A Start to the formalization of Linear Algebra in ForTheL

Erik Sturzenhecker, Felix Thiele

March 12, 2020

Contents

1 Introduction

1.1 Naproche

In this project we build a linear algebra library in ForTheL with Naproche-SAD. ForTheL, which stands for Formal Theory Language, is a language that comes close to human language. This reduces many of the initial hurdles people encounter in the formalization of mathematics. Naproche can not only interpret ForTheL texts, but is also backed with a strong automated theorem prover, to which we shall refer to as e-prover. This makes many proofs in our library easier, since some trivial steps can be skipped. At the same time this comes with massive performance issues, since checking even this small library is a time intensive task.

1.2 Results

The complete project can be found under
`https://github.com/Felix-Thiele/linear_algebra_ftl`.
Some of the results given in this library are:
The **definitions** of:

- groups, rings, fields
- vector spaces, subspaces, dual spaces
- homomorphisms, endomorphisms, automorphisms of vector spaces
- lists, linear independence

And the **proofs** of:

- A field is a vector space over itself.
- The linear maps between K-vector spaces V and W form a vector space $\mathrm{Hom}(K,V,W)$.
- If f is linear, $\mathrm{Ker}(f)$ is a subspace.
- If f is linear and $\mathrm{Ker}(f) = \{0\}$, then f is injective.
- Any K-vector space V can be embedded into the double dual space $(V^*)^*$
- The endomorphisms of a K-vector space V form a ring $\mathrm{End}(K,V)$.
- The invertible elements of a ring form a multiplicative group.

2 The Lean File

This formalization is based on the file "vector_space.lean" found under `https://github.com/kckennylau/Lean/blob/master/linear_algebra/vector_space.lean`. It is part of a small Lean library by Kenny Lau containing formalizations of various mathematical topics. Lean is a theorem prover and a programming language based on dependent type theory. There is an extensive library of mathematical Lean texts, the mathlib (`https://github.com/leanprover-community/mathlib`), maintained by Lean users, which is used and build upon in vector_space.lean.

2.1 Mathematical Content

The file vector_space.lean covers the following definitions and statements of linear algebra:

- `field.to_vector_space`: Any field is a vector space over itself.
- `sub_vector_space`: Definition of a vector subsace. Definition and proof of the vector space structure on it.
- `linear_space K V W`: Definition of the set of linear maps between two K-vector spaces V and W. Easy proofs about linear maps.
- `ker`: Definition of the kernel of a linear map. Some small proofs.
- Definition and proof of the vector space structure on `linear_space K V W`.
- Definition of the dual vector space.
- Definition and proof of the ring structure on `linear_space K V V` by taking the function composition as multiplication.
- `invertible K V`: Definition and proof of the (general linear) group structure on the invertible elements of `linear_space K V V`

2.2 Characteristic Features

Due to the simplicity of the treated topics, there is no need for huge lambda terms and the code is fairly easy to understand.

The Lean file uses the following notions (type classes) from the mathlib: `field`, `discrete_field` (a field where equality of elements is decidable),

`add_comm_group`, `vector_space`, `ring`, `group`. The classes `has_zero`, `has_one`, `has_add`, `has_mul`, `has_neg`, `has_inv`, `has_scalar` from the built-in lean library are crucial for the description and handling of algebraic structures in Lean. The following is an example of how algebraic structures are successively defined as type classes in the mathlib, by extending existing type classes:

```
universe u

class has_zero (α : Type u) := (zero : α)
class has_add  (α : Type u) := (add : α → α → α)
class has_neg  (α : Type u) := (neg : α → α)

class add_semigroup (α : Type u) extends has_add α :=
(add_assoc : ∀ a b c : α, a + b + c = a + (b + c))

class add_comm_semigroup (α : Type u) extends add_semigroup α :=
(add_comm : ∀ a b : α, a + b = b + a)

class add_monoid (α : Type u) extends add_semigroup α, has_zero α :=
(zero_add : ∀ a : α, 0 + a = a) (add_zero : ∀ a : α, a + 0 = a)

class add_comm_monoid (α : Type u) extends add_monoid α, add_comm_semigroup α

class add_group (α : Type u) extends add_monoid α, has_neg α :=
(add_left_neg : ∀ a : α, -a + a = 0)

class add_comm_group (α : Type u) extends add_group α, add_comm_monoid α
```

