Geometry in Lean

Nicolò Cavalleri

September 12, 2020

1 Introduction

In this project we formalized with the help of the proof assistant Lean some concepts of differential geometry that have particular relevance in mathematical physics. In this document we will discuss briefly the definitions we formalized and we will discuss the most important mathematical details.

2 Contents

We will assume the reader has basic familiarity with Lean and type theory.

2.1 Product manifold

We proved the product of two smooth manifolds is naturally a smooth manifold. This was mainly needed to be able to require the product map for Lie groups to be smooth.

2.2 Smooth monoids

As a partial step towards defining Lie groups we defined smooth monoids, or more precisely the class `has_smooth_mul`:

```
@[to_additive]
class has_smooth_mul {𝕜 : Type*} [nondiscrete_normed_field 𝕜]
{H : Type*} [topological_space H]
{E : Type*} [normed_group E] [normed_space 𝕜 E] (I : model_with_corners 𝕜 E H)
(G : Type*) [has_mul G] [topological_space G] [has_continuous_mul G] [charted_space H G]
extends smooth_manifold_with_corners I G : Prop :=
(smooth_mul : smooth (I.prod I) I (λ p : G×G, p.1 * p.2))
```

We mainly need this to have an instance of monoid, analogous to an instance we defined for topology, for smooth functions valued in a smooth monoid. This is particularly useful to prove in one line that the sum of two nonnegative smooth real functions is a nonnegative real function and that such functions make up a monoid (actually a semiring with multiplication) or that $[0, 1]$-valued functions are a monoid with respect to multiplication. Since smooth monoids also have an associated Lie algebra, and the construction is exactly the same as for Llie groups, we developed the theory of Lie algebras for smooth monoids, although this was not the main purpose of introducing them. Also note that while Lie group are boundaryless, smooth monoids are in general not (one can actually prove that well behaved boundaryless smooth monoids are Lie groups), and in general can have proper corners, so this is also a nice use of Lean's smooth manifolds with corners.

The API for smooth monoids is the same as for Lie group so we refer to the paragraph on Lie groups for it.

2.3 Lie groups

We start with the definition

Definition 1. A type G is a Lie group over the nondiscrete normed field $\Bbbk$ if there is an instance of G as

- a *group*
- a *topological space*
- a *topological group*
- a *charted space* over the topological space H
- a *smooth manifold with corners* over the model with corners I.

Moreover we require that multiplication be smooth as a map $G \times G \to G$ and the inverse be smooth as a map $G \to G$.

This results in the following code

```
@[to_additive]
class lie_group {𝕜 : Type*} [nondiscrete_normed_field 𝕜]
   {H : Type*} [topological_space H]
   {E : Type*} [normed_group E] [normed_space k E] (I : model_with_corners 𝕜 E H)
   (G : Type*) [group G] [topological_space G] [topological_group G] [charted_space H G]
   extends has_smooth_mul I G : Prop :=
(smooth_inv : smooth I I (λ a:G, a⁻¹))
```

Some technical remarks:

1. We need to require an instance of topological group instead of defining an instance of Lie group as a topological group because of dangerous class instance problems: `topological_group` has one argument and `lie_group` has two.

2. A Lie group is a priori a manifold with corners and it is not cornerless because one of the main features of the geometry section of Mathlib is to regard every boundaryless manifold as a manifold with corners and not really make difference between the two. I personally do not particularly like this as I believe that there are very important areas of mathematics such as complex and symplectic geometry or Lie theory where this is not needed and will add unnecessary complications, but my definition of boundaryless manifold was not accepted, so, at list for now, this will be the definition of Lie groups.

In addition, we defined the constructor `Lie_group_core` which allows to define a Lie group without having first defined an instance of topological group. We also defined Lie group morphisms. Apart from some standard API needed to work with Lie groups and Lie group cores, we proved real numbers are a Lie group and we proved the units of a normed algebra over a normed field are a Lie group (there is actually a lot to say on this). The case of $GL(n)$ follows as a particular case. It is not possible to prove yet that other famous Lie groups such as $O(n)$ are indeed Lie groups because Mathlib is still missing the implicit function theorem. However, people are working on this and this will be possible soon.

