UNIVERSIDAD DE LA RIOJA

FACULTAD DE CIENCIAS, ESTUDIOS AGROALIMENTARIOS E INFORMÁTICA

TITULACIÓN:

Ingeniería Técnica en Informática de Gestión

TÍTULO DEL PROYECTO O TRABAJO FIN DE CARRERA:

Demostración de propiedades de espacios vectoriales finito-dimensionales en Isabelle/HOL

DIRECTOR/ES DEL PROYECTO O TRABAJO:

D. Jesús María Aransay Azofra

DEPARTAMENTO: Matemáticas y Computación

ALUMNO/S: Jose Divasón Mallagaray

CURSO ACADÉMICO: 2011 / 2012

Degree's Thesis
Proofs of properties of finite-dimensional vector spaces using Isabelle/HOL

Jose Divasón Mallagaray
Supervised by Jesús María Aransay Azofra

Universidad de La Rioja
2011/2012

To my friend Claudia.
Now that you have started the degree in mathematics, I hope you have the same luck than I have had. You deserve the best, keep it up!

Abstract

In this work we deal with finite-dimensional vector spaces over a generic field $\mathbb{K}$. First we will state properties of vector spaces independently of their dimension. Then, we will introduce the conditions to obtain finite-dimensional vector spaces. The notions of linear dependence and independence, as well as linear combinations, and hence the notion of basis will be presented. Some results about the dimension of the different basis of a vector space will be necessary, as well as on the isomorphism among vector spaces. Once we have introduced the notion of basis, and with the additional condition of it being finite, we will introduce the notion of finite-dimensional vector space. Next step is to introduce vector subspaces. We will pay attention to vector susbpaces generated by a given set of vectors and prove some of their properties. The notion of linear maps will be also required to define isomorphisms of vector spaces. Finally, we will prove that a vector space (over a field $\mathbb{K}$) of (finite) dimension n is isomorphic to $\mathbb{K}^n$.

The previous results will be presented following the book by Halmos on vector spaces [1]. Its formalization will be carried out in Isabelle/HOL [25].

Acknowledgements

I must say that this work, due to its difficulty and complexity, has been developed in close colaboration with the advisor of the project, Jesús Aransay. From here I want to say that I am very grateful to Jesús for his patience, specially in my first moments with Isabelle/HOL. It took me a lot of hard work to overcome the initial difficulties and at that moments he helped with my learning of the system, much more than I expected at first. Whenever I was stuck, I visited the office and he looked for a while to solve my doubts. I am sure that with other person as advisor, the development of this project wouldn't have been possible or at least we wouldn't have achieved the formalization of the results that we have obtained. Thanks Jesús.

There are another two persons who deserve to be named in this gratitudes: Óscar Ciaurri and Julio Rubio. It can be said that Óscar is the first person responsible for this project to get started, since he began a chain: Óscar knew that I like this topics about logic and mathematical reasoning, and he got in touch with Julio to comment it (without my knowledge). After that, Julio went to look for me at the end of an exam to offer me the possibility of this degree's dissertation with Jesús. I'm grateful to them. Both are the persons responsible for giving me the opportunity of this project and I have tried to make the most of it.

Contents

Chapter 1

Introduction

First of all we have to say that this is not a common degree's dissertation of "Ingeniería Técnica en Informática de Gestión" in which an application or a database is developed following a methodology with an analysis, design, implementation and verification processes. We are going to develop in this project something more focused on the mathematical field: we are going to formalize in Isabelle/HOL results of linear algebra, specifically, results about finite dimensional vector spaces. This project is in some sense a research project, in which we will have a learning period of the Isabelle/HOL language and after that we will try to formalize the proofs, and in which some degree of uncertainty about the feasibility of the tasks to be accomplished exists.

The first question arises is: why did I choose this kind of project? The main reason is due to the own interest in this kind of matters about formal verification of algorithms, mechanized reasoning, formal proofs... Another reason is to go out of standard projects, hence I can make use of the mathematical knowledge that I have attained in the degree.

Still another motivation exists in the study and the improvement of our knowledge on mathematical proofs and its inherent nature. Traditionally, "pencil and paper" proofs are carried out in a loose way and without an absolute rigour in some of their details. There exist mathematical results which proofs have contained mistakes for long periods, even in the light of being publicly available and widely accepted (for instance, there were some erroneus proofs of the Four Colour Theorem which were accepted for long periods, see [22] for details). The detailed implementation of proofs has gained attention since the end of the XIX century (with the foundation of

logical systems, following the influence of Frege, Hilbert...). The mechanization obtained through computers has then give place to the formalization of mathematical proofs, starting from the computer implementation of a logical system and building mathematical knowledge through the computer-assisted application of logical rules. These computer-assisted proofs are consequently full of minor details, but also formally verified, hence somehow more trustable than traditional proofs, and even available for being reproduced in any computer by external agents. The process of creating or "translating" textbook proofs and results to formalized proofs requires a deep study and insight of that proofs which is very often a challenging task.

Another reason is to check the capacities of a given logic (HOL) to implement a mathematical formalization from a practical point of view. If we are capable of implementing a proof in a given logic, this demonstrates that the proof and its reasoning techniques can be carrried out in such a logical setting.

In addition, the "Universidad de La Rioja" is involved in a European project about formalization of mathematics: ForMath [13] (the objective of this project is to develop libraries of formalized mathematics concerning algebra, linear algebra, real number computation and algebraic topology). Thus, our project will share some goals with the ForMath project.

Once we have decided that we will make a project about formal proofs, why we choose Isabelle/HOL? Isabelle is an environment for computer-assisted formal proofs. It has libraries with the development of several theories about algebra, analysis... and its capacities are well-known. Some examples of theorems proved with Isabelle are the Fundamental Theorem of Algebra and the Prime Number Theorem (some other relevant theorems proved in Isabelle can be found here: [14]). In addition, Isabelle is a system which permits declarative and procedural proofs. A declarative system is one in which one writes a proof in the normal way, although in a highly stylized language and with very small steps. In a procedural system one does not write proofs at all: the computer presents the user with proof obligations or goals, and the user then executes tactics which reduce a goal to zero or more new, and hopefully simpler, subgoals. Another proof assistants only allow one of this proof styles (for example, HOL-Light and Coq are procedural systems and Mizar is declarative). Proofs with a procedural system are easier for a beginner (as me) because they are more interactive.

Finally, we already know what we are going to do and what environment

we are going to use. Why do we focused on vector spaces?

Despite they are something quite basic in the field of linear algebra, vector spaces aren't developed in Isabelle/HOL (really there is only a result about real vector spaces: the Hanh-Banach theorem formalized by Gertrud Bauer[12]). Hence another point to do this project is to increase the amount of mathematical results that have been implemented, in order to create a corpus of mathematical results that permits further developments in the field of finite dimensional vector spaces.

In order to formalize the main results on vector space we will follow a Halmos' book: *Finite-dimensional vector spaces*[1]. This book summarizes and presents the main results of finite-dimensional vector spaces and their proofs.

At first, our objective was to manage to formalize the first 16 sections, i.e., up to prove that a vector space is isomorphic to the dual of its dual. The proof of this theorem is presented in Halmos in a simple manner (in other books the proof is omitted or it is not explicitly proved). Previous to this result, Halmos introduces vector spaces, both of finite and infinite nature. Then, properties about the dimension and the nature of basis in such spaces are proved. In particular, the notions of linear dependence and independence are introduced. The vector space $\mathbb{K}^n$ is also defined (with $\mathbb{K}$ the field over which a vector space is defined), and its correspondence to a vector space V proved. Notions of subspaces and direct sums are also presented. Along the memoir, all these notions will be defined and formalized in Isabelle/HOL.

I have to say that my initial knowledge about formal proofs was null, either in Isabelle/HOL or in another theorem proving environment. For that reason, I required a learning process, hard but nice. It is usually said that it is harder to learn to formalize proofs than to learn a programming language, and I can say that in my case it is true.

In this memoir it is impossible to include the whole code implemented due to its length, but we can see it complete in [2]. Here we will present the main results, sometimes omitting parts of the formal proofs or even the whole proofs. In addition, we will omit auxiliary results which are necessary or useful, but not crucial or essential to our goal (or their proofs do not require any interesting kind of reasoning).

The project and the memoir have been written in English by several reasons. The first one is that the Isabelle/HOL libraries are implemented in

English, and as we want to increase them we have to do it in this language. The second reason is that the diffusion that we can achieve writting a project in English is greater than in Spanish. The third reason is that we want this project to be part of the ForMath reports and documentation (and this project main language is English). Writting the memoir in English and in LaTeX poses a challenge, even more since I haven't used English for several years and I have never used LaTeX.

Chapter 2

Management

This degree's thesis started in November of 2010. As we have already said in the Introduction, our first objective is to formalize the first 16 sections in Halmos up to managing to prove that a vector space is isomorphic to the dual of its dual and we had the intention of doing it for June of 2011, i.e., in 8 months of work. Those 16 first sections take up a total of 25 pages in the book. It is very difficult to estimate the possible duration of the project, since as we have said before, there exists an inevitable uncertainty that we can't control.

We have said that the goal is to formalize 16 sections of Halmos [1] in 8 months. Every section has neither the same length nor the same difficulty, so we can't estimate that we are going to formalize 2 sections by month. In addition, it is necessary a learning period (really the learning period is continous during the whole development) before starting to implement and formalize proofs. It would be better to talk about formalized pages by month. Even more, there exists a rule that says that a line of a book should be turned into 4 lines in a formalized proof ("de Bruijn factor" [15]). As the 16 first sections in Halmos take up 25 pages, a good objective is to plan as work the formalization of 3 to 4 pages by month.

Working 2-3 hours per day during those 8 months, we have about 450 hours for the project.

We are going to show a decomposition in tasks of the project (we will divide it in the sections of the book). Every task about proofs can be decomposed in another two: first, to study, think and develop the proof in paper

and after that to implement it in Isabelle/HOL. In addition, we add at the end of each section a task of reviewing the finished proofs to try to make them clearer, shorter, more organized and legible. This task takes up about 20% of the time to develop the code.

0. PREVIOUS LEARNING

Firstly I will have to read some documents and tutorials about Isabelle/HOL. Really, these documents will be good as an introduction to Isabelle, but as in a programming language, the only way of learning to formalize is by doing it. This learning is continuous during the development, but I need this as a first contact with the system.

- Read tutorials and practice with Isabelle/HOL. **Estimated time:** 12 hours.

TOTAL ESTIMATED TIME OF THIS TASK: 12 hours.

1. FIELDS

Here Halmos presents the definition of field. The implementation of it will be not difficult for us, since it is alreay done in the Isabelle/HOL library. This is a good section to practice and learn about the formal proofs in Isabelle. We will prove the exercise 1 which is in page 2 of Halmos. It contains 7 preliminary results about fields.

- Exercise 1 of page 2. **Estimated time:** 14 hours.
- Check the code and make clearer, shorter and more legible the proofs. **Estimated time:** 3 hours.

The difficulty of this chapter is that here we start with the proofs in Isabelle. TOTAL ESTIMATED TIME OF THIS TASK: 17 hours.

2. VECTOR SPACES

In this section we will define the concept of vector space. To define algebraic structures in Isabelle is not hard, and thanks to the possibility of defining inheritances from abelian groups in the definition of vector space, this section doesn't seem hard to be implemented.

- Definition of Vector Space. **Estimated time:** 1 hour.
- Vector Space Introduction. **Estimated time:** 1 hour.

- Check the code and make clearer, shorter and more legible the proofs. **Estimated time:** 0.4 hours

TOTAL ESTIMATED TIME OF THIS TASK: 2.4 hours.

3. EXAMPLES

Here we will present an instance of vector space. We will prove that every field over itself is a vector space. In future sections we will prove another example: $\mathbb{K}^n$ is a vector space (for $\mathbb{K}^n$ a field).

- Every field over itself is a vector space. **Estimated time:** 3 hours.
- Check the code and make clearer, shorter and more legible the proofs. **Estimated time:** 0.6 hours

TOTAL ESTIMATED TIME OF THIS TASK: 3.6 hours.

4. COMMENTS

Our intention is to make the exercise 1 of this section (page 6). It consists in proving 7 properties about vector spaces, the relation between the zero and the scalar product ... Nevertheless, it is a good idea to plan for this section some proofs of another interesting and useful properties for the future which don't appear in Halmos.

- Exercise 1. **Estimated time:** 12 hours.
- Another lemmas and results. **Estimated time:** 18 hours.
- Check the code and make clearer, shorter and more legible the proofs. **Estimated time:** 6 hours

TOTAL ESTIMATED TIME OF THIS TASK: 36 hours.

5. LINEAR DEPENDENCE

With a quick look we realize that this is a hard section. We have to do the definition of the concepts: linear dependence and linear independence. At this time, it is no clear how we can make it. After that we have to check that definitions are mutually exclusive and to prove the properties that a linearly independent/dependent set satisfies. We think that it will be necessary to introduce much more lemmas and results than in Halmos. Halmos also explains briefly the definitions of these concepts in the case of

an infinite set. Depending on the difficulty, we could study the possibility of implementing them (even if our main interest lies in the finite version).

- Define the notion of a linearly independent set and a linearly dependent set. **Estimated time:** 2 hours.
- Prove that the definitions are mutually exclusive. **Estimated time:** 3 hours.
- Prove properties and results of linearly independence and dependence. **Estimated time:** 15 hours.
- POSSIBLE: Implement the notion of linearly independence and depencence in the infinite case.
- Check the code and make clearer, shorter and more legible the proofs. **Estimated time:** 4 hours.

TOTAL ESTIMATED TIME OF THIS TASK: 24 hours.

6. LINEAR COMBINATIONS

Despite this being a short section, we think it will be very difficult. We have to prove theorem 10.2.1 and it doesn't look nice, since Halmos is introducing orders to sets. We don't know at this moment how we will avoid this problem. It is a critical result because it is used several times in future sections, so we have to be careful with the development of this section. Here we also have to prove properties of linear combinations, even some which don't appear in Halmos but that will save work for us in the future. Here we have to present the definition of linear combination which is indeed a finite sum, so we will have to look for how finite sums are implemented in the Isabelle/HOL library in order to make the definition of linear combination.

- Define linear combinations. **Estimated time:** 2 hours.
- Prove some properties of them. **Estimated time:** 7 hours.
- Prove that if x is a linear combination of $\{x_i\}_{i\in\mathbb{N}}$, then if $\{x_i\}_{i\in\mathbb{N}}$ is linearly independent, then a necessary and 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. We can decompose this theorem in two implications. **Estimated time:** 7 hours.

- The linear combination of $\{\}$ is 0_V. **Estimated time:** 0.5 hours.
- Study how to prove the theorem 10.2.1 avoiding the problem of the order. **Estimated time:** 6 hours.
- Prove it. **Estimated time:** 15 hours.
- Check the code and make clearer, shorter and more legible the proofs. **Estimated time:** 7.5 hours.

TOTAL ESTIMATED TIME OF THIS TASK: 45 hours.

7. BASES

At first, this is another very difficult section. The main objective will be to prove theorem 11.2.1, which claims that every linearly independent set can be completed to a basis. The proof of this theorem looks very hard, it makes use of an iterative reasoning that we don't know how we will implement. Nevertheless, we define the notion of basis before that and we will try to prove that the coordinates of a vector in a basis are unique.

- Define the notion of basis. **Estimated time:** 0.25 hours.
- Define a finite dimensional vector space. **Estimated time:** 0.25 hours.
- Prove that the coordinates of a vector in a basis are unique. **Estimated time:** 4 hours.
- Prove the theorem 11.2.1. **Estimated time:** 15 hours.
- Check the code and make clearer, shorter and more legible the proofs. **Estimated time:** 4 hours.

TOTAL ESTIMATED TIME OF THIS TASK: 23.5 hours.

8. DIMENSION

Here Halmos presents another important result, the theorem 12.1.2, which claims that two bases of the same finite dimensional vector space have the same cardinality. It makes use of a new iterative argument for the proof again, so it seems that it will be very difficult and hard to be implemented. Here we will also try to present the proof of theorem 12.2.2 (every set of $n+1$ vectors of an n-dimensional vector space is linearly dependent) which is easier once we have proved 12.1.2.

- Prove theorem 12.1.2. **Estimated time:** 18 hours.
- Define the notion of dimension of a finite dimensional vector space. **Estimated time:** 0.25 hours.
- Prove theorem 12.2.2. **Estimated time:** 3 hours.
- Check the code and make clearer, shorter and more legible the proofs. **Estimated time:** 4 hours.

TOTAL ESTIMATED TIME OF THIS TASK: 25.25 hours.

9. ISOMORPHISM

This seems a very laborious section. We have to define the concept of isomorphism between two vector spaces and prove that every n-dimensional vector space V over a field $\mathbb{K}$ is isomorphic to $\mathbb{K}^n$. We will try to prove it, and this is a great question. We have no idea how we could define $\mathbb{K}^n$ in general, since Isabelle does not permit dependent types. It is worth noting that the notion of linear map will show up here, even if Halmos introduces it much later.

- Define the isomorphism between two vector spaces. **Estimated time:** 0.5 hours.
- Define $\mathbb{K}^n$ in general. **Estimated time:** 4 hours.
- Prove that every n-dimensional vector space V over a field $\mathbb{K}$ is isomorphic to $\mathbb{K}^n$. **Estimated time:** 20 hours.
- Check the code and make clearer, shorter and more legible the proofs. **Estimated time:** 5 hours.

TOTAL ESTIMATED TIME OF THIS TASK: 29.5 hours.

10. SUBSPACES

This seems an easier section. We will try to define a subspace of a vector space and prove several properties.

- Definition of subspace. **Estimated time:** 1 hour.
- Prove that a subspace always contains 0_V. **Estimated time:** 1 hour.

- Demonstrate that a subspace is a vector space. **Estimated time:** 5 hours.

- $\{0_V\}$ and the whole vector space V are subspaces. **Estimated time:** 2 hours.

- Check the code and make clearer, shorter and more legible the proofs. **Estimated time:** 1.5 hours.

TOTAL ESTIMATED TIME OF THIS TASK: 10.5 hours.

11. CALCULUS OF SUBSPACES

Here we want to prove the three theorems presented by Halmos.

- Prove the theorem 1: the intersection of any collection of subspaces is a subspace. **Estimated time:** 3 hours

- Define the subspace spanned by a set. **Estimated time:** 1 hour.

- Prove the theorem 2: If S is any set of vectors in a vector space V and if M is the subspace spanned by S, then M is the same as the set of all linear combinations of elements of S. **Estimated time:** 5 hours.

- Prove the theorem 3: If A and B are any two subspaces and if M is the subspace spanned by A and B together, then M is the same as the set of all vectors of the form $a \oplus_V b$, with a in A and b in B. **Estimated time:** 5 hours.

- Check the code and make clearer, shorter and more legible the proofs. **Estimated time:** 3 hours

TOTAL ESTIMATED TIME OF THIS TASK: 17 hours.

12. DIMENSION OF A SUBSPACE

In this section Halmos presents two theorems which don't seem very difficult once we have proved all previous results.

- Prove the theorem 1: A subspace M in an n-dimensional vector space V is a vector space of dimension $\leq n$. **Estimated time:** 3 hours.

- Prove the theorem 2: Given any m-dimensional subspace M in an n-dimensional vector space V, we can find a basis $\{x_1, \ldots, x_m, x_{m+1}, \ldots, x_n\}$ in V so that $x_1, \ldots, x_m$ are in M and form, therefore, a basis of M. **Estimated time:** 5 hours.

- Check the code and make clearer, shorter and more legible the proofs. **Estimated time:** 1.5 hours

TOTAL ESTIMATED TIME OF THIS TASK: 9.5 hours.

13. DUAL SPACES

- Define a linear functional on a vector space. **Estimated time:** 1 hour.

- Define the dual space of a vector space. **Estimated time:** 1 hour.

- Prove that the dual space is a vector space. **Estimated time:** 8 hours.

- Check the code and make clearer, shorter and more legible the proofs. **Estimated time:** 2 hours

TOTAL ESTIMATED TIME OF THIS TASK: 12 hours.

14. BRACKETS

This is a section about notation of linear functionals. We decide to use the notation proposed ($[x, y]$ instead of $y(x)$ for a linear functional y of a vector space V and a vector x of V) to follow exactly Halmos. We have to be careful with the implementation, so that the term $[.,.]$ could be used in Isabelle.

- Implement the notation proposed in Halmos. **Estimated time:** 1 hour.

TOTAL ESTIMATED TIME OF THIS TASK: 1 hour.

15. DUAL BASES

It is difficult to estimate the time that we will spend on the following theorems because it will depend on the implementation of the previous concepts. Nevertheless, these theorems are very important and not so laborious as some of the previous ones.

- Prove the theorem 1: If V is an n-dimensional vector space, if $\{x_1, \ldots, x_n\}$ is a basis in V, and if $\{\alpha_1, \ldots, \alpha_n\}$ is any set of n scalars, then there is one and only one linear functional y on V such that $[x_i, y] = \alpha_i$ for $i = 1, \ldots n$. **Estimated time:** 6 hours.

- Prove the theorem 2: if V is an n-dimensional vector space and if $X = \{x_1, \ldots, x_n\}$ is a basis in V, then there is a uniquely determined basis X' in the dual space of V, $X' = \{y_1, \ldots, y_n\}$, with the property that $[x_i, y_j] = \delta_{ij}$. Consequently the dual space of an n-dimensional space is n-dimensional. **Estimated time:** 6 hours.

- Formalize the theorem 3: If u and v are any two different vectors of the n-dimensional vector space V, then there exists a linear functional y on V such that $[u, y] \neq [v, y]$; or, equivalently, to any non-zero vector x in V there corresponds a y in the dual space of V such that $[x, y] \neq 0$. **Estimated time:** 6 hours.

- Check the code and make clearer, shorter and more legible the proofs. **Estimated time:** 3.5 hours

TOTAL ESTIMATED TIME OF THIS TASK: 21.5 hours.

16. REFLEXIVITY

This is the last section that we want to formalize. This is not very laborious using previous results: we want to prove that every finite dimensional vector space is isomorphic with the dual of its dual space. We can do it next way: We know using theorem 2 of previous section that the dual of a vector space has the same dimension of the vector space, so dimension of V is the same than the dimension of its dual space, V', and hence equal to the dimension of the dual of its dual, V''. We also know that an n-dimensional vector space over a field $\mathbb{K}$ is isomorphic to $\mathbb{K}^n$, so V and V'' are isomorphic to $\mathbb{K}^n$ and hence also between them. Nevertheless, the proof in Halmos gives an explicit definition of the isomorphism.

- Prove the theorem: If V is a finite-dimensional vector space, then corresponding to every linear functional z_0 on V', there is a vector x_0 in V such that $z_0(y) = [x_0, y] = y(x_0)$ for every y in V'; the correspondence $z_0 \leftrightarrows x_0$ between V'' and V is an isomorphism. **Estimated time:** 8 hours.

- Check the code and make clearer, shorter and more legible the proofs. **Estimated time:** 2 hours

TOTAL ESTIMATED TIME OF THIS TASK: 10 hours.

17. DOCUMENTATION AND MANAGEMENT

Finally, we will make the documentation of the whole project. We will try to reduce, summarize and explain the main proofs. The fact of doing it in LaTeX and in English will have as a consequence that I will be slower than we would like. Taking into account the nature of the project, it will need a continuous supervision of the advisor of the project, making meetings or replying e-mails in order to solve doubts, problems, questions .... This part is very difficult to estimate, and it will be carried out during the whole process of development.

- Make the documentation of the project. **Estimated time:** 120 hours.
- Reviews and corrections. **Estimated time:** 20 hours.
- Meetings with the advisor, resolution of problems, doubts ...

TOTAL ESTIMATED TIME OF THIS TASK: 140 hours.

If we make the sum of all estimated times, we will obtain that the estimated duration of the project is 439.75 hous, i.e, if we make every task in time we will have a margin of more than 10 hours for possible delays.

Finally a brief comment about the risk management. Besides the normal risk of a common individual final year project (to suffer some illnes, to lose the pendrive with the code, find a job and then break the initial planification ...) we have two very important risks, particular to this kind of project, that we have to analyze in detail: the difficulty and the feasibility of the project.

- **Difficulty:** We can know which parts of the project seem to be very difficult, but we can't estimate how difficult it will be, or if there will be also another hard proofs. It is very possible that we will be sometimes stuck in our development. It is impossible to avoid it due to the uncertainty that surrounds the project. If finally this risk appears, we could look for similar formalized proofs (if exist) in the Isabelle/HOL library and on the internet, we could ask to other people that have more knowledge of this matter, make questions in the user mailing list of Isabelle ...

- **Feasibility:** At first, the development created by Gertrud Bauer to formalize the Hanh-Banach theorem for real vector spaces in Isabelle/HOL[12] makes feasible the beginning of the implementation, since there are concepts that Bauer has already implemented (but for the real case, and we will generalize it for any field). If during the development we realize that there exist something that is impossible to be implemented or that it is very hard, and it needs an effort which isn't worth, then we will avoid it and we would implement another easier results, i.e., we will took a turn to the project.

Chapter 3

Mathematical Definitions

In this chapter we will define previous mathematical structures and notions that are necessary to implement a *vector space*. Luckily, the great majority of these structures are already implemented in Isabelle. Let's begin:

Definition 3.0.1 *A* monoid *is a set G together with an operation $\odot\colon G \times G \to G$ that satisfies the following axioms. Let $a, b, c \in G$:*

- *Associativity:* $a \odot (b \odot c) = (a \odot b) \odot c$
- *Identity element: there exists an element $e \in G$ that verifies*

$$e \odot a = a \odot e = a$$

In other words, a monoid *is an algebraic structure with a single associative binary operation ($\odot$) and an identity element (e).*

We may note that there is another property implicit in this definition: the *closure*: if $a, b \in G$ then $a \odot b \in G$. The set G is not empty because second axiom implies the existence of an identity element.

In addition, we must realize that the identity element will depend on the binary operation. We can talk about *additive monoids* when the operation is the sum ($+$) and then the identity element is called zero (denoted by $\mathbf{0}$).

On the other hand, we define a *multiplicative monoid* as a monoid whose operation is the multiplication (it is commonly denoted by $*$ or $\otimes$) and the identity element is the one ($\mathbf{1}$).

Nevertheless, the operation of a monoid is not necessarily an addition or a multiplication in the sense of elementary arithmetic. For example, there exist monoids given by a rotation, transformation ... instead of an additive or multiplicative operation (for example, cyclic groups are monoids and the operation is a rotation. A chain of characteres with the operation concatenate is also it. See [6]).

A brief comment about notation: once we have introduced a monoid $(G, \odot, e)$, and it is clear what we have, then we can speak of "the monoid G", though stricly speaking, this is the underlying set and is just one of the ingredients of $(G, \odot, e)$.