Especially useful for algebraic proofs is the Lean tactic `simp` (simplify). Easy statements like the ones mentioned in 2.1, which help with term rewriting, can be added to this tactic via `@[simp]`:

```
variables {K V W : Type} [discrete_field K] [add_comm_group V]
  [vector_space K V]

@[simp] def ker (A : linear_space K V W) : V → Prop := (λ v, A.T v = 0)

@[simp] lemma map_neg : A.T (-v) = -(A.T v) :=
eq_neg_of_add_eq_zero (by {rw ←[map_add], simp})

@[simp] lemma map_sub : A.T (u-v) = A.T u - A.T v := by simp
```

This allows for omitting detailed chains of equations and instead let `simp` find the proof details automatically:

```
theorem ker_neg (HV : A.ker v) : A.ker (-v) :=
by {unfold ker at *, simp [HV]}
```

```
theorem ker_sub (HU : A.ker u) (HV : A.ker v) : A.ker (u - v) :=
by {unfold ker at *, simp [HU,HV]}
```

Being able to tell lean which statements to use in the `simp` tactic, while everything else has to be directly referenced, gives a performance advantage.

2.3 Adjusting the Code

The latest version of vector_space.lean was from November 2017. Since then, the mathlib has developed to a point where it became incompatible with vector_space.lean. The following adjustments make up for this:

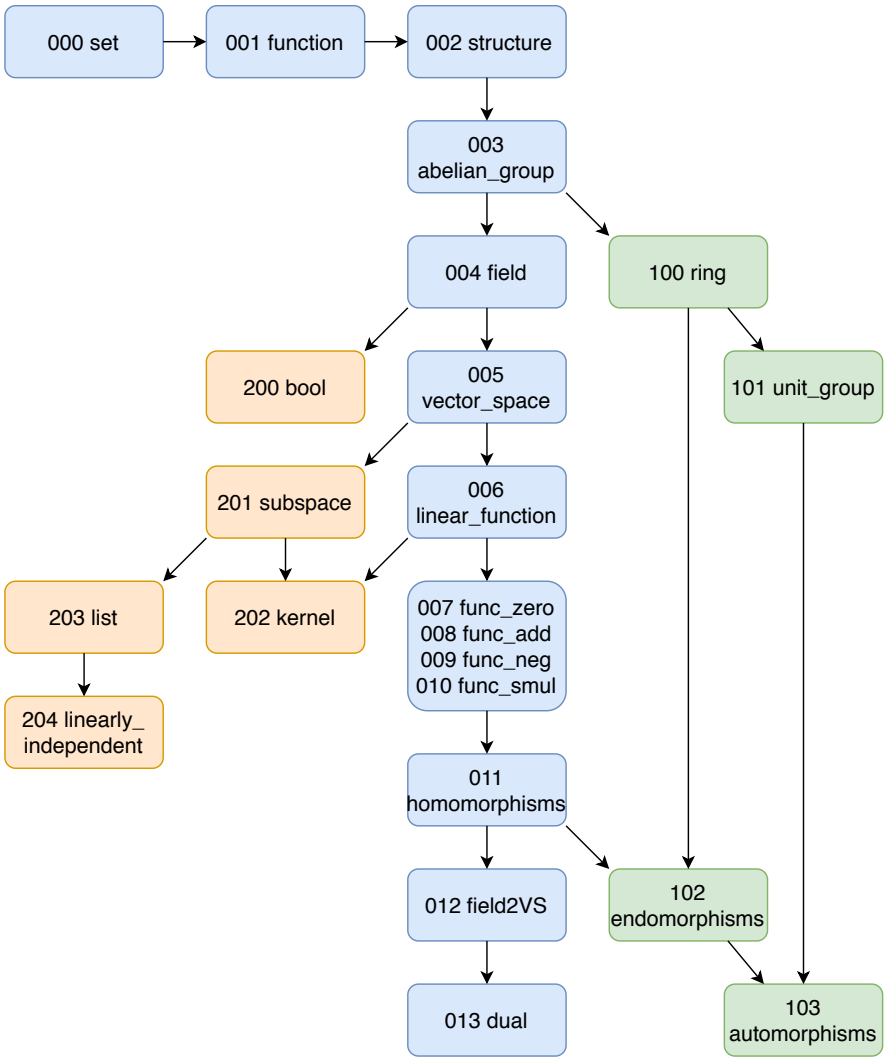
- While the original file used `[field K]` as the only assumption for working with a `vector space K V`, we had to change this to `[discrete_field K]`. Also, the mathlib definition of a vector space now requires that V is already an abelian group, so we added `[add_comm_group V]` as a second assumption at the same places.
- We renamed `subspace` to `sub_vector_space`, since the name conflicted with the (apparently new) mathlib definition of a vector subspace.
- The mathlib change described in the first point also demanded changes in places where objects like the `linear_space K V W` are proven to be a vector space over K: We had to cut parts from the old proof of the object being a vector space and pasting them into a new preceding proof of it being an abelian group.
- We adopted the mathlib renamings of `smul_left_distrib` and `smul_right_distrib` to `smul_add` and `add_smul`, respectively. These are fields of the type class `module`, respectively `vector space` in our case.
- Some minor syntactical changes.