2.4 Submanifolds

Submanifolds were defined as the image of injective smooth maps (note that this definition is different than the usual definition of embedded submanifolds). Immersed and embedded submanifolds were defined similarly. However, following the current trends of Mathlib, we encourage the user to state results on submanifolds by talking about monomorphisms, in the style of category theory, as this is more general and more natural in type theory.

2.5 Lie subgroups (and submonoid)

The same implementation for submanifolds was used. Again we encourage to prefer using monomorphism, which are there because we did define morphisms of both Lie groups and smooth monoids.

2.6 Continuous bundled maps

We formalized continuous bundled maps mainly because we wanted to define smooth bundled maps and we wanted smooth maps to be recognized also as continuous maps not to loose generality for properties that are true for continuous maps.

```
@[protect_proj]
structure continuous_map (α : Type*) (β : Type*)
[topological_space α] [topological_space β] :=
(to_fun             : α → β)
(continuous_to_fun  : continuous to_fun)
```

Before the introduction of such type the subtype of continuous function was used, but this presented numerous problems as for example the impossibility of extending such type to get other types, and all usual problems connected to the clumsiness of subtypes. This change needs not much comment: usual API for bundled structures was written and necessary refactors were carried out.

We also proved that continuous bundled functions valued in a topological algebraic structures inherit the structure (i.e. e.g. that continuous function valued in a topological group form a group). In particular we proved that continuous functions valued in a topological vector space valued in a topological semimodule over a ring R which is also a topological space (but not necessarily a topological ring) are an R-algebra.

2.7 Derivations

This was the only exclusively algebraic contribution to Mathlib and the main tool to implement the Lie algebra of a Lie group. Although derivations in non-commutative algebra are useful for mathematical physics, the definition of a derivation was given in the context of commutative algebra, because bimodules are not there yet in Mathlib. The theory should be generalized as soon as bimodules are there. The following definition of derivation was given:

```
@[protect_proj]
structure derivation (R : Type*) (A : Type*) [comm_semiring R] [comm_semiring A]
[algebra R A] (M : Type*) [add_cancel_comm_monoid M] [semimodule A M] [semimodule R M]
[is_scalar_tower R A M]
extends A →ₗ[R] M :=
(leibniz' (a b : A) : to_fun (a * b) = a • to_fun b + b • to_fun a)
```

It was no additional effort to state the theory of derivations for semirings and semimodules instead that for rings and modules, so we decided to be as general as possible. The only additional step we needed to take in order to reach this level of generality was to define *cancel monoids*, i.e. monoids where cancellation is valid on both sides. Before this only *left cancel monoids* were there in Mathlib. Introducing this concept was a good idea as it can come in useful independently from derivations. The other refactor of Mathlib that was done (mainly by Anne Baanen) in order to define derivations was introducing `is_scalar_tower` and substituting it to `is_algebra_tower`, which was used until now in Mathlib. An idea that came out by this refactor was to substitute instances of `is_scalar_tower` to the more clumsy definition `restrict_scalars`, since the two are in a certain sense equivalent. However, this was postponed to a later moment since this was too far from the purpose of the project. We did not develop much the theory of derivations but we did prove that, in the case of derivations from an algebra to itself, the commutator of derivations is a derivation and that derivations are a Lie algebra. We also proved that composing a linear map with a derivation yields a derivation.

2.8 Derivation bundle

We proved that the derivation bundle, defined as the sigma type of point derivations, is a smooth manifold and we partially proved that it is a vector bundle according to our vector bundle definition. More will be said on these definitions later on. Writing down the charts for it was harder than expected.

2.9 Smooth bundled maps

Similarly to continuous bundled maps we introduced smooth bundled maps and wrote some API concerning them. We also proved that functions valued in a smooth algebraic structure inherit that algebraic structure (for smooth monoids, Lie groups and smooth semirings), and we proved that smooth functions valued in a normed vector space over $\Bbbk$ (where smooth is meant with respect to the smooth structure coming from the normed vector space structure) are a $\Bbbk$-vector space, and that field valued functions are a $\Bbbk$-algebra.