Definition 3.0.2 *An* abelian (or commutative) monoid *is a monoid* $(G, \odot, e)$ *whose elements also verify the commutative property:*

$$\forall a, b \in G \text{ we have } a \odot b = b \odot a$$

Definition 3.0.3 *An element a of a monoid $(G, \odot, e)$ is* invertible *if there exists $b \in G$ in a way that $a \odot b = b \odot a = e$, where e is the identity element.*

For example, in an additive monoid $(G, +, \mathbf{0})$ the definition will be: $a+b = b+a = \mathbf{0}$ and the inverse element (also called unit) is usually represented by $-a$.

Analogously, in a multiplicative monoid $(G, \otimes, \mathbf{1})$ the definition will turn into: $a \otimes b = b \otimes a = \mathbf{1}$ and the invertible element b will be denoted by a^{-1}.

Definition 3.0.4 *The set of invertible elements is called* set of units.

Definition 3.0.5 *A* group *is a monoid all of whose elements are invertible.*

Definition 3.0.6 *An* abelian (or commutative) group *is a group $(G, \odot, e)$ whose elements also verify the commutative property* [1]*:*

$$\forall a, b \in G \text{ we have } a \odot b = b \odot a$$

As well as in monoids, we have *additive abelian groups* $(G, +)$ and *multiplicative abelian groups* $(G, \otimes)$.

[1] Another possibility to define it is to say that an *abelian group* is an abelian monoid all of whose elements are invertible.

Definition 3.0.7 *A* ring *is a set G equipped with two binary operations $+\colon G \times G \to G$ and $\otimes\colon G \times G \to G$ called addition and multiplication. To qualify as ring, the set and two operations (G,+,⊗) must satisfy the following requirements known as the ring axioms:*

- $(G, +, \mathbf{0})$ *is required to be an additive abelian group.*
- $(G, \otimes, \mathbf{1})$ *is required to be a multiplicative monoid.*
- $\forall a, b, c \in G$ *are satisfied the distributive laws:*

  i) $a \otimes (b + c) = (a \otimes b) + (a \otimes c)$
  ii) $(a + b) \otimes c = (a \otimes c) + (b \otimes c)$

Definition 3.0.8 *A* commutative ring *is a ring (G,+,⊗) in which the multiplication operation is commutative, that is:*

$$\textit{If } a, b \in G \textit{ then } a \otimes b = b \otimes a$$

Definition 3.0.9 *An* integral domain *is a commutative ring (G,+,⊗) in which $\mathbf{1} \neq \mathbf{0}$ and it has not zero-divisors (the integral property), that means:*

$$\textit{If } a, b \in G \textit{ and } a \otimes b = \mathbf{0}, \textit{ then } a = \mathbf{0} \textit{ or } b = \mathbf{0}$$

Chapter 4

Theorem proving: Isabelle

4.1 Some ideas on theorem proving

Here we present a brief introduction to the field of theorem proving. There are some good articles in which formal proofs are explained in detail, for example [17, 19, 18] an specially [22]. We will follow them to present this section.

Mathematics are traditionally considered as an exact science, free of imperfections. The production of a theorem requires creative ability which gives rise to a proof of it. However, once the theorem is proved, the activity of verifying if the proof is correct is an objective activity. Generally, the theorems published in journals are considered as correct, since they have been subjected to checkings by another renowned mathematicians or the publishers. However, the history of mathematics has seen publications of false results which have not been detected during long periods of time. One example of that is the demonstration of the Fermat's Last Theorem, presented by Andrew Wiles in 1993. A checker found an error in this proof and it was corrected in 1995 [20].

Nowadays, computers are support tools in the mathematical work, specially in the process of theorem proving. A computer can be useful in several ways, for example it can be used as a part of the proof in order to solve a huge number of cases, since the proof in paper could be unapproachable by a person in a reasonable time. There exist some examples of this. Perhaps the most famous are the Four-Colour Theorem [23] and the Kepler's

Conjecture [24].

A second use of computers is to try to check the correction of mathematical proofs through their formalization. Traditional mathematical proofs are written in a way to make them easily understandable by mathematicians. Routine logical steps are omitted and an enormous amount of context is assumed on the part of the reader. On the contrary, a formal proof is a proof in which every logical inference has been checked all the way back to the fundamental axioms of mathematics: all the intermediate logical steps are supplied, without exception. This makes that a formal proof be much more laborious than the traditional demonstration.

The work to formalize a proof is, usually, as follows. A formal proof begins with a traditional mathematical proof written in detail, making explicit the whole premises and all cases. After that, a proof assistant is selected to formalize it. This selection is based on the implemented logics or in the libraries developed in each one. Once we have selected it, we have to prove the result and the assistant will ensure that there won't be inferences which don't be product of the axioms and logic rules implemented. Hence, we will be sure that our proof is correct.

There exist different proof assistants with their own characteristics and logics. These proof assistants compete amongst them in the same way that the programming languages or the operating sistems. On the webpage [21] there is a list that keeps track of the formalization status of a hundred well-known theorems. If we analyze this list of theorems, it turns out that there are five proof assistants that have been significantly used for formalization of mathematics: *HOL Light, Mizar, ProofPower, Isabelle* and *Coq*. With the support of this proof assistants, several important theorems have been proved recently. Maybe the most impressive proof is the Four-Colour Theorem by Georges Gonthier using Coq in 2004[1].

[1]We have to remark the difference between the traditional proof of this theorem and the formal proof. Both use a computer: the first one as a calculation tool in order to rule out cases. In the second one a computer is used to reproduce minutely a huge number of logic steps that prove the theorem.

4.2 Isabelle introduction by example

Isabelle [25] is a generic proving assistant (in the sense of that we can implement several logics on it). It is programmed in *ML*, a functional programming language very extended. In its kernel (known as *Isabelle/Pure* or as metalogic) we can find a group of inference basic rules which correspond to a part of the higher order logic, such as Alonzo Church [38] developed, also named simple type theory. This metalogic can be defined by its syntax, its semantics and the group of inference rules.

The syntax of the metalogic results from a *type system* and the *terms* that can be defined using them. The available types in our system will be basic types: $(\sigma, \tau, \upsilon, \dots)$, and functional types of the form $\sigma \to \tau$.

The *terms* are the ones of λ-calculus – constants, variables, abstractions, combinations – with the proper restrictions of the types (for example, we can't declare the corresponding term to the application of the function f to a given x if f hasn't a functional type, $f : \sigma \to \tau$ and x is of the corresponding type $x : \sigma$).

The basic types and the constants could be increased in each logic that we want to implement. It will always include a type *prop*, representing the propositions, and the logic constants of the metalogic (true, $\top$ and false $\bot$). Any *formulae* in the system will be a term of the type *prop*. Let be ϕ and ψ formulaes or terms of the type *prop*. We define the operation $\phi \Rightarrow \psi$, which means 'ϕ implies ψ'. We also define the universal quantifier $\bigwedge$, such as $\bigwedge x.\phi$ be equivalent to 'for all x, ϕ is true'. We also have the equality $a \equiv b$.[2]

With respect to the semantics, each type of the metalogic denotes a non-empty set. The fact of allowing empty sets would make that some of the rules that we will introduce later would be false (they would need to be reformulated in a more complex way). The logics implemented over the metalogic could accept empty types. From the sets of each type, a functional type $\sigma \to \tau$ denotes the set of functions of σ to τ. A closed term of type σ denotes a value of the corresponding set. If to each constant x of a type σ we assign a value $b(x)$ of a type τ, the λ-abstractions $\lambda x\colon \sigma.b(x)$ denote functions.

The available inference rules are the next:

[2]The symbols $\Rightarrow$, $\bigwedge$ and $\equiv$ are chosen for the metalogic, keeping this way available for the logics that we will implement over it the most usual $\longrightarrow$, $\forall$, $=$.

- About the implication operator, $\Rightarrow$-introduction and $\Rightarrow$-elimination:

$$\frac{\begin{array}{c}[\phi]\\ \psi\end{array}}{\phi \Rightarrow \psi} \qquad\qquad \frac{\phi \Rightarrow \psi \qquad \phi}{\psi}$$

- About the universal quantifier $\bigwedge$, $\bigwedge$-introduction and $\bigwedge$-elimination:

$$\frac{[\phi]}{\bigwedge x.\phi} \qquad\qquad \frac{\bigwedge x.\phi}{\phi[b/x]}$$

  The first rule must satisfy that x is not a free variable in ϕ.

- About the equality, reflexivity, simmetry and transitivity:

$$a \equiv a \qquad\qquad \frac{a \equiv b}{b \equiv a} \qquad\qquad \frac{a \equiv b \qquad b \equiv c}{a \equiv c}$$

- About the abstraction operator λ, α-conversion, β-conversion and extensionality:

$$(\lambda x.a) \equiv (\lambda y.a[y/x]) \quad ((\lambda x.a)(b)) \equiv a[b/x] \qquad \frac{f(x) \equiv g(x)}{f \equiv g}$$

  The rule α-conversion requires that y is not free in a. The rule of extensionality is true if x is not free in either f or g. The extensionality is equivalent to μ-conversion, i.e. $(\lambda x.f(x)) \equiv f$ (with x not free in f).

- The abstraction and combination rules are the following ones (the abstraction rule requires that x isn't in the premises):

$$\frac{[a \equiv b]}{(\lambda x.a) \equiv (\lambda x.b)} \qquad\qquad \frac{f \equiv g \qquad a \equiv b}{f(a) \equiv g(b)}$$

- The logic equivalence means the equality of the truth values.

$$\frac{\begin{array}{cc}[\phi] & [\psi]\\ \psi & \phi\end{array}}{\phi \equiv \psi} \qquad\qquad \frac{\phi \equiv \psi \qquad \phi}{\psi}$$

As we have seen, the underlying metalogic to the system is simple. We can assert that it is the *kernel* of the logic system in which we can rely

on. Over this metalogic several logics can be implemeted. For example, the standard distribution of *Isabelle* includes implementations of first order logic, of the Zermelo-Fraenkel's set theory, of the classic computational logic... It also contains an implementation of higher order logic (HOL), in which we will focus on. HOL is the most used theory in *Isabelle* and its expressive capacity has been used to formalize numerous mathematical results.

In order to be able to implement HOL over the metalogic, we have to define a new type system for our logic which fulfills the properties of the higher order logic (also known as simple type theory), and a new group of rules (or axioms) which define the logical system.[3]

With respect to the types, it is allowed to define new types whenever they aren't empty. There also exists a mechanism which allows us to define new types as *subsets* of existing types. In this way, we can define the type *product* of two types (an hence we can work with tuples), the type *sum* of two defined types (producing the set of the direct sums of the elements of both types), or also types defined by induction. The system provides facilities which turn the definition of types into something trivial. With this facilities, we can define (in fact, they are included in the standard library of *Isabelle*) types for the natural numbers, integers, reals, complexes, groups, rings and almost every type of structure which appears in a mathematical text (in general, in a proof assistant, it is easier to represent structures than to prove properties over themselves).

With respect to the new rules (or axioms) added, the list is small:

refl: $t = t$
subst: $[\ s = t\ ;\ P\ s\ ] \Longrightarrow P\ t$
ext: $(\bigwedge x.\ f\ x = g\ x) \Longrightarrow (\lambda x.\ \ f\ x) = (\lambda x.\ \ g\ x)$
impI: $(P \Longrightarrow Q) \Longrightarrow P \longrightarrow Q$
mp: $[\ P \longrightarrow Q\ ;\ P\ ] \Longrightarrow Q$
iff: $(\ P \longrightarrow Q\ ) \longrightarrow (\ Q \longrightarrow P\ ) \longrightarrow (\ Q = P\ )$
someI: $P\ x \Longrightarrow P\ (\varepsilon x.\ P\ x)$
True_or_False: $(P = True) \vee (P = False)$

The rule **ext** represents the extensionality of the functions (with respect

[3]In general, when we work with proof assistants, we have to be careful when we include axioms, since an inadequate or wrong axiom would make our system inconsistent so that we could prove any result.

to the universial quantifier $\bigwedge$ of the metalogic). The rule **iff** imposes that the formulaes logically equivalent are equal. The rule **someI** contains the defining property of the Hilbert's descriptive operation ϵ. This operator is a quantifier $\epsilon x.P[x]$ which returns some x that verifies P, in case that x exists. The operator will be used to define the existential quantification:

$$\exists x.Px \equiv P(\epsilon x.Px)$$

Finally, the rule **True_or_False** (also known as *law of excluded middle*) makes that the implemented logic be classic. Removing this axiom, we could also implement constructive logics over the Isabelle metalogic. The other implemented axioms assign connectors of our particular logic to the connectors of the metalogic (for example,the rule **impI** links $\longrightarrow$ to $\Longrightarrow$). The rest of the elements used in the logic (the constants *True* and *False*, the connectors $\neg$, $\forall$, $\wedge$, $\vee$, the unique existencial $\exists_1 \ldots$) can be defined (without the necessity of introducing them axiomatically) from the previous connectives. For example, *True* can be defined equal to $(\lambda x.x) = (\lambda x.x)$ and $\neg$ can be defined as $\neg P = (P \longrightarrow False)$.

Now we are going to see a simple example of a proof in higher order logic, with a result of classical logic (a De Morgan's law).

```
lemma "¬ (A ∧ B) ⟶ ¬ A ∨ ¬ B"
proof
  assume n: "¬ (A ∧ B)"
  show "¬ A ∨ ¬ B"
  proof (rule ccontr)
    assume nn: "¬ (¬ A ∨ ¬ B)"
    have "¬ A"
    proof (rule notI)
      assume a: "A"
      have "¬ B"
      proof (rule notI)
        assume b: "B"
        from a and b have "A ∧ B" by (rule conjI)
        with n show False by rule
      qed
      hence "¬ A ∨ ¬ B"
        using disjI2 [of "¬ B" "¬ A"] by fast
      with nn show False by fast
```

```
    qed
    hence "¬ A ∨ ¬ B"
      using disjI1 [of "¬ A" "¬ B"] by fast
    with nn show False by fast
  qed
qed
```

The wording expresses that the negation of the conjunction of two propositions A and B is equal to the disjoint negation of each one. As we can see, the notation in the system is similar to a proof in natural language. Using the command *show* we fix a proposition that we want to demonstrate. The command *have* allows us to fix partial results necessary in the proof, which can be used later. When we use the command *proof* we open a new context in which we will try to prove the fixed proposition (by *show* or *have*). Each proof is made using *proof procedures*. Those procedures act over the fact that we want to demonstrate (in the same way that an algorithm acts over the input data). For example, in the above proof we can see some such as *rule* or *fast*. It can also obtain additional help with previous lemmas or axioms already proved. For example, in the two aforementioned procedures, *rule* tries to apply the theorem which has as a parameter to the result that we want to prove. If both results coincide (the system will try to instantiate the corresponding variables), the state of our demonstration will change, transforming itself into the premises of the rule that we have included as a parameter of *rule*. With successive changes in the result, we turn it into easier results; if these results coincide with some of our premises (in the result shown, $\neg(A \wedge B)$) or with some previous result proved in Isabelle/HOL, the proof (of the final result or of the intermediate results) will be completed (an we will write the command *qed*).

In any case, as in a mathematical demonstration, we can give different proofs for the same result. In particular, for the previous result and making use of the most advanced procedures of the system, the result can be proved in one line:

```
lemma "¬ (A ∧ B) ⟶ ¬ A ∨ ¬ B"
  by simp
```

If a user is interested in how the system has made the proof with the method *simp* in order to remake the proof in detail, he can do it inside the

system with total transparency.

We have used three procedures: *rule, fast* and *simp*, but there exist much more procedures, with different capacities and power. For example, *auto, blast, fastsimp, force, best*... A brief explanation of the main methods can be found here [26]. In addition, the Isabelle reference manual and the system manual can be consulted here [27, 28].

Nevertheless, not all the formalization of proofs is so slow. One important and useful tool for our development has been *sledgehammer* [39]. Using this tool we can automatize proofs, so we spare work. A good explanation of how it works can be found in [26]: Sledgehammer is Isabelle's subsystem for harnessing the power of first-order automatic theorem provers. Given a conjecture, it heuristically selects a few hundred relevant facts (lemmas, definitions, or axioms) from Isabelle's libraries, translates them to first-order logic along with the conjecture, and delegates the proof search to external resolution provers (E[29],SPASS[30], and Vampire[31]) and SMT solvers (CVC3[32], Yices[33], and Z3[34]). Sledgehammer is rather effective[35] and has been useful in our development, helping in the completion of non-trivial goals with its capacity to use previous results.

4.3 Locales and Abstract Algebra

In this section we will find some examples about creating environments and an infrastructure (with inheritance between structures and so on) of algebraic structures which are going to be used in our development. It is just a brief introduction to locales (a kind of module in which we can fix variables and declare assumptions) in Isabelle/HOL: we want to give an idea of how the system works in broad strokes, without going into details. We can find detailed explanations of type classes and locales in the documentation for Isabelle [3, 4].

Our first objective is to achieve a correct representation of the notion of vector space in Isabelle/HOL. For that, we will try to give an appropiate definition using existing concepts in Isabelle's libraries.

To be able to implement this structure of vector space, we will begin studying how is represented the simplest structure: a monoid. The Isabelle definition of monoid consists of, firstly, the type of the objects which have the

same structure than a monoid (with respect to the type system introduced), and of the properties that define what objects of type monoid are really monoids.

Let's see the definition of the type of monoids:

```
record 'a monoid =
  carrier :: "'a set"
  mult    :: "['a, 'a] ⇒ 'a" (infixl "⊗₁" 70)
  one     :: 'a ("1₁")
```

With respect to the defined type, we must clarify that its definition has been made using a *record*. A record is part of Isabelle/HOL type system, because they are internally encoded like a product of types in which each type is labelled with the field name assigned to that record.

In the previous case, the record consists of three components: *carrier*, which represents the set of the algebraic structure, a binary operation (*mult*) and finally a constant (*one*). The underlying type of the structure (denoted by *'a*) is a variable type and thanks to that we can represent monoids of naturals, lists, integers or any other data type implemented in the system.

Isabelle provides features to introduce simplest notation defined by the user. For example, with "**infixl** $\otimes_1$" we are creating an alias for the operation *mult G* which could be denoted using the infix operator $\otimes_G$. This operation has been declared like left-associative, with a priority order of 70 (to be compared with other operations priority). If the structure (G) is clear from the context, we can also abridge previous notation to $\otimes$. Similarly, if there exists a single monoid in our work context we can do the same with $\mathbf{1}_G$ writing $\mathbf{1}$.

At this point, we realize that we are not representing a monoid in general: giving the operation $\otimes$ and the constant $\mathbf{1}$ we are defining the data type of a *multiplicative monoid*. We will explain later how Isabelle manages to represent an additive monoid.

Now we have the data type of a monoid, but not all structures with a carrier, a binary operation and a constant are monoids. There are other properties which must be satisfied (see the definition of monoid in second chapter). Due to the lack of expressive power of the Isabelle/HOL type system, those properties must be specified separately:

```
locale monoid =
  fixes G (structure)
  assumes m_closed [intro, simp]:
        "⟦x ∈ carrier G; y ∈ carrier G⟧ ⟹ x ⊗ y ∈ carrier G"
      and m_assoc:
        "⟦x ∈ carrier G; y ∈ carrier G; z ∈ carrier G⟧
         ⟹ (x ⊗ y) ⊗ z = x ⊗ (y ⊗ z)"
      and one_closed [intro, simp]: "1 ∈ carrier G"
      and l_one [simp]: "x ∈ carrier G ⟹ 1 ⊗ x = x"
      and r_one [simp]: "x ∈ carrier G ⟹ x ⊗ 1 = x"
```

In the previous code, G makes reference to the *monoid* that we are defining, and after that each line represents one of the properties of a monoid. We can see that the binary operation ($\otimes$) is closed over the carrier (*m_closed*) and it is also associative (*m_assoc*). So we have defined the first two properties: closure[4] and associativity. To represent the last property (identity element)[5] we use three assumptions: "$\mathbf{1} \in$ *carrier* G" (*one_closed*), "$\mathbf{1} \otimes x = x$" (*l_one*) and "$x \otimes \mathbf{1} = x$" (*r_one*). To be able to define previous conditions we have used a **locale**, a kind of module in which we can fix variables (in this case G) and declare assumptions (*m_assoc*, *m_closed* ...).

We have a predicate *monoid* whose type is the type of fixed structures in the locale and it returns *True* when all premises (*m_closed, r_one* ...) are satisfied and *False* in other case.

Generally, when introducing new concepts or definitions we also provide the system with an introduction rule, which acts in the opposite way to the definition: "Whenever we have something which satisfies every property of the newly introduced definition, then it satisfies the definition". For example, if we want to check if some structure is a *monoid* we would use the introduction rule created to make it. Normally those lemmas are called "*name_of_the_structureI*" (in the following case, "monoidI"):

```
lemma monoidI:
  fixes G (structure)
```

[4]This property is implicit in the definition of *monoid*, but we must declare it in Isabelle because we have a *carrier* G (whose elements are of type *'a*) and we must demand of our binary operation to take pairs of elements in the carrier to elements in the carrier.

[5]There exists an element $e \in G$ that verifies $e \odot a = a \odot e = a$

```
  assumes m_closed:   "!!x y. [| x ∈ carrier G; y ∈ carrier G |]
    ==> x ⊗ y ∈ carrier G"
    and one_closed: "1 ∈ carrier G"
    and m_assoc:
      "!!x y z. [| x ∈ carrier G; y ∈ carrier G; z ∈ carrier G |]
      ==> (x ⊗ y) ⊗ z = x ⊗ (y ⊗ z)"
    and l_one: "!!x. x ∈ carrier G ==> 1 ⊗ x = x"
    and r_one: "!!x. x ∈ carrier G ==> x ⊗ 1 = x"
  shows "monoid G"
  by (fast intro!: monoid.intro intro: assms)
```

Using the representation of monoid in Isabelle/HOL, definition of multiplicative abelian monoids is easy:

```
locale comm_monoid = monoid +
assumes m_comm: "⟦x ∈ carrier G; y ∈ carrier G⟧
⟹ x ⊗ y = y ⊗ x"
```

With this representation we have defined a *multiplicative abelian monoid*. But for example, how we can define and *additive abelian monoid*? The confusion exists because Isabelle uses this naming convention [5, page 52]: "multiplicative structures that are commutative are called *commutative* and additive structures are called *abelian*".

Next table clarifies how are called the different structures in Isabelle:

USUAL NAME	NAME IN ISABELLE
multiplicative monoid	monoid
multiplicative abelian monoid	comm_monoid
additive abelian monoid	abelian_monoid
multiplicative group	group
multiplicative commutative group	comm_group
additive abelian group	abelian_group

Using this convention, an *additive abelian monoid* (an *abelian monoid* in Isabelle) is represented as follows:

```
locale abelian_monoid =
  fixes G (structure)
  assumes a_monoid:
     "monoid (| carrier = carrier G, mult = add G, one = zero G |)"
  and a_comm: "⟦x ∈ carrier G; y ∈ carrier G⟧ ⟹ x ⊕ y = y ⊕ x"
```

Note that we have not created a record like this:

```
record 'a abelian_monoid =
  carrier :: "'a set"
  add    :: "['a, 'a] ⇒ 'a" (infixl "⊕₁" 70)
  zero     :: 'a ("0₁")
```

We have used the definition of *monoid* but imposing that this structure must satisfy the properties of a monoid with the *add* (also denoted $\oplus$) and the constant *zero* (or **0**).[6]

Based on the definition of *monoid*, if we include a premise imposing that every element of the carrier be invertible we obtain the definition of group[7].

Firstly, we define the notion of *Units*:

```
definition  Units :: "'a monoid => 'a set"
   where "Units G == {y. y ∈ carrier G ∧
    (∃x ∈ carrier G. x ⊗ y = 1 ∧ y ⊗ x = 1)}"
```

After that we can represent a group:

```
locale group = monoid +
  assumes Units: "carrier G <= Units G"
```

Up to now, we have seen how Isabelle/HOL implements the definitions of monoid, abelian monoid and group. It is time to explain how it represents *rings*:

[6] Really, the implementation of an additive abelian monoid in Isabelle/HOL does not follow the logical relationship between algebraic structures. As we need the addition operation ($\oplus$) and the constant zero (**0**), firstly it is declared the type ring (with mult, add, zero and one) and using it an additive abelian monoid can be implemented. This is because Isabelle only admits simple inheritance between record types and thus the definition of ring cannot be done by merging record types abelian_monoid and monoid, since they both contain a *carrier* field. So the additive version of monoid operations has been made explicit only on the ring record type definition.

[7] Another possibility to define the notion of group is to include a new operation *inv* which assigns each element of the group to its inverse.

```
record 'a ring = "'a monoid" +
  zero :: 'a ("0₁")
  add :: "['a, 'a] ⇒ 'a" (infixl "⊕₁" 65)
```

The previous structure, as in the example about monoids, defines the data type that a ring must have. As we can see, we can define it making use of a sort of inheritance (of the record monoid) that Isabelle provides to us. This is a great advantage in the sense of reusing code, because type ring has been defined like a monoid with another additional properties. Therefore, and thanks to polymorphism of parameters in functions in HOL, every function or predicate which admits a variable of monoid type will also admit ring type variables.

Appart from ring type, we must make a definition which really characterizes a ring structure:

```
locale ring = abelian_group R + monoid R for R (structure) +
  assumes l_distr:"[|x ∈ carrier R;y ∈ carrier R; z ∈ carrier R|]
      ⟹ (x ⊕ y) ⊗ z = x ⊗ z ⊕ y ⊗ z"
  and r_distr:"[|x ∈ carrier R;y ∈ carrier R; z ∈ carrier R |]
      ⟹ z ⊗ (x ⊕ y) = z ⊗ x ⊕ z ⊗ y"
```

To define the properties which represent a ring we are reusing code again, so we are minimizing work. If we compare the definition given in chapter 3 with this one, we are going to realize that they are exactly equal: a ring will satisfy the properties of an additive abelian group (abelian group in Isabelle) and the properties of a multiplicative monoid (monoid in Isabelle). In addition, to this, we include the distributive laws which are represented by *l_distr* and *r_distr*.

From this definition and adding the property of commutativity we will obtain the notion of *commutative ring*:

```
locale cring = ring + comm_monoid R
```

To sum up, if we want to define correctly an algebraic structure in Isabelle/HOL following locales we must do two things:

i) Define its data type using a record.

ii) Define its properties using a locale.

In the first step we show the parts of the structure being defined (sets, operations, constants...) and in the second we define what properties and relations must be satisfied by these parts.