We are able to determine these necessary changes by checking the errors and proof states displayed in the "Lean Goal" window. An important feature of Lean that is missing in Naproche. Our final version of the file can be found under `https://github.com/Felix-Thiele/linear_algebra_ftl/blob/master/vector_space.lean`.

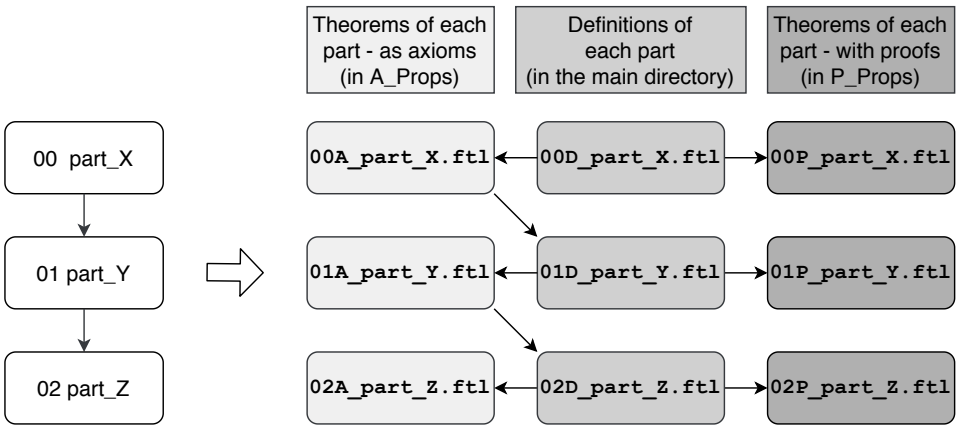
3 Formalization in ForTheL

3.1 Project Structure

Giving the project a treelike structure with a file to each topic ensures increased readability and scalability. In contrast to having one large file building up mathematics, this enables us to pick what files need to be imported to each file. Our structure is depicted in the graph below.



The e-prover is not fast enough to compile the entire library with all proofs in reasonable time. Instead, for every file we introduce two new files inserting "A_" and "P_" respectively in front of the file names. We insert "D_" before our original file. This gives us an Axiom, Proof and Definition file. The Definition file holds all the definitions of the given topic. The Proof file holds the theorems and their proofs. The Axiom file holds all the statements of the Proof file in axiomatized form. The Proof and Axiom files only read their corresponding Definition file, while Definition files read the Axiom files of all the topics they are building upon. This ensures a fast compilation since we do not need to reprove proven statements. This file reading structure is depicted below.



Additionally, it can sometimes be helpful to split the P_ files into multiple P_ files, to decrease checking times. In our library this is the case in P_vector_space, which containes five (partly independent) theorems, each needing some lemmas. Here we split P_vector_space into the files (P_vector_space_1, _2, _3 and _4), each starting with only the required statements of the preceding proof files - in axiomatized form. This is fine for the sake of functionality in a small project like this. In bigger projects we recommend also splitting the Axiom Files.

3.2 The Implementation

3.2.1 Sets and Functions

Our D_ Set and D_ Function files our kept very lightweight and only include key statements. This is due to both files being so close to the root node of the library graph. Adding more definitions will substantially slow down the e-prover in all following files.

