Formalising Lie algebras

Oliver Nash
acmsigplan@olivernash.org
Imperial College
London, United Kingdom

Abstract

Lie algebras are an important class of algebras which arise throughout mathematics and physics. We report on the formalisation of Lie algebras in Lean's Mathlib library. Although basic knowledge of Lie theory will benefit the reader, none is assumed; the intention is that the overall themes will be accessible even to readers unfamiliar with Lie theory.

Particular attention is paid to the construction of the classical and exceptional Lie algebras. Thanks to these constructions, it is possible to state the classification theorem for finite-dimensional semisimple Lie algebras over an algebraically closed field of characteristic zero.

In addition to the focus on Lie theory, we also aim to highlight the unity of Mathlib. To this end, we include examples of achievements made possible only by leaning on several branches of the library simultaneously.

CCS Concepts: • **Theory of computation → Type theory**.

Keywords: formal math, algebra, Lie theory, Lean, proof assistant

1 Introduction

1.1 Skew-symmetric matrices

Recall that a rotation in Euclidean space can be represented by an invertible matrix X whose inverse is its transpose:

$$X^{-1} = X^T.$$

At least as far back at the 19th Century, it was observed that if A is a skew-symmetric matrix and $\epsilon \in \mathbb{R}$ is small then $I + \epsilon A$ is almost a rotation. Indeed since $A^T = -A$, we have:

$$(I + \epsilon A)^T (I + \epsilon A) = I - \epsilon^2 A^2.$$

That is, the inverse of $I + \epsilon A$ is its transpose, if we neglect terms order ϵ^2.

Better yet, the exponential $e^{\epsilon A}$ is truly a rotation (no terms neglected) and for another such matrix B, the Baker-Campbell-Hausdorff formula quantifies how the composition of the rotations $e^{\epsilon A}$, $e^{\epsilon B}$ behaves in terms of skew-symmetric matrices:

$$e^{\epsilon A} e^{\epsilon B} = e^{\epsilon(A+B) + \frac{\epsilon^2}{2}[A,B] + O(\epsilon^3)}. \tag{1.1}$$

The term $[A, B]$ appearing in (1.1) is defined as:

$$[A, B] = AB - BA \tag{1.2}$$

and is an instance of a Lie bracket. It defines a natural product: if A and B are skew-symmetric then[1] so is $[A, B]$.

The Lie bracket is skew-commutative:

$$[A, B] = -[B, A],$$

and in general non-associative, but by way of compensation it satisfies the Jacobi identity:

$$[A, [B, C]] + [B, [C, A]] + [C, [A, B]] = 0. \tag{1.3}$$

1.2 Abstract Lie algebras and their ubiquity

Recognising skew-symmetric matrices merely as an example, one can consider the study of abstract Lie algebras. These are modules carrying a bilinear, skew-commutative product which satisfies the Jacobi identity (1.3).

The study of abstract Lie algebras was initiated by Lie and independently by Killing more than 140 years ago [17], [13–15], [10] and the subject now pervades much of modern mathematics and physics.

In classical physics, both linear and angular momentum are best understood as taking values in the dual of a Lie algebra. More importantly, recognising that the space of classical observables forms a Lie algebra[2] under the Poisson bracket is an important step in quantisation. Furthermore, in particle physics, elementary particles such as quarks are essentially basis vectors of irreducible representations of Lie groups [3] and thus of Lie algebras.

---

[1] If this is the first time you have seen this, then check: it's a fun calculation.
[2] In fact, a Poisson algebra, but it is the bracket structure that is more subtle.

In number theory, automorphic forms, central objects of study in the Langlands programme, satisfy a differential equation defined in terms of a reductive Lie algebra. In differential geometry, the tangent bundle is special amongst vector bundles because its sections carry a natural Lie algebra structure; moreover with just this structure one can define the de Rham cohomology, thus connecting with algebraic topology. In Riemannian geometry and gauge theory, the curvature 2-form takes values in a Lie algebra. In symplectic geometry, the moment map takes values in the dual of a Lie algebra.

Of course the lists above hardly scratch the surface. The unifying theme is that a great many types of symmetry are naturally Lie groups or algebraic groups, and thus have associated Lie algebras which are essential for their study. Understanding symmetry thus requires understanding Lie algebras and for this reason the classification of semisimple Lie algebras is rightly regarded as a landmark result, obtained just in time for the 20th Century[3].

1.3 A roadmap for this article

Much of the work discussed here was motivated by the desire to formalise the statement of the classification of semisimple Lie algebras in a proof assistant. This statement appears in section 10 of this article and the intervening sections 2 – 9 essentially correspond to the various waypoints which were necessarily passed on the way to this milestone.

Aside from section 2, which is included to give a taste for foundational design decisions, these intervening sections fall neatly into two groups.

The first group, which consists of sections 3, 4, enables us to define the class of Lie algebras we are classifying.

The second group, which consists of sections 5 – 9, enables us to define the concrete Lie algebras which, according to the classification theorem, exhaust the class of semisimple Lie algebras (up to equivalence).

Section 11 sketches the formalisation of weights and roots in Lie theory, and section 12 concludes with some general remarks.

1.4 A primer on Lean and Mathlib

The work dicussed here was implemented using Lean[4]. Lean is a dependently-typed programming language together with a proof assistant.

Like Coq, Lean is based on the Calculus of Inductive Constructions; see [8] for a detailed discussion. We do not expect the reader to be an expert in Lean. For the purposes of this article, an intuitive understanding of the following Lean keywords should suffice:

- `variables`
- `def`
- `abbreviation`
- `lemma`
- `class`
- `instance`

The keyword `variables` adds variables to the local context; `def` and its variant `abbreviation` make definitions. The keyword `lemma` is self-explanatory but `class` and its partner `instance` deserve further comment.

The keyword `class` defines typeclasses. Lean includes a powerful typeclass system which is heavily used in its Mathlib [20] library. For example Mathlib contains the typeclass `comm_ring` which defines what it means for a type to carry the structure of a commutative ring (with unit). Using this, we can say that a type R is a commutative ring by supplying a typeclass argument `[comm_ring R]` in the statement of definition or lemma.

Typeclasses have one constructor which may take several arguments. In the case of a commutative ring, the arguments correspond to an addition function, a multiplication function, associativity of addition, associativity of multiplication, commutativity of multiplication etc.

Lastly, the keyword `instance` is what makes typeclasses so useful: it allows us to register the fact that some type carries a typeclass. These instance statements can even contain mathematically non-trivial facts. For example here is Mathlib's statement that the real numbers form a commutative ring[5] ⬀ :

```
instance : comm_ring ℝ :=
begin
  -- proof using Cauchy sequences (omitted here)
end
```

This instance means Lean knows that any lemmas about commutative rings automatically hold for the real numbers[6].

