceding ones will be between the second and the last position of the set (1 and $card(A) - 1$ with the notation used in our implementation of indexed sets). The element in the first position (position 0) can't be a linear combination of the preceding ones because it would be a linear combination of the empty set, hence this element would be $\mathbf{0}_V$ and it is not in the set.

We make the proof using induction (we don't follow the proof of the book). At first, it seemed easier this way.

lemma
linear-dependent-set-contains-linear-combination:
assumes *ld-X*: *linear-dependent X*
and *not-zero*: $\mathbf{0}_V \notin X$
shows $\exists\, y \in X.\ \exists\, g.\ \exists\, k{::}nat.\ \exists\, f \in \{i.\ i{<}(card\ X)\} \rightarrow X.\ f`\{i.\ i{<}(card\ X)\} = X \wedge g \in coefficients\text{-}function\ (carrier\ V)$
$\wedge\ (1{::}nat) \le k \wedge k < (card\ X) \wedge f\,k = y \wedge y = linear\text{-}combination\ g\ (f`\{i{::}nat.\ i{<}k\})$
proof –
have *good-set-X*: *good-set X* **using** *l-dep-good-set*[*OF ld-X*] **.**
hence *finite-X*: *finite X* **unfolding** *good-set-def* **by** *simp*
thus *?thesis* **using** *ld-X* **and** *not-zero*
proof (*induct set*: *finite*)
case *empty*
show *?case*
— Contradiction: we can prove that the empty set is linearly dependent.
using *empty-set-is-linearly-independent*
and *dependent-implies-not-independent* [*OF empty.prems* (*1*)] **by** *contradiction*
next
case (*insert x X*)
show *?case*
proof –
— Some previous facts which will be useful in the proof:
have *finite-xX*: *finite* (*insert x X*)
using *insert.hyps*(*1*) **by** *auto*
have *cb-X*: *good-set X*
using *l-dep-good-set*[*OF insert.prems*(*1*)]
unfolding *good-set-def* **by** *auto*
show *?thesis*
— Now we separate the proof in cases, depending on the set X is linearly dependent.

proof (*cases linear-dependent X*)
case *True*
have *zero-not-in-X*: $\mathbf{0}_V \notin X$ **using** *insert.prems*(*2*) **by** *simp*
— We obtain the 'candidates' for the goal, using the induction hypothesis.
obtain *y f g k*
where *y-in-X*: $y \in X$
and *cf-g*: $g \in$ *coefficients-function* (*carrier V*) **and** *one-le-k*: $1 \leq k$ **and** *k-le-card-X*: $k <$ *card X*
and *fk-y*: $f\ k = y$ **and** *y-lc*: $y =$ *linear-combination g* (f ‘ $\{i.\ i < k\}$) **and** *ordenfX*: f ‘$\{i.\ i <$ *card X*$\} = X$
using *insert.hyps*(*3*)[*OF True zero-not-in-X*] **by** *auto*
have *f-buena*:$f \in \{i.\ i <$ (*card X*)$\} \rightarrow X$ **using** *ordenfX* **by** *auto*
have *y-in-xX*:$y \in$ (*insert x X*) **using** *y-in-X* **by** *simp*
obtain *h* **where** *h*: $h \in \{i.\ i <$ (*card* (*insert x X*))$\} \rightarrow$ (*insert x X*)
and *ordenxX*:h‘$\{i.\ i <$ (*card* (*insert x X*))$\} =$ (*insert x X*)
and *h-cardX-x*: h (*card X*) $= x$ **and** *h-is-f-in-X*: $\forall i.\ i<$*card*$(X) \longrightarrow h(i)=f(i)$
using *indexation-x-union-X*[*OF insert.hyps*(*1*) *insert.hyps*(*2*) *f-buena ordenfX*] **by** *auto*
show *?thesis*
— We introduce the candidates: we have to proof that satisfy the requirements:
proof (*rule bexI* [*of - y*], *rule exI* [*of - g*], *rule exI* [*of - k*], *rule bexI* [*of - h*], *rule conjI6*)
show $y \in$ *insert x X* **using** *y-in-X* **by** *fast*
show $h \in \{i.\ i <$ *card* (*insert x X*)$\} \rightarrow$ *insert x X* **using** *ordenxX* **by** *fast*
show h ‘ $\{i.\ i <$ *card* (*insert x X*)$\} =$ *insert x X* **using** *ordenxX* **.**
show $g \in$ *coefficients-function* (*carrier V*) **using** *cf-g* **.**
show $1 \leq k$ **using** *one-le-k* **.**
show $k <$ *card* (*insert x X*) **using** *k-le-card-X*
by (*metis card-insert-disjoint insert.hyps*(*1*) *insert.hyps*(*2*) *less-Suc-eq*)
show $h\ k = y$ **using** *fk-y* **and** *h-is-f-in-X* **and** *k-le-card-X* **by** *simp*
show $y =$ *linear-combination g* (h ‘ $\{i.\ i < k\}$)
using *y-lc* **and** *h-is-f-in-X* **and** *k-le-card-X*
unfolding *image-def* **by** *auto*
qed
next
case *False*
— We try to do it similarly: we define the candidates for the existencial terms (in this case we can not obtain it from the induction hypothesis) and finally we will face the thesis
have *li-X*: *linear-independent X* **using** *not-dependent-implies-independent*[*OF cb-X False*] **.**
obtain *y* **and** *g*
where *y-x*: y=x **and** *cf-g*: $g \in$ *coefficients-function* (*carrier V*)
and *x-lc-X*: $x =$ *linear-combination g X*
using *insert.prems*(*1*)
using *exists-x-linear-combination*[*OF li-X*, *of* $\{x\}$] **by** *auto*
obtain *f* **where** *ordenfX*: $X = f$ ‘ $\{i.\ i<$(*card X*)$\}$

using *finite-imp-nat-seg-image-inj-on-Pi-card*[*of X*]
using *insert.hyps* (*1*) **by** *auto*
hence *f-buena*: $f \in \{i.\ i < (card\ X)\} \rightarrow X$ **by** *auto*
obtain *h* **where** *h*: $h \in \{i.\ i < (card\ (insert\ x\ X))\} \rightarrow (insert\ x\ X)$
and *ordenxX*:$h`\{i.\ i < (card\ (insert\ x\ X))\} = (insert\ x\ X)$
and *h-cardX-x*: $h\ (card\ X) = x$ **and** *h-is-f-in-X*: $\forall i.\ i<card(X) \longrightarrow h(i)=f(i)$
using *indexation-x-union-X*[*OF insert.hyps*(*1*) *insert.hyps*(*2*) *f-buena ordenfX* [*symmetric*]] **by** *auto*
show *?thesis*
proof (*cases* $1 \leq card\ X$)
case *True*
show *?thesis*
proof (*rule bexI* [*of - x*], *rule exI* [*of - g*], *rule exI* [*of - card X*], *rule bexI* [*of - h*], *rule conjI6*)
show $h\ `\ \{i.\ i < card\ (insert\ x\ X)\} = insert\ x\ X$
using *ordenxX* **.**
show $g \in$ *coefficients-function* (*carrier V*)
using *cf-g* **.**
show $1 \leq (card\ X)$ **using** *True* **.**
show $card\ X < card\ (insert\ x\ X)$
by (*metis card-insert-disjoint insert.hyps*(*1*) *insert.hyps*(*2*) *lessI*)
show $h\ (card\ X) = x$ **using** *h-cardX-x* **.**
show $x =$ *linear-combination* $g\ (h\ `\ \{i.\ i < (card\ X)\})$
using *h-is-f-in-X ordenfX x-lc-X* **unfolding** *image-def* **by** *auto*
show $h \in \{i.\ i < card\ (insert\ x\ X)\} \rightarrow insert\ x\ X$ **using** *h* **.**
show $x \in insert\ x\ X$ **by** *fast*
qed
next
case *False*
show *?thesis*
proof (*rule FalseE*)
have $1>(card\ X)$ **using** *False* **by** *simp*
hence *X-empty*: $X = \{\}$ **using** *card-eq-0-iff* **and** *insert.hyps*(*1*) **by** *simp*
have *ld-x*: *linear-dependent* $\{x\}$ **using** *insert.prems*(*1*) **unfolding** *X-empty* **.**
have *li-x*: *linear-independent* $\{x\}$
proof (*rule unipuntual-is-li*)
show $x \in carrier\ V$
using *l-dep-good-set* [*OF ld-x*]
unfolding *good-set-def* **by** *simp*
show $x \neq \mathbf{0}_V$ **using** *insert.prems*(*2*) **by** *auto*
qed
show *False*
using *independent-implies-not-dependent* [*OF li-x*] **and** *ld-x*
by *contradiction*
qed
qed
qed

    qed
  qed
qed

lemma
  card-less-induct-good-set:
  assumes *c*: *good-set A*
  and *step*: $\bigwedge A.\ [\![\ (\bigwedge B.\ [\![\ card\ B < card\ A;\ good\text{-}set\ B\ ]\!] \Longrightarrow P\ B);$
  $good\text{-}set\ A\ ]\!] \Longrightarrow P\ A$
  shows *P A*
proof –
  have *f*: *finite A* **using** *good-set-finite* [*OF c*] **.**
  have $\bigwedge B.\ [\![card\ B \leq card\ A;\ good\text{-}set\ B]\!] \Longrightarrow P\ B$
    using *f c* **proof** (*induct*)
    case *empty*
    show *?case*
      apply (*rule step*)
      using *empty.prems* **by** *auto*
  next
    case (*insert x F*)
    show *?case*
      apply (*rule step*)
      using *insert.prems*
      using *insert.hyps*
      unfolding *good-set-def* **by** *auto*
  qed
  thus *?thesis* **using** *c* **by** *auto*
qed

Really, the result that we need to prove corresponds closer to the next theorem than we have proved in the previous theorem *linear-dependent-set-contains-linear-combination*. We have to assume that the indexation is known beforehand. This will be necessary in the future, because we will remove dependent elements in regard a gived indexation of one set (so the removed element will be unique). We will apply this theorem iteratively to a set in future proofs, so if we didn't fix the order beforehand we won't have unicity of the result (because the indexing could change in each step).

We will use the induction rule for indexed sets that we introduced before (*indexed-set-induct2*). This is a laborious and large theorem, of about 400 code lines.

theorem
  linear-dependent-set-sorted-contains-linear-combination:
  assumes *ld-A*: *linear-dependent A*
  and *not-zero*: $\mathbf{0}_V \notin A$

```
  and i: indexing (A, f)
  shows ∃ y∈A. ∃ g. ∃ k::nat.
  g ∈ coefficients-function (carrier V)
  ∧ (1::nat) ≤ k ∧ k < (card A)
  ∧ f k = y ∧ y = linear-combination g (f‘{i::nat. i<k})
  using i and ld-A and not-zero
proof (induct A f rule: indexed-set-induct2)
  show indexing (A, f) by fact
  case (empty f)
  show ∃ y∈{}. ∃ g k. g ∈ coefficients-function (carrier V) ∧ 1 ≤ k ∧ k < card
{} ∧ f k = y
    ∧ y = linear-combination g (f ‘ {i. i < k})
   using empty.prems (2) and independent-implies-not-dependent[OF empty-set-is-linearly-independent]
by contradiction
next
  case (insert a A f n)
  show ?case
  proof −
    have good-set-aA: good-set (insert a A) using l-dep-good-set [OF prems(12)] .
    hence good-set-A: good-set A unfolding good-set-def by simp
    have indexing-Af: indexing (A,f)
      using indexing-indexing-ext prems (8) prems (9) prems (10) prems (5)
      by auto
    have not-zero-A: 0_V ∉ A using prems(13) by simp
    have finite-A: finite A using prems(7) by auto
    show ?thesis
    proof (cases linear-dependent A)
      case True show ?thesis
      proof −
        have ex: ∃ y∈A. ∃ g k. g ∈ coefficients-function (carrier V) ∧
          1 ≤ k ∧
          k < card A ∧
          f k = y ∧ y = linear-combination g (f ‘ {i. i < k})
          using prems(6)[OF indexing-Af  indexing-Af True not-zero-A] .
        from this obtain y g k where cf-g: g ∈ coefficients-function (carrier V)
          and one-le-k: 1≤k and k-le-cardA: k < (card A)
          and fk-y: f k = y
          and y-lc-g-f: y = linear-combination g (f ‘ {i. i < k})
          and y-in-A: y∈A by auto
        have one-le-k-plus-one: 1≤(k+1) using one-le-k by simp
        have k-plus-one-le-card-insert-a-A: (k+1)<card(insert a A)
         using k-le-cardA and card-insert-if[OF finite-A, of a] using prems(5) by
auto
        let ?h=(λx. if x ∈ (f‘{i. i < k}) then g(x) else 0_K)
        have cb-imf: good-set (f‘{i. i < k})
          using indexing-Af unfolding indexing-def
       unfolding bij-betw-def unfolding iset-to-index-def unfolding iset-to-set-def

          using k-le-cardA one-le-k using good-set-A unfolding good-set-def
```

by *auto*
hence *cf-h*: *?h* ∈ *coefficients-function* (*carrier V*) **using** *coefficients-function-g-f-null*[*OF cf-g*] **by** *auto*
have *cb-a*: *good-set* {*a*} **using** *good-set-aA* **unfolding** *good-set-def* **by** *auto*

show *?thesis*
proof (*cases 1≤k*)
case *False* **show** *?thesis*
proof (*rule FalseE*)
have *k=0* **using** *False* **by** *simp*
hence *f* ‘ {*i. i* < *k*} = {} **by** *auto*
hence *linear-combination g* (*f* ‘ {*i. i* < *k*})=$\mathbf{0}_V$ **by** *auto*
hence $\mathbf{0}_V$=*y* **using** *y-lc-g-f* **by** *simp*
thus *False* **using** *y-in-A* **and** *not-zero-A* **by** *auto*
qed
next
case *True*
note *one-le-k* = *True*
show *?thesis*
proof (*cases k*<*n*)
case *True* **show** *?thesis*
proof −
have (*indexing-ext* (*A*, *f*) *a n*) *k* = *f k*
using *True*
unfolding *indexing-ext-def* **by** *auto*
hence *1*:*indexing-ext* (*A*, *f*) *a n k* = *y* **using** *fk-y* **by** *simp*
have *indexing-ext* (*A*, *f*) *a n* ‘ {*i. i* < *k*} = *f*‘ {*i. i* < *k*}
using *True* **unfolding** *indexing-ext-def* **by** *auto*
hence *linear-combination g* (*indexing-ext* (*A*, *f*) *a n* ‘ {*i. i* < *k*})= *linear-combination g* (*f* ‘ {*i. i* < *k*})
using *arg-cong2* **by** *auto*
hence *2*: *y*=*linear-combination g* (*indexing-ext* (*A*, *f*) *a n* ‘ {*i. i* < *k*}) **using** *y-lc-g-f* **by** *auto*
have *k* < *card* (*insert a A*)
using *prems*(*5*) **and** *k-le-cardA* **and** *card-insert-if* **and** *finite-A* **by** *auto*
thus *?thesis* **using** *1 2 one-le-k y-in-A cf-g* **by** *force*
qed
next
case *False* **note** *k-ge-n* = *False* **show** *?thesis*
proof (*cases k*=*n*)
case *True* **show** *?thesis*
proof −
have *1*:*indexing-ext* (*A*, *f*) *a n* ‘ {*i. i* < *k*} = *f*‘ {*i. i* < *k*}
using *True* **unfolding** *indexing-ext-def* **by** *auto*
hence *linear-combination g* (*indexing-ext* (*A*, *f*) *a n* ‘ {*i. i* < *k*})= *linear-combination g* (*f* ‘ {*i. i* < *k*})
using *arg-cong2* **by** *auto*
hence *y*=*linear-combination g* (*indexing-ext* (*A*, *f*) *a n* ‘ {*i. i* < *k*})

using *y-lc-g-f* **by** *auto*
have *igualdad-conjuntos*: $\{i.\ i < (k+1)\}=\{i.\ i < k\}\cup\{i.\ i = k\}$ **by** *auto*
hence *indexing-ext* (A, f) *a n* ‘ $\{i.\ i < (k+1)\}$=*indexing-ext* (A, f) *a n* ‘ $(\{i.\ i < k\}\cup\{i.\ i = k\})$ **by** *auto*
also have ...=*indexing-ext* (A, f) *a n* ‘ $\{i.\ i < k\}$ $\cup$ *indexing-ext* (A, f) *a n* ‘ $\{i.\ i = k\}$ **by** *auto*
also have ...= f‘ $\{i.\ i < k\} \cup \{a\}$ **using** *1* **and** *True* **unfolding** *indexing-ext-def* **by** *auto*
finally have *2*:*indexing-ext* (A, f) *a n* ‘ $\{i.\ i < (k+1)\}$=f‘ $\{i.\ i < k\} \cup \{a\}$ **.**
hence *y-lc-h*: *y=linear-combination ?h* (*indexing-ext* (A, f) *a n* ‘ $\{i.\ i < (k+1)\}$)
proof −
have *linear-combination ?h* (*indexing-ext* (A, f) *a n* ‘ $\{i.\ i < (k+1)\}$)
=*linear-combination ?h* (f‘ $\{i.\ i < k\} \cup \{a\}$)
using *arg-cong2* **using** *2* **by** *auto*
also have ...=*linear-combination g* (f‘ $\{i.\ i < k\}$)
using *eq-lc-when-out-of-set-is-zero*[*OF cb-a cb-imf cf-g*] **by** *auto*
also have ...=*y* **using** *y-lc-g-f* **by** *simp*
finally show *?thesis* **by** *simp*
qed
have (*indexing-ext* (A, f) *a n*) $(k+1) = f\,k$
using *True*
unfolding *indexing-ext-def* **by** *auto*
hence *3*:(*indexing-ext* (A, f) *a n*) $(k+1) = y$ **using** *fk-y* **by** *simp*
show *?thesis* **using** *cf-h one-le-k-plus-one k-plus-one-le-card-insert-a-A 3 y-lc-h y-in-A* **by** *force*
−
qed
next
case *False* **show** *?thesis*
proof −
have *k-g-n*: $k>n$ **using** *False* **and** *k-ge-n* **by** *simp*
hence (*indexing-ext* (A, f) *a n*) $(k+1) = f\,k$
unfolding *indexing-ext-def* **by** *auto*
hence *indexing-ext-y*: (*indexing-ext* (A, f) *a n*) $(k+1) = y$ **using** *fk-y* **by** *simp*
have *1*:*indexing-ext* (A, f) *a n* ‘ $\{i.\ i < n\} = f$‘ $\{i.\ i < n\}$
unfolding *indexing-ext-def* **by** *auto*
have *2*:*indexing-ext* (A, f) *a n* ‘ $\{i.\ n < i \wedge i<(k+1)\} = f$‘ $\{i.\ n \le i \wedge i<k\}$
using *k-g-n* **unfolding** *indexing-ext-def* **unfolding** *iset-to-index-def*
unfolding *image-def*
proof *auto*
show $\bigwedge xa.\ [\![n < xa;\ xa < Suc\ k]\!] \Longrightarrow \exists\,x{\ge}n.\ x < k \wedge f\ (xa - Suc\ 0) = f\ x$

proof –
fix *xa*
assume *n-l-xa*: $n<xa$ **and** *xa-l-suc-k*: $xa<(Suc\ k)$
let $?x= xa-(Suc\ 0)$
have *1*:$f(xa - Suc\ 0)=f\ (?x)$ **by** *simp*
have *2*:$?x\geq n$ **using** *n-l-xa* **by** *auto*
have *3*:$?x < k$**using** *xa-l-suc-k*
by (*metis One-nat-def diff-less gr0I gr-implies-not0*
k-g-n less-imp-diff-less linorder-neqE-nat not-less-eq zero-less-Suc)
show $\exists x\geq n.\ x < k \wedge f\ (xa - Suc\ 0) = f\ x$ **using** *1* **and** *2* **and** *3* **by** *auto*
qed
show $\bigwedge xa.\ [\![n \leq xa;\ xa < k]\!] \Longrightarrow \exists x>n.\ x < Suc\ k \wedge f\ xa = f\ (x - Suc\ 0)$
proof –
fix *xa*
assume *n-le-xa*: $n\leq xa$ **and** *xa-l-k*: $xa<k$
let $?x= xa+(Suc\ 0)$
have *1*:$f(xa)=f\ (?x - Suc\ 0)$ **by** *simp*
have *2*:$?x>n$ **using** *n-le-xa* **by** *auto*
have *3*:$?x < Suc\ k$**using** *xa-l-k* **by** *auto*
show $\exists x>n.\ x < Suc\ k \wedge f\ xa = f\ (x - Suc\ 0)$ **using** *1* **and** *2* **and** *3* **by** *auto*
qed
qed
have $\{i.\ i < (k+1)\}=\{i.\ i < n\}\cup\{i.\ i = n\}\cup\{i.\ n<i \wedge i<(k+1)\}$ **using** *k-g-n* **by** *auto*
hence *indexing-ext* (A, f) *a n* ‘ $\{i.\ i < (k+1)\}=$*indexing-ext* (A, f) *a n* ‘ $(\{i.\ i < n\}$
$\cup\{i.\ i = n\}\cup\{i.\ n<i \wedge i<(k+1)\})$ **by** *auto*
also have ...=*indexing-ext* (A, f) *a n* ‘ $\{i.\ i < n\}$ $\cup$ *indexing-ext* (A, f) *a n* ‘ $\{i.\ i = n\}$
$\cup$ *indexing-ext* (A, f) *a n* ‘ $\{i.\ n<i \wedge i<(k+1)\}$ **by** *auto*
also have ...$= f‘\ \{i.\ i < n\} \cup \{a\} \cup f‘\{i.\ n\leq i \wedge i<k\}$ **using** *1 2 k-g-n* **unfolding** *indexing-ext-def* **by** *auto*
also have ...$=f‘\ \{i.\ i < k\} \cup \{a\}$
proof –
have $\{i.\ i < k\}=\{i.\ i < n\}\cup\{i.\ n\leq i \wedge i<k\}$ **using** *k-g-n* **by** *auto*
hence $f‘\{i.\ i < k\}=f‘\{i.\ i < n\}\cup f‘\{i.\ n\leq i \wedge i<k\}$ **by** *auto*
thus *?thesis* **by** *auto*
qed
finally have *3*: *indexing-ext* (A, f) *a n* ‘ $\{i.\ i < (k+1)\}=f‘\ \{i.\ i < k\} \cup \{a\}$ **.**
have *y-lc-h*: *y=linear-combination ?h* (*indexing-ext* (A, f) *a n* ‘ $\{i.\ i < (k+1)\})$
proof –
have *linear-combination ?h* (*indexing-ext* (A, f) *a n* ‘ $\{i.\ i < (k+1)\})$
=*linear-combination ?h* $(f‘\ \{i.\ i < k\} \cup \{a\})$

using *arg-cong2* **using** *3* **by** *auto*
also have ...=*linear-combination g* (f‘ $\{i.\ i < k\}$)
using *eq-lc-when-out-of-set-is-zero*[*OF cb-a cb-imf cf-g*] **by** *auto*
also have ...=y **using** *y-lc-g-f* **by** *simp*
finally show *?thesis* **by** *simp*
qed
show *?thesis*
using *cf-h one-le-k-plus-one k-plus-one-le-card-insert-a-A indexing-ext-y 3 y-lc-h y-in-A* **by** *force*
qed
qed
qed
qed
qed
next
case *False* **show** *?thesis*
proof −
have *li-A*: *linear-independent A* **using** *False* **and** *independent-if-only-if-not-dependent* **and** *good-set-A* **by** *simp*
from *prems*(*12*) **obtain** *h*
where *cf-h*: $h \in$ *coefficients-function* (*carrier V*)
and *sum-zero*: *linear-combination h* (*insert a A*)=$\mathbf{0}_V$
and *not-all-zero*: $\neg$ ($\forall x\in$*insert a A*. $h\ x = \mathbf{0}_K$)
unfolding *linear-dependent-def* **by** *auto*
have *1*:*indexing-ext* (*A*,*f*) *a n* ‘ {..<(*card*(*insert a A*))} = (*insert a A*) **using** *prems*(*8*)
unfolding *indexing-def* **unfolding** *bij-betw-def*
unfolding *iset-to-index-def* **by** *auto*
let *?A*={$k\in$ {..<*card* (*insert a A*)}. *h* ((*indexing-ext* (*A*,*f*) *a n*) *k*) $\neq \mathbf{0}_K$}
have *finite-A*: *finite ?A* **by** *auto*
have *A-not-empty*: *?A*$\neq${} **using** *not-all-zero* **using** *1* **by** *force*
def *m* == *Max ?A*
have *m-in-A*: $m\in$ *?A* **using** *Max.closed*[*OF finite-A A-not-empty*] **unfolding** *m-def* **by** *force*
have $\forall x\in$ {..<*card* (*insert a A*)}. (x<*card*(*insert a A*)) **by** *auto*
hence *m-le-card-aA*: m<(*card*(*insert a A*)) **using** *Max-less-iff* [*OF finite-A A-not-empty*] **unfolding** *m-def* **by** *auto*
have $\neg$ ($\exists x\in$*?A*. $m < x$) **using** *Max-less-iff* [*OF finite-A A-not-empty*] **unfolding** *m-def* **by** *auto*
hence *h-indexing-m-card-zero*: $\forall x\in$ {m<..<(*card*(*insert a A*))}. *h* ((*indexing-ext* (*A*,*f*) *a n*) *x*) = $\mathbf{0}_K$ **by** *auto*
have *indexing-m-in-aA*: *indexing-ext* (*A*,*f*) *a n m* $\in$ (*insert a A*) **using** *1* **using** *m-le-card-aA* **by** *auto*
have *descomposicion-conjunto*:{..<(*card*(*insert a A*))}= {..*m*} $\cup$ {*m*<..<(*card*(*insert a A*))}
using *m-le-card-aA* **unfolding** *m-def* **by** *auto*
have *indexing-ext* (*A*,*f*) *a n* ‘{..<(*card*(*insert a A*))}
= *indexing-ext* (*A*,*f*) *a n* ‘ ({..*m*}$\cup${*m*<..<(*card*(*insert a A*))})
unfolding *descomposicion-conjunto* **..**

also have...= *indexing-ext* (A,f) *a n* ‘ $\{..m\}$ ∪ *indexing-ext* (A,f) *a n* ‘$\{m<..<(card(insert\ a\ A))\}$ **by** *auto*
finally have *descomposicion-indexing-ext*: *indexing-ext* (A, f) *a n* ‘ $\{..<card\ (insert\ a\ A)\}$ =
indexing-ext (A, f) *a n* ‘ $\{..m\}$ ∪ *indexing-ext* (A, f) *a n* ‘ $\{m<..<card\ (insert\ a\ A)\}$ **.**
have *descomposicion-conjunto2*: $\{..m\}$=*insert m* $\{..<m\}$ **by** *auto*
hence *descomposicion-indexing-ext2*:*indexing-ext* (A, f) *a n* ‘ $\{..m\}$
=(*insert* (*indexing-ext* (A, f) *a n m*) (*indexing-ext* (A, f) *a n* ‘ $\{..<m\}$))
by *auto*
have *cb-l-m*: *good-set* (*indexing-ext* (A, f) *a n* ‘ $\{..m\}$)
proof −
have *indexing-ext* (A, f) *a n* ‘ $\{..m\}$
⊆ *indexing-ext* (A, f) *a n* ‘ $\{..<card\ (insert\ a\ A)\}$ **using** *m-le-card-aA* **by** *auto*
hence *indexing-ext* (A, f) *a n* ‘ $\{..m\}$⊆ (*insert a A*) **using** *1* **by** *simp*
thus *?thesis* **using** *good-set-aA* **unfolding** *good-set-def* **by** *auto*
qed
have *i-m-in-V*: *indexing-ext* (A, f) *a n m* ∈ *carrier V* **using** *cb-l-m* **unfolding** *good-set-def* **by** *auto*
have $\mathbf{0}_V$=*linear-combination h* (*indexing-ext* (A,f) *a n* ‘ $\{..<(card(insert\ a\ A))\}$) **using** *sum-zero 1* **by** *auto*
also have ...=*linear-combination h* (*indexing-ext* (A, f) *a n* ‘ $\{..m\}$
∪ *indexing-ext* (A, f) *a n* ‘ $\{m<..<card\ (insert\ a\ A)\}$)
using *descomposicion-indexing-ext* **by** *auto*
also have ...= *linear-combination h* (*indexing-ext* (A, f) *a n* ‘ $\{..m\}$)
$\oplus_V$ *linear-combination h* (*indexing-ext* (A, f) *a n* ‘ $\{m<..<card\ (insert\ a\ A)\}$)
proof (*unfold linear-combination-def* , *rule finsum-Un-disjoint*,*force*)
show *finite* (*indexing-ext* (A, f) *a n* ‘ $\{m<..<card\ (insert\ a\ A)\}$) **using** *m-le-card-aA* **by** *auto*
show *indexing-ext* (A, f) *a n* ‘ $\{..m\}$ ∩ *indexing-ext* (A, f) *a n* ‘ $\{m<..<card\ (insert\ a\ A)\}$ = {}
proof −
have *disjuntos*: $\{..m\}$ ∩ $\{m<..<(card(insert\ a\ A))\}$={} **by** *auto*
have *indexing-ext* (A,f) *a n* ‘ $\{..m\}$∩*indexing-ext* (A,f) *a n* ‘$\{m<..<(card(insert\ a\ A))\}$=
indexing-ext (A,f) *a n* ‘ ($\{..m\}$∩$\{m<..<(card(insert\ a\ A))\}$)
proof(*rule inj-on-image-Int*[*symmetric*])
show *inj-on* (*indexing-ext* (A,f) *a n*) $\{..<card(insert\ a\ A)\}$
using *prems*(*8*)
unfolding *indexing-def* **unfolding** *iset-to-set-def iset-to-index-def*
unfolding *bij-betw-def* **by** *simp*
show $\{..m\}$ ⊆ $\{..<card\ (insert\ a\ A)\}$ **using** *m-le-card-aA* **by** *auto*
show $\{m<..<card\ (insert\ a\ A)\}$ ⊆ $\{..<card\ (insert\ a\ A)\}$ **using** *m-le-card-aA* **by** *auto*
qed
also have ...={} **using** *disjuntos* **by** *simp*
finally show *?thesis* **.**

qed
show $(\lambda y.\ h\ y \cdot y) \in$ *indexing-ext* (A, f) *a n* ‘ $\{..m\} \to$ *carrier V*
proof (*auto*,*rule mult-closed*)
fix *x*
assume *x-le-m*: $x \le m$
show *indexing-ext* (A, f) *a n x* $\in$ *carrier V*
using *1* **and** *good-set-aA*
unfolding *good-set-def* **unfolding** *indexing-ext-def* **unfolding** *iset-to-index-def*
using *x-le-m* **and** *m-le-card-aA* **by** *auto*
thus *h* (*indexing-ext* (A, f) *a n x*) $\in$ *carrier K* **using** *cf-h* **unfolding** *coefficients-function-def* **by** *auto*
qed
show $(\lambda y.\ h\ y \cdot y) \in$ *indexing-ext* (A, f) *a n* ‘ $\{m<..<card\ (insert\ a\ A)\}$ $\to$ *carrier V*
proof (*auto*,*rule mult-closed*)
fix *x*
assume *m-le-x*: $m<x$
and *x-le-card-aA*: $x<card(insert\ a\ A)$
show *indexing-ext* (A, f) *a n x* $\in$ *carrier V*
using *1* **and** *good-set-aA*
unfolding *good-set-def* **unfolding** *indexing-ext-def* **unfolding** *iset-to-index-def*
using *m-le-x* **and** *x-le-card-aA* **by** *auto*
thus *h* (*indexing-ext* (A, f) *a n x*) $\in$ *carrier K* **using** *cf-h* **unfolding** *coefficients-function-def* **by** *auto*
qed
qed
also have ...= *linear-combination h* (*indexing-ext* (A, f) *a n* ‘ $\{..m\}$) $\oplus_V$ $\mathbf{0}_V$
proof −
have *linear-combination h* (*indexing-ext* (A, f) *a n* ‘ $\{m<..<card\ (insert\ a\ A)\}$)=$\mathbf{0}_V$
proof (*unfold linear-combination-def*)
have *hy-zero*: $\bigwedge y.\ [\![y \in indexing\text{-}ext\ (A, f)\ a\ n\ ‘\ \{m<..<card\ (insert\ a\ A)\}]\!] \Longrightarrow h\ y = \mathbf{0}_K$
using *h-indexing-m-card-zero* **by** *auto*
have $(\bigoplus_V y \in indexing\text{-}ext\ (A, f)\ a\ n\ ‘\ \{m<..<card\ (insert\ a\ A)\}.\ h\ y \cdot y) =$
$(\bigoplus_V y \in indexing\text{-}ext\ (A, f)\ a\ n\ ‘\ \{m<..<card\ (insert\ a\ A)\}.\ \mathbf{0}_V)$
proof (*rule finsum-cong'*)
show *indexing-ext* (A, f) *a n* ‘ $\{m<..<card\ (insert\ a\ A)\}$
$=$ *indexing-ext* (A, f) *a n* ‘ $\{m<..<card\ (insert\ a\ A)\}$ **..**
show $(\lambda y.\ \mathbf{0}_V) \in$ *indexing-ext* (A, f) *a n* ‘ $\{m<..<card\ (insert\ a\ A)\}$ $\to$ *carrier V* **by** *auto*
show $\bigwedge i.\ i \in indexing\text{-}ext\ (A, f)\ a\ n\ ‘\ \{m<..<card\ (insert\ a\ A)\} \Longrightarrow h\ i \cdot i = \mathbf{0}_V$
proof −
fix *i*

assume *i-in-indexing*: $i \in$ *indexing-ext* (A, f) a n ' $\{m<..<card$ $(insert\ a\ A)\}$
show $h\ i \cdot i = \mathbf{0}_V$
proof −
have *hi-zero*:$h(i)$=$\mathbf{0}_K$ **using** *hy-zero*[*OF i-in-indexing*] **.**
have *i-in-V*: $i\in$ *carrier V* **using** *1*
proof −
have *indexing-ext* (A, f) a n ' $\{m<..<card$ $(insert\ a\ A)\}$
$\subseteq$ *indexing-ext* (A, f) a n ' $\{..<card$ $(insert\ a\ A)\}$ **using** *m-le-card-aA* **by** *auto*
hence *indexing-ext* (A, f) a n ' $\{m<..<card$ $(insert\ a\ A)\} \subseteq$ $(insert\ a\ A)$ **using** *1* **by** *simp*
thus *?thesis* **using** *i-in-indexing* **and** *good-set-aA* **unfolding** *good-set-def* **by** *auto*
qed
show *?thesis* **using** *zeroK-mult-V-is-zeroV* **and** *hi-zero* **and** *i-in-V* **by** *auto*
qed
qed
qed
also have ...=$\mathbf{0}_V$
proof (*rule finsum-zero*)
show *finite* (*indexing-ext* (A, f) a n ' $\{m<..<card$ $(insert\ a\ A)\})$
proof −
have *indexing-ext* (A, f) a n ' $\{m<..<card$ $(insert\ a\ A)\}$
$\subseteq$ *indexing-ext* (A, f) a n ' $\{..<card$ $(insert\ a\ A)\}$ **using** *m-le-card-aA* **by** *auto*
hence *indexing-ext* (A, f) a n ' $\{m<..<card$ $(insert\ a\ A)\}\subseteq$ $(insert\ a\ A)$ **using** *1* **by** *simp*
thus *?thesis* **using** *good-set-aA* **unfolding** *good-set-def* **by** *auto*
qed
qed
finally show $(\bigoplus_V y\in$*indexing-ext* (A, f) a n ' $\{m<..<card$ $(insert\ a\ A)\}.\ h\ y \cdot y) = \mathbf{0}_V$ **.**
qed
thus *?thesis* **by** *auto*
qed
also have ...=*linear-combination* h (*indexing-ext* (A, f) a n ' $\{..m\})$
proof (*rule V.r-zero*, *rule linear-combination-closed*)
show *good-set* (*indexing-ext* (A, f) a n ' $\{..m\})$ **using** *cb-l-m* **.**
show $h \in$ *coefficients-function* (*carrier V*) **using** *cf-h* **.**
qed
also have ...=h (*indexing-ext* (A, f) a n m) $\cdot$ (*indexing-ext* (A, f) a n m)
$\oplus_V$ *linear-combination* h (*indexing-ext* (A, f) a n ' $\{..<m\})$
proof −
have *linear-combination* h (*indexing-ext* (A, f) a n ' $\{..m\})$
= *linear-combination* h ((*insert* (*indexing-ext* (A, f) a n m) (*indexing-ext* (A, f) a n ' $\{..<m\})))$
using *arg-cong2* **and** *descomposicion-indexing-ext2* **by** *auto*

```
also have ...
  = h (indexing-ext (A, f) a n m) · indexing-ext (A, f) a n m
  ⊕V linear-combination h (indexing-ext (A, f) a n ‘ {..<m})
proof (rule linear-combination-insert)
show good-set (indexing-ext (A, f) a n ‘ {..<m}) using cb-l-m unfolding
good-set-def by auto
  show indexing-ext (A, f) a n m ∈ carrier V using i-m-in-V .
  show indexing-ext (A, f) a n m ∉ indexing-ext (A, f) a n ‘ {..<m}
  proof −
    have inj-on-m: inj-on (indexing-ext (A, f) a n) {..m}
      proof (rule subset-inj-on)
        show inj-on (indexing-ext (A, f) a n) {..<card(insert a A)}
          using prems(8) unfolding indexing-def unfolding bij-betw-def
by auto
        show {..m}⊆{..<card(insert a A)} using m-le-card-aA by auto
        qed
    hence auxiliar:{indexing-ext (A, f) a n m} ⊆ indexing-ext (A, f) a n
‘ {m} by auto
    have d1:{..m}={..<m}∪{m} by auto
    have {..<m}∩{m}={} by auto
    hence disjuntos: (indexing-ext (A, f) a n) ‘ {..<m} ∩ (indexing-ext
(A, f) a n)‘ {m}={}
      using inj-on-m and d1 by auto
    show ?thesis
      proof (cases indexing-ext (A, f) a n m ∉ indexing-ext (A, f) a n ‘
{..<m})
        case True thus ?thesis .
        next
        case False show ?thesis
          proof (rule FalseE)
            have indexing-ext (A, f) a n m ∈ indexing-ext (A, f) a n ‘
{..<m} using False by auto
            thus False using auxiliar and disjuntos by auto
            qed
  qed
qed
  show h ∈ coefficients-function (carrier V) using cf-h .
qed
finally show ?thesis by auto
qed
  finally have descomposicion-lc: 0V=h (indexing-ext (A, f) a n m) ·
indexing-ext (A, f) a n m
  ⊕V linear-combination h (indexing-ext (A, f) a n ‘ {..<m}) .
have ∃w. w ∈ coefficients-function (carrier V)
  ∧ linear-combination w (indexing-ext (A, f) a n ‘ {..<m}) = indexing-ext
(A, f) a n m
proof (rule work-out-the-value-of-x)
 show good-set (indexing-ext (A, f) a n ‘ {..<m}) using cb-l-m unfolding
good-set-def by auto
```

```
        show h ∈ coefficients-function (carrier V) using cf-h .
         show indexing-ext (A, f) a n m ∈ carrier V using cb-l-m unfolding
good-set-def by auto
        show h (indexing-ext (A, f) a n m) ≠ 0 using m-in-A by simp
        show 0_V = h (indexing-ext (A, f) a n m) · indexing-ext (A, f) a n m
            ⊕_V linear-combination h (indexing-ext (A, f) a n ` {..<m}) using
descomposicion-lc .
      qed
      from this obtain w where cf-w: w ∈ coefficients-function (carrier V) and
      lc-w: linear-combination w (indexing-ext (A, f) a n ` {..<m}) = indexing-ext
(A, f) a n m by auto
      have one-le-m: 1≤m
      proof (cases 1≤m)
        case True thus ?thesis .
      next
        case False show ?thesis
        proof (rule FalseE)
          have m-zero: m=0 using False by auto
          hence not-zero:indexing-ext (A, f) a n m ≠ 0_V using m-in-A
            by (metis 1 insert(9) imageI lessThan-iff m-le-card-aA)
         have zero: linear-combination w (indexing-ext (A, f) a n ` {..< m})=0_V
using m-zero by auto
          show False using lc-w and zero and not-zero by auto
        qed
      qed

      let ?y=indexing-ext (A, f) a n m
      have {i. i < m}={..<m} by auto
       hence ?y = linear-combination w (indexing-ext (A, f) a n ` {i. i < m})
using lc-w by auto
    thus ?thesis using cf-w and one-le-m and m-le-card-aA and indexing-m-in-aA
by force
    qed
  qed
 qed
 next
 show finite Ausing l-dep-good-set [OF ld-A] unfolding good-set-def by simp
qed
```

The proof can be also done without induction and then the proof of the theorem is shorter: "only" 200 code lines. The proof is a generalization of one of the cases in the induction above.

```
theorem
 linear-dependent-set-sorted-contains-linear-combination2:
 assumes ld-A: linear-dependent A
 and not-zero: 0_V ∉ A
 and i: indexing (A, f)
 shows ∃ y∈A. ∃ g. ∃ k::nat.
 g ∈ coefficients-function (carrier V)
```

$\wedge$ (*1::nat*) $\leq k \wedge k <$ (*card A*)
$\wedge$ *f k* $=$ *y* $\wedge$ *y* $=$ *linear-combination g* (*f*‘{*i::nat. i*$<$*k*})
proof $-$
have *good-set-A*: *good-set A* **using** *l-dep-good-set*[*OF ld-A*] **.**
from *ld-A* **obtain** *h*
where *cf-h*: *h* $\in$ *coefficients-function* (*carrier V*)
and *sum-zero*: *linear-combination h A*$=\mathbf{0}_V$
and *not-all-zero*: $\neg$ ($\forall$ *x*$\in$*A. h x* $=\mathbf{0}_K$)
unfolding *linear-dependent-def* **by** *auto*
have *1*: *f* ‘ {..$<$(*card A*)} $=$ *A* **using** *i*
unfolding *indexing-def* **unfolding** *bij-betw-def*
unfolding *iset-to-index-def* **by** *auto*
let *?A*$=${*k*$\in${..$<$*card A*}. *h* (*f k*) $\neq \mathbf{0}_K$}
have *finite-A*: *finite ?A* **by** *auto*
have *A-not-empty*: *?A*$\neq${} **using** *not-all-zero* **using** *1* **by** *force*
def *m* $\equiv$ *Max ?A*
have *m-in-A*: *m* $\in$ *?A* **using** *Max.closed*[*OF finite-A A-not-empty*] **unfolding** *m-def* **by** *force*
have $\forall$ *x*$\in${..$<$*card A*}. (*x*$<$*card A*) **by** *auto*
hence *m-le-card-aA*: *m*$<$(*card A*) **using** *Max-less-iff* [*OF finite-A A-not-empty*] **unfolding** *m-def* **by** *auto*
have $\neg$ ($\exists$ *x*$\in$*?A. m* $<$ *x*) **using** *Max-less-iff* [*OF finite-A A-not-empty*] **unfolding** *m-def* **by** *auto*
hence *h-indexing-m-card-zero*: $\forall$ *x*$\in${*m*$<$..$<$(*card A*)}. *h* (*f x*) $= \mathbf{0}_K$ **by** *auto*
have *indexing-m-in-aA*: *f m* $\in$ *A* **using** *1* **using** *m-le-card-aA* **by** *auto*
have *descomposicion-conjunto*:{..$<$(*card A*)} $=$ {..*m*} $\cup$ {*m*$<$..$<$(*card A*)}
using *m-le-card-aA* **unfolding** *m-def* **by** *auto*
have *f* ‘{..$<$(*card A*)}
$=$ *f* ‘ ({..*m*}$\cup${*m*$<$..$<$(*card A*)})
unfolding *descomposicion-conjunto* **..**
also have...$=$ *f* ‘ {..*m*} $\cup$ *f* ‘{*m*$<$..$<$(*card*(*A*))} **by** *auto*
finally have *descomposicion-indexing-ext*: *f* ‘ {..$<$*card A*} $=$
f ‘ {..*m*} $\cup$ *f* ‘ {*m*$<$..$<$*card A*} **.**
have *descomposicion-conjunto2*: {..*m*}$=$*insert m* {..$<$*m*} **by** *auto*
hence *descomposicion-indexing-ext2*: *f* ‘ {..*m*} $=$ (*insert* (*f m*) (*f* ‘ {..$<$*m*}))
by *auto*
have *cb-l-m*: *good-set* (*f* ‘ {..*m*})
proof $-$
have *f* ‘ {..*m*} $\subseteq$ *f* ‘ {..$<$*card* (*A*)} **using** *m-le-card-aA* **by** *auto*
hence *f* ‘ {..*m*} $\subseteq$ *A* **using** *1* **by** *simp*
thus *?thesis* **using** *good-set-A* **unfolding** *good-set-def* **by** *auto*
qed
have *i-m-in-V*: *f m* $\in$ *carrier V* **using** *cb-l-m* **unfolding** *good-set-def* **by** *auto*

have $\mathbf{0}_V$$=$*linear-combination h* (*f* ‘ {..$<$*card A*}) **using** *sum-zero 1* **by** *auto*
also have ...$=$*linear-combination h* (*f* ‘ {..*m*} $\cup$ *f* ‘ {*m*$<$..$<$*card A*})
using *descomposicion-indexing-ext* **by** *auto*
also have ...$=$ *linear-combination h* (*f* ‘ {..*m*})
$\oplus_V$ *linear-combination h* (*f* ‘ {*m*$<$..$<$*card A*})

proof (*unfold linear-combination-def*, *rule finsum-Un-disjoint*,*force*)
show *finite* (f ‘ {m<..<*card* A}) **using** *m-le-card-aA* **by** *auto*
show f ‘ {..m} $\cap$ f ‘ {m<..<*card* A} = {}
proof −
have *disjuntos*: {..m} $\cap$ {m<..<(*card*(A))}={} **by** *auto*
have f ‘ {..m}$\cap$$f$ ‘{m<..<(*card*(A))}=
f ‘ ({..m}$\cap${m<..<(*card*(A))})
proof(*rule inj-on-image-Int*[*symmetric*])
show *inj-on* f {..<*card*(A)}
using *i*
unfolding *indexing-def* **unfolding** *iset-to-set-def iset-to-index-def*
unfolding *bij-betw-def* **by** *simp*
show {..m} $\subseteq$ {..<*card* A} **using** *m-le-card-aA* **by** *auto*
show {m<..<*card* (A)} $\subseteq$ {..<*card* (A)} **using** *m-le-card-aA* **by** *auto*
qed
also have ...={} **using** *disjuntos* **by** *simp*
finally show *?thesis* **.**
qed
show ($\lambda y.\ h\ y \cdot y$) $\in$ f ‘ {..m} $\rightarrow$ *carrier* V
proof (*auto*,*rule mult-closed*)
fix x
assume *x-le-m*: $x \le m$
show $f\,x$ $\in$ *carrier* V
using *1* **and** *good-set-A*
unfolding *good-set-def* **unfolding** *indexing-ext-def* **unfolding** *iset-to-index-def*

using *x-le-m* **and** *m-le-card-aA* **by** *auto*
thus h ($f\,x$) $\in$ *carrier* K **using** *cf-h* **unfolding** *coefficients-function-def* **by** *auto*
qed
show ($\lambda y.\ h\ y \cdot y$) $\in$ f ‘ {m<..<*card* (A)} $\rightarrow$ *carrier* V
proof (*auto*,*rule mult-closed*)
fix x
assume *m-le-x*: $m<x$
and *x-le-card-aA*: x<*card*(A)
show $f\,x$ $\in$ *carrier* V
using *1* **and** *good-set-A*
unfolding *good-set-def* **unfolding** *indexing-ext-def* **unfolding** *iset-to-index-def*

using *m-le-x* **and** *x-le-card-aA* **by** *auto*
thus h ($f\,x$) $\in$ *carrier* K **using** *cf-h* **unfolding** *coefficients-function-def* **by** *auto*
qed
qed
also have ...= *linear-combination* h (f ‘ {..m}) $\oplus_V$ $\mathbf{0}_V$
proof −
have *linear-combination* h (f ‘ {m<..<*card* (A)})=$\mathbf{0}_V$
proof (*unfold linear-combination-def*)
have *hy-zero*: $\bigwedge y.$ $[\![y \in f$ ‘ {m<..<*card* (A)}$]\!] \Longrightarrow h\ y = \mathbf{0}_K$

```
        using h-indexing-m-card-zero by auto
      have (⨁Vy∈f ‘ {m<..<card (A)}. h y · y) =
        (⨁Vy∈f ‘ {m<..<card (A)}. 𝟬V)
      proof (rule finsum-cong')
        show f ‘ {m<..<card (A)}
          = f ‘ {m<..<card (A)} ..
        show (λy. 𝟬V) ∈ f ‘ {m<..<card (A)} → carrier V by auto
        show ⋀i. i ∈ f ‘ {m<..<card (A)} ⟹ h i · i = 𝟬V
        proof −
          fix i
          assume i-in-indexing: i ∈ f ‘ {m<..<card (A)}
          show h i · i = 𝟬V
          proof −
            have hi-zero:h(i)=𝟬K using hy-zero[OF i-in-indexing] .
            have i-in-V: i∈ carrier V using 1
            proof −
              have f ‘ {m<..<card (A)}
                ⊆ f ‘ {..<card (A)} using m-le-card-aA by auto
              hence f ‘ {m<..<card A}⊆ A using 1 by simp
            thus ?thesis using i-in-indexing and good-set-A unfolding good-set-def
by auto
            qed
            show ?thesis using zeroK-mult-V-is-zeroV and hi-zero and i-in-V by
auto
          qed
        qed
      qed
      also have ...=𝟬V
      proof (rule finsum-zero)
        show finite (f ‘ {m<..<card (A)})
        proof −
          have f ‘ {m<..<card (A)}
            ⊆ f ‘ {..<card (A)} using m-le-card-aA by auto
          hence f ‘ {m<..<card (A)}⊆ A using 1 by simp
          thus ?thesis using good-set-A unfolding good-set-def by auto
        qed
      qed
      finally show (⨁Vy∈f ‘ {m<..<card (A)}. h y · y) = 𝟬V .
    qed
    thus ?thesis by auto
  qed
  also have ...=linear-combination h (f‘ {..m})
  proof (rule V.r-zero, rule linear-combination-closed)
    show good-set (f ‘ {..m}) using cb-l-m .
    show h ∈ coefficients-function (carrier V) using cf-h .
  qed
  also have ...=h (f m) · (f m)
    ⊕V linear-combination h (f ‘ {..<m})
  proof −
```

```
  have linear-combination h (f ‘ {..m})
    = linear-combination h ((insert (f m) (f ‘ {..<m})))
    using arg-cong2 and descomposicion-indexing-ext2 by auto
  also have ...
    =h (f m) · f m ⊕V linear-combination h (f‘ {..<m})
  proof (rule linear-combination-insert)
    show good-set (f ‘ {..<m}) using cb-l-m unfolding good-set-def by auto
    show f m ∈ carrier V using i-m-in-V .
    show f m ∉ f ‘ {..<m}
    proof −
      have inj-on-m: inj-on f {..m}
      proof (rule subset-inj-on)
        show inj-on f {..<card(A)}
          using i unfolding indexing-def unfolding bij-betw-def by auto
        show {..m}⊆{..<card(A)} using m-le-card-aA by auto
      qed
      hence auxiliar:{f m} ⊆ f ‘ {m} by auto
      have d1:{..m}={..<m}∪{m} by auto
      have {..<m}∩{m}={} by auto
      hence disjuntos: f‘ {..<m} ∩ f‘ {m}={} using inj-on-m and d1 by auto
      show ?thesis
      proof (cases f m ∉ f ‘ {..<m})
        case True thus ?thesis .
      next
        case False show ?thesis
        proof (rule FalseE)
          have f m ∈ f ‘ {..<m} using False by auto
          thus False using auxiliar and disjuntos by auto
        qed
      qed
    qed
    show h ∈ coefficients-function (carrier V) using cf-h .
  qed
  finally show ?thesis by auto
qed
finally have descomposicion-lc: 0V=h (f m) · f m
  ⊕V linear-combination h (f ‘ {..<m}) .
have ∃ w. w ∈ coefficients-function (carrier V)
  ∧ linear-combination w (f ‘ {..<m}) = f m
proof (rule work-out-the-value-of-x)
  show good-set (f ‘ {..<m}) using cb-l-m unfolding good-set-def by auto
  show h ∈ coefficients-function (carrier V) using cf-h .
  show f m ∈ carrier V using cb-l-m unfolding good-set-def by auto
  show h (f m) ≠ 0 using m-in-A by simp
  show 0V = h (f m) · f m
    ⊕V linear-combination h (f ‘ {..<m}) using descomposicion-lc .
qed
from this obtain w where cf-w: w ∈ coefficients-function (carrier V) and
  lc-w: linear-combination w (f ‘ {..<m}) = f m by auto
```

```
  have one-le-m: 1≤m
  proof (cases 1≤m)
    case True thus ?thesis .
  next
    case False show ?thesis
    proof (rule FalseE)
      have m-zero: m=0 using False by auto
      hence not-zero:f m ≠ 0_V using m-in-A
        by (metis indexing-m-in-aA not-zero)
      have zero: linear-combination w (f ` {..< m})=0_V using m-zero by auto
      show False using lc-w and zero and not-zero by auto
    qed
  qed
  let ?y=f m
  have {i. i < m}={..<m} by auto
  hence ?y = linear-combination w (f ` {i. i < m}) using lc-w by auto
  thus ?thesis using cf-w and one-le-m and m-le-card-aA and indexing-m-in-aA
by force
qed

end
end
theory Basis
imports Linear-combinations
begin
```

9 Basis

```
context vector-space
begin
```

A finite spanning set is a finite set of vectors that can generate every vector in the space through such linear combinations.

```
definition spanning-set :: 'b set ⇒ bool
  where spanning-set X = (good-set X
  ∧ (∀ x. x ∈ carrier V ⟶ (∃ f. f ∈ coefficients-function (carrier V) ∧ linear-combination
f X = x)))
```

Even, we can talk about an infinite spanning set. We say that a set (finite or infinite) $X \subseteq$ *carrier V* is a spanning set (we will rename this definition as *spanning-set-ext*) if for every $x \in$ *carrier V* it is possible to choose a finite subset of X such that exists a linear combination of its elements equal to x.

As we have said before, the sums are all finite: we can not talk about an infinite sum of vectors without adding some concepts and more structure (the axioms of Vector Space do not allow it).

```
definition spanning-set-ext :: 'b set ⇒ bool
  where spanning-set-ext X = (∀ x. x ∈ carrier V ⟶
```

($\exists A. \exists f.$ *good-set* $A \wedge A \subseteq X \wedge f \in$ *coefficients-function* (*carrier* V) $\wedge$ *linear-combination* $f\ A = x$))

Let's see the compatibility between the definitions:

Now we prove that every *spanning-set* is a *spanning-set-ext*:

lemma *spanning-imp-spanning-ext*:
  assumes *sp-X*: *spanning-set X*
  shows *spanning-set-ext X*
  unfolding *spanning-set-ext-def*
  using *sp-X*
  by (*auto simp add*: *mem-def spanning-set-def subset-refl*)

Whenever we have a *spanning-set-ext* which is finite and $X \subseteq$ *carrier* V then it is a *spanning-set*.

lemma *gs-spanning-ext-imp-spanning*:
  assumes *sp-X*: *spanning-set-ext X*
  and *gs-X*: *good-set X*
  shows *spanning-set X*
proof (*unfold spanning-set-def*, *rule conjI*)
  show *good-set X* **using** *gs-X* **.**
  show $\forall x.\ x \in$ *carrier V*
    $\longrightarrow$ ($\exists f.\ f \in$ *coefficients-function* (*carrier V*)
    $\wedge$ *linear-combination* $f\ X = x$)
  proof (*auto*)
    fix x
    assume *x-in-V*: $x \in$ *carrier V*
    from *sp-X* **obtain** A **and** f **where** *A-in-X*: $A \subseteq X$
      and *gs-A*: *good-set A*
      and *cf-f*: $f \in$ *coefficients-function* (*carrier V*)
      and *lc-fA*: *linear-combination* $f\ A = x$
      unfolding *spanning-set-ext-def* **using** *x-in-V* **by** *blast*
    def $g \equiv (\lambda x.$ *if* $x \in A$ *then* $f\ x$ *else* $\mathbf{0})$
    have *cf-g*: $g \in$ *coefficients-function* (*carrier V*)
      using *cf-f*
      unfolding *coefficients-function-def g-def* **by** *force*
    have *linear-combination* $g\ X = x$
    proof −
      have x=*linear-combination* $f\ A$ **using** *lc-fA* **by** *blast*
      also have ...=*linear-combination* g ($A \cup X$) **unfolding** *g-def*
      proof (*rule eq-lc-when-out-of-set-is-zero*[*symmetric*])
        show *good-set X* **using** *gs-X* **.**
        show *good-set A* **using** *gs-A* **.**
        show $f \in$ *coefficients-function* (*carrier V*) **using** *cf-f* **.**
      qed
      also have ...=*linear-combination* $g\ X$
        using *arg-cong2* [*of g g* $A \cup X$ *X linear-combination*]
        using *A-in-X* **by** *fast*
      finally show *?thesis* **by** *fast*

qed
thus $\exists g.\ g \in$ *coefficients-function* (*carrier* V)
$\wedge$ *linear-combination* $g\ X = x$ **using** *cf-g* **by** *auto*
qed
qed

A basis is an independent spanning set. We define it in general (X could be finite or infinite).

definition *basis* :: $'b$ *set* $\Rightarrow$ *bool*
where *basis* $X = (X \subseteq$ *carrier* $V \wedge$ *linear-independent-ext* $X \wedge$ *spanning-set-ext* $X)$

If we have a finite basis, then it is a good set.

lemma *finite-basis-implies-good-set*:
assumes *basis-B*: *basis* B
and *finite-B*: *finite* B
shows *good-set* B
using *basis-B finite-B* **unfolding** *basis-def good-set-def* **by** *fast*

We introduce the definition of *span* of a determinated set A like the set of all elements which can be expressed as a linear combination of the elements of A.

definition *span* :: $'b$ *set* $=>$ $'b$ *set*
where *span* $A = \{x.\ \exists g \in$ *coefficients-function* (*carrier* V). $x =$ *linear-combination* $g\ A\}$

First of all, we prove the behavior of *span* with respect to $\{\}$.

lemma
span-empty [*simp*]:
shows *span* $\{\} = \{\mathbf{0}_V\}$
unfolding *span-def*
unfolding *linear-combination-def*
using *V.finsum-empty*
unfolding *coefficients-function-def* **by** *auto*

One auxiliar result says that $\mathbf{0}_V$ is in the span of every set.

lemma
span-contains-zero [*simp*]:
assumes *fin-A*: *finite* A
and *A-in-V*: $A \subseteq$ *carrier* V
shows $\mathbf{0}_V \in$ *span* A
proof $-$
have $\mathbf{0}_V =$ *linear-combination* $(\lambda x.\ \mathbf{0}_K)\ A$
proof (*unfold linear-combination-def*,
subst finsum-zero [*symmetric*, *OF fin-A*], — Be careful applying unfold, we enter in a loop.
rule finsum-cong')
show $A = A$ **by** (*rule refl*)

show $op \cdot \mathbf{0} \in A \to carrier\ V$
unfolding *Pi-def*
using *mult-closed* **using** *A-in-V* **by** *auto*
show $\bigwedge i.\ i \in A \Longrightarrow \mathbf{0}_V = \mathbf{0} \cdot i$
using *zeroK-mult-V-is-zeroV* **using** *A-in-V* **by** *auto*
qed
thus *?thesis*
unfolding *span-def*
unfolding *coefficients-function-def*
unfolding *Pi-def* **using** *zero-closed* **by** *auto*
qed

Now we are going to prove that if we remove an element of a set which is a linear combination of the rest of elements then the span of the set is the same than the span of the set minus the element. This will be a fundamental property to be applied in the future. First of all, we do two auxiliar proofs.

This auxiliary lemma claims that given a coefficients funcion g of $A - \{a\}$ hence there exists another one (denoted by ga) such that *linear-combination* $g\ (A - \{a\})$ = *linear-combination* $ga\ A$. The coefficients function ga will be defined as follows: $\lambda x.\ if\ x = a\ then\ \mathbf{0}\ else\ g\ x$.

lemma *exists-function-Aa-A*:
assumes *cf-g*: $g \in coefficients\text{-}function\ (carrier\ V)$
and *good-set-A*: *good-set A*
and *a-in-A*: $a \in A$
shows $\exists\, ga \in coefficients\text{-}function\ (carrier\ V).$
$(\bigoplus_V y{\in}A - \{a\}.\ g\ y \cdot y) = (\bigoplus_V y{\in}A.\ ga\ y \cdot y)$
proof
let $?f{=}(\%x.\ if\ x{=}a\ then\ \mathbf{0}_K\ else\ g(x))$
show *cf-f*: $?f \in coefficients\text{-}function\ (carrier\ V)$
using *cf-g* **unfolding** *coefficients-function-def* **unfolding** *Pi-def* **by** *auto*
show $(\bigoplus_V y{\in}A - \{a\}.\ g\ y \cdot y) = (\bigoplus_V y{\in}A.\ ?f\ y \cdot y)$
proof −
have *A-in-V*: $A \subseteq carrier\ V$ **using** *good-set-A* **unfolding** *good-set-def* **by** *simp*

hence *a-in-V*: $a \in carrier\ V$ **using** *a-in-A* **by** *auto*
have *good-set-Aa*: *good-set* $(A - \{a\})$
using *good-set-A* **unfolding** *good-set-def* **by** *auto*
have *Aa-in-V*: $(A - \{a\}) \subseteq carrier\ V$ **using** *A-in-V* **by** *auto*
have *insert-aA*: $(insert\ a\ (A - \{a\}))=A$ **using** *a-in-A* **by** *auto*
have $(\bigoplus_V y{\in}(insert\ a\ (A{-}\{a\})).\ ?f\ y \cdot y){=}\ ?f(a){\cdot}a\ \oplus_V (\bigoplus_V y{\in}A{-}\{a\}.\ ?f$
$y \cdot y)$
proof (*rule finsum-insert*)
show *finite* $(A - \{a\})$ **using** *good-set-A* **unfolding** *good-set-def* **by** *simp*
show $a \notin A - \{a\}$ **by** *simp*
show $(\lambda y.\ (if\ y = a\ then\ \mathbf{0}\ else\ g\ y) \cdot y) \in A - \{a\} \to carrier\ V$
using *A-in-V cf-g* **unfolding** *coefficients-function-def*
unfolding *Pi-def* **using** *mult-closed* **by** *auto*
show $(if\ a = a\ then\ \mathbf{0}\ else\ g\ a) \cdot a \in carrier\ V$

using *mult-closed*[*OF a-in-V K.zero-closed*] **by** *auto*
qed
also have ...$=\mathbf{0}_V \oplus_V (\bigoplus_V y{\in}A{-}\{a\}.\ ?f\ y \cdot y)$ **using** *zeroK-mult-V-is-zeroV*[*OF a-in-V*] **by** *auto*
also have ...$=(\bigoplus_V y{\in}A{-}\{a\}.\ ?f\ y \cdot y)$
using *V.l-zero*[*OF linear-combination-closed* [*OF good-set-Aa cf-f*]]
unfolding *linear-combination-def* **.**
also have ...$=(\bigoplus_V y{\in}A{-}\{a\}.\ g\ y \cdot y)$
proof (*rule finsum-cong'*)
show $A - \{a\} = A - \{a\}$ **..**
show $(\lambda y.\ g\ y \cdot y) \in A - \{a\} \rightarrow$ *carrier V*
using *Aa-in-V* **using** *cf-g*
unfolding *Pi-def* **unfolding** *coefficients-function-def*
using *mult-closed* **by** *auto*
show $\bigwedge i.\ i \in A - \{a\} \Longrightarrow$ (*if* $i = a$ *then* $\mathbf{0}$ *else* $g\ i) \cdot i = g\ i \cdot i$ **by** *auto*
qed
finally show *?thesis* **using** *insert-aA* **by** *auto*
qed
qed

This auxiliary lemma is similar to the previous one. It claims that given a coefficients function h and another one g such that $a =$ *linear-combination* g $(A - \{a\})$, there exists a coefficients function ga such that *linear-combination* h $A =$ *linear-combination* ga $(A - \{a\})$. This coefficients funcion ga is defined as follows: $\lambda x.\ h\ a \otimes g\ x \oplus h\ x$. In other words, with these premises every linear combination of elements of A can be expressed as a linear combination of elements of $A - \{a\}$.

lemma *exists-function-A-Aa*:
assumes *cf-h*:$h \in$ *coefficients-function* (*carrier V*)
and *cf-g*: $g \in$ *coefficients-function* (*carrier V*)
and *a-lc-g-Aa*: $a =$ *linear-combination* g $(A{-}\{a\})$
and *good-set-A*: *good-set A* **and** *a-in-A*: $a{\in}A$
shows $\exists\, ga \in$ *coefficients-function* (*carrier V*).
$(\bigoplus_V y{\in}A.\ h\ y \cdot y) = (\bigoplus_V y{\in}A - \{a\}.\ ga\ y \cdot y)$
proof
let $?f = (\%x.\ (h\ a \otimes g\ x) \oplus_K h\ x)$
have *cb-Aa*: *good-set* $(A - \{a\})$
using *a-in-A* **and** *good-set-A* **unfolding** *good-set-def* **by** *auto*
have *a-in-V*: $a \in$ *carrier V*
using *a-in-A* **and** *good-set-A* **unfolding** *good-set-def* **by** *auto*
have *A-in-V*: $A{\subseteq}$ *carrier V*
using *good-set-A* **unfolding** *good-set-def* **by** *auto*
have *igualdad-conjuntos*: *insert a* $(A{-}\{a\}){=}A$ **using** *a-in-A* **by** *auto*
show *cf-f*: $?f \in$ *coefficients-function* (*carrier V*)
proof (*unfold coefficients-function-def*, *unfold Pi-def*, *auto*)
fix x
assume *x-in-V*: $x{\in}$ *carrier V*
hence $(h\ a \otimes g\ x){\in}$ *carrier K*

using *cf-g cf-h a-in-V* **unfolding** *coefficients-function-def* **using** $K.m\text{-}closed$ **by** *auto*
thus $(h\ a \otimes g\ x) \oplus h\ x \in carrier\ K$
using $K.a\text{-}closed$ [*OF* - *fx-in-K* [*OF x-in-V cf-h*]] **by** *auto*
next
fix x
assume *x-notin-V*: $x \notin carrier\ V$
have $h\ a \otimes g\ x \oplus h\ x = h\ a \otimes g\ x \oplus \mathbf{0}$
using *cf-h* **unfolding** *coefficients-function-def* **using** *x-notin-V* **by** *simp*
also have ...$= h\ a \otimes \mathbf{0} \oplus \mathbf{0}$
using *cf-g* **unfolding** *coefficients-function-def* **using** *x-notin-V* **by** *simp*
also have ...$= \mathbf{0} \oplus \mathbf{0}$ **using** $K.r\text{-}null$[*OF fx-in-K*[*OF a-in-V cf-h*]] **by** *simp*
also have ...$= \mathbf{0}$ **by** *simp*
finally show $h\ a \otimes g\ x \oplus h\ x = \mathbf{0}$ **.**
qed
show $(\bigoplus_V y{\in}A.\ h\ y \cdot y) = (\bigoplus_V y{\in}A - \{a\}.\ ?f\ y \cdot y)$
proof $-$
have *linear-combination* h (*insert* a $(A{-}\{a\})) = h\ a \cdot a \oplus_V$ *linear-combination* h $(A{-}\{a\})$
— We want to apply the theorem *linear-combination-insert*, so we have to write insert a (A - a) instead of directly A.
proof (*rule linear-combination-insert*)
show *good-set* $(A - \{a\})$ **using** *cb-Aa* **.**
show $a \in carrier\ V$ **using** *a-in-V* **.**
show $a \notin A - \{a\}$ **using** *a-in-A* **by** *simp*
show $h \in$ *coefficients-function* (*carrier* V) **using** *cf-h* **.**
qed
also have ... $= h\ a \cdot$ *linear-combination* g $(A{-}\{a\}) \oplus_V$ *linear-combination* h $(A{-}\{a\})$
using *a-lc-g-Aa* **by** *auto*
also have ... $=$ *linear-combination* $(\%x.\ h(a) \otimes g\ x)$ $(A{-}\{a\}) \oplus_V$ *linear-combination* h $(A{-}\{a\})$
using *fx-in-K*[*OF a-in-V cf-h*] **and** *linear-combination-rdistrib* [*OF cb-Aa cf-g* -] **by** *auto*
also have ... $= (\bigoplus_V y{\in}A - \{a\}.\ (h\ a \otimes g\ y){\cdot}y \oplus_V (h\ y) \cdot y)$
proof (*unfold linear-combination-def*, *rule finsum-addf*[*symmetric*])
show *finite* $(A - \{a\})$ **using** *cb-Aa* **unfolding** *good-set-def* **by** *auto*
show $(\lambda y.\ (h\ a \otimes g\ y) \cdot y) \in A - \{a\} \rightarrow carrier\ V$
proof (*unfold Pi-def*, *auto*)
fix x
assume *x-in-A*: $x{\in}A$
hence *x-in-V*: $x{\in}carrier\ V$
using *good-set-A* **unfolding** *good-set-def* **by** *auto*
hence $(h\ a \otimes g\ x){\in}\ carrier\ K$
using *cf-g cf-h a-in-V* **unfolding** *coefficients-function-def*
using $K.m\text{-}closed$ **by** *auto*
thus $(h\ a \otimes g\ x) \cdot x \in carrier\ V$
using *mult-closed*[*OF x-in-V* -] **by** *auto*
qed

```
    show (λy. h y · y) ∈ A − {a} → carrier V
      unfolding Pi-def using A-in-V using fx-x-in-V[OF - cf-h] by auto
  qed
  also have ...=linear-combination (%x. (h(a) ⊗ g x) ⊕K (h x)) (A−{a})
  proof (unfold linear-combination-def, rule finsum-cong')
    show A − {a} = A − {a} ..
    show (λy. (h a ⊗ g y ⊕ h y) · y) ∈ A − {a} → carrier V
    proof (unfold Pi-def, auto)
      fix x
      assume x-in-A: x∈A
      hence x-in-V: x∈carrier V
        using good-set-A unfolding good-set-def by auto
      hence (h a ⊗ g x)∈ carrier K
        using cf-g cf-h a-in-V
        unfolding coefficients-function-def using K.m-closed by auto
      hence (h a ⊗ g x) ⊕ h x ∈ carrier K
        using K.a-closed[OF - fx-in-K[OF x-in-V cf-h]] by auto
     thus (h a ⊗ g x ⊕ h x) · x ∈ carrier V using mult-closed[OF x-in-V -] by
simp
    qed
   show ⋀i. i ∈ A − {a} ⟹ (h a ⊗ g i) · i ⊕V h i · i = (h a ⊗ g i ⊕ h i) · i
    proof (rule add-mult-distrib2[symmetric])
      fix x
      assume x-in-A: x∈A−{a}
      thus x-in-V: x∈carrier V using cb-Aa unfolding good-set-def by auto
      thus (h a ⊗ g x)∈ carrier K
     using cf-g cf-h a-in-V unfolding coefficients-function-def using K.m-closed
by auto
      show h x ∈ carrier K using fx-in-K[OF x-in-V cf-h] .
    qed
  qed
 finally show ?thesis unfolding linear-combination-def using igualdad-conjuntos
by auto
 qed
qed
```

Now we present the theorem. The proof is done by double content of both span sets and we make use of the two previous lemmas.

```
theorem
  span-minus:
  assumes good-set-A: good-set A
  and a-in-A: a ∈ A
  and exists-g: ∃ g. g∈ coefficients-function (carrier V)
  ∧ a = linear-combination g (A − {a})
  shows span A = span (A − {a})
proof
  show span (A − {a}) ⊆ span A
    unfolding span-def
    unfolding linear-combination-def
```

    using *assms* **and** *exists-function-Aa-A* **by** *auto*
next
  from *exists-g* **obtain** *g*
    where *cf-g*: $g \in$ *coefficients-function* (*carrier V*)
    and *a-lc*: $a =$ *linear-combination* g $(A-\{a\})$ **by** *auto*
  show *span* $A \subseteq$ *span* $(A - \{a\})$
  proof (*unfold span-def*, *unfold linear-combination-def*,*auto*)
    fix *f*
    assume *cf-f*: $f \in$ *coefficients-function* (*carrier V*)
    show $\exists\, ga \in$ *coefficients-function* (*carrier V*).
      $(\bigoplus{}_V y \in A.\ f\ y \cdot y) = (\bigoplus{}_V y \in A - \{a\}.\ ga\ y \cdot y)$
      using *exists-function-A-Aa*
      [*OF cf-f cf-g a-lc good-set-A a-in-A*] **.**
  qed
qed

A corollary of this theorem claims that for every linearly dependent set A, then $\exists\, a \in A$. *span* $A =$ *span* $(A - \{a\})$.

