Maintaining a Library of Formal Mathematics

Floris van Doorn[1], Gabriel Ebner[2], and Robert Y. Lewis[2(✉)]

[1] University of Pittsburgh, Pittsburgh, PA 15260, USA
fpvdoorn@gmail.com

[2] Vrije Universiteit Amsterdam, 1081 HV Amsterdam, The Netherlands
gebner@gebner.org, r.y.lewis@vu.nl

Abstract. The Lean mathematical library mathlib is developed by a community of users with very different backgrounds and levels of experience. To lower the barrier of entry for contributors and to lessen the burden of reviewing contributions, we have developed a number of tools for the library which check proof developments for subtle mistakes in the code and generate documentation suited for our varied audience.

Keywords: Formal mathematics · Library development · Linting

1 Introduction

As a tool for managing mathematical knowledge, a proof assistant offers many assurances. Once a result has been formalized, readers can confidently believe that the relevant definitions are fully specified, the theorem is stated correctly, and there are no logical gaps in the proof. A body of mathematical knowledge, represented by formal definitions and proofs in a single theorem proving environment, can be trusted to be coherent.

Logical coherence, however, is only one of many properties that one could wish of a mathematical corpus. The ideal corpus can be modified, extended, and queried by users who do not have expert knowledge of the entire corpus or the underlying system. Proof assistant libraries do not always fare so well in this respect. Most of the large mathematical libraries in existence are maintained by expert users with a significant time cost. While external contributions are easily checked for logical consistency, it typically takes manual review to check that contributions cohere with the system in other ways—e.g., that lemmas are correctly marked for use with a simplification tactic. It can be difficult or impossible for outsiders to understand the library well enough to contribute themselves.

The first author is supported by the Sloan Foundation (grant G-2018-10067). The second and third authors receive support from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka) and from the Dutch Research Council (NWO) under the Vidi program (project No. 016.Vidi.189.037, Lean Forward).

C. Benzmüller and B. Miller (Eds.): CICM 2020, LNAI 12236, pp. 251–267, 2020.
https://doi.org/10.1007/978-3-030-53518-6_16

The mathlib library [15] is a corpus of formal mathematics, programming, and tactics in the Lean proof assistant [16] that is managed and cultivated by a community of users. The community encourages contributions from novice users, and the rapid growth of the library has threatened to overwhelm its appointed maintainers. The maintenance difficulty is compounded by the library's extensive use of type classes and context-dependent tactics. Misuse of these features is not always easy to spot, but can lead to headaches in later developments.

To ease the burdens on new users and maintainers alike, we have incorporated into mathlib tools for checking meta-logical properties of declarations and collecting, generating, and displaying documentation in an accessible way. The use of these tools has already had a large impact on the community. We aim here to explain the goals and design principles of these tools. While some details are specific to Lean and mathlib, we believe that these considerations apply broadly to libraries of formal mathematical knowledge.

2 Lean and mathlib

Lean offers a powerful metaprogramming framework that allows Lean programs to access the system's syntax and core components [8]. All of the linting tools described in Sect. 3 are implemented in Lean, without the need for external plugins or dependencies. They are distributed as part of the mathlib library.

Lean metaprograms are frequently used to implement *tactics*, which transform the proof state of a declaration in progress. They can also implement top-level *commands*, which interact with an environment outside the context of a proof. Examples include `#find`, which searches for declarations matching a pattern, and `mk_simp_attribute`, which defines a new collection of simplification lemmas. *Transient* commands like `#find`, which do not modify the environment, customarily start with `#`. Tactics and commands interact with the Lean environment and proof state through the `tactic` monad, which handles side effects and failure conditions in a purely functional way. Finally, Lean supports tagging declarations with *attributes* as a way to store metadata. Within the `tactic` monad, metaprograms can access the list of declarations tagged with a certain attribute.

The mathlib project is run by a community of users and encourages contributions from people with various backgrounds. The community includes many domain experts, people with expert knowledge of the mathematics being formalized but who are less familiar with the intricacies of the proof assistant. Linting and documentation are useful for every user of every programming language, but are especially helpful for such domain experts, since they often work on deep and intricate implementations without a broad view of the library.

An example of this is seen in mathlib's *structure hierarchy* [15, Sect. 4]. The library extensively uses type classes to allow definitions and proofs to be stated at the appropriate level of generality without duplication. Type classes are a powerful tool, but seemingly innocent anti-patterns in their use can lead to unstable and unusable developments. Even experienced users find it difficult to avoid these patterns, and they easily slip through manual code review.