3.2.2 Algebraic Structures

Many of the objects we examine in this library have a similar structure. For example, abelian groups, fields, vector spaces and rings all have a carrier, a zero element, and an addition operation. We introduce an object called structure in 002D_structure.ftl.

```
Signature. lang is a set.
Axiom. lang = {carr,zero,one,add,mul,neg,inv,smul}.
Definition. A structure is a function S such that Dom(S) is subset of lang.
```

We can now define a each of the linear algebra objects above as a structure with the necessary components of `lang` in its domain. This allows for easy comparisons and expansions of these structures, one example of which we shall see in 3.2.4. Now we define the following abbreviations:

```
Let |S| stand for S[carr].
Let 0{S} stand for S[zero].
Let 1{S} stand for S[one].
Let add{S} stand for S[add].
Let mul{S} stand for S[mul].
Let neg{S} stand for S[neg].
Let inv{S} stand for S[inv].
Let smul{S} stand for S[smul].

Let a +{S} b stand for add{S}[(a,b)].
Let a *{S} b stand for mul{S}[(a,b)].
Let ~{S} a stand for neg{S}[a].
Let a -{S} b stand for add{S}[(a,neg{S}[b])].
Let a /{S} b stand for mul{S}[(a,inv{S}[b])].
Let a @{S} b stand for smul{S}[(a,b)].
Let a < S stand for a << |S|.
Let a < S* stand for a << |S|\{0{S}}.

Let (S has a) stand for (a << Dom(S)).
Let (S has a,b) stand for (a,b << Dom(S)).
...
Let (S has a,b,c,d,e,f,g,h) stand for (a,b,c,d,e,f,g,h << Dom(S)).
```

This approach resembles the notion of a structure in first-order logic: Our `structure` is the interpretation of (a subset of) the language `lang`, consisting of a set (if `|S|` is defined to be a set), constants and functions (if `zero`, `add`, etc. are defined to be suchlike). This is done, for example, in the following definition of an abelian group, additionally postulating the corresponding (first-order) axioms:

```
Definition. An abelian group is a structure G such that
     (G has carr,zero,add,neg)
 and (|G| is a set)
 and (0{G} < G)
 and (add{G} is a function from Prod(|G|,|G|) to |G|)
 and (neg{G} is a function from |G| to |G|)
 and (for all a < G     :       a +{G} 0{G} = a)
 and (for all a < G     :          a -{G} a = 0{G})
 and (for all a,b,c < G : a +{G} (b +{G} c) = (a +{G} b) +{G} c)
 and (for all a,b < G   :          a +{G} b = b +{G} a).
```

3.2.3 Homomorphisms

After the definition of various algebraic structures, we define linear maps between vector spaces. We introduce the set of homomorphisms as the carrier set of a new structure `Hom(K,V,W)`.

```
Let K denote a Field.

Definition. Let V and W be vector spaces over K. Let f be a function.
 f is linear over K from V to W iff
     (f is from |V| to |W|)
 and (for all u,v < V             : f[u +{V} v] = f[u] +{W} f[v])
 and (for all a < K for all v < V : f[a @{V} v] = a @{W} f[v]).

Signature. Let V and W be vector spaces over K.
Hom(K,V,W) is a structure.

Axiom. Let V and W be vector spaces over K.
(Hom(K,V,W) has carr).

Axiom. Let V and W be vector spaces over K.
|Hom(K,V,W)| is the set of functions f such that f is linear over K from V to W.
```

Regarding this structure as a vector space requires quite an extensive preparation, which is done in the func_ files, defining the zero, addition, negation, and scalar multiplication on `Hom(K,V,W)`. These are split from the D_homomorphisms file because the corresponding P_ files hold long proofs

which become easier to check separately. For brevity, in D_func_zero we introduce the following abbreviation:

```
Let 2VectorSpace(K,V,W) stand for
(K is a field and (V is a vector space over K) and (W is a vector space over K)).
```

The `structure` approach allows us to also regard `Hom(K,V,W)` as a ring, which is done in the files about endomorphisms.