Aside from `comm_ring`, the most important typeclasses expliclty used in the work discussed here are `add_comm_-group`, which states that a type carries the structure of a commutative group (with group operation denoted +) and `module`, which states that a type carries the structure of a module over a set of scalars. For readers not familiar with modules, simply read 'vector space' instead.

Finally, since of all the work discussed here has been merged to the Mathlib master branch, anyone wishing to run, compile, interact with, or build upon Mathlib's Lie algebras can do so by following instructions available at the Lean community website, especially:

 https://leanprover-community.github.io/get_started.html

and:

 https://leanprover-community.github.io/leanproject.html.

---

[3]Cartan submitted his thesis [9] in March 1894.
[4]More precisely the community fork of Lean 3.

[5]Notice the ⬀ icon; it is a permalink to the corresponding code in Mathlib. We provide such permalinks throughout the text so that readers using an internet-connected device may easily navigate to appropriate locations in Mathlib.
[6]Of course more is true, e.g., the real numbers are a linearly-ordered field and Mathlib knows this too.

Online documentation is automatically generated for all Mathlib code. For Lie algebras, a good entry point is:

https://leanprover-community.github.io/mathlib_-docs/algebra/lie/basic.html.

1.5   Lie algebras in Lean

Here is the definition of a Lie algebra in Mathlib ⧉ :

```
class lie_ring (L : Type v)
  extends add_comm_group L, has_bracket L L :=
(add_lie : ∀ (x y z : L),
  ⁅x + y,z⁆ = ⁅x,z⁆ + ⁅y,z⁆)
(lie_add : ∀ (x y z : L),
  ⁅x,y + z⁆ = ⁅x,y⁆ + ⁅x,z⁆)
(lie_self : ∀ (x : L), ⁅x,x⁆ = 0)
(leibniz_lie : ∀ (x y z : L),
  ⁅x,⁅y,z⁆⁆ = ⁅⁅x,y⁆,z⁆ + ⁅y,⁅x,z⁆⁆)

class lie_algebra (R : Type u) (L : Type v)
  [comm_ring R] [lie_ring L] extends module R L :=
(lie_smul : ∀ (t : R) (x y : L),
  ⁅x,t · y⁆ = t · ⁅x,y⁆)
```

The skew-commutative property follows from the `lie_self` axiom and the Jacobi identity is equivalent to the `leibniz_-lie` axiom[7].

There exist computer algebra systems such as SageMath, GAP, MAGMA as well as a Mathematica package available at http://katlas.org, and the Lie-specific package LiE [22], that are capable of performing calculations involving Lie algebras. However to the best of our knowledge, there is no previous work formalising the theory of Lie algebras.

As of December 2021, Mathlib contains over 6,000 lines of code about Lie algebras and their representations, broadly following Bourbaki [5–7]. Material covered includes ⧉ :

- Lie algebras and Lie modules
- Morphisms and equivalences of Lie algebras and Lie modules
- Lie subalgebras, Lie submodules, Lie ideals, and quotients
- Extension and restriction of scalars
- Direct sums of Lie modules and Lie algebras
- Tensor product of Lie modules
- Lie ideal operations, the lower central series, the derived series, and derived length
- Nilpotent, solvable, simple, semisimple Lie algebras, the radical, and the centre of a Lie algebra
- Cartan subalgebras
- Weight spaces of a Lie module, and thus root spaces of a Lie algebra
- The universal enveloping algebra (and its universal property)
- The free Lie algebra (and its universal property)
- Definition of the classical Lie algebras

- Definition of the exceptional Lie algebras

The final item is worth highlighting. There is no easy route to the definition of the exceptional Lie algebras (section 9) and it is an important milestone since it allows us to state the classification of semisimple Lie algebras (section 10). A *proof* of this classification within Mathlib would be a significant undertaking but now looks achievable.

1.6   A note about notation

We draw the reader's attention to the brackets appearing in Lean code such as the equation:

```
⁅x,⁅y,z⁆⁆ = ⁅⁅x,y⁆,z⁆ + ⁅y,⁅x,z⁆⁆
```

appearing above. These brackets, associated with the `has_-bracket` typeclass ⧉ , were introduced to Mathlib to provide a convenient notation for the Lie bracket[8].

They are also used in the notation for morphisms in Lie theory. For example, the following is Mathlib's notation for an R-linear map of modules:

```
M₁ →ₗ[R] M₂
```

whereas the following is the notation for a morphism of Lie algebras:

```
L₁ →ₗ[R] L₂
```

and the following is the notation for a morphism of Lie modules over a Lie algebra L with coefficients in R:

```
M₁ →ₗ[R,L] M₂
```

Finally, similar remarks apply to equivalences, i.e., we use the notations:

```
L₁ ≃ₗ[R] L₂
```

and:

```
M₁ ≃ₗ[R,L] M₂
```

2   Design choices: Leibniz vs. Jacobi

The choice of the axiom `leibniz_lie` in the definition exhibited in section 1.5 deserves explanation, if only because it serves as a simple example of the sorts of choices that repeatedly came up in the course of formalisation.

In the presence of the other Lie algebra axioms, each of the following are equivalent:

$$[x, [y, z]] + [y, [z, x]] + [z, [x, y]] = 0, \quad (2.1)$$

$$[[x, y], z] = [x, [y, z]] - [y, [x, z]], \quad (2.2)$$

$$[x, [y, z]] = [[x, y], z] + [y, [x, z]]. \quad (2.3)$$

Note that (2.3) is the axiom `leibniz_lie` and that given $x : L$, if we define $D_x : L \to L$ by:

$$D_x y = [x, y]$$

---

[7]We comment on the choice of axioms in section 2.

[8]Actually they have since been used elsewhere, e.g., for the commutator of two subgroups of a group.

then (2.3) says that D_x satisfies the Leibniz product rule:

$$D_x[y, z] = [D_x y, z] + [y, D_x z],$$

i.e., it says that D_x is a derivation.

One might think that the Jacobi identity (2.1) is the best choice since it looks the most symmetric; in fact it is the worst choice. This becomes clear when one considers that we also need a theory of Lie modules.

Informally, if a Lie algebra L acts linearly on a module M, and if we denote the action of $x : L$ on $m : M$ by $[x, m] : M$ then this action turns M into a Lie module for L iff:

$$[x, [y, m]] = [[x, y], m] + [y, [x, m]],$$

for all $x, y : L$ and $m : M$.

Now consider the case $L = M$ and observe that any Lie algebra is thus a module over itself. This so-called adjoint action is extremely important in Lie theory. Observe also that if we replace $z : L$ in equations (2.1) – (2.3) by $m : M$ then the terms $[y, [z, x]], [z, [x, y]]$ do not make sense since there is no action of M on L. Thus (2.1) cannot be used as an axiom for Lie modules.