The mathlib library and community are growing at a fast pace. As of May 15, 2020, the library contains over 170,000 lines of non-whitespace, non-comment code, representing a 25% increase over five months, and 42,000 declarations, excluding internal and automatically generated ones, a 23% increase. Contributions have been made by 85 people, a 16% increase over the same time period. 264 commits were made to the mathlib git repository in April 2020; while a small number were automatically generated, each commit typically corresponds to a single approved pull request. We display more statistics about the project's growth on the community website.[1] The library covers a wide range of subject matter, enough to serve as a base for numerous projects that have formalized complex and recent mathematical topics [5,7,11].

3 Semantic Linting

Static program analysis, the act of analyzing computer code without running the code, is widely used in many programming languages. An example of this is *linting*, where source code is analyzed to flag faulty or suspicious code. Linters warn the user about various issues, such as syntax errors, the use of undeclared variables, calls to deprecated functions, spacing and formatting conventions, and dangerous language features.

In typed languages like Lean, some of these errors are caught by the elaborator or type checker. The system will raise an error if a proof or program has a different type than the declared type or if a variable is used that has not been introduced. However, other problems can still be present in developments that have been accepted by Lean. It is also possible that there are problems with the metadata of a declaration, such as its attributes or documentation. These mistakes are often not obvious at the time of writing a declaration, but will manifest at a later time. For example, an instance might be declared that will never fire, or is likely to cause the type class inference procedure to loop.

We have implemented a package of semantic linters in mathlib to flag these kinds of mistakes. These linters are *semantic* in the sense that they take as input a fully elaborated declaration and its metadata. This is in contrast to a *syntactic* linter, which takes as input the source code as plain text. The use of semantic linters allows us to automatically check for many commonly made mistakes, using the abstract syntax tree (the elaborated term in Lean's type theory) for the type or value of a declaration. Syntactic linters would allow for testing of e.g. the formatting of the source code, but would not help with many of the tests we want to perform.

The linters can be used to check one particular file or all files in mathlib. Running the command `#lint` at any point in a file prints all the linter errors up to that line. The command `#lint_mathlib` tests all imported declarations in mathlib. Occasionally a declaration may be permitted to fail a lint test, for example, if it takes an unused argument to satisfy a more general interface. Such lemmas are tagged with the attribute `@[nolint]`, which takes a list of tests that

[1] https://leanprover-community.github.io/mathlib_stats.html.

```
/-- Reports definitions and constants that are missing doc strings -/
meta def doc_blame_report_defn : declaration → tactic (option string)
| (declaration.defn n _ _ _ _ _) := doc_string n >> return none <|>
    return "def missing doc string"
| (declaration.cnst n _ _ _) := doc_string n >> return none <|> return
    "constant missing doc string"
| _ := return none

/-- A linter for checking definition doc strings -/
@[linter, priority 1450] meta def linter.doc_blame : linter :=
{ test := λ d, mcond (bnot <$> has_attribute' `instance d.to_name)
    (doc_blame_report_defn d) (return none),
  no_errors_found := "No definitions are missing documentation.",
  errors_found := "DEFINITIONS ARE MISSING DOCUMENTATION STRINGS" }
```

Fig. 1. A linter that tests whether a declaration has a documentation string.

the declaration is allowed to fail. The continuous integration (CI) workflow of mathlib automatically runs the linters on all of mathlib for every pull request made to the library.

For some of the mistakes detected by our linters, it is reasonable to ask whether they should even be allowed by the system in the first place. The core Lean tool aims to be small, permissive, and customizable; enforcing our linter rules at the system level would cut against this philosophy. Projects other than mathlib may choose to follow different conventions, or may be small enough to ignore problems that hinder scalability. Stricter rules, of course, can create obstacles to finishing a project. By incorporating our checks into our library instead of the core Lean system, we make them available to all projects that depend on mathlib without forcing users to comply with them.

3.1 Linter Interface

A linter is a wrapper around a metaprogram with type `declaration → tactic (option string)`. Given an input declaration `d`, the test function returns `none` if `d` passes the test and `some error_msg` if it fails. These test functions work within the `tactic` monad in order to access the elaborator and environment, although some are purely functional and none modify the environment. The type `linter` bundles such a test function with formatting strings.

The package of linters is easily extended: a user simply defines and tags a declaration of type `linter`. In Fig. 1 one sees the full definition of the `doc_blame` linter, described in Sect. 3.2.

We have focused on implementing these linters with actionable warning messages. Since the errors they detect are often subtle and can seem mysterious to novice users, we try to report as clearly as possible what should change in a declaration in order to fix the warning.

3.2 Simple Linters

A first selection of mathlib linters checks for simple mistakes commonly made when declaring definition and theorems.