3.2.4 Field2VS

Proving that a field is a vector space over itself becomes easy with our structure construction. We do not have to redefine the part of the structure that already exists but can instead just add scalar multiplication to the field. This is done in the following way:

```
Let K denote a field.
Axiom. (K has smul).
Axiom. smul{K} = mul{K}.
```

We then prove in P_field2VS that this really does create a vector space:

```
Theorem. Let K be a field. Then K is a vector space over K.
proof.
 (K has carr,zero,add,neg,smul).
 K is an abelian group.
 smul{K} is a function from Prod(|K|,|K|) to |K|.
 For all u < K                 :       1{K} @{K} u = u.
 For all a,b < K for all v < K : (a *{K} b) @{K} v = a @{K} (b @{K} v).
 For all a,b < K for all v < K : (a +{K} b) @{K} v = (a @{K} v) +{K} (b @{K} v).
 For all a < K for all v,w < K : a @{K} (v +{K} w) = (a @{K} v) +{K} (a @{K} w).
qed.
```

3.2.5 Automorphism Group

While the Lean file introduces the general linear group by proving statements about invertible linear maps, we took a step back and defined the multiplicative group of an arbitrary ring in D_unit_group. This branching at the top of the project graph reduces the required reasoning in the proof file. We then define the group of automorphisms of a given vector space simply as the unit group of its endomorphism ring. P_autmorphisms contains the proof that this group is exactly the set of bijective endomorphisms.

3.2.6 Lists

A list is defined as function from a set to a structure with a carrier and a zero element. The zero element ensures, that the list is not defined over an empty set of objects.

3.3 Experiences

3.3.1 Instability regarding Changes in the Preliminaries

The e-prover is not very effective in ignoring preliminaries that are irrelevant for the current task. For example, removing definitions and axioms regarding function composition from the files D_function and A_function has a huge impact on the checking time of most other proof files although they are only needed for the files regarding endomorphisms and automorphisms. Some proofs benefited from this reduction of preliminaries, needing only half the checking time as before. P_dual, on the other hand, now needed about 20 minutes instead of seven. Both behaviors are especially surprising, since all the statements about composition use notions that do not appear and cannot be used in these other contexts.

3.3.2 Ontological Checking

The following proof is an extract form 010P_ func_ smul.ftl:

```
Theorem. Let 2VectorSpace(K,V,W).
Let g < Hom(K,V,W). Then FuncSMul(K,V,W)[(1{K},g)] = g.
proof.
  FuncSMul(K,V,W)[(1{K},g)] and g are functions.
  Dom(FuncSMul(K,V,W)[(1{K},g)]) = |V| = Dom(g).
  W[smul] is a function from Prod(|K|,|W|) to |W|.
  For all v < V : FuncSMul(K,V,W)[(1{K},g)][v] = 1{K} @{W} g[v] = g[v].
  Hence the thesis (by FunExt).
end.
```

The proof is five lines long. The reasoning is done in the lines two and four. Line five can also be considered reasoning, but is more a reminder of sort, referencing what previous statement to use. Lines one and three are of pure ontological nature. So we use almost half of our texts for ontological purposes, which is fairly common throughout our proof files. In 5 we give a proposal on how to structure these ontological parts more clearly for the human reader.

These Ontological reasoning statements are especially crucial in long algebraic terms, and in many equations. Often times these statements can

slow down the e-prover substantially just by the amount of ontological preliminaries.

3.3.3 Functional Approach to Algebraic Structures

At the start of our project, we had to decide upon a way to formalize algebraic operations: One approach, similar to some of Naproche's example texts, would have looked like

```
Signature. 0 is an element.
Signature. a + b is an element.
Signature. -a is an element.
...
Let a - b stand for a + (-b).
...
Axiom. a + 0 = a.
Axiom. a - a = 0.
Axiom. a + (b + c) = (a + b) + c.
Axiom. a + b = b + a.
```

Instead, this library uses functional approach, as described in 3.2.2, where the notion of an abelian group was introduced via a `Definition`, stating, for example, that

```
add{G} is a function from Prod(|G|,|G|) to |G|.
```

This makes it easy to distinguish between elements of and operations on different sets. This approach, is more general and closer to usual mathematics. Using a `Definition` instead of signatures and axioms, immediately allows us to formulate how a specific structure is shown to be a field, vector space, ring or group. Our approach comes with many long algebraic chained statements that require a lot of ontological reasoning, which in turn slows down the e-prover. Therefor, the first alternative would run faster in the current version of naproche.