By choosing to define Lie algebras using the `leibniz_lie` axiom (2.3) we thus obtain a theory where Lie algebras and Lie modules *definitionally* satisfy the same axiom. This is a desirable convenience that we exploit. For example, here is the code that defines the adjoint action ⬈:

```
instance lie_self_module : lie_ring_module L L :=
{ .. (infer_instance : lie_ring L) }
```

One might ask why to choose (2.3) over (2.2). This is of lesser importance but (2.3) is still the better choice. This is because (2.2) requires using subtraction which is often a secondary operation, defined via addition and inverses. This means that when constructing Lie algebras downstream, it is likely there will be more direct proof of (2.3).

On the other hand, as a simplification lemma (rather than a definition) (2.2) is excellent since it can be used to push Lie brackets right-most in nested expressions. Indeed the following `simp` lemma establishes this as the normal form in Mathlib[9] ⬈:

```
@[simp] lemma lie_lie :
  ⁅⁅x,y⁆,m⁆ = ⁅x,⁅y,m⁆⁆ - ⁅y,⁅x,m⁆⁆ :=
```

And of course we could never register (2.3) as `simp` lemma since we would get a `simp` loop: the term $[y, [x, m]]$ on the right hand side of (2.3) is of the same form as the term $[x, [y, m]]$ on the left.

Notwithstanding the words above, the choice of axiom here is of minor importance. However there are many such choices and en mass they accumulate to have non-trivial impact.

---

[9]Here and elsewhere we have omitted the proof. In the actual code (available via the permalinks) the proof follows immediately after the := syntax.

3 Ideal operations, solvable Lie algebras, nilpotent Lie modules

Several important constructions in Lie theory are conveniently stated in the language of ideal operations.

Informally, if L is a Lie algebra, M is a Lie module of L, I is an ideal of L, and N is a Lie submodule of M we define:

$$[I, N] = \text{smallest Lie submodule of } M \text{ containing}$$
$$[x, n] \text{ for all } x : I, n : N.$$

Note that since any Lie algebra can be regarded as a Lie module over itself, and since ideals are just Lie submodules, we can thus combine two ideals I, J to make another: $[I, J]$.

Taking advantage of the complete lattice structure on Lie submodules, we formalised the above as ⬈:

```
def lie_span (s : set M) : lie_submodule R L M :=
Inf {N | s ⊆ N}
```

and ⬈:

```
instance : has_bracket
  (lie_ideal R L) (lie_submodule R L M) :=
⟨λ I N, lie_span R L
  {m | ∃(x : I) (n : N), ⁅(x : L),(n : M)⁆ = m}⟩
```

This definition is compatible with the lattice structure on Lie submodules in numerous ways. For example ⬈:

```
lemma lie_le_right : ⁅I, N⁆ ≤ N :=
```

```
lemma lie_comm : ⁅I, J⁆ = ⁅J, I⁆ :=
```

```
lemma lie_le_inf : ⁅I, J⁆ ≤ I ⊓ J :=
```

```
@[simp] lemma lie_sup :
  ⁅I, N ⊔ N'⁆ = ⁅I, N⁆ ⊔ ⁅I, N'⁆ :=
```

```
lemma mono_lie (h₁ : I ≤ J) (h₂ : N ≤ N') :
  ⁅I, N⁆ ≤ ⁅J, N'⁆ :=
```

We also established alternate characterisations of the definition $[I, N]$. Firstly ⬈:

```
lemma lie_ideal_oper_eq_linear_span :
  (↑⁅I, N⁆ : submodule R M) = submodule.span R
  {m | ∃(x : I) (n : N), ⁅(x : L),(n : M)⁆ = m} :=
```

This says that if we forget the action of L and regard $[I, N]$ merely as a submodule of M then it is just the *linear* span of the generating elements (rather than the Lie span). This result was very useful.

Secondly, we established a characterisation that does not use spans at all ⬈:

```
lemma lie_ideal_oper_eq_tensor_map_range :
  ⁅I, N⁆ = ((to_module_hom R L M).comp
  (map_incl I N : ↑I ⊗ ↑N→ₗ⁅R,L⁆ L ⊗ M)).range :=
```

Informally, from the data I, N we can build a composition of morphisms of Lie modules:

$$I \otimes N \to L \otimes M \to M,$$

where the first arrow is the tensor product of inclusion maps and the second arrow is the action of L. The lemma states that the range of this composite map is $[I, N]$.

Probably the most important application of these ideal operations are the definitions of the derived series and lower central series.

Informally the derived series, $D^0 L, D^1 L, D^2 L, \ldots$ of a Lie algebra is a sequence of ideals of L, defined inductively using the ideal operation discussed above:

$$D^0 L = L$$
$$D^{k+1} L = [D^k L, D^k L].$$

We formalised the derived series as ☐ :

```
def derived_series_of_ideal (k : ℕ) :
  lie_ideal R L → lie_ideal R L :=
(λ I, [I, I])^[k]

abbreviation derived_series (k : ℕ) :
  lie_ideal R L :=
derived_series_of_ideal R L k ⊤
```

Note that we defined `derived_series` as a special case of a more general definition `derived_series_of_ideal`. This was very useful since it provides a type-theoretic expression of the fact that if we regard a Lie ideal as a Lie algebra in its own right, then the terms of its derived series are also ideals of the enclosing algebra[10]. Here is the statement that the two concepts really do agree when we regard an ideal as a Lie algebra in its own right ☐ :

```
lemma
  derived_series_eq_derived_series_of_ideal_comap
  (k : ℕ) :
  derived_series R I k =
  (derived_series_of_ideal R L k I).comap I.incl:=
```

We then used the derived series to define what it means for a Lie algebra to be solvable ☐ :

```
class is_solvable : Prop :=
(solvable : ∃ k, derived_series R L k = ⊥)
```

and built out the standard theory.

By way of example, we recall that the standard example of a solvable Lie algebra is the set of upper-triangular square matrices[11].

Similarly, we defined the lower central series ☐ , and used it to define the concept of nilpotency ☐ . In this case we generalised slightly from the standard references since the concept of nilpotency makes sense not just for Lie algebras but for Lie modules and so we made this more general definition. This turned out to be useful when formalising Engel's theorem (to appear).

---

[10]Just like with normal subgroups of a group, if I is an ideal of a Lie algebra L and J is an ideal of I it is not necessarily true that J is an ideal of L.

[11]Again, if this is the first time you've seen this, it is a fun calculation to verify that if A, B are upper-triangular matrices then so are AB and BA, and thus also $[A, B]$.

4 Case study: the radical is solvable

Whenever possible, we strove to work at the greatest reasonable level of generality. At times the unified nature of Mathlib made it possible to establish results at a level of generality beyond that of the standard references, including Bourbaki. A good example is the basic result that finite-dimensional Lie algebras possess a maximal solvable ideal.