As we have seen in this section, we can save work (in both steps) reusing code through inheritances. Note that if we are using inheritances in the second step and we are not including new components in the structure (the underlying data type is the same as the data type of the structure of which we are inheriting) then we can avoid to make the first step (Isabelle will make it instead of us).

Finally we must remark that there are another ways to define an algebraic structure in Isabelle/HOL, for example following *type classes*[3]. In spite of there also exists a developement of the algebraic structures using type classes[7] we decided to follow locales due to his advantages. In particular, the use of explicit carriers (not as type, but like sets) will going to make possible to define subsets (we can not make it with subtypes). For example, this is very important in order to define the notion of a *basis*: using locales it will be easy, but it is not clear how make it following type classes. In addition, most of algebraic structures and its properties that we need are more developed following locales (we can base on them to make our implementation).

Chapter 5

Fields

In this chapter we will cover the representation of fields in Isabelle/HOL using locales and inheritance.

First of all, let's see the definition of *field* as presented in Halmos:

Definition 5.0.1 *A field is a set* $\mathbb{K}$ *together with two binary operations called addition* $(+)$ *and multiplication* $(*)$*. The following axioms are also satisfied:*

A) *To every pair,* a *and* b*, of elements in* $\mathbb{K}$ *there corresponds a scalar* $a + b$*, called the sum of* a *and* b*, in such a way that:*

- *(1) addition is commutative,* $a + b = b + a$
- *(2) addition is associative,* $a + (b + c) = (a + b) + c$
- *(3) there exists a unique scalar* $\mathbf{0}$ *(called* zero*) such that* $a + \mathbf{0} = a$ *for every scalar* a
- *(4) to every scalar* a *there corresponds a unique scalar* $-a$ *such that* $a + (-a) = \mathbf{0}$

B) *To every pair,* a *and* b*, of elements in* $\mathbb{K}$ *there corresponds a scalar* $a * b$*, called the product of* a *and* b*, in such a way that:*

- *(1) multiplication is commutative,* $a * b = b * a$
- *(2) multiplication is associative,* $a * (b * c) = (a * b) * c$
- *(3) there exists a unique non-zero element* $\mathbf{1}$ *(called* one*) such that* $a * \mathbf{1} = a$ *for every scalar* a

(4) to every scalar a there corresponds a unique scalar a^{-1} *(or* $\frac{1}{a}$*) such that* $a * a^{-1} = \mathbf{1}$

C) *Multiplication is distributive with respect to addition:* $a * (b + c) = a * b + a * c$

Note that another alternative definition is that:

Definition 5.0.2 *A field is an integral domain all of whose elements (except* $\mathbf{0}$*) are invertible with respect to the multiplication.*

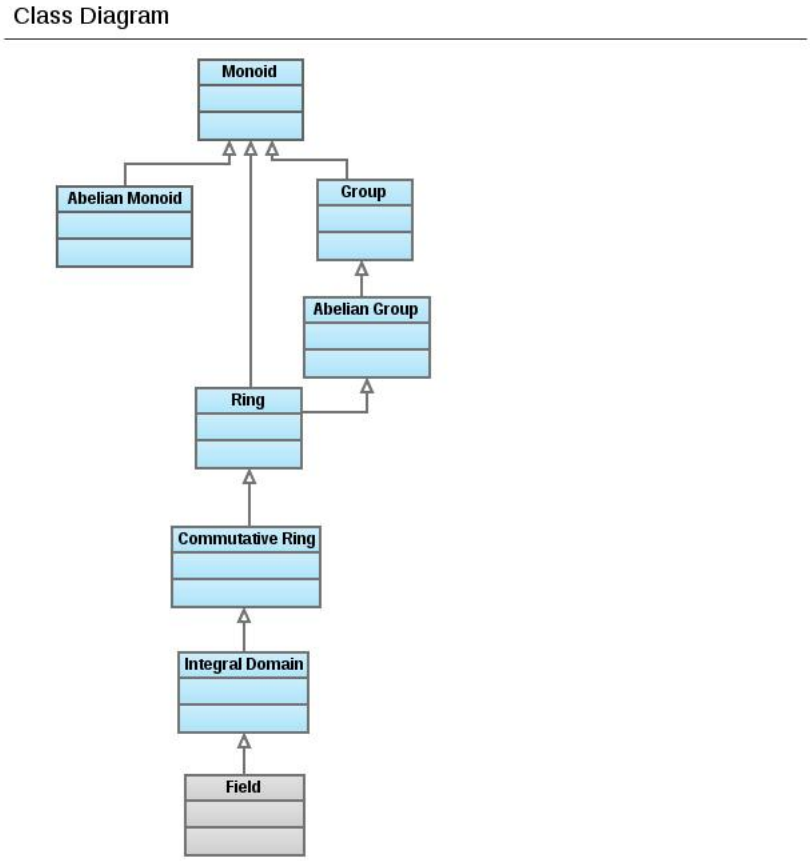
Let's see it:

- The assumptions in **A** are the definition of an additive abelian group.
- The assumptions (2),(3) of **B** are the definition of a multiplicative monoid.
- **C** is one distributive law and if we join all with the assumption (1) of **B** we obtain a commutative ring.
- (3) of **B** also requires $\mathbf{1} \neq \mathbf{0}$ (non-zero).
- Finally with assumption (4) of **B** we will obtain an integral domain (this assumption implies the integral property) and finally the alternative definition.

So using this definition and reusing code by inheritance of the domain structure, we would make a representation of a field in Isabelle/HOL easily. Fortunately, this is already done in the HOL-Algebra library.

```
locale field = "domain" +
  assumes field_Units: "Units R = carrier R - {0}"
```

The following image presents a scheme which tries to clarify the relationships between the implementation of algebraic structures in Isabelle/HOL with locales as it is done in the library of HOL-Algebra. Of course, there are many other ways to create such an infrastructure (for example commutative ring could inherit of abelian monoid and abelian group instead of ring).



Now, as introductory examples of the proofs that can be carried out in the previous algebraic structures, we present the Isabelle proofs corresponding to some of the exercises proposed in Halmos, section 1. Most of them are already done as part of the HOL-Algebra library, so their proofs are almost direct for us. Let R be a field.

For every $x \in R$, $\mathbf{0} + x = x$:

```
lemma (in field) l_zero:
  "x ∈ carrier R ⟹ 0 ⊕ x=x"
  using r_zero [of x]
  using zero_closed
  using a_comm [of x 0] by simp
```

For every x, y and z in R, if $x + y = x + z$ then $y = z$:

```
lemma (in field) a_l_cancel:
  "⟦x ∈ carrier R; y∈ carrier R;z ∈ carrier R⟧ ⟹ (x ⊕ y = x ⊕
z) = (y = z)"
  using a_l_cancel .
```

For every $x, y \in R$, $x + (y - x) = y$ (Here $y - x = y + (-x)$)[1]:

```
lemma (in field) plus_minus_cancel:
  "⟦x ∈ carrier R; y ∈ carrier R⟧ ⟹ x ⊕ (y ⊖ x) = y"
proof -
  assume x_in_R: "x ∈ carrier R"
    and y_in_R: "y∈ carrier R"
  moreover have minus_x_in_R: "⊖ x ∈ carrier R"
    using a_inv_closed [OF x_in_R] .
  have prev_eq: "(x ⊕ ⊖ x) ⊕ 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
```

For every $x \in R$, $x * \mathbf{0} = \mathbf{0} * x = \mathbf{0}$:

```
lemma (in field) r_null:
  "x ∈ carrier R⟹ x ⊗ 0=0"
  using r_null .

lemma (in field) l_null:
  "x ∈ carrier R⟹ 0 ⊗ x=0"
  using l_null .
```

For every $x \in R$, $(-\mathbf{1}) * x = -x$:

[1]Note that in the proof we use the definition of binary $\ominus$, rewriting $x \ominus y$ as $x \oplus (\ominus y)$.

```
lemma (in field) l_minus_one:
  "x∈ carrier R ⟹ (⊖1) ⊗ x = ⊖x"
proof -
  assume x_in_R: "x ∈ carrier R"
  have "(⊖1) ⊗ x = ⊖(1 ⊗ x)"
    using l_minus[OF one_closed x_in_R] .
  also have "...= ⊖x" using l_one[OF x_in_R] by presburger
  finally show ?thesis .
qed
```

For every $x, y \in R$, then $(-x) * (-y) = x * y$:

```
lemma (in field) prod_minus:
  assumes x_in_R: "x ∈ carrier R"
  and y_in_R: "y ∈ carrier R"
  shows "(⊖x) ⊗ (⊖y) = x ⊗ y"
proof -
  have minus_x_in_R: "⊖ x ∈ carrier R" and minus_y_in_R: "⊖ y ∈
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
```

For every $x, y \in R$, if $x * y = \mathbf{0}$ then $x = \mathbf{0}$ or $y = \mathbf{0}$ (or both)[2]:

```
lemma (in field) integral:
  assumes x_y_eq_0: "x ⊗ y = 0"
  and x_in_R: "x ∈ carrier R"
  and y_in_R: "y ∈ carrier R"
  shows "x = 0 | y = 0"
```

[2]Note that this exercise can be solved directly with the integral property (3.0.9). Nevertheless, we can prove it without using that property. This is because the integral property can be proved using that all non-zero elements are invertible (in fact, with this, we can define the notion of *field* without using the integral property. However, we will not change definitions already made in Isabelle/HOL).

proof *(cases "x ≠ 0")*

— We give as a parametrer to 'cases' a boolean ($x \neq 0$); 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 premisse 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 $y \neq 0$ and then we will meet with cases True and False, again.

show *?thesis*

proof *(cases "y ≠ 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 ∈ Units R"*

using *y_in_R*

using *field_Units*

using *y_neq_0* **by** *simp*

have *inv_y_in_R: "inv y ∈ 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 *"***0 = 0** ⊗ *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 *"... = (x ⊗ y) ⊗ inv y"*
— Here we make use of the original premise of the lemma: $x \otimes y = 0$
unfolding *x_y_eq_0 [symmetric]* **..**
also have *"... = x ⊗ (y ⊗ inv y)"*
unfolding *m_assoc [OF x_in_R y_in_R inv_y_in_R]* **..**
also have *"... = x ⊗* **1***"*
unfolding *Units_r_inv [OF y_un]* **..**
also have *"... = 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 *"***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

Chapter 6

Vector spaces

In this chapter we show the implementation of vector spaces. In the following we will assume, if it is not otherwise stated, that $\mathbb{K}$ is a field which plays the role of being the scalar set. We will represent the sum over this field as $+$ and the multiplication as $*$.

Firstly, the definition of a Vector Space (following Halmos):

Definition 6.0.3 *A vector space is a set V of elements called* vectors *satisfying the following axioms:*

A) *To every pair, x and y, of vectors in V there corresponds a vector $x \oplus y$, called the sum of x and y, in such a way that:*

- *(1) addition is commutative, $x \oplus y = y \oplus x$*
- *(2) addition is associative, $x \oplus (y \oplus z) = (x \oplus y) \oplus z$*
- *(3) there exists in V a unique vector $\mathbf{0_V}$ (called the* origin*) such that $x \oplus 0_V = x$ for every scalar x*
- *(4) to every scalar $x \in V$ there corresponds a unique scalar $\ominus x$ such that $x \oplus (\ominus x) = 0_V$*

B) *To every pair, a and x, where a is a scalar ($a \in \mathbb{K}$) and s is a vector in V there corresponds a vector $a \cdot x$ in V, called the product of a and x, in such a way that:*

- *(1) multiplication by scalars is associative, $a \cdot (b \cdot x) = (a * b) \cdot x$*

(2) $1_K \cdot x = x$ *for every vector* x.

C) *(1) Multiplication by scalars is distributive with respect to vector addition:* $a \cdot (x \oplus y) = a \cdot x \oplus a \cdot y$

(2) Multiplication by vectors is distributive with respect to scalars addition: $(a + b) \cdot x = a \cdot x \oplus a \cdot y$

Our objective is to implement it in Isabelle/HOL. We will try to make use, as for as possible, of the available structures in the HOL-Algebra library, since this will permit us to reuse all results already proved in the library for such structures. If we study in detail the above definition and compare it to other algebraic structures, we can realize that **A** axioms tell us that V is an (additive) abelian group. So we have and alternative definition which will be very useful to implement the concept of *vector space* using inheritance.

Definition 6.0.4 *A* vector space V *(over a field* $(\mathbb{K}, +, *)$*) is an additive abelian group* $(V, \oplus_V, 0_V)$ *together with an operation* $\cdot : \mathbb{K} \times V \to V$ *such that the following properties are satisfied. Let* $v, w \in V$ *and* $a, b \in \mathbb{K}$*:*

- $a \cdot (b \cdot v) = (a * b) \cdot v$
- $\mathbf{1} \cdot v = v$
- $a \cdot (v \oplus_V w) = a \cdot v \oplus_V a \cdot w$
- $(a + b) \cdot v = a \cdot v \oplus_V b \cdot v$

A few comments:

Even if there exist two operations called addition (the addition over the field $\mathbb{K}$ and the addition over the abelian group V), we may note that they aren't the same operation because they belong to different structures. For that reason we write the symbol $+$ when we are operating in field $\mathbb{K}$ and the symbol $\oplus_V$ when we are refering to the sum of the abelian group.

Similarly we must remark the difference between the zeros (writing $\mathbf{0}$ for the zero of $\mathbb{K}$ and $\mathbf{0}_V$ for the zero of V). This is very important in order to avoid confusions.

In a vector space, the abelian group V describes the additive structure of the system and the operation $\cdot : \mathbb{K} \times V \to V$ shows us the connection between $\mathbb{K}$ and the abelian group V.

Now we present how we have implemented a vector space in Isabelle/HOL, following locales and using inheritance. 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).

```
locale vector_space = K: field K + V: abelian_group V
  for K (structure) and V (structure) +
  fixes scalar_product:: "'a => 'b => 'b" (infixr "·" 70)
  assumes mult_closed: "⟦x ∈ carrier V;a ∈ carrier K⟧
  ⟹ a · x ∈ carrier V"
  and mult_assoc: "⟦x ∈ carrier V; a ∈ carrier K; b ∈ carrier K⟧
  ⟹ (a ⊗K b) · x = a · (b · x)"
  and mult_1: "⟦x ∈ carrier V⟧ ⟹ 1K · x = x"
  and add_mult_distrib1:
  "⟦x∈ carrier V; y ∈ carrier V; a ∈ carrier K⟧
  ⟹ a · (x ⊕V y)= a·x ⊕V a·y"
  and add_mult_distrib2:
  "⟦x∈ carrier V; a ∈ carrier K; b ∈ carrier K⟧
  ⟹ (a ⊕K b) · x = a·x ⊕V b·x"
```

If we compare this implementation of vector space with the alternative definition that we have presented previously (definition 6.0.4), we would realize that our implementation has been made following literally the alternative definition.

Using this introduction 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 "·" 70)
  assumes field_K: "field K"
  and abelian_group_V: "abelian_group V"
  and mult_closed:
  "⋀x a. ⟦x ∈ carrier V;a ∈ carrier K⟧ ⟹ a·x ∈ carrier V"
  and mult_assoc:
  "⋀x a b. ⟦ x ∈ carrier V; a ∈ carrier K; b ∈ carrier K ⟧
  ⟹ (a ⊗K b)· x = a · (b · x)"
  and mult_1: "⋀x. ⟦x ∈ carrier V⟧ ⟹ 1K · x = x"
```

```
  and add_mult_distrib1:
  "⋀x y a. ⟦x∈ carrier V; y ∈ carrier V; a ∈ carrier K⟧
  ⟹ a·(x ⊕V y)= a·x ⊕V a· y"
  and add_mult_distrib2:
  "⋀x a b. ⟦x∈ carrier V; a ∈ carrier K; b ∈ carrier K⟧
  ⟹ (a ⊕K b)· x = a·x ⊕V b·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
```

There is a concept closely related to the one of vector space, which is the one of module. Even if we are not using it in our development, we present here its definition, following the one presented in Halmos because of its inherent relevance. Actually, modules are a generalization of vector spaces, so we could now think of an alternative definition of vector spaces based on modules.

Definition 6.0.5 *If the scalars are elements of a ring (instead of a field), the generalized concept corresponding to a vector space is called a* module.

Chapter 7

Examples

Generally, when introducing a new concept in a theorem prover, it is worth to provide (and prove) some known examples of such concept, ensuring that the set of objects defined is not empty. This is also common practice in Mathematics. Following Halmos, section 3 is devoted to present some examples of vector spaces, which show the utility of the concept. We have chosen two of these examples to prove the utility of our definition of vector space. The first one corresponds with exercise 6 in Halmos, and it proves that every field $\mathbb{K}$ is a vector space over itself, considering its multiplication as scalar product. We make use of the theorem *vector_spaceI* proved in the previous section:

```
lemma field_is_vector_space:
  assumes field_K: "field K"
  shows "vector_space K K op ⊗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 "⋀x a. ⟦x ∈ carrier K; a ∈ carrier K⟧ ⟹ a ⊗K x ∈
carrier K"
```

```
    using monoid.m_closed [OF field_is_monoid [OF field_K]] by best
next
  show "⋀x a b. ⟦x ∈ carrier K; a ∈ carrier K; b ∈ carrier K⟧
    ⟹ a ⊗K b ⊗K x = a ⊗K (b ⊗K x)"
    using monoid.m_assoc [OF field_is_monoid [OF field_K]] by best
next
  show "⋀x. x ∈ carrier K ⟹ 1K ⊗K x = x"
    using monoid.l_one [OF field_is_monoid [OF field_K]] by best
next
  show "⋀x y a. ⟦x ∈ carrier K; y ∈ carrier K; a ∈ carrier K⟧
    ⟹ a ⊗K (x ⊕K y) = a ⊗K x ⊕K a ⊗K y"
    using ring.r_distr [OF field_is_ring [OF field_K]] by best
next
  show "⋀x a b. ⟦x ∈ carrier K; a ∈ carrier K; b ∈ carrier K⟧
    ⟹ (a ⊕K b) ⊗K x = a ⊗K x ⊕K b ⊗K x"
  proof -
    fix x and a and b
    assume x_in_K: "x ∈ carrier K"
      and a_in_K: "a∈ carrier K" and b_in_K:"b∈ carrier K"
    show "(a ⊕K b) ⊗K x = a ⊗K x ⊕K b ⊗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)
```

There is another relevant example in our development, which proves that $\mathbb{K}^n$, with $n \in \mathbb{N}$, is also a vector space with an adequate scalar product. We will present this example later (in chapter 13) when proving that every vector space of dimension n is isomorphic to $\mathbb{K}^n$.

Chapter 8

Comments

In this chapter we will make the proofs of some properties of vector spaces, specifically the first exercise proposed by Halmos in section 4. Most properties are basic and we will need to use them in the future, so this is an important section. Due to that, we will prove properties of vector spaces which don't appear in Halmos that will be very useful to us.

Halmos proposes some exercises, but most of them are properties already proved in abelian groups, rings... so they are in the Isabelle 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: "0_V ∈ carrier V"
using V.zero_closed .

lemma zeroK_closed: "0_K ∈ carrier K"
  using K.zero_closed .
```

A variation of *r_neg* (*x ∈ carrier V ⟹ x ⊕_V ⊖_V x = 0_V*):

```
lemma r_neg':
  assumes x_in_V: "x ∈ carrier V"
  shows "x ⊖_V x=0_V"
proof -
```

```
  have "0_V = x ⊕_V ⊖_V x"
    using V.r_neg [OF x_in_V, symmetric] .
  also have "...=x ⊖_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 ∈ carrier K "
  shows "a · 0_V ⊕_V a · 0_V = a · 0_V"
proof -
  have "a· 0_V =a· (0_V ⊕_V 0_V)"
    using V.r_zero [symmetric, OF V.zero_closed] by simp
  also
  have "...=a· 0 _V ⊕ _V a· 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 ∈ carrier V"
  and y_in_V: "y ∈ carrier V" and z_in_V: "z ∈ carrier V"
  shows "x ⊕_V y ⊖_V z = x ⊕_V (y ⊖_V z)"
proof -
  have minus_z_in_V:"⊖_V z ∈ carrier V"
    using V.a_inv_closed [OF z_in_V] .
  have "x ⊕_V y ⊖_V z = x ⊕_V y ⊕_V ⊖_V z"
    using a_minus_def [of "x ⊕_V y", OF _ z_in_V]
    using V.a_closed [OF x_in_V y_in_V] .
  also have "x ⊕_V y ⊕_V ⊖_V z = x ⊕_V (y ⊕_V ⊖_V z)"
    using V.a_assoc [OF x_in_V y_in_V minus_z_in_V] .
  also have "... = x ⊕_V (y ⊖_V z)"
    unfolding a_minus_def [symmetric, OF y_in_V z_in_V] ..
  finally show ?thesis by simp
qed
```

Now we can complete the 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 ∈ carrier K"
  shows "a · 0_V = 0_V"
proof -
  have mclosed: "a· 0 _V  ∈ carrier V"
    using mult_closed [OF V.zero_closed a_in_K] .
  have "a · 0_V = a· 0_V ⊕_V a· 0_V"
    using mult_zero_descomposition [OF a_in_K] by simp
  hence "a · 0_V ⊖_V a · 0_V = a· 0_V ⊕_V a· 0_V ⊖_V a · 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 ∈ carrier V"
  shows "0_K · x ⊕_V 0_K · x = 0_K · x"
proof -
  have "0_K · x = (0_K ⊕_K 0_K)· 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 ∈ carrier V"
  shows "0_K · x = 0_V"
proof -
  have mclosed: "0_K · x ∈ carrier V"
    using mult_closed [OF x_in_V zeroK_closed] .
  have "0_K · x = 0_K · x ⊕_V 0_K · x"
```

```
    using mult_zero_descomposition2 [OF x_in_V,symmetric] .
  hence "0_K · x ⊖_V 0_K · x = 0_K · x ⊕_V 0_K · x ⊖_V 0_K · 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 ∈ carrier V"
  shows "(⊖_K 1_K) · x = ⊖_V x"
proof (rule V.minus_equality [symmetric, of "(⊖_K 1_K) · x" x])
  show "x ∈ carrier V" using x_in_V .
  have minus_oneK_closed: "⊖ _K 1 _K ∈ carrier K"
    using K.a_inv_closed [OF K.one_closed] .
  show "⊖ 1 · x ∈ carrier V"
    using mult_closed [OF x_in_V minus_oneK_closed] .
  show "⊖1 · x ⊕_V x = 0_V"
  proof -
    have "0_V = 0_K · x"
      using zeroK_mult_V_is_zeroV [symmetric, OF x_in_V] .
    also have "... = (⊖_K 1_K ⊕_K 1_K)· x"
      unfolding K.l_neg [OF K.one_closed ] ..
    also have "... = ⊖_K 1 _K· x ⊕_V 1 _K · x"
      using add_mult_distrib2 [OF x_in_V minus_oneK_closed
K.one_closed] .
    also have "... = ⊖_K 1 _K· x ⊕_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 ∈ carrier V"
```

```
  and a_in_K: "a ∈ carrier K"
  shows "(⊖K a) · x = ⊖V (a·x)"
proof(rule V.minus_equality [symmetric, of "(⊖K  a)· x" "a·x"])
  show "a·x ∈ carrier V" using mult_closed[OF x_in_V a_in_K] .
  show "⊖ a · x ∈ carrier V"
    using mult_closed [OF x_in_V K.a_inv_closed[OF a_in_K]] .
  show "⊖a·x ⊕V a·x = 0V"
  proof -
    have "0V = 0K · x"
      using zeroK_mult_V_is_zeroV [symmetric, OF x_in_V] .
    also have "... = (⊖K a⊕K a)· x"
      unfolding K.l_neg [OF a_in_K ] ..
    also have "... = ⊖K a· x ⊕V a · 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 ∈ carrier V" and x_not_zero: "x ≠ 0V"
  and a_in_K: "a ∈ carrier K" and m_ax_0: "a · x = 0V"
  shows "a = 0K"
proof (rule classical)
  assume a_not_zero: "a ≠ 0K"
  have a_un: "a ∈ Units K"
    using a_not_zero
    using a_in_K
    using K.field_Units by simp
  have inv_a_in_K: "inv a ∈ carrier K"
    using K.Units_inv_closed [OF a_un] .
  have "x = (inv a ⊗ a)· x"
    using K.Units_l_inv [OF a_un]
    using mult_1 [OF x_in_V]
    by simp
  also have "...= inv a · (a · x)"
    using mult_assoc [OF x_in_V inv_a_in_K a_in_K] .
  also have "...= inv a · 0V" using m_ax_0 by simp
```

```
  also have "... = 0_V"
    using scalar_mult_zeroV_is_zeroV [OF inv_a_in_K] .
  finally have "x = 0_V" .
  with x_not_zero show "a=0_K" by contradiction
qed


lemma integral:
  assumes x_in_V: "x∈ carrier V"
  and a_in_K: "a ∈ carrier K"
  and m_ax_0: "a · x= 0_V"
  shows "a = 0_K | x=0_V"
proof (cases "x ≠ 0_V")
  case False show ?thesis using False by simp
next
  case True
  note x_not_zero = True
  show ?thesis
  proof (cases "a ≠ 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∈ carrier V"
  and y_in_V: "y ∈ carrier V"
  and a_in_K: "a ∈ carrier K"
  shows "a · (x ⊖_V y) = a · x ⊖_V a · y"
proof -
  have minus_y_in_V: "⊖_V y ∈ carrier V"
    using V.a_inv_closed [OF y_in_V] .
```

```
  have minus_one_in_K: "⊖K 1 ∈ carrier K"
    using K.a_inv_closed[OF K.one_closed] .
  have mclosed: "a · y ∈ carrier V"
    using mult_closed [OF y_in_V a_in_K] .
  have mclosed2: "a · x ∈ carrier V"
    using mult_closed [OF x_in_V a_in_K] .
  have "a · (x ⊖V y)=a · (x ⊕V  ⊖V y)"
    using a_minus_def[OF x_in_V y_in_V] by simp
  also have "...= a · x ⊕ V a · (⊖V y)"
    using add_mult_distrib1 [OF x_in_V minus_y_in_V a_in_K] .
  also have "...= a · x ⊕ V a · (⊖K 1K · y)"
    using negate_eq [OF y_in_V] by simp
  also have "...= a · x ⊕ V (a ⊗K (⊖K 1K))· y"
    using mult_assoc [OF y_in_V a_in_K minus_one_in_K ,symmetric]
    by simp
  also have "...= a · x ⊕ V ((⊖K 1K)⊗K a)· y"
    using K.m_comm [OF minus_one_in_K a_in_K] by simp
  also have "...= a · x ⊕ V (⊖K 1K) · a · y"
    using mult_assoc [OF y_in_V minus_one_in_K a_in_K] by simp
  also have "...= a · x ⊕ V ⊖V (a · y)"
    using negate_eq [OF mclosed] by simp
  also have "...= a · x ⊖V a· 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∈ carrier V"
  and a_in_K: "a ∈ carrier K"
  and b_in_K: "b ∈ carrier K"
  shows "(a ⊖K b) · x = a·x ⊖V b·x"
proof -
  have minus_b_in_K: "⊖K b ∈ carrier K"
    using K.a_inv_closed [OF b_in_K] .
  have bx_in_V: "b·x ∈  carrier V"
    using mult_closed [OF x_in_V b_in_K] .
  have "(a ⊖K b) · x=(a ⊕K ⊖K b)·x "
    using K.minus_eq [OF a_in_K b_in_K] by simp
```

