The Lean mathematical library

The mathlib Community*

Abstract

This paper describes mathlib, a community-driven effort to build a unified library of mathematics formalized in the Lean proof assistant. Among proof assistant libraries, it is distinguished by its dependently typed foundations, focus on classical mathematics, extensive hierarchy of structures, use of large- and small-scale automation, and distributed organization. We explain the architecture and design decisions of the library and the social organization that has led us here.

1 Introduction

Since the first mechanized proof-checking systems were released, there has been continual effort to develop libraries of formal definitions and proofs. Every major system has at least one library that can serve as a base for further formalizations. The contents, organizations, and purposes of these libraries vary considerably. Some are small, static cores bundled with the system itself; some are sprawling repositories that solicit contributions like a journal; some focus on a narrow subject area, while others are more broad-minded in their topics.

This paper describes mathlib, a formal library developed for the Lean proof assistant [20]. As a community-driven effort with dozens of contributors, there is no central organization to mathlib; it has arisen from the desires of its users to develop a repository of formal mathematical proofs. We are certainly not the first to profess this goal [1], nor is our library large in comparison to others. However, its organizational structure, focus on classical mathematics, and inclusion of automation distinguish it in the space of proof assistant libraries. We aim here to explain our design decisions and the ways in which mathlib has been put to use.

Relative to most modern proof assistant libraries, many of the contributors to mathlib have an academic background in pure mathematics. This has significantly influenced the contents and direction of the library. It is a goal of many in the community to support the formalization of modern, research-level mathematics, and various projects discussed in §7.2 suggest that we are approaching this point.

*This paper describes a community effort with 68 contributors, listed in the acknowledgments. Rather than attributing authorship to a subset, we have decided to release this paper under a collective pseudonym.

Preprint, 2019,
2019.

1.1 A history of mathlib and Lean 3

The Lean project was started by Leonardo de Moura in 2013 [20]. Its most recent version, Lean 3, was released in early 2017 [22]. A new version is under development [54]. In the summer of 2017, much of the core library for Lean was factored out of the system repository. This code became the base for mathlib. The project was initially led by Mario Carneiro and Johannes Hölzl at Carnegie Mellon University, and attracted a growing number of users, who were drawn by the system documentation [5], the library's focus on classical mathematics, and the real-time chat on Zulip.

Over the two years since the split, mathlib has grown from 15k to 130k lines of code (LOC), excluding blank lines and comments. Contributions have been made by 68 people and are managed by a team of 11 maintainers. It is the de facto standard library for both programming and proving in Lean 3. The surrounding community has developed infrastructure, promotional materials, and university courses based on mathlib.

2 Lean

Like Coq, Lean uses a system of dependent types based on the calculus of inductive constructions (CIC) [49]. Lean has a noncumulative hierarchy of universes `Prop`, `Type`, `Type 1`, `Type 2`, `Type 3` … The bottom universe `Prop` is *impredicative*, meaning that quantification of a `Prop` over a larger type is a `Prop`, and *proof-irrelevant*, meaning that all proofs of the same proposition are definitionally (or judgmentally) equal.

Lean adds two axioms to the type theory. The first axiom, `classical.choice`, produces a term of type `T` from a (propositional) proof that `T` is nonempty. This is a type-theoretic statement that implies the set-theoretic axiom of choice. The second axiom, `propext`, states that a bi-implication between two propositions implies that these propositions are equal.

Lean also extends the CIC with *quotient types*. Given an equivalence relation on a type, the quotient type identifies related terms. This feature allows us to work with equality of objects in the quotient and removes the need for setoids.

A key feature of Lean 3 is its *metaprogramming* framework [22]. This allows users to write tactics, commands, and other tools for manipulating the Lean environment directly in the language of Lean itself. In mathlib, we make extensive use of metaprogramming both for powerful automation and specialized commands to ease various tasks when formalizing. We will discuss this more in §6.

Subdirectory	LOC	Declarations
`data`	40862	10443
`topology`	16581	2561
`tactic`	11159	1537
`algebra`	9025	2552
`category_theory`	6255	1505
`analysis`	6233	914
`order`	6183	1503
`set_theory`	6155	1392
`ring_theory`	5614	1068
`measure_theory`	4606	673
`linear_algebra`	4232	768
`computability`	4203	575
`group_theory`	3391	773
`category`	1750	389
`number_theory`	1244	225
`logic`	1191	402
`field_theory`	971	119
`meta`	799	132
`geometry`	610	55
`algebraic_geometry`	192	29
	131256	27615

Table 1. Lines of code, excluding white space and comments, in the top-level directories of the mathlib source code as of October 21, 2019.

3 Contents of mathlib

The mathlib library is designed as a basis for research level mathematics, as well as a standard library for programming in Lean. We build mathlib on top of a small core library, shipped with Lean, which contains 19k LOC. The core library sets up the metaprogramming and tactic framework for Lean, but it also contains much of the algebraic hierarchy and the definitions of basic datatypes, like $\mathbb{N}$, $\mathbb{Z}$, and `list`.

Currently, much of mathlib consists of undergraduate level mathematics. Table 1 gives the size of all top-level directories in the `src` directory. This gives a rough idea of the relative sizes of the different topics in mathlib, but the contents of some directories require some extra explanation.

In the `algebra` library, we expand on the core library algebraic hierarchy, from `semigroup` to `linear_ordered_field`, `module`, and more (§4.1). The results in this folder are focused on computing with elements in these algebraic structures. The structural theory of these algebraic objects is developed in the folders `group_theory`, `ring_theory`, `field_theory`, and `linear_algebra`.

The `data` folder contains the definitions and properties of data structures,including the number systems $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$, sets and subtypes, partial and finitely supported functions, polynomials, lists, multisets, and vectors.

The folders `meta` and `tactic` use Lean's metaprogramming framework (§6) to define custom tactics. The `category` folder develops categorical programming, e.g. as used in Haskell; this differs from the `category_theory` folder, which develops the mathematical theory.

Quotients are heavily used in mathlib to avoid the need for setoids. They are used to define multisets as lists up to permutation, which are in turn used to define finite sets as multisets without duplicates. Quotients are also frequently used in algebra, for example for the definition of a quotient group, the tensor product of modules or the colimit of rings. We also use quotients when defining the Stone-Čech compactification and Cauchy completion, the latter of which is used to define the real and p-adic numbers. Quotients are also used to define cardinals (as types modulo equivalence) and ordinals (as well-ordered types modulo order-isomorphism).

We highlight some of the more advanced mathematical topics contained in mathlib.

The extensive topology library includes theories about uniform spaces, metric spaces, and algebraic topological spaces such as topological groups and rings. A more novel feature is the definition of the Gromov–Hausdorff space, i.e. the space of all nonempty compact metric spaces up to isometry, with its natural (Gromov–Hausdorff) distance, and the proof that it is a Polish space.[1]

The definition of a manifold in mathlib is very general compared to the only other known formalization [37]. In particular, the base field for differentiable manifolds can be any non-discrete normed field, including $\mathbb{R}$, $\mathbb{C}$, and $\mathbb{Q}_p$. Arbitrary models and structure groupoids can be used. Examples currently include C^k and C^∞ manifolds, possibly with boundary or corners. Potential future examples include analytic manifolds, contact or symplectic manifolds, and translation surfaces.

The category theory library includes (co)limits, monadic adjunctions, and monoidal categories. It uses extensive automation to hide easy proofs (§6.2). The category theory library is used in other parts of the library, for example describing the Giry monad in measure theory, or showing that products and equalizers in complete separated uniform spaces can be calculated in the underlying uniform spaces.

Since Lean's type theory contains a universe hierarchy, it has a higher consistency strength than ZFC, and the set theory library defines a model of ZFC. Additionally, cardinals and ordinals are defined using quotients of a universe, along with related notions like ordinal arithmetic, $\aleph_\alpha$, cofinality, inaccessible cardinals, etc. This machinery is used in other parts of the library (§5.3).

Structural results about groups, rings, and fields include Sylow's theorems, the law of quadratic reciprocity, integral

[1] The possibility of formalizing the Gromov–Hausdorff space was the motivation for the switch of a user from Isabelle/HOL to Lean, as it makes heavy use of the machinery of dependent types.

and perfect closures, and Hilbert's basis theorem. The linear algebra library is described in detail in §5.

A library on computability theory [16] proves the undecidability of the halting problem, and a library on the p-adic numbers proves Hensel's lemma [43].

Analysis is a weaker point of mathlib, although the Fréchet derivative and the Bochner integral have been formalized, as well as basic properties of trigonometric functions.