As we have seen, Lie algebras admit a notion of being solvable. For the purposes of this discussion, the precise meaning is unimportant. What is important is that if I, J are ideals of a Lie algebra and if, regarding them as Lie algebras in their own right, they are both solvable, then their sum $I + J$ is solvable. Mathlib knows this fact. Indeed here is the statement and proof for a Lie algebra L over a commutative ring R ☐ :

```
instance is_solvable_add {I J : lie_ideal R L}
  [hI : is_solvable R I] [hJ : is_solvable R J] :
  is_solvable R ↑(I + J) :=
begin
  tactic.unfreeze_local_instances,
  obtain ⟨k, hk⟩ := hI,
  obtain ⟨l, hl⟩ := hJ,
  exact ⟨⟨k+l,
    lie_ideal.derived_series_add_eq_bot hk hl⟩⟩,
end
```

The (solvable) radical of a Lie algebra is the sum of all solvable ideals, or more precisely, the supremum of the subset of solvable ideals in the complete lattice of ideals of a Lie algebra. Here is the definition in Mathlib ☐ :

```
def radical :=
Sup { I : lie_ideal R L | is_solvable R I }
```

It is clear that if R is a field and L is finite-dimensional then the radical itself is finite-dimensional and can thus be represented as a sum of finitely-many solvable ideals. By iterating `is_solvable_add` we thus see the radical is solvable. This was the greatest level of generality in which this fact was established in any reference the author could find.

However it is not necessary to make such strong assumptions. Indeed the result is true over any commutative ring R as long as L is Noetherian, as can be seen from the following proof in Mathlib ☐ :

```
instance radical_is_solvable [is_noetherian R L] :
  is_solvable R (radical R L) :=
begin
  have h := lie_submodule.
    well_founded_of_noetherian R L L,
  rw ← complete_lattice.
    is_sup_closed_compact_iff_well_founded at h,
  refine
    h { I : lie_ideal R L | is_solvable R I } _ _,
  { use ⊥,
    exact lie_algebra.is_solvable_bot R L, },
  { intros I J hI hJ,
```

```
    apply lie_algebra.is_solvable_add R L;
    [exact hI, exact hJ], },
end
```

The key lemma is `complete_lattice.is_sup_closed_-`
`compact_iff_well_founded` ☑ which the author added
to the lattice theory library for the purposes of proving
`radical_is_solvable`. This addition was only possible be-
cause Mathlib already contained a comprehensive lattice
theory library and numerous key results about well-founded
relations. Furthermore the lemma `lie_submodule.well_-`
`founded_of_noetherian` ☑ ultimately depends upon re-
sults which were originally introduced to Mathlib for the
purposes of formalising results about Noetherian modules
over commutative rings with a view toward algebraic geom-
etry.

Different people with different aims in different corners
of Mathlib are enabling each other to push boundaries into
new territory.

5 Lie algebras from associative algebras

Any associative algebra A carries a natural Lie algebra struc-
ture via the definition:

$$[x, y] = xy - yx.$$

This is an extremely important[12] class of Lie algebras which
we needed early on.

We thus registered the following data-bearing typeclass
instance ☑ :

```
instance {A : Type*} [ring A] : has_bracket A A :=
⟨λ x y, x*y - y*x⟩
```

together with instances containing proofs that this definition
satisfies the required axioms ☑ :

```
instance {A : Type*} [ring A] : lie_ring A :=
```

and ☑ :

```
instance {R A : Type*} [comm_ring R] [ring A]
  [algebra R A] : lie_algebra R A :=
```

We also established basic properties about this correspon-
dence. In particular we needed to establish that a morphism
of associative algebras can be regarded as a morphism of Lie
algebras ☑ :

```
def alg_hom.to_lie_hom (f : A →ₐ[R] B) :
  A →ₗ[R] B :=
```

All of the above followed easily using standard tactics.

---

[12]Indeed any Lie algebra that injects into its universal enveloping algebra
is a Lie subalgebra of such a Lie algebra, or better yet (in finite dimensions)
see Ado's theorem [5] I §7.3.

6 Skew adjoint endomorphisms

If an associative algebra A carries appropriate additional
structure, it contains distinguished Lie subalgebras when
regarded as a Lie algebra in the sense of section 5.

The most important examples of this phenomenon oc-
cur when A is the endomorphisms of a module M, i.e. $A = \text{End}(M)$. These include:

1. If M is free with finite rank, A contains the distin-
   guished Lie subalgebra of trace zero elements.
2. If M carries a bilinear form, A contains the distin-
   guished Lie subalgebra of skew-adjoint elements.
3. If M carries a bilinear multiplication, A contains the
   distinguished Lie subalgebra of derivations.
4. If M carries both a bilinear form and a compatible
   bilinear multiplication, A contains the distinguished
   Lie subalgebra of skew-adjoint derivations.

Our focus here is the second item above: the Lie subalgebra
of skew-adjoint endomorphisms obtained from a module
carrying a bilinear form.

If the bilinear form is the dot product, the skew-adjoint
endomorphisms are just the skew-symmetric matrices of
section 1.1, but as we shall see, by allowing more general
bilinear forms, we obtain more general Lie algebras.

Informally, given R-modules M, M' carrying bilinear forms:

$$B : M \times M \to R,$$
$$B' : M' \times M' \to R,$$

we say that linear maps $f : M \to M'$ and $g : M' \to M$ are
adjoint[13] iff:

$$B'(fx, y) = B(x, gy),$$

for all $x : M, y : M'$. Building on top of the existing theory of
bilinear forms, we formalised this as follows ☑ :

```
def bilin_form.is_adjoint_pair :=
∀ {x y}, B' (f x) y = B x (g y)
```

In the special case $M = M'$ and $B = B'$ we say that f is
self-adjoint if:

$$B(fx, y) = B(x, fy),$$

for all $x, y : M$ and we say f is skew-adjoint if:

$$B(fx, y) = -B(x, fy),$$

for all $x, y : M$.

We formalised these concepts in Mathlib as ☑ :

```
def is_self_adjoint := is_adjoint_pair B B f f
```

and ☑ :

```
def is_skew_adjoint := is_adjoint_pair B B f (-f)
```

and proved that the subsets of self-adjoint and skew-adjoint
endomorphisms are both submodules of $\text{End}(M)$. To avoid
code duplication we introduced the concept of 'pair-self-
adjointness' ☑ :

---

[13]More precisely, f is left adjoint to g.

```
def is_pair_self_adjoint :=
is_adjoint_pair B B′ f f
```

where we remain specialised to a single module $M = M'$ but bring back the second bilinear form B'.

When a module carries a single bilinear form B, the usual concept of self-adjointness is pair-self-adjointness for the pair of bilinear forms (B, B) and the usual concept of skew-adjointness is pair-self-adjointness for the pair of bilinear forms $(-B, B)$. The relevant formal statement is ☐ :

```
lemma is_skew_adjoint_iff_neg_self_adjoint :
  B.is_skew_adjoint f ↔
  is_adjoint_pair (-B) B f f :=
```

We then proved that for any pair of bilinear forms, the subset of pair-self-adjoint endomorphisms forms a submodule of $\text{End}(M)$.