Duplicated Namespaces. Declaration names in Lean are hierarchical, and it is typical to build an interface for a declaration in its corresponding namespace. For example, functions about the type `list` have names such as `list.reverse` and `list.sort`. Lean's `namespace` sectioning command inserts these prefixes automatically. However, users often write a lemma with a full name and then copy it inside the namespace. This creates identifiers like `list.list.reverse`; it can be difficult to notice the duplication without careful review. The `dup_namespace` linter flags declarations whose names contain repeated components.

Definitions vs. Theorems. Lean has separate declaration kinds for definitions and theorems. The subtle differences relate to byte code generation and parallel elaboration. It is nearly always the case that a declaration should be declared as a theorem if and only if its type is a proposition. Because there are rare exceptions to this, the system does not enforce it. The `def_lemma` linter checks for this correspondence, so that the user must explicitly approve any exceptions.

Illegal Constants. The Lean core library defines `a > b` to be `b < a`, and similarly for `a ≥ b`. These statements are convertible, but some automation, including the simplifier, operates only with respect to syntactic equality. For this reason, it is convenient to pick a normal form for equivalent expressions. In mathlib, we prefer theorems to be stated in terms of `<` instead of `>`. The `ge_or_gt` linter checks that the disfavored constants do not appear in the types of declarations.

Unused Arguments. A very common beginner mistake is to declare unnecessary arguments to a definition or theorem. Lean's useful mechanisms for auto-inserting parameters in namespaces and sections can unfortunately contribute to this. The `unused_arguments` linter checks that each argument to a declaration appears in either a subsequent argument or the declaration type or body.

Missing Documentation. The mathlib documentation guidelines require every definition to have a doc string (Sect. 4). Since doc strings are accessible by metaprograms, we are able to enforce this property with a linter, called `doc_blame` (Fig. 1). Missing doc strings are the most common linter error caught in CI.

3.3 Type Class Linters

Lean and mathlib make extensive use of *type classes* [21] for polymorphic declarations. Of the 42,000 declarations in mathlib, 465 are type classes and 4600 are type class instances. In particular, type classes are used to manage the hierarchy of mathematical structures. Their use allows definitions and theorems to be stated at high levels of generality and then applied in specific cases without extra effort. Arguments to a declaration are marked as *instance implicit* by surrounding them with square brackets. When this declaration is applied, Lean runs a depth-first backward search through its database of instances to satisfy the argument. Type classes are a powerful tool, but users often find the underlying algorithms opaque, and their misuse can lead to performance issues [20]. A collection of linters aims to warn users about this misuse.

Guiding Type Class Resolution. Instances can be assigned a positive integer *priority*. During type class resolution the instances with a higher priority are tried first. Priorities are optional, and in mathlib most instances are given the default priority. Assigning priorities optimally is difficult. On the one hand, we want to try instances that are used more frequently first, since they are most likely to be applicable. On the other hand, we want to try instances that fail more quickly first, so that the depth-first search does not waste time on unnecessary searches.

While we cannot automatically determine the optimal priority of instances, there is one class of instances we want to apply last, namely the *forgetful instances*. A forgetful instance is an instance that applies to every goal, like the instance `comm_group α → group α`, which forgets that a commutative group is commutative. Read backward as in the type class inference search, this instance says that to inhabit `group α` it suffices to inhabit `comm_group α`.

Forgetful instances contrast with *structural instances* such as `comm_group α → comm_group β → comm_group (α × β)`. We want to apply structural instances before forgetful instances, because if the conclusion of a structural instance unifies with the goal, it is almost always the desired instance. This is not the case for forgetful instances, which are always applicable, even if the extra structure or properties are not available for the type in question. In this case, the type class inference algorithm will do an exhaustive search of the new instance problem, which can take a long time to fail. The `instance_priority` linter enforces that all forgetful instances have priority below the default.

Another potential problem with type class inference is the introduction of metavariables in the instance search. Consider the following definition of an R-module type class.

```
class module (R : Type u) (M : Type v) :=
(to_ring : ring R)
(to_add_comm_group : add_comm_group M)
(to_has_scalar : has_scalar R M)
/- some propositional fields omitted -/
```

If we make the projection `module.to_ring` an instance, we have an instance of the form `module R M → ring R`. This means that during type class inference, whenever we search for the instance `ring α`, we will apply `module.to_ring` and then search for the instance `module α ?m`, where `?m` is a metavariable. This type class problem is likely to loop, since most `module` instances will apply in the case that the second argument is a variable.

To avoid this, in mathlib the type of `module` actually takes as arguments the ring structure on `R` and the group structure on `M`. The declaration of module looks more like this:

```
class module (R : Type u) (M : Type v) [ring R] [add_comm_group M] :=
(to_has_scalar : has_scalar R M)
/- some propositional fields omitted -/
```

Using this definition, there is no instance from modules to rings. Instead, the ring structure of `R` is carried as an argument to the module structure on `M`. The `dangerous_instance` raises a warning whenever an instance causes a new type class problem that has a metavariable argument.