```
  also have "...=a·x ⊕V (⊖K b)·x"
    using add_mult_distrib2 [OF x_in_V a_in_K minus_b_in_K] .
  also have "...=a·x ⊕V (⊖K (1K ⊗K b))·x"
    using K.l_one [OF b_in_K] by simp
  also have "...=a·x ⊕V (⊖K 1K ⊗K b)·x"
    using K.l_minus [OF K.one_closed b_in_K,symmetric] by simp
  also have "...=a·x ⊕V (⊖K 1K)·b·x"
    using mult_assoc [OF x_in_V K.a_inv_closed[OF K.one_closed]
b_in_K]
    by simp
  also have "...=a·x  ⊕V  ⊖V (b·x)"
    using negate_eq [OF bx_in_V] by simp
  also have "...=a·x ⊖V b·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 ∈ carrier V" and a_in_K:"a∈ carrier K"
  shows "(⊖K a) · (⊖V x) = a · x"
proof -
  have "(⊖Ka) · (⊖Vx) = (⊖Ka ⊗ (⊖K1K)) · 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 "...=(a⊗1) · 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 ∈ carrier V"
  and a_in_K: "a∈ carrier K"
  and b_in_K:"b∈ carrier K"
```

```
  shows "a · b · x = b · a · x"
proof -
  have "a·b·x=(a⊗b)·x"
    using mult_assoc[OF x_in_V a_in_K b_in_K, symmetric] .
  also have "...=(b⊗a)·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 ∈ carrier V"
  and y_in_V: "y∈carrier V"
  and a_in_K: "a∈carrier K"
  and a_not_zero: "a≠0_K"
  shows "(a · x = a · y) = (x = y)"
proof
  assume ax_ay:"a·x=a·y"
  have a_in_Units: "a ∈ Units K"
    using K.field_Units and a_in_K and a_not_zero by simp
  have "x=1_K · x" using mult_1[OF x_in_V, symmetric] .
  also have "...=((inv a)⊗_K a)·x"
    using K.Units_l_inv [OF a_in_Units] by simp
  also have "...=(inv a)· a·x"
    using mult_assoc[OF x_in_V
      K.Units_inv_closed[OF a_in_Units] a_in_K]
    by simp
  also have "...=(inv a)· a·y" using ax_ay by simp
  also have "...=((inv a)⊗_K a)·y"
    using mult_assoc[OF y_in_V K.Units_inv_closed
      [OF a_in_Units] a_in_K] by simp
  also have "...=1_K · 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·x=a·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 ∈ carrier V"
  and a_in_K: "a ∈ carrier K"
  and b_in_K: "b∈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
```

Most of the previous results have been completed by calculational reasoning, as presented in the Isabelle lemma "integral" (see chapter 5). The idea is, in order to prove an equality, one starts from its left hand side, and then by applying rewriting rules, one reaches the right hand side. Then, by summarizing the chain of equalities, the original equality is obtained. Therefore, we can conclude that most of the proofs that we have completed up to

now are based on rewriting rules, case distinction (with the rule "cases" and some boolean condition), and *reductio ad absurdum*. For more information in calculational reasoning in Isabelle, see [36].

Chapter 9

Linear dependence

In this chapter we will present the notion of linearly dependent and independent set. Halmos defines them next way:

Definition 9.0.6 *A finite set* $\{x_i\}_{i\in\mathbb{N}}$ *of vectors* ($x_i \in V$) *is* linearly dependent *if there exists a corresponding set* $\{\alpha_i\}_{i\in\mathbb{N}}$ *of scalars in* $\mathbb{K}$*, not all zero, such that*

$$\sum_{i\in\mathbb{N}} \alpha_i \cdot x_i = 0$$

Definition 9.0.7 *If, on the other hand,* $\sum_{i\in\mathbb{N}} \alpha_i \cdot x_i = 0$ *implies that* $\alpha_i = 0$ *for each* i*, the set* $\{x_i\}_{i\in\mathbb{N}}$ *is* linearly independent.

We are formalizing both concepts in Isabelle/HOL. The definitions are also generalized in Halmos to the case where the set of indexes is infinite. We will also implement such generalizations.

Due to the definitions, first of all we will introduce the definition of *linear combination*, even if it appears in Halmos later.

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 ∈ X → carrier K ∧ (∀x. x∉X ⟶ f x = 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 ∈ X → 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 (`∀x. x ∉ X ⟶ f x = 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) = (∀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 ∈ carrier V"
  and cf_f: "f ∈ coefficients_function (carrier V)"
  shows "f(x) ∈ carrier K"
  using assms unfolding coefficients_function_def by auto
```

For every `x ∈ carrier V`, multiplication between the scalar and the vector (`f x · x`) is in `carrier V`.

```
lemma fx_x_in_V:
  assumes x_in_V: "x ∈ carrier V"
  and cf_f: "f ∈ coefficients_function (carrier V)"
  shows "f(x)·x ∈ 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 ⇒ 'a) ⇒ 'b set ⇒ 'b"
  where "linear_combination f X = finsum V (λy. f(y)·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 ∧ X ⊆ 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 ⊆ 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 ⇒ bool"*
  where *"linear_dependent X = (good_set X*
  ∧ (∃f. f∈ coefficients_function (carrier V) ∧
linear_combination f X = $\mathbf{0}_V$
  ∧ ¬(∀x ∈ X. f x = $\mathbf{0}_K$)))"

This definition is equivalent to the previous one:

definition *linear_dependent_2 :: "'b set ⇒ bool"*
  where *"linear_dependent_2 X =*
  (∃f. f ∈ coefficients_function (carrier V) ∧ good_set X
  ∧ linear_combination f X = $\mathbf{0}_V$ ∧ ¬ (∀x ∈ X. f x = $\mathbf{0}_K$))"

Here we present the equivalence of the definitions:
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)=$\mathbf{0}_K$)))"

Abusing of the notation, we will sometimes use "dependent sets" and "independent sets" to refer us to "linearly independent sets" and "linearly dependent sets" respectively.

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 = 0_V"
    and not_all_zero: "¬(∀x ∈ X. f(x)=0_K)"
    unfolding linear_independent_def using good_set by best
  have " f ∈ coefficients_function (carrier V)
    ∧ linear_combination f X = 0_V ∧ ¬ (∀x∈X. f x = 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 = 0_V ∧ ¬ (∀x∈X. f x = 0)"
    by (rule exI [of _ "f"])
  thus "good_set X ∧ (∃f. f ∈ coefficients_function (carrier V)
    ∧ linear_combination f X = 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 = 0_V "
      and not_all_zero: " ¬(∀x ∈ X. f x = 0_K)"
      unfolding linear_dependent_def by auto
    have "¬ (∀f. f ∈ coefficients_function (carrier V)
      ∧ linear_combination f X = 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 = 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 $[\![\neg\ P;\ \neg\ R \Longrightarrow P]\!] \Longrightarrow R$ 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 ⟹ ¬ linear_dependent X"
proof -
  assume li: "linear_independent X"
  have imp: "linear_dependent X ⟹ ¬ linear_independent X"
```

```
    using dependent_implies_not_independent .
  show "¬ 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 ⟷ ¬ 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 ⇒ bool ⇒ bool"
  where "xor A B ≡ (A ∧ ¬ B) ∨ (¬A ∧ 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 "¬ 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≠{}"
 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 ⊆ 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 = 0_V"
```

and *not_all_zero_f:" ¬ (∀x∈Y. f x = 0)"*
and *coefficients_function_f:*
"f ∈ coefficients_function (carrier V) "
unfolding *linear_dependent_def*
by *best*

— Now we define the function and prove that is a coefficients function:

let *?g= "(λx. if x ∈ Y then f(x) else* $\mathbf{0}_K$*)"*
have *coefficients_function_g:*
"?g ∈ 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 ∩ B = {}; g ∈ A → carrier V; g ∈ B → carrier V*⟧ ⟹ *finsum V g (A ∪ B) = finsum V g A* $\oplus_V$ *finsum V g B* and that the decomposition of the sets is disjoint.

— Some properties which we will need for the proof:

have *descomposicion_conjuntos:"X=Y∪(X-Y)"*
using *Y_subset_X* **by** *auto*
have *disjuntos: "Y ∩ (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 ∈ Y → carrier K"*
...
have *g2:"?g ∈ (X-Y) → carrier K"*
...

let *?h="(λx. ?g(x)·x)"*
have *h1: "?h ∈ Y → carrier V"*
...
have *h2: "?h ∈ (X-Y) → carrier V"*
...

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

have *"linear_combination ?g X*
= linear_combination ?g (Y∪(X-Y))"
using *descomposicion_conjuntos* **by** *simp*
also have *descomposicion:*
"...=linear_combination ?g Y ⊕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 *"...=0V ⊕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 *"(⨁V y∈Y. (if y ∈ Y then f y else 0) · y)*
= (⨁V y∈Y. f y · y)"
using *finsum_cong [OF iguales]* **using** *h1* **by** *auto*
qed
also have *"...=0V"* **using** *sum_zero_f_Y* **.**
finally show *?thesis* **by** *simp*
qed
also have *"...=0V ⊕V 0V"*
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 ∈ carrier V ⟹ 0 · x = 0V*). Then we are adding a finite sum of zeros, so it will be zero using *finsum_zero*: *finite A ⟹ (⨁V i∈A. 0V) = 0V*.

have *sum_g_X_minus_Y:"linear_combination ?g (X-Y)=0V"*
proof -

```
      have X_subset_V: "X ⊆ carrier V"
        using good_set
        unfolding good_set_def by auto
      hence X_minus_Y_subset_V:"(X-Y) ⊆ carrier V"  by auto
      have not_in_Y: "x∈ (X-Y)⟹ x∉ Y" by auto
      have "linear_combination ?g (X-Y)=(⨁V y∈X - Y.  𝟬 · y)"
      proof (unfold linear_combination_def)
        have igualesX_minus_Y: "X-Y=X-Y"..
        show "(⨁V y∈X - Y. (if y ∈ Y then f y else 𝟬) · y)
          = finsum V (op · 𝟬) (X - Y)"
          using finsum_cong [OF igualesX_minus_Y eqTrueI [OF h2]]
          by auto
      qed
      also have "...=(⨁V y∈X - Y. 𝟬V)"
      proof (rule finsum_cong')
        show "X - Y = X - Y" ..
        show "(λy. 𝟬V) ∈ X - Y → carrier V" by simp
        show "⋀i. i ∈ X - Y ⟹ 𝟬 · i = 𝟬V"
          using zeroK_mult_V_is_zeroV
          using X_minus_Y_subset_V by auto
      qed
      also have "...=𝟬V"
        using finsum_zero [OF finite_X_minus_Y] .
      finally show ?thesis .
    qed
    thus ?thesis by simp
  qed
  also have "...=𝟬V" by simp
  finally show ?thesis .
qed
    — Second property is easy:
have not_all_zero_g: "¬ (∀x∈X. ?g x = 𝟬)"
  using Y_subset_X
  using not_all_zero_f by auto
have "?g ∈ coefficients_function (carrier V)
  ∧ linear_combination ?g X = 𝟬V ∧ ¬ (∀x∈X. ?g x = 𝟬)"
  using coefficients_function_g and good_set
    and sum_zero_g_X and not_all_zero_g by fast
hence
  "∃f. f ∈ coefficients_function (carrier V)
```

```
    ∧ linear_combination f X = 0_V ∧ ¬ (∀x∈X. f x = 0)"
    by (rule exI[of _ ?g])
  thus "good_set X ∧ (∃f. f ∈ coefficients_function (carrier V)
    ∧ linear_combination f X = 0_V ∧ ¬ (∀x∈X. f x = 0))"
    using good_set by simp
qed
```

It may be worth noting that the proof, which closely follows the mathematical sketch of the proof, consumes space mainly to prove that the coefficients function satisfies the right properties.

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 "0_V ∉ A"
proof (cases "0_V ∉ 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: "0_V ∈ A" using False by simp
    let ?g="(λx. if x=0_V then 1_K else 0_K)"
    have cf_g: "?g ∈ coefficients_function (carrier V)"
      unfolding coefficients_function_def by auto
    have lc_zero: "linear_combination ?g A=0_V"
    proof (unfold linear_combination_def)
      have "(⨁_V y∈A. (if y = 0_V then 1 else 0) · y)
        =(⨁_V y∈A. 0_V)"
      proof (rule finsum_cong',auto)
        show "1 · 0_V = 0_V"
          using scalar_mult_zeroV_is_zeroV by auto
        fix i
        assume i_in_A: "i ∈ A" and i_not_zero: "i ≠ 0_V"
        show "0 · i = 0_V"
          using zeroK_mult_V_is_zeroV and i_in_A and cb_A
          unfolding good_set_def by auto
      qed
      also have "...=0_V"
        using finsum_zero using good_set_finite[OF cb_A] by auto
      finally show
```

```
      "(⨁V y∈A. (if y = 0V then 1 else 0) · y) = 0V" .
  qed
  have not_all_zero: "¬(∀x∈A. ?g x = 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* (the tool for automatic discovery of proofs).

```
lemma independent_set_implies_independent_subset:
  assumes A_in_B: "A ⊆ 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 ⇒ bool"
  where "linear_independent_ext X
  = (∀A. finite A ∧ A ⊆ X ⟶ linear_independent A)"
```

Otherwise, it is linearly dependent.

```
definition linear_dependent_ext:: "'b set ⇒ bool"
  where "linear_dependent_ext X
  = (∃A. A ⊆ X ∧ 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 ⊆ 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 ⊆ 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 ⊆ 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
```

Chapter 10

Linear combinations

10.1 Sets indexation

Here we present the notion of an indexed set in Isabelle. This is one of the biggest problems that we have found in our development. Following Halmos, we realize that there are some proofs where he expresses a set in a determinate order (for example, $A = \{a_1, \ldots, a_n\}$) and he uses such orders to build proofs upon (as we will see). However, a set has not an order by default and of course, it is not implemented in Isabelle/HOL.

We have, at least, two different options:

- Try to make the proofs in other way, looking for alternative proofs of the theorems in which the order of the sets is not used.
- Follow exactly Halmos and for that we would need to implement the structure of an indexed set (a set with an order).

Firstly we tried to follow the first option: work without indexations because they are not inherent to sets and thus they require an additional effort of implementation that we tried to avoid. It is also remarkable that Halmos never mentions explicitly the need or relevance of having explicit orders for sets. A finite set has not any order (although we can be able to assign one for it). However, we will realize that the order is indispensable for the proofs of the book (as we will see later).

So finally, we decided to implement the notion of indexed sets. The main

advantage is that once we have done it, we will be able to follow the proofs of the book literally and we will not have to look for alternative proofs, think how we can express a lemma without taking the order into account, ...

To implement indexed sets we have also several alternatives, for example:

- Implement the order as a bijection between the set of naturals smaller than the cardinal of the set and the elements of the set. Each natural will be assigned to an element and that natural represents the position of the element in the set.
- Use lists, where the position of each element represents its index in the set.
- Use tuples or relations of one element and one natural.

We decided to use the first option due to its advantages: indexed sets are isomorphic to sorted canonical lists (we can define an isomorphism mapping $\{(a,1),(b,2),(c,3)\}$ to the list $[a,b,c]$) but operations over sorted canonical lists should preserve canonicity, which makes operations such as "insert" or "remove" in given positions more complicated than their equivalent operations over sets with an indexing function. For example, if we want to add one element to a list in a concrete position we have to do lot of things: create another list with the elements of the first one until the position in which we want to add the element, after that we have to add the element and then continue interating the first list adding the rest of the elements to the second one... With indexed sets we only have to make a function definition (this will be the function *insert_iset*). Nevertheless, lists have a great advantage: with them we could execute our algorithms. Using tuples or relations is similar to the first option.

We implement an indexation of a set A like a tuple of the set A and a function $f\colon \mathbb{N} \to V$ which is bijective between the naturals up to the number "card A - 1" and A. We have to give to the function a number n (n is a natural number between 0 and $card(A)-1$) and the function will return us the element in the nth position in the set A (with respect to the given order). In other words: $A = f'\{.. < card(A)\}$. [1] Note that is correct: between 0 and $card(A)-1$ (both included) there are card A elements.

[1] The case that A is empty is also included: $\{\} = A = f'(\{.. < card(A)\} = f'\{.. < card(\{\})\} = f'\{.. < 0\} = f'\{\} = \{\}$.

To sum up, in Halmos (and in any book) a finite set A is represented like follows:

$$A = \{a_1, \ldots, a_n\} \text{ where } n = card(A)$$

and with our definition, we are representing it similarly[2]:

$$A = \{a_0, \ldots, a_n\} \text{ where } n = card(A) - 1$$

Now we can see how we have implemented an indexed set in Isabelle/HOL.

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 Isabelle definition of *card* assigns to an infinite set cardinality equal to *0*.

First of all, we present the definitions of bijection and injectivity in Isabelle. A function $f : A \to B$ is bijective if is injective and surjective. f will be injective if whenever the images of two points in A are equal, then the points are the same. f is surjective if $f`A = B$.

```
inj_on f A = (∀x∈A. ∀y∈A. f x = f y ⟶ x = y)

bij_betw f A B = (inj_on f A ∧ f ` A = B)
```

[2]Really, we could have made equal to Halmos, but then the notation in Isabelle would be more difficult to be managed in the proofs.

Finally, we present the definition of indexing using previous concepts:

```
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 -
  ...
  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
```

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 ∈ A"
  using assms unfolding indexing_def bij_betw_def by auto
```

We present two auxiliary lemmas about indexings and their behaviour 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 ∈ {..<card A}" and y: "y ∈ {..<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 ∈ {..<card A}" and y: "y ∈ {..<card A}"
  and f: "x ≠ y"
  shows "f x ≠ 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 proof is not 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 "(∃n::nat. ∃f∈{i. i < n} → A.
  ((f ` {i. i < n} = A) ∧ inj_on f {i. i < n}))"
proof -
  obtain f and n
    where a1: "f ` {i. i < (n::nat)} = A ∧ inj_on f {i. i < n}"
    and a2: "f ∈ {i. i < n} → 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 "(∃f ∈ {i. i < (card A)} → A.
  ((f ‘ {i. i < (card A)} = A)
  ∧ inj_on f {i. i < (card A)}))"
  using f proof (induct)
  case empty
  show ?case by auto
next
  case (insert b B)
  show "∃f∈{i::nat. i <
    card (insert b B)} → insert b B.
    f ‘ {i::nat. i < card (insert b B)} = insert b B ∧
    inj_on f {i::nat. i < card (insert b B)}"
  proof -
    obtain g
      where g1: "g ∈ {i. i < (card B)} → B"
      and g2: "g ‘ {i::nat. i < card B} = B ∧ inj_on g
      {i::nat. i < card B}"
      using insert.hyps (3) by auto
    let ?f = "(λn::nat. if n ∈ {i. i < card B} then g n
      else if n = card B then b else g n)"
    have f1: "?f ∈{i::nat. i < card (insert b B)}
      → insert b B"
    proof
      ...
    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
      ...
```

```
    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 "∃f. indexing (A,f)"
proof (unfold indexing_def,unfold bij_betw_def,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 work with 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 should 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)}"])
      ...
    qed
```

```
    next
      show "indexing_ext F x n xa = indexing_ext F x n y"
      proof (cases "xa < n")
        ...
      qed
    qed
  qed
  show "?h ‘ {..<card ?F} = ?F"
  proof -
    ...
  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 ∉ 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)
    ...
  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
    ...
  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},
  (λk. if k < n then (snd A) k  else (snd A) (Suc k)))"
```

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))}"
  ...
  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)"
  ...
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 ∉ A"
  and n_l_c: "n < card A"
  shows "remove_iset (insert_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⟧ ⟹ A ∧ B ∧ C ∧ 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 ∉ X"
  and f_buena:"f ∈ {i. i < (card X)} → X" and ordenFX: "f ‘ {i. i
< (card X)} = X"
  shows  "∃h. (h ∈ {i. i < (card (insert x X))} → (insert x X)
  ∧ h‘{i. i < (card (insert x X))} = (insert x X)
  ∧ h (card X) = x ∧ (∀i. i<card(X) ⟶ 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 ∈ {i. i < card (insert x X)} → 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 "(∀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∩B={}"
  and finite_A: "finite A"
  and A_not_empty: "A≠{}" — If not the result is trivial.
  and finite_B: "finite B"
  shows "∃h. indexing (A∪B,h) ∧ h‘ {..<card(A)}= A
  ∧ h‘ ({..<(card(A)+card(B))}-{..<card(A)})=B"
proof -
  have "∃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)))"])
    ...
    show "indexing(A ∪ B,?h) ∧
      ?h `{..<card A} = A ∧
      ?h `({..<card A + card B} - {..<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 ∈ A`, since functions are total in HOL, and without the premise `a ∈ 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 < 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 ⇒'c iset ⇒ nat"
  where "obtain_position a A = (THE n. (snd A) n = a
  ∧ n < card (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_and_less_card_obtain_position:
  assumes  a_in_A: "a ∈ A"
  and indexing_A: "indexing (A,f)"
  shows "∃n::nat. f n = a ∧ n < (card A)"
proof -
  have "A≠{}" using a_in_A by blast
  hence cardA_g_0: "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 "
  ...
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 ∈ A"
  and indexing_A: "indexing (A,f)"
  shows "f (obtain_position a (A,f)) = 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 "f n = a" unfolding n_def
    by (metis ex_THE)
  thus "f (THE n. snd (A, f) n = a ∧ 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 ∈ A"
  and indexing_A: "indexing (A,f)"
  shows "a ∉ 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 an important theorem 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.

This induction rule is similar to the proper of finite sets, ⟦*finite F; P {};* ⋀*x F.* ⟦*finite F; x ∉ F; P F*⟧ ⟹ *P (insert x F)*⟧ ⟹ *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 know 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
```

10.2 Linear combinations

The notion of linear combination has been implementend in the previous chapter in order to define the concept of linear dependence and independence.

Nevertheless, we will prove some properties of linear combinations in this chapter, but firstly we must say that the main objective of this chapter is to prove following theorem labelled as Theorem 1 in Halmos, section 6:

Theorem 10.2.1 *The set of non-zero vectors $x_1, \ldots, x_n$ is linearly dependent if and only if some x_k, $2 \leq k \leq n$, is a linear combination of the preceding ones.*

At this point, we realized that we had to solve a big problem: Halmos is assuming the existence of an inherent order to a set, but this is not implemented so in Isabelle. We had to implement something to define the order of a set or look for an equivalent or similar theorem, not using orders. The next theorem looks like similar and it doesn't need the order of a set, but it is not useful for the proof of theorem 10.2.1 because we are not proving that there exists one element which is a linear combination of the preceding ones, but that there exists an element which is a linear combination of the rest elements of the set (nevertheless, we will also prove it in our developement because we will use it in the future in order to prove some properties in chapter 11).

Theorem 10.2.2 *If the finite set X of non-zero vectors is linearly dependent then there exists an element $x \in X$ such that x is a linear combination of $X - \{x\}$.*

However, this was not our unique attempt. We were stuck for a time thinking how we could prove the theorem and future theorems which use it (specially, how to complete an independent set up to a basis).

We did three attempts more to demonstrate it before defining the indexed sets in Isabelle/HOL. It took up several code lines which finally was not useful and we count with ideas of Julio Rubio and Tobias Niphow. We had to reject them because we found that we were defining functions which were not commutative[3], we lost the unicity of the elements, the proofs were getting difficult because we had to move to the quotient space [16] ... Finally we followed the development of the book, although it involved a great deal of work and made a sudden stop to generate the theory of indexed sets.

As we pretended to follow Halmos and this is a very important theorem which will be used by him in future proofs, we realized the relevance of having made the definition of indexing and indexed sets.

During the developement, we thought that a good alternative proof for theorem 10.2.1 (theorem 1 in section 6 in Halmos) could be the next one. It seems easier (to implement) and shorter:

- **Case empty:** If $A = \{\}$ we have a contradiction, at the same time we obtain that A is linearly dependent and independent.
- **Case insert:** Suppose that it is true for $A = \{f(1), \ldots, f(m)\}$, so there exists one element that is a linear combination of the preceding ones and let be k the position of that element (logically k is between 1 and m). We have to prove that if we insert an element in this set, the result is also true. Suppose that we insert a new element a in the position $1 \leq n \leq m = card(A)$, so the new set is

$$\text{(insert a A)} = \{f(1), \ldots, f(n-1), \underbrace{a}_{nth\ position}, f(n), \ldots, f(m)\}$$

[3] For instance, we defined the following function in order to prove theorem 11.2.1: *f x A = (if linear_independent (insert x A) then (insert x A) else A)* but we realized that it wasn't left-commutative: *f x (f y Z)* $\neq$ *f y (f x Z)*. For example, if we take $x = (0,1)$, $y = (1,1)$ and $Z = \{(1,0)\}$ we have that:

f x (f y Z) = {(1,0),(1,1)} $\neq$ *{(0,1),(1,1)}= f y (f x Z)*

The problem was that the result (the set obtained) depends on the order that we apply the function.

Note that we have now a set of $m+1$ elements where a is in the position n (and the element which was in the position n of A is now in position $n+1$ of the new set). We will obtain the result depending on the position (denoted by k) of the element which is a linear combination of the preceding ones: the element a.

- If $k < n$, the proof of the result will be more or less easy: the element which will be a linear combination of the preceding ones is the same. Example, let be:

  $$A = \{f(1), \ldots, \underbrace{f(k)}, \ldots, f(n), \ldots, f(m)\}$$

  Where $f(k)$ is the element which is a linear combination of the preceding ones. Hence, if we insert the element a in a position n after k, we will have:

  $$\text{(insert a A)} = \{f(1), \ldots, \underbrace{f(k)}, \ldots, f(n-1), \underbrace{a}_{nth\ position}, f(n), \ldots, f(m)\}$$

  Logically, $f(k)$ is also the element which will be a linear combination of the preceding ones and it will be also in the position k of the set obtained inserting a in A.
- If $k = n$ the case is similar: We have that $f(k)$ is a linear combination of the preceding elements of A.

  $$A = \{f(1), \ldots, f(k-1), \underbrace{f(k)}, f(k+1), \ldots, f(m)\}$$

  If we insert a exactly in the position k:

  $$\text{(insert a A)} = \{f(1), \ldots, f(k-1), \overbrace{a}^{n\ position}, \underbrace{f(k)}, f(k+1), \ldots, f(m)\}$$

  Then the element $f(k)$ will be now in the position $k+1$ and it will be also a combination of the preceding ones (if $f(k)$ was a linear combination of $f(1), \ldots, f(k-1)$ then $f(k)$ will be also a linear combination of $f(1), \ldots, f(k-1), a$).
- If $k \geq n$, the reasoning is as above. Really, the previous reasoning is a particular case of this one:

  $$A = \{f(1), \ldots, f(k-1), \underbrace{f(k)}, f(k+1), \ldots, f(m)\}$$

Then:

$$(\text{insert a A}) = \{f(1), \ldots, a, f(n), \ldots, f(k), \ldots, f(m)\}$$

$f(k)$ was a linear combination of $f(1), \ldots, f(n-1), f(n), \ldots, f(k-1)$ and hence also of $f(1), \ldots, f(n-1), a, f(n), \ldots, f(k-1)$.

We tried to formalize the previous proof in Isabelle/HOL. However, when we were making it we found a new case which we had not considered: In the case insert, we have supposed tacitly that A is dependent to apply the induction hypothesis. However we can not suppose it, we can not prove it using our premises (in this case we have that $(insert\ a\ A)$ is linearly dependent, but from this we can not prove that A is linearly dependent). So we also have to prove the theorem in this case and the proof of it is very similar to the demonstration which has been followed in the book without induction.

In conclusion, instead of what looks firstly, the proof is longer and more difficult with induction than without it. In fact, one case of the induction proof is very similar to the whole non-inductive proof. Nevertheless, we made both proofs in Isabelle/HOL.

Finally, we present the proof of theorem 10.2.1 without induction. We explain it line by line, comparing the Halmos' proof with the formalized one in Isabelle/HOL.

First, we are going to write the theorem in Isabelle. Due to the definition of an indexed set, our element will be between 1 and $card(A-1)$ and not between 2 and $card(A)$ (or n in the notation of the book). Halmos represents the second element as x_2 and we do it as x_1.

The wording of the theorem in Isabelle is as follows. Note that as a premise we are assuming a certain order (indexing) to the set (in our first attemps we proved that there exists an order where the property is true, but really it holds for any order):

```
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)
```

```
∧ (1::nat) ≤ k ∧ k < (card A)
∧ f k = y ∧ y = linear_combination g (f`{i::nat. i<k})"
```

We are assuming three premises: we have a linearly independent set in which the zero is not contained and as we have already said, a certain indexing of the set. The result in the book (theorem 10.2.1) claims that there exists an element in the linearly dependent set A (this element is represented as y) which is in a position k ($f(k) = y$) between 2 and $card(A)$ ($1 \leq k \wedge k < card(A)$ due to our representation of indexed sets) which is a linear combination of the preceding ones ($y = linear\_combination\ g\ (f`\{i :: nat.\ i < k\})$).

Now the proof. We are writing in italic letters the lines of the proof in the book and after that we will present how we have implemented it in Isabelle.

Let us suppose that the vectors $x_1, \ldots, x_n$ are linearly dependent.

We have assumed it in the premises. From this, we can obtain a linear combination equal to zero where not all scalars are zero (this is obvious, but in Isabelle we have to make it step by step). We also prove that the set is a good set and we take the function which indexes it (we have given an order to the set in the premises with the assumption i):

```
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 ∈ coefficients_function (carrier V)"
    and sum_zero: "linear_combination h A=0_V"
    and not_all_zero: "¬ (∀x∈A. h x =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 k the first integer between 2 and n for which $x_1, \ldots, x_n$ are linearly dependent (if worse comes to worst, our assumption assures us that k=n will do).