3.3.4 Other Experiences

Starting a file with

```
Let K denote a field.
```

is very convenient because it allows for omitting the same sentence in every definition and theorem. However, the line

```
Let V denote a vector space over K.
```

does not work in the current Naproche version, because `K` is a free undeclared variable.

4 Comparison of Lean and Naproche/ForTheL

Our approach to formalize algebraic objects as instances of a more general `structure` object, which enables us to successively enhance these objects with more algebraic structure, was inspired by the type class setup in vector_space.lean and the Lean mathlib as described in 2.2. While the mathlib establishes many basic algebraic structures in small steps, we restricted the definitions in ForTheL to only those structures used in our library.

In this and some more matters, our formalization is much closer to lecture or textbook style mathematics than the Lean version is: Of course, Naproche and ForTheL are designed to resemble natural mathematical language, which makes them much more readable for mathematicians than the code-like texts written in Lean. For algebraic expressions, the usual mathematical abbreviations can be realized in ForTheL by using patterns as seen in 3.2.2. While Lean can deal with implicit arguments like the domain of an algebraic operation, we had to be more explicit:

```
(a +{K} b) @{V} v = (a @{V} v) +{V} (b @{V} v)).
```

Such level of detail usually occurs only in the first week of a linear algebra lecture.

In both the Lean and our ForTheL files, groups and abelian groups are defined completely separately, since the first uses multiplicative notation, whereas the second is written additively. While it is no problem for humans to switch from one notation to the other, it would not have been practical in our `structure` approach.

Lean detects ontolocial errors immediately as they are just type errors, whereas the e-prover's problems with ontological checking, require additional reasoning. Examples are

```
FuncSMul(K,V,W)[(1{K},g)] and g are functions.
```

as described in 3.3.2, or

```
For all a < K and all v < V : (a,v) << Dom(smul{V}).
```

which we needed to state in some places. These have no real counterparts in textbook mathematics, contain no new information for the human reader and force the user to think about usually suppressed details like the functional aspects of these objects.

In total, vector_space.lean takes about 15 seconds to compile and check. In contrast, checking all files in our library takes Naproche about 100 minutes.

5 Next Steps in Naproche and ForTheL

It seems like this project has hit somewhat of a ceiling in what Naproche is capable of. Even small changes in beginning files can massively impact the check times of files further down the project graph. While the main problem that needs to be addressed is a more efficient structuring and analysis of the cache, we propose the following ideas to improve the writing experience.

Firstly Naproche needs an increased transparency in what the e-prover is doing. On the one hand the e-prover is a huge blessing, simplifying many steps. On the other hand it is the source of many problems, since the mathematician cannot guess where the e-prover gets stuck. Also, the e-prover will often take longer or even cannot compile previously working passages, if they are moved to a later part of the file. This is probably due to the increased breadth of the internal search tree. To counter these effects, while keeping the luxury of an assisting e-prover, we suggest the e-prover return the proofs of each statement, which can then be either pasted into the ForTheL text or saved in a separate file. This decoupling of the e-prover to the checking process will not only save time, but also guarantee more stability.

ForTheL has a

```
Let us show statementX.
reasoning
end.
```

functionality, to clean up the proof structure. We recommend adding a similar functionality

```
Since
reasoning
we have statementX.
```

that is only for ontological reasoning. In this project we find that these parts of the proofs make up for an extensive part of our texts. This leads to less readability, since these parts are mostly superfluous from a human point of view.

Additionally to this, we think ForTheL would greatly benefit from code sectioning functionality. This would not only improve transparency of the texts, but also speed up checking times. At the moment, the e-prover can only be restricted by commands like

```
statementX (by theoremY).
```

Introducing code sections will allow broader restrictions. This would also reduce the instabilities described in 3.3.1.

At the moment, functions, tuples and sets are hard coded into the Naproche system. Bigger projects will require more transparency in precisely what is axiomatized and how these basic notions are defined. The observations described in 3.3.3 suggest a built-in support for n-ary functions.

Integrating LaTeX code into ForTheL would be another nice feature bringing the texts even closer to usual mathematical language.