Restricting our attention to just the skew-adjoint endomorphisms[14], we then proved ☐ :

```
lemma bilin_form.is_skew_adjoint_bracket
  (f g : module.End R M)
  (hf : f ∈ B.skew_adjoint_submodule)
  (hg : g ∈ B.skew_adjoint_submodule) :
  ⁅f, g⁆ ∈ B.skew_adjoint_submodule :=
```

and so deduced that they form a Lie subalgebra, as required.

Finally, recalling that a square matrix J defines a bilinear form on vectors:

$$(v, w) \mapsto v^T J w,$$

and that another square matrix A defines an endomorphism of vectors:

$$v \mapsto Av,$$

we introduced the concept of adjointness for matrices. This turns out to be ☐ :

```
def matrix.is_adjoint_pair := Aᵀ · J′ = J · B
```

We constructed an API for matrices similar to the one for bilinear forms and proved that the notions of adjointness correspond:

```
lemma matrix_is_adjoint_pair_bilin_form :
  matrix.is_adjoint_pair J J′ A B ↔
  bilin_form.is_adjoint_pair
    J.to_bilin_form J′.to_bilin_form
    A.to_lin B.to_lin :=
```

7 The classical Lie algebras

We made an early effort to construct the classical Lie algebras:

- the special linear algebra $\mathfrak{sl}(n, R)$,
- the (special) orthogonal algebra $\mathfrak{so}(n, R)$,
- the symplectic algebra $\mathfrak{sp}(n, R)$,

---

[14]In fact the subset of self-adjoint endomorphisms also carries some extra structure: they form a Jordan subalgebra under the product $x \circ y = xy + yx$.

for a finite type n and commutative ring R.

These were all constructed as Lie subalgebras of the algebra of square matrices with entries in R. Note that this already includes a design choice: the algebra of $n \times n$ matrices is equivalent, but not equal, to the algebra of endomorphisms of the free module on n. In Mathlib, the free module on n is denoted n → R. We thus also formalised the equivalence of Lie algebras ☐ :

```
lie_equiv_matrix′ :
  module.End R (n → R) ≃ₗ⁅R⁆ matrix n n R
```

so that results could be transported. It is not clear if constructions as matrices or endomorphisms should be preferred.

7.1 The special linear algebra

The Lie subalgebra $\mathfrak{sl}(n, R)$ is the square matrices of trace zero. We thus added a definition of the trace of a matrix and proved basic properties including:

$$\text{tr}(AB) = \text{tr}(BA),$$

for matrices A, B with entries in a commutative semiring. In fact this was deduced from the following result about transposes which does not assume commutativity ☐ :

```
@[simp] lemma trace_transpose_mul
  (A : matrix m n R) (B : matrix n m R) :
  trace n R R (Aᵀ · Bᵀ) = trace m R R (A · B) :=
finset.sum_comm
```

Note that the proof is a direct invocation of an existing lemma `finset.sum_comm`; this is unsurprising, Mathlib contains a comprehensive library about finite sums and products. With the above in hand, it was easy to prove ☐ :

```
@[simp] lemma matrix_trace_commutator_zero
  (A B : matrix n n R) :
  matrix.trace n R R ⁅A, B⁆ = 0 :=
```

from which it follows that $\mathfrak{sl}(n, R)$ is indeed a Lie subalgebra.

7.2 The skew-adjoint algebras

We already met $\mathfrak{so}(n, R)$ in section 1.1, it is the subset of matrices A such that:

$$A^T = -A.$$

More generally, by the results of section 6, given any square matrix J, the subset of matrices A such that:

$$A^T J = -JA \tag{7.1}$$

form a Lie subalgebra. The remaining classical Lie algebra $\mathfrak{sp}(n, R)$ can be defined as the subset of $2n \times 2n$ matrices A satisfying (7.1) with:

$$J = \begin{bmatrix} 0_n & -I_n \\ I_n & 0_n \end{bmatrix}$$

where 0_n is the $n \times n$ zero matrix and I_n is the identity matrix. Here is the formal definition of the above ☐ :

```
def J : matrix (n ⊕ n) (n ⊕ n) R :=
matrix.from_blocks 0 (-1) 1 0

def sp :
  lie_subalgebra R (matrix (n ⊕ n) (n ⊕ n) R) :=
  skew_adjoint_matrices_lie_subalgebra (J n R)
```

However this is not the end of the story: different choices of J yield alternate models of the classical Lie algebras. These alternate models are different as Lie subalgebras but equivalent as abstract Lie algebras. For example, the subset of matrices A satisfying (7.1) with:

$$J = \begin{bmatrix} 0_n & I_n \\ I_n & 0_n \end{bmatrix} \tag{7.2}$$

is equivalent to $\mathfrak{so}(2n, R)$, and there is a corresponding alternate model for the odd case $\mathfrak{so}(2n+1, R)$. Furthermore, each model has its advantages[15] so it is important to cater for the various choices of J.

We thus also formalized these alternate models, together with relevant proofs of equivalence as abstract Lie algebras. For example, here is the formal definition of (7.2) ☐:

```
def JD : matrix (n ⊕ n) (n ⊕ n) R :=
matrix.from_blocks 0 1 1 0
```

and here is the corresponding alternate model ☐:

```
def type_D :=
skew_adjoint_matrices_lie_subalgebra (JD n R)
```

and the statement of its equivalence to a model with a diagonal J matrix ☐:

```
noncomputable def type_D_equiv_so′
  [invertible (2 : R)] :
  type_D n R ≃ₗ[R] so′ n n R :=
```

8 General non-associative algebra and the free Lie algebra

We formalised a construction of the free Lie algebra on a type X with coefficients in a commutative ring R. Here is the statement of the universal property (i.e., left adjointness) as stated in Mathlib with respect to a Lie algebra L ☐:

```
def lift :
  (X → L) ≃ (free_lie_algebra R X →ₗ[R] L) :=
```

This definition, and the proof of its universality, was hard-won and is worth comment.

The construction is to take a quotient of the free non-unital, non-associative algebra[16] on X with coefficients in R. We thus needed to define free_non_unital_non_assoc_algebra and prove its universal property ☐:

---

[15]The main advantage of the model obtained using (7.2) is that there is a Cartan subalgebra of diagonal matrices.

[16]Strictly speaking the terminology should be 'not-necessarily-unital' and 'not-necessarily-associative' but it is common and easier to say simply 'non-unital' and 'non-associative'.

```
def lift :
  (X → A) ≃ non_unital_alg_hom R
    (free_non_unital_non_assoc_algebra R X) A :=
```

In the above, A is a general non-unital, non-associative algebra and non_unital_alg_hom is the type of morphisms of such algebras.

Establishing the above result while adhering to the standards of Mathlib was not straightforward. The problem was that Mathlib's theories of rings and algebras were entirely specialised to the unital, associative setting. To handle this without fragmenting the algebraic hierarchy, it was necessary to insert new classes, notably the typeclass non_unital_non_assoc_semiring, low down in the hierarchy. In a library as large as Mathlib, such changes are significant undertakings.