We also need to use *linear-dependent* $Y \Longrightarrow \exists\, y \in Y.\ \exists\, g.\ g \in$ *coefficients-function* (*carrier V*) $\wedge\ y =$ *linear-combination* g $(Y - \{y\})$

corollary
  span-minus2:
  assumes *ld-A*: *linear-dependent A*
  shows $\exists\, a \in A$. *span* $A =$ *span* $(A - \{a\})$
proof −
  have $\exists\, a \in A.\ \exists\, g.\ g \in$ *coefficients-function* (*carrier V*) $\wedge\ a =$ *linear-combination* g $(A - \{a\})$
    using *exists-x-linear-combination2*[*OF ld-A*] **.**
  thus *?thesis* **using** *span-minus l-dep-good-set*[*OF ld-A*] **by** *auto*
qed

If an element y is not in the span of a set A, hence that element is not in that set. The proof is completed by *reductio ad absurdum.* If $a \in A$, then there is a linear combination of the elements of A, and thus $a \in span(A)$, which is a contradiction with one of the premises.

lemma *not-in-span-impl-not-in-set*:
  assumes *y-notin-span*: $y \notin$ *span A*
  and *cb-A*: *good-set A*
  and *y-in-V*: $y \in$ *carrier V*
  shows $y \notin A$
proof (*cases* $y \notin A$)
  case *True* **thus** *?thesis* **.**
next
  case *False*
  show *?thesis*
  proof −
    def $g \equiv (\%x.$ *if* $x=y$ *then* $\mathbf{1}$ *else* $\mathbf{0})$
    have *cf-g*: $g \in$ *coefficients-function* (*carrier V*)

unfolding *g-def coefficients-function-def* **using** *y-in-V*
by *simp*
have *linear-combination g A* = *y*
proof −
have *igualdad-conjuntos*: $A=(insert\ y\ (A-\{y\}))$
using *False* **by** *fast*
hence *linear-combination g A*
=*linear-combination g* (*insert y* $(A-\{y\})$)
using *arg-cong2* **by** *force*
also have ...$=g(y)\cdot y \oplus_V$ *linear-combination g* $(A-\{y\})$
proof (*rule linear-combination-insert*)
show *good-set* $(A - \{y\})$ **using** *cb-A*
unfolding *good-set-def* **by** *fast*
show $y \in$ *carrier V* **using** *False cb-A*
unfolding *good-set-def* **by** *fast*
show $y \notin A - \{y\}$ **by** *simp*
show $g \in$ *coefficients-function* (*carrier V*) **using** *cf-g* **.**
qed
also have ...$=g(y)\cdot y \oplus_V \mathbf{0}_V$
proof −
have *linear-combination g* $(A-\{y\})=\mathbf{0}_V$
proof −
have $(\bigoplus_V y{\in}A - \{y\}.\ g\ y \cdot y)=(\bigoplus_V y{\in}A - \{y\}.\ \mathbf{0}_V)$
apply (*rule finsum-cong′*) **apply** *auto*
unfolding *g-def* **apply** *simp*
apply (*rule zeroK-mult-V-is-zeroV*)
using *cb-A* **unfolding** *good-set-def* **by** *blast*
also have ...$=\mathbf{0}_V$
using *finsum-zero cb-A*
unfolding *good-set-def* **by** *blast*
finally show *?thesis* **unfolding** *linear-combination-def* **.**
qed
thus *?thesis* **by** *simp*
qed
also have ...$=g(y)\cdot y$
using *r-zero* **and** *mult-closed* **and** *False cb-A*
unfolding *good-set-def g-def* **by** *auto*
also have ...$=y$
using *mult-1 False cb-A*
unfolding *good-set-def g-def* **by** *auto*
finally show *?thesis* **.**
qed
thus *?thesis*
using *cf-g y-notin-span* **unfolding** *span-def* **by** *fast*
qed
qed

If we have an element which is not in the span of an independent set, then the result of inserting this element into that set is a linearly independent set.

The proof is done dividing the goal into cases. The case where $A \neq \{\}$ again is divided in cases with respect to the boolean *linear-independent* (*insert y A*). In the case where *linear-independent* (*insert y A*) is false, again we proceed by *reductio ad absurdum.* It is a long lemma of 129 lines.

lemma *insert-y-notin-span-li*:
  assumes *y-notin-span*: $y \notin span\ A$
  and *y-in-V*: $y \in carrier\ V$
  and *li-A*: *linear-independent A*
  shows *linear-independent* (*insert y A*)
proof (*cases* $A=\{\}$)
  case *True* **thus** *?thesis* — If A is empty it is trivial.
    using *insertI1 span-empty*
      unipuntual-is-li y-in-V y-notin-span **by** *auto*
next
  case *False* **note** *A-not-empty=False*
  show *?thesis*
  proof (*cases linear-independent* (*insert y A*))
    case *True* **thus** *?thesis* **.**
  next
    case *False* **show** *?thesis*
    proof −
      have *y-not-zero*: $y \neq \mathbf{0}_V$
        using *y-notin-span good-set-finite good-set-in-carrier*
          l-ind-good-set li-A span-contains-zero
        by *auto*
      have *cb-A*: *good-set A* **using** *l-ind-good-set li-A* **by** *fast*
      have *finite-A*: *finite A* **using** *good-set-finite l-ind-good-set li-A* **by** *fast*
    have *ld-Ay*: *linear-dependent* $(A \cup \{y\})$ **using** *not-independent-implies-dependent*
False cb-A y-in-V
        unfolding *good-set-def* **by** *auto*
     have *zero-not-in*: $\mathbf{0}_V \notin A \cup \{y\}$ **using** *zero-not-in-linear-independent-set*[*OF li-A*] *y-not-zero* **by** *fast*
      have $\exists h.\ indexing\ (A \cup \{y\},\ h) \wedge h\ `\ \{..<card\ A\} = A \wedge h\ `\ (\{..<card\ A + card\ \{y\}\} - \{..<card\ A\}) = \{y\}$
      proof (*rule indexing-union*)
       show $A \cap \{y\} = \{\}$ **using** *not-in-span-impl-not-in-set*[*OF y-notin-span cb-A y-in-V*] **by** *simp*
        show *finite A* **using** *finite-A* **.**
        show $A \neq \{\}$ **using** *A-not-empty* **.**
        show *finite* $\{y\}$ **by** *simp*
      qed
      from *this* **obtain** *h* **where** *indexing*: *indexing* $(A \cup \{y\},\ h)$ **and** *surj-h-A*: $h\ `\ \{..<card\ A\} = A$
       and *surj-h-y*: $h\ `\ (\{..<card\ A + card\ \{y\}\} - \{..<card\ A\}) = \{y\}$ **by** *fastsimp*

      let $?P = (\lambda k.\ \exists\, b{\in}A{\cup}\{y\}.\ \exists\, g.\ g \in coefficients\text{-}function\ (carrier\ V) \wedge 1 \leq k \wedge$
        $k < card\ (A{\cup}\{y\}) \wedge h\ k = b \wedge b = linear\text{-}combination\ g\ (h\ `\ \{i.\ i < k\}))$

have *exK*: ($\exists$ *k*. *?P k*)
using *linear-dependent-set-sorted-contains-linear-combination* [
OF ld-Ay zero-not-in indexing] **by** *auto*
have *ex-LEAST*: *?P* (*LEAST k*. *?P k*)
using *LeastI-ex* [*OF exK*] **.**
let *?k* = (*LEAST k*. *?P k*)
have $\exists$ *b*$\in$*A*$\cup${*y*}. $\exists$ *g*. *g* $\in$ *coefficients-function* (*carrier V*) $\wedge$ *1* $\leq$ *?k* $\wedge$
?k < *card* (*A*$\cup${*y*}) $\wedge$ *h ?k* = *b* $\wedge$ *b* = *linear-combination g* (*h* ' {*i*. *i* < *?k*})
using *ex-LEAST* **by** *simp*
then obtain *b g*
where *one-le-k*: *1* $\leq$ *?k* **and** *k-l-card*: *?k* < *card* (*A*$\cup${*y*}) **and** *h-k-eq-b*: *h*
?k = *b*
and *cf-g*: *g* $\in$ *coefficients-function* (*carrier V*) **and**
combinacion-anteriores: *b* = *linear-combination g* (*h* ' {*i*. *i* < *?k*})
and *b-in-Ay*: *b* $\in$ (*A* $\cup$ {*y*})
by *blast*
show *?thesis*
proof (*cases b*$\in${*y*})
case *True* **note** *b-in-y*=*True*
have *k-eq-card*: *?k*=*card A*
proof −
— I will prove that k is less or equal to card A. If k¡card A we will obtain a contradiction (because the element will be in A). So k = card A
have *card* (*A*$\cup${*y*})= *card A* + *1*
using *not-in-span-impl-not-in-set*[*OF y-notin-span cb-A y-in-V*] *finite-A card-insert-if* **by** *auto*
hence *k-le-cardA*: *?k* $\leq$ *card A* **using** *k-l-card* **by** *auto*
thus *?thesis*
proof (*cases ?k*<*card A*)
case *True*
have *h ?k* $\in$ *A* **using** *surj-h-A True* **by** *auto*
thus *?thesis* **using** *not-in-span-impl-not-in-set*[*OF y-notin-span cb-A y-in-V*] *h-k-eq-b b-in-y* **by** *auto*
next
case *False* **thus** *?thesis* **using** *k-le-cardA* **by** *auto*
qed
qed
have *linear-combination g A* = *y*
proof −
have *h* ' {*i*. *i* < *?k*}=*A* **using** *surj-h-A k-eq-card* **by** *auto*
hence *linear-combination g A* = *linear-combination g* (*h* ' {*i*. *i* < *?k*})
using *arg-cong2*[*of g g A h*'{..<*card A*}] **by** *presburger*
also have ...=*b* **using** *combinacion-anteriores* **by** *simp*
also have ...=*y* **using** *True* **by** *simp*
finally show *?thesis* **.**
qed
thus *?thesis* **using** *cf-g y-notin-span* **unfolding** *span-def* **by** *auto*
next
case *False*

show *?thesis*
proof –
have *b-in-A*: $b \in A$ **using** *False b-in-Ay* **by** *simp*
have *k-le-cardA*: $?k < card(A)$
using *b-in-A* **and** *h-k-eq-b* **and** *surj-h-A* **and** *k-l-card* **and** *indexing*
unfolding *indexing-def* **and** *bij-betw-def* **and** *inj-on-def*
by *force*
have *ld-insert*: *linear-dependent* (*insert b* (h‘$\{i.\ i<?k\}$))
proof (*rule lc1*)
show *linear-independent* (h‘$\{i.\ i<?k\}$)
proof (*rule independent-set-implies-independent-subset*)
show *linear-independent A* **using** *li-A* **.**
show h ‘ $\{i.\ i < ?k\} \subseteq A$ **using** *surj-h-A k-le-cardA* **by** *auto*
qed
show $b \in$ *carrier V* **using** *b-in-A cb-A* **unfolding** *good-set-def*
by *auto*
show $b \notin h$‘$\{i.\ i<?k\}$
using *b-in-A* **and** *h-k-eq-b* **and** *surj-h-A* **and** *k-l-card* **and** *indexing*
unfolding *indexing-def* **and** *bij-betw-def* **and** *inj-on-def*
by *force*
show $\exists f.\ f \in$ *coefficients-function* (*carrier V*) $\wedge$
linear-combination f (h ‘$\{i.\ i<?k\}$)$=b$
using *cf-g* **and** *combinacion-anteriores* **by** *auto*
qed
have *linear-dependent* (h‘$\{..<card(A)\}$)
proof (*rule linear-dependent-subset-implies-linear-dependent-set*)
show *insert b* (h‘$\{i.\ i<?k\}$)$\subseteq h$ ‘ $\{..<card\ A\}$
proof –
have *igualdad-conjuntos*: $\{i.\ i<?k\} \cup \{?k\}=\{..?k\}$ **using** *atMost-def*[*of ?k*] *ivl-disj-un*(*2*) **by** *auto*
have *insert b* (h‘$\{i.\ i<?k\}$)$=$(h‘$\{i.\ i<?k\}$) $\cup \{b\}$ **by** *simp*
also have ...$=h$‘$\{i.\ i<?k\} \cup h$‘ $\{?k\}$ **using** *h-k-eq-b* **by** *auto*
also have ...$=h$‘ ($\{i.\ i<?k\} \cup \{?k\}$) **by** *auto*
also have ...$=h$‘$\{..?k\}$ **using** *igualdad-conjuntos* **by** *auto*
also have ... $\subseteq h$ ‘ $\{..<card\ A\}$ **using** *k-le-cardA* **by** *auto*
finally show *?thesis* **.**
qed
show *good-set* (h ‘ $\{..<card\ A\}$)
using *surj-h-A cb-A* **by** *auto*
show *linear-dependent* (*insert b* (h‘$\{i.\ i<?k\}$)) **using** *ld-insert* **.**
qed
— Contradiction: we have linear dependent A and linear independent A
thus *?thesis* **using** *surj-h-A li-A cb-A independent-implies-not-dependent* **by** *auto*
qed
qed
qed
qed

qed

We can unify the concepts of *spanning-set*, *span* and *basis* and illustrate the relationships that exist among them.

The *span* of a *spanning-set* is *carrier V*.

lemma *span-basis-implies-spanning-set*:
  assumes *span-A-V*: *span A = carrier V*
  and *good-set-A*: *good-set A*
  shows *spanning-set A*
  unfolding *spanning-set-def*
  using *span-A-V good-set-A*
  unfolding *span-def good-set-def* **by** *force*

The opposite implication:

lemma *spanning-set-implies-span-basis*:
  assumes *sg-A*: *spanning-set A*
  shows *span A = carrier V*
  using *sg-A* **and** *linear-combination-closed*
  unfolding *spanning-set-def* **and** *span-def*
  by *fast*

Now we present the relationship between *spanning-set* and *span*: if *span A = carrier V* then *A* is a *spanning set*.

lemma *span-V-eq-spanning-set*:
  assumes *cb-A*: *good-set A*
  shows *span A = carrier V* $\longleftrightarrow$ *spanning-set A*
  using *span-basis-implies-spanning-set*
    and *spanning-set-implies-span-basis*
    and *cb-A* **by** *auto*

Now we can introduce in Isabelle a new definition of basis (in the case of finite dimensional vector spaces). A finite basis will be a set *A* which is *linear-independent* and satisfies *span A = carrier V*. We use the previous lemma to check that it is equivalent to *basis X* = (*X* $\subseteq$ *carrier V* $\wedge$ *linear-independent-ext X* $\wedge$ *spanning-set-ext X*).

lemma *basis-def'*:
  assumes *cb-A*: *good-set A*
  shows *basis A* $\longleftrightarrow$ (*linear-independent A* $\wedge$ *span A = carrier V*)
  using *assms basis-def fin-ind-ext-impl-ind good-set-def*
    good-set-in-carrier gs-spanning-ext-imp-spanning
    independent-imp-independent-ext
    span-V-eq-spanning-set spanning-imp-spanning-ext
    spanning-set-implies-span-basis **by** *auto*

If we have a finite basis, we can forget extended versions of linear independence and spanning set:

lemma *finite-basis*:

assumes *fin-A*: *finite A*
shows *basis A* $\longleftrightarrow$ (*linear-independent A* $\wedge$ *spanning-set A*)
using *assms basis-def basis-def' fin-ind-ext-impl-ind*
l-ind-good-set span-V-eq-spanning-set spanning-set-implies-span-basis
by *metis*

end

9.1 Finite Dimensional Vector Space

For working in a finite vector space we need to fix a finite basis.

The definition of finite dimensional vector spaces in Isabelle/HOL is direct. It consists of a vector space in which we assume that there exists a finite basis. Note that we have not proved yet that every vector space contains a basis.

locale *finite-dimensional-vector-space* = *vector-space* +
fixes *X* :: *'c set*
assumes *finite-X*: *finite X*
and *basis-X*: *basis X*

context *finite-dimensional-vector-space*
begin

From this point the fixed basis is denoted by X.

We add to simplifier both premisses.

lemmas [*simp*] = *finite-X basis-X*

It is easy to show that the basis is a good set, is linearly independent and a spanning set.

lemma *good-set-X*:
shows *good-set X*
using *basis-X*
unfolding *basis-def*
using *finite-X*
unfolding *good-set-def* **by** *simp*

lemma *linear-independent-X*:
shows *linear-independent X*
using *basis-X*
unfolding *basis-def*
unfolding *linear-independent-ext-def*
using *finite-X* **by** *simp*

lemma *spanning-set-X*:
shows *spanning-set X*

using *basis-X good-set-X*
unfolding *basis-def*
using *gs-spanning-ext-imp-spanning* **by** *fast*

We add to simplifier these three lemmas.

lemmas [*simp*] = *good-set-X linear-independent-X spanning-set-X*

For all $x \in$ *carrier V* exists a linear combination of elements of the basis (we can write $x \in$ *carrier V* in combination of the elements of a basis).

lemma *exists-combination*:
assumes *x-in-V*: $x \in$ *carrier V*
shows $\exists f.$ ($f \in$ *coefficients-function* (*carrier V*) $\wedge$ $x =$ *linear-combination f X*)
using *x-in-V spanning-set-X*
unfolding *spanning-set-def*
by *fast*

Next lemma shows us that coordinates of a vector are unique for each basis

lemma *unique-coordenates*:
assumes *x-in-V*: $x \in$ *carrier V*
and *cf-f*: $f \in$ *coefficients-function* (*carrier V*)
and *lc-f*: $x =$ *linear-combination f X*
and *cf-g*: $g \in$ *coefficients-function* (*carrier V*)
and *lc-g*: $x =$ *linear-combination g X*
shows $\forall x \in X.\ g\ x = f\ x$
proof −
have *linear-combination f X* $\oplus_V \ominus_V$ *linear-combination g X*
$= x \oplus_V \ominus_V x$
using *lc-f* **and** *lc-g* **by** *auto*
hence $\mathbf{0}_V =$ *linear-combination f X*
$\oplus_V$ (($\ominus_K \mathbf{1}_K$)$\cdot$ *linear-combination g X*)
using *V.r-neg* [*OF x-in-V*]
negate-eq[*OF linear-combination-closed*[*OF good-set-X cf-g*]]
by *auto*
also have . . .=*linear-combination f X*
$\oplus_V$ *linear-combination* (%*i*. ($\ominus_K \mathbf{1}_K$)$\otimes$ $g(i)$) *X*
using *linear-combination-rdistrib*[*OF*
good-set-X cf-g K.a-inv-closed[*OF K.one-closed*]] **by** *auto*
also have . . .=*linear-combination* (%*x*. $f(x) \oplus_K \ominus_K g(x)$) *X*
unfolding *linear-combination-def*
proof −
have ($\bigoplus_V y{\in}X.\ f\ y \cdot y$) $\oplus_V$ ($\bigoplus_V y{\in}X.\ (\ominus\ \mathbf{1} \otimes g\ y) \cdot y$) $=$
($\bigoplus_V y{\in}X.\ (f\ y \cdot y) \oplus_V (\ominus\ \mathbf{1} \otimes g\ y) \cdot y$)
proof (*rule finsum-addf*[*symmetric*])
show *finite X* **using** *finite-X* **.**
show ($\lambda y.\ f\ y \cdot y$) $\in X \rightarrow$ *carrier V*
using *mult-closed* **and** *cf-f* **and** *good-set-X*
unfolding *good-set-def* **and** *coefficients-function-def* **and** *Pi-def* **by** *auto*
show ($\lambda y.\ (\ominus\ \mathbf{1} \otimes g\ y) \cdot y$) $\in X \rightarrow$ *carrier V*
proof (*unfold Pi-def*, *auto*, *rule mult-closed*)

fix y
assume *y-in-X*: $y \in X$
hence $y \in carrier\ V$ **using** *good-set-X* **unfolding** *good-set-def* **by** *auto*
thus $y \in carrier\ V$ **.**
thus $\ominus\ \mathbf{1} \otimes g\ y \in carrier\ K$
using *cf-g* **and** *y-in-X* **unfolding** *coefficients-function-def*
using *K.m-closed*[*OF K.a-inv-closed*[*OF K.one-closed*] -] **by** *auto*
qed
qed
also have $\ldots = linear\text{-}combination\ (\%x.\ f(x) \oplus_K ((\ominus_K \mathbf{1}_K) \otimes g(x)))\ X$
proof (*unfold linear-combination-def*, *rule finsum-cong′*)
show $X = X$ **..**
show $(\lambda y.\ (f\ y \oplus \ominus\ \mathbf{1} \otimes g\ y) \cdot y) \in X \to carrier\ V$
proof (*unfold Pi-def*, *auto*, *rule mult-closed*)
fix y
assume *y-in-X*: $y \in X$
thus *y-in-V*: $y \in carrier\ V$ **using** *good-set-X* **unfolding** *good-set-def* **by** *auto*
show $f\ y \oplus \ominus\ \mathbf{1} \otimes g\ y \in carrier\ K$
using *fx-in-K*[*OF y-in-V cf-f*]
using *fx-in-K*[*OF y-in-V cf-g*]
using *K.m-closed*[*OF K.a-inv-closed*[*OF K.one-closed*] -] **and** *K.a-closed* **by** *blast*
qed
show $\bigwedge i.\ i \in X \Longrightarrow f\ i \cdot i \oplus_V (\ominus\ \mathbf{1} \otimes g\ i) \cdot i = (f\ i \oplus \ominus\ \mathbf{1} \otimes g\ i) \cdot i$
proof −
fix y
assume *y-in-X*: $y \in X$
hence *y-in-V*: $y \in carrier\ V$ **using** *good-set-X* **unfolding** *good-set-def*
by *auto*
thus $f\ y \cdot y \oplus_V (\ominus\ \mathbf{1} \otimes g\ y) \cdot y = (f\ y \oplus \ominus\ \mathbf{1} \otimes g\ y) \cdot y$
proof (*rule add-mult-distrib2*[*symmetric*])
show $f\ y \in carrier\ K$ **using** *cf-f* **and** *y-in-V*
unfolding *coefficients-function-def* **by** *auto*
show $\ominus\ \mathbf{1} \otimes g\ y \in carrier\ K$
using *cf-g* **and** *y-in-V* **unfolding** *coefficients-function-def*
using *K.m-closed*[*OF K.a-inv-closed*[*OF K.one-closed*] -] **by** *auto*
qed
qed
qed
also have $\ldots = linear\text{-}combination\ (\%x.\ f(x) \oplus_K \ominus_K g(x))\ X$
proof (*unfold linear-combination-def*, *rule finsum-cong′,auto*)
fix y
assume *y-in-X*: $y \in X$
hence *y-in-V*: $y \in carrier\ V$ **using** *good-set-X* **unfolding** *good-set-def* **by** *auto*
show $(f\ y \oplus_K \ominus_K g\ y) \cdot y \in carrier\ V$
proof (*rule mult-closed*)
show $y \in carrier\ V$ **using** *y-in-V* **.**

show $f\ y \oplus_K \ominus_K g\ y \in$ *carrier K*
using *fx-in-K* [*OF y-in-V cf-f*]
using *fx-in-K* [*OF y-in-V cf-g*]
unfolding *coefficients-function-def* **using** *K.a-inv-closed* **using** *K.a-closed* **by** *auto*
qed
have *fy-in-K*: $f(y) \in$ *carrier K*
using *cf-f* **and** *y-in-V* **unfolding** *coefficients-function-def* **by** *auto*
have *gy-in-K*: $g(y) \in$ *carrier K*
using *cf-g* **and** *y-in-V* **unfolding** *coefficients-function-def* **by** *auto*
show $(f\ y \oplus \ominus \mathbf{1} \otimes g\ y) \cdot y = (f\ y \oplus_K \ominus_K g\ y) \cdot y$
proof −
have $(f\ y \oplus \ominus \mathbf{1} \otimes g\ y) = (f\ y \oplus_K \ominus_K g\ y)$ **using** *K.l-minus-one*[*OF gy-in-K*] **by** *auto*
thus *?thesis* **by** *auto*
qed
qed
finally show $(\bigoplus_V y{\in}X.\ f\ y \cdot y) \oplus_V (\bigoplus_V y{\in}X.\ (\ominus \mathbf{1} \otimes g\ y) \cdot y) = (\bigoplus_V y{\in}X.\ (f\ y \oplus \ominus g\ y) \cdot y)$
unfolding *linear-combination-def* **by** *auto*
qed
finally have
lc-fg: $\mathbf{0}_V$=*linear-combination* $(\%x.\ f(x) \oplus_K \ominus_K g(x))\ X$
by *simp*
have *cf-fg*: $(\%x.\ (f(x) \oplus_K \ominus_K g(x)))$
$\in$ *coefficients-function* (*carrier V*)
proof (*unfold coefficients-function-def*, *auto*)
fix x
assume *x-in-V*: $x \in$ *carrier V*
show $f\ x \oplus \ominus g\ x \in$ *carrier K*
using *fx-in-K*[*OF x-in-V cf-f*] *fx-in-K*[*OF x-in-V cf-g*] **by** *fast*
next
fix x
assume *x-notin-V*: $x \notin$ *carrier V*
show $f\ x \oplus \ominus g\ x = \mathbf{0}$ **using** *cf-f cf-g* **unfolding** *coefficients-function-def* **using** *x-notin-V* **by** *simp*
qed
hence *fg-0*: $\forall x{\in}X.\ f(x) \oplus_K \ominus_K g(x)=\mathbf{0}_K$
using *linear-independent-X* **and** *lc-fg*[*symmetric*]
unfolding *linear-independent-def* **by** *auto*
show $\forall x \in X.\ g(x)=f(x)$
proof
fix y
assume *y-in-X*: $y{\in}X$
hence *y-in-V*: $y\in$*carrier V*
using *good-set-X* **unfolding** *good-set-def*
by *auto*
have *fg-y0*: $f\ y \oplus \ominus g\ y = \mathbf{0}$
using *y-in-X* **and** *fg-0* **by** *auto*

have *fy-in-K*: $f(y)\in$ *carrier K*
  using *cf-f* **and** *y-in-V*
  unfolding *coefficients-function-def* **by** *auto*
have *gy-in-K*: $g(y)\in$ *carrier K*
  using *cf-g* **and** *y-in-V*
  unfolding *coefficients-function-def* **by** *auto*
hence $\ominus_K(\ominus_K\ g\ y)=f\ y$
  using *K.minus-equality*
  [*OF fg-y0 K.a-inv-closed*[*OF gy-in-K*] *fy-in-K*]
  by *auto*
thus $g(y)=f(y)$ **using** *K.minus-minus*[*OF gy-in-K*] **by** *auto*
qed
qed

We have fixed a finite basis and now we can prove some theorems about the *span*. Note that the concept of finitude of the basis is very important in the proofs.

The span of a basis is the total, so it's easy to prove that *carrier V* $\subseteq$ *span X*. The other implication is also easy: we have only to unfold the definition and use $[\![$*good-set ?X*; *?f* $\in$ *coefficients-function* (*carrier V*)$]\!] \Longrightarrow$ *linear-combination ?f ?X* $\in$ *carrier V*.

lemma *span-basis-is-V*: *span X* = *carrier V*
proof
show *span X* $\subseteq$ *carrier V*
  unfolding *span-def*
  using *linear-combination-closed* **by** *auto*
show *carrier V* $\subseteq$ *span X*
  unfolding *span-def*
  using *spanning-set-X* **unfolding** *spanning-set-def* **by** *auto*
qed

The span of every set joined with a basis is the total. Before proving this theorem, we make two auxiliar lemmas.

First one:

lemma *exists-linear-combination-union-basis*:
assumes *fin-A*: *finite A*
and *A-in-V*: *A* $\subseteq$ *carrier V*
and *x-in-V*: *x* $\in$ *carrier V*
shows $\exists$ *g. g* $\in$ *coefficients-function* (*carrier V*) $\wedge$ *x* = *linear-combination g* (*A* $\cup$ *X*)
proof −
from *spanning-set-X* **obtain** *f* **where** *cf-f*: *f* $\in$ *coefficients-function* (*carrier V*)

  and *x-lc-fX*: *x=linear-combination f X*
  unfolding *spanning-set-def* **using** *x-in-V* **by** *auto*
let *?g*=(%*x. if x* $\in$ *X then f*(*x*) *else* $\mathbf{0}_K$)
have *cf-g*: *?g*$\in$*coefficients-function* (*carrier V*)

using *coefficients-function-g-f-null*[*OF cf-f*] **.**
have *good-set-A*: *good-set A*
using *fin-A A-in-V* **unfolding** *good-set-def* **by** *auto*
have *linear-combination ?g* $(A \cup X)$=*linear-combination ?g* $(X \cup A)$
— It is easier apply $[\![?a = ?b; ?c = ?d]\!] \Longrightarrow ?f\ ?a\ ?c = ?f\ ?b\ ?d$ than $[\![?A = ?B; ?g \in ?B \rightarrow carrier\ V; \bigwedge i.\ i \in ?B \Longrightarrow ?f\ i = ?g\ i]\!] \Longrightarrow finsum\ V\ ?f\ ?A = finsum\ V\ ?g\ ?B$, the unique different is in the arguments of the function.
by (*rule arg-cong2* [*of ?g ?g - - linear-combination*], *auto*)
also have ...=*linear-combination f X*
using *eq-lc-when-out-of-set-is-zero*[*OF good-set-A good-set-X cf-f*] **.**
also have ... = x **using** *x-lc-fX* [*symmetric*] **.**
finally have *x-lc-g-AX*: x = *linear-combination ?g* $(A \cup X)$ **by** (*rule sym*)
hence *?g* $\in$ *coefficients-function* (*carrier V*) $\wedge$ x = *linear-combination ?g* $(A \cup X)$
using *cf-g* **by** *auto*
thus *?thesis* **by** (*rule exI*[*of - ?g*])
qed

Second one

lemma *span-union-basis-eq*:
assumes *fin-A*: *finite A*
and *A-in-V*: $A \subseteq$ *carrier V*
shows *span* $(A \cup X)$ = *span X*
using *span-basis-is-V*
proof (*unfold span-def,auto*)
fix x
assume *x-in-V*: $x \in$*carrier V*
show $\exists g \in$*coefficients-function* (*carrier V*). x = *linear-combination g* $(A \cup X)$
using *exists-linear-combination-union-basis*[*OF fin-A A-in-V x-in-V*] **by** *auto*
next
fix g
assume *cf-g*: $g \in$ *coefficients-function* (*carrier V*)
show *linear-combination g* $(A \cup X) \in$ *carrier V*
proof (*rule linear-combination-closed*)
show *good-set* $(A \cup X)$
using *good-set-X* **and** *fin-A* **and** *A-in-V*
unfolding *good-set-def* **by** *auto*
show $g \in$ *coefficients-function* (*carrier V*) **using** *cf-g* **.**
qed
qed

Finally the theorem: the span of every set joined with a basis is the total

corollary *span-union-basis-is-V*:
assumes *fin-A*: *finite A*
and *A-in-V*: $A \subseteq$ *carrier V*
shows *span* $(A \cup X)$ = *carrier V*
using *assms span-union-basis-eq* **and** *span-basis-is-V*
by *auto*

9.2 Theorem 1.

From this, we are going to center into the proof that every linearly independent set can be extended to a basis.

The function *remove-ld* takes an element of type *'a iset* and returns other element of that type in which in the set has been removed the first element that is a combination of the preceding ones, and the indexation has removed the corresponding index.

In the next definition, making use of previous theorem:

$[\![$*linear-dependent Xa*; $\mathbf{0}_V \notin$ *Xa*$]\!] \Longrightarrow \exists\, y{\in}Xa.\ \exists\, g\ k.\ \exists f{\in}\{i.\ i < card\ Xa\} \rightarrow Xa.\ f\ \text{`}\ \{i.\ i < card\ Xa\} = Xa \wedge g \in coefficients\text{-}function\ (carrier\ V) \wedge 1 \leq k \wedge k < card\ Xa \wedge f\,k = y \wedge y = linear\text{-}combination\ g\ (f\ \text{`}\ \{i.\ i < k\})$, we remove the *least* element that verifies the property that it can be expressed as a linear combination of the preceding ones. The existence of this element is guaranteed by the fact that the set is linearly dependent. If we iterate the function *remove-ld* we can be sure that it will terminate because sooner or later we will achieve a linearly independent set.

It is important to note that we have to provide a fixed indexation *f* for the elements to be removed are uniquely determined.

The function *remove-ld* must be only applied to an indexation of a linearly dependent set that does not contain $\mathbf{0}_V$, since these are the uniques conditions where we have ensured the existence of the element to be removed using:

linear-dependent-set-contains-linear-combination: $[\![$*linear-dependent Xa*; $\mathbf{0}_V \notin Xa]\!] \Longrightarrow \exists\, y{\in}Xa.\ \exists\, g\ k.\ \exists f{\in}\{i.\ i < card\ Xa\} \rightarrow Xa.\ f\ \text{`}\ \{i.\ i < card\ Xa\} = Xa \wedge g \in coefficients\text{-}function\ (carrier\ V) \wedge 1 \leq k \wedge k < card\ Xa \wedge f\ k = y \wedge y = linear\text{-}combination\ g\ (f\ \text{`}\ \{i.\ i < k\})$.

definition *remove-ld* :: *'c iset* => *'c iset*
  where *remove-ld A* =
  (let n = (LEAST k::nat. ∃ y∈(fst A). ∃ g.
  g ∈ coefficients-function (carrier V)
  ∧ (1::nat) ≤ k ∧ k < (card (fst A))
  ∧ (snd A) k = y
  ∧ y = linear-combination g ((snd A) ` {i::nat. i<k}))
  in remove-iset A n)

Next lemma expresses another notation for $remove\text{-}ld\ ?A = Let\ (LEAST\ k.\ \exists\, y{\in}fst\ ?A.\ \exists\, g.\ g \in coefficients\text{-}function\ (carrier\ V) \wedge 1 \leq k \wedge k < card\ (fst\ ?A) \wedge snd\ ?A\ k = y \wedge y = linear\text{-}combination\ g\ (snd\ ?A\ \text{`}\ \{i.\ i < k\}))\ (remove\text{-}iset\ ?A)$.

lemma *remove-ld-def'*:

remove-ld $(A, f) = ($*let* $n = ($*LEAST* k::*nat*. $\exists\, y \in A.\ \exists\, g.$
  $g \in$ *coefficients-function* $($*carrier* $V) \land (1$::*nat*$) \leq k$
  $\land\ k < ($*card* $A) \land f\, k = y \land y =$ *linear-combination* g $(f`\{i$::*nat*. $i<k\}))$
in $(A - \{f\, n\}, (\lambda k.$ *if* $k < n$ *then* $f\, k$ *else* f $($*Suc* $k))))$
unfolding *remove-ld-def*
unfolding *Let-def*
unfolding *remove-iset-def'* **by** *simp*

Now we can prove some properties of the function *remove-ld*: it preserves the carrier, is monotone and decrease the cardinality.

lemma *remove-ld-preserves-carrier*:
  assumes b: $B \subseteq$ *carrier* V
  shows *fst* $($*remove-ld* $(B, h)) \subseteq$ *carrier* V
  using b
  unfolding *remove-ld-def'*
  unfolding *Let-def* **by** *auto*

lemma *remove-ld-monotone*:
  assumes b: $B \subseteq$ *carrier* V
  shows *fst* $($*remove-ld* $(B, h)) \subseteq B$
  using b
  unfolding *remove-ld-def'*
  unfolding *Let-def* **by** *auto*

lemma *remove-ld-decr-card*:
  assumes *ld-A*: *linear-dependent* A
  and *not-zero*: $\mathbf{0}_V \notin A$
  and *indexing-A-f*: *indexing* (A, f)
  shows *card* $($*fst* $($*remove-ld* $(A, f))) =$ *card* $A - 1$
proof $-$
  let $?P = (\lambda k.\ \exists\, y \in A.\ \exists\, g.\ g \in$ *coefficients-function* $($*carrier* $V) \land 1 \leq k \land$
    $k <$ *card* $A \land f\, k = y \land y =$ *linear-combination* g $(f\ `\ \{i.\ i < k\}))$
  have *fin-A*: *finite* A
    using *l-dep-good-set* [*OF ld-A*]
    unfolding *good-set-def* **by** *fast*
  have *exK*: $(\exists\, k.\ ?P\ k)$
    using *linear-dependent-set-sorted-contains-linear-combination* [
      OF ld-A not-zero indexing-A-f] **by** *auto*
  have *ex-LEAST*: $?P$ $($*LEAST* $k.\ ?P\ k)$
    using *LeastI-ex* [*OF exK*] **.**
  let $?k = ($*LEAST* $k.\ ?P\ k)$
  have $\exists\, y \in A.\ \exists\, g.\ g \in$ *coefficients-function* $($*carrier* $V) \land 1 \leq ?k \land$
    $?k <$ *card* $A \land f\ ?k = y \land y =$ *linear-combination* g $(f\ `\ \{i.\ i < ?k\})$
    using *ex-LEAST* **by** *simp*
  then obtain y
    where *one-le-k*: $1 \leq ?k$ **and** *k-l-card*: $?k <$ *card* A **and** *f-k-eq-y*: $f\ ?k = y$
    by *auto*
  then have *rem-eq*: *fst* $($*remove-ld* $(A, f)) = (A - \{y\})$ **and** *y-in-A*: $y \in A$
    using *indexing-equiv-img* [*OF indexing-A-f*]

unfolding *Pi-def* **unfolding** *remove-ld-def'* **by** *auto*
show *?thesis*
unfolding *rem-eq*
using *card-Diff-singleton fin-A y-in-A* **by** *auto*
qed

corollary *remove-ld-decr-card2*:
assumes *ld-A*: *linear-dependent A*
and *not-zero*: $\mathbf{0}_V \notin A$
and *indexing-A-f*: *indexing* (A, f)
shows *card* (*fst* (*remove-ld* (A, f))) $<$ *card A*
proof $-$
have *card-A-g-zero*: *card A* > 0
proof $-$
have *not-empty*: $A \neq \{\}$ **using** *dependent-not-empty* [*OF ld-A*] **.**
have *finite-A*: *finite A* **using** *l-dep-good-set*[*OF ld-A*] **unfolding** *good-set-def* **by** *simp*
show *?thesis* **using** *card-gt-0-iff* [*of A*] **and** *not-empty* **and** *finite-A* **by** *auto*
qed
have *card* (*fst* (*remove-ld* (A, f))) $=$ *card A* $-$ *1*
using *remove-ld-decr-card*[*OF ld-A not-zero indexing-A-f*] **.**
also have ...$<$*card A* **using** *card-A-g-zero* **by** *auto*
finally show *?thesis* **.**
qed

This is an indispensable result: our function *remove-ld* preserves the propiety of *span*. For this proof is very important the theorem *span-minus*: $[\![$*good-set ?A*; *?a* $\in$ *?A*; $\exists\, g.\ g \in$ *coefficients-function* (*carrier V*) $\wedge$ *?a* $=$ *linear-combination g* (*?A* $-$ $\{$*?a*$\}$)$]\!] \Longrightarrow$ *span ?A* $=$ *span* (*?A* $-$ $\{$*?a*$\}$).

lemma *remove-ld-preserves-span*:
assumes *ld-A*: *linear-dependent A*
and *not-zero*: $\mathbf{0}_V \notin A$
and *indexing-A-f*: *indexing* (A, f)
shows *span* (*fst* (*remove-ld* (A, f))) $=$ *span A*
proof $-$
let *?P* $= (\lambda k.\ \exists\, y{\in}A.\ \exists\, g.\ g \in$ *coefficients-function* (*carrier V*) $\wedge$ *1* $\leq k \wedge$
$k <$ *card A* $\wedge f\, k = y \wedge y =$ *linear-combination g* $(f$ ‘ $\{i.\ i < k\}))$
have *fin-A*: *finite A*
using *l-dep-good-set* [*OF ld-A*]
unfolding *good-set-def* **by** *fast*
have *exK*: ($\exists\, k.$ *?P k*)
using *linear-dependent-set-sorted-contains-linear-combination* [
OF ld-A not-zero indexing-A-f] **by** *auto*
have *ex-LEAST*: *?P* (*LEAST k. ?P k*)
using *LeastI-ex* [*OF exK*] **.**
let *?k* $=$ (*LEAST k. ?P k*)
def $k ==$ *?k* — I introduce a new constant named k to make some goals more legible. When we want to unfold it we will have to use *k-def*.

have $\exists\, y \in A.\ \exists\, g.\ g \in$ *coefficients-function* (*carrier* V) $\wedge$ *1* $\leq$ *?k* $\wedge$
?k $<$ *card* $A \wedge f$ *?k* $= y \wedge y =$ *linear-combination* g (f ‘ $\{i.\ i <$ *?k*$\}$)
using *ex-LEAST* **by** *simp*
then
obtain $y\ g$
where *one-le-k*: *1* $\leq k$ **and** *k-l-card*: $k <$ *card* A **and** *f-k-eq-y*: $f\ k = y$
and *cf-g*: $g \in$ *coefficients-function* (*carrier* V)
and *y-lc-gf*: $y =$ *linear-combination* g (f ‘ $\{i.\ i < k\}$) **and** *y-in-A*: $y \in A$
unfolding *k-def* **by** *auto*
have *rem-eq*: *fst* (*remove-ld* $(A, f)) = (A - \{y\})$ **and** *y-in-A*: $y \in A$
using *indexing-equiv-img* [*OF indexing-A-f*] **and** *one-le-k* **and** *k-l-card* **and**
f-k-eq-y
unfolding *Pi-def* **unfolding** *k-def remove-ld-def′* **by** *auto*
— I have to prove that this y is a linear combination of $A - \{y\}$.
have *contenido*: f ‘ $\{i.\ i < k\} \subseteq A - \{y\}$
proof −
have *bb*: *bij-betw* f $\{i.\ i \leq k\}$ (f ‘ $\{i.\ i \leq k\}$)
proof (*rule bij-betw-subset* [*of f* $\{..<$*card* $A\}$ A $\{i.\ i \leq k\}$])
show *bij-betw* f $\{..<$*card* $A\}$ A
using *indexing-A-f* **unfolding** *indexing-def* **by** *simp*
show $\{i.\ i \leq k\} \subseteq \{..<$*card* $A\}$
using *k-l-card* **unfolding** *k-def* **by** *auto*
qed
have f ‘ $(\{i.\ i < k\}) = f$ ‘ $(\{i.\ i \leq k\} - \{k\})$ **by** *auto*
also have f ‘ $(\{i.\ i \leq k\} - \{k\}) = f$ ‘ $\{i.\ i \leq k\} - \{f\ k\}$
by (*rule bij-betw-image-minus*, *rule bb*, *simp*)
finally have f ‘ $(\{i.\ i < k\}) = f$ ‘ $\{i.\ i \leq k\} - \{f\ k\}$ **by** *fast*
thus *?thesis*
using *indexing-equiv-img* [*OF indexing-A-f*]
unfolding *f-k-eq-y*
using *k-l-card* **by** *auto*
qed
hence *union*: (f‘$\{i.\ i < k\} \cup (A - \{y\})) = A - \{y\}$ **by** *auto*
have *good-set-A*: *good-set* A
using *l-dep-good-set*[*OF ld-A*] **.**
hence *good-set-Ay*: *good-set* $(A - \{y\})$
using *y-in-A* **unfolding** *good-set-def* **by** *auto*
hence *good-set-f*: *good-set* (f‘$\{i.\ i < k\}$)
using *contenido* **unfolding** *good-set-def* **by** *auto*
let *?h*=(*%y. if* $y \in f$ ‘ $\{i.\ i < k\}$ *then* $g\ y$ *else* $\mathbf{0}_K$)
have *cf-h*: *?h* $\in$ *coefficients-function* (*carrier* V)
using *coefficients-function-g-f-null*[*OF cf-g*] **.**
have *linear-combination* g (f ‘ $\{i.\ i < k\}$) $=$
linear-combination ?h (f‘$\{i.\ i < k\} \cup (A - \{y\})$)
using *eq-lc-when-out-of-set-is-zero*[*OF good-set-Ay good-set-f cf-g*, *symmetric*]
by *fast*
also have ...=*linear-combination ?h* $(A - \{y\})$
using *arg-cong2* [*of ?h ?h* (f‘$\{i.\ i <$ *?k*$\} \cup (A - \{y\})$) $(A - \{y\})$ *linear-combination*]

using *union* **by** *presburger*
finally have $y = \textit{linear-combination ?h } (A - \{y\})$ **using** *y-lc-gf* **by** *fastsimp*
hence *exists-h*: $\exists h.\ h \in \textit{coefficients-function } (\textit{carrier } V) \wedge y = \textit{linear-combination } h\ (A - \{y\})$
using *cf-h* **by** *fast*
have $\textit{span } A = \textit{span } (A - \{y\})$
using *span-minus*[*OF good-set-A y-in-A exists-h*] **.**
also have ... $= \textit{span } (\textit{fst } (\textit{remove-ld } (A, f)))$ **using** *rem-eq* **by** *simp*
finally show *?thesis* **by** *blast*
qed

The next function *iterate-remove-ld* has done that we have to install *Isabelle2011*. In previous versions we have to make use of *function* (*tailrec*), but this element had some bugs. In particular, we could not use *function* (*tailrec*) in the next definition.

partial-function (*tailrec*) *iterate-remove-ld* :: *'c set* => (*nat* => *'c*) => *'c set*
where *iterate-remove-ld A f* = (*if linear-independent A then A*
else iterate-remove-ld (*fst* (*remove-ld* (*A*, *f*)))
(*snd* (*remove-ld* (*A*, *f*))))

declare *iterate-remove-ld.simps* [*simp del*]

Its behaviour is the next: from a set and a indexation of it, we apply recursively the operation *remove-ld* up to we achieve a linearly independent set. The reiterated elimination of the linearly dependent elements would have to keep the span.

If we call to the function *iterate-remove-ld* with a linearly independent set, it will return us that set.

lemma *iterate-remove-ld-empty* [*simp*]: *iterate-remove-ld* {} *f* = {}
unfolding *iterate-remove-ld.simps* [*of* {}] **by** *simp*

lemma
iterate-remove-ld-li [*simp*]:
assumes *li-A*: *linear-independent A*
shows *iterate-remove-ld A f* = *A*
using *iterate-remove-ld.simps* **using** *li-A* **by** *simp*

Now we are going to prove some lemmas about indexings and *remove-iset*. Note that we can not put this lemmas in the file *Indexed-Set.thy* because the axioms *good-set* and *linear-dependent* are sometimes in the premises.

The next lemma express that the result of aplying *remove-iset* preserves the good set property. In our context we need to prove it for *remove-ld*...but it does not cease to be a particular case of *remove-iset*.

lemma
remove-iset-good-set:

assumes *c*: *good-set A*
and *i*: *indexing* (*A*, *h*)
shows *good-set* (*fst* (*remove-iset* (*A*, *h*) *n*))
using *c*
unfolding *good-set-def*
unfolding *remove-iset-def* **by** *auto*

lemma
remove-ld-good-set:
assumes *c*: *good-set A*
and *i*: *indexing* (*A*, *h*)
shows *good-set* (*fst* (*remove-ld* (*A*, *h*)))
unfolding *remove-ld-def*
unfolding *Let-def*
by (*rule remove-iset-good-set*) *fact+*

Next theorem applies $[\![$*indexing* (*?B*, *?h*); *?n* < *card ?B*$]\!] \Longrightarrow$ *indexing* (*remove-iset* (*?B*, *?h*) *?n*) to the function *remove-ld*. We can omit the good set condition: it is implicit in the fact that the set is linearly dependent.

theorem *indexing-remove-ld*:
assumes *l*: *linear-dependent A*
and *i*: *indexing* (*A*, *f*)
and *n*: $\mathbf{0}_V \notin A$
shows *indexing* (*remove-ld* (*A*, *f*))
unfolding *remove-ld-def*
unfolding *Let-def*
proof (*rule indexing-remove-iset*, *unfold fst-conv snd-conv*)
show *indexing* (*A*, *f*) **by** *fact*
show (*LEAST k*. ∃ *y*∈*A*. ∃ *g*. *g* ∈ *coefficients-function* (*carrier V*) ∧
1 ≤ *k* ∧
k < *card A* ∧
f k = *y* ∧
y = *linear-combination g* (*f* ‘ {*i*. *i* < *k*})) < *card A*
proof −
let *?P* = (λ*k*. ∃ *y*∈*A*. ∃ *g*. *g* ∈ *coefficients-function* (*carrier V*) ∧ *1* ≤ *k* ∧
k < *card A* ∧ *f k* = *y* ∧ *y* = *linear-combination g* (*f* ‘ {*i*. *i* < *k*}))
have *fin-A*: *finite A*
using *l-dep-good-set* [*OF l*]
unfolding *good-set-def* **by** *fast*
have *exK*: (∃ *k*. *?P k*)
using *linear-dependent-set-sorted-contains-linear-combination* [
OF l n i] **by** *auto*
have *ex-LEAST*: *?P* (*LEAST k*. *?P k*)
using *LeastI-ex* [*OF exK*] **.**
let *?k* = (*LEAST k*. *?P k*)
have ∃ *y*∈*A*. ∃ *g*. *g* ∈ *coefficients-function* (*carrier V*) ∧ *1* ≤ *?k* ∧
?k < *card A* ∧ *f ?k* = *y* ∧ *y* = *linear-combination g* (*f* ‘ {*i*. *i* < *?k*})
using *ex-LEAST* **by** *simp*
thus *?thesis* **by** *auto*

qed
qed

Next lemma shows us that first element of a indexed set is in the carrier. Note that we can not put this lemma in the file *Indexed Set* due to the axiom $A \subseteq$ *carrier* V (we have not a structure of carrier in that file).

lemma *f0-in-V*:
assumes *indexing-A*: *indexing* (A,f)
and *A-in-V*: $A \subseteq$ *carrier* V
and *A-not-empty*: $A \neq \{\}$ — Essential to cardinality
shows $f\ 0 \in$ *carrier* V
proof −
have $A \neq \{\}$ **using** *A-not-empty* **.**
hence $0 \in \{..<card\ A\}$
using *card-gt-0-iff* **and** *indexing-finite*[*OF indexing-A*]
using *card-gt-0-iff lessThan-iff* **by** *blast*
thus *?thesis*
using *indexing-A A-in-V* **unfolding** *indexing-def bij-betw-def* **by** *auto*
qed

If A is independent, then its firts element is not zero.

lemma *f0-not-zero*:
assumes *indexing-A*: *indexing* (A,f)
and *li-A*: *linear-independent* A
and *A-not-empty*: $A \neq \{\}$
shows $f\ 0 \neq \mathbf{0}_V$
proof −
have *zero-not-in-A*: $\mathbf{0}_V \notin A$ **using** *zero-not-in-linear-independent-set*[*OF li-A*] **.**
have $0 \in \{..<card\ A\}$ **using** *A-not-empty* **and** *indexing-finite*[*OF indexing-A*]
using *card-gt-0-iff lessThan-iff* **by** *blast*
thus *?thesis* **using** *indexing-A* **and** *zero-not-in-A* **unfolding** *indexing-def bij-betw-def*
by *force*
qed

We can also prove that apply the function *insert-iset* return us a good set.

lemma *insert-iset-good-set*:
assumes *a-notin-A*: $a \notin A$
and *indexing*: *indexing* (A,f)
and *a-in-V*: $a \in$ *carrier* V
and *cb-A*: *good-set* A
shows *good-set* (*fst*(*insert-iset* (A,f) a n))
unfolding *insert-iset-def* **using** *a-in-V cb-A* **unfolding** *good-set-def* **by** *simp*

Remove an element and after that insert it is a good set

lemma *good-set-insert-remove*:
assumes *B-in-V*: $B \subseteq$ *carrier* V
and *A-in-V*: $A \subseteq$ *carrier* V
and *A-not-empty*: $A \neq \{\}$

and *indexing-A*: *indexing* (A,f)
and *indexing-B*: *indexing* (B,g)
and *a-in-B*: $a \in B$
shows *good-set* (*fst* (*insert-iset* (*remove-iset* $(B,\ g)$ (*obtain-position* a $(B,\ g)$))) a n))
proof −
have *cb-A*: *good-set* A **using** *A-in-V* *indexing-finite*[*OF* *indexing-A*] **unfolding** *good-set-def* **by** *simp*
have *cb-B*: *good-set* B **using** *B-in-V* *indexing-finite*[*OF* *indexing-B*] **unfolding** *good-set-def* **by** *simp*
have *good-set* (*fst* (*insert-iset* ((*fst*(*remove-iset* $(B,\ g)$
(*obtain-position* a $(B,\ g)$))),*snd*(*remove-iset* $(B,\ g)$ (*obtain-position* a $(B,\ g)$)))) a n))
proof (*rule* *insert-iset-good-set*)
show $a \notin$ *fst* (*remove-iset* $(B,\ g)$ (*obtain-position* a $(B,\ g)$)) **using** *a-notin-remove-iset*[*OF* *a-in-B* *indexing-B*] **.**
show *indexing*
(*fst* (*remove-iset* $(B,\ g)$ (*obtain-position* a $(B,\ g)$)),
snd (*remove-iset* $(B,\ g)$ (*obtain-position* a $(B,\ g)$)))
using *indexing-remove-iset*[*OF* *indexing-B* *obtain-position-less-card*[*OF* *a-in-B* *indexing-B*]] **by** *simp*
show $a \in$ *carrier* V **using** *a-in-B* *B-in-V* **by** *fast*
show *good-set* (*fst* (*remove-iset* $(B,\ g)$ (*obtain-position* a $(B,\ g)$)))
using *remove-iset-good-set*[*OF* *cb-B* *indexing-B*] **.**
qed
thus *?thesis* **by** *simp*
qed

The result of applying the function *iterate-remove-ld* to any finite set in *carrier* V will be always independent (the function finishes).

We are going to make the proof firstly by dividing in cases (with respect to the condiction *linear-independent* A) and after that by total induction over the cardinal of the set: $(\bigwedge x.\ (\bigwedge y.\ f\ y < f\ x \Longrightarrow P\ y) \Longrightarrow P\ x) \Longrightarrow P\ a$.