Misused Instances and Arguments. Misunderstanding the details of type class inference can cause users to write instances that can never be applied. As an example, consider the theorem which says that given a continuous ring homomorphism f between uniform spaces, the lift of f to the completion of its domain is also a ring homomorphism. The predicate `is_ring_hom f` is a type class in mathlib, and this theorem was originally written as a type class instance:

```
is_ring_hom f → continuous f → is_ring_hom (completion.map f)
```

However, `continuous f` is not a type class, and this argument does not appear in the codomain `is_ring_hom (completion.map f)`. There is no way for the type class resolution mechanism to infer this argument and thus this instance will never be applied. The `impossible_instance` linter checks declarations for this pattern, warning if a non-type class argument does not appear elsewhere in the type of the declaration.

A dual mistake to the one above is to mark an argument as instance implicit even though its type is not a type class. Since there will be no type class instances of this type, such an argument will never be inferable. The `incorrect_type_class_argument` linter checks for this. While the linter is very simple, it checks for a mistake that is difficult to catch in manual review, since it requires complete knowledge of the mathlib instance database.

Missing and Incorrect Instances. Most theorems in mathlib are type-polymorphic, but many hold only on *inhabited* types. (Readers used to HOL-based systems should note that Lean's type theory permits empty types, e.g. an inductive type with no constructors.) Inhabitedness is given by a type class argument, so in order to apply these theorems, the library must contain many instances of the `inhabited` type class. The `has_inhabited_instance` linter checks, for each concrete `Type`-valued declaration, that conditions are given to derive that the type is inhabited.

The `inhabited` type class is itself `Type`-valued. One can computably obtain a witness `t : T` from an instance of `inhabited T`; it is possible to have multiple distinct (nonconvertible) instances of `inhabited T`. Sometimes the former property is not necessary, and sometimes the latter property can create problems. For instance, instances deriving `inhabited T` from `has_zero T` and `has_one T` would

```
@[simp] lemma zero_add (x : ℕ) : 0 + x = x := /- ... -/

example (x : ℕ) : 0 + (0 + x) = x := by simp
```

Fig. 2. Example usage of the simplifier.

lead to non-commuting diamonds in the type class hierarchy. To avoid this, mathlib defines a weaker type class, `nonempty`, which is `Prop`-valued. Lean propositions are *proof-irrelevant*, meaning that any two terms of the same `Prop`-valued type are indistinguishable. Thus `nonempty` does not lead to non-commuting diamonds, and is safe to use in situations where `inhabited` instances would cause trouble.

The `inhabited_nonempty` linter checks for declarations with `inhabited` arguments that can be weakened to `nonempty`. Suppose that a `Prop`-valued declaration takes an argument `h : inhabited T`. Since Lean uses dependent types, `h` may appear elsewhere in the type of the declaration. If it doesn't, it can be weakened to `nonempty T`, since the elimination principles are equivalent for `Prop`-valued targets. Weakening this argument makes the declaration more widely applicable.

3.4 Linters for Simplification Lemmas

Lean contains a `simp` tactic for (conditional) term rewriting. Similar tactics, such as Isabelle's `simp` [17], are found in other proof assistants. Users can tag theorems using the `@[simp]` attribute. The theorems tagged with this attribute are collectively called the *simp set*. The `simp` tactic uses lemmas from the simp set, optionally with extra user-provided lemmas, to rewrite until it can no longer progress. We say that such a fully simplified expression is in *simp-normal form* with respect to the given simp set.

The simplifier is used widely: mathlib contains over 7000 simp lemmas, and the string `by simp` occurs almost 5000 times, counting only a small fraction of its invocations. However, care needs to be taken when formulating simp lemmas. For example, if both `a = b` and `b = a` are added as simp lemmas, then the simplifier will loop. Other mistakes are more subtle. We have integrated several linters that aid in declaring effective simp lemmas.

Redundant Simplification Lemmas. We call a simp lemma redundant if the simplifier will never use it for rewriting. This redundancy property depends on the whole simp set: a simp lemma is not redundant by itself, but due to other simp lemmas that break or subsume it. One way a simp lemma can be redundant is if its left-hand side is not in simp-normal form.

Simplification proceeds from the inside out, starting with the arguments of a function before simplifying the enclosing term. Given a term `f (0 + a)`, Lean will first simplify `a`, then it will simplify `0 + a` to `a` using the simp lemma `zero_add` (Fig. 2), and then finally simplify `f a`.