Eric Wieser generously took on this challenge[17] and also showed how to encode a general non-unital, non-associative algebra:

```
variables {R A : Type*}
  [comm_ring R] [non_unital_non_assoc_semiring A]
  [module R A] [is_scalar_tower R A A]
  [smul_comm_class R A A]
```

After Wieser's work (see also [24]) it was essentially straightforward to define free_non_unital_non_assoc_algebra by dropping the assumption of associativity in the existing monoid algebra construction and proving the corresponding universal property with respect to a magma M ☐:

```
def lift_magma [has_mul M] :
  mul_hom M A ≃ non_unital_alg_hom R
    (monoid_algebra R M) A :=
```

With this in hand, the author was able to make the key definition:

```
def free_non_unital_non_assoc_algebra :=
monoid_algebra R (free_magma X)
```

and the corresponding universal property followed trivially.

Finally the free Lie algebra was constructed as a quotient using the following relation ☐:

```
local notation `lib` :=
free_non_unital_non_assoc_algebra

inductive rel : lib R X → lib R X → Prop
| lie_self (a : lib R X) : rel (a*a) 0
| leibniz_lie (a b c : lib R X) :
    rel (a*(b*c)) (((a*b)*c) + (b*(a*c)))
| smul (t : R) (a b : lib R X) :
    rel a b → rel (t·a) (t·b)
| add_right (a b c : lib R X) :
    rel a b → rel (a+c) (b+c)
| mul_left (a b c : lib R X) :
    rel b c → rel (a*b) (a*c)
| mul_right (a b c : lib R X) :
```

---

[17]This pull request shows what was required after all other preparatory work had been completed.

```
    rel a b → rel (a*c) (b*c)
```

and its universal property followed easily.

It should be noted that the use of `inductive` above was necessary because Mathlib does not yet contain a theory of ideals and their quotients for general non-associative algebras. Filling this gap would improve the construction even further, though the benefit would be slight.

We should say that it would have been easy to establish what we needed without any of the above work by ignoring most of Mathlib's algebra library and taking a quotient of an inductively-defined type with a constructor for every term of X as well as separate constructors corresponding to the scalar action, additive law, and Lie bracket. We rejected this low-level approach because it would require a significant quantity of single-use code, because it would be harder to maintain, because the alternative approach was an opportunity to start developing a general theory of non-associative rings and algebras, and because this is very unlikely to be the approach that the informal mathematician would take.

We should also say that we rejected an approach that constructs the free Lie algebra as the smallest Lie subalgebra of the free unital, associative algebra containing the generating type X. This can be expressed in Lean as:

```
lie_subalgebra.lie_span R (free_algebra R X)
  (set.range (free_algebra.ι R))
```

This approach is mathematically appealing but the proof that this construction satisfies the universal property appears to need a powerful version of the Poincaré-Birkhoff-Witt theorem (see [18] as well as [5] I §2.7, §3.1).

9 The exceptional Lie algebras

We assume for now that the coefficients R form an algebraically closed field of characteristic zero. The work under discussion does not make this assumption but it will simplify the discussion here if we do.

There are numerous beautiful ways to construct the five exceptional Lie algebras $\mathfrak{g}_2, \mathfrak{f}_4, \mathfrak{e}_6, \mathfrak{e}_7, \mathfrak{e}_8$ (e.g., [1], [21], [23], [11], [12]) but the most useful construction from the point of view of proving the classification theorem (see section 10) is probably[18] an approach due to Serre [19]. This approach takes a square matrix of integers as input and yields a Lie algebra. When the matrix is the Cartan matrix of a semisimple Lie algebra, we recover the corresponding Lie algebra, together with a splitting Cartan subalgebra.

If A is an $l \times l$ Cartan matrix, the corresponding Lie algebra is defined to be the quotient of the free Lie algebra on $3l$ generators: $H_1, H_2, \ldots H_l, E_1, E_2, \ldots, E_l, F_1, F_2, \ldots, F_l$ by the following relations:

$$[H_i, H_j] = 0$$
$$[E_i, F_i] = H_i$$

---

[18]The only real competitor being [12].

$$[E_i, F_j] = 0 \quad \text{if } i \neq j$$
$$[H_i, E_j] = A_{ij} E_j$$
$$[H_i, F_j] = -A_{ij} F_j$$
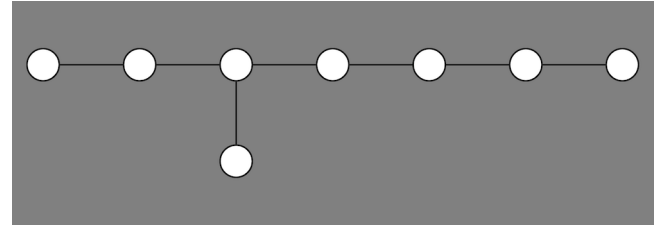$$\mathrm{ad}(E_i)^{1-A_{ij}}(E_j) = 0 \quad \text{if } i \neq j$$
$$\mathrm{ad}(F_i)^{1-A_{ij}}(F_j) = 0 \quad \text{if } i \neq j$$

Thanks to the construction of the free Lie algebra described in section 8, it was easy to implement Serre's construction in Mathlib and thus to define the exceptional Lie algebras. For example, here is Mathlib's definition of $\mathfrak{f}_4$ ☐ :

```
def cartan_matrix.F₄ : matrix (fin 4) (fin 4) ℤ :=
![![ 2, -1,  0,  0],
  ![-1,  2, -2,  0],
  ![ 0, -1,  2, -1],
  ![ 0,  0, -1,  2]]

abbreviation f₄ :=
cartan_matrix.F₄.to_lie_algebra R
```

What's more, thanks to Ed Ayers's Lean Widgets [2], it was easy to generate the Dynkin diagram corresponding to a Cartan matrix. For example, here is a screenshot from the author's proof-of-concept widget, written in Lean, which reads Mathlib's definition of the E_8 Cartan matrix and renders the corresponding Dynkin diagram:

Mathlib's E_8 Dynkin diagram rendered using Lean

Finally, we should confess that we have yet to prove almost anything about the exceptional Lie algebras. The path is clear but much work remains until we can prove key facts such as their simplicity, or for example:

```
def dimension_g₂ : Prop := finrank (g₂ ℂ) = 14
```

10 Stating the classification

An important milestone, passed in the course of this work, was teaching Lean the statements of the classification of semisimple Lie algebras. Using the above work, we defined what it means for a Lie algebra to be simple ☐ :

```
class is_simple extends
  lie_module.is_irreducible R L L : Prop :=
(non_abelian : ¬is_lie_abelian L)
```

and likewise what it means to be semisimple ☐ :

```
class is_semisimple : Prop :=
(semisimple : radical R L = ⊥)
```