With respect to the induction, it is important to note that we can not make induction over the *structure* of the set, with the following induction rule for indexed set that we have introduced in section **??**:

indexed-set-induct2: $[\![$*indexing* $(A,\ f)$; *finite* A; $\bigwedge f.$ *indexing* $(\{\},\ f) \Longrightarrow P$ $\{\}$ f; $\bigwedge a\ A\ f\ n.$ $[\![a \notin A$; *indexing* $(A,\ f) \Longrightarrow P\ A\ f$; *finite* (*insert* a A); *indexing* (*insert* a A, *indexing-ext* $(A,\ f)$ a n); $0 \leq n$; $n \leq$ *card* $A]\!] \Longrightarrow P$ (*insert* a A) (*indexing-ext* $(A,\ f)$ a n)$]\!] \Longrightarrow P\ A\ f$

If we make induction over the structure, we will have to prove the case *insert* a A and the induction hypothesis will say that the result is true for A. However, independently of in what position of the indexation we place the element a, we can not know if *remove-ld* (*insert* a A, *indexing-ext* $(A,\ f)$ a n) will return the same set A or it will return another set. In other words: the result of inserting the element a in any position of the A set and after that removing the least element which is a linear combination of the preceding ones (*remove-ld* does it) is not equal to the original set. We can not ensure it even when we insert the element a in the least position that it can be expressed as a linear combination of the preceding ones: we can not be sure that *remove-ld* will remove that element. For example, in the set $\{(1,\ 0), (2,\ 0), (0,\ 1)\}$, if we insert the element $(0,\ 2)$ in the least position where it is a linear combination of the preceding ones we achieve the set $\{(1,\ 0), (2,\ 0), (0,\ 1), (0,\ 2)\}$. However, if we apply *remove-ld* to this set, it will return $\{(1,\ 0), (0,\ 1), (0,\ 2)\}$ and this is not equal to the original set.

lemma
  linear-independent-iterate-remove-ld:
  assumes *A-in-V*: $A \subseteq$ *carrier V*
  and *not-zero*: $\mathbf{0}_V \notin A$
  and *indexing-A-f*: *indexing* (A, h)
  shows *linear-independent* (*iterate-remove-ld A h*)
proof (*cases linear-independent A*)
  case *True*
  show *?thesis* **using** *True* **by** *simp*
next
  case *False*
  have *fin-A*: *finite A* **using** *indexing-finite*[*OF indexing-A-f*] **.**
  have *ld-A*: *linear-dependent A*
    using *not-independent-implies-dependent* [*OF - False*]
    unfolding *good-set-def* **using** *fin-A A-in-V* **by** *fast*
  show *?thesis*
    using *fin-A ld-A A-in-V not-zero indexing-A-f*
    — HERE WE APPLY THE INDUCTION RULE:
  proof (*induct A arbitrary*: *h rule*:
      measure-induct-rule [**where** f = *card*])
    case (*less B h*)
    show *linear-independent* (*iterate-remove-ld B h*)
    proof (*cases B* = {})
      case *True*
      thus *?thesis* **by** *simp*
    next
      case *False*
      have *not-lin-indep*: $\neg$ *linear-independent B*
        using *dependent-implies-not-independent*
        [*OF less.prems* (*2*)] **.**
      obtain *Y* **where** *Y-def*: *Y* = *fst* (*remove-ld* (*B*, *h*))
        and *card-less*: *card Y* < *card B*
        using *False*
        using *remove-ld-decr-card2*
        [*OF less.prems* (*2*) *less.prems* (*4*) *less.prems* (*5*)]
        by *fast*
      def h' == *snd* (*remove-ld* (*B*, *h*))

have *i-Y-h'*: *indexing* (Y, h')
unfolding *Y-def h'-def pair-collapse*
by (*rule indexing-remove-ld*) *fact+*
show *?thesis*
proof (*cases linear-independent* (*fst* (*remove-ld* (B, h))))
case *True*
show *?thesis*
apply (*subst iterate-remove-ld.simps*)
apply (*subst iterate-remove-ld.simps*)
using *not-lin-indep* **using** *True* **by** *simp*
next
case *False*
show *?thesis*
proof −
have *linear-independent* (*iterate-remove-ld* Y h')
proof (*rule less.hyps*)
show *card* $Y <$ *card* B
using *card-less* **.**
show *finite* Y
using *Y-def good-set-finite l-dep-good-set*
less(*3*) *less*(*6*) *remove-ld-good-set* **by** *presburger*
show *linear-dependent* Y
unfolding *Y-def*
apply (*rule not-independent-implies-dependent*
[*OF* - *False*])
apply (*rule remove-ld-good-set*)
apply (*unfold good-set-def*, *intro conjI*)
by (*rule less.prems* (*1*), *rule less.prems* (*3*),
rule less.prems (*5*))
show $Y \subseteq$ *carrier* V
unfolding *Y-def*
using *remove-ld-preserves-carrier*
[*OF less.prems* (*3*), *of h*] **.**
show $\mathbf{0}_V \notin Y$
unfolding *Y-def*
using *remove-ld-monotone* [*OF less.prems* (*3*), *of h*]
using *less.prems* (*4*) **by** *auto*
show *indexing* (Y, h')
unfolding *Y-def h'-def pair-collapse*
by (*rule indexing-remove-ld*) *fact+*
qed
thus *?thesis*
unfolding *Y-def h'-def*
by (*subst iterate-remove-ld.simps*,
simp add: *not-lin-indep*)
qed
qed
qed
qed

qed

Similarly to the previous theorem, we can prove that the function *iterate-remove-ld* preserves the *span*.

```
lemma iterate-remove-ld-preserves-span:
  assumes A-in-V: A ⊆ carrier V
  and indexing-A-f: indexing (A,h)
  and not-zero: 0_V ∉ A
  shows span (iterate-remove-ld A h) = span A
proof (cases linear-independent A)
  case True
  show ?thesis using True by simp
next
  case False
  have fin-A: finite A using indexing-finite[OF indexing-A-f] .
  have ld-A: linear-dependent A
    using not-independent-implies-dependent [OF - False]
    unfolding good-set-def using fin-A A-in-V by fast
  show ?thesis
    using fin-A ld-A A-in-V not-zero indexing-A-f
  proof (induct A arbitrary: h rule: measure-induct-rule [where f = card])
    case (less B h)
    show span (iterate-remove-ld B h) = span B
    proof (cases B = {})
      case True
      thus ?thesis by simp
    next
      case False
      have not-lin-indep: ¬ linear-independent B
        using dependent-implies-not-independent [OF less.prems (2)] .
      obtain Y where Y-def: Y = fst (remove-ld (B, h))
        and card-less: card Y < card B
        using False
         using remove-ld-decr-card2 [OF less.prems (2) less.prems (4) less.prems
(5)] by fast
      def h' == snd (remove-ld (B, h))
      have i-Y-h': indexing (Y, h')
        unfolding Y-def h'-def pair-collapse
        by (rule indexing-remove-ld) fact+
      show ?thesis
      proof (cases linear-independent (fst (remove-ld (B, h))))
        case True
        show ?thesis
          apply (subst iterate-remove-ld.simps)
          apply (subst iterate-remove-ld.simps)
          using not-lin-indep using True
          apply simp
          proof (rule remove-ld-preserves-span)
            show linear-dependent B using less(3) .
```

show $\mathbf{0}_V \notin B$ **using** *less*(*5*) **.**
show *indexing* (*B*, *h*) **using** *less*(*6*) **.**
qed
next
case *False*
show *?thesis*
proof −
have *span* (*iterate-remove-ld Y h'*) = *span Y*
proof (*rule less.hyps*)
show *card Y* < *card B*
using *card-less* **.**
show *finite Y*
using *Y-def less*(*2*) *less*(*4*) *remove-ld-monotone rev-finite-subset* **by** *metis*
show *linear-dependent Y*
unfolding *Y-def*
proof (*rule not-independent-implies-dependent*)
show *good-set* (*fst* (*remove-ld* (*B*, *h*)))
using *remove-ld-good-set less.prems*(*1*) *less.prems*(*3*) *less.prems*(*5*)
unfolding *good-set-def* **by** *auto*
show ¬ *linear-independent* (*fst* (*remove-ld* (*B*, *h*))) **using** *False* **.**
qed
show $Y \subseteq$ *carrier V*
unfolding *Y-def*
using *remove-ld-preserves-carrier* [*OF less.prems* (*3*), *of h*] **using** *A-in-V* **by** *auto*
show $\mathbf{0}_V \notin Y$
unfolding *Y-def*
using *remove-ld-monotone* [*OF less.prems* (*3*), *of h*]
using *less.prems* (*4*) **by** *auto*
show *indexing* (*Y*, *h'*)
unfolding *Y-def h'-def pair-collapse*
by (*rule indexing-remove-ld*) *fact+*
qed
thus *?thesis*
unfolding *Y-def h'-def*
using *iterate-remove-ld.simps less*(*3*) *less*(*5*) *less*(*6*) *not-lin-indep remove-ld-preserves-span* **by** *auto*
qed
qed
qed
qed
qed

If we have an *indexing* ($A \cup B$, h) where elements of an independent set A are in its first positions and after those the elements of a set B, then A will be in *remove-ld* ($A \cup B$, h) (we will have removed an element of B). The premisse of $A \cap B = \{\}$ is indispensable in order to avoid the notion of multisets. In the book, Halmos doesn't worry about this: he simply create a set with all elements of A in the first positions and after that all elements of B...but what does it happen if a element of B are in A? we will have a multiset because we have the same element in two positions. However, this is not a limitation for our theorem if we make a trick like these: $A \cup B = A \cup (B - A)$. Using that we avoid the problem.

```
lemma A-in-remove-ld:
  assumes indexing: indexing (A∪B,h)
  and ld-AB: linear-dependent (A∪B)
  and surj-h-A:h‘ {..<card(A)}= A

  and li-A: linear-independent A
  and zero-not-in: 0_V ∉ (A∪B)
  and disjuntos: A∩B={}
  shows A ⊆ fst (remove-ld ((A∪B),h))
proof –
  have cb-A: good-set A and cb-B: good-set B
    using l-dep-good-set[OF ld-AB] unfolding good-set-def by auto
  let ?P = (λk. ∃ y∈A∪B. ∃ g. g ∈ coefficients-function (carrier V) ∧ 1 ≤ k ∧
    k < card (A∪B) ∧ h k = y ∧ y = linear-combination g (h ‘ {i. i < k}))
  have exK: (∃ k. ?P k)
    using linear-dependent-set-sorted-contains-linear-combination [
      OF ld-AB zero-not-in indexing] by auto
  have ex-LEAST: ?P (LEAST k. ?P k)
    using LeastI-ex [OF exK] .
  let ?k = (LEAST k. ?P k)
  have ∃ y∈A∪B. ∃ g. g ∈ coefficients-function (carrier V) ∧ 1 ≤ ?k ∧
    ?k < card (A∪B) ∧ h ?k = y ∧ y = linear-combination g (h ‘ {i. i < ?k})
    using ex-LEAST by simp
  then obtain y s
    where one-le-k: 1 ≤ ?k and k-l-card: ?k < card (A∪B) and h-k-eq-y: h ?k =
y
    and cf-s: s∈ coefficients-function (carrier V) and
    combinacion-anteriores: y = linear-combination s (h ‘ {i. i < ?k})
    by auto
   have rem-eq: fst (remove-ld (A∪B, h)) = ((A∪B) – {y}) and y-in-AB: y ∈
A∪B
    using indexing-equiv-img [OF indexing] one-le-k k-l-card h-k-eq-y
    unfolding Pi-def unfolding remove-ld-def' by auto
  show ?thesis
  proof (cases y∈B)
    case True thus ?thesis using rem-eq and disjuntos by auto
  next
    case False show ?thesis
    proof –
      have y-in-A: y∈A using False and y-in-AB by simp
      have k-le-cardA: ?k<card(A) — It takes about a seconds
        using y-in-A and h-k-eq-y and surj-h-A and k-l-card and indexing
        unfolding indexing-def and bij-betw-def and inj-on-def
```

```
        by force
      have ld-insert: linear-dependent (insert y (h‘{i. i<?k}))
      proof (rule lc1)
        show linear-independent (h‘{i. i<?k})
        proof (rule independent-set-implies-independent-subset)
          show linear-independent A using li-A .
          show h ‘ {i. i < ?k} ⊆ A using surj-h-A k-le-cardA by auto
        qed
        show y∈ carrier V using y-in-A cb-A unfolding good-set-def
          by auto
        show y∉ h‘{i. i<?k}
          using y-in-A and h-k-eq-y and surj-h-A and k-l-card and indexing
          unfolding indexing-def and bij-betw-def and inj-on-def
          by force
        show ∃f. f ∈ coefficients-function (carrier V) ∧
          linear-combination f (h ‘{i. i<?k})=y
          using cf-s and combinacion-anteriores by auto
      qed
      have linear-dependent (h‘{..<card(A)})
      proof (rule linear-dependent-subset-implies-linear-dependent-set)
        show insert y (h‘{i. i<?k})⊆ h ‘ {..<card A}
        proof −
           have igualdad-conjuntos: {i. i<?k} ∪ {?k}={..?k} using atMost-def[of
?k] ivl-disj-un(2) by auto
          have insert y (h‘{i. i<?k})=(h‘{i. i<?k}) ∪ {y} by simp
          also have ...=h‘{i. i<?k} ∪ h‘ {?k}  using h-k-eq-y by auto
          also have ...=h‘ ({i. i<?k} ∪ {?k}) by auto
          also have ...=h‘{..?k} using igualdad-conjuntos by auto
          also have ... ⊆ h ‘ {..<card A} using k-le-cardA by auto
          finally show ?thesis .
        qed next
        show good-set (h ‘ {..<card A})
          using surj-h-A cb-A by auto
        show linear-dependent (insert y (h‘{i. i<?k})) using ld-insert .
      qed
      — Contradiction: we have linear dependent A and linear independent A
      thus ?thesis using surj-h-A li-A cb-A independent-implies-not-dependent by
auto
    qed
  qed
qed
```

This lemma is an extended version of previous one. It shows that we are removing one element of the second set and preserving the indexation.

```
lemma descomposicion-remove-ld:
  assumes indexing: indexing (A∪B,h)
  and B-not-empty: B≠{} — Due to cardinality, it is indispensable.
  and surj-h-A:h‘ {..<card(A)}= A
  and surj-h-B:h‘ ({..<(card(A)+card(B))}−{..<card(A)})=B
```

```
  and li-A: linear-independent A
  and zero-not-in: 0_V ∉ (A∪B)
  and ld-AB: linear-dependent (A∪B)
  and disjuntos: A∩B={}
 shows ∃y. fst (remove-ld ((A∪B),h))=A∪(B−{y}) ∧ y∈B
  ∧ (snd (remove-ld (A∪B, h))) ‘ ({..<card A + card (B−{y})} − {..<card A})
= (B−{y})
  ∧ snd (remove-ld((A∪B), h)) ‘ {..<card A}=A ∧ indexing (A ∪ (B−{y}), snd
(remove-ld (A∪B, h)))
proof −
 have cb-AB: good-set (A∪B) using l-dep-good-set[OF ld-AB] unfolding good-set-def
by auto
  have cb-A: good-set A and cb-B:good-set B
    using l-dep-good-set[OF ld-AB] unfolding good-set-def by auto
  let ?P = (λk. ∃y∈A∪B. ∃g. g ∈ coefficients-function (carrier V) ∧ 1 ≤ k ∧
    k < card (A∪B) ∧ h k = y ∧ y = linear-combination g (h ‘ {i. i < k}))
  have exK: (∃k. ?P k)
    using linear-dependent-set-sorted-contains-linear-combination [
      OF ld-AB zero-not-in indexing] by auto
  have ex-LEAST: ?P (LEAST k. ?P k)
    using LeastI-ex [OF exK] .
  let ?k = (LEAST k. ?P k)
  have ∃y∈A∪B. ∃g. g ∈ coefficients-function (carrier V) ∧ 1 ≤ ?k ∧
    ?k < card (A∪B) ∧ h ?k = y ∧ y = linear-combination g (h ‘ {i. i < ?k})
    using ex-LEAST by simp
  then obtain y s
    where one-le-k: 1 ≤ ?k and k-l-card: ?k < card (A∪B) and h-k-eq-y: h ?k =
y
    and cf-s: s∈ coefficients-function (carrier V) and
    combinacion-anteriores: y = linear-combination s (h ‘ {i. i < ?k})
    by auto
   have rem-eq: fst (remove-ld (A∪B, h)) = ((A∪B) − {y}) and y-in-AB: y ∈
A∪B
    using indexing-equiv-img [OF indexing] one-le-k k-l-card h-k-eq-y
    unfolding Pi-def unfolding remove-ld-def′ by auto
  show ?thesis
  proof (cases y∈B)
    case True thus ?thesis
    proof −
      have y-notin-A: y∉A using True disjuntos y-in-AB by blast
      have k-in-conjunto:?k∈{..<card(A)+card(B)}−{..<card(A)}
      proof −
        have card(A∪B)=card(A)+card(B) using disjuntos
          card-Un-disjoint cb-A cb-B unfolding good-set-def by blast
        hence k-in-cardAB: ?k∈{..<card(A)+card(B)} using k-l-card by auto
        have ?k∉{..<card(A)} using h-k-eq-y True surj-h-A y-notin-A by auto
        thus ?thesis using k-in-cardAB by simp
      qed
      have 1: fst (remove-ld (A ∪ B, h)) = A ∪ (B−{y})
```

using rem-eq **and** disjuntos True **by** auto
have 2: (snd (remove-ld (A∪B, h))) ‘ ({..<card A + card (B−{y})} − {..<card A}) = (B−{y})
∧ snd (remove-ld((A∪B), h)) ‘ {..<card A}=A
∧ indexing (A ∪ (B−{y}), snd (remove-ld (A∪B, h)))
proof −
have eq-card: card(fst(remove-iset((A∪B),h) ?k))=card(A)+card(B−{y})
proof −
have cardB-g-zero: card B > 0 **using** B-not-empty cb-B **unfolding** good-set-def **by** auto
hence finite-B: finite B **using** cb-B **unfolding** good-set-def **by** simp
have 1: card(B−{y})=card(B)−Suc 0 **using** card-Diff-singleton[OF finite-B True] **by** simp
have card (fst (remove-ld ((A∪B), h))) = card (A∪B) − Suc 0
using remove-ld-decr-card indexing-remove-ld indexing ld-AB zero-not-in **by** auto
also have ...=card(A)+card(B)−Suc 0 **using** disjuntos card-Un-disjoint cb-A cb-B
unfolding good-set-def **by** force
also have ...=card(A)+(card(B)−Suc 0) **using** cardB-g-zero **by** auto
finally have card (fst (remove-ld (A ∪ B, h))) = card A + (card B − Suc 0) .
thus ?thesis **using** 1 **unfolding** remove-ld-def **by** auto
qed
have eq: snd (remove-ld (A ∪ B, h)) = snd (remove-iset ((A∪B),h) ?k)
unfolding remove-ld-def **using** snd-conv **using** remove-iset-def[of (A∪B,h) ?k] **by** auto
have surj-rmiset-A: snd (remove-iset((A∪B), h) ?k) ‘ {..<card A}=A
proof −
have ?k≥card A
proof (cases ?k<card A)
case False **thus** ?thesis **by** simp
next
case True **thus** ?thesis **using** surj-h-A h-k-eq-y y-notin-A **by** auto
qed
hence snd (remove-iset((A∪B), h) ?k) ‘ {..<card A}=h‘{..<card A} **unfolding** remove-iset-def **by** auto
thus ?thesis **using** surj-h-A **by** simp
qed
have indexing2: indexing (A ∪ (B−{y}), snd (remove-ld (A∪B, h)))
proof −
have indexing (A ∪ (B−{y}), snd (remove-ld (A∪B, h)))
=indexing (fst(remove-ld (A∪B,h)), snd (remove-ld (A∪B, h)))
using eq 1 **by** auto
also have ...=indexing (remove-ld (A∪B,h)) **by** auto
finally show ?thesis **using** indexing-remove-ld[OF ld-AB indexing zero-not-in] **by** auto
qed
have snd (remove-iset ((A∪B),h) ?k) ‘{..<card(fst(remove-iset((A∪B),h)

```
?k))}=fst(remove-iset ((A∪B),h) ?k)
        using indexing-remove-iset[OF indexing k-l-card]
        unfolding indexing-def and bij-betw-def by auto
      also have ...=(A∪B)−{h ?k} unfolding remove-iset-def by auto
      also have ...=A∪(B−{y}) using h-k-eq-y y-notin-A by auto
    finally have eq-final: snd (remove-iset ((A∪B),h) ?k) `{..<card(fst(remove-iset((A∪B),h)
?k))}=A∪(B−{y}) .
     have (snd (remove-ld (A∪B, h))) ` ({..<card A + card (B−{y})} − {..<card
A})
        = (snd (remove-ld (A∪B, h))) ` {..<card A + card (B−{y})}
        − (snd (remove-ld (A∪B, h))) ` {..<card A}
      proof (rule inj-on-image-set-diff[of - {..<card A + card (B−{y})}],auto)
       show inj-on (snd (remove-ld (A ∪ B, h))) {..<card A + card (B − {y})}
          using eq and eq-card
          using indexing-remove-iset[OF indexing k-l-card]
          unfolding indexing-def and bij-betw-def by auto
      qed
    also have ...=snd (remove-iset ((A∪B),h) ?k) ` {..<card A + card (B−{y})}

        − snd (remove-iset ((A∪B),h) ?k) `{..<card(A)}
        using eq by auto
      also have ...=(A∪(B−{y}))−A using eq-final surj-rmiset-A eq eq-card by
auto
      also have ...=B−{y} using disjuntos y-notin-A True by auto
      finally show ?thesis using surj-rmiset-A eq indexing2 by auto
    qed
    show ?thesis using 1 2 True by auto
  qed
 next
  case False show ?thesis
  proof −
    have y-in-A: y∈A using False and y-in-AB by simp
    have k-le-cardA: ?k<card(A) — It takes about a seconds
      using y-in-A and h-k-eq-y and surj-h-A and k-l-card and indexing
      unfolding indexing-def and bij-betw-def and inj-on-def
      by force
    have ld-insert: linear-dependent (insert y (h`{i. i<?k}))
    proof (rule lc1)
      show linear-independent (h`{i. i<?k})
      proof (rule independent-set-implies-independent-subset)
        show linear-independent A using li-A .
        show h ` {i. i < ?k} ⊆ A using surj-h-A k-le-cardA by auto
      qed
      show y∈ carrier V using y-in-A cb-A unfolding good-set-def
        by auto
      show y∉ h`{i. i<?k}
        using y-in-A and h-k-eq-y and surj-h-A and k-l-card and indexing
        unfolding indexing-def and bij-betw-def and inj-on-def
        by force
```

show $\exists f.\ f \in$ *coefficients-function* (*carrier V*) $\wedge$
linear-combination f (h ‘$\{i.\ i<?k\}$)=y
using *cf-s* **and** *combinacion-anteriores* **by** *auto*
qed
have *linear-dependent* (h‘$\{..<card(A)\}$)
proof (*rule linear-dependent-subset-implies-linear-dependent-set*)
show *insert y* (h‘$\{i.\ i<?k\}$)$\subseteq$ h ‘ $\{..<card\ A\}$
proof –
have *igualdad-conjuntos*: $\{i.\ i<?k\} \cup \{?k\}=\{..?k\}$ **using** *atMost-def*[*of* $?k$] *ivl-disj-un*(2) **by** *auto*
have *insert y* (h‘$\{i.\ i<?k\}$)=(h‘$\{i.\ i<?k\}$) $\cup$ $\{y\}$ **by** *simp*
also have ...=h‘$\{i.\ i<?k\}$ $\cup$ h‘ $\{?k\}$ **using** *h-k-eq-y* **by** *auto*
also have ...=h‘ ($\{i.\ i<?k\} \cup \{?k\}$) **by** *auto*
also have ...=h‘$\{..?k\}$ **using** *igualdad-conjuntos* **by** *auto*
also have ... $\subseteq$ h ‘ $\{..<card\ A\}$ **using** *k-le-cardA* **by** *auto*
finally show *?thesis* **.**
qed
show *good-set* (h ‘ $\{..<card\ A\}$)
using *surj-h-A cb-A* **by** *auto*
show *linear-dependent* (*insert y* (h‘$\{i.\ i<?k\}$)) **using** *ld-insert* **.**
qed
— Contradiction: we have linear dependent A and linear independent A

thus *?thesis* **using** *surj-h-A li-A cb-A independent-implies-not-dependent* **by** *auto*
qed
qed
qed

Finally an important lemma proved using ($\bigwedge x.\ (\bigwedge y.\ ?f\ y < ?f\ x \Longrightarrow ?P\ y) \Longrightarrow ?P\ x) \Longrightarrow ?P\ ?a$ such as we do in *linear-independent-iterate-remove-ld* and in *iterate-remove-ld-preserves-span*. We need above lemmas to prove it. It shows us that *iterate-remove-ld* does not remove any element of A if elements of A are in first positions and A is linearly independent.

lemma *A-in-iterate-remove-ld*:
assumes *indexing*: *indexing* ($A \cup B$,h)
and *B-in-V*: $B \subseteq$ *carrier V*
and *surj-h-A*:h‘ $\{..<card(A)\}$= A
and *surj-h-B*:h‘ ($\{..<(card(A)+card(B))\}-\{..<card(A)\}$)=$B$
and *li-A*: *linear-independent A*
and *zero-not-in*: $\mathbf{0}_V \notin (A \cup B)$
and *disjuntos*: $A \cap B=\{\}$
shows $A \subseteq$ (*iterate-remove-ld* ($A \cup B$) h)
proof (*cases linear-dependent* ($A \cup B$))
have *cb-A*: *good-set A* **using** *l-ind-good-set*[*OF li-A*] **.**
have *cb-B*: *good-set B* **using** *indexing-finite*[*OF indexing*] *B-in-V* **unfolding** *good-set-def* **by** *fast*
case *False* **thus** *?thesis*
proof –

have *linear-independent* $(A \cup B)$
using *cb-A cb-B not-dependent-implies-independent*[*OF - False*]
unfolding *good-set-def* **by** *auto*
hence *iterate-remove-ld* $(A \cup B)$ $h = A \cup B$
using *iterate-remove-ld-li* **by** *simp*
thus *?thesis* **by** *simp*
qed
next
case *True*
have *cb-B*: *good-set B* **using** *indexing-finite*[*OF indexing*] *B-in-V* **unfolding** *good-set-def* **by** *fast*
show *?thesis*
using *cb-B* **and** *True* **and** *surj-h-A* **and** *surj-h-B* **and** *zero-not-in* **and** *disjuntos* **and** *indexing*
proof (*induct B arbitrary*: *h rule*: *measure-induct-rule* [**where** f = *card*])
case (*less B h*)
show $A \subseteq$ *iterate-remove-ld* $(A \cup B)$ h
proof (*cases* $B=\{\}$)
case *True*
thus *?thesis*
using *Int-lower1 Un-absorb2 disjuntos iterate-remove-ld-li li-A subset-refl* **by** *force*
next
case *False*
have $\exists y.$ *fst* (*remove-ld* $((A \cup B),h)) = A \cup (B - \{y\}) \wedge y \in B$
$\wedge$ (*snd* (*remove-ld* $(A \cup B,\ h)))$ ‘ $(\{..<$*card* A + *card* $(B - \{y\})\} - \{..<$*card* $A\}) = (B - \{y\})$
$\wedge$ *snd* (*remove-ld*$((A \cup B),\ h))$ ‘ $\{..<$*card* $A\} = A \wedge$ *indexing* $(A \cup (B - \{y\})$, *snd* (*remove-ld* $(A \cup B,\ h)))$
proof (*rule descomposicion-remove-ld*)
show *indexing* $(A \cup B,\ h)$ **using** *less.prems*(*7*) **.**
show *linear-dependent* $(A \cup B)$ **using** *less*(*3*) **.**
show $\mathbf{0}_V \notin A \cup B$ **using** *less.prems*(*5*) **.**
show $A \cap B = \{\}$ **using** *less.prems*(*6*) **.**
show *linear-independent* A **using** *li-A* **.**
show h ‘ $\{..<$*card* $A\} = A$ **using** *less.prems*(*3*) **.**
show h ‘ $(\{..<$*card* A + *card* $B\} - \{..<$*card* $A\}) = B$ **using** *less.prems*(*4*) **.**
show $B \neq \{\}$ **using** *False* **.**
qed
from *this* **obtain** y
where *descomposicion*: *fst* (*remove-ld* $((A \cup B),h)) = A \cup (B - \{y\})$
and *y-in-B*: $y \in B$
and *h'-B*: (*snd* (*remove-ld* $(A \cup B,\ h)))$ ‘ $(\{..<$*card* A + *card* $(B - \{y\})\} - \{..<$*card* $A\}) = (B - \{y\})$
and *h'-A*: *snd* (*remove-ld*$((A \cup B),\ h))$ ‘ $\{..<$*card* $A\} = A$
and *indexing2*: *indexing* $(A \cup (B - \{y\})$, *snd* (*remove-ld* $(A \cup B,\ h)))$
by *auto*
have *card-less*: *card*$(B - \{y\}) <$ *card*(B) **using** *y-in-B* **and** *less*(*2*)
unfolding *good-set-def* **using** *card-Diff1-less*[*of B y*] **by** *auto*

```
    have By-subset-B: (B−{y})⊆B by blast
    have not-lin-indep: ¬ linear-independent (A∪B)
      using dependent-implies-not-independent [OF less.prems (2)] .
    def h' == snd (remove-ld (A∪B, h))
    show ?thesis
    proof (cases linear-independent (fst (remove-ld (A∪B, h))))
      case True
      show ?thesis
        apply (subst iterate-remove-ld.simps)
        apply (subst iterate-remove-ld.simps)
        using not-lin-indep and True
        apply simp
       using A-in-remove-ld[OF less.prems(7) less(3) less.prems(3) li-A less.prems(5) less.prems(6)] by simp
    next
      case False
      show ?thesis
      proof −
          have cb-By: good-set (B−{y}) using less.prems(1) y-in-B unfolding good-set-def by auto
        have A⊆iterate-remove-ld (A∪(B−{y})) h'
        proof (cases linear-dependent (A∪(B−{y})))
          case False
          show ?thesis
            using cb-By iterate-remove-ld-li not-dependent-implies-independent[OF - False]
            using l-ind-good-set[OF li-A]
            unfolding good-set-def by auto
        next
          case True show ?thesis
          proof (rule less.hyps)
            show card (B−{y}) < card B
              using card-less .
            show good-set (B−{y}) using cb-By .
            show linear-dependent (A∪(B−{y})) using True .
            show  h' ‘ {..<card A} = A using h'-A h'-def by auto
            show h' ‘ ({..<card A + card (B−{y})} − {..<card A}) = (B−{y}) using h'-B h'-def by simp
            show 0_V ∉ A ∪ (B−{y}) using By-subset-B less.prems(5) by auto
            show A ∩ (B−{y}) = {}  using By-subset-B less.prems(6) by auto
            show indexing (A ∪ (B−{y}), h') using indexing2 h'-def by simp
          qed
        qed
        thus ?thesis
            using descomposicion h'-def iterate-remove-ld.simps not-lin-indep by simp
      qed
    qed
  qed
```

qed
qed

Now we are in position to prove that every independent set can be extended to a basis. First we prove it for any non-empty set.

lemma *extend-not-empty-independent-set-to-a-basis*:
  assumes *li-A*: *linear-independent A*
  and *A-not-empty*: $A \neq \{\}$
  shows $\exists B.$ *basis* $B \wedge A \subseteq B$
proof −
  have *cb-A*: *good-set A* **using** *l-ind-good-set*[*OF li-A*] **.**
  def $C \equiv X - A$
  have *igualdad-conjuntos*: $A \cup X = A \cup C$ **using** *C-def* **by** *auto*
  have *finite-C*: *finite C* **using** *finite-X* **and** *cb-A C-def* **unfolding** *good-set-def* **by** *auto*
  have *disjuntos*: $A \cap C = \{\}$ **using** *C-def* **by** *auto*
  have $\exists h.$ *indexing* $(A \cup C, h) \wedge h$ ‘ $\{..<$*card* $A\} = A \wedge h$ ‘ $(\{..<$*card* $A +$ *card* $C\} - \{..<$*card* $A\}) = C$
    using *indexing-union* [*OF disjuntos - A-not-empty finite-C*]
    using *cb-A* **unfolding** *good-set-def* **by** *auto*
  from *this* **obtain** *h* **where** *indexing-AC-h*: *indexing* $((A \cup C), h)$ **and**
    surj-h-A: h ‘ $\{..<$*card* $A\} = A$ **and** *surj-h-B*: h ‘ $(\{..<$*card* $A +$ *card* $C\} - \{..<$*card* $A\}) = C$ **by** *auto*
  have *li-iterate*: *linear-independent*(*iterate-remove-ld* $(A \cup C)$ h)
  proof (*rule linear-independent-iterate-remove-ld*)
    show $A \cup C \subseteq$ *carrier V*
      using *l-ind-good-set*[*OF li-A*] *good-set-in-carrier C-def good-set-X*
      unfolding *good-set-def* **by** *auto*
    show $\mathbf{0}_V \notin A \cup C$
      using *li-A zero-not-in-linear-independent-set C-def* **by** *auto*
    show *indexing* $(A \cup C, h)$ **using** *indexing-AC-h* **.**
  qed
  have *span*(*iterate-remove-ld* $(A \cup C)$ h)=*span*$(A \cup C)$
  proof (*rule iterate-remove-ld-preserves-span*)
    show $A \cup C \subseteq$ *carrier V*
      using *l-ind-good-set*[*OF li-A*] *good-set-in-carrier C-def good-set-X*
      unfolding *good-set-def* **by** *auto*
    show *indexing* $(A \cup C, h)$ **using** *indexing-AC-h* **.**
    show $\mathbf{0}_V \notin A \cup C$ **using** *li-A zero-not-in-linear-independent-set C-def* **by** *auto*
  qed
  also have ...=*carrier V*
    using *span-union-basis-is-V cb-A igualdad-conjuntos*
    unfolding *good-set-def* **by** *force*
  finally have *span-iterate-remove-V*:
    span(*iterate-remove-ld* $(A \cup C)$ h)=*carrier V* **.**
  have *basis-iterate*: *basis* (*iterate-remove-ld* $(A \cup C)$ h)
  proof (*unfold basis-def*, *rule conjI3*)
    show *iterate-remove-ld* $(A \cup C)$ $h \subseteq$ *carrier V*
      using *igualdad-conjuntos l-ind-good-set li-iterate*

```
    unfolding good-set-def
    by presburger
    show linear-independent-ext (iterate-remove-ld (A ∪ C) h)
    unfolding linear-independent-ext-def
    using li-iterate good-set-finite l-ind-good-set C-def
    using independent-set-implies-independent-subset by blast
  show spanning-set-ext (iterate-remove-ld (A ∪ C) h)
    using l-ind-good-set li-iterate span-V-eq-spanning-set
    span-basis-implies-spanning-set[OF span-iterate-remove-V] spanning-imp-spanning-ext

    by presburger
 qed
 have A-in-iterate: A ⊆ (iterate-remove-ld (A∪C) h)
 proof (rule A-in-iterate-remove-ld)
  show indexing (A ∪ C, h) using indexing-AC-h .
  show C ⊆ carrier V using cb-A C-def good-set-X
    unfolding good-set-def by auto
  show h ` {..<card A} = A using surj-h-A .
  show  h ` ({..<card A + card C} − {..<card A}) = C using surj-h-B .
  show linear-independent A using li-A .
  show 0_V ∉ A ∪ C using li-A zero-not-in-linear-independent-set C-def by auto
  show A ∩ C = {} using disjuntos .
 qed
 show ?thesis using A-in-iterate and basis-iterate by auto
qed
```

And finally the theorem (case empty is trivial since we add all elements of our fixed basis X to it.

```
theorem extend-independent-set-to-a-basis:
 assumes li-A: linear-independent A
 shows ∃ B. basis B ∧ A ⊆ B
proof (cases A={})
 case True show ?thesis
  using basis-X True empty-subsetI by fast
 next
 case False show ?thesis
  using extend-not-empty-independent-set-to-a-basis[OF li-A False] .
qed
```

We have proved that any independent set can be extended to a basis, but in anywhere we have proved that there exists a basis (we have supposed it as a premisse in the case of finite dimensional vector spaces). The proof that every vector space has a basis is not made in Halmos: some additional results as Zorn's lemma, chains or well-ordering are required. See http://planetmath.org/encyclopedia/EveryVectorSpaceHasABasis.html for a detailed proof.

However, we can prove the existence of a basis in a particular case: when *carrier V* is finite.

To prove this result, we are going to apply the function *iterate-remove-ld* to *carrier* $V - \{\mathbf{0}_V\}$. This function requires that $\mathbf{0}_V$ doesn't belong to the set where we apply it, so we will not apply it to *carrier V*, but to *carrier* $V - \{\mathbf{0}_V\}$. This function will return us a linearly independent set which span is the same as the span of *carrier* $V - \{\mathbf{0}_V\}$. Proving that *span* (*carrier* $V - \{\mathbf{0}_V\}$) = *carrier V* we will obtain the result (because *carrier* $V - \{\mathbf{0}_V\}$ is a spanning set).

Let's see the proof. Firstly, we can see that the set V is a *spanning-set*. It is trivial.

```
lemma spanning-set-V:
  assumes finite-V: finite (carrier V)
  shows spanning-set (carrier V)
  using Un-absorb2 assms good-set-X good-set-def
    span-union-basis-is-V span-basis-implies-spanning-set
    subset-refl by metis
```

Thanks to that, the span of V is itself (trivially).

```
lemma span-V-is-V:
  assumes finite-V: finite (carrier V)
  shows span (carrier V) = carrier V
  using assms good-set-def spanning-set-V span-V-eq-spanning-set
    subset-refl by simp
```

Now we need to prove that *spanning-set* (*carrier* $V - \{\mathbf{0}_V\}$).

```
lemma spanning-set-V-minus-zero:
  assumes finite-V: finite (carrier V−{0_V})
  shows spanning-set (carrier V−{0_V})
proof (unfold spanning-set-def,auto)
  show good-set (carrier V − {0_V})
    using finite-V unfolding good-set-def by blast
  next
  fix x
  assume x-in-V: x ∈ carrier V
  let ?g=(λa. if a=x then 1_K else 0_K)
  show (∃f. f ∈ coefficients-function (carrier V)
    ∧ linear-combination f (carrier V − {0_V}) = x)
  proof (cases x=0_V)
    case True
    let ?f=(λa. 0_K) show ?thesis
    proof (rule exI[of - ?f])
      have cf-f: ?f ∈ coefficients-function (carrier V)
        unfolding coefficients-function-def by auto
      have lc: linear-combination ?f (carrier V − {0_V}) = x
      proof −
        have linear-combination ?f (carrier V − {0_V})
          = (⊕_V y::'c∈(carrier V − {0_V}). 0_K · y)
          unfolding linear-combination-def by simp
```

```
      also have ...=(⨁ V y::'c∈(carrier V − {0 V}). 0 V)
      proof (rule finsum-cong',auto)
        fix i
        assume i-in-V: i∈ carrier V
        show 0 · i = 0 V
          using zeroK-mult-V-is-zeroV[OF i-in-V] .
      qed
      also have ...=0 V using finsum-zero finite-V by auto
      finally show ?thesis using True by simp
    qed
    show ?f ∈ coefficients-function (carrier V)
      ∧ linear-combination ?f (carrier V − {0 V}) = x
      using cf-f and lc by auto
  qed
next
  case False show ?thesis
  proof (rule exI[of - ?g])
    have cf-g: ?g ∈ coefficients-function (carrier V)
      unfolding coefficients-function-def using x-in-V
      by simp
    have lc: linear-combination ?g (carrier V − {0 V}) = x
    proof −
      have x-not-zero: x≠0 V using False by simp
      have disjuntos: {x}∩ ((carrier V − {0 V})−{x})={} by auto
     have igualdad-conjuntos: carrier V − {0 V}=({x}∪ ((carrier V − {0 V})−{x}))
using x-in-V x-not-zero by auto
       hence linear-combination ?g (carrier V − {0 V})=linear-combination ?g
({x}∪ ((carrier V − {0 V})−{x}))
        by auto
      also have ...=linear-combination ?g {x} ⊕ V linear-combination ?g ((carrier
V − {0 V})−{x})
        unfolding linear-combination-def
      proof (rule finsum-Un-disjoint)
        show finite {x} by simp
        show finite (carrier V − {0 V} − {x}) using finite-V by auto
        show {x} ∩ (carrier V − {0 V} − {x}) = {} using disjuntos .
           show (λy. (if y = x then 1 else 0) · y) ∈ {x} → carrier V using
mult-closed[OF x-in-V -] by auto
         show (λy. (if y = x then 1 else 0) · y) ∈ carrier V − {0 V} − {x} →
carrier V
          unfolding Pi-def using zeroK-mult-V-is-zeroV by auto
      qed
      also have ...=1·x ⊕ V 0 V
      proof −
       have linear-combination ?g (carrier V − {0 V} − {x})=(⨁ V y::'c∈(carrier
V − {0 V}−{x}). 0 V)
        proof (unfold linear-combination-def,rule finsum-cong',auto)
          fix i
          assume i-in-V: i ∈ carrier V
```

show $\mathbf{0} \cdot i = \mathbf{0}_V$ **using** *zeroK-mult-V-is-zeroV*[*OF i-in-V*] **.**
qed
also have ...=$\mathbf{0}_V$ **using** *finsum-zero finite-V* **by** *auto*
finally show *?thesis* **using** *linear-combination-singleton*[*OF cf-g x-in-V*] **by** *auto*
qed
also have ...= x
using *V.add.r-one mult-1 x-in-V* **by** *presburger*
finally show *?thesis* **.**
qed
show $?g \in$ *coefficients-function* (*carrier V*) $\wedge$ *linear-combination* $?g$ (*carrier* $V - \{\mathbf{0}_V\}) = x$
using *cf-g* **and** *lc* **by** *auto*
qed
qed
qed

As a corollary we have that *span* (*carrier* $V - \{\mathbf{0}_V\}$) = *carrier V*

corollary *span-V-minus-zero-is-V*:
assumes *finite-V*: *finite* (*carrier* $V - \{\mathbf{0}_V\}$)
shows *span* (*carrier* $V - \{\mathbf{0}_V\}$)=*carrier V*
using *assms spanning-set-V-minus-zero*
spanning-set-implies-span-basis **by** *blast*

Finally, the theorem:

theorem *finite-V-implies-ex-basis*:
assumes *finite-V*: *finite* (*carrier V*)
shows $\exists B.$ *basis B*
proof −
have *finite-V-zero*: *finite* (*carrier* $V - \{\mathbf{0}_V\}$)
using *finite-V* **by** *simp*
from *finite-V-zero* **obtain** *f*
where *indexing*: *indexing* (*carrier* $V - \{\mathbf{0}_V\}$,*f*)
using *obtain-indexing* **by** *auto*
have *1*:*span* (*iterate-remove-ld* (*carrier* $V - \{\mathbf{0}_V\}$) *f*)=*carrier V*
using *iterate-remove-ld-preserves-span*[*OF - indexing -*]
and *span-V-minus-zero-is-V*[*OF finite-V-zero*]
by *auto*
have *2*:
linear-independent (*iterate-remove-ld* (*carrier* $V - \{\mathbf{0}_V\}$) *f*)
using *DiffE Diff-subset indexing insertI1*
linear-independent-iterate-remove-ld **by** *metis*
have *3*:*good-set* (*iterate-remove-ld* (*carrier* $V - \{\mathbf{0}_V\}$) *f*)
using *2 l-ind-good-set* **by** *fast*
have *basis* (*iterate-remove-ld* (*carrier* $V - \{\mathbf{0}_V\}$) *f*)
using *1* **and** *2* **and** *3* **using** *basis-def'* **by** *auto*
thus *?thesis*
by (*rule exI*[*of - iterate-remove-ld* (*carrier* $V - \{\mathbf{0}_V\}$) *f*])
qed

A similar result than *spanning-set-V-minus-zero* is the next. We will use this theorem in the future.

lemma *spanning-set-minus-zero*:
  assumes *finite-B*: *finite B*
  and *B-in-V*: $B \subseteq$ *carrier V*
  and *sg-B*: *spanning-set B*
  shows *spanning-set* $(B-\{\mathbf{0}_V\})$
proof (*unfold spanning-set-def*,*auto*)
  show *good-set* $(B - \{\mathbf{0}_V\})$
    unfolding *good-set-def* **using** *finite-B B-in-V* **by** *fast*
  show $\bigwedge x.\ x \in$ *carrier V* $\Longrightarrow \exists f.\ f \in$ *coefficients-function* (*carrier V*) $\wedge$ *linear-combination f* $(B - \{\mathbf{0}_V\}) = x$
  proof (*cases* $\mathbf{0}_V \in B$)
    case *False*
    fix *x*
    assume *x-in-V*: $x \in$ *carrier V*
    from *this* **obtain** *f* **where** *cf-f*: $f \in$ *coefficients-function* (*carrier V*) **and** *lc-B*: *linear-combination f B* $= x$
      using *sg-B* **unfolding** *spanning-set-def* **by** *blast*
    show $\exists f.\ f \in$ *coefficients-function* (*carrier V*) $\wedge$ *linear-combination f* $(B - \{\mathbf{0}_V\}) = x$
      using *Diff-idemp Diff-insert-absorb False cf-f lc-B* **by** *auto*
  next
    case *True*
    fix *x*
    assume *x-in-V*: $x \in$ *carrier V*
    from *this* **obtain** *f* **where** *cf-f*: $f \in$ *coefficients-function* (*carrier V*) **and** *lc-B*: *linear-combination f B* $= x$
      using *sg-B* **unfolding** *spanning-set-def* **by** *blast*
    show $\exists f.\ f \in$ *coefficients-function* (*carrier V*) $\wedge$ *linear-combination f* $(B - \{\mathbf{0}_V\}) = x$
    proof −
      have *lc-B0*: *linear-combination f* $(B - \{\mathbf{0}_V\}) = x$
      proof −
        have *igualdad-conjuntos*: (*insert* $\mathbf{0}_V$ $(B-\{\mathbf{0}_V\})$)=*B* **using** *True* **by** *fast*
        have *x=linear-combination f B* **using** *lc-B* **by** *simp*
        also have ...=*linear-combination f* (*insert* $\mathbf{0}_V$ $(B-\{\mathbf{0}_V\})$)
          using *arg-cong2*[*OF - igualdad-conjuntos*, *of f f linear-combination*] **by** *simp*
        also have ... = $(f\ \mathbf{0}_V) \cdot \mathbf{0}_V \oplus_V$ *linear-combination f* $(B-\{\mathbf{0}_V\})$
        proof (*rule linear-combination-insert*,*auto*)
          show *good-set* $(B - \{\mathbf{0}_V\})$ **using** *B-in-V finite-B* **unfolding** *good-set-def* **by** *fast*
          show $f \in$ *coefficients-function* (*carrier V*) **using** *cf-f* **.**
        qed
        also have ...=$\mathbf{0}_V \oplus_V$*linear-combination f* $(B-\{\mathbf{0}_V\})$
          using *scalar-mult-zeroV-is-zeroV*[*of f* $\mathbf{0}_V$] *cf-f zero-closed*
          unfolding *coefficients-function-def* **by** *force*
        also have ...=*linear-combination f* $(B-\{\mathbf{0}_V\})$

using *l-zero*[*OF linear-combination-closed*[*OF - cf-f*]] *B-in-V finite-B*
unfolding *good-set-def* **by** *blast*
finally show *?thesis* **by** *simp*
qed
thus *?thesis* **using** *cf-f* **by** *fast*
qed
qed
qed

Every finite or infinite vector space contains a *spanning-set-ext* (in particular, *carrier V* fullfills this condition):

lemma *spanning-set-ext-carrier-V*:
shows *spanning-set-ext* (*carrier V*)
proof (*unfold spanning-set-ext-def*, *auto*)
fix x
assume *x-in-V*: $x \in$ *carrier V*
show $\exists A$. *good-set A* $\wedge$ $A \subseteq$ *carrier V* $\wedge$ ($\exists f$. $f \in$ *coefficients-function* (*carrier V*) $\wedge$ *linear-combination f A* $= x$)
proof (*rule exI*[*of -* $\{x\}$], *rule conjI3*)
show *good-set* $\{x\}$ **unfolding** *good-set-def* **using** *x-in-V* **by** *fast*
show $\{x\} \subseteq$ *carrier V* **using** *x-in-V* **by** *fast*
show $\exists f$. $f \in$ *coefficients-function* (*carrier V*) $\wedge$ *linear-combination f* $\{x\} = x$
proof (*rule exI*[*of -* (λy. *if y=x then* **1** *else* **0**)], *rule conjI*)
show *cf*: (λy. *if* $y = x$ *then* **1** *else* **0**) $\in$ *coefficients-function* (*carrier V*)
unfolding *coefficients-function-def* **using** *x-in-V* **by** *simp*
show *linear-combination* (λy. *if* $y = x$ *then* **1** *else* **0**) $\{x\} = x$
proof $-$
have *linear-combination* (λy. *if* $y = x$ *then* **1** *else* **0**) $\{x\}$= (λy. *if* $y = x$ *then* **1** *else* **0**) $x \cdot x$
using *linear-combination-singleton*[*OF cf x-in-V*] **.**
also have ...= **1** $\cdot\, x$ **by** *simp*
also have ...= x **using** *mult-1*[*OF x-in-V*] **.**
finally show *?thesis* **.**
qed
qed
qed
qed

lemma *vector-space-contains-spanning-set-ext*:
shows $\exists A$. *spanning-set-ext A* $\wedge$ $A \subseteq$ *carrier V*
using *spanning-set-ext-carrier-V* **by** *blast*

end
end
theory *Dimension*
imports *Basis*
begin

10 Dimension

context *finite-dimensional-vector-space*
begin

Now we are going to prove that every basis of a finite vector space has the same cardinality than any other basis.

First of all, we are going to define a function that remove the first element of an iset. We will use the function *remove-iset*. Note that this redefinition is essential: we can not iterate *remove-iset* because is *remove-iset*: $::iset \times \mathbb{N} \rightarrow iset$

definition *remove-iset-0* :: *'e iset* => *'e iset*
  where *remove-iset-0 A* = *remove-iset A 0*

A property about this function and the empty set:

lemma *remove-iset-empty*:
  shows *fst* (*remove-iset-0* ({},*f*))={}
  unfolding *remove-iset-0-def remove-iset-def*
  by *simp*

Now the definition of the function by means of we are going to prove the theorem.

definition *swap-function* :: (*'c iset* × *'c iset*)
  => (*'c iset* × *'c iset*)
  where *swap-function A* = (*remove-iset-0* (*fst A*),
  if (((*snd*(*fst A*) *0*)) ∈ *fst*(*snd A*)) *then*
  insert-iset (*remove-iset* (*snd A*)
  (*obtain-position* ((*snd*(*fst A*) *0*)) (*snd A*))) (*snd*(*fst A*) *0*) *0*
  else
  remove-ld (*insert-iset* (*snd A*) ((*snd*(*fst A*) *0*)) *0*))

From this, we will prove some basic properties that *swap-function* satisfies.

The set of the first component of the result is finite:

lemma *finite-fst-swap-function*:
  assumes *indexing-A*: *indexing* (*A*,*f*)
  shows *finite* (*iset-to-set*(*fst*(*swap-function* ((*A*,*f*),(*B*,*g*)))))
proof −
  have *finite-A*: *finite A* **using** *indexing-finite*[*OF indexing-A*] **.**
  thus *?thesis* **unfolding** *swap-function-def remove-iset-0-def remove-iset-def* **by** *simp*
qed

The set of the first component of the result is in the carrier:

lemma *swap-function-fst-in-carrier*:
  assumes *A-in-V*: *A* ⊆ *carrier V*
  shows *iset-to-set*(*fst*(*swap-function* ((*A*,*f*),(*B*,*g*)))) ⊆ *carrier V*

using *A-in-V*
unfolding *swap-function-def remove-iset-0-def remove-iset-def* **by** *auto*

If the first set is not empty, then the set of the first component of the result is contained (strictly) in it.

lemma *fst-swap-function-subset-fst*:
assumes *indexing-A*: *indexing* (*A*,*f*)
and *A-not-empty*: *A*≠{} — INDISPENSABLE: IF NOT THE EMPTY CASE WILL NOT BE STRICT
shows *iset-to-set*(*fst*(*swap-function* ((*A*,*f*),(*B*,*g*)))) ⊂ *A*
proof −
have *0*∈{..<*card A*} **using** *A-not-empty* **and** *indexing-finite*[*OF indexing-A*]
by (*metis card-gt-0-iff lessThan-iff*)
hence *f 0* ∈ *A* **using** *indexing-A* **unfolding** *indexing-def bij-betw-def* **by** *auto*
thus *?thesis*
unfolding *swap-function-def remove-iset-0-def remove-iset-def*
by *auto*
qed

If we not demand that content be strict, then the result is trivial.

lemma *fst-swap-function-subseteq-fst*:
shows *iset-to-set*(*fst*(*swap-function* ((*A*,*f*),(*B*,*g*)))) ⊆ *A*
unfolding *swap-function-def remove-iset-0-def remove-iset-def*
by *auto*

We are goint to prove that the set of the second component of the result is a *good-set*. To prove it we will make use of ⟦*B* ⊆ *carrier V*; *A* ⊆ *carrier V*; *A* ≠ {}; *indexing* (*A*, *f*); *indexing* (*B*, *g*); *a* ∈ *B*⟧ ⟹ *good-set* (*fst* (*insert-iset* (*remove-iset* (*B*, *g*) (*obtain-position a* (*B*, *g*))) *a n*)).

lemma *swap-function-snd-good-set*:
assumes *B-in-V*: *B* ⊆ *carrier V*
and *A-in-V*: *A* ⊆ *carrier V*
and *A-not-empty*: *A*≠{}
and *indexing-A*: *indexing* (*A*,*f*)
and *indexing-B*: *indexing* (*B*,*g*)
shows *good-set* (*iset-to-set*(*snd*(*swap-function* ((*A*,*f*),(*B*,*g*)))))
proof (*unfold swap-function-def*, *simp*, *rule conjI*)
have *cb-A*: *good-set A* **using** *A-in-V indexing-finite*[*OF indexing-A*] **unfolding** *good-set-def* **by** *simp*
have *cb-B*: *good-set B* **using** *B-in-V indexing-finite*[*OF indexing-B*] **unfolding** *good-set-def* **by** *simp*
show *f 0* ∈ *B* ⟶ *good-set* (*fst* (*insert-iset* (*remove-iset* (*B*, *g*) (*obtain-position* (*f 0*) (*B*, *g*))) (*f 0*) *0*))
proof
assume *f0-in-B*: *f 0* ∈ *B*
show *good-set* (*fst* (*insert-iset* (*remove-iset* (*B*, *g*) (*obtain-position* (*f 0*) (*B*, *g*))) (*f 0*) *0*))
using *good-set-insert-remove*[*OF B-in-V A-in-V A-not-empty indexing-A indexing-B f0-in-B*] **.**

qed
show *f 0* $\notin$ *B* $\longrightarrow$ *good-set* (*fst* (*remove-ld* (*insert-iset* (*B*, *g*) (*f 0*) *0*)))
proof
assume *f0-notin-B*: *f 0* $\notin$ *B*
have *good-set* (*fst* (*remove-ld* ((*fst*(*insert-iset* (*B*, *g*) (*f 0*) *0*)), *snd* (*insert-iset* (*B*, *g*) (*f 0*) *0*))))
proof (*rule remove-ld-good-set*)
show *good-set* (*fst* (*insert-iset* (*B*, *g*) (*f 0*) *0*))
proof (*rule insert-iset-good-set*)
show *f 0* $\notin$ *B* **using** *f0-notin-B* **.**
show *indexing* (*B*, *g*) **using** *indexing-B* **.**
show *f 0* $\in$ *carrier V* **using** *f0-in-V*[*OF indexing-A A-in-V A-not-empty*] **.**
show *good-set B* **using** *cb-B* **.**
qed
show *indexing* (*fst* (*insert-iset* (*B*, *g*) (*f 0*) *0*), *snd* (*insert-iset* (*B*, *g*) (*f 0*) *0*))
using *insert-iset-indexing*[*OF indexing-B f0-notin-B -*] **by** *auto*
qed
thus *good-set* (*fst* (*remove-ld* (*insert-iset* (*B*, *g*) (*f 0*) *0*))) **by** *simp*
qed
qed

corollary *swap-function-snd-in-carrier*:
assumes *B-in-V*: *B* $\subseteq$ *carrier V*
and *A-in-V*: *A* $\subseteq$ *carrier V*
and *A-not-empty*: *A*$\neq${}
and *indexing-A*: *indexing* (*A*,*f*)
and *indexing-B*: *indexing* (*B*,*g*)
shows (*iset-to-set*(*snd*(*swap-function* ((*A*,*f*),(*B*,*g*))))) $\subseteq$ *carrier V*
using *swap-function-snd-good-set assms* **unfolding** *good-set-def* **by** *simp*

If the first set is independent, our function will preserve it.

lemma *fst-swap-function-preserves-li*:
assumes *li-A*: *linear-independent A*
shows *linear-independent* (*iset-to-set*(*fst*(*swap-function* ((*A*,*f*),(*B*,*g*)))))
unfolding *swap-function-def remove-iset-0-def* **and** *remove-iset-def*
using *independent-set-implies-independent-subset*[*of A*−{*f 0*},*OF - li-A*] **by** *auto*

If the first element of the iset (A,f) is in B, the function will preserve the second set (but it will have changed the indexation, putting that element in first position of B).

lemma *swap-function-preserves-B-if-fst-element-of-A-in-B*:
assumes *f0-in-B*: *f 0* $\in$*B*
and *indexing-A*: *indexing* (*A*,*f*)
and *indexing-B*: *indexing* (*B*,*g*)
shows *iset-to-set*(*snd*(*swap-function* ((*A*,*f*),(*B*,*g*))))=*B*
unfolding *swap-function-def* **using** *f0-in-B* **apply** *simp*
unfolding *insert-iset-def remove-iset-def obtain-position-def* **apply** *auto*
proof −

assume *gn-not-f0*: g (*THE n. g n = f 0* $\wedge$ *n* < *card B*) $\neq$ *f 0*
let *?P* = (λ*n. g n = f 0* $\wedge$ *n* < *card B*)
have *exK*: ($\exists$!*k. ?P k*) **using** *exists-n-and-is-unique-obtain-position*[*OF f0-in-B indexing-B*] **.**
have *ex-THE*: *?P* (*THE k. ?P k*)
using *theI′* [*OF exK*] **.**
def *n*$\equiv$(*THE k. ?P k*)
have *g n = f 0* **unfolding** *n-def*
by (*metis ex-THE*)
thus *False* **using** *gn-not-f0* **unfolding** *n-def* **by** *contradiction*
qed

This is an auxiliar lemma which says that if we insert an element into a spanning set, the result will be a linearly dependent set. We will need this result to assure the existence of the element to remove of the second set using the function *swap-function* through the theorem $[\![$*linear-dependent A*; $\mathbf{0}_V \notin A$; *indexing* (*A*, *f*)$]\!]$ $\Longrightarrow$ $\exists$ *y*$\in$*A*. $\exists$ *g k*. *g* $\in$ *coefficients-function* (*carrier V*) $\wedge$ *1* $\leq$ *k* $\wedge$ *k* < *card A* $\wedge$ *f k* = *y* $\wedge$ *y* = *linear-combination g* (*f* ‘ {*i*. *i* < *k*})

lemma *linear-dependent-insert-spanning-set*:
assumes *f0-notin-B*: *f 0* $\notin$ *B*
and *indexing-A*: *indexing* (*A*,*f*)
and *indexing-B*: *indexing* (*B*,*g*)
and *A-in-V*: *A* $\subseteq$ *carrier V*
and *B-in-V*: *B* $\subseteq$ *carrier V*
and *A-not-empty*: *A* $\neq$ {} — Essential to cardinality
and *sg-B*: *spanning-set B*
shows *linear-dependent* (*iset-to-set* (*insert-iset* (*B*,*g*) (*f 0*) *0*))
proof (*cases linear-dependent B*)
case *True* **show** *?thesis*
proof (*rule linear-dependent-subset-implies-linear-dependent-set*)
show *B* $\subseteq$ *iset-to-set* (*insert-iset* (*B*, *g*) (*f* (*0*)) *0*) **unfolding** *insert-iset-def iset-to-set-def* **by** *auto*
show *good-set* (*iset-to-set* (*insert-iset* (*B*, *g*) (*f* (*0*)) *0*))
unfolding *insert-iset-def iset-to-set-def good-set-def*
using *A-in-V B-in-V indexing-finite*[*OF indexing-A*] *indexing-finite*[*OF indexing-B*]

and *f0-in-V*[*OF indexing-A A-in-V A-not-empty*] **by** *simp*
show *linear-dependent B* **using** *True* **.**
qed
next
case *False* **show** *?thesis* **unfolding** *insert-iset-def* **apply** *simp*
proof (*rule lc1*)
show *li-B*: *linear-independent B*
using *not-dependent-implies-independent*[*OF - False*]
unfolding *good-set-def*
using *B-in-V indexing-finite*[*OF indexing-B*] **by** *simp*
show *f 0* $\in$ *carrier V* **using** *f0-in-V*[*OF indexing-A A-in-V A-not-empty*] **.**

show *f 0* $\notin$ *B* **using** *f0-notin-B* **.**
show $\exists$*fa. fa* $\in$ *coefficients-function* (*carrier V*) $\wedge$ *linear-combination fa B* = *f 0*
using *sg-B* **and** *f0-in-V*[*OF indexing-A A-in-V A-not-empty*]
unfolding *spanning-set-def* **by** *blast*
qed
qed

This result is similar to *linear-dependent-insert-spanning-set* but using sets directly, not isets.

lemma *spanning-set-insert*:
assumes *sg-B*: *spanning-set B*
and *finite-B*: *finite B*
and *B-in-V*: *B* $\subseteq$ *carrier V*
and *a-in-V*: *a* $\in$ *carrier V*
shows *spanning-set* (*insert a B*)
proof (*unfold spanning-set-def*, *auto*)
show *good-set* (*insert a B*) **using** *finite-B B-in-V a-in-V* **unfolding** *good-set-def* **by** *fast*
next
fix *x*
assume *x-in-V*: *x* $\in$ *carrier V*
from *this* **obtain** *f* **where** *cf-f*: *f* $\in$ *coefficients-function* (*carrier V*) **and** *lc-B*: *linear-combination f B* = *x*
using *sg-B* **unfolding** *spanning-set-def* **by** *blast*
show $\exists$*f. f* $\in$ *coefficients-function* (*carrier V*) $\wedge$ *linear-combination f* (*insert a B*) = *x*
proof −
def *g*≡(λ*x. if x* $\in$ *B then f x else* **0**)
have *linear-combination f B* = *linear-combination g* (*B* $\cup$ {*a*})
proof (*unfold g-def*, *rule eq-lc-when-out-of-set-is-zero*[*symmetric*])
show *good-set* {*a*} **using** *a-in-V* **unfolding** *good-set-def* **by** *fast*
show *good-set B* **using** *finite-B B-in-V* **unfolding** *good-set-def* **by** *blast*
show *f* $\in$ *coefficients-function* (*carrier V*) **using** *cf-f* **.**
qed
also have ...=*linear-combination g* (*insert a B*) **using** *arg-cong2* **by** *simp*
finally have *lc-Ba*: *x*=*linear-combination g* (*insert a B*) **using** *lc-B* **by** *simp*
have *g* $\in$ *coefficients-function* (*carrier V*) **unfolding** *g-def* **using** *coefficients-function-g-f-null*[*of f B*] *cf-f* **by** *auto*
thus *?thesis* **using** *lc-Ba* **by** *auto*
qed
qed

Our function will preserve that the second term is a *spanning-set*.

lemma *swap-function-preserves-sg*:
assumes *indexing-A*: *indexing* (*A*,*f*)
and *indexing-B*: *indexing* (*B*,*g*)
and *B-in-V*: *B*⊆*carrier V*
and *A-not-empty*: *A*≠{} — Essential to cardinality

```
  and li-A: linear-independent A
  and sg-B: spanning-set B
  and zero-notin-B: 0_V ∉ B
  shows spanning-set (iset-to-set(snd(swap-function ((A,f),(B,g)))))
proof (cases f 0 ∈ B)
  case True show ?thesis
    using swap-function-preserves-B-if-fst-element-of-A-in-B [OF True indexing-A
indexing-B] sg-B
    by simp
next
  case False thus ?thesis
  proof (unfold swap-function-def, simp)
    have A-in-V: A ⊆ carrier V
      by (metis good-set-def li-A linear-independent-def)
    show spanning-set (fst (remove-ld (insert-iset (B, g) (f 0) 0)))
    proof -
        have span (fst (remove-ld (insert-iset (B, g) (f 0) 0)))=span (iset-to-set
(insert-iset (B, g) (f 0) 0))
      proof -
        have 1: linear-dependent (fst (insert-iset (B, g) (f 0) 0))
        using linear-dependent-insert-spanning-set[OF False indexing-A indexing-B
A-in-V B-in-V A-not-empty sg-B]
          by simp
      have 2: 0_V ∉ fst (insert-iset (B, g) (f 0) 0) using f0-not-zero[OF indexing-A
li-A A-not-empty] zero-notin-B
          unfolding insert-iset-def by simp
        have 3:indexing (fst (insert-iset (B, g) (f 0) 0), snd (insert-iset (B, g) (f
0) 0))
          unfolding insert-iset-def apply simp
           using surjective-pairing
           and insert-iset-indexing[OF indexing-B False -] unfolding insert-iset-def
by auto
        show ?thesis
            using remove-ld-preserves-span[of fst (insert-iset (B, g) (f 0) 0) snd
(insert-iset (B, g) (f 0) 0) ]
          using surjective-pairing[of insert-iset (B, g) (f 0) 0] 1 2 3 by auto
      qed
      also have ...=carrier V
      proof (rule spanning-set-implies-span-basis)
        show spanning-set(iset-to-set (insert-iset (B, g) (f 0) 0))
          unfolding insert-iset-def
         using spanning-set-insert[OF sg-B  indexing-finite[OF indexing-B] B-in-V
           f0-in-V[OF indexing-A A-in-V A-not-empty]] by simp
      qed
      finally have span (fst (remove-ld (insert-iset (B, g) (f 0) 0)))=carrier V .
      thus ?thesis
      proof (rule span-basis-implies-spanning-set)
        show  good-set (fst (remove-ld (insert-iset (B, g) (f 0) 0)))
          by (metis A-in-V A-not-empty B-in-V f0-in-V finite.insertI finite-subset
```

fst-conv good-set-def indexing-A indexing-B indexing-finite insert-iset-def insert-subset iset-to-set-def remove-ld-monotone remove-ld-preserves-carrier)
qed
qed
qed
qed

swap-function preserves the cardinality of the second iset.

lemma *snd-swap-function-preserves-card*:
assumes *indexing-A*: *indexing* (*A*,*f*)
and *indexing-B*: *indexing* (*B*,*g*)
and *B-in-V*: $B \subseteq carrier\ V$
and *A-not-empty*: $A \neq \{\}$
and *li-A*: *linear-independent A*
and *sg-B*: *spanning-set B*
and *zero-notin-B*: $\mathbf{0}_V \notin B$
shows *card* (*iset-to-set* (*snd* (*swap-function* ((*A*,*f*),(*B*,*g*))))) = *card B*
proof (*cases f 0* ∈ *B*)
case *True* **thus** *?thesis*
using *swap-function-preserves-B-if-fst-element-of-A-in-B*[*OF True indexing-A indexing-B*] **by** *presburger*
next
case *False* **thus** *?thesis*
proof (*unfold swap-function-def*, *simp*)
have *A-in-V*: $A \subseteq carrier\ V$
by (*metis good-set-in-carrier l-ind-good-set li-A*)
have *eq-card*: *card* (*iset-to-set* (*insert-iset* (*B*, *g*) (*f 0*) *0*))= *card B* + *1*
using *insert-iset-increase-card*[*OF indexing-B False*] **.**
have *zero-notin-insert*: $\mathbf{0}_V \notin$ (*iset-to-set* (*insert-iset* (*B*, *g*) (*f 0*) *0*))
using *f0-not-zero*[*OF indexing-A li-A A-not-empty*] **and** *zero-notin-B*
unfolding *insert-iset-def* **by** *simp*
have *card* (*fst* (*remove-ld* (*insert-iset* (*B*, *g*) (*f 0*) *0*)))= *card* (*iset-to-set* (*insert-iset* (*B*, *g*) (*f 0*) *0*)) − *1*
using *surjective-pairing*
using *remove-ld-decr-card*[*OF linear-dependent-insert-spanning-set* [*OF False indexing-A indexing-B A-in-V B-in-V A-not-empty sg-B*] *zero-notin-insert*]
by (*metis eq-card False Suc-eq-plus1 diff-Suc-1 fst-conv indexing-B insert-iset-def insert-iset-indexing iset-to-set-def le0*)
also have ...= (*card B* + *1*) − *1* **using** *eq-card*
by *presburger*
finally show *card* (*fst* (*remove-ld* (*insert-iset* (*B*, *g*) (*f 0*) *0*))) = *card B* **by** *simp*
qed
qed

Next lemmas shows us how our function decreases the cardinality of the first term.

lemma *fst-swap-function-decr-card*:

assumes *indexing-A*: *indexing* (*A*,*f*)
shows *card* (*iset-to-set*(*fst*(*swap-function* ((*A*,*f*),(*B*,*g*))))) = *card A* − *1*
proof (*cases A*={})
case *True* **show** *?thesis* **unfolding** *swap-function-def remove-iset-0-def remove-iset-def*
using *True* **by** *auto*
next
case *False* **note** *A-not-empty*=*False*
show *?thesis*
proof (*unfold swap-function-def*, *unfold remove-iset-0-def*, *unfold remove-iset-def*, *simp*)
have *card A* > *0* **using** *A-not-empty indexing-finite*[*OF indexing-A*] *card-gt-0-iff*
by *metis*
hence *0*∈{..<*card A*} **by** *fast*
hence *f 0* ∈ *A* **using** *indexing-A* **unfolding** *indexing-def bij-betw-def* **by** *auto*
thus *card* (*A* − {*f 0*}) = *card A* − *Suc 0*
by (*metis One-nat-def* ‹*0* < *card A*› *card-Diff-singleton card-infinite less-zeroE*)
qed
qed

Now we are going to prove that exists an element of the second iset such that if we apply the *swap-function*, the second term will be able to be writen as the second set removing that element and adding the first element of the first set.