A lemma stating `f (0 + x) = g x` will never be used by the simplifier: the left-hand side `f (0 + x)` contains the subterm `0 + x` which is not in simp-normal form. Whenever the simplifier tries to use this lemma to rewrite a term, the arguments to `+` have already been simplified, so this subterm can never match.

It is often not immediately clear whether a term is in simp-normal form. The first version of the `simp_nf` linter only checked that the arguments of the left-hand side of a simp lemma are in simp-normal form. This first version identified more than one hundred lemmas across mathlib violating this condition. In some cases, the lemma satisfied this condition in the file where it was declared, but later files contained simp lemmas that simplified the left-hand side.

Simp lemmas can also be redundant if one simp lemma generalizes another simp lemma. The simplifier always picks the *last* simp lemma that matches the current term. (It is possible to override this order using the `@[priority]` attribute.) If a simp lemma is followed by a more general version, then the first lemma will never be used, such as `length_singleton` in the following example. It is easy to miss this issue at first glance since `[x]` and `x::xs` look very different, but `[x]` is actually parsed as `x::[]`.

```
@[simp] lemma length_singleton : length [x] = 1 := rfl
@[simp] lemma length_cons : length (x::xs) = length xs + 1 := rfl
```

Both of these issues are checked by the `simp_nf` linter. It runs the simplifier on the left-hand side of the simp lemma, and examines the proof term returned by the simplifier. If the proof of the simplification of the left-hand side uses the simp lemma itself, then the simp lemma is not redundant. In addition, we also assume that the simp lemma is not redundant if the left-hand side does not simplify at all, as is the case for conditional simp lemmas. Otherwise the linter outputs a warning including the list of the simp lemmas that were used.

Commutativity Lemmas. Beyond conditional term rewriting, Lean's simplifier also has limited support for ordered rewriting with commutativity lemmas such as `x + y = y + x`. Naively applying such lemmas clearly leads to non-termination, so the simplifier only uses these lemmas if the result is smaller as measured by a total order on Lean terms. Rewriting with commutativity lemmas results in nice normal forms for expressions without nested applications of the commutative operation. For example, it reliably solves the goal `f (m + n) = f (n + m)`. However, in the presence of nested applications, the results are unpredictable:

```
example (a b : ℤ) : (a + b) + -a = b := by simp /- works -/
example (a b : ℤ) : a + (b + -a) = b := by simp /- fails -/
```

The `simp_comm` linter checks that the simp set contains no commutativity lemmas.

Variables as Head Symbols. Due to the implementation of Lean's simplifier, there are some restrictions on simp lemmas. One restriction is that the head symbol of the left hand side of a simp lemma must not be a variable. For example, in the hypothetical (conditional) lemma

```
∀ f, is_homomorphism f → f (x + y) = f x + f y
```

the left-hand side has head symbol `f`, which is a bound variable, and therefore the simplifier will not rewrite with this lemma. The `simp_var_head` linter ensures that no such lemmas are accidentally added to the simp set.

4 Documentation

Programming language documentation serves very different purposes for different audiences, and proof assistant library documentation is no different. When creating documentation for Lean and mathlib, we must address users who

- are new to Lean and unfamiliar with its syntax and paradigms;
- would like an overview of the contents of the library;
- would like to understand the design choices made in an existing theory;
- would like a quick reference to the interface for an existing theory;
- need to update existing theories to adjust to refactorings or updates;
- would like to learn to design and implement tactics or metaprograms; and
- would like a quick reference to the metaprogramming interface.

Many of these goals are best served with user manuals or tutorials [2]. Such documents are invaluable, but there is a high cost to maintaining and updating them. They are most appropriate for material that does not often change, such as the core system syntax and logical foundations. From the perspective of library maintenance, we are particularly interested in *internal documentation*, that is, documentation which is directly written in the mathlib source files. Since the library evolves very quickly, it is essential to automatically generate as much of the reference material as possible. Furthermore, human-written text should be close to what it describes, to make it harder for the description and implementation to diverge.

We focus here on a few forms of this internal documentation. *Module documentation*, written at the top of a mathlib source file, is intended to describe the theory developed in that file, justify its design decisions, and explain how to use it in further developments. (A Lean source file is also called a *module*.) *Declaration doc strings* are written immediately before definitions and theorems. They describe the behavior or content of their subject declarations. In supported editors, these doc strings are automatically displayed when the cursor hovers over a reference to the declaration. *Decentralized documentation* is not localized to a particular line or file of the library, although it may originate in a certain place; it is expected to be collected and displayed post hoc. An example of this is tactic documentation: mathlib defines hundreds of interactive tactics in dozens of files, but users expect to browse them all on a single manual page.