4 Type classes

Type classes are predicates or extra data attached to types, which are systematically inferred by a Prolog-like backtracking search. The idea of using type classes to permit polymorphism over types with a particular structure was pioneered in Haskell [56], and the usefulness of type classes for organizing mathematical structures and theorems on these structures was recognized in Isabelle [31, 57] and Coq [53]. Lean's core library builds on these ideas by using type classes ubiquitously, including for:

- notations, e.g. the `has_add α` type class that is inferred when the + symbol is used on a type α;
- mathematical structures, e.g. the type `ring α` that equips a type α with a ring structure;
- coercions, via the type class `has_coe α β`, a wrapper around the type `α → β` that allows Lean to accept an element of α where a β is expected;
- and decidability of propositions, via the `decidable p` type class, which enables case analysis in constructive contexts and if statements in programming.

This usage is significantly expanded into all parts of mathlib.

Mathematicians and computer scientists often speak of an *algebraic hierarchy*, but insofar as that brings to mind the groups, rings, and fields of an introductory course in abstract algebra, the phrase has the wrong connotations. Lean does have a hierarchy of algebraic structures, but this only scratches the surface. For example, in mathlib, the real numbers are an instance of a normed space, a metric space, a uniform space, and a normal topological space. Morphisms between structures have algebraic properties that also need to be managed. The library instantiates the category of groups and morphisms between them, and the functor mapping a measure space X to the space of probability measures on X is an instance of a monad, namely, the *Giry monad* [25]. To convey the richness of such a network of definitions, we will refer to it rather as a *structure hierarchy*.

4.1 Organizing the structure hierarchy

A fragment of the structure hierarchy in Figure 1 shows the classes that can be derived from a `normed_field` instance, which appears near the top of the hierarchy. The bottom of the graph is populated by notation classes such as `has_add`, `has_le`, `has_one`, etc. The fragment omits structures such as `partial_order` and `lattice`, as well as ordered structures like `ordered_group` and `ordered_ring`.

Most of the classes in the hierarchy are unary: they have the form $C\ \alpha$ where α is the carrier type. All of the classes in the fragment above are unary, but there are some binary classes, such as `module R M` that provides an R-module structure on M. The fragment constitutes less than a third of the current structure hierarchy, which contains 201 unary classes and 266 instances of the form $C\ \alpha \to D\ \alpha$, which only change the class associated to a type.

Except for notation classes, all the classes in mathlib have some independent interest. Uniform spaces are defined because they unify theorems of topological groups and metric spaces; extended metric spaces exist because they appear in constructions such as the unbounded ℓ^p spaces. This does not hold for some core library classes, e.g. `zero_ne_one_class`, which exist only to be mixins for other classes.

Diamonds are common—the structure hierarchy is far from being a tree. Some of these diamonds arise naturally from the mathematics, some arise when a class is defined as a least upper bound of two other classes (such as `comm_monoid`), and others arise as a result of encoding a canonical projection as a parent class.

In some cases, we have a situation in which one structure canonically implies another: for example, a metric space is not generally considered to *contain* a topology as a subcomponent, but the metric uniquely defines a particular topology of interest. We opt to have `metric_space` extend `topological_space`, with an additional constraint asserting that the topology agrees with the induced topology of the metric. This is done because Lean depends on definitional equality of projections and this approach makes more squares commute definitionally. Using this approach, the induced topology on a product metric becomes definitionally equal to the product topology on the induced metrics.

4.1.1 Bundled type classes

When creating a type class, one important design decision is which parts of the definition to put as parameters to the type class and which parts to store within the element itself, accessible via a projection. We refer to a type class as *unbundled* if it has many parameters and *bundled* if the parameters are moved to projections. For example, given the definitions

1. Group = $\{(X, \circ) \mid (X, \circ)$ is a group$\}$
2. group $X = \{\circ \mid (X, \circ)$ is a group$\}$
3. is_group $X\ \circ\ \leftrightarrow\ (X, \circ)$ is a group

we would call Group a bundled definition, is_group an unbundled definition, and group a semi-bundled definition. All of these say essentially the same thing from a mathematical

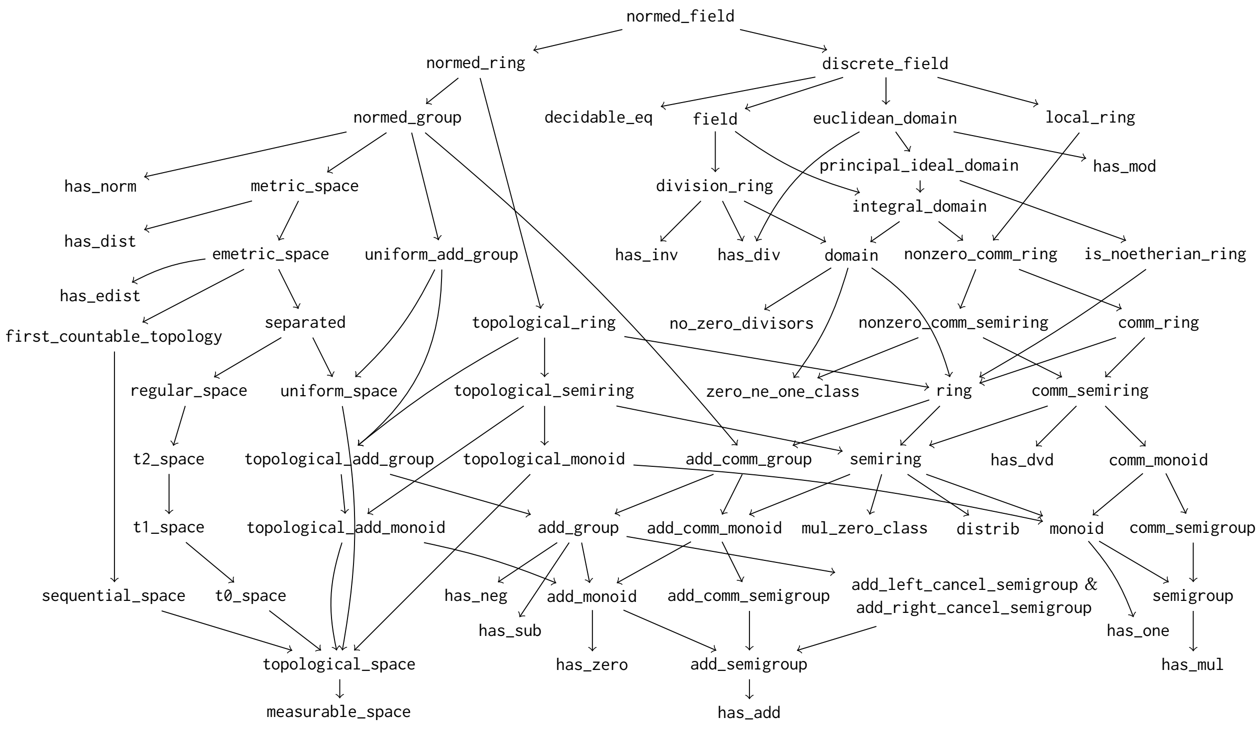

Figure 1. The structure hierarchy underlying normed_field. An edge from S to T indicates that an instance of T can be derived from an instance of S. Some arrows to the leaves are omitted for readability (e.g. from zero_ne_one_class to has_one).

point of view, but the choice of which to use has a significant impact on formalization.

We primarily use semi-bundled definitions for type classes in mathlib, with all operations being bundled except for the carrier types. For use with type classes, fully bundling (as with Group) is not an option: a type class problem should have (essentially) at most one solution, and only the parameters affect the type class search. When the type is exposed but not the operation, as with group X, we can register a canonical group associated to the type X if there is one. For instance, add_group $\mathbb{Z}$ will find the canonical additive group $(\mathbb{Z}, +)$ of addition on integers. By contrast, fully bundled definitions work best when using canonical structures, as in the Mathematical Components library [45].

The drawback of further unbundling, as in is_group X $\circ$, is that it makes matching problems difficult. For example, a theorem saying that $f\ x\ y = f\ y\ x$ under the condition is_comm_group X f, with x, y, f all variables, leads to a higher-order matching problem: f could be substituted for a lambda term. This kind of theorem will always be applicable, since essentially any term can be written in the form $f\ x\ y$ for some choice of f. Especially when the simplification tactic simp is invoked, this can cause a significant performance problem, with many false positives and failed type class searches. With the partially unbundled approach this statement instead has the form add $X\ m\ x\ y$ = add $X\ m\ y\ x$ where m : comm_group X. We need only look for terms which have a literal appearance of the constant add in them rather than considering all terms and excluding them after the fact by a failed type class search.

4.1.2 Bundled morphisms

A similar distinction arises when talking about morphisms between structures. In most cases, a morphism will be a function with an additional property, for example a continuous function or a group homomorphism. In these cases we have two options: to bundle the function with the property, producing a type $\mathrm{Hom}(A, B)$ of structure-respecting functions from A to B, or to have a function $f : A \to B$ and a type class is_hom f asserting that f respects the structure of A and B.