We will prove it by cases, first the case that B is not empty

lemma *swap-function-exists-y-in-B-not-empty*:
assumes *indexing-A*: *indexing* (*A*,*f*)
and *indexing-B*: *indexing* (*B*,*g*)
and *B-in-V*: *B* ⊆ *carrier V*
and *A-not-empty*: *A* ≠ {}
and *B-not-empty*: *B* ≠ {}
and *li-A*: *linear-independent A*
and *sg-B*: *spanning-set B*
and *zero-notin-B*: $\mathbf{0}_V$ ∉ *B*
shows ∃ *y*∈*B*. *iset-to-set* (*snd*(*swap-function* ((*A*,*f*),(*B*,*g*)))) = (*insert* (*f 0*) (*B*−{*y*}))
unfolding *swap-function-def*
proof (*simp*,*auto*)
show *f 0* ∈ *B* ⟹ ∃ *y*∈*B*. *fst* (*insert-iset* (*remove-iset* (*B*, *g*) (*obtain-position* (*f 0*) (*B*, *g*))) (*f 0*) *0*)
= *insert* (*f 0*) (*B* − {*y*})
using *swap-function-preserves-B-if-fst-element-of-A-in-B*[*OF - indexing-A indexing-B*]

unfolding *swap-function-def* **by** *auto*
show *f 0* ∉ *B* ⟹ ∃ *y*∈*B*. *fst* (*remove-ld* (*insert-iset* (*B*, *g*) (*f 0*) *0*)) = *insert* (*f 0*) (*B* − {*y*})
proof −
assume *f0-notin-B*: *f 0* ∉ *B*
— Usar el teorema: **thm** *descomposicion-remove-ld*

have *finite-B*: *finite B* **using** *indexing-finite*[*OF indexing-B*] **.**
have *A-in-V*: *A* ⊆ *carrier V*
by (*metis good-set-in-carrier l-ind-good-set li-A*)
have *insert-iset* (*B*, *g*) (*f 0*) *0*=(*fst* (*insert-iset* (*B*, *g*) (*f 0*) *0*,*snd* (*insert-iset* (*B*, *g*) (*f 0*) *0*)))
using *surjective-pairing* **by** *simp*
also have ...=({*f 0*} ∪ *B*,*snd* (*insert-iset* (*B*, *g*) (*f 0*) *0*)) **unfolding** *insert-iset-def* **by** *simp*
finally have *eq-pairing*: *insert-iset* (*B*, *g*) (*f 0*) *0* = ({*f 0*} ∪ *B*, *snd* (*insert-iset* (*B*, *g*) (*f 0*) *0*)) **.**
hence *fst* (*remove-ld* (*insert-iset* (*B*, *g*) (*f 0*) *0*))=*fst*(*remove-ld* ({*f 0*} ∪ *B*, *snd* (*insert-iset* (*B*, *g*) (*f 0*) *0*)))
by *simp*
hence *indexing-insert*: *indexing* ({*f 0*} ∪ *B*, *snd* (*insert-iset* (*B*, *g*) (*f 0*) *0*))
using *insert-iset-indexing* [*OF indexing-B f0-notin-B -*] **using** *eq-pairing* **by** *auto*
have ∃*y*. *fst* (*remove-ld* ({*f 0*} ∪ *B*, *snd* (*insert-iset* (*B*, *g*) (*f 0*) *0*))) = {*f 0*} ∪ (*B* − {*y*}) ∧ *y* ∈ *B* ∧
snd (*remove-ld* ({*f 0*} ∪ *B*, *snd* (*insert-iset* (*B*, *g*) (*f 0*) *0*))) ‘ ({..<*card* {*f 0*} + *card* (*B* − {*y*})} − {..<*card* {*f 0*}})
= *B* − {*y*} ∧ *snd* (*remove-ld* ({*f 0*} ∪ *B*, *snd* (*insert-iset* (*B*, *g*) (*f 0*) *0*))) ‘ {..<*card* {*f 0*}} = {*f 0*}
∧ *indexing* ({*f 0*} ∪ (*B* − {*y*}), *snd* (*remove-ld* ({*f 0*} ∪ *B*, *snd* (*insert-iset* (*B*, *g*) (*f 0*) *0*))))
proof (*rule descomposicion-remove-ld*)
show *indexing* ({*f 0*} ∪ *B*, *snd* (*insert-iset* (*B*, *g*) (*f 0*) *0*)) **using** *indexing-insert* **.**
show *B* ≠ {} **using** *B-not-empty* **.**
show *snd* (*insert-iset* (*B*, *g*) (*f 0*) *0*) ‘ {..<*card* {*f 0*}} = {*f 0*} **unfolding** *insert-iset-def indexing-ext-def* **by** *auto*
show *snd* (*insert-iset* (*B*, *g*) (*f 0*) *0*) ‘ ({..<*card* {*f 0*} + *card B*} − {..<*card* {*f 0*}}) = *B*
unfolding *insert-iset-def indexing-ext-def*
unfolding *image-def*
proof (*auto*)
show ⋀*xa*. ⟦*xa* < *Suc* (*card B*); *g* (*xa* − *Suc 0*) ∉ *B*⟧ ⟹ *xa* = *0*
proof (*rule FalseE*)
fix *xa*
assume *xa-l-cardB1*: *xa* < *Suc* (*card B*) **and** *gx-notin-B*: *g* (*xa* − *Suc 0*) ∉ *B*
have *surj*: *g* ‘{..<*card B*}=*B* **using** *indexing-B* **unfolding** *indexing-def bij-betw-def* **by** *simp*
have (*xa* − *Suc 0*) ∈ {..< *card B*} **using** *xa-l-cardB1*
by (*metis B-not-empty card-eq-0-iff diff-Suc-less finite-B gr0I lessThan-iff less-antisym less-imp-diff-less*)
hence *g* (*xa* − *Suc 0*) ∈ *B* **using** *surj* **by** *fast*
thus *False* **using** *gx-notin-B* **by** *contradiction*
qed
show ⋀*x*. *x* ∈ *B* ⟹ ∃*xa*∈{..<*Suc* (*card B*)} − {..<*Suc 0*}. *x* = *g* (*xa* −

Suc 0)
proof −
fix *x*
assume *x-in-B*: *x*∈*B*
have *surj*: *g* ‘{..<*card B*}=*B* **using** *indexing-B* **unfolding** *indexing-def bij-betw-def* **by** *simp*
hence ∃ *y*∈{..<*card B*}. *g y* = *x* **using** *x-in-B* **unfolding** *image-def* **by** *force*
from *this* **obtain** *y* **where** *y-l-card*: *y* ∈ {..<*card B*} **and** *gy-x*: *g y* = *x* **by** *fast*
show ∃ *xa*∈{..<*Suc* (*card B*)} − {..<*Suc 0*}. *x* = *g* (*xa* − *Suc 0*)
proof (*rule bexI*[*of - y* + *Suc 0*])
show *x* = *g* (*y* + *Suc 0* − *Suc 0*) **using** *gy-x* **by** *simp*
show *y* + *Suc 0* ∈ {..<*Suc* (*card B*)} − {..<*Suc 0*} **using** *y-l-card* **by** *simp*
qed
qed
qed
show *linear-independent* {*f 0*}
using *unipuntual-is-li*[*OF f0-in-V*[*OF indexing-A A-in-V A-not-empty*] *f0-not-zero*[*OF indexing-A li-A A-not-empty*]] **.**
show $\mathbf{0}_V$ ∉ {*f 0*} ∪ *B* **using** *f0-not-zero*[*OF indexing-A li-A A-not-empty*] **and** *zero-notin-B* **by** *simp*
show *linear-dependent* ({*f 0*} ∪ *B*)
proof −
have *eq-iset*: *iset-to-set* (*insert-iset* (*B*, *g*) (*f 0*) *0*)= {*f 0*} ∪ *B* **apply** *simp*
by (*metis fst-conv insert-iset-def iset-to-set-def*)
have *linear-dependent* (*iset-to-set* (*insert-iset* (*B*, *g*) (*f 0*) *0*))
using *linear-dependent-insert-spanning-set*[*OF f0-notin-B indexing-A indexing-B A-in-V B-in-V*
A-not-empty sg-B] **.**
thus *?thesis* **using** *eq-iset* **by** *simp*
qed
show {*f 0*} ∩ *B* = {} **using** *f0-notin-B* **by** *fast*
qed
from *this* **obtain** *y* **where**
eq-remove: *fst* (*remove-ld* ({*f 0*} ∪ *B*, *snd* (*insert-iset* (*B*, *g*) (*f 0*) *0*))) = {*f 0*} ∪ (*B* − {*y*}) **and** *y-in-B*: *y* ∈ *B*
by *metis*
show *?thesis* **using** *eq-remove* **and** *y-in-B eq-pairing* **by** *auto*
qed
qed

And now the case that B is empty. It is an inconsistent case: if *B* is empty and a spanning set, then the vector space is $\{\mathbf{0}_V\}$. A is not empty, so $A=\{\mathbf{0}_V\}$. However, we will have a contradiction: *A* will be dependent ($\{\mathbf{0}_V\}$ is dependent) and also independent (by hypothesis).

lemma *swap-function-exists-y-in-B-empty*:
assumes *indexing-A*: *indexing* (*A*,*f*)

and *A-not-empty*: $A \neq \{\}$
and *B-empty*: $B = \{\}$
and *li-A*: *linear-independent A*
and *sg-B*: *spanning-set B*
shows $\exists\, y \in B.$ *iset-to-set* (*snd*(*swap-function* ((*A*,*f*),(*B*,*g*))))=(*insert* (*f 0*) ($B - \{y\}$))
by (*metis A-not-empty B-empty Un-absorb1 Un-empty-right good-set-in-carrier empty-set-is-linearly-independent l-ind-good-set li-A sg-B span-V-eq-spanning-set*

span-empty subset-insert zero-not-in-linear-independent-set)

lemma *swap-function-exists-y-in-B*:
assumes *indexing-A*: *indexing* (*A*,*f*)
and *indexing-B*: *indexing* (*B*,*g*)
and *B-in-V*: $B \subseteq$ *carrier V*
and *A-not-empty*: $A \neq \{\}$
and *li-A*: *linear-independent A*
and *sg-B*: *spanning-set B*
and *zero-notin-B*: $\mathbf{0}_V \notin B$
shows $\exists\, y \in B.$ *iset-to-set* (*snd*(*swap-function* ((*A*,*f*),(*B*,*g*))))=(*insert* (*f 0*) ($B - \{y\}$))
proof (*cases* $B = \{\}$)
case *True* **show** *?thesis* **using** *swap-function-exists-y-in-B-empty*[*OF indexing-A A-not-empty True li-A sg-B*] **.**
next
case *False* **show** *?thesis*
using *swap-function-exists-y-in-B-not-empty*[*OF indexing-A indexing-B B-in-V*

A-not-empty False li-A sg-B zero-notin-B] **.**
qed

From this we can obtain a corollary: $\mathbf{0}_V$ is not in the second term of the result of applying *swap-function* to a *spanning-set*.

corollary *zero-notin-snd-swap-function*:
assumes *indexing-A*: *indexing* (*A*,*f*)
and *indexing-B*: *indexing* (*B*,*g*)
and *B-in-V*: $B \subseteq$ *carrier V*
and *A-not-empty*: $A \neq \{\}$
and *li-A*: *linear-independent A*
and *sg-B*: *spanning-set B*
and *zero-notin-B*: $\mathbf{0}_V \notin B$
shows $\mathbf{0}_V \notin$ *iset-to-set* (*snd*(*swap-function* ((*A*,*f*),(*B*,*g*))))
using *swap-function-exists-y-in-B*[*OF indexing-A indexing-B B-in-V A-not-empty li-A sg-B zero-notin-B*]
using *f0-not-zero*[*OF indexing-A li-A A-not-empty*] **using** *zero-notin-B* **by** *force*

The first term of the result of applying *swap-function* is an indexing.

lemma *fst-swap-function-indexing*:
assumes *indexing-A*: *indexing* (*A*,*f*)
and *A-in-V*: $A \subseteq$ *carrier V*
shows *indexing* (*fst*(*swap-function* ((*A*,*f*),(*B*,*g*))))

proof (*cases A*={})
 case *True* **show** *?thesis* **using** *True* **unfolding** *swap-function-def remove-iset-0-def remove-iset-def*
  using *indexing-empty* **by** *auto*
next
 case *False* **note** *A-not-empty*=*False*
 show *?thesis*
 proof (*unfold swap-function-def* ,*unfold remove-iset-0-def* , *simp*, *rule indexing-remove-iset*)
  have *finite-A*: *finite A***using** *indexing-finite*[*OF indexing-A*] **.**
  show *indexing* (*A*, *f*) **using** *indexing-A* **.**
  show *0* < *card A* **using** *finite-A A-not-empty* **by** *fastsimp*
 qed
qed

Similarly with the second term:

lemma *snd-swap-function-indexing*:
 assumes *indexing-A*: *indexing* (*A*,*f*)
 and *indexing-B*: *indexing* (*B*,*g*)
 and *A-in-V*: *A* ⊆ *carrier V*
 and *B-in-V*: *B* ⊆ *carrier V*
 and *A-not-empty*: *A*≠{}
 and *li-A*: *linear-independent A*
 and *sg-B*: *spanning-set B*
 and *zero-notin-B*: $\mathbf{0}_V \notin B$
 shows *indexing* (*snd*(*swap-function* ((*A*,*f*),(*B*,*g*))))
proof (*unfold swap-function-def* , *simp*, *rule conjI*)
 show *f 0* ∈ *B* ⟶ *indexing* (*insert-iset* (*remove-iset* (*B*, *g*) (*obtain-position* (*f 0*) (*B*, *g*))) (*f 0*) *0*)
 proof
  assume *f0-in-B*: *f 0* ∈ *B*
  show *indexing* (*insert-iset* (*remove-iset* (*B*, *g*) (*obtain-position* (*f 0*) (*B*, *g*))) (*f 0*) *0*)
  proof −
   have *indexing* (*insert-iset* (*fst* (*remove-iset* (*B*, *g*) (*obtain-position* (*f 0*) (*B*, *g*))),
    snd (*remove-iset* (*B*, *g*) (*obtain-position* (*f 0*) (*B*, *g*)))) (*f 0*) *0*)
   proof (*rule insert-iset-indexing*)
    have *indexing* (*remove-iset* (*B*, *g*) (*obtain-position* (*f 0*) (*B*, *g*)))
    proof (*rule indexing-remove-iset*)
     show *indexing* (*B*, *g*) **using** *indexing-B* **.**
    show *obtain-position* (*f 0*) (*B*, *g*) < *card B* **using** *obtain-position-less-card*[*OF f0-in-B indexing-B*] **.**
    qed
    thus *indexing*
     (*fst* (*remove-iset* (*B*, *g*) (*obtain-position* (*f 0*) (*B*, *g*))),
     snd (*remove-iset* (*B*, *g*) (*obtain-position* (*f 0*) (*B*, *g*)))) **by** *simp*
   show *f 0* ∉ *fst* (*remove-iset* (*B*, *g*) (*obtain-position* (*f 0*) (*B*, *g*))) **unfolding** *remove-iset-def*
     using *obtain-position-element*[*OF f0-in-B indexing-B*] **by** *simp*

show $0 \leq$ *card* (*fst* (*remove-iset* (B, g) (*obtain-position* $(f\ 0)$ (B, g)))) **by** *fast*
qed
thus *?thesis* **by** *simp*
qed
qed
show $f\ 0 \notin B \longrightarrow$ *indexing* (*remove-ld* (*insert-iset* (B, g) $(f\ 0)$ 0))
proof
assume *f0-notin-B*: $f\ 0 \notin B$
show *indexing* (*remove-ld* (*insert-iset* (B, g) $(f\ 0)$ 0))
proof −
have *indexing* (*remove-ld* ((*fst* (*insert-iset* (B, g) $(f\ 0)$ 0), *snd* (*insert-iset* (B, g) $(f\ 0)$ 0))))
proof (*rule indexing-remove-ld*)
show *linear-dependent* (*fst* (*insert-iset* (B, g) $(f\ 0)$ 0))
using *linear-dependent-insert-spanning-set*[*OF f0-notin-B indexing-A indexing-B A-in-V*
B-in-V A-not-empty sg-B] **by** *simp*
show *indexing* (*fst* (*insert-iset* (B, g) $(f\ 0)$ 0), *snd* (*insert-iset* (B, g) $(f\ 0)$ 0))
using *insert-iset-indexing*[*OF indexing-B f0-notin-B* -] **by** *auto*
show $\mathbf{0}_V \notin$ *fst* (*insert-iset* (B, g) $(f\ 0)$ 0)
by (*metis A-not-empty f0-not-zero fst-conv indexing-A insertE insert-iset-def iset-to-set-def li-A zero-notin-B*)
qed
thus *?thesis* **by** *simp*
qed
qed
qed

If the first argument is an empty iset, then *swap-function* will also return the empty set (in first component).

lemma *swap-function-empty*:
shows *iset-to-set*(*fst*(*swap-function* (({},*f*),(*B*,*g*))))={}
unfolding *swap-function-def*
unfolding *remove-iset-0-def*
unfolding *remove-iset-def* **by** *simp*

lemma *swap-function-empty2*:
assumes *A-empty*: *A*={}
shows *iset-to-set*(*fst*(*swap-function* ((*A*,*f*),(*B*,*g*))))={}
using *A-empty*
unfolding *swap-function-def*
unfolding *remove-iset-0-def*
unfolding *remove-iset-def* **by** *simp*

end

Up to now we have proved properties of *swap-function*. However, we want to iterate it a specific number of times (compose with itself several times). We need to implement the power of a function because (surprisingly) it is not in the library. We are interpreting the power of a function as a composition with itself.

We will have to be careful with the types: we can not iterate (compose) every function: a function can be composed with itself if the result and the arguments are of the same type (and the number of arguments is the same as the number of arguments of the result).

We can do the instantiation out of our context, since it is more general:

instantiation *fun* :: (*type*, *type*) *power*
begin

definition *one-fun* :: $'a => 'a$
  where *one-fun-def*: *one-fun* = *id*

definition *times-fun* :: $('a => 'a) => ('a => 'a) => 'a => 'a$
  where *times-fun f g* = $(\%x.\ f\ (g\ x))$

instance
proof
qed

end

Once we have finished the instatiation, we can prove some general properties about the power of a function.

For example: the power of the identity function is also the identity.

lemma *id-n*: **shows** *id* ^ *n* = *id*
  apply (*induct n*)
  apply *auto*
  unfolding *one-fun-def times-fun-def*
  unfolding *id-def*
  apply *auto*
  done

Any function power to zero is the identity.

lemma *power-zero-id*: *f*^*0*=*id*
  by (*metis one-fun-def power-0*)

A corollary of this lemma will be indispensable for the proofs by induction.

lemma *fun-power-suc*: **shows** *f*^(*Suc n*)= $f \circ$ (*f*^*n*)
  unfolding *power.simps* [*of f*]
  apply (*rule ext*)
  unfolding *times-fun-def* **by** *simp*

corollary *fun-power-suc-eq*:
  shows $(f\hat{}(Suc\ n))\ x = f\ ((f\hat{}n)\ x)$
  using *fun-power-suc* **by** (*metis id-o o-eq-id-dest*)

context *finite-dimensional-vector-space*
begin

Now we will begin with the proofs of properties that *swap-function* iterated several times satisfies. In general, we have proved a property in the case $n = 1$ and now we are going to generalize it for any n by induction.

Most properties are invariants of the *swap-function*, so we will have proved a property in case $n = 1$. To generalize it we will apply induction: we suppose that a property is true for f^n and we want to prove it for $f^{(Suc(n))}$. By induction hypothesis, f^n satisfies the property and thanks to *fun-power-suc-eq* we can write $f^{Suc\ n}\ x = f\ (f^n\ x)$. As we have the property proved in case $n = 1$, we will obtain the result generalized.

For example, we have proved *swap-function-empty*: *iset-to-set* (*fst* (*swap-function* $((\{\}, f), B, g))) = \{\}$ and now we will generalize it.

lemma *swap-function-power-empty*:
  shows *iset-to-set*(*fst*((*swap-function*ˆ*n*) $((\{\},f),(B,g))))=\{\}$
proof (*induct n*)
  show *iset-to-set* (*fst* ((*swap-function* ˆ *0*) $((\{\}, f), B, g))) = \{\}$ **using** *id-apply power-zero-id*
    by (*metis bot-nat-def fst-conv iset-to-set-def*)
  case *Suc*
  fix *n*
  assume *hip-induct*: *iset-to-set* (*fst* ((*swap-function* ˆ *n*) $((\{\}, f), B, g))) = \{\}$
  show *iset-to-set* (*fst* ((*swap-function* ˆ *Suc n*) $((\{\}, f), B, g))) = \{\}$
  proof −
    have *iset-to-set*(*fst*((*swap-function* ˆ *Suc n*) $((\{\}, f), B, g)))$
      =*iset-to-set*(*fst*((*swap-function* ((*swap-function* ˆ *n*) $((\{\}, f), B, g)))))$
      using *fun-power-suc-eq* **by** *metis*
    also have ...= *iset-to-set*
      (*fst* (*swap-function*
      ((*iset-to-set* (*fst* ((*swap-function* ˆ *n*) $((\{\}, f), B, g)))$,
      iset-to-index (*fst* ((*swap-function* ˆ *n*) $((\{\}, f), B, g)))$),
      iset-to-set (*snd* ((*swap-function* ˆ *n*) $((\{\}, f), B, g)))$,
      iset-to-index (*snd* ((*swap-function* ˆ *n*) $((\{\}, f), B, g))))))$ **by** *auto*
    also have ...=$\{\}$
      using *hip-induct swap-function-empty2*[*of iset-to-set* (*fst* ((*swap-function* ˆ *n*) $((\{\}, f), B, g)))$
        (*iset-to-index* (*fst* ((*swap-function* ˆ *n*) $((\{\}, f), B, g))))$
        (*iset-to-set* (*snd* ((*swap-function* ˆ *n*) $((\{\}, f), B, g))))$
        (*iset-to-index* (*snd* ((*swap-function* ˆ *n*) $((\{\}, f), B, g))))$] **by** *simp*
    finally show *?thesis* **.**

qed
qed

lemma *swap-function-power-empty2*:
assumes *A-empty*: *A*={}
shows *iset-to-set*(*fst*((*swap-function*^*n*) ((*A*,*f*),(*B*,*g*))))={}
by (*metis A-empty swap-function-power-empty*)

The generalized lemma for *swap-function-fst-in-carrier*.

lemma *swap-function-power-fst-in-carrier*:
assumes *A-in-V*: *A* ⊆ *carrier V*
shows *iset-to-set*(*fst*((*swap-function*^*n*) ((*A*,*f*),(*B*,*g*)))) ⊆ *carrier V*
proof (*induct n*)
show *iset-to-set* (*fst* ((*swap-function* ^ *0*) ((*A*, *f*), *B*, *g*))) ⊆ *carrier V*
using *power-zero-id id-apply A-in-V*
by (*metis iset-to-set-def fst-conv*)
case *Suc*
fix *n*
assume *hip-induct*: *iset-to-set* (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))) ⊆ *carrier V*
show *iset-to-set* (*fst* ((*swap-function* ^ *Suc n*) ((*A*, *f*), *B*, *g*))) ⊆ *carrier V*
proof −
have (*swap-function* ^ *Suc n*) ((*A*, *f*), *B*, *g*)
=*swap-function* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*)) **using** *fun-power-suc-eq* **by** *metis*
thus *?thesis*
using *swap-function-fst-in-carrier*[*of iset-to-set* (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*)))
iset-to-index (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*)))
iset-to-set (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*)))
iset-to-index (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*)))] *hip-induct* **by** *simp*
qed
qed

Iterating the function the independence (in first argument) is preserved.

lemma *fst-swap-function-power-preserves-li*:
assumes *li-A*: *linear-independent A*
shows *linear-independent* (*iset-to-set*(*fst*(((*swap-function*^(*n*))) ((*A*,*f*),(*B*,*g*)))))
proof (*induct n*)
case *0* **show** *linear-independent* (*iset-to-set* (*fst* ((*swap-function* ^ *0*) ((*A*, *f*), *B*, *g*))))
proof −
have *iset-to-set* (*fst* ((*swap-function* ^ *0*) ((*A*, *f*), *B*, *g*)))=
iset-to-set (*fst* ((*id*) ((*A*, *f*), *B*, *g*)))
using *power-zero-id* **by** *metis*
also have ...=*A* **using** *id-apply* **by** *simp*
finally show *?thesis* **using** *li-A* **by** *presburger*
qed
next

case *Suc*
fix *n*
assume *hip-induct*: *linear-independent* (*iset-to-set* (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))))
show *linear-independent* (*iset-to-set* (*fst* ((*swap-function* ^ *Suc n*) ((*A*, *f*), *B*, *g*))))
proof −
have (*swap-function* ^ *Suc n*) ((*A*, *f*), *B*, *g*)
= *swap-function* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*)) **using** *fun-power-suc-eq* **by** *metis*
thus *?thesis* **using** *fst-swap-function-preserves-li*[*OF hip-induct*,
of (*iset-to-index* (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))))
(*iset-to-set* (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))))
(*iset-to-index* (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))))] **by** *simp*
qed
qed

The first term is always an indexing. This is the generalization of *fst-swap-function-indexing*.

lemma *fst-swap-function-power-indexing*:
assumes *indexing-A*: *indexing* (*A*,*f*)
and *A-in-V*: *A* ⊆ *carrier V*
shows *indexing* (*fst*((*swap-function*^*n*) ((*A*,*f*),(*B*,*g*))))
proof (*induct n*)
show *indexing*(*fst* ((*swap-function* ^ *0*) ((*A*, *f*), *B*, *g*)))
using *power-zero-id id-apply indexing-A*
by (*metis fst-conv*)
case *Suc*
fix *n*
assume *hip-induct*: *indexing* (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*)))
show *indexing* (*fst* ((*swap-function* ^ *Suc n*) ((*A*, *f*), *B*, *g*)))
proof −
have (*swap-function* ^ *Suc n*) ((*A*, *f*), *B*, *g*)
=*swap-function* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*)) **using** *fun-power-suc-eq* **by** *metis*
thus *?thesis*
using *fst-swap-function-indexing*[*of iset-to-set* (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*)))
iset-to-index (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*)))
iset-to-set (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*)))
iset-to-index (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*)))]
swap-function-power-fst-in-carrier[*OF A-in-V*]
using *hip-induct* **by** *simp*
qed
qed

Now we can prove that if we compose n-times *swap-function*, the cardinality of the set of the first term will be decreased in n. Note that to use the induction hypothesis, we have to have proved previously *fst-swap-function-power-indexing* (and obviously also *fst-swap-function-decr-card*).

lemma *fst-swap-function-power-decr-card*:
  assumes *indexing-A*: *indexing* (*A*, *f*)
  and *A-in-V*: $A \subseteq$ *carrier V*
  shows *card* (*iset-to-set* (*fst* ((*swap-function*^*n*) ((*A*, *f*), *B*, *g*)))) = *card A* − *n*
proof (*induct n*)
  show *card* (*iset-to-set* (*fst* ((*swap-function* ^ *0*) ((*A*, *f*), *B*, *g*)))) = *card A* − *0*
using *power-zero-id id-apply*
    by (*metis fst-conv iset-to-set-def minus-nat.diff-0*)
  case *Suc*
  fix *n*
  assume *hip-induct*: *card* (*iset-to-set* (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))))
= *card A* − *n*
  show *card* (*iset-to-set* (*fst* ((*swap-function* ^ *Suc n*) ((*A*, *f*), *B*, *g*)))) = *card A*
− *Suc n*
  proof (*cases A*={})
    case *True* **show** *?thesis*
    proof −
     have *card* (*iset-to-set* (*fst* ((*swap-function* ^ *Suc n*) ((*A*, *f*), *B*, *g*))))=*card* {}
        using *swap-function-power-empty2*[*OF True*] **by** (*metis True card.empty*
card-eq-0-iff)
      thus *?thesis* **using** *True* **by** *simp*
    qed
  next
    case *False* **note** *A-not-empty=False*
    show *?thesis*
    proof −
      have (*swap-function* ^ *Suc n*) ((*A*, *f*), *B*, *g*)
       =*swap-function* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*)) **using** *fun-power-suc-eq*
by *metis*
      thus *?thesis*
        using *fst-swap-function-power-indexing*[*OF indexing-A A-in-V*]
         using *fst-swap-function-decr-card*[*of* (*iset-to-set* (*fst* ((*swap-function* ^ *n*)
((*A*, *f*), *B*, *g*))))
          (*iset-to-index* (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))))
          (*iset-to-set* (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))))
         (*iset-to-index* (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))))] **using** *hip-induct*
by *simp*
    qed
  qed
qed

The generalization of *finite-fst-swap-function*:

lemma *finite-fst-swap-function-power*:
  assumes *indexing-A*: *indexing* (*A*,*f*)
  and *A-in-V*: $A \subseteq$ *carrier V*
  shows *finite* (*iset-to-set*(*fst*((*swap-function*^*n*) ((*A*,*f*),(*B*,*g*)))))
proof (*induct n*)
  show *finite* (*iset-to-set* (*fst* ((*swap-function* ^ *0*) ((*A*, *f*), *B*, *g*))))
    using *power-zero-id id-apply indexing-finite*[*OF indexing-A*]

by (*metis fst-conv iset-to-set-def*)
case *Suc*
fix *n*
assume *hip-induct*: *finite* (*iset-to-set* (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))))
show *finite* (*iset-to-set* (*fst* ((*swap-function* ^ *Suc n*) ((*A*, *f*), *B*, *g*))))
proof −
have *indexing*: *indexing*
(*iset-to-set* (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))),
iset-to-index (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))))
using *fst-swap-function-power-indexing*[*OF indexing-A A-in-V*, *of n*] **by** *auto*
have *finite*: *finite* (*iset-to-set*
(*fst* (*swap-function*
((*iset-to-set* (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))),
iset-to-index (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*)))),
iset-to-set (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))),
iset-to-index (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*)))))))
using *finite-fst-swap-function*[*of iset-to-set* (*fst* ((*swap-function*^*n*) ((*A*, *f*), *B*, *g*)))
iset-to-index (*fst* ((*swap-function*^*n*) ((*A*, *f*), *B*, *g*)))
iset-to-set (*snd* ((*swap-function*^ *n*) ((*A*, *f*), *B*, *g*)))
iset-to-index (*snd* ((*swap-function*^*n*) ((*A*, *f*), *B*, *g*)))]
using *indexing*
using *hip-induct*
by *simp*
have *iset-to-set* (*fst* ((*swap-function* ^ *Suc n*) ((*A*, *f*), *B*, *g*)))
=*iset-to-set* (*fst* (*swap-function* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*)))) **using** *fun-power-suc-eq* **by** *metis*
also have ...=(*iset-to-set*
(*fst* (*swap-function*
((*iset-to-set* (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))),
iset-to-index (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*)))),
iset-to-set (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))),
iset-to-index (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))))))) **by** *auto*
finally have *eq*: *iset-to-set* (*fst* ((*swap-function* ^ *Suc n*) ((*A*, *f*), *B*, *g*))) =
iset-to-set(*fst* (*swap-function* ((*iset-to-set* (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))),
iset-to-index (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*)))),
iset-to-set (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))),
iset-to-index (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*)))))) **.**
thus *?thesis* **using** *finite* **by** *presburger*
qed
qed

If we iterate cardinality of A times the function, where A is the set of the first argument, then the first term of the result will be the empty set (we have removed card A elements in A).

corollary *swap-function-power-card-fst-empty*:
assumes *indexing-A*: *indexing* (*A*,*f*)
and *A-in-V*: *A* ⊆ *carrier V*

shows *iset-to-set*(*fst*((*swap-function*^(*card A*)) ((*A*,*f*),(*B*,*g*))))={}
proof −
have *finite*: *finite* (*iset-to-set*(*fst*((*swap-function*^(*card A*)) ((*A*,*f*),(*B*,*g*)))))
using *finite-fst-swap-function-power*[*OF indexing-A A-in-V*] **by** *simp*
have *card* (*iset-to-set* (*fst* ((*swap-function* ^ (*card A*)) ((*A*, *f*), *B*, *g*)))) = *card A* − *card A*
using *fst-swap-function-power-decr-card*[*OF indexing-A A-in-V*] **.**
also have ...= *0* **by** *fastsimp*
finally show *?thesis* **using** *finite*
by (*metis card-gt-0-iff le0 less-le-not-le*)
qed

And if we iterate a number of times less than card A, then the (first) result set will not be empty:

corollary *swap-function-power-fst-not-empty-if-n-l-cardA*:
assumes *indexing-A*: *indexing* (*A*,*f*)
and *A-in-V*: *A* ⊆ *carrier V*
and *n-l-card*: *n* < *card A*
shows *iset-to-set*(*fst*((*swap-function*^*n*) ((*A*,*f*),(*B*,*g*))))≠{}
proof −
have *card* (*iset-to-set* (*fst* ((*swap-function*^*n*) ((*A*, *f*), *B*, *g*)))) = *card A* − *n*
using *fst-swap-function-power-decr-card*[*OF indexing-A A-in-V*] **.**
thus *?thesis* **using** *n-l-card* **by** *auto*
qed

This is a very important property which shows us how is the result of applying the function *remove-iset-0* a specific number of times.

lemma *remove-iset-0-eq*:
assumes *i*: *indexing* (*A*,*f*)
and *k-l-card*: *k*<*card A*
shows (*remove-iset-0*^*k*) (*A*,*f*)=(*f*‘{*k*..<*card A*},λ*n*. *f*(*n*+*k*))
using *k-l-card*
proof (*induct k*)
case *0* **show** *?case* **unfolding** *power-zero-id* **unfolding** *id-apply* **using** *i* **unfolding** *indexing-def bij-betw-def*
by *fastsimp*
next
case (*Suc k*)
hence *k-l-card*: *k*<*card A* **and** *hyp*: (*remove-iset-0* ^ *k*) (*A*, *f*) = (*f* ‘ {*k*..<*card A*}, λ*n*. *f* (*n* + *k*)) **by** *auto*
show *?case*
proof −
have (*remove-iset-0* ^ *Suc k*) (*A*, *f*) =*remove-iset-0* ((*remove-iset-0* ^ *k*) (*A*, *f*)) **using** *fun-power-suc-eq* **by** *metis*
also have ...=*remove-iset-0* (*f* ‘ {*k*..<*card A*}, λ*n*. *f* (*n* + *k*)) **unfolding** *hyp*
..
also have ...=(*f* ‘ {*Suc k*..<*card A*}, λ*n*. *f* (*n* + *Suc k*)) **unfolding** *remove-iset-0-def remove-iset-def*
unfolding *snd-conv fst-conv*

proof (*rule*, *rule conjI*)
show ($\lambda n.$ *if* $n < 0$ *then* f $(n + k)$ *else* f (*Suc* $n + k$)) = ($\lambda n.$ f (n + *Suc* k)) **by** *simp*
next
show f ‘ $\{k..<card\ A\} - \{f\ (0 + k)\} = f$ ‘ $\{Suc\ k..<card\ A\}$
proof (*auto*)
fix *xa*
assume *fxa-notin*: $f\ xa \notin f$ ‘ $\{Suc\ k..<card\ A\}$ **and** *k-le-xa*: $k \leq xa$ **and** *xa-l-card*: $xa < card\ A$
have $xa < Suc\ k$ **using** *fxa-notin xa-l-card* **by** *fastsimp*
hence $k{=}xa$ **using** *k-le-xa* **by** *presburger*
thus $f\ xa = f\ k$ **by** *simp*
next
fix *xa*
assume *fxa-eq-fk*: $f\ xa = f\ k$ **and** *suc-k-le-xa*: $Suc\ k \leq xa$ **and** *xa-l-cardA*: $xa < card\ A$
have $f\ xa \neq f\ k$
proof (*rule inj-on-contraD*[*of* f $\{..<card\ A\}$])
show *inj-on* f $\{..<card\ A\}$ **using** *i* **unfolding** *indexing-def bij-betw-def* **by** *simp*
show $xa \neq k$ **using** *suc-k-le-xa* **by** *fastsimp*
show $xa \in \{..<card\ A\}$ **using** *xa-l-cardA* **by** *simp*
show $k \in \{..<card\ A\}$ **using** *suc-k-le-xa xa-l-cardA* **by** *simp*
qed
thus *False* **using** *fxa-eq-fk* **by** *contradiction*
qed
qed
finally show *?thesis* **.**
qed
qed

corollary *corollary-remove-iset-0-eq*:
assumes *i*: *indexing* (*A*,*f*)
and *n-l-card*: $n < card\ A$
shows *snd* ((*remove-iset-0*^*n*) (*A*,*f*)) $0 = f\ n$
using *remove-iset-0-eq*[*OF i n-l-card*] **by** *simp*

In the next lemma we prove some properties at same the time. We have done like that because in the induction case the properties need each others. We can not prove one separately: for example, to prove that $\mathbf{0}_V \notin$ *iset-to-set* (*snd* (*swap-function*$^{Suc\ n}$ ((A, f), B, g))) we would write that *swap-function*$^{Suc\ n}$ ((A, f), B, g) = *swap-function* (*swap-function*^{n} ((A, f), B, g)) and we would apply the theorem *zero-notin-snd-swap-function*:

$[\![$*indexing* (A, f); *indexing* (B, g); $B \subseteq$ *carrier* V; $A \neq \{\}$; *linear-independent* A; *spanning-set* B; $\mathbf{0}_V \notin B]\!] \Longrightarrow \mathbf{0}_V \notin$ *iset-to-set* (*snd* (*swap-function* ((A, f), B, g)))

However, to apply this theorem we need that *spanning-set* (*iset-to-set* (*snd* (*swap-function*^{n} ((*A*, *f*), *B*, *g*)))). To prove that we would need to use *swap-function-preserves-sg*:

$[\![$*indexing* (*A*, *f*); *indexing* (*B*, *g*); $B \subseteq$ *carrier* *V*; $A \neq \{\}$; *linear-independent* *A*; *spanning-set* *B*; $\mathbf{0}_V \notin B]\!] \Longrightarrow$ *spanning-set* (*iset-to-set* (*snd* (*swap-function* ((*A*, *f*), *B*, *g*))))

And a premise would be that $\mathbf{0}_V \notin$ *iset-to-set* (*snd* (*swap-function*^{n} ((*A*, *f*), *B*, *g*)))...but this is what we want to prove. Bringing all together in the same theorem we will have everything we need like induction hypothesis, so we can prove it. Next we will separate the properties.

```
lemma zeronotin-sg-carrier-indexing:
  assumes indexing-A: indexing (A,f)
  and indexing-B: indexing (B,g)
  and A-in-V: A ⊆ carrier V
  and B-in-V: B ⊆ carrier V
  and A-not-empty: A≠{}
  and li-A: linear-independent A
  and sg-B: spanning-set B
  and zero-notin-B: 0_V ∉ B
  and n-l-cardA: n < card A
  shows 0_V ∉ iset-to-set (snd ((swap-function^n) ((A, f), B, g)))
  ∧ spanning-set(iset-to-set(snd((swap-function^n)((A,f),(B,g)))))
  ∧ (iset-to-set(snd((swap-function^n) ((A,f),(B,g)))))
  ⊆ carrier V
  ∧ indexing (snd((swap-function^n) ((A,f),(B,g))))
  using n-l-cardA
proof (induct n)
  show 0_V ∉ iset-to-set (snd ((swap-function ^ 0) ((A, f), B, g))) ∧
    spanning-set (iset-to-set (snd ((swap-function ^ 0) ((A, f), B, g)))) ∧
    iset-to-set (snd ((swap-function^0) ((A, f), B, g))) ⊆ carrier V ∧
    indexing (snd ((swap-function ^ 0) ((A, f), B, g)))
  proof (rule conjI4)
     show 0_V ∉ iset-to-set (snd ((swap-function ^ 0) ((A, f), B, g))) using
power-zero-id id-apply
      by (metis fst-conv iset-to-set-def one-fun-def snd-conv zero-notin-B)
   show spanning-set (iset-to-set (snd ((swap-function ^ 0) ((A, f), B, g)))) using
power-zero-id id-apply
      by (metis fst-conv iset-to-set-def one-fun-def sg-B snd-conv)
    show iset-to-set (snd ((swap-function^0) ((A, f), B, g))) ⊆ carrier V using
power-zero-id id-apply
      by (metis fst-conv iset-to-set-def one-fun-def B-in-V snd-conv)
    show indexing (snd ((swap-function ^ 0) ((A, f), B, g)))
      using power-zero-id id-apply
      by (metis fst-conv iset-to-set-def one-fun-def indexing-B snd-conv)
  qed
  case Suc
  fix n
  assume hip-induct: n < card A ⟹ 0_V ∉ iset-to-set (snd ((swap-function ^ n)
```

$((A, f), B, g))) \wedge$
spanning-set (*iset-to-set* (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*)))) $\wedge$
iset-to-set (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))) $\subseteq$ *carrier V* $\wedge$
indexing (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))) **and** *Suc-l-card*: *Suc n* < *card A*
hence *n-l-card*: *n*<*card A*
by *linarith*
hence *hi-zero*: $\mathbf{0}_V \notin$ *iset-to-set* (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*)))
and *hi-sg*: *spanning-set* (*iset-to-set* (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))))
and *hi-carrier*: *iset-to-set* (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))) $\subseteq$ *carrier V*
and *hi-indexing*: *indexing* (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))) **using** *hip-induct* **by** *fast+*
show $\mathbf{0}_V \notin$ *iset-to-set* (*snd* ((*swap-function* ^ *Suc n*) ((*A*, *f*), *B*, *g*))) $\wedge$
spanning-set (*iset-to-set* (*snd* ((*swap-function* ^ *Suc n*) ((*A*, *f*), *B*, *g*)))) $\wedge$
iset-to-set (*snd* ((*swap-function* ^ *Suc n*) ((*A*, *f*), *B*, *g*))) $\subseteq$ *carrier V* $\wedge$
indexing (*snd* ((*swap-function* ^ *Suc n*) ((*A*, *f*), *B*, *g*)))
proof (*rule conjI4*)
have *eq-fi*: (*swap-function* ^ *Suc n*) ((*A*, *f*), *B*, *g*)
=*swap-function* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*)) **using** *fun-power-suc-eq* **by** *metis*
show $\mathbf{0}_V \notin$ *iset-to-set* (*snd* ((*swap-function* ^ *Suc n*) ((*A*, *f*), *B*, *g*)))
using *fst-swap-function-power-indexing*[*OF indexing-A A-in-V*, *of n B g*]
using *hi-indexing*
using *hi-carrier*
using *hi-sg*
using *hi-zero*
using *fst-swap-function-power-preserves-li*[*OF li-A*, *of n f B g*]
using *swap-function-power-fst-not-empty-if-n-l-cardA*[*OF indexing-A A-in-V n-l-card*, *of B g*]
using *zero-notin-snd-swap-function*[*of* (*iset-to-set* (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))))
(*iset-to-index* (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))))
(*iset-to-set* (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))))
(*iset-to-index* (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))))] **using** *eq-fi* **by** *simp*
show *spanning-set* (*iset-to-set* (*snd* ((*swap-function* ^ *Suc n*) ((*A*, *f*), *B*, *g*))))
using *fst-swap-function-power-indexing*[*OF indexing-A A-in-V*, *of n B g*]
using *hi-indexing*
using *hi-carrier*
using *hi-sg*
using *hi-zero*
using *fst-swap-function-power-preserves-li*[*OF li-A*, *of n f B g*]
using *swap-function-power-fst-not-empty-if-n-l-cardA*[*OF indexing-A A-in-V n-l-card*, *of B g*]
using *swap-function-preserves-sg*[*of* (*iset-to-set* (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))))
(*iset-to-index* (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))))
(*iset-to-set* (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))))

(*iset-to-index* (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))))] **using** *eq-fi* **by** *simp*
show *iset-to-set* (*snd* ((*swap-function* ^ *Suc n*) ((*A*, *f*), *B*, *g*))) ⊆ *carrier V*
using *fst-swap-function-power-indexing*[*OF indexing-A A-in-V*, *of n B g*]
using *hi-indexing*
using *hi-carrier*
using *swap-function-power-fst-in-carrier*[*OF A-in-V*, *of n f B g*]
using *swap-function-power-fst-not-empty-if-n-l-cardA*[*OF indexing-A A-in-V n-l-card*, *of B g*]
using *swap-function-snd-in-carrier*[*of* (*iset-to-set* (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))))
(*iset-to-set* (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))))
(*iset-to-index* (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))))
(*iset-to-index* (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))))] **using** *eq-fi* **by** *simp*
show *indexing* (*snd* ((*swap-function* ^ *Suc n*) ((*A*, *f*), *B*, *g*)))
using *fst-swap-function-power-indexing*[*OF indexing-A A-in-V*, *of n B g*]
using *hi-indexing*
using *swap-function-power-fst-in-carrier*[*OF A-in-V*, *of n f B g*]
using *hi-carrier*
using *swap-function-power-fst-not-empty-if-n-l-cardA*[*OF indexing-A A-in-V n-l-card*, *of B g*]
using *fst-swap-function-power-preserves-li*[*OF li-A*, *of n f B g*]
using *hi-sg*
using *hi-zero*
using *snd-swap-function-indexing*[*of* (*iset-to-set* (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))))
(*iset-to-index* (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))))
(*iset-to-set* (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))))
(*iset-to-index* (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))))] **using** *eq-fi* **by** *simp*
qed
qed

Now we can obtain the properties separately as corollaries.

corollary *zero-notin-snd-swap-function-power*:
assumes *indexing-A*: *indexing* (*A*,*f*)
and *indexing-B*: *indexing* (*B*,*g*)
and *A-in-V*: *A* ⊆ *carrier V*
and *B-in-V*: *B* ⊆ *carrier V*
and *A-not-empty*: *A*≠{}
and *li-A*: *linear-independent A*
and *sg-B*: *spanning-set B*
and *zero-notin-B*: $\mathbf{0}_V$ ∉ *B*
and *n-l-cardA*: *n*<*card A*
shows $\mathbf{0}_V$ ∉ *iset-to-set* (*snd* ((*swap-function*^*n*) ((*A*, *f*), *B*, *g*)))
using *zeronotin-sg-carrier-indexing assms* **by** *simp*

corollary *swap-function-power-preserves-sg*:
assumes *indexing-A*: *indexing* (A,f)
and *indexing-B*: *indexing* (B,g)
and *A-in-V*: $A \subseteq$ *carrier* V
and *B-in-V*: $B \subseteq$ *carrier* V
and *A-not-empty*: $A \neq \{\}$
and *li-A*: *linear-independent* A
and *sg-B*: *spanning-set* B
and *zero-notin-B*: $\mathbf{0}_V \notin B$
and *n-l-cardA*: $n <$ *card* A
shows *spanning-set* (*iset-to-set* (*snd* ((*swap-function*^n) ((A, f), B, g))))
using *zeronotin-sg-carrier-indexing assms* **by** *simp*

corollary *swap-function-power-snd-in-carrier*:
assumes *indexing-A*: *indexing* (A,f)
and *indexing-B*: *indexing* (B,g)
and *A-in-V*: $A \subseteq$ *carrier* V
and *B-in-V*: $B \subseteq$ *carrier* V
and *A-not-empty*: $A \neq \{\}$
and *li-A*: *linear-independent* A
and *sg-B*: *spanning-set* B
and *zero-notin-B*: $\mathbf{0}_V \notin B$
and *n-l-cardA*: $n <$ *card* A
shows *iset-to-set* (*snd* ((*swap-function*^n) ((A, f), B, g))) $\subseteq$ *carrier* V
using *zeronotin-sg-carrier-indexing assms* **by** *simp*

corollary *snd-swap-function-power-indexing*:
assumes *indexing-A*: *indexing* (A,f)
and *indexing-B*: *indexing* (B,g)
and *A-in-V*: $A \subseteq$ *carrier* V
and *B-in-V*: $B \subseteq$ *carrier* V
and *A-not-empty*: $A \neq \{\}$
and *li-A*: *linear-independent* A
and *sg-B*: *spanning-set* B
and *zero-notin-B*: $\mathbf{0}_V \notin B$
and *n-l-cardA*: $n <$ *card* A
shows *indexing* (*snd* ((*swap-function* ^ n) ((A, f), B, g)))
using *zeronotin-sg-carrier-indexing assms* **by** *simp*

Swap-function preserves the cardinality of the second iset.

lemma *snd-swap-function-power-preserves-card*:
assumes *indexing-A*: *indexing* (A, f)
and *indexing-B*: *indexing* (B, g)
and *A-in-V*: $A \subseteq$ *carrier* V
and *B-in-V*: $B \subseteq$ *carrier* V
and *A-not-empty*: $A \neq \{\}$
and *li-A*: *linear-independent* A
and *sg-B*: *spanning-set* B

and *zero-notin-B*: $\mathbf{0}_V \notin B$
and *n-l-card*: *n*<*card A*
shows *card* (*iset-to-set* (*snd* ((*swap-function*ˆ*n*) ((*A*, *f*), *B*, *g*)))) = *card B*
using *n-l-card*
proof (*induct n*)
show *card* (*iset-to-set* (*snd* ((*swap-function* ˆ *0*) ((*A*, *f*), *B*, *g*)))) = *card B*
using *id-apply power-zero-id*
by (*metis fst-conv iset-to-set-def snd-conv*)
case *Suc*
fix *n*
assume *hip*: *n* < *card A* ⟹ *card* (*iset-to-set* (*snd* ((*swap-function* ˆ *n*) ((*A*, *f*), *B*, *g*)))) = *card B*
and *suc-l-card*: *Suc n* < *card A*
hence *hip-induct*: *card* (*iset-to-set* (*snd* ((*swap-function* ˆ *n*) ((*A*, *f*), *B*, *g*)))) = *card B*
and *n-l-card*: *n*<*card A* **by** *fastsimp+*
show *card* (*iset-to-set* (*snd* ((*swap-function* ˆ *Suc n*) ((*A*, *f*), *B*, *g*)))) = *card B*
proof −
have (*swap-function* ˆ *Suc n*) ((*A*, *f*), *B*, *g*)
=*swap-function* ((*swap-function* ˆ *n*) ((*A*, *f*), *B*, *g*)) **using** *fun-power-suc-eq* **by** *metis*
thus *?thesis*
using *fst-swap-function-power-indexing*[*OF indexing-A A-in-V*]
using *snd-swap-function-power-indexing*[*OF indexing-A indexing-B A-in-V B-in-V A-not-empty li-A sg-B zero-notin-B n-l-card*]
using *swap-function-power-snd-in-carrier*[*OF indexing-A indexing-B A-in-V B-in-V A-not-empty li-A sg-B zero-notin-B n-l-card*]
using *fst-swap-function-power-preserves-li*[*OF li-A*, *of n f B g*]
using *swap-function-power-fst-not-empty-if-n-l-cardA*[*OF indexing-A A-in-V n-l-card*, *of B g*]
using *swap-function-power-preserves-sg*[*OF indexing-A indexing-B A-in-V B-in-V A-not-empty li-A sg-B zero-notin-B n-l-card*]
using *zero-notin-snd-swap-function-power*[*OF indexing-A indexing-B A-in-V B-in-V A-not-empty li-A sg-B zero-notin-B n-l-card*]
using *snd-swap-function-preserves-card* [*of iset-to-set* (*fst* ((*swap-function* ˆ *n*) ((*A*, *f*), *B*, *g*)))
iset-to-index (*fst* ((*swap-function* ˆ *n*) ((*A*, *f*), *B*, *g*)))
iset-to-set (*snd* ((*swap-function* ˆ *n*) ((*A*, *f*), *B*, *g*)))
iset-to-index (*snd* ((*swap-function* ˆ *n*) ((*A*, *f*), *B*, *g*)))] *hip-induct* **by** *simp*
qed
qed

The first term of *swap-function* iterated is the same than *remove-iset-0* iterated.

lemma *fst-swap-function-power-eq*:
fst ((*swap-function* ˆ *n*) ((*A*, *f*), *B*, *g*)) = (*remove-iset-0* ˆ *n*) (*A*, *f*)
proof (*induct n*)
case *0* **show** *?case* **using** *power-zero-id id-apply fst-conv* **by** *metis*
next

case (*Suc n*)
show *?case*
proof −
have *fst*((*swap-function* ^ *Suc n*) ((*A*, *f*), *B*, *g*))
=*fst*(*swap-function* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))) **using** *fun-power-suc-eq*
by *metis*
also have ...=*fst*(*swap-function* (*fst*((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*)),
snd((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*)))) **by** *simp*
also have ...=*fst*(*swap-function* ((*remove-iset-0* ^ *n*) (*A*, *f*), *snd*((*swap-function*
^ *n*) ((*A*, *f*), *B*, *g*))))
using *Suc.hyps* **by** *simp*
also have ...=*remove-iset-0* ((*remove-iset-0* ^ *n*) (*A*, *f*)) **unfolding** *swap-function-def*
fst-conv **..**
also have ...=(*remove-iset-0* ^ *Suc n*) (*A*, *f*) **using** *fun-power-suc-eq* **by** *metis*
finally show *?thesis* **.**
qed
qed

The first element of the result of the first term in the nth iteration is f(n).

lemma *snd-fst-swap-function-image-0*:
assumes *indexing-A*: *indexing* (*A*,*f*)
and *c*: *n* < *card A*
shows *snd* (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))) *0* = *f* (*n*)
proof −
have *fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*)) = (*remove-iset-0*^*n*) (*A*,*f*)
using *fst-swap-function-power-eq*[*of n A f B g*] **.**
hence *snd* (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))) *0* = *snd* ((*remove-iset-0*^*n*)
(*A*,*f*)) *0*
by *presburger*
also have ...= *f n* **using** *corollary-remove-iset-0-eq*[*OF indexing-A c*] **.**
finally show *?thesis* **.**
qed

If we compose n times the *swap-function*, the first term will be the first set minus the first n elements of it.

lemma *swap-function-fst-image-until-n*:
assumes *indexing-A*: *indexing* (*A*,*f*)
and *A-not-empty*: *A*≠{}
and *n-l-cardA*: *n*<*card A*
shows *iset-to-set* (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))) = *f* ' {*n*..<*card A*}
using *n-l-cardA*
proof (*induct n*)
show *iset-to-set* (*fst* ((*swap-function* ^ *0*) ((*A*, *f*), *B*, *g*))) = *f* ' {*0*..<*card A*}
using *id-apply power-zero-id*
using *indexing-A* **unfolding** *indexing-def bij-betw-def*
by (*metis atLeast0LessThan fst-conv iset-to-index-def iset-to-set-def snd-conv*)
case *Suc*
fix *n*

assume $n < card\ A \Longrightarrow iset\text{-}to\text{-}set\ (fst\ ((swap\text{-}function\ \hat{}\ n)\ ((A,\ f),\ B,\ g))) = f\ `\ \{n..<card\ A\}$
and $Suc\ n < card\ A$
hence *hip-induct*: $iset\text{-}to\text{-}set\ (fst\ ((swap\text{-}function\ \hat{}\ n)\ ((A,\ f),\ B,\ g))) = f\ `\ \{n..<card\ A\}$
and *n-l-card*: $n < card\ A$ **by** *auto*
show $iset\text{-}to\text{-}set\ (fst\ ((swap\text{-}function\ \hat{}\ Suc\ n)\ ((A, f), B, g))) = f\ `\ \{Suc\ n..<card\ A\}$
proof −
have *fn*: $snd\ (fst\ ((swap\text{-}function\ \hat{}\ n)\ ((A,\ f),\ B,\ g)))\ 0 = f\ n$
using *snd-fst-swap-function-image-0*[*OF indexing-A n-l-card*] **by** *simp*
have $iset\text{-}to\text{-}set\ (fst((swap\text{-}function\ \hat{}\ Suc\ n)\ ((A, f), B, g)))$
$=iset\text{-}to\text{-}set\ (fst(swap\text{-}function\ ((swap\text{-}function\ \hat{}\ n)\ ((A,\ f),\ B,\ g))))$
using *fun-power-suc-eq* **by** *metis*
also have $...=iset\text{-}to\text{-}set\ (fst(swap\text{-}function\ (fst((swap\text{-}function\ \hat{}\ n)\ ((A,\ f),\ B,\ g)),$
$snd((swap\text{-}function\ \hat{}\ n)\ ((A,\ f),\ B,\ g)))))$ **by** *simp*
also have $...=iset\text{-}to\text{-}set\ (fst(swap\text{-}function\ ((fst(fst((swap\text{-}function\ \hat{}\ n)\ ((A,\ f),\ B,\ g))),$
$snd(fst((swap\text{-}function\ \hat{}\ n)\ ((A,\ f),\ B,\ g)))),\ snd((swap\text{-}function\ \hat{}\ n)\ ((A,\ f),\ B,\ g)))))$ **by** *auto*
also have $...=iset\text{-}to\text{-}set\ (fst(swap\text{-}function\ ((f\ `\ \{n..<card\ A\},$
$snd(fst((swap\text{-}function\ \hat{}\ n)\ ((A,\ f),\ B,\ g)))),\ snd((swap\text{-}function\ \hat{}\ n)\ ((A,\ f),\ B,\ g)))))$
using *hip-induct* **by** *simp*
also have $...=f\ `\ \{n..<card\ A\} - \{f\ n\}$ **unfolding** *swap-function-def remove-iset-0-def remove-iset-def*
using *fn* **by** *force*
also have $...=f\ `\ \{n..<card\ A\} - f`\ \{n\}$ **by** *fast*
also have $...=f`(\{n..<card\ A\} - \{n\})$
proof (*rule inj-on-image-set-diff*[*symmetric*])
show $inj\text{-}on\ f\ \{..<card\ A\}$ **using** *indexing-A* **unfolding** *indexing-def bij-betw-def* **by** *simp*
show $\{n..<card\ A\} \subseteq \{..<card\ A\}$
by (*metis Un-upper2 atLeastLessThan-empty ivl-disj-un*(*8*)
lessThan-0 lessThan-subset-iff less-eq-nat.simps(*1*) *nat-le-linear*)
show $\{n\} \subseteq \{..<card\ A\}$
proof −
have $card\ A > 0$ **using** *A-not-empty indexing-finite*[*OF indexing-A*] **by** *fastsimp*
thus *?thesis* **using** *n-l-card* **by** *fast*
qed
qed
also have $...=f`\{Suc\ n..<card\ A\}$
by (*metis atLeastLessThan-singleton ivl-diff le-Suc-eq le-refl*)
finally show *?thesis* **.**
qed
qed

Now an auxiliar and ugly lemma which we will use to prove the swap theorem. It is very laborious and hard lemma, similar that *swap-function-exists-y-in-B* but much more precisse and difficult (over 400 lines). It represents properties that has the function during the process of iterating.

lemma *aux-swap-theorem1*:
assumes *indexing-A*: *indexing* (*A*,*f*) — In this set are the elements that we have not included in second term yet.
and *indexing-B*: *indexing* (*B*,*g*)
and *B-in-V*: $B \subseteq$ *carrier V*
and *A-not-empty*: $A \neq \{\}$
and *sg-B*: *spanning-set B*
and *zero-notin-B*: $\mathbf{0}_V \notin B$
and *li-Z*: *linear-independent Z* — Z is the first independent set, the set over we would apply our function the first time. A is the subset of Z where there are the elements of Z that we have not added to B yet. The elements that we have added to B are in C.
and *A-union-C*: $A \cup C = Z$ — Of course, the union of A and C is Z.
and *disjoint*: $A \cap C = \{\}$ — The sets are disjoints.
and *surj-g-C*: *g*‘{..<*card C*}= *C* — In first positions of B there are elements of Z that we have already included. This set will be independent, so when we apply *remove-ld* we will delete an element of (B-C)
shows $\exists\, y \in B$. *iset-to-set* (*snd*(*swap-function* ((*A*,*f*),(*B*,*g*))))
=(*insert* (*f 0*) ($B - \{y\}$))
$\wedge\ y \notin C$
$\wedge$ *iset-to-index* (*snd*(*swap-function* ((*A*,*f*),(*B*,*g*))))
‘ {..<*card* (*C*) + *1*} = $C \cup \{f\ 0\}$
proof (*unfold swap-function-def*, *auto*)
have *li-A*: *linear-independent A* **and** *li-C*: *linear-independent C*
using *independent-set-implies-independent-subset*[*OF - li-Z*] *A-union-C* **by** *auto*
show $f\ 0 \in B \Longrightarrow$
$\exists\, y \in B$. *fst* (*insert-iset* (*remove-iset* (*B*, *g*) (*obtain-position* (*f 0*) (*B*, *g*))) (*f 0*) *0*) = *insert* (*f 0*) ($B - \{y\}$) $\wedge$
$y \notin C \wedge$ *snd* (*insert-iset* (*remove-iset* (*B*, *g*) (*obtain-position* (*f 0*) (*B*, *g*))) (*f 0*) *0*) ‘ {..<*Suc* (*card C*)} =
insert (*f 0*) *C*
proof −
assume *f0-in-B*: $f\ 0 \in B$
show $\exists\, y \in B$. *fst* (*insert-iset* (*remove-iset* (*B*, *g*) (*obtain-position* (*f 0*) (*B*, *g*))) (*f 0*) *0*) = *insert* (*f 0*) ($B - \{y\}$) $\wedge$
$y \notin C \wedge$
snd (*insert-iset* (*remove-iset* (*B*, *g*) (*obtain-position* (*f 0*) (*B*, *g*))) (*f 0*) *0*) ‘ {..<*Suc* (*card C*)} =
insert (*f 0*) *C*
proof (*rule bexI*[*of - f 0*], *rule conjI*)
show *fst* (*insert-iset* (*remove-iset* (*B*, *g*) (*obtain-position* (*f 0*) (*B*, *g*))) (*f 0*) *0*) = *insert* (*f 0*) ($B - \{f\ 0\}$)
unfolding *insert-iset-def remove-iset-def*
using *f0-in-B indexing-B obtain-position-element* **by** *force*
show $f\ 0 \notin C \wedge$
snd (*insert-iset* (*remove-iset* (*B*, *g*) (*obtain-position* (*f 0*) (*B*, *g*))) (*f 0*) *0*)