We will change a little bit the proof. As $A = \{x_1, \ldots, x_n\}$ is linearly dependent, then every linear combination of A equal to zero has scalars not equal to zero. Then we take the last scalar which is not zero (so after it, all scalars will be zero). For that, first of all we have to take the set of the scalars

which are not zero, prove that this set is not empty (this is obvious because the set A was linearly dependent) and then we can obtain the maximum (the book takes the least, but the proof is easier if we take the maximum because, as we have said before, all scalars after the maximun will be zero).

```
let ?A="{k∈{..<card A}. h (f k) ≠ 0_K}"
have finite_A: "finite ?A" by auto
have A_not_empty: "?A≠{}"
  using not_all_zero using 1 by force
def m ≡ "Max ?A"
have m_in_A: "m ∈ ?A"
  using Max.closed[OF finite_A A_not_empty]
  unfolding m_def by force
have "∀x∈{..<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 "¬ (∃x∈?A. m < x)"
  using Max_less_iff [OF finite_A A_not_empty]
  unfolding m_def by auto
hence h_indexing_m_card_zero: "∀x∈{m<..<(card A)}. h (f x) = 0_K"
  by auto
```

Then $\alpha_1 x_1 + \cdots + \alpha_m x_m = 0$ *for a suitable set of* α*'s (not all zero).*

In Isabelle this line requires a lot of work. We have to divide the linear combination of the whole set into two linear combinations: one until m and the other until the end. The second linear combination will be zero because every scalar is zero. The first will be also zero (this is what we want to prove and it is true because the sum of both linear combinations is zero and if the second is zero, then the first will be also zero).

```
have indexing_m_in_aA: "f m ∈ A" using 1
  using m_le_card_aA by auto
have descomposicion_conjunto:
  "{..<(card A)} = {..m} ∪ {m<..<(card A)}"
  using m_le_card_aA unfolding m_def by auto
have "f `{..<(card A)}= f ` ({..m}∪{m<..<(card A)})"
  unfolding descomposicion_conjunto ..
also have"...= f ` {..m} ∪ f `{m<..<(card(A))}" by auto
finally have descomposicion_indexing_ext:
```

```
    "f ‘ {..<card A} = f ‘ {..m} ∪ 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} ⊆ f ‘ {..<card (A)}"
      using m_le_card_aA by auto
    hence "f ‘ {..m} ⊆ 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 ∈ carrier V"
    using cb_l_m unfolding good_set_def by auto
  have "0_V=linear_combination h (f ‘ {..<card A})"
    using sum_zero 1 by auto
  also have
    "...=linear_combination h (f ‘ {..m} ∪ f ‘ {m<..<card A})"
    using descomposicion_indexing_ext by auto
  also have "...= linear_combination h (f ‘ {..m})
    ⊕_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} ∩ f ‘ {m<..<card A} = {}"
    ...
    show "(λy. h y · y) ∈ f ‘ {..m} → carrier V"
    proof (auto,rule mult_closed)
      ...
    qed
    show "(λy. h y · y) ∈ f ‘ {m<..<card (A)} → carrier V"
    proof (auto,rule mult_closed)
      ...
    qed
  qed
  also have "...= linear_combination h (f ‘ {..m}) ⊕_V 0_V"
  proof -
    ...
  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
```

Moreover, whatever the α's, we can not have $\alpha_k = 0$, for then we should have a linear dependence relation among $x_1, \ldots, x_{k-1}$, contrary to the definition of k.

In our proof, we have denoted m where the book uses k. This part is proved with our definition of m (the last scalar which is not zero). Remember:

```
let ?A="{k∈{..<card A}. h (f k) ≠ 0_K}"
def m ≡ "Max ?A"
```

Hence:

$$x_k = \frac{-\alpha_1}{\alpha_k} x_1 + \ldots \frac{-\alpha_{k-1}}{\alpha_k} x_{k-1}$$

as was to be proved.

This is the end of the proof. It is basic and trivial doing it "by hand", but in Isabelle requires more work. We descompose the linear combination until m into two sums: a linear combination until the element $m-1$ and the $m-th$ element multiplicated by it scalar. Hence we can work out the value of x_k ($f(m)$ in our development). To do it we make use of an auxiliary lemma created by us and named *word_out_the_value_of*.

Once we have proved it, we also have to prove that the position is between 2 and $card(A)$ (in our case between 1 and $card(A-1)$). Halmos doesn't prove it in his book. Proving that $m \leq card(A)$ is trivial because $m \in \{0 \leq .. < card(A)\}$ (see the definition of $?A$ above). To demonstrate that $1 \leq m$ we will do it by *reductio ad absurdum*: if not $1 \leq m$ then $m = 0$. By the definition of m, we have that $f(m) \neq 0_V$. Nevertheless, we have proved that $f(m)$ is a linear combination of the preceding ones, and as $f(m)$ is the first element ($m = 0$), then it is a linear combination of the empty set. But we have a contradiction: $f(m)$ is not zero and the linear combination of the empty set is equal to zero.

```
also have  "...
  =h (f m) · (f m) ⊕_V linear_combination h (f ‘ {..<m})"
```

```
  proof -
    ...
  qed
  finally have descomposicion_lc:
    "0_V=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
    ...
  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
```

Thus, we have completed the proof. As it can be observed, in Isabelle the

proof gets complicated and longer, mainly due to the manipulation of finite sums and some details not covered in Halmos.

From this, we are going to present the rest of the code that we have created to implement this section.

As we have said before, 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 simplification 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 ∈ coefficients_function (carrier V) "
  shows "linear_combination f X ∈ carrier V"
proof (unfold linear_combination_def, rule finsum_closed)
  show "finite X" using good_set unfolding good_set_def by auto
  show "(λy. f y · y) ∈ X → carrier V"
  proof (unfold Pi_def, auto)
    fix y
    assume y_in_X: "y ∈ X"
    hence y_in_V: "y ∈ carrier V" using good_set unfolding
good_set_def by fast
    show "f y · y ∈ 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 ⟷ x = 0_V"
proof
  assume l_combination_x: "linear_combination f {} = x"
  have l_combination_zero: "linear_combination f {}=0_V"
    unfolding linear_combination_def
    using finsum_empty by auto
  show "x = 0_V"
    using l_combination_x and l_combination_zero by auto
```

```
next
  assume x_zero: "x = 0_V"
  have l_combination_x: "linear_combination f {} = 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 simplify rule.

```
corollary linear_combination_empty_set [simp]:
  shows "linear_combination f {} = 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 ∈ coefficients_function (carrier V)"
  and x_in_V: "x ∈ carrier V"
  shows "linear_combination f {x} = f x · x"
proof -
  ...
qed
```

A `linear_combination` of `insert x X` is equal to `f x · x` $\oplus_V$ `linear_combination f X`

```
lemma linear_combination_insert:
  assumes good_set_X: "good_set X"
  and x_in_V: "x ∈ carrier V"
  and x_not_in_X: "x ∉ X"
  and cf_f: "f ∈ coefficients_function (carrier V)"
  shows "linear_combination f (insert x X)
  = f x · x ⊕_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 ∉ X" using x_not_in_X .
  show "(λy. f y · y) ∈ X → carrier V"
  proof (unfold Pi_def,auto)
    show "⋀x. x ∈ X ⟹ f x · x ∈ carrier V"
```

```
    proof (rule fx_x_in_V)
      fix y
      assume y_in_X: "y ∈ X"
      show "y ∈ carrier V"
        using good_set_X
        unfolding good_set_def using y_in_X by auto
      show "f ∈ coefficients_function (carrier V)" using cf_f .
    qed
  qed
  show "f x · x ∈ 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 ∈ coefficients_function (carrier V)"
  and all_zero: "⋀x. x ∈ X ⟹ f (x) · x = 0_V"
  shows "linear_combination f X = 0_V"
proof -
  have "linear_combination f X = (⨁_V y∈X. f y · y)"
    unfolding linear_combination_def ..
  also have "...=(⨁_V y∈X. 0_V)"
  proof (rule finsum_cong',auto)
    fix x
    assume x_in_X: "x∈X"
    show "f x · x = 0_V"
      using all_zero[OF x_in_X] .
  qed
  also have "...=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 · linear_combination f X = linear_combination (λi. a ⊗ 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 ⊆ carrier V; a∈ carrier K; f ∈ X→ carrier K⟧
  ⟹ a · (⨁V y∈X. f y · y)=(⨁V y∈X. a· (f y · 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: "(⨁V y∈X. f y · y) ∈ carrier V"
    proof (rule finsum_closed)
      show "finite X" using insert.hyps (1) .
      show "(λy. f y · y) ∈ X → carrier V"
        using insert.prems (1) and insert.prems (3)
        and mult_closed
        by auto
    qed
    have fx_x_in_V: "f x · x∈ carrier V"
      using insert.prems (1) and insert.prems (3)
      and mult_closed
      by auto
    have "(⨁V y∈insert x X. f y · y)=f(x)·x ⊕V(⨁V y∈X. f y · y)"
    proof (rule finsum_insert)
      show "finite X" using insert.hyps (1) .
      show "x∉ X" using insert.hyps (2) .
      show "f x · x ∈ carrier V" using fx_x_in_V .
      show "(λy. f y · y) ∈ X → carrier V"
        using insert.prems (1) and insert.prems (3)
        and mult_closed
        by auto
    qed
    hence "a·(⨁V y∈insert x X. f y · y)
      =a·f(x)·x ⊕V a·(⨁V y∈X. f y · y)"
      using add_mult_distrib1[OF fx_x_in_V
        sum_closed insert.prems(2)] by auto
    also have "...=a·f(x)·x ⊕V (⨁V y∈X. a· f y · y)"
    proof -
      have X_subset_V: "X ⊆ carrier V"
        using insert.prems(1) by auto
      have f1: "f∈ X→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 "...=(⨁V y∈insert x X. a · f y · 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 · y) ∈ X → carrier V"
      ...
    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 "(⨁V y∈X. (a ⊗ f y) · y)=(⨁V y∈X. a · f y · y)"
```

```
    proof (rule finsum_cong')
      ...
    qed
    also have "...=a·(⨁V y∈X. f y · y)"
      using finsum_aux
        [OF finite_X X_subset_V a_in_K f, symmetric] .
    finally show "a · (⨁V y∈X. f y · y) = (⨁V y∈X. (a ⊗ f y) · y)"
      by auto
  qed
qed
```

Now some useful lemmas which will be helpful to prove later 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_linear_dependent_set:* $[\![Y \subseteq X;$ *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 -
  ...
qed
```

Another auxiliary lemma. It will be very useful to prove properties in future sections. If we have an equality of the form **0**$_V$ = *g x* · *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 ∈ coefficients_function (carrier V)"
  and x_in_V: "x ∈ carrier V"
  and gx_not_zero: "g x ≠ 0_K"
  and lc_descomposicion: "0_V = g(x)·x ⊕_V linear_combination g X"
  shows "∃f. f ∈ coefficients_function (carrier V)
  ∧ linear_combination f X = x"
proof -
  ...
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 -
  ...
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 ∈ carrier V"
  and x_not_in_X: "x ∉ X"
  shows "linear_dependent (insert x X)
  ⟹ (∃ f. f∈ coefficients_function (carrier V)
  ∧ linear_combination f X = x)"
proof -
  ...
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 ∈ carrier V" and x_not_in_X:"x ∉ X"
  shows "linear_dependent (insert x X) ⟷
  (∃ f. f∈ coefficients_function (carrier V)
  ∧ 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 10.2.1). 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 ∪ Y)"
  shows "∃y∈Y. ∃g. g ∈ coefficients_function (carrier V)
  ∧ y = linear_combination g (X ∪ (Y - {y}))"
proof -
  ...
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 "∃y∈Y. ∃g. g ∈ coefficients_function (carrier V)
  ∧ y = linear_combination g (Y - {y})"
proof -
  have ld_empty_Y: "linear_dependent({} ∪ Y)" using ld_Y by simp
  have "∃y∈Y. ∃g. g ∈ coefficients_function (carrier V)
    ∧ y = linear_combination g ({} ∪ (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 $a \cdot x = 0 \Longrightarrow a = 0$ because $x \neq 0$.

```
lemma unipuntual_is_li:
  assumes x_in_V: "x ∈ carrier V" and x_not_zero: "x ≠ 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 ∈ coefficients_function (carrier V)"
      and lc: "linear_combination f {x} = 0_V"
      and not_all_zero: " ¬ (∀x∈{x}. f x = 0)"
      unfolding linear_dependent_def by auto
    have fx_not_zero: "f x ≠ 0" using not_all_zero by auto
    have "(f x) · x = 0_V" thm finsum_insert
    proof -
      — We could have used ⟦fa ∈ coefficients_function (carrier V); xa
∈ carrier V⟧ ⟹ linear_combination fa {xa} = fa xa · xa directly or
```

next calculation:

```
      have "linear_combination f (insert x {})
        = (f x) · x ⊕V linear_combination f {}"
        using linear_combination_insert[OF _ x_in_V _ cf_f]
        by auto
      also have "... = (f x) · x ⊕V 0V"
        using linear_combination_of_zero by auto
      also have "... = (f x) · 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 = 0K"
      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 there 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.

The following lemma was the first attempt to prove the theorem 10.2.1. 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: "0V ∉ X"
  shows "∃y ∈ X. ∃g. ∃k::nat.
  ∃f ∈ {i. i<(card X)} → X. f`{i. i<(card X)} = X
```

```
  ∧ g ∈ coefficients_function (carrier V)
  ∧ (1::nat) ≤ k ∧ k < (card X) ∧ f k = y
  ∧ y = linear_combination g (f`{i::nat. i<k})"
proof -
  ...
qed
```

Really, the result that we need to prove in this section correspond closer to the next lemma than the one 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).

Then, the following theorem is the proof of 10.2.1. 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. The proof was explained in detail at the begining of this section (10.2.)

```
theorem
  linear_dependent_set_sorted_contains_linear_combination:
  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)
  ∧ (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)
  ...
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 and it has been explained line by line at the begining of this section.

Chapter 11

Bases

11.1 Definitions

The objective of this section is to present the notions of *basis* and *finite dimensional vector space* as presented in Halmos and how we can implement them in Isabelle/HOL.

Definition 11.1.1 *A basis in a vector space V is a set X of linearly independent vectors such that every vector in V is a linear combination of elements of X.*

We are not going to implement this definition literally, but through the concept of *spanning set*. There are some concepts that Halmos doesn't introduce and we consider that they are important. We are talking about the notions of *span* and *spanning set*.

Definition 11.1.2 *The span of a set A is the set of all linear combinations of the elements of A. In other words, if $A = \{a_1, \ldots, a_n\}$, then*

$$span(A) = \{\lambda_1 a_1 + \cdots + \lambda_n a_n | \lambda_1, \ldots, \lambda_n \in \mathbb{K}\}$$

We present its implementation in Isabelle/HOL. The span of a set A will be the set of the elements $x \in carrier\ V$ for which there exists a *coefficients function* such that we can write x as a linear combination of the elements of A.

definition *span :: "'b set => 'b set"*
where *"span A = {x. ∃g ∈ coefficients_function (carrier V). x = linear_combination g A}"*

We first prove some properties of the *span* of a set.

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 auxiliary 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 ⊆ carrier V"*
shows *"$\mathbf{0}_V$ ∈ span A"*
proof -
have *"$\mathbf{0}_V$ = linear_combination (λx. $\mathbf{0}_K$) 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 · $\mathbf{0}$ ∈ A → carrier V"*
unfolding *Pi_def*
using *mult_closed* **using** *A_in_V* **by** *auto*
show *"⋀i. i ∈ A ⟹ $\mathbf{0}_V$ = $\mathbf{0}$ · 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 auxiliary 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: λ*x. if x = a then* **0** *else g x*.

lemma *exists_function_Aa_A:*
  assumes *cf_g: "g ∈ coefficients_function (carrier V)"*
  and *good_set_A: "good_set A"*
  and *a_in_A: "a ∈ A"*
  shows *"∃ga ∈ coefficients_function (carrier V).*
  (⨁V y∈A - {a}. g y · y) = (⨁V y∈A. ga y · y)"
proof
  ...
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: λ*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 ∈ coefficients_function (carrier V)"*
  and *cf_g: "g ∈ 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∈A"*
  shows *"∃ga ∈ coefficients_function (carrier V).*
  (⨁V y∈A. h y · y) = (⨁V y∈A - {a}. ga y · y)"
proof
  ...
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 ∈ coefficients_function (carrier V)"
    and a_lc: "a = linear_combination g (A-{a})" by auto
  show "span A ⊆ span (A - {a})"
  proof (unfold span_def, unfold linear_combination_def,auto)
    fix f
    assume cf_f: "f∈ coefficients_function (carrier V)"
    show "∃ga ∈ coefficients_function (carrier V).
      (⨁V y∈A. f y · y) = (⨁V y∈A - {a}. ga y · 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 `∃a∈A. span A = span (A - {a})`.

We also need to use `linear_dependent Y ⟹ ∃y∈Y. ∃g. g ∈ coefficients_function (carrier V) ∧ y = linear_combination g (Y - {y})`

```
corollary
  span_minus2:
  assumes ld_A: "linear_dependent A"
  shows "∃a∈A. span A = span (A - {a})"
proof -
```

```
  have "∃a∈A. ∃g. g ∈ coefficients_function (carrier V) ∧ 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 ∉ span A"
  and cb_A: "good_set A"
  and y_in_V: "y ∈ carrier V"
  shows "y ∉ A"
proof (cases "y ∉ A")
  case True thus ?thesis .
next
  case False
  show ?thesis
  proof -
    def g≡"(%x. if x=y then 1 else 0)"
    have cf_g: "g ∈ 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)·y ⊕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 ∈ carrier V" using False cb_A
          unfolding good_set_def by fast
        show "y ∉ A - {y}" by simp
        show "g ∈ coefficients_function (carrier V)" using cf_g .
```

```
      qed
      also have "...=g(y)·y ⊕V 0V "
      proof -
        have "linear_combination g (A-{y})=0V"
        proof -
          have "(⨁V y∈A - {y}. g y · y)=(⨁V y∈A - {y}. 0V)"
            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 "...= 0V"
            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)·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 ∉ span A"
  and y_in_V: "y ∈ 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 -
      ...
    qed
  qed
qed
```

Now we present the definition of a *spanning set*.

Definition 11.1.3 *We say that a finite set $A = \{a_1, \ldots, a_n\}$ is a spanning set of a vector space V if $span(A) = V$.*

We have implemented it in Isabelle in the following way:

```
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)))"
```

The definition consists of two assumptions: first we have that the set is a *good set* (it is finite and in V) and the second is the condition to have that $span(A) = V$.

Moreover, we can generalize this concept for infinite sets:

Definition 11.1.4 *We say that a set A (finite or infinite) is a spanning set of a vector space V if for every $x \in V$ it is possible to choose $a_1, \ldots, a_n \in A$ and $\lambda_1, \ldots, \lambda_n \in \mathbb{K}$ such that $x = \lambda_1 a_1 + \cdots + \lambda_n a_n$. In other words: we can write every element of V as a linear combination of a finite subset (which will depend on the element x) of A.*

Now we show how we have implemented it in Isabelle/HOL:

```
definition spanning_set_ext :: "'b set ⇒ bool"
  where "spanning_set_ext X = (∀x. x ∈ carrier V
  ⟶ (∃A. ∃f. good_set A ∧ A ⊆ X
  ∧ f ∈ coefficients_function (carrier V)
  ∧ linear_combination f A = x))"
```

As we have said before, sums are always 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).

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* ⊆ *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 "∀x. x ∈ carrier V
    ⟶ (∃f. f ∈ coefficients_function (carrier V)
    ∧ linear_combination f X = x)"
  proof (auto)
    fix x
    assume x_in_V: "x ∈ carrier V"
    from sp_X obtain A and f where A_in_X: "A ⊆ X"
      and gs_A: "good_set A"
      and cf_f: "f ∈ 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 ≡ "(λx. if x ∈ A then f x else 0)"
```

```
    have cf_g: "g ∈ 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∪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 ∈ coefficients_function (carrier V)" using cf_f .
      qed
      also have "...=linear_combination g X"
        using arg_cong2 [of g g "A∪X" X "linear_combination"]
        using A_in_X by fast
      finally show ?thesis by fast
    qed
    thus "∃g. g ∈ coefficients_function (carrier V)
      ∧ linear_combination g X = x" using cf_g by auto
  qed
qed
```

From the previous definitions, we can introduce an alternative definition of *basis*. We will use in Isabelle/HOL this one, and later we will prove the equivalence with the definition 11.1.1.

Definition 11.1.5 *A set A in a vector space V is a basis if it is linearly independent and a spanning set.*

Note that A could be finite or infinite (if the set is infinite then we have to take the generalized notions of independence and spanning set).

We show the implementation of the concept of *basis* in Isabelle:

```
definition basis :: "'b set ⇒ bool"
  where "basis X = (X ⊆ carrier V
  ∧ linear_independent_ext X ∧ spanning_set_ext X)"
```

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 ⟷ 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 ⊆ carrier V ∧ linear_independent_ext X ∧ spanning_set_ext X)*.

```
lemma basis_def':
  assumes cb_A: "good_set A"
  shows "basis A ⟷ (linear_independent A ∧ 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 ⟷ (linear_independent A ∧ 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
```

Finally we present the definition of a finite dimensional vector space:

Definition 11.1.6 *A vector space V is finite dimensional if it has 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.[1]

```
locale finite_dimensional_vector_space = vector_space +
  fixes X :: "'c set"
  assumes finite_X: "finite X"
  and basis_X: "basis X"
```

We are fixing a finite set X as the basis of the vector space (now on, X will be a finite basis, not any set). We open this context in order to work with this structure:

```
context finite_dimensional_vector_space
begin
```

11.2 Theorems

Once we are in the context of a *finite dimensional vector space* we can present the proof of some theorems proposed by Halmos and another ones that don't appear in the book. Our objective will be to prove the following theorems:

[1] We will later prove that, for instance, $\mathbb{K}^n$ has a (canonical) finite basis, thus making the previous definition consistent.

- The coordinates of a vector in a given basis are uniquely determined.
- Any linearly independent set can be extended to a basis.
- The proof that every vector space V where V is finite has a basis.[2]

We start with the result that claims that the coordinates of a vector in a basis are unique.

The proof consists in supposing that we have two linear combinations of the elements of one fixed basis equal to the same element $x \in V$. Hence if we define the difference between them, we will obtain another linear combination equal to zero. Due to the basis being independent, then all scalars (which will be the difference between the scalars of the first linear combination and the second one) will be zero, so the scalars of the first linear combination and the second ones are equal and then we have the result.