Both options are workable, and mathlib contains traces of both approaches. However, we have found the bundled approach to behave better for a number of reasons:

- It is important for compositions of homs to be a hom, but problems of the form is_hom $(f \circ g)$ can be difficult for type class inference. With bundled homs, one instead defines a custom composition operator $\mathrm{Hom}(A, B) \times \mathrm{Hom}(B, C) \to \mathrm{Hom}(A, C)$ that is used explicitly, obviating the need to search for is_hom $(f \circ g)$.
- The type $\mathrm{Hom}(A, B)$ itself often has interesting structure. For example, the type `M →ₗ[R] N` of R-linear

maps from M to N is an R-module where the zero element is the constant zero map, and addition and scalar multiplication work pointwise (§5.1).

4.2 The decidable type class

The class decidable p is defined essentially as p ⊕ ¬ p: it is a constructive, Type-valued witness to the law of excluded middle at proposition p. Although mathlib primarily deals in classical mathematics, it is useful to track decidability because it can be used in algorithms and for small scale computation in the executable fragment.

The notation if p then t else e is syntactic sugar for ite p t e, where:

```
ite : ∀ (p : Prop) [d : decidable p],
  ∀ {α : Sort u}, α → α → α
```

This function is defined by case analysis on the witness d. In Lean syntax, parentheses () denote explicit arguments, curly brackets {} implicit arguments, and square brackets [] arguments to be inferred by type class resolution. Notice that p is an explicit argument, but p, being a proposition, is erased during execution, with the real condition in d. Morally, we can think of d as a boolean value, but it carries evidence with it connecting it to the truth or falsity of p.

The Mathematical Components library relies heavily on the technique of *small-scale reflection* (from which the tactic language SSReflect gets its name), where many predicates and propositions are defined to live in type bool. The computational behavior of the logic can be used to discharge certain proof obligations by reflection. This means that most properties have two versions, one that is a Prop and one which is a bool, with an additional predicate reflect b p asserting that the boolean value b is true iff p holds.

A proof of decidable p is equivalent to Σ b,reflect b p in this sense, but because it is a type class the management of this side condition is almost entirely automatic. This means we can treat the proposition p as primary, as in the definition of ite, and need not define two versions of every proposition. We can still prove true decidable propositions by computational reflection via dec_trivial:

```
def as_true (c : Prop) [decidable c] : Prop :=
if c then true else false

def of_as_true {c : Prop} [decidable c] :
  as_true c → c := ...

notation `dec_trivial` := of_as_true trivial
```

The considerations in this section apply equally to proving, programming, and metaprogramming in Lean. All propositions can be shown to be (noncomputably) decidable with the choice axiom; this instance is used locally with low priority in mathlib to use declarations with decidability hypotheses in classical mathematics.

4.3 Problems with type classes

As mathlib has grown, given all the aforementioned uses of type classes, and the fact that most of the instances are available in most of the higher-level files, it has become a scalability test of the type class inference system. A file that imports all of mathlib has access to 4256 instances among 386 classes. While it has held up remarkably well, aspects of the inference algorithm have lead to limitations in some areas and performance walls in others.

First, Lean performs a backtracking search on every type class problem. Thus, an instance such as C α ← C α (that is, "to obtain C α it suffices to prove C α") will cause the inference routine to enter a loop, blocking resolution even if a successful path exists elsewhere. As a result, mathlib requires that the complete instance graph be acyclic. This is a global problem but generally single instances can be identified as the culprit when an instance loop is discovered.

Second, and relatedly, because Lean cannot tell when it is retreading the same paths, even if the graph of instances has no cycles, it will traverse all paths through the graph, which is exponential time in the worst case. (On successes it is often significantly less if it gets lucky, but failures will have to search the whole graph, and pathological examples can be constructed in which the search is successful but still exponential time.) Instance searches can take hundreds of thousands of steps through our thousands of instances.

Third, instance searches are performed exclusively backward, reasoning from the goal back through instances. This means that if for example G : group α is in the context, a search for has_mul α may end up exploring upward through the entire structure hierarchy, looking for monoid α, then ring α, then field α, then normed_field α, prompting a search for has_norm α and so on, before eventually failing and attempting group α instead. This search will get larger as our structure hierarchy grows.

While the first two problems seem dire, a combination of good caching and a very efficient C++ implementation have helped to keep costs manageable even at mathlib's scale. Moreover, efficient graph traversal algorithms are known, and Lean 4 is expected to make improvements in this area.

The third problem could potentially be fixed by using a combination of forward and backward reasoning. If we reason forward from group α to monoid α, semigroup α, has_mul α, then work backward from the goal has_mul α, we find the solution quickly without exploring the entire structure hierarchy. Working forward from instances such as ring ℤ entails generating additional instances for monoid ℤ, add_group ℤ, and so on, which is currently not required but is often done to speed up instance searches.

5 Linear algebra

The development of linear algebra in mathlib showcases many of the design principles explained so far. To make earlier discussions more concrete, we describe in detail part of the linear algebra development. This description does not encompass all of the linear algebra in mathlib: the library also develops the theories of dual vector spaces, bilinear and sesquilinear forms, direct sums, and tensor products. This theory contains structures and theorems that are mathematically nontrivial and are used to prove noteworthy results (§7.2). Some of the structures are close to the top of the type class hierarchy, and they illustrate the use of bundled and semi-bundled type classes and quotient types.

Our formalization is certainly not unique: we have been heavily inspired by existing linear algebra developments, particularly in Isabelle/HOL [3, 21, 42] and Coq [26].

5.1 Modules

The fundamental structures of linear algebra are *modules*. Given a ring R, an R-module consists of an abelian group M and a distributive, associative scalar multiplication operator $\bullet : R \times M \longrightarrow M$. In mathlib, the type class `module R M` defines an `R`-module structure on the group `M`.

```
class mul_action (α : Type u) (β : Type v)
  [monoid α] extends has_scalar α β :=
(one_smul : ∀ b : β, (1 : α) • b = b)
(mul_smul : ∀ (x y : α) (b : β),
            (x * y) • b = x • y • b)

class distrib_mul_action (α : Type u)
  (β : Type v) [monoid α] [add_monoid β]
  extends mul_action α β :=
(smul_add : ∀ (r : α) (x y : β),
            r • (x + y) = r • x + r • y)
(smul_zero : ∀ (r : α), r • (0 : β) = 0)

class semimodule (R : Type u) (M : Type v)
  [semiring R] [add_comm_monoid M]
  extends distrib_mul_action R M :=
(add_smul : ∀ (r s : R) (x : M),
            (r + s) • x = r • x + s • x)
(zero_smul : ∀x : M, (0 : R) • x = 0)

class module (R : Type u) (M : Type v)
  [ring R] [add_comm_group M]
  extends semimodule R M
```

We provide an alternative constructor for `module R M` that does not require the fields `smul_zero` and `zero_smul`, since they follow from other properties.

When R is replaced with a field K, this structure is called a K-vector space. All results about modules also apply to vector spaces. In mathlib, we favor working with discrete fields, which fix $1/0 = 0$.

```
class vector_space (K : Type u) (V : Type v)
  [discrete_field K] [add_comm_group V]
  extends module K V
```

For the rest of this subsection we consider R-modules. We omit the following parameters to all declarations, and sometimes elide proof terms with . . . :

```
(R : Type u) (M : Type v) [ring R]
[add_comm_group M] [module R M]
```

When $(M, \bullet)$ is an R-module, a subgroup of M closed under $\bullet$ is called a *submodule*. These are also defined as bundled structures in mathlib. Ordered by set inclusion, the submodules of a module form a complete lattice.

```
structure submodule :=
(carrier : set M)
(zero : (0:M) ∈ carrier)
(add  : ∀ {x y}, x ∈ carrier →
        y ∈ carrier → x + y ∈ carrier)
(smul : ∀ (c:R) {x},
        x ∈ carrier → c • x ∈ carrier)
```

The lattice structure on submodules enables us to use the notation $\bot$ for the trivial submodule 0 and $\top$ for the universal submodule M. We also use the lattice structure to define the *span* of a set, the space of all linear combinations of its elements:

```
def span (s : set M) : submodule R M :=
Inf {p | s ⊆ p}
```

The *quotient* M/N of an R-module M by a submodule $N \subseteq M$ equates $m_1, m_2 \in M$ if $m_1 - m_2 \in N$. The quotient is itself an R-module, and is defined in mathlib as a quotient type.

```
def quotient_rel (N : submodule R M) :
  setoid M := ⟨λ x y, x - y ∈ N, ...⟩

def submodule.quotient (N : submodule R M) :
  Type v := quotient (quotient_rel N)

instance (N : submodule R M) :
  module R (quotient N) := ...
```

A function between two R-modules that preserves addition and scalar multiplication is called a *linear map*. If it is invertible, it is called a *linear equivalence*. We work with linear maps as bundled structures in mathlib, and coercions typically allow us to treat these structures as functions. We use the notation `M` $\to_l$`[R] N` to stand for `linear_map R M N` and `M` $\simeq_l$`[R] N` for `linear_equiv R M N`.

```
structure linear_map (N : Type w)
  [add_comm_group N] [module R N] :=
(to_fun : M → N)
(add  : ∀ x y,
        to_fun (x + y) = to_fun x + to_fun y)
(smul : ∀ (c : R) x,
        to_fun (c • x) = c • to_fun x)
```

The types `M` $\to_l$`[R] N` and `M` $\simeq_l$`[R] N` are themselves both R-modules.