2.10 Lie algebra of a Lie group

Despite the title we actually defined more generally the Lie algebra of a smooth monoid. This should be one of the most original contributions to Mathlib of this projects. In the current state of Mathlib there are two possible ways to implement Lie algebra. The first is the traditional way, by using left invariant vector fields. The second is to use the adjoint representation: there is a differential defined for the current tangent bundle which would permit to define easily the Lie algebra by deriving the adjoint representation of the group. The first definition is way harder as it requires vector fields, which in turn require vector bundles. The second definition is by far easier to give, but much harder to work with (it is for example harder to prove the Lie algebra of a group is indeed a Lie algebra). It also has the disadvantage of being slightly incoherent with mathematics literature: although the two are isomorphic in the category of Lie algebras, left invariant vector fields and adjoint representation are introduced as separate concepts and one would possibly want both to be present in and formalized with a proof assistant. We chose to use the first definition but with a tweak: we used the result that says that derivations from $C^\infty(M;\Bbbk)$ to itself are isomorphic to vector fields in the category of Lie algebras over $\Bbbk$. This permits us to define the Lie algebra of a smooth monoid by using almost exclusively algebra, which is an advantage because algebra is usually more easily formalized with proof assistants than geometry. All the geometric information of a smooth monoid necessary to define the Lie algebra is contained in the $\Bbbk$-algebra $C^\infty(M;\Bbbk)$, which is an interesting point by itself.

To define a left invariant derivation though we need the concept of point derivation, as much as to define a left invariant vector field we need the concept of tangent vector at a point. We defined the space of point derivations at a point $x : M$ as the vector space of $\Bbbk$-derivations from $C^\infty(M;\Bbbk)$ to $\Bbbk$ where the module structure of $C^\infty(M;\Bbbk)$ over $\Bbbk$ is given by evaluation at a point, meaning that given $f : C^\infty(M;\Bbbk)$ and $k : \Bbbk$ we define $f \cdot k$ to be $f(x) \cdot k$. Instead of defining the module structure with a definition and use it with the @ notation, we preferred using the other common strategy of giving a type synonym to $C^\infty(M;\Bbbk)$ depending on the point. $C^\infty(M;x)$ is notation for $C^\infty(M;\Bbbk)$ and is meant to be used as $C^\infty(M;\Bbbk)$ with the above module structure over $\Bbbk$ dependent on x. At this stage we can define a function `eval` (standing for *evaluate*) that takes an input a point of M and a global derivation over $C^\infty(M;\Bbbk)$ and returns a point derivation at x. This function is meant to be the analogue of the evaluation of a vector field at a point giving a tangent vector. Clearly the definition of such function, given a global derivation X, a smooth function f and a point x, is given by evaluating $X(f)$ at x. More precisely, the evaluation is the function $X \mapsto (f \mapsto X(f)(x))$. One needs to prove that the result of evaluation is indeed a derivation but this is not difficult. The other concept which is missing to define left invariance is that of differential, but this is clearly defined as it is usually defined when the tangent bundle is implemented with derivations.

Note that one can now define the derivation bundle as the Σ type of spaces of point derivations, and give it a smooth structure which will make it isomorphic as a vector bundle to the tangent bundle (and this is what we did, as explained above), but this is not at all needed to define the Lie algebra of a smooth monoid: as we said we need just the algebra.