```
lemma unique_coordenates:
  assumes x_in_V: "x ∈ carrier V"
  and cf_f: "f ∈ coefficients_function (carrier V)"
  and lc_f: "x = linear_combination f X"
  and cf_g: "g ∈ coefficients_function (carrier V)"
  and lc_g: "x = linear_combination g X"
  shows "∀x ∈ X. g x = f x"
proof -
  have "linear_combination f X ⊕_V ⊖_V linear_combination g X
    = x ⊕_V ⊖_V x"
    using lc_f and lc_g by auto
  hence "0_V = linear_combination f X
    ⊕_V ((⊖_K 1_K )· 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
    ⊕_V linear_combination (%i. (⊖_K 1_K )⊗ 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) ⊕_K ⊖_K g(x)) X"
```

[2]We are saying that V is a finite set, not that V is a finite dimensional vector space (if V is finite then it is a finite dimensional vector space, but the opposite implication is false).

```
    unfolding linear_combination_def
  proof -
    ...
  qed
  finally have
    lc_fg: "0_V=linear_combination (%x. f(x) ⊕_K ⊖_K g(x)) X"
    by simp
  have cf_fg: " (%x. (f(x) ⊕_K ⊖_K g(x)))
    ∈ coefficients_function (carrier V)"
  proof (unfold coefficients_function_def, auto)
    ...
  qed
  hence fg_0:"∀x∈X. f(x) ⊕_K ⊖_K g(x)=0_K"
    using linear_independent_X and lc_fg[symmetric]
    unfolding linear_independent_def by auto
  show "∀x ∈ X. g(x)=f(x)"
  proof
    fix y
    assume y_in_X: "y∈X"
    hence y_in_V: "y∈carrier V"
      using good_set_X unfolding good_set_def
      by auto
    have fg_y0: "f y ⊕ ⊖ g y = 0"
      using y_in_X and fg_0 by auto
    have fy_in_K: "f(y)∈ carrier K"
      using cf_f and y_in_V
      unfolding coefficients_function_def by auto
    have gy_in_K: "g(y)∈ carrier K"
      using cf_g and y_in_V
      unfolding coefficients_function_def by auto
    hence "⊖_K(⊖_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
```

Now we present the result which claims that we can complete a linearly independent set to a basis. First of all, let us see the wording of the theorem

(as it appears in Halmos):

Theorem 11.2.1 *If V is a finite-dimensional vector space and if $A = \{y_1, \ldots, y_m\}$ is any set of linearly independent vectors in V, then, unless the $y's$ already form a basis, we can find vectors $y_{m+1}, \ldots, y_{m+p}$ so that the totality of the $y's$, that is, $\{y_1, \ldots, y_m, y_{m+1}, \ldots, y_{m+p}\}$, is a basis. In other words, every linearly independent set can be extended to a basis.*

We are going to see how we have made the proof in Isabelle/HOL comparing it with the proof presented in Halmos. We have to make several auxiliary and previous results to complete it (for example, as we will see later, we have to define a function which removes the first element which is a linear combination of the preceding ones in a set, iterate it until achieve an independent set, prove some properties of this function, ...).

First we present the theorem with the proof in case that the independent set is not the empty set. The wording is easy:

```
lemma extend_not_empty_independent_set_to_a_basis:
  assumes li_A: "linear_independent A"
  and A_not_empty: "A≠{}"
  shows "∃B. basis B ∧ A ⊆ B"
```

We are going to follow the proof in Halmos line by line explaining how we have implemented it in Isabelle. We write in Italics the proof in the book and after that we show the Isabelle code with its explanation.

Since V is a finite-dimensional, it has a finite basis, say $\{x_1, \ldots, x_n\}$.

Let us suppose that the linear independent set is $A = \{y_1, \ldots, y_m\}$. In our definition of finite-dimensional vector space, we have fixed a finite basis, denoted by X. We can define a new set C as $C = (X - A)$. In the next step we will explain why we have defined this set this way.

```
proof -
  have cb_A: "good_set A" using l_ind_good_set[OF li_A] .
  def C ≡"X-A"
  have igualdad_conjuntos: "A∪X=A∪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∩C={}" using C_def by auto
```

We consider the set S of vectors: $\{y_1, \ldots, y_m, x_1, \ldots, x_n\}$ *in this order.*

Here we realize one question that Halmos doesn't explain...what happens if exist elements of $A = \{y_1, \ldots, y_m\}$ that are also in the basis $X = \{x_1, \ldots, x_n\}$? We would have a multiset, since some elements may appear in S more than once. We avoid this problem considering the set $C = (X \cap A)$ and doing $A \cup C = A \cup (X - A)$ instead of $A \cup X$. Now the union is disjoint. To obtain its indexing we make use of an auxiliary lemma created by us in the section 10.1 named *indexing_union*.

```
  have "∃h. indexing (A ∪ C, h) ∧ h ‘ {..<card A} = A ∧ 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∪C),h)"
    and surj_h_A: "h ‘ {..<card A} = A"
    and surj_h_B: "h ‘ ({..<card A + card C} - {..<card A}) = C"
    by auto
```

We apply to this set the theorem 10.2.1 several times in succession.

Firstly, we define a function called called *remove_ld* which does the next: given an indexed set it returns another one where we have removed the first element which is a linear combination of the preceding ones (and the indexing has been accordingly modified).

Now we present the implementation of this function in Isabelle/HOL:

In the definition, making use of previous theorem:

```
⟦linear_dependent Xa; 0_V ∉ Xa⟧ ⟹ ∃y∈Xa. ∃g k. ∃f∈{i.
i < card Xa} → Xa. f ‘ {i. i < card Xa} = Xa ∧ g ∈
coefficients_function (carrier V) ∧ 1 ≤ k ∧ k < card Xa ∧ f k
= y ∧ y = linear_combination g (f ‘ {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*⟧ ⟹ ∃*y*∈*Xa*. ∃*g k*. ∃*f*∈{*i*. *i* < *card Xa*} → *Xa*. *f* ‘ {*i*. *i* < *card Xa*} = *Xa* ∧ *g* ∈ *coefficients_function (carrier V)* ∧ *1* ≤ *k* ∧ *k* < *card Xa* ∧ *f k* = *y* ∧ *y* = *linear_combination g* (*f* ‘ {*i*. *i* < *k*}).

The definition of *remove_ld*, for a set with an indexing A, is as follows:

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)"

Once we have the definition of the function *remove_ld*, we want to apply it iteratively to a linearly dependent set until we achieve a linearly independent set. To do that we have defined a function named *iterate_remove_ld*. This function takes an indexed set and removes the first element of it which is a linear combination of the preceding ones until we achieve a linearly independent set. We have to make use of the *partial_function* package, which permits the definition in Isabelle/HOL of functions which termination cannot be proved. We remember here that HOL is a logic of total functions and thus every recursive function must be terminating. The trick here consist in defining a domain predicate for the function, where it will always terminate. Results can be proved about this definition as long as we include this domain conditions. For instance, in our case, *iterate_remove_ld* terminates if we have a finite set which is a subset of *carrier* V and f is a proper indexing. More information on the function package and the definition of partial functions can be found in [37].

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))))"
```

In the first place, the set is linearly dependent, since the y's are (as are all vectors) linear combinations of the x's. Hence some vector of S is a linear combination of the preceding ones. Let z be such vector. Then z is different from any y_i, $i = 1 \ldots m$ (since y's are linearly independent). So that z is equal to some x, say $z = x_i$. We consider the new set S' of vectors: $\{y_1, \ldots, y_m, x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n\}$. We observe that every vector in B is a linear combination of vectors in S', since by means of $\{x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n\}$ we may express any vector (The x's form a basis). If S' is linearly independent, we are done.

This paragraph could be summed up in two properties that must be satisfied by our function *iterate_remove_ld*: that it preserves the span and that its result is a linear independent set.

We have proved such properties auxiliary lemmas, called *linear_independent_iterate_remove_ld* and *iterate_remove_ld_preserves_span*. To use this auxiliary lemmas in our theorem, we only have to prove that their premises hold.

```
have li_iterate: "linear_independent(iterate_remove_ld (A∪C) h)"
proof (rule linear_independent_iterate_remove_ld)
  show "A ∪ C ⊆ carrier V"
    ...
  show "0_V ∉ A ∪ C"
    using li_A zero_not_in_linear_independent_set C_def by auto
  show "indexing (A ∪ C, h)" using indexing_AC_h .
qed
have "span(iterate_remove_ld (A∪C) h)=span(A∪C)"
proof (rule iterate_remove_ld_preserves_span)
  show "A ∪ C ⊆ carrier V"
    ...
  show "indexing (A ∪ C, h)" using indexing_AC_h .
  show "0_V ∉ A ∪ 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∪C) h)=carrier V" .
  have basis_iterate: "basis (iterate_remove_ld (A∪C) h)"
  proof (unfold basis_def, rule conjI3)
    show "iterate_remove_ld (A ∪ C) h ⊆ 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
```

If is not, we apply the theorem 10.2.1 again and again the same way till we reach a linearly independent set containing $\{y_1, \ldots, y_m\}$ *in terms of which we may express every vector in* V*.*

With the function *iterate_remove_ld* we are applying the theorem again and again. Last thing we have to prove is that the independent set A is contained in the result that the function returns. This is another auxiliary lemma, named *A_in_iterate_remove_ld*. In our theorem, we prove its premises.

```
  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
```

This last set is a basis containing the y's.

The result that returns the function *iterate_remove_ld* is the required basis; we have proved the three properties that we needed:

- The result is a linearly independent set

  (using the lemma *linear_independent_iterate_remove_ld*)

- The result is a spanning set (using the lemma *iterate_remove_ld_preserves_span*).

- The independent set A is contained in the result of the function (using the lemma *A_in_iterate_remove_ld*).

So we can finish the proof with the next sentence:

```
  show ?thesis using A_in_iterate and basis_iterate by auto
qed
```

Just a little comment: in the case that A is the empty set, then the result is trivial, we only have to consider the finite basis X (fixed by the assumptions of a finite-dimensional vector space) and then $A = \{\} \subseteq X$.

In Isabelle it is also easy. We present the general statement, covering both the cases where $A = \{\}$ and $A \neq \{\}$:

```
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 not shown the proofs of the auxiliary lemmas that we have used. These lemmas are long and hard to be shown here and they need another auxiliary results in order to prove them (we have to descompose much more the results to be able to prove them in Isabelle/HOL). For example, to give an idea of the effort made to prove them, we can say that all previous proofs of auxiliary lemmas and results take up about 900 code lines (in the book the proof is over 15 lines).

For example, we present the proof of the property *linear_independent_iterate_remove_ld* which requires a complex induction reasoning and takes about 100 lines. This lemma claims that the result of applying the function *iterate_remove_ld* to any finite set A in *carrier* V will be always independent.

We are going to make the proof firstly by dividing in cases (with respect to the condition `linear_independent A`) and after that by total induction over the cardinal of the set: `(⋀x. (⋀y. f y < f x ⟹ P y) ⟹ P x) ⟹ 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 10.1:

```
indexed_set_induct2: ⟦indexing (A, f); finite A; ⋀f. indexing
({}, f) ⟹ P {} f; ⋀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)⟧ ⟹ 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 ⊆ carrier V"
  and not_zero: "0_V ∉ 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: "¬ 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 ⊆ carrier V"
          unfolding Y_def
          using remove_ld_preserves_carrier
          [OF less.prems (3), of h] .
        show "0_V ∉ 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
```

We have proved that any independent set can be extended to a basis, but we have not proved that there exists a basis (we have supposed it as a premise 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 [9] 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 - {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 - {0$_V$}*. This function will return us a linearly independent set which span is the same as the span of *carrier V - {0$_V$}*. Proving that *span (carrier V - {0$_V$}) = carrier V* we will obtain the result (because *carrier V - {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 - {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])
      ...
    qed
  next
    case False show ?thesis
    proof (rule exI[of _ ?g])
      ...
    qed
  qed
qed
```

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

```
corollary span_V_minus_zero_is_V:
  assumes finite_V: "finite (carrier V-{0_V})"
  shows "span (carrier V-{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 "∃B. basis B"
proof -
  have finite_V_zero: "finite (carrier V - {0_V})"
    using finite_V by simp
  from finite_V_zero obtain f
    where indexing: "indexing (carrier V-{0_V},f)"
    using obtain_indexing by auto
  have 1:"span (iterate_remove_ld (carrier V-{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-{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-{0_V}) f)"
    using "2" l_ind_good_set by fast
  have "basis (iterate_remove_ld (carrier V-{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-{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 ⊆ carrier V"
  and sg_B: "spanning_set B"
  shows "spanning_set (B-{0_V})"
proof (unfold spanning_set_def,auto)
  ...
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)
  ...
qed

lemma vector_space_contains_spanning_set_ext:
  shows "∃A. spanning_set_ext A ∧ A ⊆ carrier V"
  using spanning_set_ext_carrier_V by blast
```

Chapter 12

Dimension

12.1 Theorems

Our objective in this section is to prove that every basis of a given vector space V has the same cardinality (theorem 1 of section 8 in Halmos). The proof is based on an iterative algorithm over a given set of elements Y, and thus we will need to fix an indexation for it. We are not going to follow exactly the proof presented in the book; we will obtain the result as a corollary of the next theorem:

Theorem 12.1.1 (Swap theorem) *If X is a linearly independent set of V and Y is any spanning set of V, then $card(X) \leq card(Y)$.*

The statement of theorem 12.1.1 can be also presented as "any set of linearly independent elements will have cardinality smaller or equal than any spanning set".

Corollary 12.1.2 *The number of elements in any basis of a finite-dimensional vector space V is the same as in any other basis.*

To prove the theorem 12.1.1 we will make use of the same iterative reasoning that Halmos makes in the book. (a proof of it is in [8]). Really, Halmos is precisely proving theorem 12.1.1 inside his proof of 12.1.2.

The proof of the corollary is easy: let A and B two bases (so both of them are linearly independent and spanning sets). We have:

- As A is linearly independent and B is a spanning set, then

$$card(A) \leq card(B)$$

- As B is linearly independent and A is a spanning set, then

$$card(B) \leq card(A)$$

So we have the result: $card(A) = card(B)$

Now we present the proof of the corollary in Isabelle. We will follow the proof presented above (we will use twice the swap theorem):

```
theorem eq_cardinality_basis:
  assumes basis_B: "basis B"
  and finite_B: "finite B"
  shows "card X = card B"
proof -
  have "∃f. indexing (X,f)" using obtain_indexing[OF finite_X] .
  from this obtain f where indexing_X: "indexing (X,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
  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 ≤ card B"
  proof (rule swap_theorem)
    show "indexing (X, f)" using indexing_X .
    show "indexing (B, g)" using indexing_B .
    show "X ⊆ carrier V"
      using finite_basis_implies_good_set[OF basis_X finite_X]
      unfolding good_set_def by simp
    show "B ⊆ 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 "0_V ∉ B"
      using zero_not_in_linear_independent_set[OF li_B] .
  qed
  have cardX_ge_cardB: "card X ≥ card B"
  proof (rule swap_theorem)
    show "indexing (B, g)" using indexing_B .
    show "indexing (X, f)" using indexing_X .
    show "X ⊆ carrier V"
      using finite_basis_implies_good_set[OF basis_X finite_X]
      unfolding good_set_def by simp
    show "B ⊆ 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 "0_V ∉ 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)
```

Now is time to present the proof of the swap theorem in Isabelle. We are going to define a new function which takes two indexed sets and returns another two such that:

swap_function ((A,f), (B,g)):

- **First component:** We remove the first element of A, in other words: the function returns the set $A - \{f(0)\}$ (and the corresponding indexation).

- **Second component:**
  - If $f(0) \in B$ then simply we change the indexation moving that element to the first position of B.
  - If $f(0) \notin B$, then we add it in the first position of B and after that we will remove the first element which is a linear combination of the preceding ones using the function *remove_ld*[1]

The intuitive idea of this function is that given a tuple of a linearly independent set A in the first component and a spanning set B in the second one (with the corresponding indexing functions, so they are indexed sets), then, the function returns us another two indexed sets: in the second one the function adds the first element of A to B and then it removes the least element which is a linear combination the preceding ones of this set. The function also removes the first element of A in the first component of the result. In other words:

$$swap\_function\ (\{a_1, \ldots, a_n\} \times \{b_1, \ldots, b_m\}) = (\{a_2, \ldots, a_n\} \times \{a_1, b_1, \ldots, b_{i-1}, b_{i+1}, \ldots, b_m\}) \tag{12.1}$$

Where b_i is the first element which is a linear combination of the preceding ones $(a_1, \ldots, b_{i-1})$

This function preserves the linear independence of the first component and the span of the second one (amongst other properties). In addition, as in the proof that every independent set can be extended to a basis and supposing that A is independent, we will be removing one element of B in the second component (because we are adding elements of an independent set in the first positions).

If A is linearly independent and B is a spanning set, then we have to prove that $card(A) \leq card(B)$. Suppose that $card(A) > card(B)$ and then we apply *swap_function* $card(B)$ times. We will obtain that in the second component of the result there will be only elements of A (but not all). This is because we will have removed *card* B elements of B in the second component (one in each iteration, so we will have removed all elements of B). Let be C that set, we will have:

[1]This is the same function that we have used to make the proof that every independent set can be extended to a basis.

- $C \subset A$ (strict).
- $span(C) = V$ (because the second component was a spanning set and the function preserves the span). So C is a spanning set.

Let be $x \in A$ but $x \notin C$ (this element exists because $C \subset A$ strictly). As C is a spanning set, we can express x as a linear combination of elements of C. However, this is a contradiction with A being linearly independent (because $C \cup \{x\}$ would be linearly dependent and as $C \cup \{x\} \subseteq A$ then A would be dependent).

Note that in the proof we are applying the function *swap_function* $card(B)$ times. This will be a problem for our implementation: the power (multiple composition with itself) of a function is not implemented in Isabelle. We have to define it. In addition, we have to prove each property of *swap_function* in general (as an invariant of the function) using induction (in this case, structural induction over the natural numbers will be enough).

We also have to note why we have had to separate the second component in cases, where $f(0) \in B$ and not. We can't apply the function *remove_ld* to an independent set, and we don't know if B is it or not (is a spanning set but it could be dependent or independent). If $f(0) \notin B$ then we are adding the first element of A in B and as B is a spanning set, hence the result is dependent and we can apply *remove_ld*. However, if $f(0) \in B$ then if we add it to B the result will be B again, and we don't know if it is dependent. In this case, we move it to the first position of B (we want to situate the elements of A in the first positions of B).

Now we are going to see how we implement it in Isabelle.

```
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))"
```

In order to remove the first element in the first component of the result, we have made use of a new function called *remove_iset_0*. We just have a function denoted by *remove_iset* to remove one element of an indexed set (we give an indexed set and one position to this function and it returns us a new

indexed set without the element of that position), but we have to do this redefinition because we can not iterate (compose over itself) the function (it has two input arguments and returns only one: $remove\_iset: iset \times \mathbb{N} \rightarrow iset$). For this reason we have to make a new function which can be iterated:

```
definition remove_iset_0 :: "'e iset => 'e iset"
  where "remove_iset_0 A = remove_iset A 0"
```

As we have said before, the second component has been divided in cases. If $f(0) \in B$, then we want to move this element to the first position of B. How could we make it? Firstly, we will remove that element of the set using the function *remove_iset* (we need to know the position where it is, so we will use also the function *obtain_position*[2]). After that we will insert it in the first position (position 0) using the function *insert_iset*[2].

The second case is if $f(0) \notin B$ and then we have to add this element in the first position of B and after that remove the first element which is a linear combination of the preceding ones. The implementation of this part is simple: we will insert the element with the function *insert_iset* and after that we will apply to the result the function *remove_ld* (we can apply it because the result of the function *insert_iset* will be in this case a linearly dependent set).

We have seen that we want to iterate *swap_function* a specific number of times, so we need to implement the power of a function because (surprisingly) it is not in the Isabelle library. We are interpreting the power of a function as a composition with itself and for that we will do an *instantiation* of the operation *power* (we define the product as the composition and the one as the identity function).

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"
```

[2]The functions *obtain_position* and *insert_iset* have been defined in section 10.1

```
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 ∘ (f^n)"
  unfolding power.simps [of f]
  apply (rule ext)
  unfolding times_fun_def by simp

corollary fun_power_suc_eq:
  shows "(f^(Suc n)) x = f ((f^n) x)"
  using fun_power_suc by (metis id_o o_eq_id_dest)
```

Once we have defined the power of a function, then we can prove some properties of *swap_function* and generalize them by induction in case we iterate the function n times. There are lot of properties that this function verifies, for example: the first component preserves the independence, the

second one preserves the span, both are indexings, in each iteration we reduce in one the cardinality of first component ...

One difficult that we have found is to prove the properties for the second component: we have to prove it in the two cases that we have defined, when $f(0) \in B$ and when not. The effort to demonstrate all these properties was very high, it took up about 1800 code lines (the whole theorem takes up 25 lines in Halmos).

Most of them are basic properties but we need to prove it in order to be able to demonstrate another ones less simple. In addition, lot of them are invariants of the *swap_function*, so first we prove the property in case $n = 1$. To generalize it we will do by 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.

In order to give examples, here we present the proof of that the first component of the function *swap_function* preserves the independence of the set.

First we prove in case that $n = 1$ (we apply the function once):

```
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
```

After that, we can generalize it by induction (the property is preserved if we apply the function n times).

```
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
```

We find more difficulties while we were making the formalization in Isabelle. There are properties that demand some assumptions over *swap_function* and we had to generalize these assumptions before.

For example, we want to prove that the first component is always an indexing. In case $n = 1$ we have the following wording:

```
lemma fst_swap_function_indexing:
  assumes  indexing_A: "indexing (A,f)"
  and A_in_V: "A ⊆ carrier V"
  shows "indexing (fst(swap_function ((A,f),(B,g))))"
proof

  ...

qed
```

Appart from the premise $indexing(A, f)$ we have another one: $A \subseteq carrier\ V$. Hence, we can generalize it:

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

...

qed

In order to prove it we want to apply the lemma *fst_swap_function_indexing* to $swap\_function^n((A, f), (B, g))$. By induction hypothesis, we know that $indexing(fst(swap\_function^n((A, f), (B, g))))$. However we need one property that we have not proved yet: $fst(fst(swap\_function^n((A, f), (B, g)))) \subseteq carrier\ V$. For this reason, we have to prove it first.

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 ⊆ carrier V; A ≠ {};*
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 ⊆ carrier V; A ≠ {};*
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 hypoth-

esis, 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

   ...

qed
```

Now we can obtain the properties separately as corollaries, here we present for example the first one:

```
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: "0_V ∉ B"
  and n_l_cardA: "n<card A"
  shows
  "0_V ∉ iset_to_set (snd ((swap_function^n) ((A, f), B, g)))"
  using zeronotin_sg_carrier_indexing assms by simp
```

There are two important, longer and difficult auxiliary lemmas that we have to prove to demonstrate the swap theorem. The first one represents properties that the function has during the process of iterate and it is a laborious and ugly lemma, of 400 code lines (we don't present here the proof):

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 ⊆ carrier V"*

and *A_not_empty: "A≠{}"*

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∪C=Z"* — Of course, the union of A and C is Z.

and *disjoint: "A∩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 *"∃y∈B. iset_to_set (snd(swap_function ((A,f),(B,g))))*
=(insert (f 0) (B-{y}))
∧ y ∉ C
∧ iset_to_index (snd(swap_function ((A,f),(B,g))))
` {..<card (C) + 1} = C ∪ {f 0}"

proof

...

qed

The lemma claims, when applying *swap_function*, in the second component we are removing elements of B and introducing the first elements of Z in the first positions.. Now we present the second auxiliary lemma, it takes up to 200 code lines (we ommit the proof).

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: "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)

   ...

qed
```

Finally we present the proof of *swap_theorem*. We make use of the auxiliary lemmas presented above and as we have done in other proofs, we separe the proof into cases, whether the independent set is not empty or is (this last case is trivial, because $card\ \{\} = 0 \leq card\ B$). In case that the independent set is the empty one, then we follow literally the proof presented at the beginning of this section:

```
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 12.1.2 (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
```

12.2 Definition and other dimension theorems

Now we present the definition of dimension as presented in Halmos and its implementation in Isabelle/HOL.

Definition 12.2.1 *The dimension of a finite-dimensional vector space V is the number of elements in a basis of V.*

As we are in the context *finite dimensional vector space* in Isabelle/HOL, we have a finite basis fixed denoted by X. So that, the definition of *dimension* is easy: it will be the cardinality of X.

```
definition dimension :: "nat"
  where "dimension = card X"
```

If we have another basis, the dimension is equal to its cardinality (thanks to lemma *eq_cardinality_basis* that we presented before).

```
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)
```

Now we present the proofs of theorems about dimension and the relation between this concept and the notions of linear dependence and independence.