If $f : M \to N$ is linear, the image of a submodule $M' \subseteq M$ under f is a submodule of N, and the preimage of a submodule $N' \subseteq M$ under f is a submodule of M. These are defined respectively as map and comap. In particular, the *kernel* of f—defined to be the set $\{x \in M \mid f(x) = 0\}$—is a submodule, as is the range.

```
def ker (f : M →ₗ[R] N) : submodule R M :=
comap f ⊥

def range (f : M →ₗ[R] N) : submodule R N :=
map f ⊤
```

These definitions and their surrounding proofs are enough to prove the first and second *isomorphism laws* for modules:

```
def quot_ker_equiv_range :
  f.ker.quotient ≃ₗ[R] f.range := ...

def sup_quotient_equiv_quotient_inf
  (p p' : submodule R M) :
  (comap p.subtype (p ⊓ p')).quotient ≃ₗ[R]
    (comap (p ⊔ p').subtype p').quotient := ...
```

5.2 Bases

A set of elements of an R-module $\{v_i\}_{i \in I}$ is said to be *linearly dependent* if there is a finite subset $I' \subseteq I$ and a family of scalars $\{c_i\}_{i \in I'}$, not all zero, such that $\sum_{i \in I'} c_i \cdot v_i = 0$. We are more often interested in the negation of this notion. In mathlib, we define the linear independence of a family of vectors indexed by a base type:

```
def linear_independent
  {ι : Type u} (R : Type v) {M : Type w}
  [ring R] [add_comm_group M] [module R M]
  (v : ι → M) : Prop :=
(finsupp.total ι M R v).ker = ⊥
```

The function finsupp.total ι M R v : (ι →₀ R) →ₗ[R] M sends a finitely supported function c : ι → R to the sum over ι of c i • v i.

Through the rest of this subsection we fix parameters as given to linear_independent. A linearly independent family of vectors is a *basis* for a module if it spans the entire module.

```
def is_basis (v : ι → M) : Prop :=
linear_independent R v ∧ span R (range v) = ⊤
```

Given such a v and a proof hv : is_basis R v, we define the linear map that decomposes a vector into a weighted sum of vectors in v:

```
def is_basis.repr : M →ₗ (ι →₀ R) :=
hv.1.repr.comp
  (linear_map.id.cod_restrict _ hv.mem_span)
```

A classic result in linear algebra shows that every vector space has a basis. (This result uses Zorn's lemma and does not hold constructively.)

```
lemma exists_is_basis [discrete_field K]
  [add_comm_group V] [vector_space K V] :
  ∃b : set V, is_basis K (λ i : b, i.val) := ...
```

5.3 Dimension

Any two bases for a vector space V have equal cardinality. This cardinality defines the *dimension* of V. Vector spaces are not necessarily finite dimensional, so we define the dimension function to take values in the type cardinal. The theory of cardinal numbers in mathlib is located in the set_theory subfolder. To avoid inconsistency, the type cardinal must be parametrized by a universe level.

```
def vector_space.dim (K : Type u) (V : Type v)
  [discrete_field K] [add_comm_group V]
  [vector_space K V] : cardinal.{v} :=
cardinal.min (nonempty_subtype.2
               (@exists_is_basis K V _ _ _))
               (λ b, cardinal.mk b.1)
```

Because the relevant cardinals may live in different universes, it takes some care to state the dimension theorem.

```
theorem mk_eq_mk_of_basis
  {v : ι → V} {v' : ι' → V}
  (hv : is_basis K v) (hv' : is_basis K v') :
  cardinal.lift.{w w'} (cardinal.mk ι) =
    cardinal.lift.{w' w} (cardinal.mk ι') := ...
```

Many dimension computations are proved in mathlib, including results about the dimensions of quotients and the rank-nullity theorem.

```
theorem dim_quotient (V' : submodule K V) :
  dim K p.quotient + dim K V' = dim K V := ...

theorem dim_range_add_dim_ker (f : V →ₗ[K] E) :
  dim K f.range + dim K f.ker = dim K V := ...
```

A number of these computations specialize to finite-dimensional vector spaces. For instance, the functions $A \to k$ form a k-vector space with dimension $|A|$ when A is finite.

```
lemma dim_fun [fintype η] :
  dim K (η → K) = fintype.card η := ...

lemma dim_fin_fun (n : ℕ) :
  dim K (fin n → K) = n := ...
```

The vector space fin n → K is a common way to represent n-tuples of elements of K.

5.4 Matrices

It is convenient to consider matrices as functions $m \to n \to R$, where m and n are finite sets of indices. We define this type in the data directory of mathlib, along with the expected operations and algebraic instances. For a ring R, R-valued $m \times n$ matrices form an R-module.

```
def matrix (m n : Type u) (R : Type v)
  [fintype m] [fintype n] : Type (max u v) :=
m → n → R
```

```
instance [ring R] : module R (matrix m n R) :=
...
```

There is a direct correspondence between R-valued matrices and linear maps between R-modules. To begin with, every such $m \times n$ matrix gives rise to a linear map from R^n to R^m. In fact, this evaluation function is itself a linear map.

```
def eval : (matrix m n R) →ₗ[R]
  ((n → R) →ₗ[R] (m → R)) :=
linear_map.mk₂ R mul_vec ...
```

Similarly, a linear map exists in the opposite direction.

```
def to_matrix : ((n → R) →ₗ[R] (m → R))
  →ₗ[R] matrix m n R :=
linear_map.mk
  (λ f i j, f (λ n, ite (j = n) 1 0) i) ...
```

These linear maps are inverses of each other, and thus form an equivalence between the space of linear maps $R^n \to R^m$ and the space of R-valued $m \times n$ matrices. It follows quickly that, given finite bases for two R-modules M_1 and M_2 indexed by ι and κ, the space of linear maps $M_1 \to M_2$ is linearly equivalent to the space of R-valued $\iota \times \kappa$ matrices.

```
def lin_equiv_matrix {ι : Type i} {κ : Type k}
  [fintype ι] [decidable_eq ι]
  [fintype κ] [decidable_eq κ]
  {v₁ : ι → M₁} (hv₁ : is_basis R v₁)
  {v₂ : κ → M₂} (hv₂ : is_basis R v₂) :
  (M₁ →ₗ[R] M₂) ≃ₗ[R] matrix κ ι R := ...
```

6 Metaprogramming

Due to Lean's powerful metaprogramming framework [22], many features that might otherwise require changes to the prover or extension through plugins are implemented in mathlib itself. These include general purpose decision procedures, utilities, debugging tools, and special purpose automation. All of the following tactics are implemented in Lean as metaprograms, with the exceptions of the core simplification routine, E-matching [19], and congruence closure [51], which are exposed in the metalanguage as atomic procedures.

6.1 Simplification

The tactic `simp` is the primary tool in the Lean automation arsenal. Similar to the simplifier in Isabelle, it performs non-definitional directional rewriting with equality and biconditional lemmas. Lean's *attribute* mechanism allows the user to annotate definitions and theorems with extra information, which can be accessed by metaprograms; the default set of rewrite rules for `simp` is extended by adding the `simp` attribute to declarations. A variant, `dsimp`, performs only definitional reductions.

The simplifier matches rewrite lemmas up to syntactic, not definitional, equality. For this reason, lemmas in mathlib are typically stated in `simp`-normal form. When there are multiple equivalent ways to express a term, one is preferred, and others are simplified to the preferred form; subsequent lemmas need only be stated for this single case. An example of this design pattern can be seen in the development of the p-adic numbers `ℚ_[p]`, where the generic norm notation ‖x‖ from the `normed_space` type class is preferred to the definitionally equal `padic_norm p x`. We use `simp` lemmas to transform the latter to the former, and develop the library for the p-adic norm based on the generic notation.

```
@[simp] lemma is_norm (q : ℚ_[p]) :
  (padic_norm_e q) = ‖q‖ := rfl

@[simp] lemma mul (q r : ℚ_[p]) :
  ‖q * r‖ = ‖q‖ * ‖r‖ := ...
```

6.2 Tactics and automation

Besides `simp`, we briefly describe notable examples of automation in mathlib, which we roughly classify according to whether they are "big" (general-purpose, powerful, and meant to discharge certain classes of goals on their own) or "small" (specialized and providing finer control over the proof state). Alongside tactics, we include user commands, which manipulate the prover environment outside of any particular proof state, and attributes, which can trigger metaprograms when applied to declarations.