Basic related results like the fact that a simple Lie algebra is semisimple were also proved ⬚ :

```
instance is_semisimple_of_is_simple
  [h : is_simple R L] : is_semisimple R L :=
```

Modulo some boilerplate to establish notation, the classification statements are then:

```
variables (K L : Type*)
```

```
/- Let K be an algebraically closed field of
   characteristic zero. -/
variables [field K] [is_alg_closed K]
  [char_zero K]
```

```
/- Let L be a finite-dimensional Lie algebra
   over K. -/
variables [lie_ring L] [lie_algebra K L]
  [finite_dimensional K L]
```

```
def simple_classification : Prop :=
  is_simple K L ↔
  ((L ≅ₗ⟦K⟧ g₂ K) ∨
   (L ≅ₗ⟦K⟧ f₄ K) ∨
   (L ≅ₗ⟦K⟧ e₆ K) ∨
   (L ≅ₗ⟦K⟧ e₇ K) ∨
   (L ≅ₗ⟦K⟧ e₈ K) ∨
   (∃ l, (L ≅ₗ⟦K⟧ sl n K) ∧ 1 < n) ∨
   (∃ l, (L ≅ₗ⟦K⟧ sp n K) ∧ 2 < n) ∨
   (∃ l, (L ≅ₗ⟦K⟧ so n K) ∧ 4 < n ∧ n ≠ 6))
```

```
def semisimple_classification : Prop :=
  is_semisimple K L ↔
  ∃ n (I : fin n → lie_ideal K L),
    (L ≅ₗ⟦K⟧ (⊕ i, I i)) ∧ ∀ i, is_simple K (I i)
```

Note that `simple_classification` contains several of the "exceptional isomorphisms". E.g., $\mathfrak{so}(6)$ is simple so it must be isomorphic to one of the other algebras on the list. For dimensional reasons, this has to be $\mathfrak{sl}(4)$. Likewise for the other cases excluded.

Note also that if we pursue a proof of the classification, we will probably restate `simple_classification` in terms of algebras constructed from Cartan matrices of types A, B, C, D rather than the models $\mathfrak{sl}$, $\mathfrak{sp}$, $\mathfrak{so}$ defined in terms of matrices.

11 Weight spaces and root spaces

Just as a key tool when studying the behaviour of a linear operator is to decompose the space on which it acts into a sum of (generalised) eigenspaces, a key tool when studying a Lie module M of Lie algebra L is to decompose M into a sum of simultaneous eigenspaces of x, as x ranges over L. These simultaneous generalised eigenspaces are known as the weight spaces of M.

When L is nilpotent, it follows from the binomial theorem that weight spaces are Lie submodules. Even when L is not nilpotent, it may be useful to study its Lie modules by restricting them to a nilpotent subalgebra (e.g., a Cartan subalgebra). In the particular case when we view L as a module over itself via the adjoint action, the weight spaces of L restricted to a nilpotent subalgebra are known as root spaces.

We formalised these concepts in Lean as follows ⬚ :

```
def pre_weight_space (χ : L → R) :
  submodule R M :=
⊓ (x : L), (to_endomorphism R L M x).
  maximal_generalized_eigenspace (χ x)
```

and ⬚ :

```
def weight_space [lie_algebra.is_nilpotent R L]
    (χ : L → R) : lie_submodule R L M :=
{ lie_mem := ...
  .. pre_weight_space M χ }
```

and finally ⬚ :

```
abbreviation root_space (H : lie_subalgebra R L)
  [lie_algebra.is_nilpotent R H] (χ : H → R) :
  lie_submodule R H L := weight_space L χ
```

There is actually quite a lot going on above. For one thing, the definition of `root_space` requires Lean to recognise that we can regard L as a Lie module over H. This is achieved (in part) via the following typeclass instance registered far away in the Lie subalgebra theory ⬚ :

```
instance (H : lie_subalgebra R L) :
  lie_module R H M :=
```

This is a good example of typeclasses working well: the informal mathematician would not waste space being explicit about details like this here, and thanks to the typeclass system, the formal mathematician need not do so either.

More significantly, the proof of `lie_mem` which we have omitted in the above definition of `weight_space` is not quite trivial. The key step is the following lemma (applied with $\chi_1 = 0$ and $\chi_2 = \chi$) ⬚ :

```
lemma
  lie_mem_pre_weight_space_of_mem_pre_weight_space
  {χ₁ χ₂ : L → R} {x : L} {m : M}
  (hx : x ∈ pre_weight_space L χ₁)
  (hm : m ∈ pre_weight_space M χ₂) :
  ⁅x, m⁆ ∈ pre_weight_space M (χ₁ + χ₂) :=
```

The proof is similar (though not quite the same) as the proof that if a, b are two commuting nilpotent elements of a semiring, then their sum $a + b$ is nilpotent. The standard proof of this is to apply the binomial theorem. In our case, for each element of L we obtain commuting elements of $\mathrm{End}(L \otimes M)$ and again the proof is to apply the binomial theorem for this ring. Happily, tensor products, and a general version of the binomial theorem had already been formalised in Mathlib, so we could just appeal to this theory.

The function χ appearing in these definitions is the candidate family of eigenvalues, and is said to be a weight or root when the corresponding weight space or root space is non-empty ⬀ :

```
def is_weight : Prop := weight_space M χ ≠ ⊥
```

and ⬀ :

```
abbreviation is_root := is_weight H L
```

Weights and roots are the start of a sizeable branch of Lie theory. Various foundational results such as ⬀ :

```
def root_space_weight_space_product
  (χ₁ χ₂ χ₃ : H → R) (hχ : χ₁ + χ₂ = χ₃) :
  (root_space H χ₁) ⊗[R]
  (weight_space M χ₂) →ₗ[R,H]
  weight_space M χ₃ :=
```

are in place but much work remains to be done.

12 Final words

12.1 Trivial proofs should be trivial

When building a library the size of Mathlib, one must constantly try to be mindful of how one's work will scale as more is built upon it. One metric for the health of a particular area of the library is how much effort one is forced to put into proving trivialities. We share an example of what this looks like when things go well.

Given a type X and a commutative ring R one can use this data to build the free unital, associative algebra $A(R, X)$. However, there is another way to build a unital, associative algebra from this data: one first builds the free Lie algebra $L(R, X)$ and then takes its universal enveloping algebra $U(L(R, X))$. A simple diagram chase reveals that these are the same, in particular:

$$U(L(R, X)) \simeq A(R, X).$$

Mathlib knows this fact; here is the proof (using some notational shortcuts for readability) ⬀ :

```
def universal_enveloping_equiv_free_algebra :
  universal_enveloping_algebra R
    (free_lie_algebra R X) ≃ₐ[R]
  free_algebra R X :=
alg_equiv.of_alg_hom
  (liftu R $ liftl R $ ιa R)
  (lifta R $ (ιu R) ∘ (ιl R))
  (by { ext, simp, })
  (by { ext, simp, })
```

The point of the above is the two lines that read (by { ext, simp, }). This is the Lean code discharging the proof obligations which correspond to the informal mathematician's diagram chase. It is encouraging that they are trivial applications of standard tactics.