‘ {..<*Suc* (*card* *C*)}
= *insert* (*f* *0*) *C*
proof (*rule conjI*)
have *0* ∈ {..< *card* *A*} **using** *A-not-empty*
by (*metis card-gt-0-iff indexing-A indexing-finite lessThan-iff*)
hence *f* *0* ∈ *A* **using** *indexing-A* **unfolding** *indexing-def bij-betw-def* **by** *auto*
thus *f0-notin-C*: *f* *0* ∉ *C* **using** *disjoint* **by** *fast*
show *snd* (*insert-iset* (*remove-iset* (*B*, *g*) (*obtain-position* (*f* *0*) (*B*, *g*))) (*f* *0*) *0*) ‘ {..<*Suc* (*card* *C*)}
= *insert* (*f* *0*) *C*
proof −
have *snd* (*insert-iset* (*remove-iset* (*B*, *g*) (*obtain-position* (*f* *0*) (*B*, *g*))) (*f* *0*) *0*) ‘ {..<*Suc* (*card* *C*)}=
snd (*insert-iset* (*remove-iset* (*B*, *g*) (*obtain-position* (*f* *0*) (*B*, *g*))) (*f* *0*) *0*) ‘ {*0*} ∪
snd (*insert-iset* (*remove-iset* (*B*, *g*) (*obtain-position* (*f* *0*) (*B*, *g*))) (*f* *0*) *0*) ‘ {*0*<..<*Suc* (*card* *C*)}
proof −
have {..<*Suc* (*card* *C*)}={*0*}∪{*0*<..<*Suc* (*card* *C*)} **by** *fastsimp*
thus *?thesis* **by** *blast*
qed
also have ...=*snd* (*insert-iset* (*remove-iset* (*B*, *g*) (*obtain-position* (*f* *0*) (*B*, *g*))) (*f* *0*) *0*) ‘ {*0*} ∪ *C*
proof −
have *snd* (*insert-iset* (*remove-iset* (*B*, *g*) (*obtain-position* (*f* *0*) (*B*, *g*))) (*f* *0*) *0*) ‘ {*0*<..<*Suc* (*card* *C*)} = *C*
proof −
have *cardC-le-obt-pos*: *card* *C* ≤ *obtain-position* (*f* *0*) (*B*, *g*)
by (*metis f0-in-B f0-notin-C indexing-B insert-image insert-subset leI lessThan-iff mem-def obtain-position-element subset-refl surj-g-C*)
have *image-C*: *snd* (*remove-iset* (*B*, *g*) (*obtain-position* (*f* *0*) (*B*, *g*))) ‘ {..<(*card* *C*)} = *C*
unfolding *remove-iset-def*
proof (*auto*)
show $\bigwedge k.$ ⟦ *k* < *card* *C*; *k* < *obtain-position* (*f* *0*) (*B*, *g*)⟧ ⟹ *g* *k* ∈ *C*
by (*metis imageI lessThan-iff surj-g-C*)
show $\bigwedge k.$ ⟦*k* < *card* *C*; ¬ *k* < *obtain-position* (*f* *0*) (*B*, *g*)⟧ ⟹ *g* (*Suc* *k*) ∈ *C*
using *cardC-le-obt-pos* **by** *simp*
show $\bigwedge x.$ ⟦*x* ∈ *C*; *x* ∉ (λ*k*. *g* (*Suc* *k*)) ‘ ({..<*card* *C*} ∩ {*k*. ¬ *k* < *obtain-position* (*f* *0*) (*B*, *g*)})⟧
⟹ *x* ∈ *g* ‘ ({..<*card* *C*} ∩ {*k*. *k* < *obtain-position* (*f* *0*) (*B*, *g*)})
using *surj-g-C cardC-le-obt-pos* **by** *force*
qed
show *?thesis* **unfolding** *insert-iset-def indexing-ext-def* **using** *image-C*
proof (*auto*)
fix *x*

assume *x-in-C*: $x \in C$
show $x \in (\lambda k.$ *snd* (*remove-iset* (B, g) (*obtain-position* $(f\ 0)$ $(B, g)))$ $(k - Suc\ 0))$ ‘
($\{0<..<Suc\ (card\ C)\} \cap Collect\ (op < 0)$)
proof (*unfold image-def*, *auto*)
have $\exists xa \in \{..<card\ C\}.\ x = snd$ (*remove-iset* (B, g) (*obtain-position* $(f\ 0)$ $(B, g)))$ *xa*
using *image-C x-in-C* **unfolding** *image-def* **by** *auto*
from *this* **obtain** *xa* **where** *xa-in-l-card*: $xa \in \{..<card\ C\}$
and *x-eq*: $x = snd$ (*remove-iset* (B, g) (*obtain-position* $(f\ 0)$ $(B, g)))$ *xa* **by** *blast*
show *ex-xa*: $\exists xa \in \{0<..<Suc\ (card\ C)\} \cap Collect\ (op < 0).$
$x = snd$ (*remove-iset* (B, g) (*obtain-position* $(f\ 0)$ $(B, g)))$ $(xa - Suc\ 0)$
proof (*rule bexI*[*of* - $xa + 1$])
show $x = snd$ (*remove-iset* (B, g) (*obtain-position* $(f\ 0)$ $(B, g)))$ $(xa + 1 - Suc\ 0)$
using *x-eq* **by** *auto*
show $xa + 1 \in \{0<..<Suc\ (card\ C)\} \cap Collect\ (op < 0)$
using *xa-in-l-card* **by** *auto*
qed
qed
qed
qed
thus *?thesis* **by** *presburger*
qed
also have $...=\{f\ 0\} \cup C$ **unfolding** *insert-iset-def indexing-ext-def* **by** *fastsimp*
also have $...=$*insert* $(f\ 0)$ C **by** *simp*
finally show *?thesis* **.**
qed
qed
show $f\ 0 \in B$ **using** *f0-in-B* **.**
qed
qed
show $f\ 0 \notin B \Longrightarrow$
$\exists y \in B.$ *fst* (*remove-ld* (*insert-iset* (B, g) $(f\ 0)$ $0)) =$ *insert* $(f\ 0)$ $(B - \{y\}) \wedge$
$y \notin C \wedge$ *snd* (*remove-ld* (*insert-iset* (B, g) $(f\ 0)$ $0))$ ‘ $\{..<Suc\ (card\ C)\} =$ *insert* $(f\ 0)$ C
proof −
assume *f0-notin-B*: $f\ 0 \notin B$
show $\exists y \in B.$ *fst* (*remove-ld* (*insert-iset* (B, g) $(f\ 0)$ $0)) =$ *insert* $(f\ 0)$ $(B - \{y\}) \wedge$
$y \notin C \wedge$ *snd* (*remove-ld* (*insert-iset* (B, g) $(f\ 0)$ $0))$ ‘ $\{..<Suc\ (card\ C)\} =$ *insert* $(f\ 0)$ C
proof −
have *A-in-V*: $A \subseteq$ *carrier* V **using** *l-ind-good-set*[*OF li-Z*] *A-union-C* **unfolding** *good-set-def* **by** *fast*
def $P' \equiv$*iset-to-set* (*insert-iset* (B, g) $(f\ 0)$ 0)

def *h*′≡*iset-to-index*(*insert-iset* (*B*, *g*) (*f 0*) *0*)
have *ld-P*′:*linear-dependent P*′
proof (*unfold P*′*-def*, *rule linear-dependent-insert-spanning-set*)
show *f 0* ∉ *B* **using** *f0-notin-B* **.**
show *indexing* (*A*, *f*) **using** *indexing-A* **.**
show *indexing* (*B*, *g*) **using** *indexing-B* **.**
show *A* ⊆ *carrier V* **using** *A-in-V* **.**
show *B* ⊆ *carrier V* **using** *B-in-V* **.**
show *A* ≠ {} **using** *A-not-empty* **.**
show *spanning-set B* **using** *sg-B* **.**
qed
have *indexing*: *indexing* (*P*′,*h*′)
unfolding *P*′*-def h*′*-def* **using** *insert-iset-indexing*[*OF indexing-B f0-notin-B* -] **by** *simp*
have *zero-not-in*: $\mathbf{0}_V$ ∉ *P*′
using *P*′*-def zero-notin-B f0-not-zero*[*OF indexing-A li-A A-not-empty*]
unfolding *insert-iset-def* **by** *simp*
let *?P* = (λ*k*. ∃ *y*∈*P*′. ∃ *g*. *g*∈ *coefficients-function* (*carrier V*) ∧ *1* ≤ *k* ∧
k < *card P*′ ∧ *h*′ *k* = *y* ∧ *y* = *linear-combination g* (*h*′ ‘ {*i*. *i* < *k*}))
have *exK*: (∃ *k*. *?P k*)
using *linear-dependent-set-sorted-contains-linear-combination*[*OF ld-P*′ *zero-not-in indexing*] **by** *auto*
have *ex-LEAST*: *?P* (*LEAST k*. *?P k*)
using *LeastI-ex* [*OF exK*] **.**
let *?k* = (*LEAST k*. *?P k*)
have ∃ *y*∈*P*′. ∃ *g*. *g* ∈ *coefficients-function* (*carrier V*) ∧ *1* ≤ *?k* ∧
?k < *card P*′ ∧ *h*′ *?k* = *y* ∧ *y* = *linear-combination g* (*h*′ ‘ {*i*. *i* < *?k*})
using *ex-LEAST* **by** *simp*
then obtain *y s*
where *one-le-k*: *1* ≤ *?k* **and** *k-l-card*: *?k* < *card P*′ **and** *h*′*-k-eq-y*: *h*′ *?k* = *y*
and *cf-s*: *s* ∈ *coefficients-function* (*carrier V*) **and**
combinacion-anteriores: *y* = *linear-combination s* (*h*′ ‘ {*i*. *i* < *?k*}) **by** *blast*
have *rem-eq*: *fst* (*remove-ld* (*P*′, *h*′)) = *P*′ − {*y*} **and** *y-in-P*′: *y* ∈ *P*′
using *indexing-equiv-img* [*OF indexing*] *one-le-k k-l-card h*′*-k-eq-y*
unfolding *Pi-def* **unfolding** *remove-ld-def*′ **by** *auto*
show *?thesis*
proof (*rule bexI*[*of - y*], *rule conjI*)
show *y-in-B*: *y*∈*B* — WE HAVE TO PROVE THAT y is different to f 0
using *y-in-P*′ **unfolding** *P*′*-def* **unfolding** *insert-iset-def*
proof (*simp*)
assume *y-f0-or-in-B*: *y*=*f 0* ∨ *y* ∈ *B*
show *y* ∈ *B*
proof (*cases y*= *f 0*)
case *False* **thus** *?thesis* **using** *y-f0-or-in-B* **by** *fast*
next
case *True*
have *inj-on-h*′: *inj-on h*′ {..<*card P*′} **using** *indexing* **unfolding** *indexing-def bij-betw-def* **by** *simp*

have h' *0* = *f 0* **using** h'*-def* **unfolding** *insert-iset-def indexing-ext-def* **by** *simp*
hence h' *0* = *y* **using** *True* **by** *simp*
hence h' *0*= h' *?k* **using** h'*-k-eq-y* **by** *simp*
hence *?k=0*
using *inj-on-eq-iff* [*OF inj-on-h'*] **using** *k-l-card* **by** *simp*
thus *?thesis* **using** *one-le-k* **by** *presburger* — CONTRADICTION, WE HAVE k=0 and k greater or equal to 1
qed
qed
show *fst* (*remove-ld* (*insert-iset* (*B*, *g*) (*f 0*) *0*)) = *insert* (*f 0*) (*B* − {*y*})
proof −
have *fst* (*remove-ld* (*insert-iset* (*B*, *g*) (*f 0*) *0*))
=*fst* (*remove-ld* (*fst*(*insert-iset* (*B*, *g*) (*f 0*) *0*),*snd*(*insert-iset* (*B*, *g*) (*f 0*) *0*))) **by** *simp*
also have ...=(*insert* (*f 0*) *B*) − {*y*} **using** *rem-eq* **unfolding** P'*-def* h'*-def insert-iset-def* **by** *simp*
also have ...=*insert* (*f 0*) (*B* − {*y*}) **using** *f0-notin-B y-in-B* **by** *blast*
finally show *?thesis* **.**
qed
show *y* ∉ *C* ∧ *snd* (*remove-ld* (*insert-iset* (*B*, *g*) (*f 0*) *0*)) ‘ {..<*Suc* (*card C*)} = *insert* (*f 0*) *C*
proof (*rule conjI*)
show *y* ∉ *C*
proof (*cases y*∉*C*)
case *True* **thus** *?thesis* **.**
next
case *False* **note** *y-in-C=False* **show** *?thesis*
proof −
have *image-h-C*: h'‘{*0*<..< *Suc* (*card C*)} = *C*
proof (*unfold image-def*, *unfold* h'*-def*,*unfold insert-iset-def*
, *unfold indexing-ext-def*, *auto*)
fix *xa*
assume *0* < *xa* **and** *xa* < *Suc* (*card C*)
thus *g* (*xa* − *Suc 0*) ∈ *C* **using** *surj-g-C* **by** *auto*
next
fix *x*
assume *x-in-C*: *x* ∈ *C*
have ∃ *xa*∈{..<(*card C*)}. *x* = *g* (*xa*) **using** *surj-g-C x-in-C* **unfolding** *image-def* **by** *auto*
from *this* **obtain** *xa* **where** *g-xa-x*: *x* = *g* (*xa*) **and** *xa-in-set*: *xa*∈{..<(*card C*)} **by** *auto*
show ∃ *xb*∈{*0*<..<*Suc* (*card C*)}. *x* = *g* (*xb* − *Suc 0*) — Sera xb=xa+1
using *g-xa-x xa-in-set* **by** *force*
qed
have *image-h-BC*: h'‘{*i*. *Suc* (*card C*)≤*i* ∧ *i*<(*card* P')}=*B*−*C*
proof (*unfold image-def*, *unfold* h'*-def*,*unfold insert-iset-def*
, *unfold indexing-ext-def*, *auto*)

fix *xa*
assume $xa < card\ P'$
hence *xa-l-suc-cardB*: $xa < Suc\ (card\ B)$ **unfolding** *P′-def*
by (*metis P′-def card-insert-if f0-notin-B fst-conv indexing-B indexing-finite insert-iset-def iset-to-set-def*)
have $card\ B > 0$ **using** *y-in-B indexing-finite*[*OF indexing-B*]
by (*metis card-gt-0-iff equals0D*)
thus $g\ (xa - Suc\ 0) \in B$
using *indexing-B* **unfolding** *indexing-def bij-betw-def* **using** *xa-l-suc-cardB* **by** *auto*
next
fix *x*
assume *x-in-B*: $x \in B$ **and** *x-notin-C*: $x \notin C$
show $\exists\, xa \geq Suc\ (card\ C).\ xa < card\ P' \wedge x = g\ (xa - Suc\ 0)$ — El testigo es a+1
proof −
from *x-in-B* **obtain** *a* **where** *x-eq-ga*: $x=g\ a$ **and** *a-l-cardB*:$a < card\ B$
using *indexing-B* **unfolding** *indexing-def bij-betw-def* **by** *auto*
have *a-ge-cardC*: $a \geq card\ C$
by (*metis imageI lessThan-iff not-leE surj-g-C x-eq-ga x-notin-C*)
hence *a-plus-one-ge-suc-card-C*: $a + 1 \geq Suc\ (card\ C)$ **by** *simp*
have *x-eq*:$x = g\ (a + 1 - Suc\ 0)$ **using** *x-eq-ga* **by** *simp*
have $a + 1 < card\ P'$ **using** *P′-def*
by (*metis Suc-eq-plus1 Suc-n-not-n a-l-cardB f0-notin-B indexing-B*

insert-iset-increase-card less-trans-Suc linorder-neqE-nat nat-add-commute not-add-less2)
thus *?thesis* **using** *a-plus-one-ge-suc-card-C* **and** *x-eq* **by** *fast*
qed
next
fix *xa*
assume *suc-C-le-xa*: $Suc\ (card\ C) \leq xa$ **and** *xa-l-cardP*: $xa < card\ P'$
and *g-xa0-in-C*:$g\ (xa - Suc\ 0) \in C$ — We will obtain a contradiction thanks to injectivity.
def $b \equiv g\ (xa - Suc\ 0)$
have *xa0-l-B*:$xa - Suc\ 0 < card\ B$ **using** *xa-l-cardP*
by (*metis One-nat-def P′-def Suc-less-SucD Suc-pred add-Suc-shift f0-notin-B gr0I gr-implies-not0 indexing-B insert-iset-increase-card less-eq-Suc-le nat-add-commute plus-nat.add-0 suc-C-le-xa*)
have *cardC-le-cardB*: $card\ C \leq card\ B$
by (*metis One-nat-def P′-def Suc-diff-1 Suc-le-lessD diff-add-inverse f0-notin-B indexing-B insert-iset-increase-card le-add1 le-trans nat-add-commute not-less-eq-eq order-less-not-sym suc-C-le-xa xa-l-cardP*)
hence *C-subset-B*: $C \subseteq B$ **using** *indexing-B surj-g-C* **unfolding**

```
indexing-def bij-betw-def
                unfolding image-def by fastsimp
              have b-in-C: b ∈ C using b-def g-xa0-in-C by auto
              from b-in-C obtain a where ga-eq-b: g a = b and a-l-cardC: a <
card C
                using surj-g-C unfolding image-def by force
              hence a ≠ xa − Suc 0  using suc-C-le-xa by auto
                thus False using indexing-B ga-eq-b a-l-cardC xa-l-cardP xa0-l-B
cardC-le-cardB
                inj-on-eq-iff [of g {..<card B} a xa − Suc 0]
                unfolding indexing-def bij-betw-def b-def
                by fastsimp
            qed
            hence  y∉ h′‘{i. Suc (card C)≤i ∧ i<(card P′)} using y-in-C by
simp
            hence k-l-cardC: ?k≤card C using image-h-C h′-k-eq-y k-l-card by
auto
          have image-h-card-in-Z: h′ ‘ {..<card C} ⊆ Z
          proof −
            have {..< card C}={0} ∪ {0<..< card C} using one-le-k k-l-cardC
by force
            hence h′ ‘ {..< card C}=h′ ‘{0} ∪ h′‘{0<..< card C} by blast
            also have ...= {f 0} ∪ h′‘{0<..< card C}
              using h′-def unfolding insert-iset-def indexing-ext-def by auto
            also have ...⊆ Z
            proof −
              have f 0 ∈ A
                using indexing-in-set[OF indexing-A -]
                A-not-empty indexing-finite[OF indexing-A] by (metis card-eq-0-iff
gr0I)
              thus ?thesis using image-h-C A-union-C by auto
            qed
            finally show ?thesis .
          qed
          have ld-insert: linear-dependent (insert y (h′‘{i. i<?k}))
          proof (rule lc1)
            show linear-independent (h′‘{i. i<?k})
            proof (rule independent-set-implies-independent-subset)
              show linear-independent Z using li-Z .
            next
              show h′ ‘ {i. i < ?k} ⊆ Z using image-h-card-in-Z k-l-cardC by
auto
            qed
            show y ∈ carrier V by (metis B-in-V subsetD y-in-B)
            show y∉ h′‘{i. i<?k}
              using y-in-C and h′-k-eq-y and k-l-card and indexing
              unfolding indexing-def and bij-betw-def and inj-on-def
              by force
            show ∃f. f ∈ coefficients-function (carrier V) ∧
```

linear-combination f $(h' \text{ ‘}\{i.\ i<?k\})=y$
using *cf-s* **and** *combinacion-anteriores* **by** *auto*
qed
have *linear-dependent Z*
proof (*rule linear-dependent-subset-implies-linear-dependent-set* [*of insert y* $(h'\text{‘}\{i.\ i<?k\})$])
show *insert y* $(h' \text{ ‘ } \{i.\ i<?k\}) \subseteq Z$
proof −
have $y \in Z$ **using** *y-in-C A-union-C* **by** *auto*
thus *?thesis* **using** *image-h-card-in-Z k-l-cardC* **by** *auto*
qed
show *good-set Z*
by (*metis l-ind-good-set li-Z*)
show *linear-dependent* (*insert y* $(h' \text{ ‘}\{i.\ i<?k\})$) **using** *ld-insert* **.**
qed
— Contradiction: we have linear dependent Z and linear independent Z
thus *?thesis* **using** *independent-implies-not-dependent*[*OF li-Z*] **by** *contradiction*
qed
qed
show *snd* (*remove-ld* (*insert-iset* (*B*, *g*) (*f 0*) *0*)) ‘ {..<*Suc* (*card C*)} = *insert* (*f 0*) *C*
proof −
have *eq*: *snd* (*remove-ld* (*insert-iset* (*B*, *g*) (*f 0*) *0*))=*snd*(*remove-iset*(*insert-iset* (*B*, *g*) (*f 0*) *0*) *?k*)
unfolding *remove-ld-def* **using** *snd-conv* **using** *remove-iset-def*[*of* (*insert-iset* (*B*, *g*) (*f 0*) *0*) *?k*]
unfolding P'*-def* h'*-def* **by** *force*
have {..<*Suc* (*card C*)}={*0*}∪{*0*<..<*Suc* (*card C*)} **by** *auto*
hence *snd* (*remove-ld* (*insert-iset* (*B*, *g*) (*f 0*) *0*)) ‘ {..<*Suc* (*card C*)}
=*snd* (*remove-ld* (*insert-iset* (*B*, *g*) (*f 0*) *0*)) ‘ {*0*}
∪ *snd* (*remove-ld* (*insert-iset* (*B*, *g*) (*f 0*) *0*)) ‘{*0*<..<*Suc* (*card C*)}
by *blast*
also have ...= {*f 0*} ∪ *snd* (*remove-ld* (*insert-iset* (*B*, *g*) (*f 0*) *0*)) ‘{*0*<..<*Suc* (*card C*)}
proof −
have *snd* (*insert-iset* (*B*, *g*) (*f 0*) *0*) ‘ {*0*}={*f 0*} **unfolding** *insert-iset-def indexing-ext-def* **by** *simp*
hence *snd* (*remove-iset*(*insert-iset* (*B*, *g*) (*f 0*) *0*) *?k*)‘{*0*}={*f 0*}
unfolding *remove-iset-def* **using** *one-le-k* **by** *auto*
thus *?thesis* **using** *eq* **by** *presburger*
qed
also have ...= {*f 0*} ∪ *C*
proof −
have *k-g-cardC*: $?k \geq$ *Suc* (*card C*) — No puede ser menor porque C es independiente!
proof (*cases* $?k \geq$ *Suc* (*card C*))
case *True* **thus** *?thesis* **.**

next
case *False* **note** *k-l-suc-cardC=False*
have *image-eq*:h'‘{*0*<..<*Suc* (*card C*)}=*g*‘{..<*card C*}
unfolding h'-*def insert-iset-def indexing-ext-def*
unfolding *image-def* **by** *force*
have *image-eq2*: h' *0* = *f 0* **unfolding** h'-*def insert-iset-def indexing-ext-def* **by** *simp*
have *ld-f0-C*: *linear-dependent* ({*f 0*} ∪ *g*‘{..<*card C*})
proof (*rule linear-dependent-subset-implies-linear-dependent-set*)

show *insert* (h' *?k*) (h' ‘{..<*?k*})⊆ {*f 0*} ∪ *g*‘{..<*card C*}
proof −
have *igualdad-conjuntos*: {..<*?k*} ∪ {*?k*}={*0*} ∪ {*0*<..*?k*} **using** *one-le-k* **by** *auto*
have *insert* (h' *?k*) (h'‘{..<*?k*})= h'‘ {*?k*} ∪ h'‘{..<*?k*} **by** *auto*
also have ...=h'‘ ({..<*?k*} ∪ {*?k*}) **by** *auto*
also have ...=h'‘ ({*0*} ∪ {*0*<..*?k*}) **using** *igualdad-conjuntos* **by** *auto*
also have ...={h' *0*} ∪ h'‘{*0*<..*?k*} **by** *auto*
also have ... ⊆ {*f 0*} ∪ *g*‘{..<*card C*} **using** *image-eq image-eq2 k-l-suc-cardC* **by** *auto*
finally show *?thesis* **.**
qed
show *good-set* ({*f 0*} ∪ *g* ‘ {..<*card C*})
using *f0-in-V*[*OF indexing-A A-in-V A-not-empty*] *surj-g-C l-ind-good-set*[*OF li-C*]
unfolding *good-set-def* **by** *simp*
show *linear-dependent* (*insert* (h' *?k*) (h' ‘{..<*?k*}))
proof (*rule lc1*)
show *linear-independent* (h' ‘{..<*?k*})
proof (*rule independent-set-implies-independent-subset*)

have h' ‘{..<*?k*}⊆{*f 0*} ∪ *g* ‘ {..<*card C*} **using** *image-eq image-eq2 k-l-suc-cardC* **by** *force*
also have ...⊆ *Z*
using *A-union-C A-not-empty surj-g-C*
indexing-in-set[*OF indexing-A*] *indexing-finite*[*OF indexing-A*]
by *force*
finally show h' ‘{..<*?k*} ⊆ *Z* **.**
next
show *linear-independent Z* **using** *li-Z* **.**
qed
show h' *?k* ∈ *carrier V*
proof −
have h' *?k* ∈ h'‘{..<*card* P'} **using** *k-l-card* **by** *blast*
also have ...=P' **using** *indexing* **unfolding** *indexing-def bij-betw-def* **by** *simp*
also have ...⊆ *carrier V* **using** P'-*def B-in-V f0-in-V*[*OF indexing-A A-in-V A-not-empty*]

```
          unfolding insert-iset-def by simp
        finally show ?thesis .
      qed
      show h′ ?k ∉ h′‘ {..<?k}
      proof (cases h′ ?k ∉ h′‘ {..<?k})
        case True thus ?thesis .
      next case False
          from this obtain s where hk-hs: h′ ?k = h′ s and s-in-set: s∈{..<?k} by auto
         hence s-not-k: s≠?k  and s-l-card: s<card P′ using k-l-card by auto
     have inj-on h′ {..<card P′} using indexing unfolding indexing-def bij-betw-def by auto
          hence h′ ?k ≠ h′ s using inj-on-eq-iff s-not-k s-l-card k-l-card by fastsimp
        thus ?thesis
          using hk-hs by contradiction
      qed
    next
   show ∃f. f ∈ coefficients-function (carrier V) ∧ linear-combination f (h′ ‘ {..<?k}) = h′ (?k)
      proof −
        have {i. i<?k}={..<?k} by fast
        thus ?thesis
          using cf-s and combinacion-anteriores h′-k-eq-y by auto
      qed
    qed
  qed
  have li-f0-C: linear-independent ({f 0} ∪ g‘{..<card C})
  proof (rule independent-set-implies-independent-subset)
    show {f 0} ∪ g ‘ {..<card C} ⊆ Z
      using A-union-C A-not-empty surj-g-C
        indexing-in-set[OF indexing-A] indexing-finite[OF indexing-A]
      by force
    show linear-independent Z  using li-Z .
  qed
   thus ?thesis using dependent-implies-not-independent[OF ld-f0-C] by contradiction
    — Contradiction
qed
  have snd (remove-iset(insert-iset (B, g) (f 0) 0) ?k) ‘ {0<..<Suc (card C)}=
  snd (insert-iset (B, g) (f 0) 0) ‘{0<..<Suc (card C)}
  unfolding remove-iset-def using k-g-cardC by auto
also have ...=C
proof (unfold insert-iset-def, unfold indexing-ext-def, unfold image-def, auto)
  fix xa
  assume 0 < xa and xa < Suc (card C)
```

thus *g* (*xa* − *Suc 0*) ∈ *C* **using** *surj-g-C* **by** *force*
next
fix *x*
assume *x-in-C*: *x* ∈ *C*
from *this* **obtain** *a* **where** *x* = *g a* **and** *a* < *card C* **using** *surj-g-C* **by** *blast*
thus ∃ *xa*∈{*0*<..<*Suc* (*card C*)}. *x* = *g* (*xa* − *Suc 0*) **using** *bexI*[*of - a+1*] **by** *force*
qed
finally show *?thesis* **using** *eq* **by** *presburger*
qed
also have ...=*insert* (*f 0*) *C* **by** *simp*
finally show *?thesis* **.**
qed
qed
qed
qed
qed
qed

Another important auxiliary lemma. Applying the swap function n-times (with $n < card(A)$) to $((A, f), B, g)$, where A is independent and B a spanning set, we will have that the first n elements of A will be in the first positions of the second component of the result. Of course, these elements come from A and thus they are independent. We make use of *aux-swap-theorem1* to prove this lemma.

lemma *aux-swap-theorem2*:
assumes *indexing-A*: *indexing* (*A*,*f*)
and *indexing-B*: *indexing* (*B*,*g*)
and *B-in-V*: *B* ⊆ *carrier V*
and *A-not-empty*: *A*≠{}
and *li-A*: *linear-independent A*
and *sg-B*: *spanning-set B*
and *zero-notin-B*: $\mathbf{0}_V$ ∉ *B*
and *n-l-cardA*: *n* < *card A*
shows *f*‘{..<*n*}
= *iset-to-index*(*snd*((*swap-function*^(*n*)) ((*A*,*f*),(*B*,*g*))))‘{..<*n*}
∧ *iset-to-index*(*snd*((*swap-function*^(*n*)) ((*A*,*f*),(*B*,*g*))))‘{..<*n*}
⊂ *A*
∧ *linear-independent*
(*iset-to-index*(*snd*((*swap-function*^(*n*)) ((*A*,*f*),(*B*,*g*))))‘{..<*n*})
∧ *n* = (*card* (*iset-to-index*(*snd*((*swap-function*^(*n*))
((*A*,*f*),(*B*,*g*))))‘{..<*n*}))
using *n-l-cardA*
proof (*induct n*)
show *f* ‘ {..<*0*} = *iset-to-index* (*snd* ((*swap-function* ^ *0*) ((*A*, *f*), *B*, *g*))) ‘ {..<*0*} ∧
iset-to-index (*snd* ((*swap-function* ^ *0*) ((*A*, *f*), *B*, *g*))) ‘ {..<*0*} ⊂ *A* ∧
linear-independent (*iset-to-index* (*snd* ((*swap-function* ^ *0*) ((*A*, *f*), *B*, *g*))) ‘

{..<0}) ∧
0 = card (iset-to-index (snd ((swap-function ^ 0) ((A, f), B, g))) ‘ {..<0})
proof −
have *iset-to-index* (*snd* ((*swap-function* ^ *0*) ((*A*, *f*), *B*, *g*))) ‘ {..<*0*}={} **by** *simp*
hence *li*: *linear-independent* (*iset-to-index* (*snd* ((*swap-function* ^ *0*) ((*A*, *f*), *B*, *g*))) ‘ {..<*0*})
using *empty-set-is-linearly-independent* **by** *presburger*
show *?thesis* **using** *li* **and** *A-not-empty* **by** *auto*
qed
case *Suc*
fix *n*
assume *hip-induct*: *n* < *card A* ⟹
f ‘ {..<*n*} = *iset-to-index* (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))) ‘ {..<*n*} ∧
iset-to-index (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))) ‘ {..<*n*} ⊂ *A* ∧
linear-independent (*iset-to-index* (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))) ‘ {..<*n*}) ∧
n = *card* (*iset-to-index* (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))) ‘ {..<*n*})
and *suc-n-l-card*: *Suc n* < *card A*
have *n-l-card*: *n* < *card A* **using** *suc-n-l-card* **by** *simp*
hence *hip*: *f* ‘ {..<*n*} = *iset-to-index* (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))) ‘ {..<*n*} ∧
iset-to-index (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))) ‘ {..<*n*} ⊂ *A* ∧
linear-independent (*iset-to-index* (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))) ‘ {..<*n*}) ∧
n = *card* (*iset-to-index* (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*))) ‘ {..<*n*})
using *hip-induct* **by** *simp*
show *f* ‘ {..<*Suc n*} = *iset-to-index* (*snd* ((*swap-function* ^ *Suc n*) ((*A*, *f*), *B*, *g*))) ‘ {..<*Suc n*} ∧
iset-to-index (*snd* ((*swap-function* ^ *Suc n*) ((*A*, *f*), *B*, *g*))) ‘ {..<*Suc n*} ⊂ *A* ∧
linear-independent (*iset-to-index* (*snd* ((*swap-function* ^ *Suc n*) ((*A*, *f*), *B*, *g*))) ‘ {..<*Suc n*}) ∧
Suc n = *card* (*iset-to-index* (*snd* ((*swap-function* ^ *Suc n*) ((*A*, *f*), *B*, *g*))) ‘ {..<*Suc n*})
proof (*rule conjI4*)
have *A-in-V*: *A* ⊆ *carrier V*
by (*metis good-set-in-carrier l-ind-good-set li-A*)
def *C*==*iset-to-index* (*snd* ((*swap-function* ^ *Suc n*) ((*A*, *f*), *B*, *g*))) ‘ {..<*Suc n*}
show *image-C*: *f* ‘ {..<*Suc n*} = *C*
proof −
def *A'*≡*iset-to-set* (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*)))
def *f'*≡*iset-to-index* (*fst* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*)))
def *B'*≡*iset-to-set* (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*)))
def *g'*≡*iset-to-index* (*snd* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*)))
def *C'*≡*f* ‘ {..<*n*}
have *snd* ((*swap-function* ^ *Suc n*) ((*A*, *f*), *B*, *g*))

$=$*snd* (*swap-function* ((*swap-function* ^ *n*) ((*A*, *f*), *B*, *g*)))
using *fun-power-suc-eq* **by** *metis*
also have ...=*snd* (*swap-function* ((*A'*,*f'*),(*B'*,*g'*)))
using *A'-def B'-def f'-def g'-def* **by** *simp*
finally have *descomposicion*: *snd* ((*swap-function* ^ *Suc n*) ((*A*, *f*), *B*, *g*)) =
snd (*swap-function* ((*A'*, *f'*), *B'*, *g'*)) **.**
have $\exists\, y \in B'$. *iset-to-set* (*snd* (*swap-function* ((*A'*,*f'*),(*B'*,*g'*))))=(*insert* (*f'* *0*) ($B' - \{y\}$)) $\wedge$ $y \notin C'$ $\wedge$
iset-to-index (*snd* (*swap-function* ((*A'*, *f'*), *B'*, *g'*)))' {..<*card* (*C'*) + *1*} = $C' \cup \{f'\ 0\}$
proof (*rule aux-swap-theorem1*)
show *indexing* (*A'*, *f'*)
using *fst-swap-function-power-indexing*[*OF indexing-A A-in-V*, *of n B g*]
unfolding *A'-def f'-def* **by** *simp*
show *indexing* (*B'*, *g'*)
using *snd-swap-function-power-indexing*
[*OF indexing-A indexing-B A-in-V B-in-V A-not-empty li-A sg-B zero-notin-B n-l-card*]
unfolding *B'-def g'-def* **by** *simp*
show $B' \subseteq$ *carrier V* **unfolding** *B'-def*
using *swap-function-power-snd-in-carrier*[*OF indexing-A indexing-B A-in-V B-in-V*
A-not-empty li-A sg-B zero-notin-B n-l-card] **.**
show $A' \neq \{\}$
unfolding *A'-def*
using *swap-function-power-fst-not-empty-if-n-l-cardA*[*OF indexing-A A-in-V n-l-card*]
by *presburger*
show *spanning-set B'*
unfolding *B'-def*
using *swap-function-power-preserves-sg*[*OF indexing-A indexing-B A-in-V B-in-V*
A-not-empty li-A sg-B zero-notin-B n-l-card] **.**
show $\mathbf{0}_V \notin B'$
unfolding *B'-def* **using** *zero-notin-snd-swap-function-power*[*OF indexing-A indexing-B A-in-V B-in-V*
A-not-empty li-A sg-B zero-notin-B n-l-card] **.**
show *g'* ' {..<*card C'*} = *C'*
unfolding *g'-def C'-def* **using** *hip* **by** *presburger*
show $A' \cup C' = A$
proof −
have *A'*=*f*'{*n*..<*card A*} **using** *swap-function-fst-image-until-n*[*OF indexing-A A-not-empty n-l-card*]
unfolding *A'-def* **by** *auto*
hence $A' \cup C'$=*f*'{*n*..<*card A*} $\cup$ *f*'{..<*n*} **unfolding** *C'-def* **by** *fast*
also have ...=*f*'{..<*card A*}
by (*metis C'-def Un-commute* ‹*A'* = *f* ' {*n*..<*card A*}›
image-Un ivl-disj-un(*8*) *n-l-card nat-less-le*)
also have ...=*A* **using** *indexing-A* **unfolding** *indexing-def bij-betw-def* **by**

simp
finally show *?thesis* **.**
qed
show $A' \cap C' = \{\}$
proof −
have A'=f‘{*n*..<*card A*}
using *swap-function-fst-image-until-n*[*OF indexing-A A-not-empty n-l-card*]
unfolding A'-*def* **by** *auto*
hence $A' \cap C'$=f‘{*n*..<*card A*} ∩ C' **by** *simp*
also have ...=f‘{*n*..<*card A*} ∩ f‘{..<*n*} **unfolding** C'-*def* **..**
also have ...=f‘({*n*..<*card A*} ∩ {..<*n*})
proof (*rule inj-on-image-Int*[*symmetric*])
show *inj-on f* {..<*card A*} **using** *indexing-A* **unfolding** *indexing-def bij-betw-def* **by** *simp*
show {*n*..<*card A*} ⊆ {..<*card A*} **using** *n-l-card* **by** *fastsimp*
show {..<*n*} ⊆ {..<*card A*} **using** *n-l-card* **by** *simp*
qed
also have ...=f‘{} **by** *auto*
also have ...={} **by** *simp*
finally show *?thesis* **.**
qed
show *linear-independent A* **using** *li-A* **.**
qed
hence *image*: *iset-to-index* (*snd* (*swap-function* ((A', f'), B', g')))‘ {..<*card* (C') + *1*} = C'∪{f' *0*}
by *fast*
have f‘{..<*Suc n*}=f ‘ {..<*n*}∪{f' *0*}
proof −
have f'*0-fn*:f' *0*=f *n* **unfolding** f'-*def*
using *snd-fst-swap-function-image-0*[*OF indexing-A n-l-card*, *of B g*] **by** *simp*
have f‘{..<*Suc n*}=f ‘ {..<*n*}∪{f *n*}
by (*metis* C'-*def Un-empty-right Un-insert-right image-insert lessThan-Suc*)
also have ...=f ‘ {..<*n*}∪{f' *0*} **using** f'*0-fn* **by** *presburger*
finally show *?thesis* **.**
qed
also have ...=C'∪{f' *0*} **using** C'-*def* **by** *simp*
also have ...=*iset-to-index* (*snd* (*swap-function* ((A', f'), B', g')))‘ {..<*card* (C') + *1*}
using *image* **by** *fast*
also have ...=*iset-to-index* (*snd* (*swap-function* ((A', f'), B', g')))‘ {..<*Suc n*}
proof −
have *igualdad-conjuntos*: {..<*card* (C') + *1*}={..<*Suc n*}
proof −
have *card* C'= *card* {..<*n*}
proof (*unfold* C'-*def*, *rule card-image*)
show *inj-on f* {..<*n*}**using** *subset-inj-on indexing-A -n-l-card*

unfolding *indexing-def bij-betw-def* **by** *fastsimp*
qed
also have ...=n **by** *simp*
finally show *?thesis* **by** *simp*
qed
thus *?thesis* **by** *presburger*
qed
also have ...=*iset-to-index* (*snd* ((*swap-function* ^ *Suc n*) ((*A*, *f*), *B*, *g*)))‘ {..<*Suc n*}
using *C-def A′-def B′-def f′-def g′-def descomposicion* **by** *presburger*
finally show *?thesis* **using** *C-def* **by** *presburger*
qed
show *linear-independent C*
proof –
have *f* ‘ {..<*Suc n*}⊆ *A* **using** *indexing-A suc-n-l-card* **unfolding** *indexing-def bij-betw-def* **by** *fastsimp*
thus *?thesis* **using** *independent-set-implies-independent-subset* **using** *image-C li-A* **unfolding** *C-def* **by** *force*
qed
show *C* ⊂ *A*
proof –
have *f*‘ {..<*Suc n*} ⊂ *f*‘ {..<*card A*}
proof (*rule inj-on-strict-subset*)
show *inj-on f* {..<*card A*} **using** *indexing-A* **unfolding** *indexing-def bij-betw-def* **by** *simp*
show {..<*Suc n*} ⊂ {..<*card A*} **using** *suc-n-l-card* **by** *fastsimp*
qed
thus *?thesis* **using** *indexing-A image-C* **unfolding** *indexing-def bij-betw-def C-def* **by** *auto*
qed
show *Suc n* = *card C*
proof –
have *card C* = *card* (*f* ‘ {..<*Suc n*}) **using** *image-C* **by** *fast*
also have ...= *card* {..<*Suc n*}
proof (*rule card-image*)
show *inj-on f* {..<*Suc n*}
using *subset-inj-on indexing-A suc-n-l-card*
unfolding *indexing-def bij-betw-def*
by *fastsimp*
qed
also have ...= *Suc n* **using** *card-lessThan* **by** *simp*
finally show *?thesis* **by** *presburger*
qed
qed
qed

At last, we can prove the swap theorem. We separate it in cases, when A is empty and when it is not. We use the auxiliar lemma *aux-swap-theorem2*.

theorem *swap-theorem-not-empty*:

```
  assumes indexing-A: indexing (A,f)
  and indexing-B: indexing (B,g)
  and A-in-V: A ⊆ carrier V
  and B-in-V: B ⊆ carrier V
  and A-not-empty: A≠{}
  and li-A: linear-independent A
  and sg-B: spanning-set B
  and zero-notin-B: 0_V ∉ B
shows card A ≤ card B
proof (cases card A ≤ card B)
  case True thus ?thesis .
next
  case False
  have cardB-l-cardA: card A > card B using False by linarith
  def C≡iset-to-index(snd((swap-function^(card B)) ((A,f),(B,g))))‘{..<card B}
  have C-eq: C=iset-to-set(snd((swap-function^(card B)) ((A,f),(B,g))))
    using snd-swap-function-power-indexing
    [OF indexing-A indexing-B A-in-V B-in-V A-not-empty
      li-A sg-B zero-notin-B cardB-l-cardA]
    unfolding C-def indexing-def bij-betw-def
    using snd-swap-function-power-preserves-card
    [OF indexing-A indexing-B A-in-V B-in-V
      A-not-empty li-A sg-B zero-notin-B cardB-l-cardA]
    by simp
  have surjf-B-C: f‘{..<card B}=C
    and C-subset-A:C ⊂ A
    and li-C:linear-independent C
    and cB-eq-cC:card B=card C
    using aux-swap-theorem2 assms cardB-l-cardA
    unfolding C-def by auto
  have spanning-set-C: spanning-set C
    using swap-function-power-preserves-sg
    [OF indexing-A indexing-B A-in-V B-in-V A-not-empty
      li-A sg-B zero-notin-B cardB-l-cardA] C-eq
    unfolding C-def by presburger
  have linear-dependent A
  proof −
    have ∃ x. x∈A ∧ x∉C using C-subset-A by fast
    from this obtain x where x-in-A: x∈A and x-notin-C: x∉C
      by blast
    show ?thesis
    proof (rule linear-dependent-subset-implies-linear-dependent-set
        [of insert x C])
      show insert x C ⊆ A using C-subset-A and x-in-A by simp
      show good-set A using li-A linear-independent-def by blast
      show linear-dependent (insert x C)
      proof (rule lc1)
        show linear-independent C using li-C .
        show x-in-V: x ∈ carrier V
```

```
        by (metis good-set-def li-A linear-independent-def
          subsetD x-in-A)
      show x ∉ C using x-notin-C .
      show ∃f. f ∈ coefficients-function (carrier V)
        ∧ linear-combination f C = x
        using spanning-set-C x-in-V
        unfolding spanning-set-def by blast
    qed
  qed
qed
hence ¬ linear-independent A
  using dependent-implies-not-independent by simp
thus ?thesis using li-A by contradiction
qed
```

Finally the theorem (every independent set has cardinal less than or equal to every spanning set) and some corollaries:

```
theorem swap-theorem:
  assumes indexing-A: indexing (A,f)
  and indexing-B: indexing (B,g)
  and A-in-V: A ⊆ carrier V
  and B-in-V: B ⊆ carrier V
  and li-A: linear-independent A
  and sg-B: spanning-set B
  and zero-notin-B: 0_V ∉ B
  shows card A ≤ card B
proof (cases A={})
  case True show ?thesis by (metis True card-eq-0-iff le0)
next
  case False show ?thesis using swap-theorem-not-empty assms False by force
qed
```

The next corollary omits the need of indexing functions for A and B (these are obtained through auxiliary lemmas).

```
corollary swap-theorem2:
  assumes finite-B: finite B
  and B-in-V: B ⊆ carrier V
  and A-in-V: A ⊆ carrier V
  and li-A: linear-independent A
  and sg-B: spanning-set B
  and zero-notin-B: 0_V ∉ B
  shows card A ≤ card B
proof -
  have ∃f. indexing (A,f) using obtain-indexing
    by (metis good-set-finite l-ind-good-set li-A)
  from this obtain f where indexing-A: indexing (A,f) by fast
  have ∃g. indexing (B,g) using obtain-indexing[OF finite-B] .
  from this obtain g where indexing-B: indexing (B,g) by fast
  show ?thesis using swap-theorem
```

[*OF indexing-A indexing-B A-in-V B-in-V*
li-A sg-B zero-notin-B] **.**
qed

Now we can prove that the number of elements in any (finite) basis (of a finite-dimensional vector space) is the same as in any other (finite) basis.

theorem *eq-cardinality-basis*:
assumes *basis-B*: *basis B*
and *finite-B*: *finite B*
shows *card X* = *card B*
proof −
have $\exists f$. *indexing* (X,f) **using** *obtain-indexing*[*OF finite-X*] **.**
from *this* **obtain** f **where** *indexing-X*: *indexing* (X,f) **by** *fast*
have $\exists g$. *indexing* (B,g) **using** *obtain-indexing*[*OF finite-B*] **.**
from *this* **obtain** g **where** *indexing-B*: *indexing* (B,g) **by** *fast*
have *li-X*: *linear-independent X* **and** *sg-X*: *spanning-set X*
using *linear-independent-X* **and** *spanning-set-X* **by** *fast+*
have *gs-B*: *good-set B*
using *finite-basis-implies-good-set*[*OF basis-B finite-B*] **.**
have *li-B*: *linear-independent B* **and** *sg-B*: *spanning-set B*
using *basis-B finite-B* **unfolding** *basis-def*
using *fin-ind-ext-impl-ind*
gs-spanning-ext-imp-spanning gs-B **by** *blast+*
have *cardX-le-cardB*: *card X* $\leq$ *card B*
proof (*rule swap-theorem*)
show *indexing* (X, f) **using** *indexing-X* **.**
show *indexing* (B, g) **using** *indexing-B* **.**
show $X \subseteq$ *carrier V*
using *finite-basis-implies-good-set*[*OF basis-X finite-X*]
unfolding *good-set-def* **by** *simp*
show $B \subseteq$ *carrier V*
using *finite-basis-implies-good-set*[*OF basis-B finite-B*]
unfolding *good-set-def* **by** *simp*
show *linear-independent X* **using** *li-X* **.**
show *spanning-set B* **using** *sg-B* **.**
show $\mathbf{0}_V \notin B$
using *zero-not-in-linear-independent-set*[*OF li-B*] **.**
qed
have *cardX-ge-cardB*: *card X* $\geq$ *card B*
proof (*rule swap-theorem*)
show *indexing* (B, g) **using** *indexing-B* **.**
show *indexing* (X, f) **using** *indexing-X* **.**
show $X \subseteq$ *carrier V*
using *finite-basis-implies-good-set*[*OF basis-X finite-X*]
unfolding *good-set-def* **by** *simp*
show $B \subseteq$ *carrier V*
using *finite-basis-implies-good-set*[*OF basis-B finite-B*]
unfolding *good-set-def* **by** *simp*
show *linear-independent B* **using** *li-B* **.**

show *spanning-set X* **using** *sg-X* **.**
show $\mathbf{0}_V \notin X$
using *zero-not-in-linear-independent-set*[*OF li-X*] **.**
qed
show *?thesis*
using *cardX-le-cardB* **and** *cardX-ge-cardB* **by** *presburger*
qed

corollary *eq-cardinality-basis2*:
assumes *basis-A*: *basis A*
and *finite-A*: *finite A*
and *basis-B*: *basis B*
and *finite-B*: *finite B*
shows *card A = card B*
by (*metis basis-A basis-B eq-cardinality-basis finite-A finite-B*)

We can make the definicion of dimension of a vector space and relationate the concept with above theorems.

The dimension of a vector space is the cardinality of one of its basis. We have fixed X as a basis, so the definition is trivial:

definition *dimension* :: *nat*
where *dimension = card X*

If we have another basis, the dimension is equal to its cardinality.

lemma *eq-dimension-basis*:
assumes *basis-A*: *basis A*
and *finite-A*: *finite A*
shows *dimension = card A*
by (*metis basis-A dimension-def eq-cardinality-basis finite-A*)

Whenever we have an independent set, we will know that its cardinality is less than the dimension of the vector space.

lemma *card-li-le-dim*:
assumes *li-A*: *linear-independent A*
shows *card A* $\leq$ *dimension*
proof −
have $\exists f$. *indexing* (*X*,*f*) **using** *obtain-indexing*[*OF finite-X*] **.**
from *this* **obtain** *f* **where** *indexing-X*: *indexing* (*X*,*f*) **by** *fast*
have *finite-A*: *finite A*
by (*metis assms good-set-finite l-ind-good-set*)
have $\exists g$. *indexing* (*A*,*g*) **using** *obtain-indexing*[*OF finite-A*] **.**
from *this* **obtain** *g* **where** *indexing-A*: *indexing* (*A*,*g*) **by** *fast*
have *li-X*: *linear-independent X* **and** *sg-X*: *spanning-set X*
by *auto*
show *?thesis*
proof (*unfold dimension-def*, *rule swap-theorem*)
show *indexing* (*A*, *g*) **using** *indexing-A* **.**
show *indexing* (*X*, *f*) **using** *indexing-X* **.**

show $A \subseteq$ *carrier V*
by (*metis assms good-set-in-carrier l-ind-good-set*)
show $X \subseteq$ *carrier V*
by (*metis good-set-X good-set-in-carrier*)
show *linear-independent A* **using** *li-A* **.**
show *spanning-set X* **using** *sg-X* **.**
show $\mathbf{0}_V \notin X$ **by** (*metis li-X zero-not-in-linear-independent-set*)
qed
qed

Whenever the cardinality of a set is greater (strictly) than the dimension of V then the set is dependent.

corollary *card-g-dim-implies-ld*:
assumes *card-g-dim*: *card A* $>$ *dimension*
and *A-in-V*: $A \subseteq$ *carrier V*
shows *linear-dependent A*
proof $-$
have *finite-A*: *finite A*
using *card-g-dim finite-X* **unfolding** *dimension-def*
by (*metis card.empty card-g-dim*
card-infinite card-li-le-dim dimension-def
empty-set-is-linearly-independent linorder-not-le)
hence *cb-A*: *good-set A*
using *A-in-V* **unfolding** *good-set-def* **by** *fast*
thus *?thesis* **using** *card-li-le-dim*
by (*metis card-g-dim dependent-if-only-if-not-independent*
dimension-def less-not-refl xt1(8))
qed

The following lemma proves that the cardinality of any spanning set is greater than the dimension. In the infinite case (when A is not finite but is a *spanning-set-ext*) it would be trivial, but Isabelle assigns 0 as the cardinality of an infinite set.

We will use *swap-theorem*, so $\mathbf{0}_V$ must not be in the *spanning-set* over we apply it.

lemma *card-sg-ge-dim*:
assumes *sg-A*: *spanning-set A*
shows *card A* $\geq$ *dimension*
proof $-$
have *finite-A*: *finite A* **and** *A-in-V*: $A \subseteq$ *carrier V*
using *sg-A* **unfolding** *spanning-set-def* **and** *good-set-def*
by *fast+*
have $\exists f$. *indexing* (X,f) **using** *obtain-indexing*[*OF finite-X*] **.**
from *this* **obtain** *f* **where** *indexing-X*: *indexing* (X,f) **by** *fast*
have $\exists g$. *indexing* $(A-\{\mathbf{0}_V\},g)$ **using** *obtain-indexing finite-A*
by *blast*
from *this* **obtain** *g* **where** *indexing-A*: *indexing* $(A-\{\mathbf{0}_V\},g)$
by *fast*

```
  have li-X: linear-independent X and sg-X: spanning-set X
    by auto
  have card (A−{0_V}) ≥ dimension
  proof (unfold dimension-def, rule swap-theorem)
    show indexing (A−{0_V}, g) using indexing-A .
    show indexing (X, f) using indexing-X .
    show (A−{0_V}) ⊆ carrier V using A-in-V by blast
    show  linear-independent X by simp
    show X ⊆ carrier V
      by (metis good-set-X good-set-in-carrier)
    show  spanning-set (A−{0_V})
      by (metis A-in-V finite-A sg-A spanning-set-minus-zero)
    show  0_V ∉ (A−{0_V}) by fast
  qed
  thus ?thesis by (metis card-Diff1-le finite-A le-trans)
qed
```

There not exists a *spanning-set* with cardinality less than the dimension.

```
corollary card-less-dim-implies-not-sg:
  assumes cardA-l-dim: card A < dimension
  shows ¬ spanning-set A
  by (metis assms card-sg-ge-dim dimension-def
    less-not-refl3 xt1(8))
```

If we have a set which cardinality is equal to the dimension of a finite vector space, then it is a finite set. We have to assume that the basis is not empty: if X is empty, then $card(X) = 0 = card(A)$. However and due to the implementation of cardinality in Isabelle (giving 0 as the cardinality of an infinite set), we could only prove that either A is infinite or empty.

```
lemma card-eq-not-empty-basis-implies-finite:
  assumes cardA-dim: card A = dimension
  and X-not-empty: X≠{}
  shows finite A
  by (metis X-not-empty cardA-dim card-eq-0-iff
    card-infinite dimension-def finite-X)
```

Assuming that A is in V, the problem is solved.

```
lemma card-eq-basis-implies-finite:
  assumes cardA-dim: card A = dimension
  and A-in-V: A ⊆ carrier V
  shows finite A
proof (cases X={})
  case True show ?thesis
    by (metis A-in-V True finite.insertI finite-X
      finite-subset span-basis-is-V span-empty)
next
  case False show ?thesis
    using card-eq-not-empty-basis-implies-finite
```

```
  [OF cardA-dim False] .
qed
```

If a set has cardinality equal to the dimension, if it is a basis then is independent.

```
lemma card-eq-basis-imp-li:
  assumes cardA-dim: card A = dimension
  shows basis A ⟹ linear-independent A
proof –
  assume basis-A: basis A
  hence A-in-V: A ⊆ carrier V unfolding basis-def by fast
  show linear-independent A
  proof (cases X={})
    case False show ?thesis
      using card-eq-not-empty-basis-implies-finite
      [OF cardA-dim False]
        and basis-A
      unfolding basis-def linear-independent-ext-def
      by (metis subset-refl)
  next
    case True
    have A={} using A-in-V True
      unfolding basis-def spanning-set-def
      by (metis all-not-in-conv assms card.empty
        card-eq-0-iff dimension-def finite.emptyI
        finite.insertI finite-subset mem-def
        span-basis-is-V span-empty)
    thus ?thesis
      using empty-set-is-linearly-independent by simp
  qed
qed
```

If we have an independent set with cardinality equal to the dimension, then this set is a basis.

```
lemma card-li-set-eq-basis-imp-li:
  assumes card-eq-dim: card A = dimension
  shows linear-independent A ⟹ basis A
proof –
  assume li-A: linear-independent A
  have finite-A: finite A
    by (metis good-set-finite l-ind-good-set li-A)
  have cb-A: good-set A using l-ind-good-set[OF li-A] .
  show ?thesis
  proof (unfold basis-def, rule conjI3)
    show A ⊆ carrier V
      using cb-A unfolding good-set-def by fast
    show linear-independent-ext A
      using independent-imp-independent-ext[OF li-A] .
    show spanning-set-ext A
```

proof (*cases spanning-set A*)
case *True* **thus** *?thesis*
using *spanning-imp-spanning-ext* **by** *fast*
next
case *False*
show *?thesis*
proof −
have $\exists\, y.\ y \in (carrier\ V\ -\ span\ A)$
using *False cb-A*
unfolding *span-def spanning-set-def* **by** *fast*
from *this* **obtain** *y*
where *y-in-V-minus-span*: $y \in (carrier\ V\ -\ span\ A)$
by *fast*
hence *linear-independent* (*insert y A*)
using *insert-y-notin-span-li*[*OF - - li-A*]
y-in-V-minus-span **by** *fast*
hence *card* (*insert y A*) $\leq$ *dimension*
using *card-li-le-dim* **by** *simp*
hence *card A* + *1* $\leq$ *dimension*
using *y-in-V-minus-span card-insert-if*[*OF finite-A*]
not-in-span-impl-not-in-set[*OF - cb-A*]
by *simp*
thus *?thesis* **using** *card-eq-dim* **by** *linarith*
— Contradiction: we have proved that card(A+1)≤dimension and card(A)=dimension.
qed
qed
qed
qed

If a spanning set has cardinality equal to the dimension, then is independent (so a basis).

lemma *card-sg-set-eq-basis-imp-li*:
assumes *card-eq-dim*: *card A* = *dimension*
shows *spanning-set A* $\Longrightarrow$ *linear-independent A*
proof−
assume *sg-A*: *spanning-set A*
hence *A-in-V*: $A \subseteq carrier\ V$
unfolding *spanning-set-def good-set-def* **by** *fast*
show *?thesis*
proof (*cases linear-independent A*)
case *True* **thus** *?thesis* **.**
next
case *False*
show *?thesis*
proof (*cases X*={})
case *True*
have *A*={}
by (*metis A-in-V True bot-apply card-eq-0-iff card-eq-dim*

dimension-def ext finite.emptyI finite.insertI
rev-finite-subset span-basis-is-V span-empty)
thus *?thesis* **using** *empty-set-is-linearly-independent* **by** *simp*
next
case *False*
have *finite-A*: *finite A*
by (*metis False card-eq-dim*
card-eq-not-empty-basis-implies-finite dimension-def)
have *ld-A*: *linear-dependent A*
by (*metis A-in-V* ‹¬ *linear-independent A*› *good-set-def*
dependent-if-only-if-not-independent finite-A)
have $\exists y \in A.\ \exists g.\ g \in$ *coefficients-function* (*carrier V*)
$\wedge\ y =$ *linear-combination g* $(A - \{y\})$
using *exists-x-linear-combination2*[*OF ld-A*] **.**
from *this* **obtain** *y g* **where** *y-in-A*: $y \in A$
and *cf-g*: $g \in$ *coefficients-function* (*carrier V*)
and *y-lc-Ay*: $y =$ *linear-combination g* $(A - \{y\})$ **by** *blast*
have *span A* = *span* $(A-\{y\})$
proof (*rule span-minus*)
show *good-set A*
by (*metis l-dep-good-set ld-A*)
show $y \in A$ **using** *y-in-A* **.**
show $\exists g.\ g \in$ *coefficients-function* (*carrier V*)
$\wedge\ y =$ *linear-combination g* $(A - \{y\})$
by (*metis cf-g y-lc-Ay*)
qed
hence *sg-Ay*: *spanning-set* $(A-\{y\})$ **using** *sg-A*
by (*metis A-in-V Diff-subset finite-A finite-Diff*
good-set-def span-V-eq-spanning-set
spanning-set-implies-span-basis subset-trans)
have ¬ *spanning-set* $(A-\{y\})$
proof (*rule card-less-dim-implies-not-sg*)
show *card* $(A - \{y\}) <$ *dimension*
by (*metis False card-Diff-singleton-if card-eq-dim*
card-gt-0-iff diff-less dimension-def finite-A
finite-X y-in-A zero-less-one)
qed
thus *?thesis* **using** *sg-Ay* **by** *contradiction*
— CONTRADICTION: we have proved that the set A minus the element y is a *spanning-set* and at the same time that it is not.
qed
qed
qed

corollary *card-sg-set-eq-basis-imp-basis*:
assumes *card-eq-dim*: *card A* = *dimension*
shows *spanning-set A* $\Longrightarrow$ *basis A*
by (*metis assms card-li-set-eq-basis-imp-li*
card-sg-set-eq-basis-imp-li)

lemma *basis-iff-linear-independent*:
assumes *card-eq*: *card A* = *dimension*
shows *basis A* ⟷ *linear-independent A*
by (*metis assms card-eq-basis-imp-li*
card-li-set-eq-basis-imp-li)

We can remove from *eq-cardinality-basis2* the premises about finiteness: we can prove that in a finite dimensional vector space there not exist infinite bases.

lemma
not-finite-A-contains-empty-set:
assumes *A*: ¬ *finite A*
shows {} ⊆ *A*
using *empty-subsetI* [*of A*] **.**

lemma *not-finite-diff*:
assumes *A*: ¬ *finite A*
shows ¬ *finite* (*A* − {*x*})
using *A* **by** *auto*

lemma *not-finite-diff-set*:
assumes *A*: ¬ *finite A*
and *B*: *finite B*
shows ¬ *finite* (*A* − *B*)
using *A B* **by** *auto*

We can obtain a subset with the number of elements that we want from an infinite set:

lemma *subset-card-n*:
assumes *A*: ¬ *finite A*
shows ∀ *k*::*nat*. ∃ *B*. *B* ⊆ *A* ∧ *card B* = *k*
proof (*rule allI*)
fix *k*
show ∃ *B*⊆*A*. *card B* = *k*
proof (*induct k*)
let *?P* = (λ*k C*. *C* ⊆ *A* ∧ *card C* = *k*)
case *0*
show *?case*
by (*rule exI* [*of ?P 0* {}], *intro conjI*)
(*rule not-finite-A-contains-empty-set* [*OF A*], *simp*)
next
case (*Suc k*)
show *?case*
proof −
obtain *B* **where** *b*: *B* ⊆ *A* **and** *card-b*: *card B* = *k*
using *Suc.hyps* **by** *auto*
show *?thesis*
using *b* **and** *card-b*

proof (*cases* $k = 0$)
case *True*
obtain x **where** x: $x \in A$
using A **by** (*metis ex-in-conv finite.emptyI*)
hence $\{x\} \subseteq A$ **and** *card* $\{x\} = Suc\ 0$ **by** *simp-all*
thus *?thesis* **unfolding** *True* **by** *auto*
next
case *False*
have *fin-B*: *finite B* **using** *card-b False* **by** (*metis card-eq-0-iff*)
hence *nfin-A-B*: $\neg$ *finite* $(A - B)$
using A **by** *auto*
then obtain x **where** x: $x \in A - B$ **by** (*metis ex-in-conv finite.emptyI*)
show *?thesis*
apply (*rule exI* [*of - insert x B*])
using *x card-b b*
by *auto* (*metis card-b card-insert-disjoint fin-B*)
qed
qed
qed
qed

Every basis of a finite dimensional vector space is finite (because each set of cardinality greater than the dimension is linearly dependent (*card-g-dim-implies-ld*), so we can not have an infinite basis).

lemma *basis-not-infinite*:
assumes *basis-A*: *basis A*
shows *finite A*
proof (*rule classical*)
assume *not-finite*: $\neg$ *finite A*
from *not-finite* **obtain** B **where** *card*: *card* B = *dimension* + *1*
and *B-in-A*: $B \subseteq A$ **using** *subset-card-n* **by** *blast*
have *li-ext-A*: *linear-independent-ext A* **by** (*metis assms basis-def*)
have *linear-dependent-ext A*
proof (*unfold linear-dependent-ext-def*, *rule exI*[*of - B*], *rule conjI*)
show *linear-dependent B*
proof (*rule card-g-dim-implies-ld*)
show *dimension* < *card B* **using** *card* **by** *simp*
show $B \subseteq$ *carrier V* **using** *B-in-A basis-A* **unfolding** *basis-def* **by** *fast*
qed
show $B \subseteq A$ **using** *B-in-A* **.**
qed
thus *?thesis* **using** *independent-ext-implies-not-dependent-ext*[*OF li-ext-A*] **by** *contradiction*
qed

Finally the theorem:

lemma *eq-cardinality-basis'*:
assumes A: *basis A* **and** B: *basis B*
shows *card A* = *card B*

by (*metis A B basis-not-infinite eq-cardinality-basis*)

end
end

theory *Isomorphism*
imports *Dimension*
begin

11 Isomorphism

The *types* keyword seems to be replaced by *type-synonym* in the Isabelle 2011 release.

The following definition of *vector* has been obtained from the *AFP*, where a similar one is defined over *real*, instead of $'a$, for defining the Cauchy-Schwarz Inequality http://afp.sourceforge.net/entries/Cauchy.shtml.

22-07-2011: JE: For some time I thought that many of the proofs required the vector spaces to be non empty (not of dimension zero). This is why one can meet a lot of premises of the type $(0::'a) < n$ or about the dimension being non zero (all these premises are now enclosed between comments). After a closer look I could remove each of these premises and make everything general for every finite dimension.

types $'a$ *vector* = (*nat* => $'a$) * *nat*

definition
ith :: $'a$ *vector* => *nat* => $'a$
where *ith v i* = *fst v i*

definition
vlen :: $'a$ *vector* => *nat*
where *vlen v* = *snd v*

Before getting into the definition of the vector space K_n, we introduce a generic lemma that states that the decomposition of an element $x \in$ *carrier* V as a linear combination of the elements of a given basis is unique.

The lemma requires the basis X to be finite, because otherwise there would be a linear combination of the infinite number of elements of the basis equal to zero, but the *finsum* of an infinite set is undefined, and thus we cannot complete the proof.

context *abelian-group*
begin

Some previous lemmas about addition in abelian monoids.

lemma
  add-minus-add-minus:
  assumes *a*: $a \in$ *carrier G*
  and *b*: $b \in$ *carrier G*
  and *c*: $c \in$ *carrier G*
  shows $a \oplus b \ominus c = a \oplus (b \ominus c)$
proof –
  have $a \oplus b \ominus c = a \oplus b \oplus \ominus c$
    using *minus-eq* [*OF a-closed* [*OF a b*] *c*] **.**
  also have ... $= a \oplus (b \oplus \ominus c)$
    using *a-assoc* [*OF a b a-inv-closed* [*OF c*]] **.**
  also have ... $= a \oplus (b \ \ominus c)$
    unfolding *minus-eq* [*symmetric*, *OF b c*] **..**
  finally show *?thesis* **.**
qed

lemma
  minus-add-minus-minus:
  assumes *a*: $a \in$ *carrier G*
  and *b*: $b \in$ *carrier G*
  and *c*: $c \in$ *carrier G*
  shows $a \ominus (b \oplus c) = a \ominus b \ominus c$
proof –
  have $a \ominus (b \oplus c) = a \oplus \ominus (b \oplus c)$
    using *minus-eq* [*OF a a-closed* [*OF b c*]] **.**
  thm *minus-add*
  also have ... $= a \oplus (\ominus b \oplus \ominus c)$
    unfolding *minus-add* [*OF b c*] **..**
  also have ... $= a \oplus \ominus b \oplus \ominus c$
    using *a-assoc* [*symmetric*, *OF a a-inv-closed* [*OF b*] *a-inv-closed* [*OF c*]] **.**
  also have ... $= a \ominus b \oplus \ominus c$
    unfolding *minus-eq* [*symmetric*, *OF a b*] **..**
  also have ... $= a \ominus b \ominus c$
    using *minus-eq* [*symmetric*, *OF minus-closed* [*OF a b*] *c*] **.**
  finally show *?thesis* **.**
qed

lemma
  add-minus-add-minus-add-minus:
  assumes *a*: $a \in$ *carrier G*
  and *b*: $b \in$ *carrier G*
  and *c*: $c \in$ *carrier G*
  and *d*: $d \in$ *carrier G*
  shows $a \oplus b \ominus (c \oplus d) = a \ominus c \oplus (b \ominus d)$
proof –
  have $a \oplus b \ominus (c \oplus d) = a \oplus b \oplus \ominus (c \oplus d)$
    using *minus-eq* [*OF a-closed* [*OF a b*] *a-closed* [*OF c d*]] **.**
  also have ... $= a \oplus (b \oplus \ominus (c \oplus d))$
    using *a-assoc* [*OF a b a-inv-closed* [*OF a-closed* [*OF c d*]]] **.**

also have ... = $a \oplus (b \ominus (c \oplus d))$
unfolding *minus-eq* [*symmetric*, *OF b a-closed* [*OF c d*]] ..
also have ... = $a \oplus (b \ominus c \ominus d)$
unfolding *minus-add-minus-minus* [*OF b c d*] ..
also have ... = $a \oplus (b \oplus \ominus c \oplus \ominus d)$
unfolding *minus-eq* [*OF minus-closed* [*OF b c*] *d*]
unfolding *minus-eq* [*OF b c*] ..
also have ... = $a \oplus (\ominus c \oplus b \oplus \ominus d)$
unfolding *a-comm* [*OF b a-inv-closed* [*OF c*]] ..
also have ... = $a \oplus (\ominus c \oplus (b \oplus \ominus d))$
unfolding *a-assoc* [*OF a-inv-closed* [*OF c*] *b a-inv-closed* [*OF d*]] ..
also have ... = $a \oplus \ominus c \oplus (b \oplus \ominus d)$
unfolding *a-assoc* [*OF a a-inv-closed* [*OF c*] *a-closed* [*OF b a-inv-closed* [*OF d*]]] ..
also have ... = $a \ominus c \oplus (b \ominus d)$
unfolding *minus-eq* [*OF a c*]
unfolding *minus-eq* [*OF b d*] ..
finally show *?thesis* .
qed

corollary *add-minus-add-minus-add-minus-comm*:
assumes *a*: $a \in$ *carrier G*
and *b*: $b \in$ *carrier G*
and *c*: $c \in$ *carrier G*
and *d*: $d \in$ *carrier G*
shows $a \oplus b \ominus (c \oplus d) = b \ominus d \oplus (a \ominus c)$
proof –
have $a \oplus b \ominus (c \oplus d) = a \ominus c \oplus (b \ominus d)$
using *add-minus-add-minus-add-minus* [*OF a b c d*] .
also have ... = $(b \ominus d) \oplus (a \ominus c)$
unfolding *a-comm* [*OF minus-closed* [*OF a c*] *minus-closed* [*OF b d*]] ..
finally show *?thesis* .
qed

lemma *finsum-minus-eq*:
assumes *fin-A*: *finite A*
and *f-PI*: $f \in A \to$ *carrier G*
shows $\ominus$ *finsum G f A* = *finsum G* ($\lambda x. \ominus f\ x$) *A*
using *fin-A f-PI* **proof** (*induct*)
case *empty*
show *?case* **by** *simp*
next
case (*insert a A*)
have *f-PI*: $f \in A \to$ *carrier G*
and *fa*: $f\ a \in$ *carrier G*
and *minus-f-PI*: $(\lambda x. \ominus f\ x) \in A \to$ *carrier G*
and *minus-fa*: $\ominus f\ a \in$ *carrier G*
using *insert* (*4*) **unfolding** *Pi-def* **by** *simp-all*
have *fG*: *finsum G f A* $\in$ *carrier G*

by (*rule finsum-closed* [*OF insert* (*1*) *f-PI*])
show *?case*
unfolding *finsum-insert* [*OF insert* (*1*, *2*) *f-PI*, *OF fa*]
unfolding *finsum-insert* [*OF insert* (*1*, *2*) *minus-f-PI*, *OF minus-fa*]
unfolding *minus-add* [*OF fa fG*]
unfolding *insert.hyps* (*3*) [*OF f-PI*] **..**
qed

end

context *vector-space*
begin

The following function should replace to *coefficients-function*; the problem with *coefficients-function* is that it does not impose any condition over functions out of their domain, *carrier* V; thus, we cannot prove that two coefficient functions which are equal over their corresponding domain (the basis X) are equal. We have to impose an additional restriction that the function out of its domain is equal to **0**

end

11.1 Definition of $\mathbb{K}^n$

context *field*
begin

The following definition represents the carrier set of the vector space. Note that the type variable is now *'a*, so we define only the following concepts over the field of the coefficients.