Big automation For general-purpose proof search, the most powerful tools in mathlib are `finish` and `tidy`. The former combines a tableau prover (complete for propositional logic) with simplification, E-matching, and congruence closure. The latter implements heuristics and repeatedly attempts to apply a preselected list of tactics (recursing into subgoals) until none succeed. In the category theory library, `tidy` is used extensively to check the functoriality and naturality. The constructors for many structures call `tidy` by default, so that proofs of these "obvious" properties need not even be mentioned.

The tactics `ring` and `abel` normalize expressions in commutative (semi)rings and abelian groups. They follow the approach described by Grégoire and Mahboubi [30], but produce proof terms tracing each normalization step instead of verifying by reflection.

Two tactics are used for solving linear inequalities. For linear ordered commutative semirings, `linarith` implements an algorithm based on Fourier-Motzkin variable elimination [58]; it is complete for dense linear orders. On ℕ and ℤ, `omega` partially implements the omega decision procedure for Presburger arithmetic [6, 50].

Small automation While Lean's metaprogramming engine is powerful enough for large tactics, it shines in the context of special-purpose tools, which are often simple to create and deploy. We highlight here a few of the dozens of such tactics and commands implemented in mathlib.

The `norm_cast` tactic manipulates type coercions [44]. Because Lean has no subtyping, a coercion (written with a prefix ↑) is required, for instance, when using a natural number in place of an integer. The presence of these coercions in terms can hinder simplification and unification: it can be tedious to reduce the integer inequality `↑m + ↑n > 5` to the natural number inequality `m + n > 5`. Using attributes to track lemmas that pass coercions through operations and relations, `norm_cast` performs such simplification automatically.

Arithmetic expressions involving only literals are evaluated efficiently by `norm_num`. Over the natural numbers, goals such as `1 + 2 < 4` can be proved by kernel computation. However, this is inefficient for large expressions, and impossible on noncomputable types such as ℝ and on arbitrary linear ordered semirings. As an alternative, `norm_num` uses the syntactic binary representation of numerals and lemmas about the relevant algebraic structures to simplify the arithmetic expressions as far as possible.

For a type class `C`, the tactic `pi_instance` generalizes instances of the form `C α` to `C (β → α)`. With this, we obtain much of the algebraic structure of function spaces for free.

The `reassoc` attribute improves the ability of `simp` to reason about compositions of arrows in a category modulo associativity. By default, `simp` normalizes such expressions by associating composition to the right. As a consequence, one often encounters series of arrow compositions that are bracketed as `a ≫ (b ≫ c)`. A lemma `foo` that rewrites `a ≫ b` to `d` would conflict with this `simp` normal form. Applying the attribute `reassoc` to `foo` produces a companion lemma of the shape `∀ X, a ≫ (b ≫ X) = d ≫ X` that can be used by the simplifier.

Maintenance and fine-tuning Another group of metaprograms is used for library development and maintenance.

Some search-based tactics can report traces of a successful search. Variants of `tidy` and `simp` list the tactic script and used lemmas, respectively, in a format that is faster and more robust than the original tactic call. In editors that support Lean's *hole commands*, calls to these tactics can be literally self-replacing.

The command `#lint` is a linting tool that performs various tests on declarations in a file or environment. Among other things, it checks for unused arguments, malformed names, and whether a file meets mathlib's documentation requirements. It is easily extended with custom tests.

Lean's namespacing and sectioning mechanisms allow the use of abbreviated identifiers, special notation, and instances locally in files, as well as to automatically insert parameters to declarations within a section. The `#where` command prints information about the currently open namespaces and parameters; the `localized` command associates local notation and instances with a namespace, making it easy to set up a particular local environment.

6.3 Lessons

The various tools listed above are provided as a part of mathlib, without need for extensions or plugins to the Lean system. (One exception, namely `simp`, is built into Lean.)

Collaboration between mathematically-oriented formalizers and more experienced programmers has driven much of the tactic development in mathlib. Tools such as `norm_num`, `ring`, and `norm_cast` arose after user requests to automate mathematically trivial proofs. Others, such as `pi_instance`, make clever use of the metaprogramming API to eliminate boilerplate code. It is hard to separate the metaprogram components of mathlib from the mathematical formalizations, as many tactics were inspired by particular patterns of use and rely on nontrivial theories or the proper use of attributes on library lemmas.

While our approach is not unique, our emphasis is a shift from the traditional point of view: we consider tactic and tool development as part of library design rather than system development. This division of labor has been possible thanks to the flexibility of metaprogramming in Lean.

7 Community

The mathlib library is not developed to support any single project. The interests of its contributors range from research mathematics, to STEM education, to automated proof search, to program verification. These contributors come primarily from academia; some are renowned researchers and some are bachelors students. Design decisions and future directions are discussed openly and publicly, with little central control over the library's content.

That a cohesive library has been developed by a community with such diverse motivations and backgrounds is surprising and unusual. Unlike most proof assistant libraries, which are developed or overseen by dedicated research groups, mathlib is organized as an open-source community project. While we have witnessed many of the familiar difficulties with such projects, this organizational scheme is arguably one of the reasons the library has been successful.

The communal nature of the project is, perhaps, one reason that Lean and mathlib have attracted many mathematicians. Mathematical topics are delicately intertwined. Rather than individually building projects on top of a generic core library, the involved mathematicians have integrated their projects with each other and with mathlib itself. To many, the main research question in mathlib is how to design a library of formal mathematics that is both broad and deep. This question cannot be answered without the collaboration of many people; the range of contributors is a vital aspect of the project.

A side effect of the adoption of mathlib by mathematicians is that Lean is being used in mathematics education. While it is not uncommon to see proof assistants in certain computer science courses, we know of few cases where they have been introduced to undergraduate mathematicians. Some of these undergraduates interact on the Lean Zulip chat room, contribute to mathlib, and participate in the review process.

7.1 GitHub and Zulip

Communication about mathlib occurs mainly over two channels. The project is hosted on GitHub,[2] where contributions in the form of pull requests can be reviewed and edited. More casual conversations occur in a Zulip chat room.[3]

Eleven community members have been designated as maintainers of the library. These people are responsible for approving pull requests in their areas of expertise. Pull request reviews are welcomed from all members of the community; in addition to the formal process on GitHub, PRs are often discussed on Zulip. Community members are given write access to non-master branches of the repository, and many PRs are developed as group efforts on these branches.

The Zulip chat room is home to a broad range of discussions. In particular, a channel for new members welcomes elementary questions about Lean and formalization. Another channel focuses on formalizing advanced mathematics. A number of people have cited the accessibility of this chat room as a reason for deciding to use Lean.

7.2 Projects based on mathlib

As well as being a cohesive collection of mathematics on its own, mathlib should also serve as a library on which to build more specialized projects. Users have undertaken many such projects, and the variety of topics reflects the diverse interests of the mathlib community. These projects range from published research papers to collaborative weekend efforts coordinated on Zulip. We list here a non-exhaustive selection, to give an idea of what mathlib can support.

The cap set problem In 2016, Ellenberg and Gijswijt discovered a solution to the cap set problem, a longstanding open question in combinatorics. Their celebrated proof [23], published in the *Annals of Mathematics* in 2017, was noted for its use of elementary methods from linear algebra. Dahmen, Hölzl, and Lewis [18] formalized this result in 2019. The formalization is based on mathlib and resulted in significant contributions to the linear algebra theory (§5), as well as those related to finite combinatorics. It is a rare example of contemporary research mathematics being formalized in a proof assistant, made possible by collaboration between a mathematician and experts in formalization.

The continuum hypothesis Han and van Doorn [33] verified the unprovability in ZFC of the continuum hypothesis. They have since shown the unprovability of ¬CH, completing the notoriously intricate full independence proof. The formalization builds on the mathlib embedding of ZFC and develops the syntactic theory of first order logic. The independence of CH was one of the few remaining results on Wiedijk's list of formalization targets[4] that had yet to be formalized in any proof assistant.

Perfectoid spaces Scholze was awarded a Fields Medal in 2018, in part for introducing the definition of a perfectoid space. To test the ability of a proof assistant to understand extremely complicated mathematical structures, Buzzard, Commelin, and Massot defined perfectoid spaces in Lean [15]. Defining this structure relies on a large amount of algebraic and topological theory, and the months-long effort to formalize it resulted in thousands of lines of Lean code.

The sensitivity conjecture Huang's sensational proof of the boolean sensitivity conjecture [36] in 2019 was widely discussed, including on the Lean Zulip chat. Following Knuth's simplified writeup,[5] a number of community members formalized the proof in a matter of days. The formalization reuses linear algebra machinery from the cap set project and amounts to fewer than 450 lines of heavily commented code.

Cubing a cube After a challenge was issued at the 2019 Big Proofs meeting in Edinburgh, van Doorn formalized J. E. Littlewood's "elegant" proof that any dissection of a cube into smaller cubes must contain at least two cubes of equal width. This had been considered an exemplar of a proof with a simple intuitive argument that is hard to make formal. It was another remaining item on Wiedijk's list of targets.