Some features of proof assistants (and of Lean and mathlib in particular) encourage a different style of documentation from traditional programming languages. Since Lean propositions are proof-irrelevant, only the statement of a theorem, not its proof term, can affect future declarations. Thus theorems are self-documenting in a certain sense: the statement of a theorem gives a complete account of its content, in contrast to a definition of type $\mathbb{N} \to \mathbb{N}$, for example. We require doc strings on all mathlib definitions but allow them to be omitted from theorems. While it is often helpful to have the theorem restated or explained in natural language, the manual burden of writing and maintaining these strings for the large amount of simple lemmas in mathlib outweighs the gain of the natural language restatement. Nonetheless, doc strings are strongly encouraged on important theorems and results with nonstandard statements or names.

4.1 Generation Pipeline

In the style of many popular programming languages, we generate and publish HTML documentation covering the contents of mathlib. The generation is part of mathlib's continuous integration setup.

Perhaps unusually for this kind of tool, our generator does not examine the mathlib source files. Instead, it builds a Lean environment that imports the entire library and traverses it using a metaprogram. The metaprogramming interface allows access to the file name, line number, and doc string for any particular declaration, along with module doc strings. By processing a complete environment we can display terms using notation declared later in the library, and include automatically generated declarations that do not appear in the source. We can also associate global information with declarations: for example, we can display a list of instances for each type class.

The generation metaprogram produces a JSON file that contains all information needed to print the module, declaration, and decentralized documentation. A separate script processes this database into a searchable HTML website.[2]

4.2 Declaration Display

The majority of the documentation is oriented around modules. For each Lean source file in mathlib, we create a single HTML page displaying the module documentation and information for each declaration in that file. Declarations appear in the same order as in the source, with an alphabetical index in a side panel. For each declaration, we print various pieces of information (Fig. 3).

The declaration name is printed including its full namespace prefix. Lean declarations have four possible kinds: `theorem`, `definition`, `axiom`, and `constant`. We print the declaration kind and use it to color the border of the entry for a visual cue. The type of the declaration is printed with implicit arguments hidden by default. This gives an easy reference as to how the declaration can be applied. Each type can be expanded to display all arguments. When a declaration has a doc string, it is displayed beneath the type.

Lean represents the type former and constructors of an inductive type as separate constants. We display them together, mirroring the Lean syntax for an

[2] https://leanprover-community.github.io/mathlib_docs/.

```
@[class]                                                    view source
structure normed_space (α : Type u_5) (β : Type u_6) {…} :
    Type (max u_5 u_6)

(to_module : module α β)
(norm_smul : ∀ (a : α) (b : β), ‖a • b‖ = ‖a‖ * ‖b‖)
```

A normed space over a normed field is a vector space endowed with a norm which satisfies the equality ‖c • x‖ = ‖c‖ ‖x‖.

▼ Instances

- `complex.normed_space.restrict_scalars_real`
- `normed_field.to_normed_space`
- `prod.normed_space`
- `pi.normed_space`

Fig. 3. The generated documentation entry for the `normed_space` type class. The implicit arguments can be expanded by clicking on `{...}`.

inductive definition. Similarly, we print the constructor and fields of a structure mirroring the input syntax.

We do not display all of the attributes applied to a declaration, but show those in a predefined list, including `simp` and `class`. For declarations tagged as type classes, we display a collapsible list of instances of this class that appear elsewhere in the library. For definitions, we display a collapsible list of the equational lemmas that describe their associated reduction rules. We also link to the exact location where the declaration is defined in the source code.

We believe that this display achieves many of our design goals. The module documentation provides an overview of a particular theory for newcomers and general implementation details for experts. The declaration display serves as an API reference, displaying information concisely with more details readily available. The same framework works to document both the formalization and the metaprogramming components of mathlib.

4.3 Tactic Database

Lean proofs are often developed using tactics. Custom tactics can be written in the language of Lean as metaprograms, and mathlib includes many such tactics [15, Sect. 6]. It is essential for us to provide an index of the available tools explaining when and how to use them. Tactic explanations are an example of decentralized documentation. Their implementations appear in many different files, interspersed with many other declarations, but users must see a single unified list. These same concerns apply to the commands defined in mathlib, as well as to attributes and hole commands, which we do not discuss in this paper.

It is inconvenient to maintain a database of tactics separate from the library. Since mathlib changes rapidly, such a database would likely diverge from the

```
structure tactic_doc_entry :=
(entry_name               : string)
(category                 : doc_category)
(decl_names               : list name)
(tags                     : list string := [])
(description              : string      := "")
(inherit_description_from : option name := none)

add_tactic_doc
{ entry_name := "linarith",
  category   := doc_cagetory.tactic,
  tags       := ["arithmetic", "decision procedure"],
  decl_names := [`tactic.interactive.linarith] }
```

Fig. 4. The information stored in a tactic documentation entry, and the standard way to register an entry. The text associated with this entry will be the declaration doc string of `tactic.interactive.linarith`.

library before long. In addition, the doc strings for tactics—which appear as tooltips in supported editors—often contain the same text as a tactic database entry. To avoid these issues, we provide a command `add_tactic_doc` that registers a new tactic documentation entry. Another command retrieves all tactic doc entries that exist in the current environment.