— Seleccionamos un representante cannico para cada elemento, haciendo que todas las coordenadas sean cero por encima de la dimensin del espacio vectorial

— Adems, debemos asegurar que la dimensin del vector, o la longitud del mismo, sea igual al nmero de componenetes en el que estamos interesados; sino perderamos la inyectividad de algunas operaciones

— Hay que tener en cuenta que en una lista de 1 elemento (por ejemplo, los elementos del carrier de K1) nos interesa nicamente el elemento en la posicin 0, de ah que nos interesen los elementos con vlen = n - 1;

Para los elementos en *K-n-carrier A* (*0*::*'d*) debemos observar que su primera componente ser **0** y su segunda componente ser tambin *0*, lo que nos deja con un K_0 cuyo nico elemento es el *0*::*'c* de la estructura correspondiente (K_n).

definition *K-n-carrier* :: *'a set* => *nat* => (*'a vector*) *set*
where *K-n-carrier A n* = {*v*. ((∀ *i*<*n*. *ith v i* ∈ *A*)) ∧
(∀ *i*≥*n*. *ith v i* = **0**) ∧ (*vlen v* = (*n* − *1*))}

lemma *ith-closed*:
  assumes k: $k \in$ *K-n-carrier A n* **and** i: $i \in \{..<n\}$
  shows *ith k i* $\in A$
  using k
  unfolding *K-n-carrier-def* **using** i **by** *fast*

lemma *K-n-carrier-zero*:
  K-n-carrier A 0 $= \{v.\ (ith\ v\ 0 = \mathbf{0}) \wedge (\forall i{>}0.\ ith\ v\ i = \mathbf{0}) \wedge (vlen\ v = 0)\}$
  unfolding *K-n-carrier-def*
  by *rule* (*auto*, *case-tac i*, *force+*)

lemma *K-n-carrier-zero-ext*: *K-n-carrier A 0* $= \{(\lambda i.\ \mathbf{0},\ 0)\}$
  unfolding *K-n-carrier-zero ith-def vlen-def*
  by *auto* (*rule ext*, *metis gr0I*)

lemma *K-n-carrier-one*:
  K-n-carrier A 1 $= \{v.\ ith\ v\ 0 \in A \wedge (\forall i{\geq}1.\ ith\ v\ i = \mathbf{0}) \wedge (vlen\ v = 0)\}$
  unfolding *K-n-carrier-def* **by** *auto*

definition
  K-n-add :: *nat => 'a vector => 'a vector => 'a vector* (**infixr** $\oplus$ı *65*)
  where *K-n-add n* $= (\lambda v\ w.\ ((\lambda i.\ ith\ v\ i \oplus_R ith\ w\ i),\ n - 1))$

lemma *K-n-add-zero*:
  shows *K-n-add 0* $= (\lambda v\ w.\ ((\lambda i.\ ith\ v\ i \oplus_R ith\ w\ i),\ 0))$
  using *K-n-add-def* [*of 0*] **by** *simp*

definition *K-n-mult* :: *nat => 'a vector => 'a vector => 'a vector*
  where *K-n-mult n* $= (\lambda v\ w.\ ((\lambda i.\ ith\ v\ i \otimes_R\ ith\ w\ i),\ n - 1))$

lemma *K-n-mult-zero*:
  shows *K-n-mult 0* $= (\lambda v\ w.\ ((\lambda i.\ ith\ v\ i \otimes_R ith\ w\ i),\ 0))$
  using *K-n-mult-def* **by** *auto*

definition *K-n-zero* :: *nat => 'a vector*
  where *K-n-zero n* $= ((\lambda i.\ \mathbf{0}_R),\ n - 1)$

lemma *K-n-zero-zero*:
  shows *K-n-zero 0* $= ((\lambda i.\ \mathbf{0}_R),\ 0)$
  using *K-n-zero-def* **by** *auto*

definition *K-n-one* :: *nat => 'a vector*
  where *K-n-one n* $= ((\lambda i.\ \mathbf{1}_R),\ n - 1)$

Actually, in the following case, one should be equal to zero

lemma *K-n-one-zero*:
  shows *K-n-one 0* $= ((\lambda i.\ \mathbf{1}_R),\ 0)$
  using *K-n-one-def* **by** *auto*

We are now forced to define also operations *K-n-mult* and *K-n-one* for our *abelian group* K^n. This is due to the fact that the abelian group predicate in the Algebra Library is defined over rings, and even if we have no interest in using that operations (they are not required to prove that an algebraic structure is an abelian group), they must be defined somehow. In our case this is not a major problem, since they can be defined just following the previous definitions of *K-n-zero* and *K-n-add*.

```
definition K-n :: nat => 'a vector ring
  where
  K-n n = (| carrier = K-n-carrier (carrier R) n,
             mult = (λv w. K-n-mult n v w),
             one = K-n-one n,
             zero = K-n-zero n,
             add = (λv w. K-n-add n v w)|)

lemma abelian-group-K-n:
  shows abelian-group (K-n n)
  unfolding K-n-def
proof (intro abelian-groupI)
  let ?K-n = (| carrier = K-n-carrier (carrier R) n,
             mult = (λv w. K-n-mult n v w),
             one = K-n-one n,
             zero = K-n-zero n,
             add = (λv w. K-n-add n v w)|)
  fix x y
  assume x: x ∈ carrier ?K-n and y: y ∈ carrier ?K-n
  show x ⊕?K-n y ∈ carrier ?K-n
    using x y
    unfolding K-n-carrier-def
    unfolding K-n-add-def
    unfolding ith-def vlen-def by auto
next
  let ?K-n = (| carrier = K-n-carrier (carrier R) n,
     mult = (λv w. K-n-mult n v w),
     one = K-n-one n,
     zero = K-n-zero n,
     add = (λv w. K-n-add n v w)|)
  show 0?K-n ∈ carrier ?K-n
    unfolding K-n-carrier-def
    unfolding K-n-zero-def
    unfolding ith-def vlen-def by auto
next
  let ?K-n = (| carrier = K-n-carrier (carrier R) n,
     mult = (λv w. K-n-mult n v w),
     one = K-n-one n,
     zero = K-n-zero n,
     add = (λv w. K-n-add n v w)|)
  fix x y z
  assume x: x ∈ carrier ?K-n and y: y ∈ carrier ?K-n and z: z ∈ carrier ?K-n
```

```
  show x ⊕?K-n y ⊕?K-n z = x ⊕?K-n (y ⊕?K-n z)
    using x y z
    unfolding K-n-carrier-def
    unfolding K-n-add-def
    unfolding ith-def vlen-def
  proof (auto)
    assume x1: ∀ i<n. fst x i ∈ carrier R and y1: ∀ i<n. fst y i ∈ carrier R
      and z1: ∀ i<n. fst z i ∈ carrier R
    assume x2: ∀ i≥n. fst x i = 0 and y2: ∀ i≥n. fst y i = 0 and z2: ∀ i≥n. fst
z i = 0
    show (λi. fst x i ⊕ fst y i ⊕ fst z i) = (λi. fst x i ⊕ (fst y i ⊕ fst z i))
    proof (rule ext)
      fix i
      show fst x i ⊕ fst y i ⊕ fst z i = fst x i ⊕ (fst y i ⊕ fst z i)
      proof (cases i < n)
        case True
        show ?thesis using x1 y1 z1 using True by (metis a-assoc)
      next
        case False
        show ?thesis using x2 y2 z2 using False
          by (metis add.one-closed cring.cring-simprules(16)
            is-cring less-or-eq-imp-le linorder-neqE-nat)
      qed
    qed
  qed
next
  let ?K-n = (| carrier = K-n-carrier (carrier R) n,
    mult = (λv w. K-n-mult n v w),
    one = K-n-one n,
    zero = K-n-zero n,
    add = (λv w. K-n-add n v w)|)
  fix x y
  assume x: x ∈ carrier ?K-n and y: y ∈ carrier ?K-n
  show x ⊕?K-n y = y ⊕?K-n x
    using x y
    unfolding K-n-carrier-def
    unfolding K-n-add-def
    unfolding ith-def vlen-def apply auto
  proof (rule ext)
    fix i
    assume x1: ∀ i<n. fst x i ∈ carrier R and y1: ∀ i<n. fst y i ∈ carrier R
    assume x2: ∀ i≥n. fst x i = 0 and y2: ∀ i≥n. fst y i = 0
    show fst x i ⊕ fst y i = fst y i ⊕ fst x i
    proof (cases i < n)
      case True
      show ?thesis using x1 y1 using True by (metis a-comm)
    next
      case False
      show ?thesis using x2 y2 using False
```

by (*metis less-or-eq-imp-le linorder-neqE-nat*)
qed
qed
next
let *?K-n* = (| *carrier* = *K-n-carrier* (*carrier R*) *n*,
mult = ($\lambda v\ w$. *K-n-mult n v w*),
one = *K-n-one n*,
zero = *K-n-zero n*,
add = ($\lambda v\ w$. *K-n-add n v w*)|)
fix *x*
assume *x*: $x \in$ *carrier ?K-n*
show $\mathbf{0}_{?K\text{-}n} \oplus_{?K\text{-}n} x = x$
using *x*
unfolding *K-n-carrier-def*
unfolding *K-n-add-def*
unfolding *K-n-zero-def*
unfolding *ith-def vlen-def*
apply *auto*
apply (*subst* (*2*) *surjective-pairing* [*of x*])
apply *simp*
apply (*rule ext*)
by (*metis add.l-one add.one-closed le-eq-less-or-eq linorder-neqE-nat*)
next
let *?K-n* = (| *carrier* = *K-n-carrier* (*carrier R*) *n*,
mult = ($\lambda v\ w$. *K-n-mult n v w*),
one = *K-n-one n*,
zero = *K-n-zero n*,
add = ($\lambda v\ w$. *K-n-add n v w*)|)
fix *x*
assume *x*: $x \in$ *carrier ?K-n*
show $\exists\, y \in$ *carrier ?K-n*. $y \oplus_{?K\text{-}n} x = \mathbf{0}_{?K\text{-}n}$
apply (*rule bexI* [*of* - ((λi. $\ominus$ (*fst x i*)), $n - $ *Suc 0*)])
using *x*
unfolding *K-n-carrier-def*
unfolding *K-n-add-def*
unfolding *K-n-zero-def*
unfolding *ith-def vlen-def*
apply *auto*
apply (*rule ext*)
by (*metis add.l-inv add.one-closed le-eq-less-or-eq linorder-neqE-nat*)
qed

corollary *abelian-monoid-K-n*:
shows *abelian-monoid* (*K-n n*)
using *abelian-group-K-n* [*of n*]
unfolding *abelian-group-def* **..**

We are later to consider *K-n* like one abelian group over which R gives place to a vector space. We must define first the scalar product between both structures.

definition
  K-n-scalar-product :: $'a => 'a$ *vector* $=> 'a$ *vector*
  (**infixr** $\odot$ *65*)
  where $a \odot b = (\lambda n{::}nat.\ a \otimes_R ith\ b\ n,\ vlen\ b)$

lemma *K-n-scalar-product-closed*:
  assumes a: $a \in carrier\ R$
  and b: $b \in carrier\ (K\text{-}n\ n)$
  shows $a \odot b \in carrier\ (K\text{-}n\ n)$
  unfolding *K-n-scalar-product-def*
  using *a b*
  unfolding *ith-def vlen-def K-n-def K-n-carrier-def* **by** *simp*

lemma *field-R*: *field R*
  by (*metis cring-fieldI field-Units*)

lemma
  vector-space-K-n:
  shows *vector-space R* (*K-n n*) (*op* $\odot$)
  unfolding *K-n-def*
proof (*intro vector-spaceI*)
  show *field R* **using** *field-R* **.**
  show *abelian-group* (|*carrier* = *K-n-carrier* (*carrier R*) *n*,
      mult = *K-n-mult n*, *one* = *K-n-one n*,
      zero = *K-n-zero n*, *add* = *K-n-add n*|)
    using *abelian-group-K-n* [*of n*]
    unfolding *K-n-def* **.**
next
  let *?K-n* = (|*carrier* = *K-n-carrier* (*carrier R*) *n*,
    mult = *K-n-mult n*, *one* = *K-n-one n*,
    zero = *K-n-zero n*, *add* = *K-n-add n*|)
  fix x :: $'a$ *vector* **and** a :: $'a$
  assume x: $x \in$ *carrier ?K-n*
  assume a: $a \in carrier\ R$
  show $a \odot x \in$ *carrier ?K-n*
    using *x a*
    unfolding *K-n-scalar-product-def*
    unfolding *K-n-carrier-def*
    unfolding *vlen-def ith-def* **by** *simp*
next
  let *?K-n* = (|*carrier* = *K-n-carrier* (*carrier R*) *n*,
    mult = *K-n-mult n*, *one* = *K-n-one n*,
    zero = *K-n-zero n*, *add* = *K-n-add n*|)
  fix *x a b*
  assume x: $x \in$ *carrier ?K-n*
  assume a: $a \in carrier\ R$ **and** b: $b \in carrier\ R$
  show $a \otimes b \odot x = a \odot b \odot x$
    using *x a b*

unfolding *K-n-scalar-product-def*
unfolding *K-n-carrier-def*
unfolding *vlen-def ith-def* **apply** *auto* **apply** (*rule ext*)
by (*metis add.one-closed le-refl linorder-neqE-nat m-assoc nat-less-le*)

next

let *?K-n* = (|*carrier* = *K-n-carrier* (*carrier R*) *n*,
mult = *K-n-mult n*, *one* = *K-n-one n*,
zero = *K-n-zero n*, *add* = *K-n-add n*|)
fix *x*
assume *x*: $x \in$ *carrier ?K-n*
show $\mathbf{1} \odot x = x$
using *x*
unfolding *K-n-scalar-product-def*
unfolding *K-n-carrier-def*
unfolding *vlen-def ith-def*
apply (*subst* (*3*) *surjective-pairing* [*of x*])
apply *auto*
apply (*rule ext*)
by (*metis add.one-closed l-one less-or-eq-imp-le linorder-neqE-nat*)

next

let *?K-n* = (|*carrier* = *K-n-carrier* (*carrier R*) *n*,
mult = *K-n-mult n*, *one* = *K-n-one n*,
zero = *K-n-zero n*, *add* = *K-n-add n*|)
fix *x y a*
assume *x*: $x \in$ *carrier ?K-n* **and** *y*: $y \in$ *carrier ?K-n*
assume *a*: $a \in$ *carrier R*
show $a \odot (x \oplus_{?K\text{-}n} y) = (a \odot x) \oplus_{?K\text{-}n} (a \odot y)$
using *x y a*
unfolding *K-n-scalar-product-def*
unfolding *K-n-carrier-def*
unfolding *K-n-add-def*
unfolding *vlen-def ith-def*
apply *auto*
apply (*rule ext*)
by (*metis add.one-closed less-or-eq-imp-le linorder-neqE-nat r-distr*)

next

let *?K-n* = (|*carrier* = *K-n-carrier* (*carrier R*) *n*,
mult = *K-n-mult n*, *one* = *K-n-one n*,
zero = *K-n-zero n*, *add* = *K-n-add n*|)
fix *x a b*
assume *x*: $x \in$ *carrier ?K-n*
assume *a*: $a \in$ *carrier R* **and** *b*: $b \in$ *carrier R*
show $(a \oplus b) \odot x = (a \odot x) \oplus_{?K\text{-}n} (b \odot x)$
using *x a b*
unfolding *K-n-scalar-product-def*
unfolding *K-n-carrier-def*
unfolding *K-n-add-def*
unfolding *vlen-def ith-def* **apply** *auto* **apply** (*rule ext*)
by (*metis add.one-closed l-distr le-eq-less-or-eq linorder-neqE-nat*)

qed

11.2 Canonical basis of $\mathbb{K}^n$:

In the following section we introduce the elements that generate the canonical basis of the vector space *K-n n* and prove some properties of them.

The elements of the canonical basis of *K-n* are the following ones:

definition *x-i* :: *nat* => *nat* => *'a vector*
  where *x-i j n* = ((λ*i*. *if i* = *j then* **1** *else* **0**), *n* − *1*)

The elements *x-i* are part of the *carrier* (*K-n n*).

lemma
  x-i-closed:
  assumes *j-l-n*: *j* < *n*
  shows *x-i j n* ∈ *carrier* (*K-n n*)
  unfolding *K-n-def*
  unfolding *K-n-carrier-def*
  unfolding *ith-def vlen-def x-i-def* **using** *j-l-n* **by** *auto*

Any two elements of the basis are different:

lemma *x-i-ne-x-j*:
  assumes *i-ne-j*: *i* ≠ *j*
  shows *x-i i n* ≠ *x-i j n*
proof (*rule ccontr*, *simp*)
  assume *eq*: *x-i i n* = *x-i j n*
  have *fst* (*x-i i n*) *i* = **1**
    unfolding *x-i-def* **by** *simp*
  moreover have *fst* (*x-i j n*) *i* = **0**
    unfolding *x-i-def* **using** *i-ne-j* **by** *force*
  ultimately show *False* **using** *eq* **by** *simp*
qed

In the following lemma we can even omit the premise of i being smaller than n, so the result is also true for vectors which are not part of the canonical basis. It claims that an element of the canonical basis is not equal to $\mathbf{0}_{K\text{-}n\ n}$

lemma *x-i-ne-zero*:
  shows *x-i i n* ≠ $\mathbf{0}_{K\text{-}n\ n}$
proof (*rule ccontr*, *simp*)
  assume *eq*: *x-i i n* = $\mathbf{0}_{K\text{-}n\ n}$
  have *fst* (*x-i i n*) *i* = **1**
    unfolding *x-i-def* **by** *simp*
  moreover have *fst* ($\mathbf{0}_{K\text{-}n\ n}$) *i* = **0**
    unfolding *K-n-def K-n-zero-def* **by** *force*
  ultimately show *False* **using** *eq* **by** *simp*
qed

end

context *vector-space*
begin

lemma
coefficients-function-Pi:
assumes x: $x \in$ *carrier V*
and *cf-f*: $f \in$ *coefficients-function A*
shows $f\ x \in$ *carrier K*
using *cf-f*
unfolding *coefficients-function-def* **by** *auto*

end

context *abelian-group*
begin

lemma
finsum-twice:
assumes f: $f \in \{i,j\} \rightarrow$ *carrier G*
and *i-ne-j*: $i \neq j$
shows *finsum G f* $\{i, j\} = f\ i \oplus f\ j$
proof −
have *finsum G f* $\{i, j\} = f\ i \oplus$ *finsum G f* $\{j\}$
apply (*rule finsum-insert*) **using** *i-ne-j f* **by** *auto*
also have ... $= f\ i \oplus (f\ j \oplus$ *finsum G f* $\{\})$
using *finsum-insert* [*of* $\{\}$ *j f*] **using** *f* **by** *fastsimp*
also have ... $= f\ i \oplus f\ j$
unfolding *finsum-empty* **using** *f* **by** *force*
finally show *?thesis* **.**
qed

end

context *comm-monoid*
begin

lemma *mult-if*:
shows $(\lambda k.\ x \otimes (\mathit{if}\ k = i\ \mathit{then}\ y\ \mathit{else}\ z)) = (\lambda k.\ \mathit{if}\ k = i\ \mathit{then}\ x \otimes y\ \mathit{else}\ x \otimes z)$
by (*rule ext*, *auto*)

end

lemma
fun-eq-contr:
assumes *fg*: $f = g$

and *x*: $f\ x \neq g\ x$
shows *False* **by** (*metis fg x*)

context *abelian-monoid*
begin

lemma
finsum-singleton-set:
assumes *f*: $f\ a \in carrier\ G$
shows $finsum\ G\ f\ \{a\} = f\ a$
using *finsum-insert* [*of* {} *a f*]
using *finsum-empty* **using** *f* **by** *force*

end

context *field*
begin

lemma *comm-monoid-R*: *comm-monoid R* **by** *intro-locales*

lemma *abelian-monoid-R*: *abelian-monoid R* **by** *intro-locales*

Some previous about the linear independece of the elements of the canonical basis:

lemma *x-i-li*:
assumes *j-l-n*: $j < n$
shows *vector-space.linear-independent R* (*K-n n*) (*op* $\odot$) {(*x-i j n*)}
proof (*unfold vector-space.linear-independent-def* [*OF vector-space-K-n*], *intro conjI*)
interpret *vector-space R K-n n op* $\odot$ **using** *vector-space-K-n* **.**
show *good-set* {*x-i j n*}
unfolding *good-set-def*
using *x-i-closed* [*OF j-l-n*] **by** *blast*
show $\forall f.\ f \in$ *coefficients-function* (*carrier* (*K-n n*)) $\wedge$
linear-combination *f* {*x-i j n*} $= \mathbf{0}_{K\text{-}n\ n} \longrightarrow (\forall x\in\{x\text{-}i\ j\ n\}.\ f\ x = \mathbf{0})$
proof (*rule*+, *erule conjE*)
fix *f x*
assume *f*: $f \in$ *coefficients-function* (*carrier* (*K-n n*))
and *f1*: *linear-combination f* {*x-i j n*} $= \mathbf{0}_{K\text{-}n\ n}$
and $x \in \{x\text{-}i\ j\ n\}$
hence *x*: $x = x\text{-}i\ j\ n$ **by** *fast*
have $\mathbf{0}_{K\text{-}n\ n} =$ *linear-combination f* {*x-i j n*}
using *f1* [*symmetric*] **.**
also have *linear-combination f* {*x-i j n*} $= f\ (x\text{-}i\ j\ n) \odot (x\text{-}i\ j\ n)$
unfolding *linear-combination-def*
apply (*rule abelian-monoid.finsum-singleton-set* [*OF abelian-monoid-K-n* [*of* *n*]])
apply (*rule K-n-scalar-product-closed*)
using *f x-i-closed* [*OF j-l-n*]
unfolding *coefficients-function-def* **by** *fast*+

finally have *zero*: $\mathbf{0}_{K\text{-}n\ n} = f\ (x\text{-}i\ j\ n) \odot (x\text{-}i\ j\ n)$ **.**
show $f\ x = \mathbf{0}$
proof (*rule mult-zero-uniq* [*OF x-i-closed* [*OF j-l-n*]])
show $x\text{-}i\ j\ n \neq \mathbf{0}_{K\text{-}n\ n}$
by (*rule x-i-ne-zero* [*of j n*])
show $f\ x \in$ *carrier R* **unfolding** *x* **using** *f x-i-closed* [*OF j-l-n*]
unfolding *coefficients-function-def* **by** *fast+*
show $f\ x \odot x\text{-}i\ j\ n = \mathbf{0}_{K\text{-}n\ n}$ **unfolding** *x* **by** (*rule zero* [*symmetric*])
qed
qed
qed

Any two different elements of the canonical basis are linearly independent:

lemma *x-i-x-j-li*:
assumes *j-l-n*: $j < n$
and *i-l-n*: $i < n$
and *i-ne-j*: $i \neq j$
shows *vector-space.linear-independent R* (*K-n n*) (*op* $\odot$) {(*x-i i n*), (*x-i j n*)}
proof −
interpret *vector-space R K-n n op* $\odot$ **using** *vector-space-K-n* **.**
show *?thesis*
proof (*unfold linear-independent-def*, *rule*)
show *vector-space.good-set* (*K-n n*) {*x-i i n*, *x-i j n*}
unfolding *good-set-def*
using *x-i-closed* [*OF j-l-n*] *x-i-closed* [*OF i-l-n*] **by** *blast*
show $\forall f.\ f \in$ *coefficients-function* (*carrier* (*K-n n*)) $\wedge$
linear-combination f {*x-i i n*, *x-i j n*} $= \mathbf{0}_{K\text{-}n\ n} \longrightarrow (\forall x \in \{x\text{-}i\ i\ n,\ x\text{-}i\ j\ n\}.\ f$
$x = \mathbf{0})$
proof *auto*
fix *f*
assume *f*: $f \in$ *coefficients-function* (*carrier* (*K-n n*))
and *lc*: *linear-combination f* {*x-i i n*, *x-i j n*} $= \mathbf{0}_{K\text{-}n\ n}$
have *fxii*: $f\ (x\text{-}i\ i\ n) \in$ *carrier R* **and** *fxij*: $f\ (x\text{-}i\ j\ n) \in$ *carrier R*
using *fx-in-K* [*OF x-i-closed* [*OF i-l-n*] *f*]
using *fx-in-K* [*OF x-i-closed* [*OF j-l-n*] *f*] **by** *fast+*
show $f\ (x\text{-}i\ i\ n) = \mathbf{0}$
proof (*rule ccontr*)
assume *xii*: $f\ (x\text{-}i\ i\ n) \neq \mathbf{0}$
have $\mathbf{0}_{K\text{-}n\ n} =$ *linear-combination f* {*x-i i n*, *x-i j n*}
by (*rule lc* [*symmetric*])
also have *linear-combination f* {*x-i i n*, *x-i j n*} =
$(f\ (x\text{-}i\ i\ n) \odot (x\text{-}i\ i\ n)) \oplus_{K\text{-}n\ n} (f\ (x\text{-}i\ j\ n) \odot (x\text{-}i\ j\ n))$
unfolding *linear-combination-def*
apply (*rule finsum-twice* [*of* ($\lambda i.\ f\ i \odot i$) *x-i i n x-i j n*])
using *fx-x-in-V* [*OF - f*]
using *x-i-closed* [*OF i-l-n*] *x-i-closed* [*OF j-l-n*]
using *x-i-ne-x-j* [*OF i-ne-j*, *of n*]
unfolding *x-i-def* **by** *simp-all*
also have ... $= ((\lambda k.\ \textit{if}\ k = i\ \textit{then}\ f\ (x\text{-}i\ i\ n)\ \textit{else}\ \mathbf{0})\ ,\ n - 1) \oplus_n$

```
((λk. if k = j then f (x-i j n) else 0), n − 1)
apply (subst (2 4) x-i-def)
apply (unfold K-n-scalar-product-def)
unfolding ith-def vlen-def apply simp
unfolding K-n-def apply auto
unfolding comm-monoid.mult-if [OF comm-monoid-R]
unfolding r-one [OF fxii] r-one [OF fxij] r-null [OF fxii] r-null [OF fxij]
..
also have ... = ((λk. if k = i then f (x-i i n)
  else if k = j then f (x-i j n) else 0) , n − 1)
  unfolding K-n-add-def ith-def
  unfolding fst-conv
  apply rule+ using i-ne-j apply auto
  unfolding abelian-monoid.r-zero [OF abelian-monoid-R fxii]
  unfolding abelian-monoid.l-zero [OF abelian-monoid-R fxij] by fast+
finally have 0_{K-n n} = ((λk. if k = i then f (x-i i n)
  else if k = j then f (x-i j n) else 0) , n − 1) by fast
thus False
  unfolding K-n-def
  unfolding K-n-zero-def
  using xii i-ne-j
  apply simp
  apply (rule fun-eq-contr [of (λi. 0) (λk. if k = i
    then f (x-i i n) else if k = j then f (x-i j n) else 0) i])
  by simp-all
qed
next
fix f
assume f: f ∈ coefficients-function (carrier (K-n n))
  and lc: linear-combination f {x-i i n, x-i j n} = 0_{K-n n}
have fxii: f (x-i i n) ∈ carrier R and fxij: f (x-i j n) ∈ carrier R
  using fx-in-K [OF x-i-closed [OF i-l-n] f]
  using fx-in-K [OF x-i-closed [OF j-l-n] f] by fast+
show f (x-i j n) = 0
proof (rule ccontr)
  assume xii: f (x-i j n) ≠ 0
  have 0_{K-n n} = linear-combination f {x-i i n, x-i j n}
    by (rule lc [symmetric])
  also have linear-combination f {x-i i n, x-i j n} =
    (f (x-i i n) ⊙ (x-i i n)) ⊕_{K-n n} (f (x-i j n) ⊙ (x-i j n))
    unfolding linear-combination-def
    apply (rule finsum-twice [of (λi. f i ⊙ i) x-i i n x-i j n])
    using fx-x-in-V [OF - f]
    using x-i-closed [OF i-l-n] x-i-closed [OF j-l-n]
    using x-i-ne-x-j [OF i-ne-j, of n]
    unfolding x-i-def by simp-all
  also have ... = ((λk. if k = i then f (x-i i n) else 0) , n − 1)
    ⊕_n ((λk. if k = j then f (x-i j n) else 0), n − 1)
    apply (subst (2 4) x-i-def)
```

```
        apply (unfold K-n-scalar-product-def)
        unfolding ith-def vlen-def apply simp
        unfolding K-n-def apply auto
        unfolding comm-monoid.mult-if [OF comm-monoid-R]
        unfolding r-one [OF fxii] r-one [OF fxij] r-null [OF fxii] r-null [OF fxij]
..
      also have ... = ((λk. if k = i then f (x-i i n)
        else if k = j then f (x-i j n) else 0) , n − 1)
        unfolding K-n-add-def ith-def
        unfolding fst-conv
        apply rule+ using i-ne-j apply auto
        unfolding abelian-monoid.r-zero [OF abelian-monoid-R fxii]
          abelian-monoid.l-zero [OF abelian-monoid-R fxij] by fast+
      finally have 0_{K-n n} = ((λk. if k = i then f (x-i i n)
        else if k = j then f (x-i j n) else 0) , n − 1) by fast
      thus False
        unfolding K-n-def
        unfolding K-n-zero-def
        using xii i-ne-j
        apply simp
        apply (rule fun-eq-contr [
          of (λi. 0)
          (λk. if k = i then f (x-i i n) else if k = j then f (x-i j n) else 0) j])
        by simp-all
    qed
  qed
 qed
qed
```

We did not find a better way to define the elements of the canonical basis than accumulating them iteratively. In order to define them as a range, from *x-i 0 n* up to *x-i* $(n - 1)$ *n*, the underlying type, in this case *'a vector*, should be of sort "order" (which in general is not, only the elements of the basis have some notion of order.)

The following fuction iteratively joins all the elements of the form *x-i k n* in order to create the canonical basis of *K-n n*.

We have considered as a special case the situation where both indexes are equal to *0*. This case will give us the basis of *K-n 0*, which is the empty set. Note that a linear combination over an empty set is equal to $(\lambda i.\ \mathbf{0}_K,\ 0)$, which is the only element in *carrier (K-n 0)*.

```
fun canonical-basis-acc :: nat => nat => 'a vector set
  where
  canonical-basis-acc 0 0 = {}
  | canonical-basis-acc 0 n = {x-i 0 n}
  | canonical-basis-acc (Suc i) n
  = (if (Suc i < n) then
  insert (x-i (Suc i) n) (canonical-basis-acc i n) else {})
```

We now prove some lemmas trying to establish the relation between the elements of the form *x-i i n* and the ones in *canonical-basis-acc*.

lemma
  finite-canonical-basis-acc:
  shows *finite* (*canonical-basis-acc k n*)
  by (*induct k*, *induct n*, *auto*)

lemma
  canonical-basis-acc-closed:
  assumes *i-l-j*: $i < j$
  shows *canonical-basis-acc i j* $\subseteq$ *carrier* (*K-n j*)
  using *i-l-j* **using** *x-i-closed* **by** (*induct i*, *induct j*, *auto*)

The canonical basis in dimension n is given by all elements ranging from *x-i 0 n* up to *x-i* $(n - 1)$ *n*

definition *canonical-basis-K-n* :: *nat* => *'a vector set* **where**
  canonical-basis-K-n n = *canonical-basis-acc* $(n - 1)$ *n*

lemma
  canonical-basis-acc-insert:
  assumes *j-l-k*: $j < k$
  and *k-l-n*: $k < n$
  shows *x-i k n* $\notin$ *canonical-basis-acc j n*
  using *j-l-k k-l-n* **proof** (*induct j*)
  case *0*
  show *?case*
    unfolding *canonical-basis-acc.simps*
    using *0.prems* (*1*) **using** *x-i-ne-x-j* [*of 0 k n*] **by** (*cases n*, *auto*)
next
  case (*Suc j*)
  show *?case*
  proof (*cases* $j < k$)
    case *True*
    show *?thesis*
      apply (*subst canonical-basis-acc.simps*)
      using *Suc.hyps* [*OF True Suc.prems* (*2*)]
      using *Suc.prems*
      using *x-i-ne-x-j* [*of Suc j k n*]
      using *x-i-ne-x-j* [*of j k n*] **by** *force*
  next
    case *False*
    with *Suc.prems* **have** *False* **by** *linarith*
    thus *?thesis* **by** *fast*
  qed
qed

lemma
  card-canonical-basis-acc:
  assumes *k-le-n*: $k < n$

shows *card* (*canonical-basis-acc k n*) = *Suc k*
using *k-le-n*
proof (*induct k*)
case *0*
show *?case* **using** *0* **by** (*cases n*, *auto*)
next
case (*Suc k*)
have *k-l-n*: *k* < *n* **using** *Suc.prems* **by** *presburger*
show *?case*
apply (*subst canonical-basis-acc.simps*)
using *Suc.prems*
using *canonical-basis-acc-insert* [*OF* - *Suc.prems*, *of k*]
using *card.insert* [*OF finite-canonical-basis-acc* [*of k n*],
of x-i (*Suc k*) *n*]
using *Suc.hyps* [*OF k-l-n*] **by** *simp*
qed

end

lemma
n-minus-one-l-n:
assumes *n-g-0*: *0* < *n*
shows *n* − (*1*::*nat*) < *n*
by (*metis assms diff-Suc-1 gr0-implies-Suc lessI*)

context *field*
begin

The following lemma is true for dimension *0* thanks to the special case *canonical-basis-acc 0 0* = {} previously introduced:

lemma
canonical-basis-K-n-closed:

shows *canonical-basis-K-n n* ⊆ *carrier* (*K-n n*)
proof (*cases n*)
case *0*
show *?thesis*
unfolding *0*
unfolding *canonical-basis-K-n-def* **by** *simp*
next
case (*Suc n*)
show *?thesis*
unfolding *Suc canonical-basis-K-n-def*
by (*rule canonical-basis-acc-closed* [*OF n-minus-one-l-n*], *fast*)
qed

The following lemma is true for dimension *0* thanks to the special case *canonical-basis-acc 0 0* = {} previously introduced:

lemma

card-canonical-basis-K-n:

shows *card* (*canonical-basis-K-n n*) = *n*
proof (*cases n*)
case *0*
show *?thesis* **unfolding** *0*
unfolding *canonical-basis-K-n-def* **by** *simp*
next
case (*Suc n*)
show *?thesis* **unfolding** *Suc*
unfolding *canonical-basis-K-n-def*
using *card-canonical-basis-acc* [*OF n-minus-one-l-n* [*of Suc n*]] **by** *fastsimp*
qed

The following lemma does not even require to have a dimension greater than *0*.

lemma
finite-canonical-basis-K-n:

shows *finite* (*canonical-basis-K-n n*)
by (*metis canonical-basis-K-n-def finite-canonical-basis-acc*)

lemma
canonical-basis-acc-insert2:
assumes *j-le-k*: $j \leq k$
and *k-l-n*: $k < n$
shows *x-i j n* ∈ *canonical-basis-acc k n*
using *j-le-k k-l-n* **proof** (*induct k*)
case *0*
show *?case* **using** *0.prems* **by** (*cases n*, *auto*)
next
case (*Suc k*)
show *?case*
proof (*cases j* = *Suc k*)
case *False*
hence *j-le-k*: $j \leq k$ **using** *Suc.prems* (*1*) **by** *presburger*
have *k-l-n*: $k < n$ **using** *Suc.prems* (*2*) **by** *presburger*
show *?thesis*
using *Suc.hyps* [*OF j-le-k k-l-n*]
using *Suc.prems* (*2*) **by** *simp*
next
case *True*
show *?thesis*
apply (*subst canonical-basis-acc.simps*)
using *True*
using *Suc.prems* (*2*)
using *x-i-ne-x-j* [*of k Suc k n*] **by** (*cases k*, *auto*)
qed
qed

lemma
canonical-basis-K-n-elements:
assumes *j-in-n*: $j \in \{..<n\}$
shows *x-i j n* $\in$ *canonical-basis-K-n n*
proof (*cases n*)
case *0*
show *?thesis* **using** *j-in-n* **unfolding** *0* **by** *fast*
next
case (*Suc n*)
show *?thesis*
using *j-in-n*
unfolding *Suc*
unfolding *canonical-basis-K-n-def*
using *canonical-basis-acc-insert2* [*of j Suc n* $-$ *1 Suc n*] **by** *simp*
qed

lemma
canonical-basis-K-n-good-set:

shows *vector-space.good-set* (*K-n n*) (*canonical-basis-K-n n*)
proof (*unfold vector-space.good-set-def* [*OF vector-space-K-n*], *rule*)
show *finite* (*canonical-basis-K-n n*)
unfolding *canonical-basis-K-n-def*
by (*rule finite-canonical-basis-acc* [*of n* $-$ *1 n*])
show *canonical-basis-K-n n* $\subseteq$ *carrier* (*K-n n*)
by (*rule canonical-basis-K-n-closed*)
qed

end

JE: I have moved this definition to *Finite-Vector-Space*, so I remove it from here. This is to be checked with the other files.

11.3 Theorem on bijection

context *abelian-monoid*
begin

We need to prove the following lemma which is a generic version of the theorem *finsum-cong*:

$[\![A = B;\ (f \in B \to \mathit{carrier}\ G) = \mathit{True};\ \bigwedge i.\ i \in B =simp=> f\ i = g\ i]\!] \Longrightarrow$ *finsum G f A* = *finsum G g B* in the case where finite sums are defined over sets of different type, but isomorphic (in *finsum-cong* only the case where both sets of both finite sums are equal is considered).

lemma *finsum-cong''*:
assumes *fB*: *finite B*
and *bb*: *bij-betw h B A*

and *f*: $f : A \;{-}{>}\; carrier\ G$ **and** *g*: $g : B \;{-}{>}\; carrier\ G$
and *eq*: $(\bigwedge x.\ x \in B =simp{=}{>}\ g\ x = f\ (h\ x))$
shows $finsum\ G\ f\ A = finsum\ G\ g\ B$
proof −
have $finsum\ G\ g\ B = finsum\ G\ (f \circ h)\ B$
by (*rule finsum-cong*, *simp-all add*: *g*) (*rule eq*)
also have ... $= (\bigoplus x{\in}B.\ f\ (h\ x))$
proof (*rule finsum-cong*)
show $B = B$ **..**
show $\bigwedge i.\ i \in B =simp{=}{>}\ (f \circ h)\ i = f\ (h\ i)$ **by** *simp*
show $(f \circ h \in B \to carrier\ G) = True$
using *bij-betw-imp-funcset* [*OF bb*] **using** *f* **by** *auto*
qed
also have ... $= finsum\ G\ f\ (h\ \text{`}\ B)$
proof (*rule finsum-reindex* [*symmetric*])
show *finite B* **by** *fact*
show $f \in h\ \text{`}\ B \to carrier\ G$
using *f* **using** *bij-betw-imp-funcset* [*OF bb*] **by** *auto*
show *inj-on h B* **using** *bb* **unfolding** *bij-betw-def* **by** *fast*
qed
also have ... $= finsum\ G\ f\ A$
proof (*rule finsum-cong*)
show $h\ \text{`}B = A$ **using** *bb* **unfolding** *bij-betw-def* **by** *fast*
show $(f \in A \to carrier\ G) = True$ **using** *f* **by** *fast*
show $\bigwedge i.\ i \in A =simp{=}{>}\ f\ i = f\ i$ **by** *simp*
qed
finally show *?thesis* **by** *simp*
qed

end

lemma *n-notin-lessThan-n*: $(n{::}nat) \notin \{..{<}n\}$
by (*metis lessThan-iff less-not-refl3*)

context *field*
begin

lemma
snd-in-carrier:
assumes *x*: $x \in carrier\ (K\text{-}n\ n)$
shows $snd\ x = n - 1$
using *x*
unfolding *K-n-def K-n-carrier-def* **unfolding** *vlen-def* **by** *auto*

The following lemma gives a different representation of the elements of *K-n n*; this representation will be later used to prove that the elements of *K-n n* can be expressed as linear combinations of the elements of *canonical-basis-K-n n*.

lemma

x-in-carrier:
assumes *x*: $x \in$ *carrier* (*K-n n*)
shows $x = (\lambda i.$ *if* $i \in \{..<n\}$ *then fst x i else* $\mathbf{0}$, $n - 1)$
using *x*
unfolding *K-n-def K-n-carrier-def*
unfolding *ith-def vlen-def*
apply (*subst surjective-pairing*)
unfolding *snd-in-carrier* [*OF x*] **apply** *simp*
apply (*rule ext*)
by (*metis less-Suc-eq-le not-less-eq*)

The following lemma was later unused; every element can be "embedded" into a smaller dimension by means of "forgetful" function (we forget the last position of the vector).

lemma
K-n-carrier-embed:
assumes *x*: $x \in$ *carrier* (*K-n* (*Suc k*))
shows (($\lambda n.$ *if* $n \in \{..<k\}$ *then fst x n else* $\mathbf{0}$), $k - 1$) $\in$ *carrier* (*K-n k*)
using *x*
unfolding *K-n-def K-n-carrier-def ith-def vlen-def* **by** *auto*

Functions with only a single nonzero element can be expressed as scalar products of *x-i* elements.

lemma
singleton-function-x-i:
assumes *x*: $x \in$ *carrier R*
shows ($\lambda i.$ *if* $i = j$ *then x else* $\mathbf{0}$, $n - 1$) $= x \odot$ *x-i j n*
unfolding *K-n-scalar-product-def*
unfolding *x-i-def ith-def vlen-def fst-conv snd-conv*
apply (*rule*, *rule conjI*)
apply (*rule ext*) **using** *x* **by** *auto*

The following lemma is rather important, since it shows how to express any element in *carrier* (*K-n k*) in a canonical way: it proves that any element in *carrier* (*K-n k*) can be expressed as a finite sum of the elements *x-i j k*.

It is important to note that in the proof we have introduced an extra natural variable *n*, with $n \leq k$, which permits to prove the result by induction in *n* over the field *K-n k*.

If we do not use the extra variable *n* and we apply induction directly over *k*, the induction step will produce two different algebraic structures, *K-n k*, where the property holds, and *K-n* (*Suc k*), where the property must be proved, but then the induction hypothesis cannot be used.

lemma
lambda-finsum:
assumes *cl*: $\forall i \in \{..<n\}.$ $x\ i \in$ *carrier R*
and *n-le-k*: $n \leq k$

shows ($\lambda i.$ *if* $i \in \{..<n\}$ *then* $x\ i$ *else* $\mathbf{0}$, $k - 1$) =
finsum (*K-n k*) ($\lambda i.\ x\ i \odot$ *x-i i k*) $\{..<n\}$
using *cl n-le-k* **proof** (*induct n*)
case *0*
show *?case*
unfolding *lessThan-0*
unfolding *abelian-monoid.finsum-empty* [*OF abelian-monoid-K-n* [*of k*]]
unfolding *K-n-def K-n-zero-def* **by** *simp*
next
case (*Suc n*)
have *prem*: $\forall i \in \{..<n\}.\ x\ i \in$ *carrier R* **and** *prem2*: $n \leq k$
and *x-n*: $x\ n \in$ *carrier R*
and *hypo*: ($\lambda i.$ *if* $i \in \{..<n\}$ *then* $x\ i$ *else* $\mathbf{0}$, $k - 1$)
= ($\bigoplus_{K\text{-}n\ k} i \in \{..<n\}.\ x\ i \odot$ *x-i i k*)
using *Suc.prems Suc.hyps* **by** *simp-all*
show *?case*
proof −
have ($\bigoplus_{K\text{-}n\ k} i \in \{..<Suc\ n\}.\ x\ i \odot$ *x-i i k*)
= ($\bigoplus_{K\text{-}n\ k} i \in (insert\ n\ \{..<n\}).\ x\ i \odot$ *x-i i k*)
unfolding *lessThan-Suc* **..**
also have ... = ($x\ n \odot$ *x-i n k*)
$\oplus_{K\text{-}n\ k}$ ($\bigoplus_{K\text{-}n\ k} i \in \{..<n\}.\ x\ i \odot$ *x-i i k*)
proof (*rule abelian-monoid.finsum-insert* [*OF abelian-monoid-K-n*])
show *finite* $\{..<n\}$ **by** *simp*
show $n \notin \{..<n\}$ **by** *simp*
show ($\lambda i.\ x\ i \odot$ *x-i i k*) $\in \{..<n\} \rightarrow$ *carrier* (*K-n k*)
proof
fix *xa* **assume** *xa*: $xa \in \{..<n\}$
show $x\ xa \odot$ *x-i xa k* $\in$ *carrier* (*K-n k*)
unfolding *K-n-def K-n-carrier-def*
unfolding *K-n-scalar-product-def ith-def vlen-def x-i-def*
using *xa prem Suc.prems* (*2*) **by** *fastsimp*
qed
show $x\ n \odot$ *x-i n k* $\in$ *carrier* (*K-n k*)
unfolding *K-n-def K-n-carrier-def*
unfolding *K-n-scalar-product-def ith-def vlen-def x-i-def*
using *Suc.prems* (*1*) *Suc.prems* (*2*) **by** *simp*
qed
also have ... = ($x\ n \odot$ *x-i n k*)
$\oplus_{K\text{-}n\ k}$ ($\lambda i.$ *if* $i \in \{..<n\}$ *then* $x\ i$ *else* $\mathbf{0}$, $k - 1$)
unfolding *Suc.hyps* [*symmetric*, *OF prem prem2*] **..**
also have ... = ($\lambda i.$ *if* $i = n$ *then* $x\ n$ *else* $\mathbf{0}$, $k - 1$)
$\oplus_{K\text{-}n\ k}$ ($\lambda i.$ *if* $i \in \{..<n\}$ *then* $x\ i$ *else* $\mathbf{0}$, $k - 1$)
unfolding *x-i-def* [*of n k*]
unfolding *K-n-scalar-product-def ith-def vlen-def fst-conv snd-conv*
unfolding *mult-if* **unfolding** *r-null* [*OF x-n*] *r-one* [*OF x-n*] **..**

also have ... = $(\lambda i.\ (\mathit{if}\ i = n\ \mathit{then}\ x\ n\ \mathit{else}\ \mathbf{0})$
$\oplus (\mathit{if}\ i < n\ \mathit{then}\ x\ i\ \mathit{else}\ \mathbf{0}),\ k - \mathit{Suc}\ 0)$
unfolding *K-n-def K-n-add-def ith-def* **by** *simp*
also have ... = $((\lambda i.\ \mathit{if}\ i < (\mathit{Suc}\ n)\ \mathit{then}\ x\ i\ \mathit{else}\ \mathbf{0}),\ k - 1)$
proof (*rule*, *intro conjI*)
show $k - \mathit{Suc}\ 0 = k - 1$ **by** *simp*
show $(\lambda i.\ (\mathit{if}\ i = n\ \mathit{then}\ x\ n\ \mathit{else}\ \mathbf{0})$
$\oplus (\mathit{if}\ i < n\ \mathit{then}\ x\ i\ \mathit{else}\ \mathbf{0})) =$
$(\lambda i.\ \mathit{if}\ i < \mathit{Suc}\ n\ \mathit{then}\ x\ i\ \mathit{else}\ \mathbf{0})$
proof (*rule ext*)
fix *i* :: *nat*
show $(\mathit{if}\ i = n\ \mathit{then}\ x\ n\ \mathit{else}\ \mathbf{0})$
$\oplus (\mathit{if}\ i < n\ \mathit{then}\ x\ i\ \mathit{else}\ \mathbf{0}) =$
$(\mathit{if}\ i < \mathit{Suc}\ n\ \mathit{then}\ x\ i\ \mathit{else}\ \mathbf{0})$
proof (*cases* $i < \mathit{Suc}\ n$)
case *False*
thus *?thesis* **by** *simp*
next
case *True*
show *?thesis* **using** *True* **using** *Suc.prems* (*1*)
by (*cases* $i = n$, *auto*)
qed
qed
qed
finally show *?thesis* **by** *simp*
qed
qed

Now, as a corollary of the previous result, we obtain that any element of *K-n n* can be expressed as a finite sum of the elements of the form *x-i j n*.

lemma *lambda-finsum-n*:
assumes *cl*: $\forall i \in \{..<n\}.\ x\ i \in \mathit{carrier}\ R$
shows $(\lambda i.\ \mathit{if}\ i \in \{..<n\}\ \mathit{then}\ x\ i\ \mathit{else}\ \mathbf{0},\ n - 1) =$
finsum (*K-n n*) $(\lambda i.\ x\ i \odot x\text{-}i\ i\ n)\ \{..<n\}$
using *lambda-finsum* [*OF cl*, *of n*] **by** *fast*

Finally, we get the lemma that states tha any element of the set *K-n-carrier n* is a linear combination of elements of *canonical-basis-K-n n*:

lemma
K-n-carrier-finsum-x-i:
assumes *x*: $x \in \mathit{carrier}\ (K\text{-}n\ n)$
shows $x = \mathit{finsum}\ (K\text{-}n\ n)\ (\lambda j.\ \mathit{fst}\ x\ j \odot x\text{-}i\ j\ n)\ \{..<n\}$
apply (*subst x-in-carrier* [*OF x*])
apply (*rule lambda-finsum-n*)
using *x* **unfolding** *K-n-def K-n-carrier-def ith-def vlen-def*
by *force*

11.4 Bijection between basis:

In the following lemmas we try to establish an explicit bijection between the sets X, which is a basis of V, and the set *canonical-basis-K-n n*. This bijection will be later extended, by linearity, to a bijection between *carrier V* and *carrier (K-n n)*

lemma *canonical-basis-acc-eq-x-i*:
  assumes *x*: $x \in$ *canonical-basis-acc k n*
  and *k-l-n*: $k < n$
  shows $\exists j\in\{..<Suc\ k\}$. *x-i j n* $= x$
  using *x k-l-n*
proof (*induct k*)
  case *0* **thus** *?case* **unfolding** *canonical-basis-acc.simps* **by** (*cases n*, *auto*)
next
  case (*Suc k*)
  show *?case*
  proof (*cases x* = *x-i* (*Suc k*) *n*)
    case *False*
    have *k-l-n*: $k < n$ **and** *cb*: $x \in$ *canonical-basis-acc k n*
      and *hypo*: $\exists j\in\{..<(Suc\ k)\}$. *x-i j n* $= x$
      using *Suc.prems Suc.hyps False* **by** *simp-all*
    thus *?thesis* **by** *fastsimp*
  next
    case *True*
    show *?thesis*
      using *True* **by** *fast*
  qed
qed

corollary
  canonical-basis-acc-isom-x-i:
  assumes *x*: $x \in$ *canonical-basis-acc k n*
  and *k-l-n*: $k < n$
  shows $\exists!j\in\{..<Suc\ k\}$. $x =$ *x-i j n*
proof –
  obtain *j* :: *nat* **where** *j*: $j \in \{..<Suc\ k\}$ **and** *x*: $x =$ *x-i j n*
    using *canonical-basis-acc-eq-x-i* [*OF x k-l-n*] **by** *blast*
  show *?thesis*
  proof (*rule ex1I* [*of - j*], *rule conjI*)
    show $j \in \{..<Suc\ k\}$ **by** *fact*
    show $x =$ *x-i j n* **by** (*rule x*)
    fix *ja*
    assume *ja*: $ja \in \{..<Suc\ k\} \wedge x =$ *x-i ja n*
    show $ja = j$
      using *x ja* **unfolding** *x-i-def*
      by (*metis ja x x-i-ne-x-j*)
  qed
qed

corollary
canonical-basis-acc-isom-x-i2:
assumes *x*: $x \in$ *canonical-basis-acc k n*
and *k-l-n*: $k < n$
shows $\exists!j\in\{..<n\}$. $x = $ *x-i j n*
proof −
obtain *j* :: *nat* **where** *j*: $j \in \{..<Suc\ k\}$ **and** *x*: $x = $ *x-i j n*
using *canonical-basis-acc-eq-x-i* [*OF x k-l-n*] **by** *blast*
show *?thesis*
proof (*rule ex1I* [*of - j*], *rule conjI*)
show $j \in \{..<n\}$ **using** *j k-l-n* **by** *fastsimp*
show $x = $ *x-i j n* **by** (*rule x*)
fix *ja*
assume *ja*: $ja \in \{..<n\} \wedge x = $ *x-i ja n*
show $ja = j$
using *x ja* **unfolding** *x-i-def*
by (*metis ja x x-i-ne-x-j*)
qed
qed

lemma
canonical-basis-is-x-i:
assumes *x*: $x \in$ *canonical-basis-K-n n*

shows $\exists j\in\{..<n\}$. $x = $ *x-i j n*
using *x*
unfolding *canonical-basis-K-n-def*
using *canonical-basis-acc-eq-x-i* [*of x n − 1 n*] **by** (*cases n*, *auto*)

corollary
canonical-basis-isom-x-i:
assumes *x*: $x \in$ *canonical-basis-K-n n*

shows $\exists!j\in\{..<n\}$. $x = $ *x-i j n*
proof −
obtain *j* :: *nat* **where** *j*: $j \in \{..<n\}$ **and** *x*: $x = $ *x-i j n*
using *canonical-basis-is-x-i* [*OF x*] **by** *blast*
show *?thesis*
proof (*rule ex1I* [*of - j*], *rule conjI*)
show $j \in \{..<n\}$ **by** *fact*
show $x = $ *x-i j n* **by** *fact*
fix *ja*
assume *ja*: $ja \in \{..<n\} \wedge x = $ *x-i ja n*
show $ja = j$
using *x ja* **unfolding** *x-i-def*
by (*metis ja x x-i-ne-x-j*)
qed
qed

The function *preim* maps vectors of the basis *canonical-basis-K-n n* to their index.

definition
preim :: *'a vector* => *nat* => *nat*
where *preim x n* = (*THE j*. *j* ∈ {..<*n*} ∧ *x* = *x-i j n*)

lemma
preim-x-i-x-eq-x:
assumes *x-l-n*: *x* < *n*

shows *preim* (*x-i x n*) *n* = *x*
unfolding *preim-def*
proof
show *x* ∈ {..<*n*} ∧ *x-i x n* = *x-i x n*
using *x-l-n* **by** *fast*
fix *j* :: *nat*
assume *j*: *j* ∈ {..<*n*} ∧ *x-i x n* = *x-i j n*
show *j* = *x*
using *j*
unfolding *x-i-def* **by** (*metis j x-i-ne-x-j*)
qed

lemma
preim-eq-x-i-acc:
assumes *x*: *x* ∈ *canonical-basis-acc k n*
and *k-l-n*: *k* < *n*
shows *x-i* (*preim x n*) *n* = *x*
unfolding *preim-def*
using *theI'* [*OF canonical-basis-acc-isom-x-i2* [*OF x k-l-n*]] **by** *presburger*

lemma
preim-eq-x-i:
assumes *x*: *x* ∈ *canonical-basis-K-n n*

shows *x-i* (*preim x n*) *n* = *x*
unfolding *preim-def*
using *theI'* [*OF canonical-basis-isom-x-i* [*OF x*]] **by** *presburger*

lemma
preim-lessThan:
assumes *x*: *x* ∈ *canonical-basis-K-n n*

shows *preim x n* ∈ {..<*n*}
unfolding *preim-def*
using *theI'* [*OF canonical-basis-isom-x-i* [*OF x*]] **by** *fast*

11.5 Properties of *canonical-basis-K-n n*:

The following lemma proves that two different ways of writing down an element of *K-n n* as a linear combination of the elements of the basis *canonical-basis-K-n n* are equivalent.:

```
lemma
  finsum-canonical-basis-acc-finsum-card:
  assumes k-l-n: k < n
  and f: f ∈ carrier (K-n n) → carrier R
  shows (⨁ K-n n x∈canonical-basis-acc k n. f x ⊙ x)
  = (⨁ K-n n k∈{..<Suc k}. f (x-i k n) ⊙ x-i k n)
proof (rule abelian-monoid.finsum-cong'' [of - -  (λk. x-i k n)])
  show abelian-monoid (K-n n)
    using abelian-monoid-K-n .
  show finite {..<Suc k} using finite-lessThan .
  show bij-betw (λk. x-i k n) {..<Suc k} (canonical-basis-acc k n)
  proof (rule bij-betwI [of - - - (λj. preim j n) ])
    show (λk. x-i k n) ∈ {..<Suc k} → canonical-basis-acc k n
      using canonical-basis-acc-insert2 [OF - k-l-n] by force
    show (λj. preim j n) ∈ canonical-basis-acc k n → {..<Suc k}
    proof
      fix x assume x: x ∈ canonical-basis-acc k n
      obtain j where x-i-x: x-i j n = x and j-lessThan: j < Suc k
        using canonical-basis-acc-eq-x-i [OF x k-l-n] by blast
      show preim x n ∈ {..<Suc k}
        unfolding x-i-x [symmetric]
        using preim-x-i-x-eq-x [of j n] k-l-n j-lessThan by force
    qed
    fix x assume x: x ∈ {..<Suc k}
    show preim (x-i x n) n = x
      using preim-x-i-x-eq-x [of x n] k-l-n x by simp
  next
    fix y assume y: y ∈ canonical-basis-acc k n
    show x-i (preim y n) n = y
      using preim-eq-x-i-acc [OF y k-l-n] .
  qed
  show (λx. f x ⊙ x) ∈ canonical-basis-acc k n → carrier (K-n n)
  proof
    fix x assume x: x ∈ canonical-basis-acc k n
    obtain j where xi: x-i j n = x and j: j ∈ {..<Suc k}
      using canonical-basis-acc-eq-x-i [OF x k-l-n] by fast
    show f x ⊙ x ∈ carrier (K-n n)
      apply (rule K-n-scalar-product-closed)
      unfolding xi [symmetric] using f using x-i-closed j k-l-n by auto
  qed
  show (λk. f (x-i k n) ⊙ x-i k n) ∈ {..<Suc k} → carrier (K-n n)
  proof
    fix x assume x: x ∈ {..<Suc k}
    show f (x-i x n) ⊙ x-i x n ∈ carrier (K-n n)
      apply (rule K-n-scalar-product-closed)
      using f using x-i-closed x k-l-n by auto
  qed
  show ⋀x. x ∈ {..<Suc k} =simp=> f (x-i x n) ⊙ x-i x n = f (x-i x n) ⊙ x-i x
```

n
  by *presburger*
qed

lemma
  finsum-canonical-basis-K-n-finsum-card:
  assumes f: $f \in$ *carrier* (*K-n n*) $\rightarrow$ *carrier R*
  shows $(\bigoplus_{K\text{-}n\ n} x \in (canonical\text{-}basis\text{-}K\text{-}n\ n).\ f\ x \odot x)$
  $= (\bigoplus_{K\text{-}n\ n} k \in \{..<n\}.\ f\ (x\text{-}i\ k\ n) \odot x\text{-}i\ k\ n)$
proof (*cases n*)
  case *0*
  interpret *vector-space R K-n 0 op* $\odot$ **using** *vector-space-K-n* **.**
  show *?thesis*
    unfolding *0*
    unfolding *canonical-basis-K-n-def* **by** *simp*
next
  case (*Suc n*)
  interpret *vector-space R K-n* (*Suc n*) *op* $\odot$ **using** *vector-space-K-n* **.**
  show *?thesis*
    using *f*
    unfolding *Suc canonical-basis-K-n-def*
    using *finsum-canonical-basis-acc-finsum-card* [*of Suc n* $-$ *1 Suc n f*]
    by *simp*
qed

The space generated by the *vector-space.span* of *canonical-basis-K-n n* is equal to the vector space *K-n n*.

lemma
  span-canonical-basis-K-n-carrier-K-n:

  shows *vector-space.span R* (*K-n n*) (*op* $\odot$) (*canonical-basis-K-n n*) = *carrier* (*K-n n*)
proof
  interpret *vector-space R K-n n op* $\odot$ **using** *vector-space-K-n* **.**
  show *span* (*canonical-basis-K-n n*) $\subseteq$ *carrier* (*K-n n*)
  proof
    fix *x*
    assume *x*: $x \in$ *span* (*canonical-basis-K-n n*)
    obtain g :: (*nat* $\Rightarrow$ *'a*) $\times$ *nat* => *'a*
      where *g*: $g \in$ *coefficients-function* (*carrier* (*K-n n*))
      and *gx*: $x =$ *linear-combination g* (*canonical-basis-K-n n*)
      using *x* **unfolding** *span-def* **by** *blast*
    show $x \in$ *carrier* (*K-n n*)
      unfolding *gx*
      by (*rule linear-combination-closed*,
        rule canonical-basis-K-n-good-set,
        rule g)
  qed
  show *carrier* (*K-n n*) $\subseteq$ *span* (*canonical-basis-K-n n*)

```
  proof
    fix x
    assume x: x ∈ carrier (K-n n)
    def lc ≡ finsum (K-n n) (λj. fst x j ⊙ x-i j n) {..<n}
    def reindex ≡ (λt. if t ∈ (canonical-basis-K-n n) then fst x (preim t n) else 0)
    have x = lc
      using K-n-carrier-finsum-x-i [OF x]
      unfolding lc-def .
    also have lc ∈ span (canonical-basis-K-n n)
      unfolding lc-def
      unfolding span-def
      unfolding coefficients-function-def
      unfolding linear-combination-def
      apply auto
      apply (rule exI [of - reindex])
      apply (rule conjI3)
    proof −
      show (⨁K-n n j∈{..<n}. fst x j ⊙ x-i j n)
        = (⨁K-n n y∈canonical-basis-K-n n. reindex y ⊙ y)
      proof (rule abelian-monoid.finsum-cong'' [
          symmetric, OF abelian-monoid-K-n [of n], of - (λj. x-i j n)])
        show finite {..<n} by simp
        show bij-betw (λj. x-i j n) {..<n} (canonical-basis-K-n n)
         proof (rule bij-betwI [of (λj. x-i j n) {..<n} canonical-basis-K-n n (λx.
preim x n)])
          show (λj. x-i j n) ∈ {..<n} → canonical-basis-K-n n
            using canonical-basis-K-n-elements [OF ] by fast
        next
          show (λx. preim x n) ∈ canonical-basis-K-n n → {..<n}
            using preim-lessThan [OF - ] by blast
        next
          fix x assume x: x ∈ {..<n}
          show preim (x-i x n) n = x
            using preim-x-i-x-eq-x [OF - , of x]
            using x by fast
        next
          fix y assume y: y ∈ canonical-basis-K-n n
          show x-i (preim y n) n = y
            by (rule preim-eq-x-i [OF y])
        qed
        show (λy. reindex y ⊙ y) ∈ canonical-basis-K-n n → carrier (K-n n)
        proof
          fix xa
          assume xa: xa ∈ canonical-basis-K-n n
          hence xa2: xa ∈ carrier (K-n n)
            using canonical-basis-K-n-closed [OF ] by fast
          have xa-l-n: preim xa n ∈ {..<n}
            by (rule preim-lessThan [OF xa ])
          hence f: fst x (preim xa n) ∈ carrier R
```

```
          using x unfolding K-n-def K-n-carrier-def ith-def by auto
        show reindex xa ⊙ xa ∈ carrier (K-n n)
          unfolding reindex-def
          using xa
          using K-n-scalar-product-closed [OF f xa2] by presburger
      qed
      show (λj. fst x j ⊙ x-i j n) ∈ {..<n} → carrier (K-n n)
      proof
        fix xa assume xa: xa ∈ {..<n}
        hence f: fst x xa ∈ carrier R using x
          unfolding K-n-def K-n-carrier-def ith-def by auto
        have x-i: x-i xa n ∈ carrier (K-n n)
          using x-i-closed [of xa n] xa by fast
        show fst x xa ⊙ x-i xa n ∈ carrier (K-n n)
          by (rule K-n-scalar-product-closed, rule f, rule x-i)
      qed
      show ⋀xa. xa ∈ {..<n} =simp=>
        fst x xa ⊙ x-i xa n = reindex (x-i xa n) ⊙ x-i xa n
        unfolding reindex-def
        using canonical-basis-K-n-elements [of - n]
        using preim-x-i-x-eq-x [OF -, of -] by force
    qed
    show reindex ∈ carrier (K-n n) → carrier R
    proof
      fix xa
      assume xa: xa ∈ carrier (K-n n)
      show reindex xa ∈ carrier R
        unfolding reindex-def
        using preim-lessThan [of xa n]
        using x unfolding K-n-def K-n-carrier-def ith-def by fastsimp
    qed
    show ∀ a b. (a, b) ∉ carrier (K-n n) ⟶ reindex (a, b) = 0
    proof (rule+)
      fix a b assume notin-carrier: (a,b) ∉ carrier (K-n n)
      have (a,b) ∉ canonical-basis-K-n n
        using canonical-basis-K-n-closed[of n] notin-carrier
        by fast
      thus reindex (a, b) = 0 unfolding reindex-def by presburger
    qed
  qed
  finally show x ∈ span (canonical-basis-K-n n) .
 qed
qed

lemma
 canonical-basis-K-n-spanning-set:

 shows vector-space.spanning-set R (K-n n) (op ⊙) (canonical-basis-K-n n)
 apply (unfold vector-space.spanning-set-def [OF vector-space-K-n], auto)
```

apply (*metis canonical-basis-K-n-good-set*)
using *span-canonical-basis-K-n-carrier-K-n* [*OF*]
using *vector-space.span-def* [*OF vector-space-K-n*] **by** *force*

The elements of *canonical-basis-acc j n* are linearly independent.

lemma
canonical-basis-acc-linear-independent-ext:
assumes *j-l-n*: $j < n$
shows *vector-space.linear-independent-ext R* (*K-n n*) (*op* $\odot$) (*canonical-basis-acc j n*)
proof –
— We first produce the interpretation of the locale *vector-space*
interpret *vector-space R* (*K-n n*) (*op* $\odot$)
using *vector-space-K-n* [*of n*] **.**
have *linear-independent-ext* (*canonical-basis-acc j n*) =
linear-independent (*canonical-basis-acc j n*)
unfolding *linear-independent-ext-def*
using *finite-canonical-basis-acc* [*of j n*]
by (*metis independent-set-implies-independent-subset subset-refl*)
also have *linear-independent* (*canonical-basis-acc j n*)
proof (*rule ccontr*)
assume *n*: ¬ *linear-independent* (*canonical-basis-acc j n*)
have *ld*: *linear-dependent* (*canonical-basis-acc j n*)
proof (*rule not-independent-implies-dependent*)
show ¬ *linear-independent* (*canonical-basis-acc j n*) **by** (*rule n*)
show *good-set* (*canonical-basis-acc j n*)
unfolding *good-set-def*
using *finite-canonical-basis-acc* [*of j n*]
using *canonical-basis-acc-closed* [*OF j-l-n*] **by** *fast*
qed
then obtain *f* **where** *f*: $f \in$ *coefficients-function* (*carrier* (*K-n n*))
and *lc*: *linear-combination f* (*canonical-basis-acc j n*) = $\mathbf{0}_{K\text{-}n\ n}$
and *nzero*: ¬ ($\forall x \in$(*canonical-basis-acc j n*). $f\ x = \mathbf{0}$)
unfolding *linear-dependent-def* **by** *fast*
have $\mathbf{0}_{K\text{-}n\ n}$ = *linear-combination f* (*canonical-basis-acc j n*)
by (*rule lc* [*symmetric*])
also have *linear-combination f* (*canonical-basis-acc j n*) =
finsum (*K-n n*) ($\lambda x.\ f\ x \odot x$) (*canonical-basis-acc j n*)
unfolding *linear-combination-def* **..**
also have ... = *finsum* (*K-n n*) ($\lambda k.\ f$ (*x-i k n*) $\odot$ *x-i k n*) {..<(*Suc j*)}
apply (*rule finsum-canonical-basis-acc-finsum-card*, *rule j-l-n*)
using *f* **unfolding** *coefficients-function-def* **by** *fast*
also have ... = ($\lambda k.$ *if* $k \in$ {..<*Suc j*} *then f* (*x-i k n*) *else* $\mathbf{0}$, $n - 1$)
apply (*rule lambda-finsum* [*symmetric*])
using *f* **unfolding** *coefficients-function-def* **using** *x-i-closed* [*of - n*]
using *j-l-n* **by** *auto*
finally have $\mathbf{0}_{K\text{-}n\ n}$ = ($\lambda k.$ *if* $k \in$ {..<*Suc j*} *then f* (*x-i k n*) *else* $\mathbf{0}$, $n - 1$) **.**
hence *p*: ($\lambda i.\ \mathbf{0}$) = ($\lambda k.$ *if* $k \in$ {..<*Suc j*} *then f* (*x-i k n*) *else* $\mathbf{0}$)
unfolding *K-n-def K-n-zero-def* **by** *auto*

have *j-zero*: $\forall k \in \{..<Suc\ j\}.\ f\ (x\text{-}i\ k\ n) = \mathbf{0}$
using *fun-cong* [*OF p*] **by** *metis*
have $\forall x \in (canonical\text{-}basis\text{-}acc\ (Suc\ j)\ n).\ f\ x = \mathbf{0}$
proof
fix *x* **assume** *x*: $x \in canonical\text{-}basis\text{-}acc\ (Suc\ j)\ n$
obtain *k* **where** *xi*: $x = x\text{-}i\ k\ n$ **and** *k*: $k \in \{..<Suc\ j\}$
by (*metis assms canonical-basis-acc-eq-x-i j-zero nzero*)
show $f\ x = \mathbf{0}$ **unfolding** *xi* **using** *j-zero k* **by** *blast*
qed
hence $\forall x \in (canonical\text{-}basis\text{-}acc\ j\ n).\ f\ x = \mathbf{0}$
by (*metis assms canonical-basis-acc-eq-x-i j-zero*)
thus *False*
using *nzero* **by** *fast*
qed
finally show *?thesis* **.**
qed

end

context *vector-space*
begin

The following lemma should be moved to the place where *linear-independent-ext* has been defined, like a *simp* rule:

lemma *linear-independent-ext-empty* [*simp*]:
shows *linear-independent-ext* {}
unfolding *linear-independent-ext-def*
using *empty-set-is-linearly-independent* **by** *simp*

end

context *field*
begin

lemma
canonical-basis-K-n-linear-independent-ext:

shows *vector-space.linear-independent-ext R* (*K-n n*) (*op* $\odot$) (*canonical-basis-K-n n*)
unfolding *canonical-basis-K-n-def*
using *canonical-basis-acc-linear-independent-ext* [*of* $n - 1$ *n*]
using *vector-space.linear-independent-ext-empty* [*OF vector-space-K-n*]
by (*cases n*, *auto*)

We finally prove that *canonical-basis-K-n n* is a basis for *K-n*.

lemma
canonical-basis-K-n-basis:

shows *vector-space.basis R* (*K-n n*) (*op* $\odot$) (*canonical-basis-K-n n*)

unfolding *vector-space.basis-def* [*OF vector-space-K-n*]
using *canonical-basis-K-n-linear-independent-ext* [*OF*]
using *canonical-basis-K-n-spanning-set* [*OF*]
by (*metis canonical-basis-K-n-closed vector-space.spanning-imp-spanning-ext vector-space-K-n*)

corollary
canonical-basis-K-n-basis-card-n:

shows *vector-space.basis R* (*K-n n*) (*op* ⊙) (*canonical-basis-K-n n*) ∧
card (*canonical-basis-K-n n*) = *n*
using *canonical-basis-K-n-basis* [*OF*]
and *card-canonical-basis-K-n* [*OF*] **by** *fastsimp*

end

context *finite-dimensional-vector-space*
begin

After proving the most relevant properties of *field.K-n K n*, we fix one indexing of the basis elements (of X) that will allow us to define later the function which given any element of the carrier set decomposes it into the coefficients for each term if the indexation.