Decision procedures for modal logics Wu and Goré [59] verified decision procedures for the modal logics K, KT, and S4. The decision procedures are formalized as programs in Lean with total correctness proved. The formalization makes extensive use of sublist permutation which is fully supported by mathlib. The need for proving termination which involves reasoning about modal degrees has in turn resulted in further development of list theory of mathlib.

8 Comparison with other libraries

In this section, we compare and contrast mathlib with other substantial formal libraries for mathematics, including libraries for Mizar [9, 28], HOL Light [34], Isabelle/HOL [47], Coq/SSReflect [10, 45], and Metamath [46]. Our goal here is not to provide detailed comparisons of the various design choices, but, rather to sketch the design space in broad strokes, situate mathlib within it, and explain some of the decisions we have made. Our choice of comparisons is not meant to be exhaustive: there are also substantial mathematical libraries in HOL4 [52], ACL2 [41], PVS [48], and

[2]https://github.com/leanprover-community/mathlib/
[3]https://leanprover.zulipchat.com/
[4]http://www.cs.ru.nl/~freek/100/
[5]https://www.cs.stanford.edu/~knuth/papers/huang.pdf

NuPRL [17], as well as notable libraries built on standard Coq, such as the Coquelicot analysis library [14].

8.1 The design space

We will focus on three specific aspects of the design space, namely, the choice of axiomatic foundation, the mechanisms used to represent structures and the relationships between them, and the use of automation.

Most contemporary interactive proof systems are based on either set theory, simple type theory, or dependent type theory. The decision to use an untyped framework like set theory or a typed alternative is fundamental. Types play two important roles. First, input in a typed system is often less verbose because users can elide details that can be inferred from the types of the objects involved. For example, users can write `x + y` and allow the system to infer the meaning of the plus sign from the types of its arguments. Second, a type system can more easily catch and report errors, such as sending the wrong number or kinds of arguments to a function, or sending them in the wrong order. Another important choice is whether or not the axiomatic framework is constructive, and, in particular, whether it specifies a computational behavior for objects defined in the framework.

8.2 Libraries based on set theory

As an axiomatic foundation, set theory has two important advantages. First, it is accepted by many mathematicians at being the official foundation for mathematics, or, at least, an uncontroversial one. And, second, it is fairly easy to implement and check proofs in this framework.

The Metamath system [46] is a generic system for representing formal axiomatic frameworks, but its largest library is built on set theory. That library currently has about 23,000 theorems and 600k lines of code, covering algebra, analysis, topology, number theory, and other areas. Because the foundation is so simple, the source files need to provide explicit proofs that formulas are well formed, and there are front ends that insert that automatically. But the system uses very little automation beyond that. There are a number of reference checkers on offer that can check the entire library in a few seconds. Sets are used to relativize quantifiers to specific domains. For example, the following states that every continuous function on a closed interval is bounded:

```
((A ∈ ℝ ∧ B ∈ ℝ ∧ F ∈ ((A[,]B)cn→ℂ)) →
  ∃x ∈ ℝ ∀y ∈ (A[,]B)(abs‘(F‘y)) ≤ x)
```

The need to write proofs that are fully detailed and explicit, however, places a high burden on the user.

Mizar [9, 28], which dates back to the early 1970s, is based on set theory with Grothendieck-Tarski universes. Its vast library [8] currently contains over 3.1 million lines of code and spans many fields of mathematics. Proofs in the system are designed to mirror mathematical vernacular. They are written in a declarative way, and a fixed checker determines whether inferences are valid [28]. To support a structure hierarchy, Mizar uses a *soft typing* system whereby users register associations that are used by the checker. The library has developed quite an elaborate hierarchy in this way [29] (see also [28, 38]). To our knowledge, there is no formal specification of the inferences that are accepted by the system, making it hard to implement an independent reference checker. Nonetheless, Kaliszyk et al. have had success reimplementing the system in the Isabelle framework and checking some of the Mizar files [38, 39], and Urban and Sutcliffe have shown that Mizar's inferences can be cross validated by automated theorem provers [55].

8.3 Libraries based on simple type theory

A number of contemporary proof systems implement versions of *simple type theory*, among them HOL4 [52], HOL Light [34], and Isabelle/HOL [47]. In simple type theory, types and objects are separate things; one defines types and type constructions, and then one defines objects of those types. Types cannot depend on objects; for example, one cannot define a type $\mathbb{R}^n$ that depends on a parameter n.

Simple type theory excels at dealing with concrete structures like the integers, the reals, and the complex numbers, but when it comes to algebraic reasoning, the fact that types cannot depend on parameters is a severe restriction. Structures in mathematics are often parameterized by other objects: for $n \geq 1$, the type of $M_n(R)$ of $n \times n$ over a ring form a ring, and for every prime number p, the integers modulo p $\mathbb{Z}/p\mathbb{Z}$ form a field; L^p spaces, rings of p-adic integers, and spaces $C^k(X)$ consisting of k-times continuously differentiable real-valued functions on a subset X of the reals all depend on parameters. In simple type theory, one has to model these using predicates on fixed ambient types, erasing many of the benefits of using type theory in the first place. Another limitation is that one cannot form spaces or structures whose elements are structures, such as the space of nonempty compact metric spaces with the Gromov-Hausdorff distance or the category of groups in some universe.

HOL Light [34] is John Harrison's implementation of simple type theory, inspired by Mike Gordon's original HOL system. The library, close to 800k lines of code, is especially strong in multivariate real analysis and complex analysis. It served as the basis for the Flyspeck project [32]. Harrison uses a trick [35] to formalize theorems about $\mathbb{R}^n$ for arbitrary n, using type variables to stand proxy for their cardinality. But this trick has limited utility: in simple type theory, one cannot even state that for every n there is a type of cardinality n, and so there is no way of instantiating such a generic theorem to particular parameters.

Many aspects of our structure hierarchy, including the algebraic hierarchy, the axiomatization of topological spaces and normed spaces, our development of measure theory, and our use of filters in analysis, were modeled after similar developments in Isabelle [47]. Isabelle's extensive library has about 900k lines of code, and its Archive of Formal Proofs [12] has an additional 2.3 million lines of code. To treat common data types as instances of structures, Isabelle adopts a conservative extension of simple type theory with *axiomatic type classes* [31, 57]. But type classes can only depend on a single type parameter, and, moreover, the use of type classes suffers from the limitation described above. For example, the extensive libraries and automation developed for instances of rings like the integers and the reals cannot be applied to the ring of integers modulo some number m. For such structures, Isabelle also provides the mechanism of *locales* [7, 40], but, once again, this means relinquishing some of the benefits of the use of type theory in the first place. We therefore consider it a substantial accomplishment that we are beginning to approximate the depth and range of Isabelle's structure hierarchy in a dependently typed setting.

Isabelle's supporting automation is especially good. The system provides internal automation, decision procedures, and a conditional term rewriter [47], but can also call external resolution provers and SMT solvers and reconstruct their proofs [11, 13]. Using axiomatic type classes, internal automation can work generically with types that instantiate the relevant structures, just as our `norm_num` works for types that support numerals and the relevant operations. Isabelle's automation is much more powerful than that of Lean, but we again consider it notable that we are making progress towards the use of such automation with dependent types.

8.4 Libraries based on dependent type theory

Turning to dependent type theory, the best point of comparison for mathlib is the Mathematical Components library [45], based on Coq and the SSReflect proof language [27]. Coq's version of dependent type theory is very similar to Lean's. The Mathematical Components library was the basis for the landmark formalization of the odd order theorem [27], and focuses on related parts of mathematics, including group theory, linear algebra, and representation theory. The library itself is about 110k lines of code, not including the odd order theorem. Other libraries have been built on it, including an analysis library [2] that is still under development.

In contrast to mathlib, the Mathematical Components library is constructive throughout (although the analysis library just mentioned uses classical logic). Another distinguishing feature of the Mathematical Components library is that it uses very little external automation, and instead relies heavily on the computational interpretation of the underlying axiomatic framework, so that inferences can be verified by computational unfolding of expressions.

Like mathlib, Mathematical Components relies on powerful elaboration mechanisms to support a structure hierarchy, though the specific mechanisms are different: instead of type classes, Mathematical Components uses *canonical structures* [24], a particular kind of unification hint [4]. The core algebraic hierarchy in the Mathematical Components library [45, Section 7.3] includes key structures such as $\mathbb{Z}$-modules (essentially, abelian groups), rings, commutative rings, fields, algebraically closed fields, modules, and algebras. Other parts of the library define vector spaces, and structures with orders and norms, that are designed to support reasoning about subfields of the complex numbers. There are additional structures to support group theory and representation theory, and the analysis library includes structures such as topological spaces and normed modules.