At this point one might be tempted to define left invariant derivations as

```
structure left_invariant_vector_field (I : model_with_corners 𝕜 E H)
(G : Type*) [topological_space G] [charted_space H G] [smooth_manifold_with_corners I
 G]
```

```
[group G] [topological_group G] [lie_group I G] extends vector_field_derivation I G :=
(left_invariant' : ∀ g, to_vector_field_derivation.eval g =
   (fd (Lb I G g)) (1 : G) (to_vector_field_derivation.eval (1 : G)))
```

However, this is not possible in Lean as the type of `o_vector_field_derivation.eval g`, i.e. `point_derivation I G g` is equal but not definitionally equal to the type of `(fd (Lb I G g)) (1 : G) (to_vector_field_derivation .eval (1 : G))`, i.e. `point_derivation I G ((Lb I G g) (1 : G))`. Because the types of the two sides of the equal side must be definitinally equal, the heterogeneous equal sign, `==` must be used instead, which is there precisely for this kind of situations. However, `==` is mostly avoided in Lean as it creates many problems. In our case, proving that some vector field is left invariant is not a problem as one could write an extensionality rule that rewrites the heterogeneous equality in terms of a more classical equality by applying both sides (that are point derivations) to a function. Unfortunately it is not possible to work this way with a hypothesis that a derivation is left invariant because Lean does not have an axiom that guarantees that congruence involving heterogeneous equality is true, an there are no plans to add such an axiom. The trick here (to my knowledge is the first time it is used, although I need to verify this) is to incorporate extensionality in the definition of left invariance. This should not be a problem because extensionality is in any case almost always used normally in proofs of this kind. We thus get the following definition

```
structure left_invariant_vector_field (I : model_with_corners 𝕜 E H)
(G : Type*) [topological_space G] [charted_space H G] [smooth_manifold_with_corners I G]
[group G] [topological_group G] [lie_group I G] extends vector_field_derivation I G :=
(left_invariant' : ∀ f g, to_vector_field_derivation.eval g f =
   (fd (Lb I G g)) (1 : G) (to_vector_field_derivation.eval (1 : G)) f)
```

At this point proving that left invariant vector fields are a lie algebra is just a matter of standard geometry bookwork (the standard proof is exactly what we used). One can also write an equivalence between left invariant vector fields and the tangent space at the identity and transfer through this the instance of Lie algebra to the tangent space at the identity, however this is not really needed.

2.11 Curves

Also in the case of curves it was necessary to choose between different design scenarios. Since one would possibly want to treat all curves, defined on any type of interval, in a single case, since what really matters for many purposes in geometry is defining speed, there are many possible definitions. The first that could come to mind, and the first I tried, is to define a curve as a smooth map from a connected one dimensional manifold to the considered manifold (where, as always in Lean, manifolds are meant to be with corners). This definition is rather free and flexible but rather clumsy and, most importantly, each curve is its own type and we cannot really take into account the type of all curves. Another possibility is to define curves from an appositely defined inductive type `interval`, which has its advantages but it is a bit artificial. The definition we ended up using is modelled on the idea of local homeomorphisms. We defined a curve to be a bundled `smooth_on` map on ℝ, for which we care only for the values on it takes on its source.

For the purpose of defining complete vector fields we need the concept of maximal curve. It is particularly useful, in this context, to define an ordering on curves so to get for free all concepts related to orders, such as maximality. To define a partial order on curves we need antisymmetry and for this we need some extensionality rule for curves for which two curves are equal if they have the same domains and they are equal on them. With the definition we described above this is not true as two curves that take different values outside their source are different. This is fixed by requiring that the manifolds are nonempty and that any curve takes a default value outside its source. Thanks to this we are able to save extensionality and we get an order. The final definition is thus

```
structure curve {E : Type*} [normed_group E] [normed_space ℝ E]
{H : Type*} [topological_space H] (I : model_with_corners ℝ E H)
(M : Type*) [inhabited M] [topological_space M] [charted_space H M]
[smooth_manifold_with_corners I M] (n : with_top ℕ) extends C_l[n](Isf(ℝ), ℝ; I, M) :=
(connected_source    : is_connected source)
(default_value       : ∀ x ∉ source, to_fun x = default M)
```

With this definition we can define integral curves, complete vector fields and we can define the flow of a vector field. The speed is defined as a derivation in the classical way one can find on almost every geometry book.