A tactic doc entry (Fig. 4) contains six fields. The command `add_tactic_doc` takes this information as input. To avoid duplicating information, the `description` field is optional, as this string has often already been written as a declaration doc string. When `description` is empty, the command will source it from the declaration named in `inherit_description_from` (if provided) or the declaration named in `decl_names` (if this list has exactly one element). The HTML generation tool links each description to its associated declarations.

The `entry_name` field titles the entry. This is typically the name of the tactic or command, and is used as the header of the doc entry. The `category` field is either `tactic`, `command`, `hole_command`, or `attribute`. These categories are displayed on separate pages. The `decl_names` field lists the declarations associated with this doc entry. Many entries document only a single tactic, in which case this list will contain one entry, the implementation of this tactic.

The `tags` field contains an optional list of tags. They can be used to filter entries in the generated display. The command can be called at any point in any Lean file, but is typically used immediately after a new tactic is defined, to keep the documentation close to the implementation in the source code. The HTML display allows the user to filter declarations by tags—e.g. to view only tactics related to arithmetic.

4.4 Library Notes

The interface surrounding a definition is often developed in the same file as that definition. We typically explain the design decisions of a given module in the

```
-- declare a library note about instance priority
/-- Certain instances always apply during type class resolution... -/
library_note "lower instance priority"

-- reference a library note in a declaration doc string
/-- see Note [lower instance priority] -/
@[priority 100]
instance t2_space.t1_space [t2_space α] : t1_space α := ...

-- print all existing library notes
run_cmd get_library_notes >>= trace
```

Fig. 5. Library notes can be declared, referenced, and collected anywhere in mathlib.

file-level documentation. However, some design features have a more distributed flavor. An example is the priority of type class instances (Sect. 3.3). There are guidelines for choosing a priority for a new instance, and an explanation why these guidelines make sense, but this explanation is not associated with any particular module: it justifies design decisions made across dozens of files.

We use a mechanism that we call *library notes* (Fig. 5), inspired by a technique used in the Glasgow Haskell Compiler [14] project to document these distributed design decisions. A library note is similar to a module doc string, but it is identified by a name rather than a file and line. As with tactic doc entries, we provide commands in mathlib to declare new library notes and retrieve all existing notes.

The documentation processing tool generates an HTML page that displays every library note in mathlib. When these notes are referenced in other documentation entries with the syntax `Note [note name]`, they are linked to the entry on the notes page. Library notes are also often referenced in standard comments that are not displayed in documentation. These references are useful for library developers to justify design decisions in places that do not face the public.

5 Conclusion

Although there are a growing number of large libraries of formal proofs, both mathematical and otherwise, little has been written about best practices for maintaining and documenting these libraries. Ringer et al. [18] note the gap between proof engineering and software engineering in this respect. Andronick [1] describes the large-scale deployment of the seL4 verified microkernel, focusing on the social factors that have led to its success; Bourke et al. [4] describe technical aspects of maintaining this project. Other discussions of large libraries [3,10] touch on similar topics. Wenzel [22] explains the infrastructure underlying the Isabelle Archive of Formal Proofs (AFP), including progress toward building the AFP with semantic document markup.

Sakaguchi [19] describes a tool for checking and validating the hierarchy of mathematical structures in the Coq Mathematical Components library [13], a task in the same spirit as our type class linters. Cohen et al. [6] implement a related tool which greatly simplifies changing this hierarchy.

It is hard to quantify the effect that our linters and documentation have had on the mathlib community. Fixing issues identified by the `instance_priority` and `dangerous_instance` linters led to performance boosts in the library. Removing unusable instances and simplification lemmas has also improved performance and decluttered trace output. More noticeable is the effect on the workload of maintainers, who can now spend more review time on the deeper parts of library submissions. Similarly, inexperienced contributors worry less about introducing subtle mistakes into the library. Users at all levels report frequent use of the HTML documentation, especially to find information that is not easily available in an interactive Lean session, such as the list of instances of a given type class.

So far we have only implemented the very basic sanity checks on simp lemmas described in Sect. 3.4. There are also other properties of term rewriting systems that we want for the simp set, such as confluence and termination. Kaliszyk and Sternagel [12] have used completion of term rewriting systems to automatically derive a simp set for the HOL Light standard library. We plan to implement a more manual approach, where a linter detects the lack of local confluence and prints a list of equations for the non-joinable critical pairs. It is then up to the user to decide how to name, orient, and generalize these new equations.