Mathematical Components is generally more conservative than we are when it comes to instantiating structures and substructures. For example, the theory of the natural numbers in Mathematical Components is largely self contained, whereas our natural numbers instantiate an ordered semiring, which in turn inherits from the structures of additive and multiplicative monoids, linear orders, and so on. A calculation involving the natural numbers in mathlib may involve generic facts from all these structures.

The Mathematical Components library is also carefully designed to avoid the need for the kinds of searches described in §4. An artifact of the use of canonical structures is that concrete structures have to be instantiated to all the classes they inherit from. For example, the integers are declared as a $\mathbb{Z}$-module, a ring, a commutative ring, an integral domain, and so on. In mathlib, declaring an instance of a structure automatically handles not just substructures, but also induced structure. When we declare the reals to be a metric space, for example, it thereby inherits the structure of a uniform space hence a topological space.

9 Conclusion

With respect to the design space described in the previous section, the characteristic features of mathlib are as follows. First, it is based on a dependent type theory. We have chosen a typed framework for the reasons indicated in §8.1, and given that we want to carry out the full range of mathematical constructions, judicious use of dependent types is unavoidable. Second, it is focused on contemporary mathematics, which is resolutely classical. Nonetheless, large portions of the library are explicitly computational, and functions, say, on lists and integers can be executed. Third, it incorporates useful automation, such as a good conditional simplifier and domain-specific tactics written in the system's metaprogramming language. Our automation is still in an incipient state, but we hope and expect to expand these capabilities. Finally, it includes a large and interconnected hierarchy of mathematical structures and instances. Each of the other libraries discussed in §8 shares some of these features, but we know of no other library that shares them all.

The mathlib library and its surrounding community continue to grow. Various individuals and groups are actively working to expand its mathematical content, automated reasoning tools, and surrounding infrastructure. There is no target or end goal in mind: as in traditional mathematics, the development of new theories will follow the needs and desires of the people involved.

A new version of Lean is currently in development [54]. It will not be backward compatible with the current version, and when the system is ready, we expect to update mathlib. Lean 4 promises even more efficient metaprogramming with access to input syntax trees and a customizable parser; it might be possible to automate much of the porting process.

The design of mathlib builds on the lessons of existing libraries, but the special character of mathlib can be attributed to the range of its contributors. The combined efforts of mathematicians, computer scientists, and programmers to design a single unified library of formalizations and tools has been successful, and we are still learning from one another, as the community continues to evolve.

Acknowledgments

We thank Jasmin Blanchette, Assia Mahboubi, and Josef Urban for their helpful comments and suggestions.

The following people have either authored commits in the mathlib repository or are attributed authors of mathlib files:

Lucas Allen, Ellen Arlt, Jeremy Avigad, Tim Baanen, Seul Baek, Reid Barton, Tim Baumann, Alexander Bentkamp, Aaron Bryce, Kevin Buzzard, Louis Carlin, Mario Carneiro, Bryan Gin-ge Chen, Johan Commelin, Sander Dahmen, Floris van Doorn, Gabriel Ebner, Ramon Fernández Mir, Fabian Glöckle, Sébastien Gouëzel, Tobias Grosser, Jesse Han, Keeley Hoek, Johannes Hölzl, Michael Howes, Simon Hudon, Christopher Hughes, Michael Jendrusch, Sangwoo Jo, Kevin Kappelmann, Parikshit Khanna, Yury Kudryashov, Joey van Langen, Kenny Lau, Sean Leather, Guy Leroy, Robert Y. Lewis, Amelia Livingston, Isabel Longbottom, Jean Lo, Paul-Nicolas Madelaine, Patrick Massot, Rohan Mitta, Stephen Morgan, Scott Morrison, Leonardo de Moura, Wojciech Nawrocki, Morenikeji Neri, Casper Putz, Mitchell Rowett, Jan-David Salchow, Blair Shi, Calle Sönne, Robert Spencer, Neil Strickland, Abhimanyu Pallavi Sudhir, Callum Sutton, Andreas Swerdlow, Nathaniel Thomas, Alistair Tucker, Sebastian Ullrich, Koundinya Vajjha, Jens Wagemaker, Minchao Wu, Haitao Zhang, Zhouhang Zhou, Sebastian Zimmer, Andrew Zipperer.

References

[1] 1994. The QED Manifesto. In *Automated Deduction - CADE-12, 12th International Conference on Automated Deduction, Nancy, France, June 26 - July 1, 1994, Proceedings.* 238–251. http://link.springer.com/chapter/10.1007/3-540-58156-1_17

[2] Reynald Affeldt, Cyril Cohen, and Damien Rouhling. 2018. Formalization Techniques for Asymptotic Reasoning in Classical Analysis. *J. Formalized Reasoning* 11, 1 (2018), 43–76. https://doi.org/10.6092/issn.1972-5787/8124

[3] Jesús Aransay and Jose Divasón. 2014. Formalization and Execution of Linear Algebra: From Theorems to Algorithms. In *Logic-Based Program Synthesis and Transformation*, Gopal Gupta and Ricardo Peña (Eds.). Springer International Publishing, Cham, 1–18.

[4] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. 2009. Hints in Unification. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings.* 84–98. https://doi.org/10.1007/978-3-642-03359-9_8

[5] Jeremy Avigad, Leonardo de Moura, and Soonho Kong. 2014. *Theorem Proving in Lean.* Carnegie Mellon University.

[6] Seulkee Baek. 2019. *Reflected Decision Procedures in Lean.* Master's thesis. Carnegie Mellon University. http://www.andrew.cmu.edu/user/avigad/Students/baek_ms_thesis.pdf

[7] Clemens Ballarin. 2003. Locales and Locale Expressions in Isabelle/Isar. In *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers.* 34–50. https://doi.org/10.1007/978-3-540-24849-1_3

[8] Grzegorz Bancerek, Czeslaw Bylinski, Adam Grabowski, Artur Kornilowicz, Roman Matuszewski, Adam Naumowicz, and Karol Pak. 2018. The Role of the Mizar Mathematical Library for Interactive Proof Development in Mizar. *J. Autom. Reasoning* 61, 1-4 (2018), 9–32. https://doi.org/10.1007/s10817-017-9440-6

[9] Grzegorz Bancerek, Czesław Byliński, Adam Grabowski, Artur Korniłowicz, Roman Matuszewski, Adam Naumowicz, Karol Pąk, and Josef Urban. 2015. Mizar: State-of-the-art and Beyond. In *Intelligent Computer Mathematics*, Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe, and Volker Sorge (Eds.). Springer International Publishing, Cham, 261–279.

[10] Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions.* Springer. https://doi.org/10.1007/978-3-662-07964-5

[11] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. 2013. Extending Sledgehammer with SMT Solvers. *J. Autom. Reasoning* 51, 1 (2013), 109–128. https://doi.org/10.1007/s10817-013-9278-5

[12] Jasmin Christian Blanchette, Max W. Haslbeck, Daniel Matichuk, and Tobias Nipkow. 2015. Mining the Archive of Formal Proofs. In *Intelligent Computer Mathematics - International Conference, CICM 2015, Washington, DC, USA, July 13-17, 2015, Proceedings.* 3–17. https://doi.org/10.1007/978-3-319-20615-8_1

[13] Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. 2016. Hammering towards QED. *J. Formalized Reasoning* 9, 1 (2016), 101–148. https://doi.org/10.6092/issn.1972-5787/4593

[14] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. 2015. Coquelicot: A User-Friendly Library of Real Analysis for Coq. *Mathematics in Computer Science* 9, 1 (01 Mar 2015), 41–62. https://doi.org/10.1007/s11786-014-0181-1

[15] Kevin Buzzard, Johan Commelin, and Patrick Massot. 2019. Formalizing perfectoid spaces. (2019). https://leanprover-community.github.io/lean-perfectoid-spaces/

[16] Mario Carneiro. 2019. Formalizing Computability Theory via Partial Recursive Functions. In *10th International Conference on Interactive Theorem Proving (ITP 2019) (Leibniz International Proceedings in Informatics (LIPIcs))*, John Harrison, John O'Leary, and Andrew Tolmach (Eds.), Vol. 141. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 12:1–12:17. https://doi.org/10.4230/LIPIcs.ITP.2019.12

[17] Robert L. Constable, Stuart F. Allen, Mark Bromley, Rance Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, Todd B. Knoblock, N. P. Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. 1986. *Implementing mathematics with the Nuprl proof development system.* Prentice Hall. http://dl.acm.org/citation.cfm?id=10510

[18] Sander R. Dahmen, Johannes Hölzl, and Robert Y. Lewis. 2019. Formalizing the Solution to the Cap Set Problem. In *10th International Conference on Interactive Theorem Proving (ITP 2019) (Leibniz International Proceedings in Informatics (LIPIcs))*, John Harrison, John O'Leary, and Andrew Tolmach (Eds.), Vol. 141. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 15:1–15:19. https://doi.org/10.4230/LIPIcs.ITP.2019.15

[19] Leonardo de Moura and Nikolaj Bjørner. 2007. Efficient E-Matching for SMT Solvers. In *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings.* 183–198. https://doi.org/10.1007/978-3-540-73595-3_13

[20] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (system description). https://leanprover.github.io/papers/system.pdf

[21] Jose Divasón and Jesús Aransay. 2013. Rank-Nullity Theorem in Linear Algebra. *Archive of Formal Proofs* (Jan. 2013). http://isa-afp.org/entries/Rank_Nullity_Theorem.html, Formal proof development.