The theorem *obtain-indexing*: *finite A* ⟹ ∃*f*. *indexing* (*A*, *f*) and the premise that the vector space is finite, and so is it basis X, ensures that the following definition is sound.

definition *indexing-X* :: *nat* => *'c*
where *indexing-X-def*: *indexing-X* = (*SOME f*. *indexing* (*X*, *f*))

Relying in the fact that at least one indexing of the basis X exists, we can prove that *indexing-X* satisfies the properties of every *indexing*.

lemma *indexing-X-is-indexing*:
shows *indexing* (*X*, *indexing-X*)
using *obtain-indexing* [*OF finite-X*]
using *some-eq-ex* [*of* (λ*f*. *indexing* (*X*, *f*))]
unfolding *indexing-X-def* **by** *auto*

The following function is to be used as the inverse function of *field.preim*; this function and *field.preim* will be defined to prove an isomorphism between *field.canonical-basis-K-n K* (*card X*) and {..<*card X*}.

definition *iso-nat-can* :: *nat* => *'a vector*
where *iso-nat-can n* = (*x-i n* (*dimension*))

The composition of the functions *field.preim K* and *iso-nat-can* over the set {..<*dimension*} is equal to the identity.

lemma *preim-iso-nat-can-id*:

```
  assumes x: x ∈ {..<dimension}
  shows preim (iso-nat-can x) (dimension) = x
  unfolding iso-nat-can-def
  using preim-x-i-x-eq-x [of x dimension]
  unfolding x-i-def using x by blast
```

In a very similar way, the composition of *field.preim K* and *iso-nat-can* over the set *field.canonical-basis-K-n K dimension* is equal to the identity:

```
lemma iso-nat-can-preim-id:
  assumes y: y ∈ canonical-basis-K-n (dimension)
  shows iso-nat-can (preim y (dimension)) = y
  using preim-eq-x-i [OF y ]
  unfolding x-i-def iso-nat-can-def .

lemma
  bij-betw-iso-nat-can:
  shows bij-betw iso-nat-can {..<dimension}
  (canonical-basis-K-n (dimension))
proof (intro bij-betwI [of - - - (λi. preim i (dimension))])
  interpret field K by intro-locales
  show iso-nat-can
    ∈ {..<dimension} → field.canonical-basis-K-n K (dimension)
  proof
    fix x
    assume x: x ∈ {..<(dimension)}
    show iso-nat-can x
      ∈ field.canonical-basis-K-n K (dimension)
      unfolding iso-nat-can-def
      using canonical-basis-K-n-elements [OF x]
      unfolding x-i-def .
  qed
  show (λi. preim i (dimension))
    ∈ canonical-basis-K-n (dimension) → {..<dimension}
  proof
    fix x
    assume x: x ∈ canonical-basis-K-n (dimension)
    show preim x (dimension) ∈ {..<dimension}
      by (rule preim-lessThan [OF x])
  qed
  fix x
  assume x: x ∈ {..<dimension}
  show preim (iso-nat-can x) (dimension) = x
    by (rule preim-iso-nat-can-id [OF x])
next
  interpret field K by intro-locales
  fix y
  assume y: y ∈ canonical-basis-K-n (dimension)
  show iso-nat-can (preim y (dimension)) = y
    by (rule iso-nat-can-preim-id [OF y])
```

qed

lemma
  bij-betw-preim:
  shows *bij-betw* ($\lambda i.$ *preim* *i* (*dimension*))
  (*canonical-basis-K-n* (*dimension*)) {..<*dimension*}
proof (*intro bij-betwI* [*of* - - - *iso-nat-can*])
  interpret *field K* **by** *intro-locales*
  show *iso-nat-can*
    ∈ {..<*dimension*} → *canonical-basis-K-n* (*dimension*)
  proof
    fix *x*
    assume *x*: *x* ∈ {..<(*dimension*)}
    show *iso-nat-can x* ∈ *canonical-basis-K-n* (*dimension*)
      unfolding *iso-nat-can-def*
      using *canonical-basis-K-n-elements* [*OF x*]
      unfolding *x-i-def* **.**
  qed
  show ($\lambda i.$ *preim i* (*dimension*))
    ∈ *canonical-basis-K-n* (*dimension*) → {..<*dimension*}
  proof
    fix *x*
    assume *x*: *x* ∈ *canonical-basis-K-n* (*dimension*)
    show *preim x* (*dimension*) ∈ {..<*dimension*}
      by (*rule preim-lessThan* [*OF x*])
  qed
  fix *x*
  assume *x*: *x* ∈ {..<*dimension*}
  show *preim* (*iso-nat-can x*) (*dimension*) = *x*
    by (*rule preim-iso-nat-can-id* [*OF x*])
next
  interpret *field K* **by** *intro-locales*
  fix *y*
  assume *y*: *y* ∈ *canonical-basis-K-n* (*dimension*)
  show *iso-nat-can* (*preim y* (*dimension*)) = *y*
    by (*rule iso-nat-can-preim-id* [*OF y*])
qed

The following function will be used to define an isomorphism between the sets {..<*dimension*} and *X*, which inverse will be the inverse of the indexing function *indexing-X*.

definition
  iso-nat-X :: *nat* => *'c*
  where *iso-nat-X n* = *indexing-X n*

The inverse function of the previous *iso-nat-X* is the following function, which properties we are to prove first:

definition
  preim2 :: *'c* => *nat*

where *preim2 x* = (*THE j*. *j* ∈ {..<*dimension*} ∧ *x* = *indexing-X j*)

The *preim2* function needs to be completed, since otherwise we can not ensure for the elements out of the basis X that their value *preim2 x* is not in the set {..<*dimension*}. If the value *preim2 x* could be in {..<*dimension*} for elements out of X, then the function *fst x* (*preim2 y*), for $y \notin X$ could take values different from **0**.

The way to complete it is a bit artificial, since we can not use *0* to complete it, but some element a with *dimension* ≤ *a*, which are the natural numbers that are mapped to **0** by *coefficients-function*. In particular, we have chosen *a* = *dimension*.

definition
preim2-comp :: *'c* => *nat*
where *preim2-comp x* = (*if x* ∈ *X then* (*THE j*. *j* ∈ {..<*dimension*} ∧ *x* = *indexing-X j*) *else dimension*)

lemma
indexing-X-bij:
shows *bij-betw indexing-X* {..<*dimension*} *X*
proof −
have *f1*: *finite X* **and** *f2*: *finite* {..<*dimension*} **by** (*metis finite-X*, *simp*)
have *ex*: ∃*f*. *bij-betw f* {..<*dimension*} *X*
using *BIJ* [*OF f2 f1*] **unfolding** *dimension-def* **by** *simp*
thus *?thesis*
using *some-eq-ex* [*of* (λ*f*. *bij-betw f* {..<*dimension*} *X*)]
unfolding *indexing-X-def indexing-def dimension-def* **by** *simp*
qed

lemma
indexing-X-preimage:
assumes *x*: *x* ∈ *X*
shows ∃*j*. *j* ∈ {..<*dimension*} ∧ *x* = *indexing-X j*
proof −
obtain *j* **where** *j* ∈ {..<*dimension*} **and** *indexing-X j* = *x*
using *x* **using** *indexing-X-bij*
unfolding *bij-betw-def* **unfolding** *image-def* **by** *force*
thus *?thesis* **by** *fast*
qed

corollary
indexing-X-preimage-unique:
assumes *x*: *x* ∈ *X*
shows ∃!*j*. *j* ∈ {..<*dimension*} ∧ *x* = *indexing-X j*
proof −
obtain *j* :: *nat* **where** *j*: *j* ∈ {..<*dimension*} **and** *x*: *x* = *indexing-X j*
using *indexing-X-preimage* [*OF x*] **by** *fast*
show *?thesis*
proof (*rule ex1I* [*of - j*], *rule conjI*)

show $j \in \{..<dimension\}$ **by** *fact*
show $x = indexing\text{-}X\ j$ **by** (*rule x*)
fix *ja*
assume *ja*: $ja \in \{..<dimension\} \wedge x = indexing\text{-}X\ ja$
show $ja = j$
using *x j ja indexing-X-bij*
unfolding *bij-betw-def*
by (*metis inj-onD*)
qed
qed

lemma
preim2-in-dimension:
assumes *x*: $x \in X$
shows *preim2* $x \in \{..<dimension\}$
unfolding *preim2-def*
using *theI'* [*OF indexing-X-preimage-unique* [*OF x*]] **by** *fast*

lemma
preim2-comp-in-dimension:
assumes *x*: $x \in X$
shows *preim2-comp* $x \in \{..<dimension\}$
using *preim2-in-dimension* [*OF x*] *x*
unfolding *preim2-comp-def preim2-def* **by** *simp*

lemma
preim2-is-indexing-X:
assumes *x*: $x \in X$
shows $x = indexing\text{-}X$ (*preim2 x*)
unfolding *preim2-def*
using *theI'* [*OF indexing-X-preimage-unique* [*OF x*]] **by** *fast*

The functions *preim2-comp* and *iso-nat-X* are inverse of each other, over the sets X and $\{..<dimension\}$

lemma
preim2-comp-is-indexing-X:
assumes *x*: $x \in X$
shows $x = indexing\text{-}X$ (*preim2-comp x*)
using *preim2-is-indexing-X* [*OF x*] *x*
unfolding *preim2-def preim2-comp-def* **by** *presburger*

lemma *iso-nat-X-preim2-id*:
assumes *x*: $x \in X$
shows *iso-nat-X* (*preim2 x*) $= x$
using *theI'* [*OF indexing-X-preimage-unique* [*OF x*]]
unfolding *preim2-def*
unfolding *iso-nat-X-def* **by** *presburger*

lemma *iso-nat-X-preim2-comp-id*:

assumes *x*: $x \in X$
shows *iso-nat-X* (*preim2-comp x*) = *x*
using *iso-nat-X-preim2-id* [*OF x*]
unfolding *preim2-def preim2-comp-def* **using** *x* **by** *presburger*

lemma *preim2-iso-nat-X-id*:
assumes *n*: $n \in \{..<dimension\}$
shows *preim2* (*iso-nat-X n*) = *n*
proof –
have *i*: *iso-nat-X n* $\in X$
unfolding *iso-nat-X-def iso-nat-X-def*
using *indexing-X-is-indexing* **using** *n*
unfolding *indexing-def dimension-def* **unfolding** *bij-betw-def image-def* **by** *auto*
show *?thesis*
unfolding *preim2-def iso-nat-X-def*
apply (*rule the1-equality*)
using *indexing-X-preimage-unique* [*OF i*] *n*
unfolding *iso-nat-X-def* **by** *fast+*
qed

lemma *preim2-comp-iso-nat-X-id*:
assumes *n*: $n \in \{..<dimension\}$
shows *preim2-comp* (*iso-nat-X n*) = *n*
proof –
have *i*: *iso-nat-X n* $\in X$
unfolding *iso-nat-X-def iso-nat-X-def*
using *indexing-X-is-indexing* **using** *n*
unfolding *indexing-def dimension-def* **unfolding** *bij-betw-def image-def* **by** *auto*
show *?thesis*
using *preim2-iso-nat-X-id* [*OF n*] **using** *i*
unfolding *preim2-comp-def preim2-def* **by** *presburger*
qed

Therefore, we can prove that there exists a bijection between them:

lemma
bij-betw-iso-nat-X:
shows *bij-betw iso-nat-X* $\{..<dimension\}$ *X*
proof (*intro bij-betwI* [*of - - - preim2*])
show *iso-nat-X* $\in \{..<dimension\} \to X$
proof
fix *x* **assume** *x*: $x \in \{..<dimension\}$
show *iso-nat-X x* $\in X$
unfolding *iso-nat-X-def*
using *indexing-X-is-indexing* **using** *x*
unfolding *indexing-def bij-betw-def image-def dimension-def* **by** *auto*
qed
show *preim2* $\in X \to \{..<dimension\}$

```
  proof
    fix x assume x: x ∈ X
    show preim2 x ∈ {..<dimension}
      using theI' [OF indexing-X-preimage-unique [OF x]]
      unfolding preim2-def by fast
  qed
  fix x assume x: x ∈ {..<dimension}
  show preim2 (iso-nat-X x) = x
    by (rule preim2-iso-nat-X-id [OF x])
next
  fix y assume y: y ∈ X
  show iso-nat-X (preim2 y) = y
    by (rule iso-nat-X-preim2-id [OF y])
qed

lemma
  bij-betw-preim2:
  shows bij-betw preim2 X {..<dimension}
proof (intro bij-betwI [of - - - iso-nat-X])
  show preim2 ∈ X → {..<dimension}
  proof
    fix x assume x: x ∈ X
    show preim2 x ∈ {..<dimension}
      using theI' [OF indexing-X-preimage-unique [OF x]]
      unfolding preim2-def by fast
  qed
  show iso-nat-X ∈ {..<dimension} → X
  proof
    fix x assume x: x ∈ {..<dimension}
    show iso-nat-X x ∈ X
      unfolding iso-nat-X-def
      using indexing-X-is-indexing using x
      unfolding indexing-def bij-betw-def image-def dimension-def by auto
  qed
  fix y assume y: y ∈ X
  show iso-nat-X (preim2 y) = y
    by (rule iso-nat-X-preim2-id [OF y])
next
  fix x assume x: x ∈ {..<dimension}
  show preim2 (iso-nat-X x) = x
    by (rule preim2-iso-nat-X-id [OF x])
qed

end
```

11.6 Linear maps.

In this section we are going to introduce the notion of linear map between vector spaces. This is a previous step for the definition of an isomorphism between vector spaces. Then, we will have to prove the existence of an isomorphism between the vector spaces *K-n dimension* and *V*.

The definition between comments would be the expected and desired one. Unfortunately, it introduces changes in the namespace that are really inconvenient. The second locale hides the names of constants in vector space, demanding long names for the first locale constanst. We do not know how to control this behaviour: thus, we preferred the long version, in which locale interpretation has to be done later by hand:

locale *linear-map* =
  fixes K :: ($'a$, $'b$) *ring-scheme*
  and V :: ($'c$, $'d$) *ring-scheme*
  and W :: ($'e$, $'f$) *ring-scheme*
  and *scalar-product1* :: $'a$ => $'c$ => $'c$ (**infixr** $\cdot_V$ *70*)
  and *scalar-product2* :: $'a$ => $'e$ => $'e$ (**infixr** $\cdot_W$ *70*)
  assumes V: *vector-space* K V (*op* $\cdot_V$)
  and W: *vector-space* K W (*op* $\cdot_W$)

context *linear-map*
begin

Linear maps, as characterised in "Linear Algebra Done Right", have to satisfy the additivity and homogeneity properties:

definition *additivity* :: ($'c$ => $'e$) => *bool*
  where *additivity* T = ($\forall\, x \in$*carrier* V. $\forall\, y \in$ *carrier* V. T ($x \oplus_V y$) = $T\, x \oplus_W$ $T\, y$)

definition *homogeneity* :: ($'c$ => $'e$) => *bool*
  where *homogeneity* T = ($\forall\, k \in$*carrier* K. $\forall\, x \in$*carrier* V. T ($k \cdot_V x$) = $k \cdot_W T$ x)

definition *linear-map* :: ($'c$ => $'e$) => *bool*
  where *linear-map* T = (*additivity* T $\wedge$ *homogeneity* T)

end

We introduce a new locale for finite dimensional vector spaces, just imposing that there is a finite basis for one ot the vector spaces.

locale *linear-map-fin-dim* = *linear-map* +
  fixes X
  assumes *fin-dim*: *finite-dimensional-vector-space* K V (*op* $\cdot_V$) X

We produce two different sublocales, or interpretations, of the locale *linear-map-fin-dim* by means of the locale *finite-dimensional-vector-space*. They allow us to later define linear maps from V to *K-n* and also the opposite way, from *K-n* to V. The system forces us to make them *named* interpretations, just to avoid colliding names.

sublocale *finite-dimensional-vector-space* <
V-K-n: *linear-map-fin-dim K V K-n dimension op* · *K-n-scalar-product X*
proof (*unfold linear-map-fin-dim-def*, *intro conjI*)
show *linear-map K V* (*field.K-n K dimension*) *op* · (*field.K-n-scalar-product K*)
proof (*unfold linear-map-def*, *intro conjI*)
show *vector-space K* (*K-n dimension*) *K-n-scalar-product*
using *vector-space-K-n* **.**
show *vector-space K V op* · **by** (*intro-locales*)
qed
next
show *linear-map-fin-dim-axioms K V op* · *X*
proof (*unfold linear-map-fin-dim-axioms-def finite-dimensional-vector-space-def*,
intro conjI)
show *vector-space K V op* ·**by** *intro-locales*
show *finite-dimensional-vector-space-axioms K V op* · *X*
proof
show *finite X* **by** (*rule finite-X*)
show *basis X* **by** (*rule basis-X*)
qed
qed
qed

sublocale *finite-dimensional-vector-space* < *K-n-V*: *linear-map-fin-dim K K-n dimension V*
K-n-scalar-product op · *canonical-basis-K-n dimension*
proof (*intro-locales*)
interpret *K*: *field K* **by** *intro-locales*
interpret *V*: *vector-space K V op* · **by** *intro-locales*
interpret *K-n*: *vector-space K K-n dimension K-n-scalar-product* **using** *vector-space-K-n*
.
show *Isomorphism.linear-map K* (*K-n dimension*) *V* (*K-n-scalar-product*) *op* ·
by *unfold-locales*
show *linear-map-fin-dim-axioms K* (*K-n dimension*)
(*K-n-scalar-product*) (*canonical-basis-K-n dimension*)
proof *unfold-locales*
show *finite* (*canonical-basis-K-n dimension*)
by (*rule finite-canonical-basis-K-n*)
show *K-n.basis* (*canonical-basis-K-n dimension*)
using *canonical-basis-K-n-basis* [*of dimension*] **by** *fast*
qed
qed

11.7 Defining the isomorphism between $\mathbb{K}^n$ and V.

context *finite-dimensional-vector-space*
begin

Some properties proving that there exists a unique function of coefficients for each element in the carrier set of V; this unique function is the one that decomposes any element into its linear combination over the elements of the basis:

lemma
basis-implies-linear-combination:
assumes x: $(x::'c) \in$ *carrier* V
shows $\exists f.\ f \in$ *coefficients-function* $($*carrier* $V) \wedge x =$ *linear-combination* $f\ X$
using *spanning-set-X*
unfolding *spanning-set-def*
using x **by** *blast*

In order to ensure the uniqueness of the coefficients function we have to use *coefficients-function*, which is mapped to **0** out of its domain.

lemma
basis-implies-coeff-function-comp-linear-combination:
assumes x: $(x::'c) \in$ *carrier* V
shows $\exists f.\ f \in$ *coefficients-function* $X \wedge x =$ *linear-combination* $f\ X$
proof $-$
obtain f **where** f: $f \in$ *coefficients-function* $($*carrier* $V)$
and x: $x =$ *linear-combination* $f\ X$
using *basis-implies-linear-combination* [*OF* x] **by** *force*
let *?g* $= (\lambda x.$ *if* $x \in X$ *then* $f\ x$ *else* $\mathbf{0})$
show *?thesis*
proof (*rule exI* [*of - ?g*], *intro conjI*)
show $(\lambda y.$ *if* $y \in X$ *then* $f\ y$ *else* $\mathbf{0}) \in$ *coefficients-function* X
using f
unfolding *coefficients-function-def*
using *good-set-X* **unfolding** *good-set-def* **by** *fastsimp*
show $x =$ *linear-combination* $(\lambda y.$ *if* $y \in X$ *then* $f\ y$ *else* $\mathbf{0})\ X$
unfolding x
unfolding *linear-combination-def*
proof (*rule finsum-cong'*)
show $X = X$ **..**
show $(\lambda y.\ ($*if* $y \in X$ *then* $f\ y$ *else* $\mathbf{0}) \cdot y) \in X \rightarrow$ *carrier* V
proof
fix x **assume** x: $x \in X$
show $($*if* $x \in X$ *then* $f\ x$ *else* $\mathbf{0}) \cdot x \in$ *carrier* V
apply (*cases* $x \in X$)
using *fx-x-in-V* [*of* $x\ f$]
using $f\ x$ *good-set-X*
unfolding *good-set-def* **by** *auto*
qed
fix i **assume** i: $i \in X$
thus $f\ i \cdot i = ($*if* $i \in X$ *then* $f\ i$ *else* $\mathbf{0}) \cdot i$ **by** *fastsimp*
qed
qed
qed

Firstly we prove a theorem similar to *unique-coordenates*: $[\![x \in$ *carrier* V; $f \in$ *coefficients-function* $($*carrier* $V)$; $x =$ *linear-combination* $f\ X$; $g \in$ *coefficients-function* $($*carrier* $V)$; $x =$ *linear-combination* $g\ X]\!] \Longrightarrow \forall x{\in}X.$

$g\ x = f\ x$. It claims that the coordinates are unique in a basis.

lemma
  linear-combination-unique:
  assumes x: $x \in$ *carrier V*
  shows $\exists!f.\ f \in$ *coefficients-function X* & *linear-combination f X* $= x$
proof −
  obtain *f-cf* **where** *cf-fc*: *f-cf* $\in$ *coefficients-function* (*carrier V*)
    and *lc-cf*: *linear-combination f-cf X* $= x$
    using x **using** *spanning-set-X*
    unfolding *spanning-set-def* **by** (*metis mem-def*)
  def $f == (\lambda x.\ if\ x \in X\ then\ f\text{-}cf\ x\ else\ \mathbf{0})$
  have *cf*: $f \in$ *coefficients-function X*
    and *lc*: *linear-combination f X* $= x$
    using *cf-fc lc-cf*
    unfolding *coefficients-function-def*
    unfolding *linear-combination-def*
    unfolding *f-def* **using** *good-set-X* **unfolding** *good-set-def* **apply** *auto*
    apply (*rule finsum-cong′*)
    apply *auto* **by** (*rule mult-closed*) *auto*
  show *?thesis*
  proof (*rule ex1I* [*of - f*])
    show $f \in$ *coefficients-function X* & *linear-combination f X* $= x$ **using** *cf lc* **..**
    fix g
    assume $g \in$ *coefficients-function X* & *linear-combination g X* $= x$
   hence *cfg*: $g \in$ *coefficients-function X* **and** *lcg*: *linear-combination g X* $= x$ **by**
fast+
    have *f-y-y-Pi*: $(\lambda y.\ f\ y \cdot y) \in X \to$ *carrier V*
      and *f-y-Pi*: $(\lambda y.\ f\ y) \in X \to$ *carrier K*
      and *g-y-Pi*: $(\lambda y.\ g\ y) \in X \to$ *carrier K*
      and *g-y-y-Pi*: $(\lambda y.\ g\ y \cdot y) \in X \to$ *carrier V*
    and *f-minus-g-Pi*: $(\lambda y.\ f\ y \cdot y \ominus_V g\ y \cdot y) \in X \to$ *carrier V*

    unfolding *coefficients-function-def*
    unfolding *Pi-def*
    using *coefficients-function-Pi*[*OF - cfg*]
    using *coefficients-function-Pi*[*OF - cf*]
    using *good-set-X* **unfolding** *good-set-def* **by** (*auto simp add*: *mult-closed*)
    show $g = f$
    proof −
      have $\mathbf{0}_V =$ *linear-combination f X* $\ominus_V$ *linear-combination g X*
        unfolding *lc lcg* **using** x **by** (*metis local.r-neg′*)
      also have *linear-combination f X* $\ominus_V$ *linear-combination g X* $=$
        $(\bigoplus_V y{\in}X.\ f\ y \cdot y) \ominus_V (\bigoplus_V y{\in}X.\ g\ y \cdot y)$
        unfolding *linear-combination-def* **..**
      also have $(\bigoplus_V y{\in}X.\ f\ y \cdot y) \ominus_V (\bigoplus_V y{\in}X.\ g\ y \cdot y) = (\bigoplus_V y{\in}X.\ f\ y \cdot y)$
$\oplus_V \ominus_V (\bigoplus_V y{\in}X.\ g\ y \cdot y)$
        unfolding *minus-eq* [*OF finsum-closed* [*OF finite-X f-y-y-Pi*]
          finsum-closed [*OF finite-X g-y-y-Pi*]] **..**
      also have $(\bigoplus_V y{\in}X.\ f\ y \cdot y) \oplus_V \ominus_V (\bigoplus_V y{\in}X.\ g\ y \cdot y) = (\bigoplus_V y{\in}X.\ f\ y$

$\cdot\ y \oplus_V \ominus_V (g\ y \cdot y))$
unfolding *finsum-minus-eq* [*OF finite-X g-y-y-Pi*]
apply (*rule finsum-addf* [*symmetric*, *OF finite-X f-y-y-Pi*])
using *a-inv-closed g-y-y-Pi* **by** *auto*
also have $(\bigoplus_V y{\in}X.\ f\ y \cdot y \oplus_V \ominus_V (g\ y \cdot y)) = (\bigoplus_V y{\in}X.\ f\ y \cdot y \ominus_V (g\ y \cdot y))$
proof (*rule finsum-cong′*, *rule*, *rule f-minus-g-Pi*)
fix i **assume** x: $i \in X$
show $f\ i \cdot i \oplus_V \ominus_V (g\ i \cdot i) = f\ i \cdot i \ominus_V g\ i \cdot i$
by (*rule minus-eq* [*symmetric*], *rule funcset-mem* [*OF f-y-y-Pi x*],
rule funcset-mem [*OF g-y-y-Pi x*])
qed
also have $(\bigoplus_V y{\in}X.\ f\ y \cdot y \ominus_V g\ y \cdot y) = (\bigoplus_V y{\in}X.\ (f\ y \ominus g\ y) \cdot y)$
proof (*rule finsum-cong′* [*symmetric*], *rule*, *rule f-minus-g-Pi*)
show $\bigwedge i.\ i \in X \Longrightarrow (f\ i \ominus g\ i) \cdot i = f\ i \cdot i \ominus_V g\ i \cdot i$
proof −
fix i **assume** i: $i \in X$
hence iV: $i \in$ *carrier* V **using** *good-set-X* **unfolding** *good-set-def*
by *auto*
show $(f\ i \ominus g\ i) \cdot i = f\ i \cdot i \ominus_V g\ i \cdot i$
by (*rule diff-mult-distrib2*, *fact*)
(*rule funcset-mem* [*OF f-y-Pi i*], *rule funcset-mem* [*OF g-y-Pi i*])
qed
qed
also have ... = *linear-combination* $(\lambda x.\ f\ x \ominus g\ x)\ X$
unfolding *linear-combination-def* **..**
finally have *linear-combination* $(\lambda x.\ f\ x \ominus g\ x)\ X = \mathbf{0}_V$ **..**
— A linear combination of elements of the basis X equal to zero means that every coefficient must be zero:
moreover have $(\lambda x.\ f\ x \ominus g\ x) \in$ *coefficients-function* (*carrier* V)

using *coefficients-function-Pi*[*OF - cf*]
using *coefficients-function-Pi*[*OF - cfg*]
unfolding *coefficients-function-def*
apply (*auto simp add*: *minus-closed*)
proof −
fix x
assume *x-notin-V*: $x \notin$ *carrier* V
hence $f\ x \ominus g\ x = \mathbf{0} \ominus \mathbf{0}$
using *cfg cf good-set-X* **unfolding** *coefficients-function-def good-set-def* **by** *fastsimp*
also have ...$=\mathbf{0}$
by (*metis K.add.inv-one K.add.one-closed a-minus-def*
abelian-monoid.r-zero abelian-monoid-R insertI1 insert-absorb mem-def)
finally show $f\ x \ominus g\ x = \mathbf{0}$ **.**
qed
ultimately have *lin-comb-X-eq-0*: $\forall x{\in}X.\ (\lambda x.\ f\ x \ominus g\ x)\ x = \mathbf{0}$
using *linear-independent-X*
unfolding *linear-independent-def* **by** *auto*

have *f-eq-g-X*: $\forall x\in X.\ f\ x = g\ x$
proof (*rule ballI*)
fix x **assume** x: $x \in X$
have *fx*: $f\ x \in$ *carrier K* **and** *gx*: $g\ x \in$ *carrier K*
using x **using** *good-set-X* **unfolding** *good-set-def*
using *cf*
using *cfg*
unfolding *coefficients-function-def* **by** *auto*
have $f\ x \ominus g\ x \oplus g\ x = g\ x$
using *lin-comb-X-eq-0 fx gx x* **by** *simp*
hence $f\ x = g\ x$ **using** *fx gx*
by (*metis plus-minus-cancel cring.cring-simprules*(*16*) *is-cring lin-comb-X-eq-0 x*)
thus $f\ x = g\ x$ **.**
qed
show $g = f$
proof (*rule ext*, *case-tac* $x \in X$)
fix x **assume** x: $x \in X$ **show** $g\ x = f\ x$
using *f-eq-g-X x* **by** *simp*
next
fix x **assume** x: $x \notin X$ **show** $g\ x = f\ x$
using *cf cfg* **unfolding** *coefficients-function-def*
using x **by** *simp*
qed
qed
qed
qed

The previous lemma ensures the existence of only one function f satisfying to be a linear combination and a coefficients function which generates any x belonging to *carrier V*

definition *lin-comb* :: $'c \Rightarrow ('c \Rightarrow 'a)$
where *lin-comb* x = (*THE* f. $f \in$ *coefficients-function X*
$\wedge$ *linear-combination f X* $= x$)

lemma
lin-comb-is-coefficients-function:
assumes x: $x \in$ *carrier V*
shows *lin-comb* $x \in$ *coefficients-function X*
using *theI′* [*OF linear-combination-unique* [*OF x*]]
unfolding *lin-comb-def* **by** *fast*

lemma
lin-comb-is-the-linear-combination:
assumes x: $x \in$ *carrier V*
shows x = *linear-combination* (*lin-comb x*) *X*
unfolding *lin-comb-def*

using *theI'* [*OF linear-combination-unique* [*OF x*]] **by** *simp*

lemma
indexing-X-n-in-X:
assumes *n-dimension*: $n < dimension$
shows *indexing-X* $n \in X$
using *indexing-X-is-indexing*
unfolding *indexing-def*
using *bij-betw-imp-funcset*
using *n-dimension* **unfolding** *dimension-def* **by** *auto*

corollary
indexing-X-n-in-carrier-V:
assumes *n-dimension*: $n < dimension$
shows *indexing-X* $n \in$ *carrier V*
using *indexing-X-n-in-X* [*OF n-dimension*]
using *good-set-X* **unfolding** *good-set-def* **by** *auto*

A lemma stating that every element of the carrier set can be expressed as a finite sum over the elements of the set {..<*dimension*} thanks to the function *lin-comb*.

lemma
lin-comb-is-the-linear-combination-indexing:
assumes *x*: $x \in$ *carrier V*
shows $x =$ *finsum V* (λi. *lin-comb x* (*indexing-X i*) $\cdot$ *indexing-X i*) {..<*dimension*}
proof −
have $x =$ *linear-combination* (*lin-comb x*) *X*
by (*rule lin-comb-is-the-linear-combination* [*OF x*])
also have ... = *finsum V* (λy. *lin-comb x y* $\cdot$ *y*) *X*
unfolding *linear-combination-def* **..**
also have ... = *finsum V* (λi. *lin-comb x* (*indexing-X i*) $\cdot$ *indexing-X i*) {..<*dimension*}
proof (*rule finsum-cong''* [*of - indexing-X*])
show *finite* {..<*dimension*} **by** *fast*
show *bij-betw indexing-X* {..<*dimension*} *X* **by** (*rule indexing-X-bij*)
show (λy. *lin-comb x y* $\cdot$ *y*) $\in X \rightarrow$ *carrier V*
proof
fix *xa* **assume** *xa*: $xa \in X$
show *lin-comb x xa* $\cdot$ *xa* $\in$ *carrier V*
apply (*rule mult-closed*)
using *xa* **using** *good-set-X*
using *lin-comb-is-coefficients-function* [*OF x*]
unfolding *good-set-def coefficients-function-def* **by** *fast+*
qed
show (λi. *lin-comb x* (*indexing-X i*) $\cdot$ *indexing-X i*) $\in$ {..<*dimension*} $\rightarrow$ *carrier V*
proof
fix *xa* **assume** *xa*: *xa* $\in$ {..<*dimension*}
show *lin-comb x* (*indexing-X xa*) $\cdot$ *indexing-X xa* $\in$ *carrier V*
apply (*rule mult-closed*)

```
      using indexing-X-n-in-carrier-V [of xa] xa
      using lin-comb-is-coefficients-function [OF x]
      using coefficients-function-Pi[of indexing-X xa lin-comb x]
      unfolding coefficients-function-def by auto
    qed
    show ⋀xa. xa ∈ {..<dimension} =simp=>
     lin-comb x (indexing-X xa) · indexing-X xa =
     lin-comb x (indexing-X xa) · indexing-X xa by simp
  qed
  finally show ?thesis .
qed
```

A lemma on how the elements of the basis are mapped by *lin-comb*:

```
lemma
  lin-comb-basis:
  assumes x: x ∈ X
  shows lin-comb x = (λi. if i = x then 1 else 0)
  unfolding lin-comb-def
proof (rule the1-equality)
  have x1: x ∈ carrier V
    using good-set-X x
    unfolding good-set-def by fast
  show ∃!f. f ∈ coefficients-function X ∧ linear-combination f X = x
    using linear-combination-unique [OF x1] .
  show (λi. if i = x then 1 else 0) ∈ coefficients-function X ∧
    linear-combination (λi. if i = x then 1 else 0) X = x
  proof (rule conjI)
    show (λi. if i = x then 1 else 0) ∈ coefficients-function X
      unfolding coefficients-function-def using x by fastsimp
    show linear-combination (λi. if i = x then 1 else 0) X = x
    proof −
      thm linear-combination-def
      have linear-combination (λi. if i = x then 1 else 0) X =
        (⨁ Vy∈X. (if y = x then 1 else 0) · y) unfolding linear-combination-def
..
      also have ... = (⨁ Vy∈X. (if x = y then 1 · y else 0V))
        apply (rule finsum-cong', auto)
        using good-set-X
        unfolding good-set-def
        apply (metis mult-1 x1)
        by (metis good-set-X good-set-in-carrier subsetD zeroK-mult-V-is-zeroV)
      also have ... = 1 · x
      proof (rule finsum-singleton [OF x finite-X, of (λx. 1 · x)], rule)
        fix x assume x: x ∈ X hence xx: x ∈ carrier V
          using good-set-X
          unfolding good-set-def by fast
        show 1 · x ∈ carrier V
          by (rule mult-closed [OF xx one-closed])
      qed
```

```
      also have ... = x
        by (rule mult-1 [OF x1])
      finally show ?thesis .
    qed
  qed
qed

end

context vector-space
begin
```

The following lemma is a minor modification of ⟦*finite* $?X$; $?X \subseteq$ *carrier* V; $?a \in$ *carrier* K; $?f \in ?X \to$ *carrier* K⟧ $\Longrightarrow ?a \cdot (\bigoplus_V y \in ?X.\ ?f\ y \cdot y) = (\bigoplus_V y \in ?X.\ ?a \cdot ?f\ y \cdot y)$, but with a bit more general statement. In particular, it removes a premise stating that $X \subseteq$ *carrier* V, which is never used in the proof of ⟦*finite* $?X$; $?X \subseteq$ *carrier* V; $?a \in$ *carrier* K; $?f \in ?X \to$ *carrier* K⟧ $\Longrightarrow ?a \cdot (\bigoplus_V y \in ?X.\ ?f\ y \cdot y) = (\bigoplus_V y \in ?X.\ ?a \cdot ?f\ y \cdot y)$ and also generalizes the inner expression of the finite sum. It may either replace ⟦*finite* $?X$; $?X \subseteq$ *carrier* V; $?a \in$ *carrier* K; $?f \in ?X \to$ *carrier* K⟧ $\Longrightarrow ?a \cdot (\bigoplus_V y \in ?X.\ ?f\ y \cdot y) = (\bigoplus_V y \in ?X.\ ?a \cdot ?f\ y \cdot y)$ in the file *Vector-Space* or added besides it in the same file.

```
lemma finsum-aux2:
  ⟦finite X; a ∈ carrier K; f ∈ X → carrier K; g ∈ X → carrier V⟧
  ⟹ a · (⨁Vy∈X. f y · g y)=(⨁Vy∈X. a · (f y · g y))
proof (induct set: finite)
  case empty thus ?case
    using scalar-mult-zeroV-is-zeroV by auto
next
  case (insert x X)
  show ?case
  proof −
    have sum-closed: (⨁Vy∈X. f y · g y) ∈ carrier V
    proof (rule finsum-closed)
      show finite X using insert.hyps (1) .
      show (λy. f y · g y) ∈ X → carrier V
        using insert.prems (1,2,3) and mult-closed by auto
    qed
    have fx-gx-in-V: f x · g x ∈ carrier V
      using insert.prems (1,2,3) and mult-closed by auto
    have (⨁Vy∈insert x X. f y · g y)= f x · g x ⊕V(⨁Vy∈X. f y · g y)
    proof (rule finsum-insert)
      show finite X using insert.hyps (1) .
      show x∉ X using insert.hyps (2) .
      show f x · g x ∈ carrier V using fx-gx-in-V .
      show (λy. f y · g y) ∈ X → carrier V
        using insert.prems (1,2,3) and mult-closed by auto
    qed
```

```
    hence a · (⨁Vy∈insert x X. f y · g y) = a · f x · g x ⊕V a · (⨁Vy∈X. f y
· g y)
      using add-mult-distrib1 [
        OF fx-gx-in-V sum-closed insert.prems (1)] by auto
    also have . . . = a · f x · g x ⊕V (⨁Vy∈X. a· f y · g y)
    proof −
      have f1: f ∈ X → carrier K using insert.prems(2) by auto
      have g1: g ∈ X → carrier V using insert.prems(3) by auto
      show ?thesis
        unfolding insert.hyps (3) [OF insert.prems (1) f1 g1] ..
    qed
    also have . . . = (⨁Vy∈insert x X. a · f y · g y)
    proof (rule finsum-insert[symmetric])
      show finite X using insert.hyps(1) .
      show x ∉ X using insert.hyps(2) .
      show (λy. a · f y · g y) ∈ X → carrier V
      proof (unfold Pi-def, auto)
        fix y
        assume y-in-X: y∈ X
        show a · f y · g y ∈ carrier V
        proof (rule mult-closed)
          show f y · g y ∈ carrier V
            using y-in-X and insert.prems(1, 2, 3) and mult-closed
            by auto
          show a ∈ carrier K by (rule insert.prems(1))
        qed
      qed
      show a · f x · g x ∈ carrier V
      proof (rule mult-closed)
        show  f x · g x ∈ carrier V
          using insert.prems (1, 2, 3) and mult-closed by auto
        show a ∈ carrier K by (rule insert.prems(1))
      qed
    qed
    finally show ?thesis .
  qed
qed

end

context finite-dimensional-vector-space
begin
```

The following functions are the candidates to be proved to define the isomorphism between the vector spaces *V* and *field.K-n K dimension*. They have to be proved to be linear maps between the vector spaces, and inverse one of each other.

```
definition iso-K-n-V :: 'a vector => 'c
  where iso-K-n-V x = finsum V (λi. fst x i · indexing-X i) {..<dimension}
```

definition *iso-V-K-n* :: *'c => 'a vector*
  where *iso-V-K-n x* =
  finsum (*K-n dimension*) (λ*i*. (*K-n-scalar-product* (*lin-comb* (*x*) (*indexing-X i*)) (*x-i i dimension*))) {..<*dimension*}

We prove that *iso-K-n-V* is a linear map, this means both additive and homogeneous:

lemma *linear-map-iso-K-n-V*: *K-n-V.linear-map iso-K-n-V*
proof (*unfold K-n-V.linear-map-def*, *intro conjI*)
  show *additivity iso-K-n-V*
  proof (*unfold additivity-def*, *rule ballI*, *rule ballI*)
    fix *x y*
    assume *x*: $x \in$ *carrier* (*K-n dimension*)
      and *y*: $y \in$ *carrier* (*K-n dimension*)
    show *iso-K-n-V* ($x \oplus_{K\text{-}n\ dimension}\ y$) =
          iso-K-n-V x $\oplus_V$ *iso-K-n-V y*
    proof −
      have *iso-K-n-V* ($x \oplus_{field.K\text{-}n\ K\ dimension}\ y$) =
        ($\bigoplus_V i \in$ {..<*dimension*}. *fst* ($x \oplus_{K\text{-}n\ dimension}\ y$) *i* $\cdot$ *indexing-X i*)
        unfolding *iso-K-n-V-def* **..**
      also have ... = ($\bigoplus_V i \in$ {..<*dimension*}. (*ith x i* $\oplus$ *ith y i*) $\cdot$ *indexing-X i*)
        unfolding *K-n-def K-n-add-def* **by** *force*
      also have ... = ($\bigoplus_V i \in$ {..<*dimension*}. (*ith x i*) $\cdot$ *indexing-X i* $\oplus_V$
        (*ith y i*) $\cdot$ *indexing-X i*)
      proof (*rule finsum-cong'*)
        show {..<*dimension*} = {..<*dimension*} **by** *fastsimp*
        show (λ*i*. *ith x i* $\cdot$ *indexing-X i* $\oplus_V$ *ith y i* $\cdot$ *indexing-X i*)
          $\in$ {..<*dimension*} $\rightarrow$ *carrier V*
        proof
          fix *xa* **assume** *xa*: *xa* $\in$ {..<*dimension*}
          find-theorems *ith ?x ?i* $\in$ -
          show *ith x xa* $\cdot$ *indexing-X xa* $\oplus_V$ *ith y xa* $\cdot$ *indexing-X xa* $\in$ *carrier V*
          proof (*rule V.a-closed*)
            show *ith x xa* $\cdot$ *indexing-X xa* $\in$ *carrier V*
            proof (*rule mult-closed*)
              show *indexing-X xa* $\in$ *carrier V*
                using *indexing-X-n-in-carrier-V* [*of xa*] *xa* **by** *fastsimp*
              show *ith x xa* $\in$ *carrier K*
                apply (*rule ith-closed* [*of - - dimension*])
                using *x xa* **unfolding** *K-n-def* **by** *simp-all*
            qed
            show *ith y xa* $\cdot$ *indexing-X xa* $\in$ *carrier V*
            proof (*rule mult-closed*)
              show *indexing-X xa* $\in$ *carrier V*
                using *indexing-X-n-in-carrier-V* [*of xa*] *xa* **by** *fastsimp*
              show *ith y xa* $\in$ *carrier K*
                apply (*rule ith-closed* [*of - - dimension*])
                using *y xa* **unfolding** *K-n-def* **by** *simp-all*

qed
qed
qed
fix *xa* **assume** *xa*: $xa \in \{..<dimension\}$
show $(ith\ x\ xa \oplus ith\ y\ xa) \cdot indexing\text{-}X\ xa =$
$ith\ x\ xa \cdot indexing\text{-}X\ xa \oplus_V ith\ y\ xa \cdot indexing\text{-}X\ xa$
proof (*rule add-mult-distrib2*)
show $indexing\text{-}X\ xa \in carrier\ V$
using *indexing-X-n-in-carrier-V* [*of xa*] *xa* **by** *fastsimp*
show $ith\ x\ xa \in carrier\ K$
apply (*rule ith-closed* [*of - - dimension*])
using *x xa* **unfolding** *K-n-def* **by** *simp-all*
show $ith\ y\ xa \in carrier\ K$
apply (*rule ith-closed* [*of - - dimension*])
using *y xa* **unfolding** *K-n-def* **by** *simp-all*
qed
qed
also have ... = $(\bigoplus_V i\in\{..<dimension\}.\ fst\ x\ i \cdot indexing\text{-}X\ i \oplus_V fst\ y\ i \cdot indexing\text{-}X\ i)$
unfolding *ith-def* **..**
also have ... = $(\bigoplus_V i\in\{..<dimension\}.\ fst\ x\ i \cdot indexing\text{-}X\ i) \oplus_V$
$(\bigoplus_V i\in\{..<dimension\}.\ fst\ y\ i \cdot indexing\text{-}X\ i)$
proof (*cases dimension*)
case *0* **show** *?thesis* **unfolding** *0* **by** *simp*
next
case (*Suc n*)
show *?thesis*
unfolding *Suc*
unfolding *lessThan-Suc-atMost*
proof (*rule V.finsum-add* [*of* $(\lambda i.\ fst\ x\ i \cdot indexing\text{-}X\ i)$ *n*
$(\lambda i.\ fst\ y\ i \cdot indexing\text{-}X\ i)$])
show $(\lambda i.\ fst\ x\ i \cdot indexing\text{-}X\ i) \in \{..n\} \to carrier\ V$
proof
fix *xa* **assume** *xa*: $xa \in \{..n\}$
show $fst\ x\ xa \cdot indexing\text{-}X\ xa \in carrier\ V$
proof (*rule mult-closed*)
show $indexing\text{-}X\ xa \in carrier\ V$
using *indexing-X-n-in-carrier-V* [*of xa*] *xa* **using** *Suc* **by** *fastsimp*
show $fst\ x\ xa \in carrier\ K$
apply (*unfold ith-def* [*symmetric*])
apply (*rule ith-closed* [*of - - dimension*])
using *x xa* **unfolding** *K-n-def* **using** *Suc* **by** *simp-all*
qed
qed
show $(\lambda i.\ fst\ y\ i \cdot indexing\text{-}X\ i) \in \{..n\} \to carrier\ V$
proof
fix *xa* **assume** *xa*: $xa \in \{..n\}$
show $fst\ y\ xa \cdot indexing\text{-}X\ xa \in carrier\ V$
proof (*rule mult-closed*)

```
          show indexing-X xa ∈ carrier V
            using indexing-X-n-in-carrier-V [of xa] xa using Suc by fastsimp
          show fst y xa ∈ carrier K
            apply (unfold ith-def [symmetric])
            apply (rule ith-closed [of - - dimension])
            using y xa unfolding K-n-def using Suc by simp-all
        qed
      qed
    qed
  qed
  also have ... = iso-K-n-V x ⊕V iso-K-n-V y
    unfolding iso-K-n-V-def ..
  finally show ?thesis .
 qed
qed
show homogeneity iso-K-n-V
proof (unfold homogeneity-def, rule ballI, rule ballI)
 fix k x
 assume k: k ∈ carrier K and x: x ∈ carrier (K-n dimension)
 show iso-K-n-V (K-n-scalar-product k x) = k · iso-K-n-V x
 proof −
  have iso-K-n-V (K-n-scalar-product k x) =
    (⨁Vi∈{..<dimension}. (k ⊗ fst x i) · indexing-X i)
    unfolding iso-K-n-V-def K-n-scalar-product-def fst-conv ith-def ..
  also have ... = (⨁Vi∈{..<dimension}. k · (fst x i) · indexing-X i)
  proof (rule finsum-cong')
    show {..<dimension} = {..<dimension} ..
    show (λi. k · fst x i · indexing-X i) ∈ {..<dimension} → carrier V
    proof
      fix xa assume xa: xa ∈ {..<dimension}
      hence fst: fst x xa ∈ carrier K
        and i: indexing-X xa ∈ carrier V using x xa
        using indexing-X-n-in-carrier-V [of xa] xa
        unfolding K-n-def K-n-carrier-def ith-def by auto
      show k · fst x xa · indexing-X xa ∈ carrier V
        unfolding mult-assoc [symmetric, OF i k fst]
        by (rule mult-closed [OF i m-closed [OF k fst]])
    qed
    fix xa assume xa: xa ∈ {..<dimension}
    hence fst: fst x xa ∈ carrier K
      and i: indexing-X xa ∈ carrier V using x xa
      using indexing-X-n-in-carrier-V [of xa] xa
      unfolding K-n-def K-n-carrier-def ith-def by auto
    show (k ⊗ fst x xa) · indexing-X xa = k · fst x xa · indexing-X xa
      by (rule mult-assoc [OF i k fst])
  qed
  also have ... = k · (⨁Vi∈{..<dimension}. (fst x i) · indexing-X i)
  proof (rule finsum-aux2 [symmetric])
    show finite {..<dimension} by simp
```

show $k \in$ *carrier K* **by** (*rule k*)
show *fst x* $\in \{..<$*dimension*$\} \rightarrow$ *carrier K*
using *x*
unfolding *K-n-def K-n-carrier-def ith-def* **by** *auto*
show *indexing-X* $\in \{..<$*dimension*$\} \rightarrow$ *carrier V*
using *indexing-X-n-in-carrier-V* **by** *auto*
qed
also have ... $=$ *k* $\cdot$ *iso-K-n-V x*
unfolding *iso-K-n-V-def* **..**
finally show *?thesis* **.**
qed
qed
qed

The following lemma states that the function *lin-comb* satisfies the additivity condition. It will be later used to prove that the function *iso-V-K-n* is also an additive function.

lemma
lin-comb-additivity:
assumes *x*: $x \in$ *carrier V*
and *y*: $y \in$ *carrier V*
shows *lin-comb* $(x \oplus_V y) = (\lambda i.$ *lin-comb x i* $\oplus$ *lin-comb y i*$)$
apply (*subst lin-comb-def*)
proof (*rule the1-equality*)
show $\exists!f.\ f \in$ *coefficients-function X* $\wedge$ *linear-combination f X* $= x \oplus_V y$
using *linear-combination-unique* [*OF V.a-closed* [*OF x y*]] **.**
next
show $(\lambda i.$ *lin-comb x i* $\oplus$ *lin-comb y i*$) \in$ *coefficients-function X* $\wedge$
linear-combination $(\lambda i.$ *lin-comb x i* $\oplus$ *lin-comb y i*$)$ $X = x \oplus_V y$
proof (*rule conjI*)
show $(\lambda i.$ *lin-comb x i* $\oplus$ *lin-comb y i*$) \in$ *coefficients-function X*
using *lin-comb-is-coefficients-function* [*OF x*]
using *lin-comb-is-coefficients-function* [*OF y*]
unfolding *coefficients-function-def* **by** *auto*
show *linear-combination* $(\lambda i.$ *lin-comb x i* $\oplus$ *lin-comb y i*$)$ $X = x \oplus_V y$
proof $-$
have *linear-combination* $(\lambda i.$ *lin-comb x i* $\oplus$ *lin-comb y i*$)$ $X =$
$(\bigoplus_V ya \in X.$ (*lin-comb x ya* $\oplus$ *lin-comb y ya*) $\cdot$ *ya*$)$
unfolding *linear-combination-def* **..**
also have ... $= (\bigoplus_V ya \in X.$ (*lin-comb x ya* $\cdot$ *ya*) $\oplus_V$ (*lin-comb y ya* $\cdot$ *ya*)$)$
proof (*rule finsum-cong'*)
show $X = X$ **..**
show $(\lambda ya.$ *lin-comb x ya* $\cdot$ *ya* $\oplus_V$ *lin-comb y ya* $\cdot$ *ya*$) \in X \rightarrow$ *carrier V*
using *lin-comb-is-coefficients-function* [*OF x*]
using *lin-comb-is-coefficients-function* [*OF y*]
unfolding *coefficients-function-def*
using *mult-closed* **using** *good-set-X*
unfolding *good-set-def* **by** *blast*
fix *i*

assume i: $i \in X$
show $(\mathit{lin\text{-}comb}\ x\ i \oplus \mathit{lin\text{-}comb}\ y\ i) \cdot i = \mathit{lin\text{-}comb}\ x\ i \cdot i \oplus_V \mathit{lin\text{-}comb}\ y\ i \cdot i$
using *add-mult-distrib2*
using *lin-comb-is-coefficients-function* [*OF x*]
using *lin-comb-is-coefficients-function* [*OF y*]
unfolding *coefficients-function-def*
using *mult-closed i* **using** *good-set-X*
unfolding *good-set-def* **by** *blast*
qed
also have ... $= (\bigoplus_V ya{\in}X.\ (\mathit{lin\text{-}comb}\ x\ ya \cdot ya)) \oplus_V (\bigoplus_V ya{\in}X.\ (\mathit{lin\text{-}comb}\ y\ ya \cdot ya))$
using *V.finsum-addf* [*OF finite-X*,
of $(\lambda i.\ \mathit{lin\text{-}comb}\ x\ i \cdot i)$ $(\lambda i.\ \mathit{lin\text{-}comb}\ y\ i \cdot i)$]
using *lin-comb-is-coefficients-function* [*OF x*]
using *lin-comb-is-coefficients-function* [*OF y*]
unfolding *coefficients-function-def*
using *mult-closed* **using** *good-set-X*
unfolding *good-set-def* **by** *blast*
also have ... $= \mathit{linear\text{-}combination}\ (\mathit{lin\text{-}comb}\ x)\ X \oplus_V \mathit{linear\text{-}combination}\ (\mathit{lin\text{-}comb}\ y)\ X$
unfolding *linear-combination-def* [*symmetric*] **..**
also have ... $= x \oplus_V y$
unfolding *lin-comb-is-the-linear-combination* [*symmetric*, *OF x*]
unfolding *lin-comb-is-the-linear-combination* [*symmetric*, *OF y*] **..**
finally show *?thesis* **.**
qed
qed
qed

end

context *vector-space*
begin

lemma
finsum-mult-assocf:
assumes *x1*: $X \subseteq \mathit{carrier}\ V$
and *x2*: *finite X*
and k: $k \in \mathit{carrier}\ K$
and f: $f \in X \rightarrow \mathit{carrier}\ K$
and g: $g \in X \rightarrow \mathit{carrier}\ V$
shows $(\bigoplus_V y{\in}X.\ (k \otimes f\ y) \cdot g\ y) = k \cdot (\bigoplus_V y{\in}X.\ f\ y \cdot g\ y)$
using *x2 x1 f g* **proof** (*induct X*)
case *empty*
show *?case*
using *scalar-mult-zeroV-is-zeroV* [*OF k*] **by** *simp*
next
case (*insert x F*)
have F: $F \subseteq \mathit{carrier}\ V$ **using** *insert.prems* (*1*) **by** *simp*

have *f*: $f \in F \to$ *carrier K* **and** *g*: $g \in F \to$ *carrier V*
and *kfg*: $(\lambda y.\ (k \otimes f\ y) \cdot g\ y) \in F \to$ *carrier V*
and *fg*: $(\lambda y.\ f\ y \cdot g\ y) \in F \to$ *carrier V*
and *kfgx*: $(k \otimes f\ x) \cdot g\ x \in$ *carrier V*
and *fgx*: $f\ x \cdot g\ x \in$ *carrier V*
and *fx*: $f\ x \in$ *carrier K* **and** *gx*: $g\ x \in$ *carrier V*
using *insert.prems* (*2*,*3*) *k*
using *mult-closed* **by** *blast+*
have *finsum-closed*: $(\bigoplus_V y{\in}F.\ (f\ y \cdot g\ y)) \in$ *carrier V*
by (*rule finsum-closed* [*OF insert.hyps* (*1*) *fg*])
have *hypo* :$(\bigoplus_V y{\in}F.\ (k \otimes f\ y) \cdot g\ y) = k \cdot (\bigoplus_V y{\in}F.\ f\ y \cdot g\ y)$
using *insert.hyps* (*3*) [*OF F f g*] **.**
show *?case* **thm** *insert.hyps* (*2*)
unfolding *finsum-insert* [*OF insert.hyps* (*1*,*2*) *kfg*, *OF kfgx*]
unfolding *finsum-insert* [*OF insert.hyps* (*1*,*2*) *fg*, *OF fgx*]
unfolding *add-mult-distrib1* [*OF fgx finsum-closed k*]
unfolding *mult-assoc* [*OF gx k fx*]
unfolding *hypo* **..**
qed

lemma
finsum-mult-assoc:
assumes *k*: $k \in$ *carrier K*
and *f*: $f \in \{..n\} \to$ *carrier K*
and *g*: $g \in \{..n\} \to$ *carrier V*
shows $(\bigoplus_V y{\in}\{..n{::}nat\}.\ (k \otimes f\ y) \cdot g\ y) = k \cdot (\bigoplus_V y{\in}\{..n\}.\ f\ y \cdot g\ y)$
using *f g* **proof** (*induct n*)
case *0*
show *?case*
proof −
have $(\bigoplus_V y{\in}\{..0\}.\ (k \otimes f\ y) \cdot g\ y) = (\bigoplus_V y{\in}\{0\}.\ (k \otimes f\ y) \cdot g\ y)$ **by** *simp*
also have ... $= (k \otimes f\ 0) \cdot g\ 0 \oplus_V (\bigoplus_V y{\in}\{\}.\ (k \otimes f\ y) \cdot g\ y)$
apply (*rule finsum-insert* [*of* {} *0::nat* $(\lambda i.\ (k \otimes f\ i) \cdot g\ i)$])
using *0.prems k* **using** *mult-closed* [*of g 0 k* $\otimes$ *f 0*] **by** *auto*
also have ... $= (k \otimes f\ 0) \cdot g\ 0$
unfolding *finsum-empty*
using *r-zero* [*OF mult-closed* [*of g 0 k* $\otimes$ *f 0*]]
using *0.prems k* **by** *auto*
finally have *lhs*: $(\bigoplus_V y{\in}\{..0\}.\ (k \otimes f\ y) \cdot g\ y) = (k \otimes f\ 0) \cdot g\ 0$ **.**
have $k \cdot (\bigoplus_V y{\in}\{..0\}.\ f\ y \cdot g\ y) = k \cdot (\bigoplus_V y{\in}\{0\}.\ f\ y \cdot g\ y)$ **by** *simp*
also have ... $= k \cdot (f\ 0 \cdot g\ 0 \oplus_V (\bigoplus_V y{\in}\{\}.\ f\ y \cdot g\ y))$
using *finsum-insert* [*of* {} *0::nat* $(\lambda i.\ f\ i \cdot g\ i)$]
using *0.prems k* **using** *mult-closed* [*of g 0 f 0*] **by** *fastsimp*
also have ... $= k \cdot (f\ 0 \cdot g\ 0 \oplus_V \mathbf{0}_V)$
unfolding *finsum-empty* **..**
also have ... $= k \cdot (f\ 0 \cdot g\ 0)$
using *r-zero* [*OF mult-closed* [*of g 0 f 0*]]
using *0.prems* **by** *force*
also have ... $= (k \otimes f\ 0) \cdot g\ 0$

using *mult-assoc* [*symmetric*]
using *0.prems k* **using** *mult-closed* [*of g 0 f 0*] **by** *auto*
finally have *rhs*: $k \cdot (\bigoplus_V y \in \{..0\}.\ f\ y \cdot g\ y) = (k \otimes f\ 0) \cdot g\ 0$ **.**
show *?case*
unfolding *lhs rhs* **..**
qed
next
case (*Suc n*)
have *f*: $f \in \{..n\} \to carrier\ K$ **and** *g*: $g \in \{..n\} \to carrier\ V$
and *fSuc*: $f\ (Suc\ n) \in carrier\ K$ **and** *gSuc*: $g\ (Suc\ n) \in carrier\ V$
and *fgSuc*: $f\ (Suc\ n) \cdot g\ (Suc\ n) \in carrier\ V$
using *Suc.prems* **using** *mult-closed* **by** *auto*
have *fg*: $(\lambda i.\ f\ i \cdot g\ i) \in \{..n\} \to carrier\ V$
and *kfg*: $(\lambda i.\ (k \otimes f\ i) \cdot g\ i) \in \{..n\} \to carrier\ V$
and *kfgSuc*: $(k \otimes f\ (Suc\ n)) \cdot g\ (Suc\ n) \in carrier\ V$
using *Suc.prems f g k* **using** *mult-closed* **by** *blast+*
have *finsum-closed*: $(\bigoplus_V y \in \{..n\}.\ (f\ y \cdot g\ y)) \in carrier\ V$
using *finsum-closed* [*OF - fg*] **by** *fast*
have *hypo* :$(\bigoplus_V y \in \{..n\}.\ (k \otimes f\ y) \cdot g\ y) = k \cdot (\bigoplus_V y \in \{..n\}.\ f\ y \cdot g\ y)$
by (*rule Suc.hyps* [*OF f g*])
show *?case*
proof −
have $(\bigoplus_V y \in \{..Suc\ n\}.\ (k \otimes f\ y) \cdot g\ y) = (\bigoplus_V y \in insert\ (Suc\ n)\ \{..n\}.\ (k \otimes f\ y) \cdot g\ y)$
unfolding *atMost-Suc* **..**
also have ... $= (k \otimes f\ (Suc\ n)) \cdot g\ (Suc\ n) \oplus_V (\bigoplus_V y \in \{..n\}.\ (k \otimes f\ y) \cdot g\ y)$
using *finsum-insert* [*OF - - kfg*, *of Suc n*] **using** *kfgSuc* **by** *fastsimp*
finally have *lhs*: $(\bigoplus_V y \in \{..Suc\ n\}.\ (k \otimes f\ y) \cdot g\ y) =$
$(k \otimes f\ (Suc\ n)) \cdot g\ (Suc\ n) \oplus_V (\bigoplus_V y \in \{..n\}.\ (k \otimes f\ y) \cdot g\ y)$ **.**
have $k \cdot (\bigoplus_V y \in \{..Suc\ n\}.\ f\ y \cdot g\ y) = k \cdot (\bigoplus_V y \in insert\ (Suc\ n)\ \{..n\}.\ (f\ y \cdot g\ y))$
unfolding *atMost-Suc* **..**
also have ... $= k \cdot (f\ (Suc\ n) \cdot g\ (Suc\ n) \oplus_V (\bigoplus_V y \in \{..n\}.\ (f\ y \cdot g\ y)))$
using *finsum-insert* [*OF - - fg*, *of Suc n*] **using** *fgSuc* **by** *fastsimp*
also have ... $= k \cdot f\ (Suc\ n) \cdot g\ (Suc\ n) \oplus_V k \cdot (\bigoplus_V y \in \{..n\}.\ (f\ y \cdot g\ y))$
unfolding *add-mult-distrib1* [*OF fgSuc finsum-closed k*] **..**
also have ... $= (k \otimes f\ (Suc\ n)) \cdot g\ (Suc\ n) \oplus_V k \cdot (\bigoplus_V y \in \{..n\}.\ (f\ y \cdot g\ y))$
unfolding *mult-assoc* [*OF gSuc k fSuc*] **..**
also have ... $= (k \otimes f\ (Suc\ n)) \cdot g\ (Suc\ n) \oplus_V (\bigoplus_V y \in \{..n\}.\ (k \otimes f\ y) \cdot g\ y)$
unfolding *hypo* **..**
finally have *rhs*: $k \cdot (\bigoplus_V y \in \{..Suc\ n\}.\ f\ y \cdot g\ y) =$
$(k \otimes f\ (Suc\ n)) \cdot g\ (Suc\ n) \oplus_V (\bigoplus_V y \in \{..n\}.\ (k \otimes f\ y) \cdot g\ y)$ **.**
show *?case* **unfolding** *lhs rhs* **..**
qed
qed

lemma
finsum-mult-assoc-le:
assumes *k*: $k \in carrier\ K$

and f: $f \in \{..<n\} \rightarrow$ *carrier K*
and g: $g \in \{..<n\} \rightarrow$ *carrier V*
shows $(\bigoplus_V y{\in}\{..<n{::}nat\}.\ (k \otimes f\ y) \cdot g\ y) = k \cdot (\bigoplus_V y{\in}\{..<n\}.\ f\ y \cdot g\ y)$
proof (*cases n*)
case *0*
show *?thesis* **unfolding** *0* **using** *scalar-mult-zeroV-is-zeroV* [*OF k*] **by** *simp*
next
case (*Suc k*)
have f: $f \in \{..k\} \rightarrow$ *carrier K* **and** g: $g \in \{..k\} \rightarrow$ *carrier V*
using *f g*
unfolding *Suc lessThan-Suc-atMost* **by** *fast+*
show *?thesis*
unfolding *Suc*
unfolding *lessThan-Suc-atMost*
using *finsum-mult-assoc* [*OF k f g*] **.**
qed

end

context *finite-dimensional-vector-space*
begin

The following lemma states that the function *lin-comb* satisfies the homogeneous property. It will be later used to prove that the function *iso-V-K-n* is homogeneous:

lemma
lin-comb-homogeneity:
assumes k: $k \in$ *carrier K*
and x: $x \in$ *carrier V*
shows *lin-comb* $(k \cdot x) = (\lambda i.\ k \otimes$ *lin-comb* $x\ i)$
apply (*subst lin-comb-def*)
proof (*rule the1-equality*)
show $\exists!f.\ f \in$ *coefficients-function* $X \wedge$ *linear-combination* $f\ X = k \cdot x$
using *linear-combination-unique* [*OF mult-closed* [*OF x k*]] **.**
next
show $(\lambda i.\ k \otimes$ *lin-comb* $x\ i) \in$ *coefficients-function* $X \wedge$
linear-combination $(\lambda i.\ k \otimes$ *lin-comb* $x\ i)\ X = k \cdot x$
proof (*rule conjI*)
show $(\lambda i.\ k \otimes$ *lin-comb* $x\ i) \in$ *coefficients-function* X
using *lin-comb-is-coefficients-function* [*OF x*]
unfolding *coefficients-function-def*
using k **by** *auto*
show *linear-combination* $(\lambda i.\ k \otimes$ *lin-comb* $x\ i)\ X = k \cdot x$
proof −
have *linear-combination* $(\lambda i.\ k \otimes$ *lin-comb* $x\ i)\ X =$
$(\bigoplus_V y{\in}X.\ (k \otimes$ *lin-comb* $x\ y) \cdot y)$
unfolding *linear-combination-def* **..**
also have ... $= k \cdot (\bigoplus_V y{\in}X.\ ($*lin-comb* $x\ y) \cdot y)$
apply (*rule finsum-mult-assocf* [*OF - finite-X k*])

using *lin-comb-is-coefficients-function* [*OF x*]
using *good-set-X*
unfolding *good-set-def coefficients-function-def* **by** *blast+*
also have ... = $k \cdot x$
unfolding *linear-combination-def* [*symmetric*]
unfolding *lin-comb-is-the-linear-combination* [*symmetric*, *OF x*] **..**
finally show *?thesis* **.**
qed
qed
qed

end

context *abelian-monoid*
begin

lemma *finsum-add'*:
assumes *f*: $f \in \{..<n\} \to$ *carrier G*
and *g*: $g \in \{..<n\} \to$ *carrier G*
shows ($\bigoplus i\in\{..<n::nat\}.\ f\ i \oplus g\ i$) = *finsum G f* $\{..<n\} \oplus$ *finsum G g* $\{..<n\}$
proof (*cases n*)
case *0*
show *?thesis*
unfolding *0* **by** *force*
next
case (*Suc n*)
show *?thesis*
using *f g* **unfolding** *Suc*
unfolding *lessThan-Suc-atMost*
using *finsum-add* [*of f n g*] **by** *fast*
qed

end

context *finite-dimensional-vector-space*
begin

The following lemma proves that the application *iso-V-K-n* is a linear map between *V* and *field.K-n K dimension*.