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 ≤ dimension"
proof -
  have "∃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 "∃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 ⊆ carrier V"
      by (metis assms good_set_in_carrier l_ind_good_set)
    show "X ⊆ 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  "0_V ∉ X"by (metis li_X zero_not_in_linear_independent_set)
  qed
qed
```

Halmos presents this theorem:

Theorem 12.2.2 *Every set of $n+1$ vectors in an n-dimensional vector space V is linearly dependent. A set of n vectors in V is a basis if and only if it is linearly independent.*

We make a generalization of the first part of it in Isabelle (the theorem 12.2.2 is a particular case). For the second line (A set of n vectors in V is

a basis if and only if it is linearly independent we need to proof firstly some lemmas).

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 ⊆ 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 ≥ dimension"
proof -
  have finite_A: "finite A" and A_in_V: "A ⊆ carrier V"
    using sg_A unfolding spanning_set_def and good_set_def
    by fast+
  have "∃f. indexing (X,f)" using obtain_indexing[OF finite_X] .
  from this obtain f where indexing_X: "indexing (X,f)" by fast
  have "∃g. indexing (A-{0_V},g)" using obtain_indexing finite_A
    by blast
```

```
  from this obtain g where indexing_A: "indexing (A-{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 "∃y. y ∈ (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 ∈ (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) ≤ dimension"
          using card_li_le_dim by simp
        hence "card A + 1 ≤ 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 ⟹ linear_independent A"
proof-
 assume sg_A: "spanning_set A"
 hence A_in_V: "A ⊆ 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 "∃y∈A. ∃g. g ∈ coefficients_function (carrier V)
```

```
        ∧ y = linear_combination g (A - {y})"
        using  exists_x_linear_combination2[OF ld_A] .
      from this obtain y g where y_in_A: "y∈A"
        and cf_g: "g ∈ 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 ∈ A" using y_in_A .
        show "∃g. g ∈ coefficients_function (carrier V)
          ∧ 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 ⟹ basis A"
  by (metis assms card_li_set_eq_basis_imp_li
    card_sg_set_eq_basis_imp_li)
```

Finally we present the last part of the theorem 12.2.2. We make use of some previous lemmas:

```
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)

Chapter 13

Isomorphism

The development of this chapter was hard and it takes up 3500 code lines. Of course, we don't show here the complete development but only the most representative and main definitions and theorems. The reason of so many lines is that there are some concepts that in the eyes of a mathematician are simply but in Isabelle/HOL are hard to be implemented, for example the definition of $\mathbb{K}^n$ or a *canonical basis*.

The objective is to prove that there exists an isomorphism between any n-dimensional vector space V over a field $\mathbb{K}$ and $\mathbb{K}^n$.

First of all, we define the concept of $\mathbb{K}^n$:

Definition 13.0.3 *The* cartesian power *of a set* $\mathbb{K}$ *can be defined as the set of n-tuples which k-th element belongs to* $\mathbb{K}$*, in other words:*

$$\mathbb{K}^n = \underbrace{\mathbb{K} \times \mathbb{K} \times \cdots \times \mathbb{K}}_{n} = \{(x_1, \ldots, x_n) | x_i \in \mathbb{K}\ \forall i. 1 \leq i \leq n\}$$

As $\mathbb{K}$ is a field, hence $\mathbb{K}^n$ is so if we consider the natural operations (componentwise defined).

This definition looks like simple,however its implementation in Isabelle/HOL can be done in very different ways. Now we introduce the concept of the *canonical basis of* $\mathbb{K}^n$.

Definition 13.0.4 *The* canonical basis of $\mathbb{K}^n$ *is the set of n-tuples* $\{(1_{\mathbb{K}}, 0_{\mathbb{K}}, \ldots, 0_{\mathbb{K}}), (0_{\mathbb{K}}, 1_{\mathbb{K}}, 0_{\mathbb{K}}, \ldots, 0_{\mathbb{K}}), \ldots, (0_{\mathbb{K}}, \ldots, 0_{\mathbb{K}}, 1_{\mathbb{K}})\}$

We present the definition of *linear map* following [11] (Halmos introduces this concept in his chapter number two, although he uses the properties of a *linear map* in his proof).

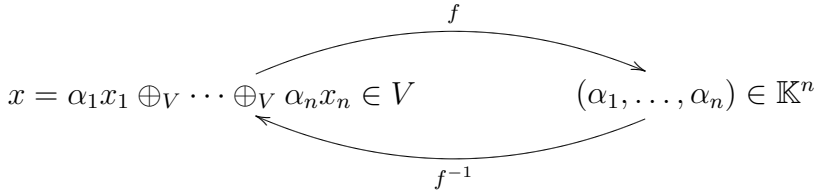
Definition 13.0.5 *A* linear map *from V to W (with V and W vector spaces over a field $\mathbb{K}$) is a function $T\colon V \to W$ with the following properties:*

- **additivity:** $T(u \oplus_V v) = Tu \oplus_W Tv$ *for all* $u, v \in V$
- **homogeneity:** $T(a \cdot_V v) = a \cdot_W (Tv)$ *for all* $a \in \mathbb{K}$ *and all* $v \in V$

And finally the notion of *isomorphism* between two *vector spaces*:

Definition 13.0.6 *Two vector spaces V and W over the same field $\mathbb{K}$ are isomorphic if there exists a linear map $f\colon V \to W$ such that is a bijection.*

Let $X = \{x_1, \ldots, x_n\}$ be a basis of V. The isomorphism between V and $\mathbb{K}^n$ is easy to understand:



However, in Isabelle/HOL is difficult to be implemented.

STEP ONE: FROM V TO $\mathbb{K}^n$

Let X a basis of V. As X is a finite set, we can give it an indexing, for example: $X = \{x_1, \ldots, x_n\}$. In order to do that, we will define a function named *indexing_X* which will return us some function that gives an order to the set (we will fixed this indexing, since we have to keep it through the proof).

Let x be any element of V. Hence we can write x as a linear combination of the elements of the basis X: $x = \alpha_1 x_1 \oplus_V \cdots \oplus_V \alpha_n x_n$ for some $\alpha_1, \ldots, \alpha_n \in \mathbb{K}$. We know that this linear combination is uniquely determined. We will obtain this linear combination of the elements of the basis X with one function that we will call *lin_comb* (it returns us the function f which makes $x =$ *linear_combination* f X)[1]. The function *lin_comb* is well-defined since we

[1] In other words: $x =$ *linear_combination* (*lin_comb* x) X

have proved previously that for every x in *carrier* V, its decomposition is unique. Note that we have to use again the ϵ definite operation.

The n scalars of this linear combination will be the components of the vector of $\mathbb{K}^n$ that we are looking for, in other words: $(\alpha_1, \ldots, \alpha_n)$.

We need to manage to represent $(\alpha_1, \ldots, \alpha_n)$ using that $x = \alpha_1 x_1 \oplus_V \cdots \oplus_V \alpha_n x_n$. We will do it in the next way: we can write $(\alpha_1, \ldots, \alpha_n)$ as a finite sum of elements of the canonical basis of $\mathbb{K}^n$:

$(\alpha_1, \ldots, \alpha_n) = \alpha_1 \cdot (1, 0, \ldots, 0) \oplus_{\mathbb{K}^n} \cdots \oplus_{\mathbb{K}^n} \alpha_n \cdot (0, \ldots, 0, 1)$

Hence the result is easy, we only have to take the scalars of the linear combination obtained with *lin_comb* for x and multiply them (with the scalar product of $\mathbb{K}^n$) with the corresponding vector of the canonical basis. Finally we will sum all.

We will do it with a function named *iso_V_K_n*. To complete the proof we have to demonstrate that this function is also a *linear map*.

STEP TWO: FROM $\mathbb{K}^n$ TO V

Let $v = (\alpha_1, \ldots, \alpha_n)$ be a vector of $\mathbb{K}^n$. Hence, the corresponding $x \in V$ will be $x = \alpha_1 x_1 \oplus_V \cdots \oplus_V \alpha_n x_n$. How could we make it in Isabelle? We have to use again that $\{x_i\}_{i \in \{1 \ldots n\}}$ are a basis and thus every x can be uniquely determined as the finite sum $\sum_{i=1}^{n} \alpha_i \cdot_V x_i$ This time is easier: we only have to multiply each component of $v = (\alpha_1, \ldots, \alpha_n)$ with the corresponding element of the basis $X = \{x_1, \ldots, x_n\}$ and finally sum all again to obtain the linear combination which will be equal to x.

In order to do that we will define a function named *iso_K_n_V*. To terminate the proof we have to demonstrate that this function is also a *linear map*.

From this we begin to implement all concepts presented above:

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

First we make a type definition of the notion of vector which will be useful to represent the elements of $\mathbb{K}^n$. The definition consists of a pair of a function f which maps natural numbers to the elements of the vector and a natural number which expresses the length of the vector minus one, that is to say, the natural which image by the function is the last element. In effect, the

last element of the vector is a_4 and $f(3) = a_4$.

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 [10].

```
types 'a vector = "(nat => 'a) * nat"
```

For example, to represent (a_1, a_2, a_3, a_4) we have a vector $(f, 3)$ where $f(0) = a_1$, $f(1) = a_2$, $f(2) = a_3$ and $f(3) = a_4$. The length of the vector is 4, so the second component will be 3. Note that we don't have unicity of representation. For example, two functions like those:

- A function **f** such that $f(0) = a_1$, $f(1) = a_2$, $f(2) = a_3$, $f(3) = a_4$ and for all $n \geq 4$ then $f(n) = 0$.
- Another function **g** such that $g(0) = a_1$, $g(1) = a_2$, $g(2) = a_3$, $g(3) = a_4$ and for all $n \geq 4$ then $g(n) = 1$.

Hence $(f, 3)$ and $(g, 3)$ represent the same vector (a_1, a_2, a_3, a_4), but their functions are different. We will have to fix this situation later, in such a way that we can use traditional HOL equality to compare vectors. Two vectors will be equal if both their functions and their lengths are equal.

Now we can define another two functions: the first one returns the ith component of a vector and the second one returns the second component of the pair (the length minus one). They are simply abbreviations.

```
definition
    ith :: "'a vector => nat => 'a"
  where "ith v i = fst v i"

definition
  vlen :: "'a vector => nat"
  where "vlen v = snd v"
```

We have defined the notion of a vector, so now we can do the same with the concept of $\mathbb{K}^n$. The following definition represents the carrier set of the vector space:

```
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))}"
```

The carrier is represented as the set of vectors of n components (vlen $n-1$) which are elements of $\mathbb{K}$ (or A with the notation of the definition). As we need to have unicity of representation of elements in $\mathbb{K}^n$, we give a canonical representative to the elements of the carrier imposing that the function of the vector maps the natural numbers greater or equal to n to 0.

For the elements in *K_n_carrier A 0* we must note that its first component will be 0 and the second one will be also 0, so the only element in $\mathbb{K}^0$ is the 0 ($\mathbb{K}^0 = \{0_{\mathbb{K}}\}$).

We have the definition of the carrier, but now we also need to define the operations of the vector space $\mathbb{K}^n$ (the addition and the scalar product). Here we present the first one:

definition

```
  K_n_add :: " nat => 'a vector => 'a vector => 'a vector"
    (infixr "⊕ı" 65)
  where "K_n_add n = (λv w. ((λi. ith v i ⊕_R ith w i), n - 1))"
```

The explanation is easy: given two vectors of $\mathbb{K}^n$, then we obtain another one of $\mathbb{K}^n$ ($vlength = n-1$) in which the components are added one by one.

The next definition that we need is the concept of zero of $\mathbb{K}^n$ which will be $\underbrace{(0,\ldots,0)}_{n\ components}$:

definition `K_n_zero :: "nat => 'a vector"`
  where `"K_n_zero n = ((λi. 0_R), n - 1)"`

We are now forced to define also operations `K_n_mult` and `K_n_one` for our *abelian group* $\mathbb{K}^n$. 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_mult :: "nat => 'a vector => 'a vector => 'a vector"
  where "K_n_mult n = (λv w. ((λi. ith v i ⊗_R   ith w i),
  n - 1))"

definition K_n_one :: "nat => 'a vector"
  where "K_n_one n = ((λi. 1_R), n - 1)"
```

Finally using the definition of carrier, add, zero, mult and one we can define the concept of K^n:

```
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) |)"
```

We can prove that $\mathbb{K}^n$ is an (additive) abelian group:

```
lemma abelian_group_K_n:
  shows "abelian_group (K_n n)"
  unfolding K_n_def
proof (intro abelian_groupI)
...
qed
```

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 "⊙"  65)
  where "a ⊙ b = (λn::nat. a ⊗_R ith b n, vlen b)"
```

This scalar product is an operation which satisfies the properties presented in chapter 6. It is an operation $\mathbb{K} \times \mathbb{K}^n \to \mathbb{K}^n$ where we are multiplying each

component of a vector $b \in \mathbb{K}^n$ by a scalar $a \in \mathbb{K}$. In other words:

$$a \odot (b_1, \ldots, b_n) = (a \cdot b_1, \ldots, a \cdot b_n)$$

Finally we can prove that $\mathbb{K}^n$ is a vector space. It takes up 81 lines, but we ommit the proof:

```
lemma
  vector_space_K_n:
  shows "vector_space R (K_n n) (op ⊙)"
  unfolding K_n_def
proof (intro vector_spaceI)
  ...
qed
```

13.2 Canonical basis

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 = ((λi. if i = j then 1 else 0), n - 1)"
```

This function return us the $j-th$ vector of the canonical basis in dimension n. For example, the three vectors of the canonical basis of $\mathbb{K}^3$ are $(1, 0, 0)$, $(0, 1, 0)$ and $(0, 0, 1)$ and we can obtain them with *x_i 0 3*, *x_i 1 3* an *x_i 2 3* respectively. We prove some properties of these elements:
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 $0_{\mathbb{K}^n}$.

```
lemma x_i_ne_zero:
  shows "x_i i n ≠ 0_K_n n"
proof (rule ccontr, simp)
  assume eq: "x_i i n = 0_K_n n"
  have "fst (x_i i n) i = 1"
    unfolding x_i_def by simp
  moreover have "fst (0_K_n n) i = 0"
    unfolding K_n_def K_n_zero_def by force
  ultimately show False using eq by simp
qed
```

We now prove that the set which is composed by one element of the canonical basis is linearly independent:

```
lemma x_i_li:
  assumes j_l_n: "j < n"
  shows "vector_space.linear_independent R (K_n n) (op ⊙)
  {(x_i j n)}"
proof (unfold vector_space.linear_independent_def [OF
vector_space_K_n], intro conjI)
...
end
```

We prove the same property but considering the set composed by any two different elements of the canonical basis:

```
lemma x_i_x_j_li:
  assumes j_l_n: "j < n"
  and i_l_n: "i < n"
  and i_ne_j: "i ≠ j"
  shows "vector_space.linear_independent R (K_n n) (op ⊙)
  {(x_i i n), (x_i j n)}"
proof -
  ...
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 `(λi. 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 ⊆ carrier (K_n j)"
```

In addition, we can prove that the $k - th$ element of the canonical basis is not in the set created by the function *canonical_basis_acc* j n whenever $j < k < n$.

```
lemma
  canonical_basis_acc_insert:
```

```
  assumes j_l_k: "j < k"
  and k_l_n: "k < n"
  shows "x_i k n ∉ 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
```

This lemma claims that the cardinality of the set created by the function *canonical_basis_acc* k n is $k + 1$.

```
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
```

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"
```

From this definition and using the previous results, we can prove that the canonical basis is contained in the carrier of $\mathbb{K}^n$ and its cardinality is n. The following lemmas are 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

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
```

There exists more properties and facts that the canonical basis satisfies. We want to proof that $\{x\ i\ n | i \in \{0, \ldots, n-1\}\} = canonical\_basis\_K\_n$. In order to do the first implication ($\subseteq$), we present the following two lemmas: the $n-th$ element of the canonical basis is in the canonical basis (obviously) and that the canonical basis is a good set. We present the wording but not the proofs (we have make them by cases and they are not interesting to be explained). The other implication ($\supseteq$) is proved later (and named *canonical_basis_acc_K_n*).

```
lemma
  canonical_basis_K_n_elements:
  assumes  j_in_n: "j ∈ {..<n}"
  shows "x_i j n ∈ canonical_basis_K_n n"
proof (cases n)
...
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)
...
qed
```

Now we make a brief disgression: we need to prove the following lemma which is a generic version of the theorem `finsum_cong`:

```
⟦A = B; (f ∈ B → carrier G) = True; ⋀i. i ∈ 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: "(⋀x. x ∈ 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 ∘ h) B"
    by (rule finsum_cong, simp_all add: g) (rule eq)
  also have "... = (⨁x∈B. f (h x))"
  proof (rule finsum_cong)
    show "B = B" ..
    show "⋀i. i ∈ B =simp=> (f ∘ h) i = f (h i)" by simp
    show "(f ∘ h ∈ B → carrier G) = True"
      using bij_betw_imp_funcset [OF bb] using f by auto
  qed
  also have "... = finsum G f (h ‘ B)"
  proof (rule finsum_reindex [symmetric])
    show "finite B" by fact
    show "f ∈ h ‘ B → 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 ‘B = A" using bb unfolding bij_betw_def by fast
    show "(f ∈ A → carrier G) = True" using f by fast
    show "⋀i. i ∈ A =simp=> f i = f i" by simp
  qed
  finally show ?thesis by simp
qed
```

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 ∈ carrier (K_n n)"
shows "x = (λi. if i ∈ {..<n} then fst x i else 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 ∈ carrier (K_n (Suc k))"
shows "((λn. if n ∈ {..<k} then fst x n else 0), k - 1)
∈ carrier (K_n k)"
using x
unfolding K_n_def K_n_carrier_def ith_def vlen_def 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: "∀ i∈{..<n}. x i ∈ carrier R"
and n_le_k: "n ≤ k"
shows "(λi. if i ∈ {..<n} then x i else 0, k - 1) =
finsum (K_n k) (λi. x i ⊙ 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: "∀i∈{..<n}. x i ∈ carrier R" and prem2: "n ≤ k"
    and x_n: "x n ∈ carrier R"
    and hypo: "(λi. if i ∈ {..<n} then x i else 𝟬, k - 1)
    = (⨁K_n ki∈{..<n}. x i ⊙ x_i i k)"
    using Suc.prems Suc.hyps by simp_all
  show ?case
  proof -
    have "(⨁K_n ki∈{..<Suc n}. x i ⊙ x_i i k)
      = (⨁K_n ki∈(insert n {..<n}). x i ⊙ x_i i k)"
      unfolding lessThan_Suc ..
    also have "... = (x n ⊙ x_i n k)
      ⊕K_n k (⨁K_n ki∈{..<n}. x i ⊙ x_i i k)"
    proof (rule abelian_monoid.finsum_insert
        [OF abelian_monoid_K_n])
      ...
    qed
    also have "... = (x n ⊙ x_i n k)
      ⊕K_n k (λi. if i ∈ {..<n} then x i else 𝟬, k - 1)"
      unfolding Suc.hyps [symmetric, OF prem prem2] ..
    also have "... = (λi. if i = n then x n else 𝟬, k - 1)
      ⊕K_n k (λi. if i ∈ {..<n} then x i else 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 "... =  (λi. (if i = n then x n else 𝟬)
      ⊕ (if i < n then x i else 𝟬), k - Suc 0)"
      unfolding K_n_def K_n_add_def ith_def by simp
    also have "... = ((λi. if i < (Suc n) then x i else 𝟬), k - 1)"
    proof (rule, intro conjI)
      ...
    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: "∀i∈{..<n}. x i ∈ carrier R"
  shows "(λi. if i ∈ {..<n} then x i else 𝟬, n - 1) =
  finsum (K_n n) (λi. x i ⊙ x_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 ∈ carrier (K_n n)"
  shows "x = finsum (K_n n) (λj. fst x j ⊙ x_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
```

13.3 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)`

The first lemma claims that if we have an element x in the set created by the function *canonical_basis_acc* k n (where $k < n$) hence there exists a natural number j less than $k + 1$ such that x is the $j - th$ element of the canonical basis.

```
lemma canonical_basis_acc_eq_x_i:
  assumes x: "x ∈ canonical_basis_acc k n"
  and k_l_n: "k < n"
  shows "∃j∈{..<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 ∈ canonical_basis_acc k n"
      and hypo: "∃j∈{..<(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
```

Using the previous lemma, we can prove that if $x \in canonical\_basis\_K\_n\ n$ then x is one element produced by the function $x\_i\ j\ n$ (for a determinated $j < n$). In other words, is an element of the canonical basis. As a corollary we can prove that j is unique.

```
lemma
  canonical_basis_is_x_i:
  assumes x: "x ∈ canonical_basis_K_n n"
  shows "∃j∈{..<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 ∈ canonical_basis_K_n n"
  shows "∃!j∈{..<n}. x = x_i j n"
proof -
  obtain j :: nat where j: "j ∈ {..<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 ∈ {..<n}" by fact
    show "x = x_i j n" by fact
    fix ja
    assume ja: "ja ∈ {..<n} ∧ 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)"
```

We present some properties of this function and its relationship with x_i. Next two lemmas show that both functions are inverse of each other:

```
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
```

Note that the previous function will be later used to define the bijection between *canonical_basis_K_n* and X, mapping $x\_i\ j\ n$ (in $\mathbb{K}^n$) to x_j (in V).

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}`. It returns us the $n-th$ vector of the canonical basis of $\mathbb{K}^n$.

```
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 .
```

We can prove that there exists a bijection between the image by *iso_nat_can* of the set $\{.. < dimension\}$ and *canonical_basis_K_n (dimension)*:

```
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 (λi. 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 "(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
```

13.4 Properties of Canonical Basis

Here we present more properties and facts that *canonical_basis_K_n n* satisfies. 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. We will make the proof using the lemma *finsum_cong"* proved in previous section. We ommit it (it takes up 80 lines).

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)"])*
...
qed

The space generated by the *vector_space.span* of *canonical_basis_K_n n* is equal to the vector space *K_n n*. The complete proof takes up 120 lines.

lemma
span_canonical_basis_K_n_carrier_K_n:
shows *"vector_space.span R (K_n n) (op ⊙)*
(canonical_basis_K_n n) = carrier (K_n n)"
proof
interpret *vector_space R "K_n n" "op ⊙"* **using** *vector_space_K_n .*
show *"span (canonical_basis_K_n n) ⊆ carrier (K_n n)"*
...
show *"carrier (K_n n) ⊆ span (canonical_basis_K_n n)"*
...
qed

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 ⊙)*
(canonical_basis_acc j n)"
proof -
...
qed

As a corollary, the set of all elements in the canonical base is *linearly_independent_ext*.

lemma

```
canonical_basis_K_n_linear_independent_ext:
shows "vector_space.linear_independent_ext R (K_n n) (op ⊙)
(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`, and, moreover, that it has a finite basis with cardinality n.

lemma

```
canonical_basis_K_n_basis:
shows "vector_space.basis R (K_n n) (op ⊙)
(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
```

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 (i.e., $\sum_{i_1}^{n} \alpha_i \cdot_V x_i$).

The theorem `obtain_indexing`: `finite A` $\Longrightarrow \exists$ `f. indexing (A, f)` and the premise that the vector space is finite, and so is its 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 "(λf. indexing (X, f))"]*
unfolding *indexing_X_def* **by** *auto*

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*. This function will return the $n-th$ element of the basis *X* of V.

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* $\leq$ *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)"

There exists a bijection between the image by the function *indexing_X* of the set $\{.. < dimension\}$ and X.

```
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 "(λf. bij_betw f {..<dimension} X)"]
    unfolding indexing_X_def indexing_def dimension_def by simp
qed
```

As a corollary we can obtain that if $x \in X$ hence there exists only one $j \in \{.. < dimension\}$ such that $x = indexing\_X\ j$:

```
lemma
  indexing_X_preimage:
  assumes x: "x ∈ X"
  shows "∃j. j ∈ {..<dimension} ∧ x = indexing_X j"
proof
...
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 ∈ {..<dimension}" by fact
    show "x = indexing_X j" by (rule x)
    fix ja
    assume ja: "ja ∈ {..<dimension} ∧ x = indexing_X ja"
    show "ja = j"
      using x j ja indexing_X_bij
      unfolding bij_betw_def
```

```
    by (metis inj_onD)
  qed
qed
```

With the next five lemmas we can prove that 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 ∈ X"
  shows "x = indexing_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 ∈ 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 ∈ 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 ∈ {..<dimension}"
  shows "preim2 (iso_nat_X n) = n"
proof -
  have i: "iso_nat_X n ∈ 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 ∈ {..<dimension}"
  shows "preim2_comp (iso_nat_X n) = n"
proof -
  have i: "iso_nat_X n ∈ 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 ∈ {..<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
  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
  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
```

In the previous section we proved that there exists an isomorphism between $\{.. < dimension\}$ and *canonical_basis_K_n n*. Here we have proved the existence of an isomorphism between $\{.. < dimension\}$ and X. We will later compose the isomorphism through $\{.. < dimension\}$ to build the isomorphism from X to *canonical_basis_K_n n* (i.e., between the basis of V and

$\mathbb{K}^n$).

13.5 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*.

We need another vector space in addition to V. We will call this additional vector space W (it will be also over $\mathbb{K}$). We will try to implement definition 13.0.5 of linear map in Isabelle/HOL. First of all we make the definition of the locale for *linear map* (because we need to fix two vector spaces over a field) and after that we will define it.
The next definition 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 the first locale, demanding long names for the first locale constants. We do not know how to control this behaviour.

```
locale linear_map' = KV: vector_space K V f + KW: vector_space K W g
  for K (structure) and V (structure) and W (structure) and f
(infixl "·V" 60) and g  (infixl "·W" 60)
```

Thus, we preferred the long version, in which locale interpretation has to be done later by hand. We present the definition that we finally will use:

```
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 "·V" 70)
  and scalar_product2 :: "'a => 'e => 'e" (infixr "·W" 70)
  assumes V: "vector_space K V (op ·V)"
  and W: "vector_space K W (op ·W)"
```

In this definition we are fixing three algebraic structures: $\mathbb{K}, V$ and W and two scalar products (one over V and another over W). The structures V and W together with its scalar product will be vector spaces over $\mathbb{K}$.

Linear maps, as characterised in [11], have to satisfy the additivity and homogeneity properties:

```
definition additivity :: "('c => 'e) => bool"
  where "additivity T
  = (∀x∈carrier V. ∀y ∈ carrier V. T (x ⊕V y) = T x ⊕W T y)"

definition homogeneity :: "('c => 'e) => bool"
  where "homogeneity T
  = (∀k∈carrier K. ∀x∈carrier V. T (k ·V x) = k ·W T x)"

definition linear_map :: "('c => 'e) => bool"
  where "linear_map T = (additivity T ∧ homogeneity T)"
```

With this last definition, we have implemented the concept of *linear map*. 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 ·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. Interpretation is a mechanism to import every result and definition provided in a locale to a given instance. In our setting, we have proved that *K_n n* is a vector space, and thus we can produce such interpretation (and some others).

```
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 "Vector_Space_K_n.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
```

13.6 Defining the isomorphism between $\mathbb{K}^n$ and V

Firstly we prove a theorem similar to *unique_coordenates*:

⟦*x* ∈ *carrier V*; *f* ∈ *coefficients_function (carrier V)*; *x* = *linear_combination f X*; *g* ∈ *coefficients_function (carrier V)*; *x* = *linear_combination g X*⟧ ⟹ ∀*x*∈*X*. *g x* = *f x*. It claims that the coordinates are unique in a basis.

```
lemma
  linear_combination_unique:
  assumes x: "x ∈ carrier V"
  shows "∃!f. f ∈ coefficients_function X
  & linear_combination f X = x"
proof -
  obtain f_cf
  where cf_fc: "f_cf ∈ 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 == "(λx. if x ∈ X then f_cf x else 0)"
  have cf: "f ∈ coefficients_function X"
    and lc: "linear_combination f X = x"
    ...
  show ?thesis
  proof (rule ex1I [of _ f])
    show "f ∈ coefficients_function X
      & linear_combination f X = x" using cf lc ..
    fix g
    assume "g ∈ coefficients_function X
      & linear_combination g X = x"
    hence cfg: "g ∈ coefficients_function X"
      and lcg: "linear_combination g X = x" by fast+
    ...
    show "g = f"
    ...
    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`. We define this function f as the *lin_comb* of a given x.

```
definition lin_comb :: "'c => ('c => 'a)"
  where "lin_comb x = (THE f. f ∈ coefficients_function X
  ∧ linear_combination f X = x)"
```

The mathematical wording of the previous function is the function that for each x and a given basis $X = \{x_1, \ldots, x_n\}$ such that $x = \sum_{i=1}^{n} \alpha_i \cdot_V x_i$, the function *lin_comb* returns the coefficients $\alpha_1 \ldots \alpha_n$.
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 ∈ carrier V"
  shows "x = finsum V (λi.
  lin_comb x (indexing_X i) · 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 · y) X"
    unfolding linear_combination_def ..
  also have "... = finsum V (λi.
  lin_comb x (indexing_X i) · 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 · y) ∈ X → carrier V"
    proof
      ...
    qed
    show "(λi. lin_comb x (indexing_X i) · indexing_X i)
    ∈ {..<dimension} → carrier V"
    proof
      ...
```

```
    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 -
      ...
    qed
  qed
qed
```

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) (λi. (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. It is a long proof of 150 lines.