[22] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. 2017. A metaprogramming framework for formal verification. *PACMPL* 1, ICFP (2017), 34:1–34:29. https://doi.org/10.1145/3110278

[23] Jordan S. Ellenberg and Dion Gijswijt. 2017. On large subsets of $\mathbb{F}_q^n$ with no three-term arithmetic progression. *Ann. of Math. (2)* 185, 1 (2017), 339–343. https://doi.org/10.4007/annals.2017.185.1.8

[24] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. 2009. Packaging Mathematical Structures. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings.* 327–342. https://doi.org/10.1007/978-3-642-03359-9_23

[25] Michèle Giry. 1982. A categorical approach to probability theory. In *Categorical aspects of topology and analysis (Ottawa, Ont., 1980).* Lecture Notes in Math., Vol. 915. Springer, Berlin-New York, 68–85.

[26] Georges Gonthier. 2011. Point-Free, Set-Free Concrete Linear Algebra. In *Interactive Theorem Proving*, Marko van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 103–118.

[27] Georges Gonthier and Assia Mahboubi. 2010. An introduction to small scale reflection in Coq. *J. Formalized Reasoning* 3, 2 (2010), 95–152. https://doi.org/10.6092/issn.1972-5787/1979

[28] Adam Grabowski, Artur Kornilowicz, and Adam Naumowicz. 2010. Mizar in a Nutshell. *Journal of Formalized Reasoning* 3, 2 (2010), 153–245. https://doi.org/10.6092/issn.1972-5787/1980

[29] Adam Grabowski, Artur Kornilowicz, and Christoph Schwarzweller. 2016. On algebraic hierarchies in mathematical repository of Mizar. In *Proceedings of the 2016 Federated Conference on Computer Science and Information Systems, FedCSIS 2016, Gdańsk, Poland, September 11-14, 2016.* 363–371. https://doi.org/10.15439/2016F520

[30] Benjamin Grégoire and Assia Mahboubi. 2005. Proving Equalities in a Commutative Ring Done Right in Coq. In *Theorem Proving in Higher Order Logics*, Joe Hurd and Tom Melham (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 98–113.

[31] Florian Haftmann and Makarius Wenzel. 2006. Constructive type classes in Isabelle. In *International Workshop on Types for Proofs and Programs.* Springer, 160–174.

[32] Thomas C. Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason M. Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. 2015. A formal proof of the Kepler conjecture. *CoRR* abs/1501.02155 (2015). arXiv:1501.02155 http://arxiv.org/abs/1501.02155

[33] Jesse Michael Han and Floris van Doorn. 2019. A Formalization of Forcing and the Unprovability of the Continuum Hypothesis. In *10th International Conference on Interactive Theorem Proving (ITP 2019) (Leibniz International Proceedings in Informatics (LIPIcs))*, John Harrison, John O'Leary, and Andrew Tolmach (Eds.), Vol. 141. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 19:1–19:19. https://doi.org/10.4230/LIPIcs.ITP.2019.19

[34] John Harrison. 2009. HOL Light: An Overview. In *Theorem Proving in Higher Order Logics*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 60–66.

[35] John Harrison. 2013. The HOL Light Theory of Euclidean Space. *J. Autom. Reasoning* 50, 2 (2013), 173–190. https://doi.org/10.1007/s10817-012-9250-9

[36] Hao Huang. 2019. Induced subgraphs of hypercubes and a proof of the Sensitivity Conjecture. *arXiv preprint arXiv:1907.00847* (2019).

[37] Fabian Immler and Bohua Zhan. 2019. Smooth manifolds and types to sets for linear algebra in Isabelle/HOL. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019.* 65–77. https://doi.org/10.1145/3293880.3294093

[38] C. Kaliszyk and K. Pąk. 2017. Progress in the independent certification of mizar mathematical library in isabelle. In *2017 Federated Conference on Computer Science and Information Systems (FedCSIS).* 227–236. https://doi.org/10.15439/2017F289

[39] Cezary Kaliszyk, Karol Pąk, and Josef Urban. 2016. Towards a Mizar Environment for Isabelle: Foundations and Language. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2016).* ACM, New York, NY, USA, 58–65. https://doi.org/10.1145/2854065.2854070

[40] Florian Kammüller, Markus Wenzel, and Lawrence C. Paulson. 1999. Locales - A Sectioning Concept for Isabelle. In *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings.* 149–166. https://doi.org/10.1007/3-540-48256-3_11

[41] M. Kaufmann, P. Manolios, and J.S. Moore. 2013. *Computer-Aided Reasoning: ACL2 Case Studies.* Springer US. https://books.google.com/books?id=EMneBwAAQBAJ

[42] Holden Lee. 2014. Vector Spaces. *Archive of Formal Proofs* (2014). http://isa-afp.org/entries/VectorSpace.html, Formal proof development.

[43] Robert Y. Lewis. 2019. A formal proof of Hensel's lemma over the p-adic integers. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019.* 15–26. https://doi.org/10.1145/3293880.3294089

[44] Paul-Nicolas Madelaine. 2019. *Arithmetic and casting in Lean.* Technical Report. Vrije Universiteit Amsterdam. https://lean-forward.github.io/internships/arithmetic_and_casting_in_lean.pdf

[45] Assia Mahboubi and Enrico Tassi. 2017. Mathematical Components.

[46] Norman Megill and David A. Wheeler. 2019. *Metamath: A Computer Language for Mathematical Proofs.* Lulu Press.

[47] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: a proof assistant for higher-order logic.* Vol. 2283. Springer Science & Business Media.

[48] Sam Owre, John M. Rushby, and Natarajan Shankar. 1992. PVS: A Prototype Verification System. In *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction,*

Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings. 748–752. https://doi.org/10.1007/3-540-55602-8_217

[49] Frank Pfenning and Christine Paulin-Mohring. 1989. Inductively Defined Types in the Calculus of Constructions. In *Mathematical Foundations of Programming Semantics, 5th International Conference, Tulane University, New Orleans, Louisiana, USA, March 29 - April 1, 1989, Proceedings*. 209–228. https://doi.org/10.1007/BFb0040259

[50] William Pugh. 1991. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing '91)*. ACM, New York, NY, USA, 4–13. https://doi.org/10.1145/125826.125848

[51] Daniel Selsam and Leonardo Moura. 2016. Congruence Closure in Intensional Type Theory. In *Proceedings of the 8th International Joint Conference on Automated Reasoning - Volume 9706*. Springer-Verlag, Berlin, Heidelberg, 99–115. https://doi.org/10.1007/978-3-319-40229-1_8

[52] Konrad Slind and Michael Norrish. 2008. A Brief Overview of HOL4. In *Theorem Proving in Higher Order Logics*, Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 28–32.

[53] Bas Spitters and Eelis van der Weegen. 2011. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science* 21, 4 (2011), 795–825. https://doi.org/10.1017/S0960129511000119

[54] Sebastian Ullrich and Leonardo de Moura. 2019. Counting Immutable Beans: Reference Counting Optimized for Purely Functional Programming. arXiv:cs.PL/1908.05647

[55] Josef Urban and Geoff Sutcliffe. 2008. ATP-based Cross-Verification of Mizar Proofs: Method, Systems, and First Experiments. *Mathematics in Computer Science* 2, 2 (2008), 231–251. https://doi.org/10.1007/s11786-008-0053-7

[56] Philip Wadler and Stephen Blott. 1989. How to Make ad-hoc Polymorphism Less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. 60–76. https://doi.org/10.1145/75277.75283

[57] Markus Wenzel. 1997. Type Classes and Overloading in Higher-Order Logic. In *Theorem Proving in Higher Order Logics, 10th International Conference, TPHOLs'97, Murray Hill, NJ, USA, August 19-22, 1997, Proceedings*. 307–322. https://doi.org/10.1007/BFb0028402

[58] H. P. Williams. 1986. Fourier's Method of Linear Programming and Its Dual. *The American Mathematical Monthly* 93, 9 (1986), 681–695. http://www.jstor.org/stable/2322281

[59] Minchao Wu and Rajeev Goré. 2019. Verified Decision Procedures for Modal Logics. In *10th International Conference on Interactive Theorem Proving (ITP 2019) (Leibniz International Proceedings in Informatics (LIPIcs))*, John Harrison, John O'Leary, and Andrew Tolmach (Eds.), Vol. 141. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 31:1–31:19. https://doi.org/10.4230/LIPIcs.ITP.2019.31