lemma *linear-map-iso-V-K-n*: *V-K-n.linear-map iso-V-K-n*
proof (*unfold V-K-n.linear-map-def*, *intro conjI*)
interpret *field K* **by** *intro-locales*
interpret *K-n*: *vector-space K K-n dimension K-n-scalar-product*
using *vector-space-K-n* **.**
show *V-K-n.additivity iso-V-K-n*
proof (*unfold V-K-n.additivity-def*, *rule ballI*, *rule ballI*)
fix *x y* **assume** *x*: $x \in$ *carrier V* **and** *y*: $y \in$ *carrier V*
show *iso-V-K-n* ($x \oplus_V y$) = *iso-V-K-n x* $\oplus_{K\text{-}n\ dimension}$ *iso-V-K-n y*
proof −

```
have iso-V-K-n (x ⊕V y) =
  (⨁K-n dimension i∈{..<dimension}. (λn. lin-comb (x ⊕V y) (indexing-X i)
⊗
  (if n = i then 1 else 0), dimension − 1))
  unfolding iso-V-K-n-def K-n-scalar-product-def
    ith-def vlen-def fst-conv snd-conv x-i-def ..
also have ... =
  (⨁K-n dimension i∈{..<dimension}.
    (λn. lin-comb x (indexing-X i) ⊗
      (if n = i then 1 else 0), dimension − 1)
    ⊕K-n dimension
    (λn. lin-comb y (indexing-X i) ⊗
       (if n = i then 1 else 0), dimension − 1))
proof (rule K-n.finsum-cong')
  show {..<dimension} = {..<dimension} ..
  show (λi. (λn. lin-comb x (indexing-X i) ⊗
             (if n = i then 1 else 0), dimension − 1)
            ⊕K-n dimension
           (λn. lin-comb y (indexing-X i) ⊗
             (if n = i then 1 else 0), dimension − 1))
        ∈ {..<dimension} → carrier (K-n dimension)
  proof
    fix xa assume xa: xa ∈ {..<dimension}
      show (λn. lin-comb x (indexing-X xa) ⊗ (if n = xa then 1 else 0),
dimension − 1)
        ⊕K-n dimension
       (λn. lin-comb y (indexing-X xa) ⊗ (if n = xa then 1 else 0), dimension
− 1)
        ∈ carrier (K-n dimension)
    proof (rule K-n.a-closed)
      have lx: lin-comb x (indexing-X xa) ∈ carrier K
        and ly: lin-comb y (indexing-X xa) ∈ carrier K
        using lin-comb-is-coefficients-function [OF x]
        using lin-comb-is-coefficients-function [OF y]
        using indexing-X-n-in-carrier-V [of xa] xa
        unfolding coefficients-function-def by auto
        show (λn. lin-comb x (indexing-X xa) ⊗ (if n = xa then 1 else 0),
dimension − 1)
        ∈ carrier (K-n dimension)
        unfolding K-n-def K-n-carrier-def ith-def vlen-def
        using xa lx by auto
        show (λn. lin-comb y (indexing-X xa) ⊗ (if n = xa then 1 else 0),
dimension − 1)
        ∈ carrier (K-n dimension)
        unfolding K-n-def K-n-carrier-def ith-def vlen-def
        using xa ly by auto
    qed
  qed
  fix i
```

assume *i*: $i \in \{..<\mathit{dimension}\}$
show (λn. *lin-comb* $(x \oplus_V y)$ (*indexing-X i*) $\otimes$ (*if* $n = i$ *then* **1** *else* **0**), *dimension* $-$ *1*) =
(λn. *lin-comb x* (*indexing-X i*) $\otimes$ (*if* $n = i$ *then* **1** *else* **0**), *dimension* $-$ *1*) $\oplus_{K\text{-}n\ dimension}$
(λn. *lin-comb y* (*indexing-X i*) $\otimes$ (*if* $n = i$ *then* **1** *else* **0**), *dimension* $-$ *1*)
proof (*unfold K-n-def K-n-add-def ith-def*, *simp*, *rule*)
fix *n*
have *lx*: *lin-comb x* (*indexing-X i*) $\in$ *carrier K*
and *ly*: *lin-comb y* (*indexing-X i*) $\in$ *carrier K*
and *lxy*: *lin-comb* $(x \oplus_V y)$ (*indexing-X i*) $\in$ *carrier K*
using *lin-comb-is-coefficients-function* [*OF x*]
using *lin-comb-is-coefficients-function* [*OF y*]
using *lin-comb-is-coefficients-function* [*OF V.a-closed* [*OF x y*]]
using *indexing-X-n-in-carrier-V* [*of i*] *i*
unfolding *coefficients-function-def* **by** *auto*
show *lin-comb* $(x \oplus_V y)$ (*indexing-X i*) $\otimes$ (*if* $n = i$ *then* **1** *else* **0**) =
lin-comb x (*indexing-X i*) $\otimes$ (*if* $n = i$ *then* **1** *else* **0**) $\oplus$
lin-comb y (*indexing-X i*) $\otimes$ (*if* $n = i$ *then* **1** *else* **0**)
proof (*cases* $n = i$)
case *False*
show *?thesis* **using** *False lx ly lxy* **by** *simp*
next
case *True*
show *?thesis* **using** *True lx ly lxy*
apply *simp*
using *lin-comb-additivity* [*OF x y*] **by** *presburger*
qed
qed
qed
also have ... = ($\bigoplus_{K\text{-}n\ dimension} i \in \{..<\mathit{dimension}\}$.
(λn. *lin-comb x* (*indexing-X i*) $\otimes$ (*if* $n = i$ *then* **1** *else* **0**), *dimension* $-$ *1*))
$\oplus_{K\text{-}n\ dimension}$
($\bigoplus_{K\text{-}n\ dimension} i \in \{..<\mathit{dimension}\}$. ($\lambda n$. *lin-comb y* (*indexing-X i*)
$\otimes$ (*if* $n = i$ *then* **1** *else* **0**), *dimension* $-$ *1*))
proof (*rule K-n.finsum-add'*)
show (λi. (λn. *lin-comb x* (*indexing-X i*) $\otimes$ (*if* $n = i$ *then* **1** *else* **0**), *dimension* $-$ *1*))
$\in \{..<\mathit{dimension}\} \to$ *carrier* (*field.K-n K dimension*)
proof
fix *xa* **assume** *xa*: $xa \in \{..<\mathit{dimension}\}$
have *i*: *indexing-X xa* $\in$ *carrier V*
using *indexing-X-n-in-carrier-V xa* **by** *fast*
have *lin-comb x* $\in$ *coefficients-function* (*carrier V*)
using *lin-comb-is-coefficients-function* [*OF x*]
unfolding *coefficients-function-def* **using** *good-set-X* **unfolding** *good-set-def* **by** *auto*
thus (λn. *lin-comb x* (*indexing-X xa*) $\otimes$ (*if* $n = xa$ *then* **1** *else* **0**), *dimension* $-$ *1*)

$\in$ *carrier* (*field.K-n K dimension*)
using *i xa*
unfolding *dimension-def*
unfolding *coefficients-function-def*
unfolding *K-n-def K-n-carrier-def ith-def vlen-def* **by** *force*
qed
show (λi. (λn. *lin-comb y* (*indexing-X i*) $\otimes$ (*if* $n = i$ *then* **1** *else* **0**), *dimension* $-$ *1*))
$\in$ {..<*dimension*} $\to$ *carrier* (*field.K-n K dimension*)
proof
fix *xa* **assume** *xa*: *xa* $\in$ {..<*dimension*}
have *i*: *indexing-X xa* $\in$ *carrier V*
using *indexing-X-n-in-carrier-V xa* **by** *fast*
have *cf-lc*: *lin-comb x* $\in$ *coefficients-function* (*carrier V*)
using *lin-comb-is-coefficients-function* [*OF x*]
unfolding *coefficients-function-def* **using** *good-set-X* **unfolding** *good-set-def* **by** *auto*
thus (λn. *lin-comb y* (*indexing-X xa*) $\otimes$ (*if* $n = xa$ *then* **1** *else* **0**), *dimension* $-$ *1*)
$\in$ *carrier* (*field.K-n K dimension*)
using *i xa*
unfolding *dimension-def*
unfolding *coefficients-function-def*
unfolding *K-n-def K-n-carrier-def ith-def vlen-def*
proof (*auto*)
show *lin-comb y* (*indexing-X xa*) $\otimes$ **1** $\in$ *carrier K*
by (*metis coefficients-function-Pi i*
lin-comb-is-coefficients-function m-closed one-closed y)
show *lin-comb y* (*indexing-X xa*) $\otimes$ **0** $\in$ *carrier K*
by (*metis K.add.one-closed coefficients-function-Pi i*
lin-comb-is-coefficients-function m-closed y)
show *lin-comb y* (*indexing-X xa*) $\otimes$ **0** = **0**
by (*metis coefficients-function-Pi i*
lin-comb-is-coefficients-function r-null y)
qed
qed
qed
also have ... = *iso-V-K-n x* $\oplus_{field.K\text{-}n\ K\ dimension}$ *iso-V-K-n y*
unfolding *iso-V-K-n-def K-n-scalar-product-def*
ith-def vlen-def fst-conv snd-conv x-i-def **..**
finally show *?thesis* **.**
qed
qed
show *V-K-n.homogeneity iso-V-K-n*
proof (*unfold V-K-n.homogeneity-def*, *rule ballI*, *rule ballI*)
fix *k x*
assume *k*: $k \in$ *carrier K* **and** *x*: $x \in$ *carrier V*
show *iso-V-K-n* ($k \cdot x$) = *K-n-scalar-product k* (*iso-V-K-n x*)
proof $-$

have *iso-V-K-n* $(k \cdot x)$ =
($\bigoplus_{K\text{-}n\ dimension} i \in \{..<dimension\}$. *K-n-scalar-product*
(*lin-comb* $(k \cdot x)$ (*indexing-X* i)) (*x-i* i *dimension*))
unfolding *iso-V-K-n-def* **..**
also have ... = ($\bigoplus_{K\text{-}n\ dimension} i \in \{..<dimension\}$. *K-n-scalar-product*
($k \otimes$ (*lin-comb* x (*indexing-X* i))) (*x-i* i *dimension*))
proof (*rule K-n.finsum-cong'*)
show $\{..<dimension\} = \{..<dimension\}$ **..**
show (λi. *K-n-scalar-product* ($k \otimes$ *lin-comb* x (*indexing-X* i)) (*x-i* i *dimension*))
$\in \{..<dimension\} \to$ *carrier* (*K-n dimension*)
proof
fix *xa* **assume** *xa*: $xa \in \{..<dimension\}$
show *K-n-scalar-product* ($k \otimes$ *lin-comb* x (*indexing-X xa*)) (*x-i xa dimension*)
$\in$ *carrier* (*K-n dimension*)
proof (*rule K-n-scalar-product-closed*)
show $k \otimes$ *lin-comb* x (*indexing-X xa*) $\in$ *carrier K*
using *k lin-comb-is-coefficients-function* [*OF x*]
unfolding *coefficients-function-def*
using *indexing-X-n-in-carrier-V* [*of xa*] *xa* **by** *auto*
show *x-i xa dimension* $\in$ *carrier* (*K-n dimension*)
using *x-i-closed xa* **by** *simp*
qed
qed
fix i
assume i: $i \in \{..<dimension\}$
show *K-n-scalar-product* (*lin-comb* $(k \cdot x)$ (*indexing-X* i)) (*x-i* i *dimension*)
=
K-n-scalar-product ($k \otimes$ *lin-comb* x (*indexing-X* i)) (*x-i* i *dimension*)
unfolding *lin-comb-homogeneity* [*OF k x*] **..**
qed
also have ... = *K-n-scalar-product* k ($\bigoplus_{K\text{-}n\ dimension} i \in \{..<dimension\}$.
K-n-scalar-product
(*lin-comb* x (*indexing-X* i)) (*x-i* i *dimension*))
apply (*rule K-n.finsum-mult-assoc-le* [*OF k*])
using *k lin-comb-is-coefficients-function* [*OF x*]
using *indexing-X-n-in-carrier-V x-i-closed*
unfolding *coefficients-function-def* **by** *auto*
also have ... = *K-n-scalar-product k* (*iso-V-K-n x*)
unfolding *iso-V-K-n-def* [*symmetric*] **..**
finally show *?thesis* **.**
qed
qed
qed

end

lemma

lessThan-remove:
assumes *i*: (*i*::*nat*) ∈ {..<*k*}
shows {..<*k*} = ({..<*k*} − {*i*}) ∪ {*i*}
using *i* **by** *blast*

context *finite-dimensional-vector-space*
begin

The functions *iso-K-n-V* and *iso-V-K-n* behave correctly in their respective domains:

lemma *iso-V-K-n-Pi*: *iso-V-K-n* ∈ *carrier V* → *carrier* (*K-n dimension*)
proof −
interpret *K-n*: *vector-space K K-n dimension K-n-scalar-product* **using** *vector-space-K-n*
.
show *?thesis*
proof
fix *x* **assume** *x*: *x* ∈ *carrier V*
show *iso-V-K-n x* ∈ *carrier* (*K-n dimension*)
unfolding *iso-V-K-n-def*
proof (*rule K-n.finsum-closed*)
show *finite* {..<*dimension*} **by** *simp*
show (λ*i*. *K-n-scalar-product* (*lin-comb x* (*indexing-X i*)) (*field.x-i K i dimension*))
∈ {..<*dimension*} → *carrier* (*K-n dimension*)
proof
fix *xa* **assume** *xa*: *xa* ∈ {..<*dimension*}
show *K-n-scalar-product* (*lin-comb x* (*indexing-X xa*)) (*x-i xa dimension*)
∈ *carrier* (*K-n dimension*)
proof (*rule K-n-scalar-product-closed*)
show *lin-comb x* (*indexing-X xa*) ∈ *carrier K*
using *lin-comb-is-coefficients-function* [*OF x*]
using *indexing-X-n-in-carrier-V xa*
unfolding *coefficients-function-def* **by** *auto*
show *x-i xa dimension* ∈ *carrier* (*K-n dimension*)
using *x-i-closed xa* **by** *simp*
qed
qed
qed
qed
qed

lemma *iso-K-n-V-Pi*: **shows** *iso-K-n-V* ∈ *carrier* (*K-n dimension*) → *carrier V*
proof −
interpret *K-n*: *vector-space K K-n dimension K-n-scalar-product* **using** *vector-space-K-n*
.
show *?thesis*
proof
fix *x* **assume** *x*: *x* ∈ *carrier* (*K-n dimension*)
show *iso-K-n-V x* ∈ *carrier V*

proof (*unfold iso-K-n-V-def*)
show $(\bigoplus_V i \in \{..<dimension\}.\ fst\ x\ i \cdot indexing\text{-}X\ i) \in carrier\ V$
proof (*rule finsum-closed*)
show *finite* $\{..<dimension\}$ **by** *simp*
show $(\lambda i.\ fst\ x\ i \cdot indexing\text{-}X\ i) \in \{..<dimension\} \to carrier\ V$
using *mult-closed*
using *indexing-X-n-in-carrier-V*
using *x* **unfolding** *K-n-def K-n-carrier-def ith-def vlen-def* **by** *auto*
qed
qed
qed
qed

lemma
lin-comb-fimsum-candidate:
assumes x: $x \in carrier\ (K\text{-}n\ dimension)$
shows $(\bigoplus_V y \in X.\ fst\ x\ (preim2\text{-}comp\ y) \cdot y) = (\bigoplus_V i \in \{..<dimension\}.\ fst\ x\ i \cdot indexing\text{-}X\ i)$
proof (*rule finsum-cong''* [*of - indexing-X*])
show *finite* $\{..<dimension\}$ **by** *simp*
show *bij-betw indexing-X* $\{..<dimension\}$ X **by** (*metis indexing-X-bij*)
show $(\lambda y.\ fst\ x\ (preim2\text{-}comp\ y) \cdot y) \in X \to carrier\ V$
proof
fix *xa* **assume** *xa*: $xa \in X$
show $fst\ x\ (preim2\text{-}comp\ xa) \cdot xa \in carrier\ V$
proof (*rule mult-closed*)
show $xa \in carrier\ V$ **using** *xa* **using** *good-set-X* **unfolding** *good-set-def* **by** *fast*
show $fst\ x\ (preim2\text{-}comp\ xa) \in carrier\ K$
using *preim2-comp-in-dimension* [*OF xa*] *x*
unfolding *K-n-def K-n-carrier-def ith-def vlen-def* **by** *auto*
qed
qed
show $(\lambda i.\ fst\ x\ i \cdot indexing\text{-}X\ i) \in \{..<dimension\} \to carrier\ V$
proof
fix *xa* **assume** *xa*: $xa \in \{..<dimension\}$
show $fst\ x\ xa \cdot indexing\text{-}X\ xa \in carrier\ V$
proof (*rule mult-closed*)
show $indexing\text{-}X\ xa \in carrier\ V$ **using** *indexing-X-n-in-carrier-V xa* **by** *simp*

show $fst\ x\ xa \in carrier\ K$ **using** *x xa*
unfolding *K-n-def K-n-carrier-def ith-def vlen-def* **by** *auto*
qed
qed
show $\bigwedge xa.\ xa \in \{..<dimension\}$ *=simp=>*
$fst\ x\ xa \cdot indexing\text{-}X\ xa =$
$fst\ x\ (preim2\text{-}comp\ (indexing\text{-}X\ xa)) \cdot indexing\text{-}X\ xa$
using *preim2-comp-iso-nat-X-id*
unfolding *iso-nat-X-def* **by** *simp*

qed

The following lemma expresses how to write down the *lin-comb* of a finite sum of the elements of the basis:

lemma
lin-comb-linear-combination-candidate:
assumes *x*: $x \in carrier\ (K\text{-}n\ dimension)$
shows *lin-comb* $(\bigoplus_V i \in \{..<dimension\}.\ fst\ x\ i \cdot indexing\text{-}X\ i) = (\lambda y.\ fst\ x\ (preim2\text{-}comp\ y))$
unfolding *lin-comb-def*
proof (*rule the1-equality*)
have *finsum-closed*: $(\bigoplus_V i \in \{..<dimension\}.\ fst\ x\ i \cdot indexing\text{-}X\ i) \in carrier\ V$
proof (*rule finsum-closed*)
show *finite* $\{..<dimension\}$ **by** *simp*
show $(\lambda i.\ fst\ x\ i \cdot indexing\text{-}X\ i) \in \{..<dimension\} \to carrier\ V$
proof
fix *xa* **assume** *xa*: $xa \in \{..<dimension\}$
show $fst\ x\ xa \cdot indexing\text{-}X\ xa \in carrier\ V$
proof (*rule mult-closed*)
show $indexing\text{-}X\ xa \in carrier\ V$ **using** *indexing-X-n-in-carrier-V xa* **by** *simp*
show $fst\ x\ xa \in carrier\ K$ **using** *x xa*
unfolding *K-n-def K-n-carrier-def ith-def vlen-def* **by** *auto*
qed
qed
qed
show $\exists!f.\ f \in coefficients\text{-}function\ X \wedge$
$linear\text{-}combination\ f\ X = (\bigoplus_V i \in \{..<dimension\}.\ fst\ x\ i \cdot indexing\text{-}X\ i)$
by (*rule linear-combination-unique* [*OF finsum-closed*])
show $(\lambda y.\ fst\ x\ (preim2\text{-}comp\ y)) \in coefficients\text{-}function\ X \wedge$
$linear\text{-}combination\ (\lambda y.\ fst\ x\ (preim2\text{-}comp\ y))\ X = (\bigoplus_V i \in \{..<dimension\}.\ fst\ x\ i \cdot indexing\text{-}X\ i)$
proof (*rule conjI*)
show $linear\text{-}combination\ (\lambda y.\ fst\ x\ (preim2\text{-}comp\ y))\ X =$
$(\bigoplus_V i \in \{..<dimension\}.\ fst\ x\ i \cdot indexing\text{-}X\ i)$
using *lin-comb-fimsum-candidate* [*OF x*]
unfolding *linear-combination-def* **.**
show $(\lambda y.\ fst\ x\ (preim2\text{-}comp\ y)) \in coefficients\text{-}function\ X$
unfolding *coefficients-function-def*
using *preim2-comp-in-dimension*
using *x*
unfolding *K-n-def K-n-carrier-def ith-def vlen-def*
unfolding *preim2-comp-def* **by** *auto*
qed
qed

With the previous lemmas, we can now prove that *iso-V-K-n* is a bijection between the correspoding carrier sets:

lemma *iso-V-K-n-bij*: **shows** *bij-betw iso-V-K-n* (*carrier V*) (*carrier* (*K-n dimen-*

sion))
proof (*rule bij-betwI* [*of - - - iso-K-n-V*])
interpret *K-n*: *vector-space K K-n dimension K-n-scalar-product* **using** *vector-space-K-n*
.
show *iso-V-K-n* $\in$ *carrier V* $\rightarrow$ *carrier* (*K-n dimension*) **by** (*rule iso-V-K-n-Pi*)
show *iso-K-n-V* $\in$ *carrier* (*K-n dimension*) $\rightarrow$ *carrier V* **by** (*rule iso-K-n-V-Pi*)
fix *x* **assume** *x*: *x* $\in$ *carrier V*
show *iso-K-n-V* (*iso-V-K-n x*) = *x*
apply (*subst* (*2*) *lin-comb-is-the-linear-combination-indexing* [*OF x*])
unfolding *iso-K-n-V-def*
proof (*rule finsum-cong'*)
show {..<*dimension*} = {..<*dimension*} **by** *simp*
show (λi. *lin-comb x* (*indexing-X i*) $\cdot$ *indexing-X i*) $\in$ {..<*dimension*} $\rightarrow$ *carrier V*
proof
fix *xa* **assume** *xa*: *xa* $\in$ {..<*dimension*}
show *lin-comb x* (*indexing-X xa*) $\cdot$ *indexing-X xa* $\in$ *carrier V*
apply (*rule mult-closed*)
using *indexing-X-n-in-carrier-V* [*of xa*] *xa*
using *lin-comb-is-coefficients-function* [*OF x*]
unfolding *coefficients-function-def* **by** *auto*
qed
fix *i* **assume** *i*: *i* $\in$ {..<*dimension*}
show *fst* (*iso-V-K-n x*) *i* $\cdot$ *indexing-X i* =
lin-comb x (*indexing-X i*) $\cdot$ *indexing-X i*
proof –
have *fst* (*iso-V-K-n x*) *i* = *fst* ($\bigoplus_{K\text{-}n\ dimension} i\in\{..<dimension\}$. *K-n-scalar-product* (*lin-comb x* (*indexing-X i*)) (*x-i i dimension*)) *i*
unfolding *iso-V-K-n-def* **..**
also have ... = *fst* (λi. *if i* $\in$ {..<*dimension*} *then* (*lin-comb x* (*indexing-X i*)) *else* **0**, *dimension* – *1*) *i*
proof –
have ($\bigoplus_{K\text{-}n\ dimension} i\in\{..<dimension\}$. *K-n-scalar-product* (*lin-comb x* (*indexing-X i*)) (*x-i i dimension*)) =
(λi. *if i* $\in$ {..<*dimension*} *then* (*lin-comb x* (*indexing-X i*)) *else* **0**, *dimension* – *1*)
apply (*rule lambda-finsum* [*symmetric*, *of dimension* (λi. *lin-comb x* (*indexing-X i*)) *dimension*])
using *lin-comb-is-coefficients-function* [*OF x*]
using *indexing-X-n-in-carrier-V*
unfolding *coefficients-function-def* **by** *auto*
thus *?thesis* **by** *simp*
qed
also have ... = (*lin-comb x* (*indexing-X i*)) **using** *i* **by** *fastsimp*
finally show *?thesis* **by** *simp*
qed
qed
next
fix *y*

assume *y*: $y \in$ *carrier* (*K-n dimension*)
show *iso-V-K-n* (*iso-K-n-V y*) = *y*
proof −
have *iso-V-K-n* (*iso-K-n-V y*) = ($\bigoplus_{K\text{-}n\ dimension} i \in \{..<dimension\}$. *K-n-scalar-product* (*lin-comb* (*iso-K-n-V y*) (*indexing-X i*)) (*x-i i dimension*)) **unfolding** *iso-V-K-n-def*
..
also have ... = (λi. *if* $i \in \{..<dimension\}$ *then lin-comb* (*iso-K-n-V y*) (*indexing-X i*) *else* **0**,
dimension − *1*)
proof (*rule lambda-finsum* [
symmetric, *of dimension* (λi. (*lin-comb* (*iso-K-n-V y*) (*indexing-X i*))) *dimension*])
show *dimension* $\leq$ *dimension* **by** *fast*
show $\forall$ $i \in \{..<dimension\}$. *lin-comb* (*iso-K-n-V y*) (*indexing-X i*) $\in$ *carrier K*
proof (*rule ballI*)
fix *i* **assume** *i*: $i \in \{..<dimension\}$
have *lin-comb* (*iso-K-n-V y*) $\in$ $\{f.\ f \in$ *carrier V* $\rightarrow$ *carrier K*$\}$
using *lin-comb-is-coefficients-function* [*of iso-K-n-V y*]
using *iso-K-n-V-Pi y*
unfolding *coefficients-function-def*
using *good-set-X* **unfolding** *good-set-def* **by** *force*
thus *lin-comb* (*iso-K-n-V y*) (*indexing-X i*) $\in$ *carrier K*
using *indexing-X-n-in-carrier-V i* **by** *auto*
qed
qed
also have ... = (λi. *if* $i \in \{..<dimension\}$ *then fst y i else* **0**, *dimension* − *1*)
proof (*rule*, *rule conjI*)
show *dimension* − *1* = *dimension* − *1* **by** (*rule refl*)
show (λi. *if* $i \in \{..<dimension\}$ *then lin-comb* (*iso-K-n-V y*) (*indexing-X i*) *else* **0**) =
(λi. *if* $i \in \{..<dimension\}$ *then fst y i else* **0**)
proof
fix *i*
show (*if* $i \in \{..<dimension\}$ *then lin-comb* (*iso-K-n-V y*) (*indexing-X i*) *else* **0**) =
(*if* $i \in \{..<dimension\}$ *then fst y i else* **0**)
proof (*cases* $i \in \{..<dimension\}$)
case *False* **show** *?thesis* **using** *False* **by** *simp*
next
case *True*
have *lin-comb* (*iso-K-n-V y*) (*indexing-X i*) = *fst y i*
unfolding *iso-K-n-V-def*
unfolding *lin-comb-linear-combination-candidate* [*OF y*]
using *preim2-comp-iso-nat-X-id* [*OF True*]
unfolding *iso-nat-X-def* **by** *simp*
thus *?thesis* **by** *simp*
qed
qed
qed

also have ... = y
  unfolding *x-in-carrier* [*symmetric*, *OF y*] by (*rule refl*)
finally show *?thesis* by *fast*
qed
qed

lemma *iso-K-n-V-bij*: **shows** *bij-betw iso-K-n-V* (*carrier* (*K-n dimension*)) (*carrier V*)
proof (*rule bij-betwI* [*of - - - iso-V-K-n*])
interpret *K-n*: *vector-space K K-n dimension K-n-scalar-product* **using** *vector-space-K-n*
.
show *iso-V-K-n* $\in$ *carrier V* $\rightarrow$ *carrier* (*K-n dimension*) **by** (*rule iso-V-K-n-Pi*)
show *iso-K-n-V* $\in$ *carrier* (*K-n dimension*) $\rightarrow$ *carrier V* **by** (*rule iso-K-n-V-Pi*)
fix *x* **assume** *x*: $x \in$ *carrier V*
show *iso-K-n-V* (*iso-V-K-n x*) = *x*
apply (*subst* (*2*) *lin-comb-is-the-linear-combination-indexing* [*OF x*])
unfolding *iso-K-n-V-def*
proof (*rule finsum-cong'*)
show {..<*dimension*} = {..<*dimension*} **by** *simp*
show (λi. *lin-comb x* (*indexing-X i*) $\cdot$ *indexing-X i*) $\in$ {..<*dimension*} $\rightarrow$ *carrier V*
proof
fix *xa* **assume** *xa*: $xa \in$ {..<*dimension*}
show *lin-comb x* (*indexing-X xa*) $\cdot$ *indexing-X xa* $\in$ *carrier V*
apply (*rule mult-closed*)
using *indexing-X-n-in-carrier-V* [*of xa*] *xa*
using *lin-comb-is-coefficients-function* [*OF x*]
unfolding *coefficients-function-def* **by** *auto*
qed
fix *i* **assume** *i*: $i \in$ {..<*dimension*}
show *fst* (*iso-V-K-n x*) *i* $\cdot$ *indexing-X i* =
lin-comb x (*indexing-X i*) $\cdot$ *indexing-X i*
proof −
have *fst* (*iso-V-K-n x*) *i* = *fst* ($\bigoplus_{K\text{-}n\ dimension} i \in \{..<dimension\}$. *K-n-scalar-product* (*lin-comb x* (*indexing-X i*)) (*x-i i dimension*)) *i*
unfolding *iso-V-K-n-def* **..**
also have ... = *fst* (λi. *if* $i \in$ {..<*dimension*} *then* (*lin-comb x* (*indexing-X i*)) *else* **0**, *dimension* − *1*) *i*
proof −
have ($\bigoplus_{K\text{-}n\ dimension} i \in \{..<dimension\}$. *K-n-scalar-product* (*lin-comb x* (*indexing-X i*)) (*x-i i dimension*)) =
(λi. *if* $i \in$ {..<*dimension*} *then* (*lin-comb x* (*indexing-X i*)) *else* **0**, *dimension* − *1*)
apply (*rule lambda-finsum* [*symmetric*, *of dimension* (λi. *lin-comb x* (*indexing-X i*)) *dimension*])
using *lin-comb-is-coefficients-function* [*OF x*]
using *indexing-X-n-in-carrier-V*
unfolding *coefficients-function-def* **by** *auto*
thus *?thesis* **by** *simp*

qed
also have ... = (*lin-comb x* (*indexing-X i*)) **using** *i* **by** *fastsimp*
finally show *?thesis* **by** *simp*
qed
qed
next
fix *y*
assume *y*: $y \in$ *carrier* (*K-n dimension*)
show *iso-V-K-n* (*iso-K-n-V y*) = *y*
proof −
have *iso-V-K-n* (*iso-K-n-V y*) = ($\bigoplus_{K\text{-}n\ dimension}$ $i \in \{..<dimension\}$. *K-n-scalar-product* (*lin-comb* (*iso-K-n-V y*) (*indexing-X i*)) (*x-i i dimension*)) **unfolding** *iso-V-K-n-def*
..
also have ... = (λi. *if* $i \in \{..<dimension\}$ *then lin-comb* (*iso-K-n-V y*) (*indexing-X i*) *else* **0**,
dimension − *1*)
proof (*rule lambda-finsum* [
symmetric, *of dimension* (λi. (*lin-comb* (*iso-K-n-V y*) (*indexing-X i*))) *dimension*])
show *dimension* $\leq$ *dimension* **by** *fast*
show $\forall i \in \{..<dimension\}$. *lin-comb* (*iso-K-n-V y*) (*indexing-X i*) $\in$ *carrier K*
proof (*rule ballI*)
fix *i* **assume** *i*: $i \in \{..<dimension\}$
show *lin-comb* (*iso-K-n-V y*) (*indexing-X i*) $\in$ *carrier K*
using *iso-K-n-V-Pi* **using** *y*
using *lin-comb-is-coefficients-function* [*of iso-K-n-V y*]
unfolding *coefficients-function-def*
using *indexing-X-n-in-carrier-V i* **by** *force*
qed
qed
also have ... = (λi. *if* $i \in \{..<dimension\}$ *then fst y i else* **0**, *dimension* − *1*)
proof (*rule*, *rule conjI*)
show *dimension* − *1* = *dimension* − *1* **by** (*rule refl*)
show (λi. *if* $i \in \{..<dimension\}$ *then lin-comb* (*iso-K-n-V y*) (*indexing-X i*) *else* **0**) =
(λi. *if* $i \in \{..<dimension\}$ *then fst y i else* **0**)
proof
fix *i*
show (*if* $i \in \{..<dimension\}$ *then lin-comb* (*iso-K-n-V y*) (*indexing-X i*) *else* **0**) =
(*if* $i \in \{..<dimension\}$ *then fst y i else* **0**)
proof (*cases* $i \in \{..<dimension\}$)
case *False* **show** *?thesis* **using** *False* **by** *simp*
next
case *True*
have *lin-comb* (*iso-K-n-V y*) (*indexing-X i*) = *fst y i*
unfolding *iso-K-n-V-def*
unfolding *lin-comb-linear-combination-candidate* [*OF y*]
using *preim2-comp-iso-nat-X-id* [*OF True*]

```
        unfolding iso-nat-X-def by simp
       thus ?thesis by simp
      qed
     qed
    qed
    also have ... = y
     unfolding x-in-carrier [symmetric, OF y] by (rule refl)
    finally show ?thesis by fast
  qed
qed

end

context linear-map
begin

definition vector-space-isomorphism :: ('c => 'e) => bool
  where vector-space-isomorphism f == bij-betw f (carrier V) (carrier W) ∧
linear-map f

end

context finite-dimensional-vector-space
begin
```

Finally, the two following lemmas state the isomorphism (in both directions actually) between *field.K-n K dimension* and V:

```
lemma V-K-n.vector-space-isomorphism iso-V-K-n
  using iso-V-K-n-bij using linear-map-iso-V-K-n
  unfolding V-K-n.vector-space-isomorphism-def by rule

lemma vector-space-isomorphism iso-K-n-V
  using iso-K-n-V-bij using linear-map-iso-K-n-V
  unfolding vector-space-isomorphism-def by rule

end

end
theory Subspaces
  imports Isomorphism
begin
```

12 Subspaces

```
context vector-space
begin

definition subspace :: 'b set => bool
  where subspace M == ((M ⊆ carrier V) ∧ M ≠ {}
```

$\wedge$ ($\forall\, \alpha \in carrier\ K.\ \forall\, \beta \in carrier\ K.\ \forall\, x \in M.\ \forall\, y \in M.$
$\alpha \cdot x \oplus_V \beta \cdot y \in M$))

lemma
zero-in-subspace:
assumes *s*: *subspace M*
shows $\mathbf{0}_V \in M$
proof −
obtain *x* **where** *x*: $x \in M$ **using** *s*
unfolding *subspace-def* **by** *fast*
hence *xV*: $x \in carrier\ V$
using *s* **unfolding** *subspace-def* **by** *fast*
have *one*: $\mathbf{1}_K \in carrier\ K$
and *minus-one*: $\ominus\ \mathbf{1}_K \in carrier\ K$ **by** *simp+*
hence $\mathbf{1}_K \cdot x \oplus_V (\ominus\ \mathbf{1}_K \cdot x) \in M$
using *s x* **unfolding** *subspace-def* **by** *blast*
thus *?thesis*
unfolding *mult-1* [*OF xV*] *negate-eq* [*OF xV*]
unfolding *V.r-neg* [*OF xV*] **.**
qed

In the following statement we can observe the operation of field updating for records:

lemma
subspace-is-vector-space:
assumes *s*: *subspace M*
shows *vector-space K* (*V*(|*carrier*:= *M*|)) (*op* ·)
proof (*unfold-locales*, *auto*)
show $\mathbf{0}_V \in M$
by (*metis assms zero-in-subspace*)
fix *x* **and** *y* **and** *z*
assume *x-in-M*: $x \in M$
and *y-in-M*: $y \in M$
and *z-in-M*: $z \in M$
hence *x-in-V*: $x \in carrier\ V$
and *y-in-V*: $y \in carrier\ V$
and *z-in-V*: $z \in carrier\ V$
by (*metis assms mem-def subsetD subspace-def*)+
show $x \oplus_V y \in M$
proof −
have $x \oplus_V y = \mathbf{1} \cdot x \oplus_V \mathbf{1} \cdot y$
by (*metis assms insert-absorb insert-subset*
mult-1 subspace-def x-in-M y-in-M)
also have ... $\in M$
using *s one-closed x-in-M y-in-M*
unfolding *subspace-def* **by** *fast*
finally show $x \oplus_V y \in M$ **.**
qed
show $\mathbf{0}_V \oplus_V x = x$

by (*metis* V.*add.l-one assms mem-def*
subsetD subspace-def x-in-M)
show $x \oplus_V \mathbf{0}_V = x$
by (*metis* V.*add.r-one assms mem-def*
subsetD subspace-def x-in-M)
show $x \oplus_V y = y \oplus_V x$
by (*metis* V.*a-comm x-in-V y-in-V*)
show $x \oplus_V y \oplus_V z = x \oplus_V (y \oplus_V z)$
using *a-assoc*[*OF x-in-V y-in-V z-in-V*] **.**
show $\mathbf{1} \cdot x = x$ **using** *mult-1*[*OF x-in-V*] **.**
show $x \in$ *Units* (|*carrier* = M, *mult* = *op* $\oplus_V$, *one* = $\mathbf{0}_V$|)
proof −
have $\exists\, y \in M.\ x \oplus_V y = \mathbf{0}_V \wedge y \oplus_V x = \mathbf{0}_V$
proof (*rule bexI*[*of* - $\ominus_V x$], *rule conjI*)
show $x \oplus_V \ominus_V x = \mathbf{0}_V$ **using** *r-neg*[*OF x-in-V*] **.**
show $\ominus_V x \oplus_V x = \mathbf{0}_V$
by (*metis* V.*add.l-inv x-in-V*)
show $\ominus_V x \in M$
proof −
have $\ominus_V x = \mathbf{0}_K \cdot x \oplus_V (\ominus_K \mathbf{1}) \cdot x$
by (*metis* K.*a-inv-closed* K.*add.l-one*
K.*add.one-closed add-mult-distrib2*
negate-eq one-closed x-in-V)
also have ... $\in M$
by (*metis* K.*add.one-closed*
abelian-group.a-inv-closed assms is-abelian-group
one-closed subspace-def x-in-M)
finally show *?thesis* **.**
qed
qed
thus *?thesis* **using** *x-in-M* **unfolding** *Units-def* **by** *force*
qed
fix a **and** b
assume *a-in-K*: $a \in$ *carrier* K **and** *b-in-K*: $b \in$ *carrier* K
show $(a \otimes b) \cdot x = a \cdot b \cdot x$
using *mult-assoc*[*OF x-in-V a-in-K b-in-K*] **.**
show $a \cdot x \in M$
proof −
have $a \cdot x = a \cdot x \oplus_V a \cdot \mathbf{0}_V$
by (*metis* V.*add.one-closed* V.*add.r-one*
a-in-K add-mult-distrib1 x-in-V)
also have ... $\in M$
by (*metis* ‹$\mathbf{0}_V \in M$› *a-in-K assms subspace-def x-in-M*)
finally show *?thesis* **.**
qed
show $a \cdot (x \oplus_V y) = a \cdot x \oplus_V a \cdot y$
using *add-mult-distrib1*[*OF x-in-V y-in-V a-in-K*] **.**
show $(a \oplus b) \cdot x = a \cdot x \oplus_V b \cdot x$
using *add-mult-distrib2*[*OF x-in-V a-in-K b-in-K*] **.**

qed

lemma
  subspace-zero:
  shows *subspace* $\{\mathbf{0}_V\}$
  unfolding *subspace-def*
  by (*simp*, *metis mult-zero-descomposition*
    scalar-mult-zeroV-is-zeroV)

lemma *subspace-V*:
  shows *subspace* (*carrier V*)
  unfolding *subspace-def*
  by (*simp*, *metis V.a-closed V.add.one-closed*
    ex-in-conv mult-closed)

As one would expect, a subspace is closed under addition:

lemma *subspace-add-closed*:
  assumes *s*: *subspace S*
  and *x*: $x \in S$ **and** *y*: $y \in S$
  shows $x \oplus_V y \in S$
proof −
  have *xv*: $x \in$ *carrier V* **and** *yv*: $y \in$ *carrier V*
    using *x y s* **unfolding** *subspace-def* **by** *auto*
  have $x \oplus_V y = \mathbf{1} \cdot x \oplus_V \mathbf{1} \cdot y$
    using *mult-1* [*OF xv*] *mult-1* [*OF yv*] **by** *simp*
  thus *?thesis*
    using *s* **unfolding** *subspace-def* **by** (*metis one-closed x y*)
qed

The definition of *finsum* (see *finsum ?G* = *finprod* (| *carrier* = *carrier ?G*, *mult* = *op* $\oplus_{?G}$, *one* = $\mathbf{0}_{?G}$|)) is done in such a way hat for any infinite set it returns *undefined* and otherwise the result of a folding operator over the finite set. Under these circumstances it seems rather hard to prove properties of subspaces considering infinite sums:

lemma *subspace-finsum-closed*:
  assumes *s*: *subspace S*
  and *f*: *finite S*
  and *y*: $Y \subseteq S$
  and *c*: $f \in Y \rightarrow$ *carrier K*
  shows *finsum V* $(\lambda i.\ f\ i \cdot i)\ Y \in S$
proof −
  have *fY*: *finite Y* **by** (*rule finite-subset* [*OF y f*])
  show *?thesis*
    using *fY y c* **proof** (*induct Y*)
    case *empty*
    show *?case*
      using *zero-in-subspace* [*OF s*] **by** *simp*
  next

— Nice Isabelle feature: we can even interpret the locale vector space with the same vector space where only the carrier set has been modified. I thought that this may not be possible because it could produce some problems, but it worked smoothly:

```
    interpret S: vector-space K V(|carrier := S|) op ·
      using subspace-is-vector-space [OF s] .
    case (insert x F)
    have finsum-S: (⨁Vi∈F. f i · i) ∈ S
      using insert.hyps (3) insert.prems by fast
    have fxS: f x · x ∈ S
      using insert.prems
      using s using S.mult-closed by auto
      have lambda: (λi. f i · i) ∈ F → carrier V
        and fx: f x · x ∈ carrier V
        using insert.prems
      using insert.hyps
      using s unfolding subspace-def using mult-closed by blast+
    show ?case
      unfolding finsum-insert [OF insert.hyps (1,2) lambda, OF fx]
      by (rule subspace-add-closed [OF s fxS finsum-S])
  qed
qed

lemma subspace-finsum-closed':
  assumes s: subspace S
  and f: finite Y
  and y: Y ⊆ S
  and c: f ∈ Y → carrier K
  shows finsum V (λi. f i · i) Y ∈ S
using f y c
proof (induct Y)
  case empty
  show ?case
    using zero-in-subspace [OF s] by simp
next
  interpret S: vector-space K V(|carrier := S|) op ·
    using subspace-is-vector-space [OF s] .
  case (insert x F)
  have finsum-S: (⨁Vi∈F. f i · i) ∈ S
    using insert.hyps (3) insert.prems by fast
  have fxS: f x · x ∈ S
    using insert.prems
    using s using S.mult-closed by auto
  have lambda: (λi. f i · i) ∈ F → carrier V
    and fx: f x · x ∈ carrier V
    using insert.prems
    using insert.hyps
    using s unfolding subspace-def using mult-closed by blast+
  show ?case
```

unfolding *finsum-insert* [*OF insert.hyps* (*1*,*2*) *lambda*, *OF fx*]
by (*rule subspace-add-closed* [*OF s fxS finsum-S*])
qed

corollary *subspace-linear-combination-closed*:
assumes *s*: *subspace S*
and *f*: *finite Y*
and *y*: $Y \subseteq S$
and *c*: $f \in$ *coefficients-function Y*
shows *linear-combination f Y* $\in S$
proof (*unfold linear-combination-def*,
rule subspace-finsum-closed')
show *subspace S* **using** *s* **.**
show *finite Y* **using** *f* **.**
show $Y \subseteq S$ **using** *y* **.**
show $f \in Y \rightarrow$ *carrier K*
using *c* **unfolding** *coefficients-function-def* **by** *blast*
qed

end

end

theory *Calculus-of-Subspaces*
imports *Subspaces Ideal*
begin

13 Calculus of Subspaces

The theory Ideal is imported in order to use the definition of the sum of two sets, given by the operation *set-add'*

context *vector-space*
begin

lemma
subspace-inter-closed:
assumes *s*: *subspace M*
and *sm*: *subspace M'*
shows *subspace* $(M \cap M')$
proof (*unfold subspace-def*, *rule conjI3*)
show $M \cap M' \subseteq$ *carrier V* **using** *s sm* **unfolding** *subspace-def* **by** *blast*
show $M \cap M' \neq \{\}$ **using** *zero-in-subspace s sm* **by** *blast*
show $\forall \alpha \in carrier\ K.\ \forall \beta \in carrier\ K.\ \forall x \in M \cap M'.\ \forall y \in M \cap M'.\ \alpha \cdot x \oplus_V \beta \cdot y \in M \cap M'$
using *s sm* **unfolding** *subspace-def* **by** *blast*

qed

13.1 Theorem 1

In the following result we have to avoid empty intersections, since the empty intersection is defined to be equal to *UNIV*. *UNIV* is not a subspace, since it is not (in general, it could be in some cases) a subset of *carrier V*.

Nevertheless, this does not mean any limitation in practice, since any set will be always a subset of the subspace *carrier V* (see *subspace* (*carrier V*))

We need to prove that intersection of subspaces is a subspace to define later the subspace spanned by any set as the intersection of every subspace in which the set is contained. Thus, assuming that the intersection will be not empty (*carrier V* will be always a member of such intersection) is natural.

```
lemma subspace-finite-inter-closed:
  assumes a: finite A
  and ne: A ≠ {}
  and kj: ∀ j∈A. subspace (P j)
  shows subspace (⋂ j∈A. P j)
  using a kj ne proof (induct A)
  case empty
  show ?case using empty.prems by simp
next
  case (insert x F)
  have Px: subspace (P x) using insert.prems (1) by blast
  show ?case
  proof (cases F = {})
    case True
    show ?thesis
      unfolding True using Px by fastsimp
  next
    case False
    have sF: subspace (⋂ a∈F. P a)
      using insert.hyps (3) using False using insert.prems (1) by blast
    show ?thesis
      unfolding INT-insert
      by (rule subspace-inter-closed [OF Px sF])
  qed
qed
```

The same lemma than ⟦*finite ?A*; *?A* ≠ {}; ∀ *j*∈*?A*. *subspace* (*?P j*)⟧ ⟹ *subspace* ($\bigcap_{j\in ?A}$ *?P j*) but for collections indexed by the natural numbers:

```
lemma subspace-finite-inter-index-closed:
  assumes smn: ∀ j∈{..(n::nat)}. subspace (M j)
  shows subspace (⋂ j∈{..n}. M j)
  using smn proof (induct n)
  case 0
```

show *?case* **using** *0* **by** *simp*
next
case (*Suc n*)
have *prem*: $\forall j\in\{..n\}$. *subspace* (*M j*) **and** *prem2*: *subspace* (*M* (*Suc n*))
using *Suc.prems* **by** *simp-all*
hence *prem1*: *subspace* ($\bigcap$ $a\leq n$. *M a*)
using *Suc.hyps* **by** *fast*
show *?case*
unfolding *atMost-Suc*
unfolding *INT-insert* [*of Suc n* {..*n*}]
by (*rule subspace-inter-closed*, *rule prem2*, *rule prem1*)
qed

We now remove the requisite of the collection of subspaces being finite. Thus, the proof cannot be longer carried out by induction in the structure of the set.

lemma *subspace-infinite-inter-closed*:
assumes *ne*: $A \neq \{\}$
and *kj*: $\forall j\in A$. *subspace* (*P j*)
shows *subspace* ($\bigcap j\in A$. *P j*)
proof (*unfold subspace-def*, *rule*)
show *INTER A P* $\subseteq$ *carrier V*
unfolding *INTER-def*
using *ne* **using** *kj* **unfolding** *subspace-def* **by** *blast*
show *INTER A P* $\neq \{\} \wedge$
($\forall \alpha\in$*carrier K*. $\forall \beta\in$*carrier K*. $\forall x\in$*INTER A P*. $\forall y\in$*INTER A P*. $\alpha \cdot x \oplus_V \beta \cdot y \in$ *INTER A P*)
proof (*rule conjI*)
show *INTER A P* $\neq \{\}$
proof (*unfold INTER-def*, *auto*, *rule exI* [*of* - $\mathbf{0}_V$], *rule*)
fix *x* **assume** *x*: $x \in A$
show $\mathbf{0}_V \in P\ x$
using *zero-in-subspace* [*of P x*]
using *kj* **using** *x* **by** *fast*
qed
show $\forall \alpha\in$*carrier K*. $\forall \beta\in$*carrier K*. $\forall x\in$*INTER A P*. $\forall y\in$*INTER A P*. $\alpha \cdot x \oplus_V \beta \cdot y \in$ *INTER A P*
proof (*rule ballI*)+
fix *x y a b*
assume *x*: $x \in$ *INTER A P* **and** *y*: $y \in$ *INTER A P*
and *a*: $a \in$ *carrier K* **and** *b*: $b \in$ *carrier K*
show $a \cdot y \oplus_V b \cdot x \in$ *INTER A P*
proof
fix *xa* **assume** *xa*: $xa \in A$
have *xp*: $x \in P\ xa$ **and** *yp*: $y \in P\ xa$ **using** *x y xa* **by** *auto*
thus $a \cdot y \oplus_V b \cdot x \in P\ xa$
using *kj xa a b* **unfolding** *subspace-def* **by** *force*
qed
qed

qed
qed

It is now clear than the previous results for finite intersections $\forall j\in\{..?n\}.$ *subspace* $(?M\ j) \Longrightarrow$ *subspace* $(\bigcap j \leq\ ?n\ ?M\ j)$ and $[\![$*finite* $?A$; $?A \neq \{\}$; $\forall j\in ?A.$ *subspace* $(?P\ j)]\!] \Longrightarrow$ *subspace* $(\bigcap_{j\in ?A}\ ?P\ j)$ can be proved as a corollary of $[\![?A \neq \{\}; \forall j\in ?A.$ *subspace* $(?P\ j)]\!] \Longrightarrow$ *subspace* $(\bigcap_{j\in ?A}\ ?P\ j)$, but we prefer to leave their induction proofs since they illustrate different ways of proving similar results depending on the context or the premises.

Here Halmos introduces the definition of the span of a set $S \subseteq$ *carrier* V as the interection of all the subsets in which S is contained. We already have a notion of the *span* of a set in our setting, as the set of all the elements which are equal to the linear combinations of the elements of this set. We will name this new notion *subspace-span*, and then prove that they both are equal:

We introduce an auxiliar definition of the set of subspaces in which one set is enclosed:

definition *subspace-encloser* :: ('b => bool) => ('b => bool) set
  where *subspace-encloser* $A = \{M.$ *subspace* $M \wedge A \subseteq M\}$

A trivial lemma stating that a set is always enclosed in the subspace *carrier* V:

lemma
  assumes m: $M \subseteq$ *carrier* V
  shows *carrier* $V \in$ *subspace-encloser* M
  unfolding *subspace-encloser-def*
  using *subspace-V m* **by** *fast*

The definition of the subspace spanned by a set, following Halmos:

definition *subspace-span* :: $('b \Rightarrow bool) \Rightarrow 'b \Rightarrow bool$
  where *subspace-span* $A = (\bigcap B\in($*subspace-encloser* $A).\ B)$

The previous lemma $[\![$*finite* $?A$; $?A \neq \{\}$; $\forall j\in ?A.$ *subspace* $(?P\ j)]\!] \Longrightarrow$ *subspace* $(\bigcap_{j\in ?A}\ ?P\ j)$ is now used to prove that *subspace-span* is a subspace itself.

lemma
  subspace-span-monotone:
  assumes s: $S \subseteq$ *carrier* V
  shows $S \subseteq$ *subspace-span* S
  unfolding *subspace-span-def*
  unfolding *subspace-encloser-def* **by** *fast*

lemma
  subspace-subspace-span:
  assumes s: $S \subseteq$ *carrier* V

shows *subspace* (*subspace-span S*)
unfolding *subspace-span-def subspace-encloser-def*
proof (*rule subspace-infinite-inter-closed*)
show $\{M.\ subspace\ M \wedge S \subseteq M\} \neq \{\}$
using *subspace-V s* **by** *blast*
show *Ball* $\{M.\ subspace\ M \wedge S \subseteq M\}$ *subspace* **by** *fast*
qed

13.2 Theorem 2.

The definition of *finsum* in Isabelle relies on the notion of finiteness of the set which elements are added up. Working in a finite dimensional vector space does not mean that every subset is finite, and thus the elements in the span of such a set cannot be written as finite sums of its elements.

The previous point is not explicit is Halmos, where it is never explained how to deal with infinite sums (or sums over not finite sets).

lemma
subspace-span-empty:
subspace-span $\{\} = \{\mathbf{0}_V\}$
proof
show $\{\mathbf{0}_V\} \subseteq$ *subspace-span* $\{\}$
unfolding *subspace-span-def subspace-encloser-def*
using *zero-in-subspace* **by** *blast*
show *subspace-span* $\{\} \subseteq \{\mathbf{0}_V\}$
unfolding *subspace-span-def subspace-encloser-def*
using *subspace-zero* **by** *force*
qed

lemma *theorem-2*:
assumes *f*: *finite S*
and *s*: $S \subseteq$ *carrier V*
shows *span S* = *subspace-span S*
proof
show *span S* $\subseteq$ *subspace-span S*
using *f s* **proof** (*induct S*)
case *empty*
show *?case* **unfolding** *span-empty subspace-span-empty* **..**
next
case (*insert x F*)
show *?case* **unfolding** *subspace-span-def subspace-encloser-def* **unfolding** *span-def* **apply** *auto*
proof −
fix *xa* **and** *g*
assume *cf-g*: $g \in$ *coefficients-function* (*carrier V*)
and *s-xa*: *subspace xa*
and *x-in-xa*: $x \in xa$ **and** *F-subset-xa*: $F \subseteq xa$
have *gs-insert*: *good-set* (*insert x F*)
by (*metis finite.insertI good-set-def insert*(*1*) *insert.prems*)

```
      show linear-combination g (insert x F) ∈ xa
      proof (rule subspace-linear-combination-closed)
        show subspace xa using s-xa .
        show finite (insert x F) using insert.hyps(1) by fast
        show insert x F ⊆ xa using x-in-xa F-subset-xa by fast
        show g ∈ coefficients-function (insert x F) sorry
      qed
    qed
  qed
  show subspace-span S ⊆ span S
    unfolding subspace-span-def subspace-encloser-def unfolding span-def apply
auto
    sorry
qed
```

13.3 Theorem 3.

The following theorem appears in Halmos as an easy consequence of the previous one; probably it should be proved based on the fact that any linear combination can be written down as the sum of two elements, being one in the first set and the other in the second one.

```
term I <+>_R J
find-theorems - <+>_?F -
lemma theorem-3:
  assumes I: subspace I
  and J: subspace J
  shows subspace-span (I ∪ J) = I <+>_V J
  unfolding AbelCoset.set-add-def'
proof
  show subspace-span (I ∪ J) ⊆ (⋃h∈I. ⋃k∈J. {h ⊕_V k})
    sorry
  show (⋃h∈I. ⋃k∈J. {h ⊕_V k}) ⊆ subspace-span (I ∪ J)
    sorry
qed
```

The following definition is simply a rewriting rule, it may be skipped; note also that produces ambiguous parse trees when parsing deducing types from expressions, so it could be avoided if it produces any clashes:

```
definition set-add2 :: 'b set => 'b set => 'b set  (infixl + 60)
  where set-add2 A B = subspace-span (A ∪ B)

corollary set-add2-set-add':
  assumes I: subspace I
  and J: subspace J
  shows I + J = I <+>_V J
  unfolding set-add2-def using theorem-3 [OF I J] .
```

The following definition is applied only to subspaces:

definition *complement* :: *'b set => 'b set => bool*
where *complement I J* = $((I \cap J = \{\mathbf{0}_V\}) \wedge (I + J = carrier\ V))$

end

end
theory *Dimension-of-a-Subspace*
imports *Calculus-of-Subspaces*
begin

14 Dimension of a Subspace

context *finite-dimensional-vector-space*
begin

14.1 Theorem 1.

The theorem states that the subspace is itself a vector space and that its dimesion is less than or equal to the one of V. We split both conclusions in two different lemmas that later will be merged.

The first part of the theorem has been already proved:

lemma *theorem-1-part-1*:
assumes *m*: *subspace M*
shows *vector-space K* $(V(\!|carrier{:=}M|\!))$ $(op\ \cdot)$
using *subspace-is-vector-space* [*OF m*] **.**

The second part of the theorem requires a definition of dimension. The dimension of a (finite) vector space should be defined as the cardinal of any of its basis, once we have proved that every basis has the same cardinal (file *Finite-Vector-Space.thy*). In the meanwhile, I use *dim*

Its proof should be direct by reduction ad absurdum, following the one in Halmos.

lemma *theorem-1-part-2*:
assumes *m*: *subspace M*
shows *dim* $(V(\!|carrier{:=}M|\!)) \leq$ *dimension*
sorry

14.2 Theorem 2.

The notation in the following statement might be a bit confusing. The indexing f is just necessary to later select the first m elements of a base, with m being the dimension of the subspace M. These m elements can be completed up to a basis of V.

The proof should be done using that M is a vector space of dimension less or equal to the one of V. Therefore we can find a basis of it which cardinal is less than or equal to *dimension*. This basis is a collection of linearly independent vectors, and therefore can be completed up to a basis of V, thanks to one of the lemmas proved in *Finite-Vector-Space.thy*.

lemma *theorem-2*:
 assumes *m*: *subspace M*
 shows $(\exists B\, f.\ (basis\ B) \wedge indexing\ (B, f) \wedge$
$(vector\text{-}space.basis\ K\ (V(\!|carrier{:=}M|\!))\ (op\ \cdot)\ (f\ \text{`}\ \{..dim\ (V(\!|carrier{:=}M|\!))\})))$
proof −
 interpret *M*: $vector\text{-}space\ K\ (V(\!|carrier{:=}M|\!))\ (op\ \cdot)$
  using *subspace-is-vector-space* [*OF m*] **.**
 show *?thesis*
  sorry
qed

end

end
theory *Dual-Spaces*
 imports *Dimension-of-a-Subspace*
begin

15 Dual Spaces

context *vector-space*
begin

This definition can be found also on Bauer's development, taking as the scalar field the set of real numbers, and with the name of linear form. We follow linear functional as Halmos' text

We split the definition of linear form into its multiplicative and additive components:

definition *additive-functional* :: $('b => 'a) => bool$
 where *additive-functional f*
 $\equiv (\forall x{\in}carrier\ V.\ \forall y{\in}carrier\ V.\ f\ (x \oplus_V y) = f\ x \oplus_K f\ y)$

definition *multiplicative-functional* :: $('b => 'a) => bool$
 where *multiplicative-functional f*
 $\equiv (\forall k{\in}carrier\ K.\ \forall x{\in}carrier\ V.\ f\ (k \cdot x) = k \otimes_K (f\ x))$

definition *linear-functional* :: $('b => 'a) => bool$
 where *linear-functional f* $\equiv$ *additive-functional f*
 $\wedge$ *multiplicative-functional f*

The following lemma appears in Halmos (as the homogeneous property) and also in Bauer's files; in Bauer there are also some properties about the difference anf lineal functionals.

lemma *linear-functional-zero*:
  assumes *linear-functional f*
  shows $f\ \mathbf{0}_V = \mathbf{0}$
  sorry

We introduce the definition of the dual space of the vector space V. We have to provide a carrier set, a zero operation and an addition. As the definition of abelian groups in Isabelle is done omver the ring type, we also have to provide some definition of unit and multiplication, that will be useless.

The dual space is also denoted in Halmos V'

definition *dual-space* :: $('b => 'a)$ *ring* (V')
  where $dual\text{-}space = (\!|\ carrier = linear\text{-}functional,$
    $mult = undefined,$
    $one = undefined,$
    $zero = (\lambda x.\ \mathbf{0}),$
    $add = (\lambda y1.\lambda y2.\lambda x.\ y1\ x \oplus y2\ x)|\!)$

We create a synonim for the previous definition to ease readability:

lemmas V'*-def* = *dual-space-def*

term *vector-space* $K\ V'$

term $(\lambda x\ f\ y.\ x \otimes f\ y)$

I guess it is not necessary to go down to finite dimensional vector spaces to prove the following lemma. If it is necessary, the context should be changed accordingly:

lemma *vector-space-*V': *vector-space* $K\ V'\ (\lambda x\ f\ y.\ x \otimes f\ y)$

  sorry

end

end
theory *Brackets*
  imports *Dual-Spaces*
begin

16 Brackets

context *vector-space*
begin

The following notation is not working properly: 1. I do not know how to invert the order of the parameters, in such a a way that $<x,f>$ denotes $f\ x$; 2. Even in the right order, where $<f,x>$ denotes $f\ x$, the notation $<f,x>$ produces problems when trying to use it.

A couple of notes on the following notation; it is done trying to mimic the similar ideas in Halmos. First of all, we have chosen the symbols < - > instead of [-] since brackets would produce ambiguous inputs with lists, forcing us to write explicitly in a lot of scenarios the type of each of the components of the pair.

Second, the *input* annotation of *abbreviation* makes the special syntax proposed to work only in the input mode, *i.e.*, when we write something. Wihtout this annotation, the output would be also changed, but that would affect to every function application in our setting, which is not our intention and apparently makes the pretty printer loop. For more details see https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2011-August/msg00007.html

abbreviation (*input*)
  app :: *'b* => (*'b* => *'a*) => *'a* (<(-),(-)> *90*)
  where <*x*, *f*> == *f x*

term <*x*, *f*> ⊕ <*y*, *f*>

end

end
theory *Dual-Bases*
  imports *Brackets*
begin

17 Dual Bases

context *finite-dimensional-vector-space*
begin

17.1 Theorem 1.

We recall here that X is a basis for the vector space V and *indexing-X* is a way to provide the basis with coordinates.

The definition of *indexing* is polymorphic, and in this lemma will be used both for the basis X and also for the set of scalars.

In this lemma will be useful the results in file *Vector-Space-K-n.thy*, for instance $?x \in \mathit{carrier}\ V \Longrightarrow \exists!f.\ f \in \mathit{coefficients\text{-}function}\ X \wedge \mathit{linear\text{-}combination}\ f\ X = ?x$ and $?x \in \mathit{carrier}\ V \Longrightarrow ?x = (\bigoplus_V i\in\{..<\mathit{dimension}\}.\ \mathit{lin\text{-}comb}\ ?x\ (\mathit{indexing\text{-}X}\ i) \cdot \mathit{indexing\text{-}X}\ i)$, where it is proved that any element in *carrier V* can be expressed in a unique way as a linear combination of the elements in X.

thm *lin-comb-is-the-linear-combination-indexing*
find-theorems (∃!*f*. -)

```
lemma theorem-1:
  assumes ia: indexing ((A::'a set), fA)
  and c: card A = dimension
  shows (∃!y. linear-functional y ∧ (∀ i∈{..<dimension}. <indexing-X i, y> = fA
i))
proof −
  def y == (λx. (⨁ K i∈{..<dimension}. (lin-comb x) (indexing-X i) ⊗ (fA i)))
  show ?thesis
  proof (rule ex1I [of - y], rule conjI)
    show linear-functional y
      unfolding y-def
      unfolding linear-functional-def additive-functional-def
        multiplicative-functional-def
      sorry

    show ∀ i∈{..<dimension}. y (indexing-X i) = fA i
    proof (rule ballI)
      fix i assume i: i ∈ {..<dimension}
      show y (indexing-X i) = fA i
        unfolding y-def
        using lin-comb-basis
        sorry
    qed
  next
     show ⋀ya. linear-functional ya ∧ (∀ i∈{..<dimension}. ya (indexing-X i) =
fA i) ⟹ ya = y
      sorry
  qed
qed
```

17.2 Theorem 2.

```
term linear-functional

definition delta :: nat => nat => 'a
  where delta i j = (if i = j then 1 else 0)

definition linear-functional-basis :: nat => ('c => 'a)
  where linear-functional-basis n = (λx. delta (preim2 x) n)

definition linear-functional-basis-set :: ('c => 'a) set
 where linear-functional-basis-set = {(λx. delta (preim2 x) n) | n. n ∈ {..<dimension}}

lemma theorem-2:
  shows vector-space.basis K V' (λx f y. x ⊗ f y) linear-functional-basis-set
proof −
  interpret V': vector-space K V' (λx f y. x ⊗ f y) using vector-space-V' .
  show ?thesis
```

sorry
qed

17.3 Theorem 3.

lemma *theorem-3*:
assumes *x-ne-0*: $x \neq \mathbf{0}_V$
shows $\exists\, y.$ *linear-functional* $y \wedge$ <x,y> $\neq \mathbf{0}_K$
sorry

corollary *theorem-3-c*:
assumes *x-ne-0*: $u \neq v$
shows $\exists\, y.$ *linear-functional* $y \wedge$ <u,y> $\neq$ <v,y>
sorry

end

end