12.2 The Lie algebra of a Lie group

Far away in a different corner of Mathlib, Sébastien Gouëzel has developed a theory of differentiable manifolds. Building on top of this, Nicolò Cavalleri, under the supervision of Anthony Bordg, has defined Lie groups and has used it to construct the Lie algebra associated to a Lie group [4].

12.3 A specific example

In section 1.2 we motivated the formalisation of Lie algebras by highlighting areas of mathematics where they appear. However there also exist specific examples where their formalisation would help directly. A striking case is the recent paper of Le Floch and Smilga [16]. This is a pure mathematics paper in which an interesting abstract problem is reduced to a finite calculation. The problem was settled by running an algorithm on a computer, and the authors used SageMath, LiE [22], and a custom Coq program[19] written specifically for their purpose.

It is not hard to imagine developing the Lie theory library described here into a platform upon which calculations such as that of Le Floch and Smilga could be run, and formally verified.

12.4 Proof of classification

With the statement of the classification theorem formalised, it is tempting to consider formalising its proof. Several key concepts such as Cartan subalgebras, weight spaces, and root spaces have also been formalised. The evidence so far is that formalising a proof of the classification would be non-trivial but is absolutely within reach.

Acknowledgements

References

[1] J. F. Adams. 1996. *Lectures on exceptional Lie groups*. University of Chicago Press, Chicago, IL. xiv+122 pages. With a foreword by J. Peter May, Edited by Zafer Mahmud and Mamoru Mimura.

[2] Edward Ayers. 2021. Widgets: interactive output in VSCode. In *Lean Together 2021, January 4–7, 2021*. url.

[3] John Baez and John Huerta. 2010. The algebra of grand unified theories. *Bull. Amer. Math. Soc. (N.S.)* 47, 3 (2010), 483–552. https://doi.org/10.1090/S0273-0979-10-01294-2

---

[19]The code for this is available at https://github.com/blefloch/lie-algebra-w0.

[4] Anthony Bordg and Nicolò Cavalleri. 2021. Elements of Differential Geometry in Lean A Report for Mathematicians. In *14th Conference on Intelligent Computer Mathematics CICM 2021, Timisoara, Romania, July 26–31, 2021*. url.

[5] Nicolas Bourbaki. 1998. *Lie groups and Lie algebras. Chapters 1–3.* Springer-Verlag, Berlin. xviii+450 pages. Translated from the French, Reprint of the 1989 English translation.

[6] Nicolas Bourbaki. 2002. *Lie groups and Lie algebras. Chapters 4–6.* Springer-Verlag, Berlin. xii+300 pages. Translated from the 1968 French original by Andrew Pressley.

[7] Nicolas Bourbaki. 2005. *Lie groups and Lie algebras. Chapters 7–9.* Springer-Verlag, Berlin. xii+434 pages. Translated from the 1975 and 1982 French originals by Andrew Pressley.

[8] Mario Carneiro. 2019. The Type Theory of Lean. arXiv:https://github.com/digama0/lean-type-theory/releases Master thesis.

[9] Élie Cartan. 1894. Sur la structure des groupes de transformations finis et continus. *Librairie Nony* (1894), 1–156. https://archive.org/details/surlastructured00bourgoog/page/n8/mode/2up

[10] A. J. Coleman. 1989. The greatest mathematical paper of all time. *Math. Intelligencer* 11, 3 (1989), 29–38. https://doi.org/10.1007/BF03025189

[11] José Figueroa-O'Farrill. 2008. A geometric construction of the exceptional Lie algebras F_4 and E_8. *Comm. Math. Phys.* 283, 3 (2008), 663–674. https://doi.org/10.1007/s00220-008-0581-7

[12] Meinolf Geck. 2017. On the construction of semisimple Lie algebras and Chevalley groups. *Proc. Amer. Math. Soc.* 145, 8 (2017), 3233–3247. https://doi.org/10.1090/proc/13600

[13] Wilhelm Killing. 1888. Die Zusammensetzung der stetigen endlichen Transformations-gruppen. *Math. Ann.* 31, 2 (1888), 252–290. https://doi.org/10.1007/BF01211904

[14] Wilhelm Killing. 1888. Die Zusammensetzung der stetigen endlichen Transformationsgruppen. *Math. Ann.* 33, 1 (1888), 1–48. https://doi.org/10.1007/BF01444109

[15] Wilhelm Killing. 1889. Die Zusammensetzung der stetigen endlichen Transformations-gruppen. *Math. Ann.* 34, 1 (1889), 57–122. https://doi.org/10.1007/BF01446792

[16] Bruno Le Floch and Ilia Smilga. 2018. Action of Weyl group on zero-weight space. *C. R. Math. Acad. Sci. Paris* 356, 8 (2018), 852–858. https://doi.org/10.1016/j.crma.2018.06.005

[17] Sophus Lie. 1880. Theorie der Transformationsgruppen I. *Math. Ann.* 16, 4 (1880), 441–528. https://doi.org/10.1007/BF01446218

[18] Mathoverflow. 2021. Is the natural map from the free Lie algebra to the free associative algebra injective?. In *MathOverflow 2021*. url.

[19] Jean-Pierre Serre. 1987. *Complex semisimple Lie algebras.* Springer-Verlag, New York. x+74 pages. https://doi.org/10.1007/978-1-4757-3910-7 Translated from the French by G. A. Jones.

[20] The mathlib community. 2020. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*. 367–381. https://doi.org/10.1145/3372885.3373824

[21] J. Tits. 1966. Algèbres alternatives, algèbres de Jordan et algèbres de Lie exceptionnelles. I. Construction. *Nederl. Akad. Wetensch. Proc. Ser. A 69 = Indag. Math.* 28 (1966), 223–237.

[22] Marc A. A. van Leeuwen, Arjeh M. Cohen, and Bert Lisser. 2000. *LiE, A package for Lie Group Computations.* Computer Algebra Nederland, Amsterdam. http://wwwmathlabo.univ-poitiers.fr/~maavl/LiE/

[23] E. B. Vinberg. 2005. Construction of the exceptional simple Lie algebras. In *Lie groups and invariant theory*. Amer. Math. Soc. Transl. Ser. 2, Vol. 213. Amer. Math. Soc., Providence, RI, 241–242. https://doi.org/10.1090/trans2/213/15 Translated from Trudy Sem. Vekt. Tenz. Anal. **13** (1966), 7–9.

[24] Eric Wieser. 2021. Scalar actions in Lean's mathlib. In *14th Conference on Intelligent Computer Mathematics CICM 2021, Timisoara, Romania, July 26–31, 2021*. to appear.