2.12 Exponential map

The exponential map of Lie theory was implemented classically: given a left invariant vector fields we take an integral curve for it passing at the identity in the moment 0 and we take its value at the moment one. The hardest part is proving that an integral curve exists (for the definition we do not need to know it is maximal and that it is unique as a maximal curve, nor that it is defined on the whole real line). For this we use the manifold structure on the derivation bundle to write the differential equation defining the flow in charts. ODE theory in Lean is not enough developed yet to prove existence but we stated the ODE and will PR the exponential map to Mathlib only after the existence theorem for the Cauchy problem will be there. Experienced people are already working on such theorems.

2.13 Diffeomorphisms and local diffeomorphisms

Such PR is not small in terms of amount of code but quite easy and quite standard. Everything was developed by imitating homeomorphisms and local homeomorphisms and by simply adapting code from topology.

2.14 Smooth fiber bundles

Always imitating topology (in this case *topological fiber bundles*) we defined smooth fiber bundles and wrote some API for them modelled on the API for topological bundles.

2.15 Smooth vector bundles

Smooth vector bundles are the other most original contribution of this project, together with Lie algebras. The problem with vector bundles is that they are a fairly complex object in mathematics and there are many possible implementation. Many people thought about this before and I will list here six proposed implementations, the first five proposed by Sébastien Goeuzel and the last one proposed by Mario Carneiro. Quoting directly their words a smooth vector bundle could be implemented as (E and F are meant to be vector spaces)

1. a manifold N with charts taking values in $E \times F$, and the changes of coordinates send fibres of the projection on E to fibers in a linear way (i.e., prescribing the groupoid as the one of linear maps in the second coordinate). We thus get a quotient manifold, and a vector bundle. This is probably a bad choice to use a quotient manifold like that, because for instance in the tangent bundle the manifold is not definitionally equal to the quotient manifold obtained by quotienting the tangent bundle by its fibres.

2. more data: a manifold N as above, but also a manifold M and a projection π from N to M that behaves nicely. Then the fibres of the projection can be endowed with a vector space structure in a canonical way. But there is the difficulty that, in the tangent bundle case, the fibres of the projection (written as `{y // pi y = x}`) are not definitionally equal to the tangent space.

3. even more data: N, M and π, but also a family of vector spaces F_x indexed by $x \in M$ and isomorphisms between the fibre above x and F_x.

4. just a manifold structure on a Pi type of the form `Π (x : M), F x`, with the right groupoid.

5. even simpler, a manifold structure on $M \times F$. This may seem crazy, but the tangent bundle has previously been implemented with this underlying type.

6. a manifold Z equipped with an equivalence to $\Sigma(x : E), F_x$.

I personally tried both implementation number 2 and number 6. Number 2 has the advantage of being the closest to what is the classical definition of vector bundle in geometry, however it is quite unnatural in type theory as the problem of the fibres not being definitionally equal to the counterimages of the projection. This problem can be solved by defining a canonical equivalence between the two types and transfer instances from the normal fibres to the fibres of the projection. One can even solve this by making equivalence a class (it is now a structure) and using typeclass inference, even if this is a bit crazy. This shows this definition works and it is totally possible to do everything with it, however it remains clumsy and unnatural, especially for performing operations on bundles such as the dual bundle, the sum of bundles etc. I finally decided to use definition six with a tweak: defining vector bundles directly as sigma types and never using equivalences. This means that when working with a vector bundle one will use directly a function `F : M → Type*` and will talk implicitely about the total space and the projection, which are automatically defined. This idea is a bit crazy as it poses a very strong constraint on the type of vector bundles: one cannot really take a random manifold and prove it is a vector bundles (although it can be proved it is diffeomorphic to one). However, if one thinks more about it, when would this not be the natural way to define a vector bundle? One needs to have a vector space structure on each fiber so there is not really any other natural way to do this then using a sigma type or a product. This will thus mean that any manifold whose underlying type is a product will need to be changed to a constant sigma type. This has already been done for the tangent bundle, precisely to adapt it to this definition (indeed this definition was accepted abstractly and should become the official definition of vector bundle in Mathlib).