```
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)
    ...
  qed
  show "homogeneity iso_K_n_V"
  proof (unfold homogeneity_def, rule ballI, rule ballI)
    ...
  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 ∈ carrier V"
  and y: "y ∈ carrier V"
  shows "lin_comb (x ⊕V y) = (λi. lin_comb x i ⊕ lin_comb y i)"
  apply (subst lin_comb_def)
proof (rule the1_equality)
  ...
qed
```

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 ∈ carrier K"
  and x: "x ∈ carrier V"
```

```
  shows "lin_comb (k · x) = (λi. k ⊗ lin_comb x i)"
  apply (subst lin_comb_def)
proof (rule the1_equality)
  show "∃!f. f ∈ coefficients_function X
  ∧ linear_combination f X = k · x"
    using linear_combination_unique [OF mult_closed [OF x k]] .
next
  show "(λi. k ⊗ lin_comb x i) ∈ coefficients_function X ∧
    linear_combination (λi. k ⊗ lin_comb x i) X = k · x"
  proof (rule conjI)
    show "(λi. k ⊗ lin_comb x i) ∈ coefficients_function X"
      using lin_comb_is_coefficients_function [OF x]
      unfolding coefficients_function_def
      using k by auto
    show "linear_combination (λi. k ⊗ lin_comb x i) X = k · x"
    proof -
      have "linear_combination (λi. k ⊗ lin_comb x i) X =
        (⨁ V y∈X. (k ⊗ lin_comb x y) · y)"
        unfolding linear_combination_def ..
      also have "... = k · (⨁ V y∈X. (lin_comb x y) · 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 · x"
        unfolding linear_combination_def [symmetric]
        unfolding lin_comb_is_the_linear_combination
        [symmetric, OF x] ..
      finally show ?thesis .
    qed
  qed
qed
```

The following lemma proves that the application *iso_V_K_n* is a linear map between *V* and *field.K_n K dimension*. Is a long lemma of 150 code lines.

```
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)
    ...
  qed
  show "V_K_n.homogeneity iso_V_K_n"
  proof (unfold V_K_n.homogeneity_def, rule ballI, rule ballI)
    ...
  qed
qed
```

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 "(λi. 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 "(⨁V i∈{..<dimension}. fst x i · indexing_X i)
      ∈ carrier V"
      proof (rule finsum_closed)
        show "finite {..<dimension}" by simp
        show "(λi. fst x i · indexing_X i)
        ∈ {..<dimension} → 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
```

Thanks to the bijection that exists between the image of $\{.. < dimension\}$ by the function *indexing_X* and X we can prove the following equivalence of the finite sum of the elements of the basis:

```
lemma
  lin_comb_fimsum_candidate:
  assumes x: "x ∈ carrier (K_n dimension)"
  shows "(⨁V y∈X. fst x (preim2_comp y) · y) =
(⨁V i∈{..<dimension}. fst x i · indexing_X i)"
proof (rule finsum_cong'' [of _ "indexing_X"])
```

```
  show "finite {..<dimension}" 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 ∈ carrier (K_n dimension)"
  shows "lin_comb (⨁V i∈{..<dimension}. fst x i · indexing_X i) =
(λy. fst x (preim2_comp y))"
  unfolding lin_comb_def
proof (rule the1_equality)
  ...
qed
```

With the previous lemmas, we can now prove that *iso_V_K_n* is a bijection between the correspoding carrier sets (not a proper isomorphism, which also requires linearity, previously proved):

```
lemma iso_V_K_n_bij: shows "bij_betw iso_V_K_n (carrier V)
(carrier (K_n dimension))"
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 ∈ carrier V → carrier (K_n dimension)"
    by (rule iso_V_K_n_Pi)
  show "iso_K_n_V ∈ carrier (K_n dimension) → carrier V"
    by (rule iso_K_n_V_Pi)
  fix x assume x: "x ∈ carrier V"
  show "iso_K_n_V (iso_V_K_n x) = x"
    ...
next
  fix y
  assume y: "y ∈ carrier (K_n dimension)"
  show "iso_V_K_n (iso_K_n_V y) = y"
  proof -
    ...
  qed
qed
```

The following lemma should not be needed, since the bijection has been

already proved and it is bidirectional, but we present it for completeness.

```
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 ∈ carrier V → carrier (K_n dimension)"
    by (rule iso_V_K_n_Pi)
  show "iso_K_n_V ∈ carrier (K_n dimension) → carrier V"
    by (rule iso_K_n_V_Pi)
  fix x assume x: "x ∈ carrier V"
  show "iso_K_n_V (iso_V_K_n x) = x"
    ...
next
  fix y
  assume y: "y ∈ carrier (K_n dimension)"
  show "iso_V_K_n (iso_K_n_V y) = y"
  proof -
    ...
  qed
qed
```

Now we can implement the notion of isomorphism between two vector spaces in Isabelle/HOL following the definition 13.0.6 presented in the beginning of this chapter:

```
definition vector_space_isomorphism :: "('c => 'e) => bool"
  where "vector_space_isomorphism f
  == bij_betw f (carrier V) (carrier W) ∧ linear_map f"
```

Finally, the two following lemmas state the isomorphism (in both directions actually) between `field.K_n K dimension` and `V` (i.e., between $\mathbb{K}^n$ and V with n the cardinality of a basis of 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*

Chapter 14

Subspaces

In this chapter we present the concept of *subspace* (of a vector space) and some properties about it. Firstly, we show the definition that appears in Halmos:

Definition 14.0.1 *A non-empty subset M of a vector space V is a* subspace *(or a* linear manifold*) if along with every pair, x and y, of vectors contained in M, every linear combination $\alpha x \oplus_V \beta y$ ($\alpha, \beta \in \mathbb{K}$) is also contained in M.*

We can write literally this definition in Isabelle/HOL:

```
definition subspace :: "'b set => bool"
  where "subspace M == ((M ⊆ carrier V) ∧ M ≠ {}
  ∧ (∀α∈carrier K. ∀β∈carrier K. ∀x∈M. ∀y∈M.
  α · x ⊕V β · y ∈ M))"
```

As a subspace is not empty, hence it contains some element x. Due to the subspace definition, it also contains $x \ominus_V x$ and hence 0_V:

```
lemma
  zero_in_subspace:
  assumes s: "subspace M"
  shows "0V ∈ M"
proof -
  obtain x where x: "x ∈ M" using s
    unfolding subspace_def by fast
  hence xV: "x ∈ carrier V"
    using s unfolding subspace_def by fast
  have one: "1K ∈ carrier K"
```

```
    and minus_one: "⊖ 1_K ∈ carrier K" by simp+
  hence "1_K · x ⊕_V (⊖ 1_K · x) ∈ 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
```

We can also prove one important property: a subspace M in a vector space V is itself a vector space.

```
lemma
  subspace_is_vector_space:
  assumes s: "subspace M"
  shows "vector_space K (V⦇carrier:= M⦈) (op ·)"
proof (unfold_locales, auto)
  show "0_V ∈ M"
    by (metis assms zero_in_subspace)
  fix x and y and z
  assume x_in_M: "x ∈ M"
    and y_in_M: "y ∈ M"
    and z_in_M: "z ∈ M"
  hence x_in_V: "x ∈ carrier V"
    and y_in_V: "y ∈ carrier V"
    and z_in_V: "z ∈ carrier V"
    by (metis assms mem_def subsetD subspace_def)+
  show "x ⊕_V y ∈ M"
  proof -
    ...
  qed
  show "0_V ⊕_V x = x"
    by (metis V.add.l_one assms mem_def
      subsetD subspace_def x_in_M)
  show "x ⊕_V 0_V = x"
    by (metis V.add.r_one assms mem_def
      subsetD subspace_def x_in_M)
  show "x ⊕_V y = y ⊕_V x"
    by (metis V.a_comm x_in_V y_in_V)
  show "x ⊕_V y ⊕_V z = x ⊕_V (y ⊕_V z)"
    using a_assoc[OF x_in_V y_in_V z_in_V] .
  show "1 · x = x" using mult_1[OF x_in_V] .
```

```
  show "x ∈ Units (|carrier = M, mult = op ⊕V, one = 0V|)"
  proof -
    ...
  qed
  fix a and b
  assume a_in_K: "a ∈ carrier K" and b_in_K: "b ∈ carrier K"
  show "(a ⊗ b) · x = a · b · x"
    using mult_assoc[OF x_in_V a_in_K b_in_K] .
  show "a · x ∈ M"
  proof -
    ...
  qed
  show "a · (x ⊕V y) = a · x ⊕V a · y"
    using add_mult_distrib1[OF x_in_V y_in_V a_in_K] .
  show "(a ⊕ b) · x = a · x ⊕V b · x"
    using add_mult_distrib2[OF x_in_V a_in_K b_in_K] .
qed
```

Now we show two examples of subspaces: the set $\{0_V\}$ and the whole space:

```
lemma
  subspace_zero:
  shows "subspace {0V}"
  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 ∈ S" and y: "y ∈ S"
  shows "x ⊕V y ∈ S"
proof -
```

```
  have xv: "x ∈ carrier V" and yv: "y ∈ carrier V"
    using x y s unfolding subspace_def by auto
  have "x ⊕_V y = 1 · x ⊕_V 1 · 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
```

Now we show that a subspace is closed under finite sums (so under linear combinations). First we present the proof in case that we are in a finite subspace:

```
lemma subspace_finsum_closed:
  assumes s: "subspace S"
  and f: "finite S"
  and y: "Y ⊆ S"
  and c: "f ∈ Y → carrier K"
  shows "finsum V (λi. f i · i) Y ∈ 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: "(⨁_V i∈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
```

Here the proof in case that the subspace is not finite but sums are over a finite subset:

```
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)
  ...
qed
```

As a corollary we can obtain that a linear combination of elements of a subspace is in the subspace:

```
corollary subspace_linear_combination_closed:
  assumes s: "subspace S"
  and f: "finite Y"
  and y: "Y ⊆ S"
  and c: "f ∈ coefficients_function Y"
  shows "linear_combination f Y ∈ S"
  proof (unfold linear_combination_def,
```

```
      rule subspace_finsum_closed')
  show "subspace S" using s .
  show "finite Y" using f .
  show "Y ⊆ S" using y .
  show "f ∈ Y → carrier K"
    using c unfolding coefficients_function_def by blast
qed
```

Chapter 15

Future Work

In the previous chapters we have shown the implementation in Isabelle/HOL of the theorems presented in the first ten sections in Halmos. Once we have finished them, we can present two lines of future work.
The first one is that it would be desirable to continue with the development of the following sections in Halmos in order to achieve a good formalization of the main results of linear algebra. In fact, the next five sections presented in Halmos (sections 11 to 15) were going to be part of this work initially. As we have said in the introduction, at first we had the objective of formalizing that a vector space is isomorphic to the dual of its dual. We have not achieved this result in time, but we have made the wording of the theorems and some proofs (but not all) of these 5 sections can be found in: [2]. Furthermore, a good idea would be to continue with the development, not only up to complete such proof but at least up to the end of the chapter 1 of Halmos.
The second line of future work is to proof properties and facts of more abstract algebra. Once we have proved that every finite-dimensional vector space of dimension n is isomorphic to $\mathbb{K}^n$, we could concentrate ourselves in the study of matrices representing linear maps and transformations of vector spaces. The study of some diagonalization algorithms could be also interesting for the aims of the ForMath project. More specifically, the algorithm computing the Smith normal form, which enables the computation of the homology group of a chain complex, would be a very interesting result.

Chapter 16

Conclusions

Once we have finished this project we can say some things about the development of a formalization in Isabelle/HOL.
First comment that we have to say is that its learning curve is hardsteep, much more than a programming language. As we have started from scratch the main difficulty of this development at first was to understand and know to cope with the proofs in Isabelle/HOL. I have never made nothing on formalization before, I have never seen a formalized proof in Isabelle (neither in any other theorem proving environment). However, I suspected how they could be: one should reduce and cut up the proof into very small steps.
The first steps were hard, we tried to formalize easy proofs but we couldn't. If I see them now I would say that I was clumsy, but in those first periods to write correctly a proof of 4 or 5 lines took me a lot of hard work. I managed them in the end, but not without a struggle.
In fact, the first conclusion that I draw is that Isabelle was not as clever as I thought: the proof methods that Isabelle has (simp, auto ...) are not self-sufficient to prove something not trivial: one has to usually provide explicitly the candidates for the existentials, other theorems and results, premises ... At first I expected that *by auto* would be able to look for in the results that I had demonstrated inside of the proof, look for theorems in the library and sort them out the best it could to prove the result, that it is to say, something similar than sledgehammer [39] to the task accomplished by.
After those first steps, I began to be at ease with the proofs, learning and familiarizing with the language. Nevertheless, it was not all achieved: now I was able to make easy proofs but either they would take up several lines (we could make it shorter) or I wrote them not clearly: a reader was going

to have to pay attention with detail in the code to understand what I have done (when, for example, doing them with calculations we can see with a quick look the reasoning followed).
I improved gradually and I was bumped into proofs each time more complicated that required demonstrations by induction (see chapter 10) or theorems that at first were not easy to state due to the order of the sets (10.2.1). This last matter was the first big problem where I was stuck and it derived to result in the theory of indexed sets 10.1. Before developing this theory we tried several attempts to tackle this proof (and in general, the development that was after it) without giving an order to a set. Our first main objective was to prove that a linearly independet set could be extended to a basis (11.2.1). We did three attempts to demonstrate it before deciding to introduce indexed sets in Isabelle/HOL. As we already said in 10.2, it took up several code lines which finally were not useful and we counted with ideas of Julio Rubio and Tobias Nipkow. We had to reject them because we found that we were defining functions which were not commutative, and finally we followed the development of the book, although it involved a great deal of work and made a sudden stop to generate the theory of indexed sets.
The difficulties did't finished here, because at that moment we found proofs that demanded reasoning over complex iterative algorithms applied to indexed sets and we had to define functions (*remove_ld*), iterate them (*iterate_remove_ld*) and shown that they verified the required properties using a special kind of induction (see chapter 11).
We had even to define later the concept of the power of a function (see chapter 12). This is something that surprised us a lot because it is something very useful and that appears in several books, it is a basic concept of mathematics.
In conclusion, not every basic definition and concept of math is implemented in Isabelle/HOL. In addition, there are notions which aren't implemented exactly as we understand them in maths, for example the cardinality of a set: Isabelle gives 0 as cardinality to an infinite set. In some cases this could be an impediment to prove some result, for example an easy result: how could we prove that a finite set has less cardinality than an infinite set? We can't.
Besides the difficulty of the proofs in Isabelle, another conlusion is that they take up much more lines in comparation to the same proofs in a book. In general, in a book they are not broken up step by step, several steps are made at the same time or even there are trivial cases that in a book are omitted (but in Isabelle we have to write all). There exists a rule that claims that a

line of a book should be formalized in four lines in Isabelle [15]. However, we couldn't fulfill it mainly because the algorithmic reasoning required functions which must verify properties hard to be proved and in addition we had to develop a whole theory to give orders to sets. Moreover, the notions of $\mathbb{K}^n$ or canonical basis are concepts easy to define in paper but laborious in Isabelle. In fact, the isomorphism between $\mathbb{K}^n$ and an n-dimensional vector space V can be understood with a diagran with a quick look (see chapter 13), but formalizing it takes some time.
The development of the ten first sections in Halmos (up to subspaces) took up a total of 12387 lines, distributed of the following way:

File .thy	**Number of lines**
Previous	55
Field2	326
Vector_Space	42
Examples	57
Comments	329
Linear_dependence	532
Linear_combinations	1921
Basis	1962
Dimension	2235
Isomorphism	3465
Indexed_set	1226
Subspaces	234
TOTAL	12387

As it can be observed, the first files (which correspond with the first sections in Halmos) are actually short. This is because they contain algebraic proofs. When we ran into proofs in which an algorithmic reasoning is used we had to formalize much more results.
We have spoken about the length of the proofs in a book and in Isabelle. We must say that the ten sections of Halmos that we have formalized take up a total of 17 pages, when our development in Isabelle/HOL takes up about 300 pages. It is an abysmal difference.
I must say that formalizing a proof with a computer is not only something beautiful to be done, but it also helps to understand the proofs better: several times in the books there are omitted details, conditions and cases which are taken into account or are not clear. When you are in front of the computer

formalizing a proof, it is not simply copying it line by line, you have to cut it up into shorter proofs, separate in cases, study it and complete it in order to achieve to finish the demonstration. Several times, when you think that you have all done you realize that not, and you have to think how to break down the proof in order to Isabelle understands it. There are proofs that Halmos says that they are clear (and he doesn't complete) and for us they took up several lines (for example, the result that says that if a dependent set A is contained in B, hence B is also dependent. See chapter 9).
To sum up, learning to formalize proofs is not something quick but it is a slow process that requires its time. One demonstration or concept that appears in a book can look like simple but its formalization could be very long and hard. Another conclusion is that the logic used (HOL) has been able to formalize our development.

16.1 Management conclusions

As we have said before in the introduction and in the management chapter, our first objective was to formalize the first 16 sections in Halmos until proving the theorem that claims that a vector space is isomorphic to the dual of its dual. At first, this project was going to be developed from November of 2010 to June of 2011 but it got longer up to September. We achieved the formalization of the first 10 sections. We have already said that those 10 sections take up 17 pages in Halmos and our development of them take up about 300. As a conclusion we can say that we underestimate the difficulty and the length of the project: there were problems that we have already anticipated but we couldn't expect that they were going to be so hard. And, as we have already said before, there appeared theorems which were in a quick look simple but their formalization needed a lot of time in Isabelle/HOL.
In any way, we can say that we have demonstrated very important theorems, such that every linearly independent set can be extended to a basis, two bases have the same cardinality or a n-dimensional vector space over a field $\mathbb{K}$ is isomorphic to $\mathbb{K}^n$. Thanks to this last result, we can ensure that the first objective (prove that a vector space is isomorphic to the dual of its dual) is feasible and with more effort and time can be proved as a future work.
Taking into account that we haven't finished the project in June of 2011 (after 8 months of work with 2-3 hours by day, which makes a total of about 450 hours) we decided to follow up to September fulltime (about 300 hours

more if we don't count holidays). It is clear that there wasn't a realistic planning: we worked more hours than we expected at first and we didn't formalize completely the first 16 sections.
There exist several reason which slowed down our development in such a way we don't reach what we want. As we have said before, this project is closer to a research project than a common degree's dissertation, and so the delay arrives due to the unavoidable uncertainty which involves a project of this kind.
First of all we have to say that we have not only proved the properties and results presented in Halmos, but another ones which don't appear in it but are crucial (and not for this reason short) and also useful in several of our demonstrations. Another concepts have been implemented, as the span or the extended definition of linear independence, dependence and spanning set which only appear named in the book. This work was necessary in order to obtain a correct and complete formalization of vector spaces in Isabelle/HOL.
Nevertheless, the main reason of the delays were the difficulties that we find during the development.
The first high block that we had was while we was trying yo prove the theorem 10.2.1. As we have explained in the documentation of the project, we looked for alternative ideas and another proofs without resorting to the ordenations and the indexed sets 10.1. This theorem is a result that we had already expected to be hard, but we didn't expected to be so blocked in it. Finally, and after several attempts in vain in which we spent lot of time and code lines that finally we rejected, we had no other choice to implement the indexed sets with the delay that it means.
The second reasong was that the iterative (algorithmic) reasonings that were made in the proofs of 11.2.1 and 12.1.2 was more difficult of implementing than we had expected. We have shown in the documentation a summary of the results that we have had to prove until formalizing the theorem.
In addition, in the theorem 12.1.1 we made a mistake with the definition of the swap function: we didn't take into account the separation in cases (Halmos doesn't say anything about a multiset when we make the union in the proof) and we can only prove that two disjoint basis had the same cardinality, but not the result. We had to redo our work giving a step backwards and changing the definition in order to formalize the theorem.
Another example of difficulties: the isomorphism in chapter 13, which takes up a page in the book, it has turned into the longer proof of our development.
Furthermore, writting the documentation in English is not simple, I had

never written so much text in English and I didn't use it in a habitual way since several years, so the beginning was hard. In addition, it is clear that a person doesn't have the same fluency writing in English than in his mother tongue. In fact, at first it was not planned to make it in English and a great part of the code of our development was written in Spanish. We had to translate it, with the loss of time that it caused. Moreover, I had to catch up with LaTeX because we had never used before.
To sum up, the reasons of they delay in our project are:

- The difficulty to estimate the length of this kind of project.
- Proofs longer and harder than we had expected (for example, to prove that an n-dimensional finite vector space over a field $\mathbb{K}$ is isomorphic to $\mathbb{K}^n$, which in the book takes up only one page, we had to develop the file ISOMORPHISM.THY which takes up 3500 code lines, i.e. about 100 pages).
- Need to create the theory of indexed sets.
- Documentation and code in English.
- Need to learn to write in LaTeX.

Bibliography

[1] P. Halmos. *Finite-dimensional vector spaces.* Springer, 1974.

[2] http://www.unirioja.es/cu/jodivaso

[3] Florian Haftmann. *Haskell-style type classes with Isabelle/Isar.* 2011. http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle2011/doc/classes.pdf

[4] Clemens Ballarin. *Tutorial to Locales and Locale Interpretation.* 2010. http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle2011/doc/locales.pdf

[5] Clements Ballarin. *The Isabelle/HOL Algebra Library.* 2010. http://cl-informatik.uibk.ac.at/users/clemens/research/algebra.pdf

[6] http://leccionesdemate.blogspot.com/2009/03/que-es-una-estructura-algebraica.html

[7] G. Bauer, T. Nipkow. ... *Theory Rings of Isabelle/HOL* http://isabelle.in.tum.de/library/HOL/Rings.html

[8] Departamento de Matemáticas, CCIR/ITESM. *Teoría de la Dimensión en Espacios Vectoriales.* 2009. www.mty.itesm.mx/etie/deptos/m/ma95-843/lecturas/l843-44.pdf

[9] http://planetmath.org/encyclopedia/EveryVectorSpaceHasABasis.html

[10] http://afp.sourceforge.net/entries/Cauchy.shtml

[11] Sheldon Axler. *Linear Algebra Done Right.* Springer, 2004.

[12] Gertrud Bauer. *The Hahn-Banach Theorem for Real Vector Space* http://www.cl.cam.ac.uk/research/hvg/isabelle/dist/library/HOL/Hahn_Banach/document.pdf

[13] Formath project http://wiki.portal.chalmers.se/cse/pmwiki.php/ForMath/ForMath

[14] Top 100 theorems in Isabelle. http://www.cse.unsw.edu.au/~kleing/top100/

[15] de Bruijn factor. http://www.cs.ru.nl/~freek/factor/

[16] http://en.wikipedia.org/wiki/Quotient_space_%28linear_algebra%29

[17] Thomas C. Hales. *Formal Proof.*

[18] Roman Murawsky. *The present state of mechanized deduction, and the present knowledge of its limitations.*

[19] Freek Wiedijk. *Formal Proof-Getting Started.*

[20] A. Wiles. Modular elliptic curves and Fermat's Last THeorem, *Annals of Mathematics* **141** (3) 1995, 443-551.

[21] http://www.cs.ru.nl/~freek/100/

[22] Jesús María Aransay Azofra and César Domínguez Pérez. *Demostración asistida por ordenador.*

[23] H. Hudson. No basta con cuatro colores, *La Gaceta de la RSME* **8** (2) 2005, 361-368.

[24] P. J. Miana and N. Romero. La historia de la conjetura de Kepler. In *Contribuciones científicas en honor a Mirian Andrés Gómez*, 367-374. L. Lambán, A. Romero y J. Rubio eds., Servicio de Publicaciones de la Universidad de La Rioja, 2010.

[25] T. Nipkow, L. C. Paulson and M. Wenzel. *Isabelle/HOL: A proof assistant for higher order logic*, vol. 2283, *Lecture Notes in Computer Science*, Springer (2002).

[26] J.C. Blanchette, L. Bulwahn and T. Nipkow. *Automatic Proof and Disproof in Isabelle/HOL.* Fakultät f"u Informatik, Technische Universität München.

[27] M. Wenzel with several contributions. *The Isabelle/Isar Reference Manual.* `http://www.cl.cam.ac.uk/research/hvg/isabelle/dist/Isabelle2011/doc/isar-ref.pdf` 2011.

[28] M. Wenzel and Stefan Berghofer. *The Isabelle System Manual.* `http://www.cl.cam.ac.uk/research/hvg/isabelle/dist/Isabelle2011/doc/system.pdf` 2011.

[29] S. Schulz. *System Description: E 0.81.* In: Basin, D.m Rusinowitch, M. (eds.) IJCAR 2004. LNAI, vol 3097, pp. 223-228. Springer, 2004.

[30] C. Weidenbach. *Combining superposition, sorts and splitting.* Handbook of Automated Reasoning. pp. 1965-2013. Elsevier, 2001.

[31] A. Riazanov and A. Voronkov. *The design and implementation of Vampire.* AI Comm., 91-110. 2002.

[32] C. Barrett, C. Tinello. *CVC3.* LNCS, vol. 4590, pp. 298-302. Springer, 2007.

[33] B. Dutertre and L. de Moura. *The Yices SMT solver (2006).* `http://yices.csl.sri.com/tool-paper.pdf`

[34] L. de Moura, N. Bjørner. *Z3: An efficient SMT solver.* LNCS, vol 4963, pp. 337-340. Springer, 2008.

[35] S. Böhme and T. Nipkow. *Sledgehammer: Judgement Day.* LNAI, vol. 6173, pp. 107-121. Springer, 2010.

[36] Bauer and Wenzel. *Calculational Reasoning revisited – An Isabelle/Isar Experience.*

[37] A. Krauss. *Defining recursive functions in Isabelle/HOL.* `http://isabelle.in.tum.de/doc/functions.pdf`

[38] A. Church. A formulation of the simple theory of types, *The Journal of Symbolic Logic* **5** (2) (1940), 56–68.

[39] J.C. Blanchette. *A User's Guide to Sledgehammer for Isabelle/HOl.* http://www.cl.cam.ac.uk/research/hvg/isabelle/dist/Isabelle2011/doc/sledgehammer.pdf