The current linter framework considers each declaration locally, but we anticipate the need for global tests. The `simp_nf` linter already goes beyond strictly local checking: it considers the entire simp set. Another global linter could check the termination of the simp set. This is a much harder challenge, since checking termination is undecidable in general. We plan to investigate the integration of external termination checkers such as AProVE [9].

While many of the features we present are specific to Lean, we believe that the general considerations apply more broadly: automated validation and documentation seem essential for a sustainable and scalable library of formal proofs. Especially in regard to documentation, there is a definite path for coordination between libraries and systems, possibly aided by tools from the mathematical knowledge management community.

Acknowledgments. We thank Jeremy Avigad and Jasmin Blanchette for comments on a draft of this paper, and Bryan Gin-ge Chen for many contributions to the mathlib documentation effort.

References

1. Andronick, J.: Successes in deployed verified software (and insights on key social factors). In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) FM 2019. LNCS, vol. 11800, pp. 11–17. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30942-8_2

2. Avigad, J., de Moura, L., Kong, S.: Theorem Proving in Lean. Carnegie Mellon University (2014)
3. Bancerek, G., et al.: The role of the Mizar Mathematical Library for interactive proof development in Mizar. J. Autom. Reasoning **61**(1–4), 9–32 (2018). https://doi.org/10.1007/s10817-017-9440-6
4. Bourke, T., Daum, M., Klein, G., Kolanski, R.: Challenges and experiences in managing large-scale proofs. In: Jeuring, J., et al. (eds.) CICM 2012. LNCS (LNAI), vol. 7362, pp. 32–48. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31374-5_3
5. Buzzard, K., Commelin, J., Massot, P.: Formalising perfectoid spaces. In: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, pp. 299–312. Association for Computing Machinery, New York (2020). https://doi.org/10.1145/3372885.3373830
6. Cohen, C., Sakaguchi, K., Tassi, E.: Hierarchy Builder: algebraic hierarchies made easy in Coq with Elpi, February 2020. https://hal.inria.fr/hal-02478907
7. Dahmen, S.R., Hölzl, J., Lewis, R.Y.: Formalizing the solution to the cap set problem. In: Harrison, J., O'Leary, J., Tolmach, A. (eds.) 10th International Conference on Interactive Theorem Proving (ITP 2019). Leibniz International Proceedings in Informatics (LIPIcs), vol. 141, pp. 15:1–15:19. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2019). https://doi.org/10.4230/LIPIcs.ITP.2019.15
8. Ebner, G., Ullrich, S., Roesch, J., Avigad, J., de Moura, L.: A metaprogramming framework for formal verification. PACMPL **1**(ICFP), 34:1–34:29 (2017). https://doi.org/10.1145/3110278
9. Giesl, J., et al.: Analyzing program termination and complexity automatically with AProVE. J. Autom. Reasoning **58**(1), 3–31 (2017). https://doi.org/10.1007/s10817-016-9388-y
10. Gonthier, G., et al.: A machine-checked proof of the odd order theorem. In: ITP 2013, pp. 163–179 (2013). https://doi.org/10.1007/978-3-642-39634-2_14
11. Han, J.M., van Doorn, F.: A formal proof of the independence of the continuum hypothesis. In: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, pp. 353–366. Association for Computing Machinery, New York (2020). https://doi.org/10.1145/3372885.3373826
12. Kaliszyk, C., Sternagel, T.: Initial experiments on deriving a complete HOL simplification set. In: Blanchette, J.C., Urban, J. (eds.) PxTP 2013. EPiC Series in Computing, vol. 14, pp. 77–86. EasyChair (2013)
13. Mahboubi, A., Tassi, E.: Mathematical Components (2017)
14. Marlow, S., Peyton-Jones, S.: The Glasgow Haskell Compiler. In: Brown, A., Wilson, G. (eds.) The Architecture of Open Source Applications, Volume II (2012)
15. The mathlib Community: The Lean mathematical library. In: CPP, pp. 367–381. ACM, New York(2020). https://doi.org/10.1145/3372885.3373824
16. de Moura, L., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The Lean theorem prover (system description). In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 378–388. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_26
17. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): Isabelle/HOL - A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
18. Ringer, T., Palmskog, K., Sergey, I., Gligoric, M., Tatlock, Z.: QED at large: a survey of engineering of formally verified software. Found. Trends® Program. Lang. **5**(2–3), 102–281 (2019). https://doi.org/10.1561/2500000045

19. Sakaguchi, K.: Validating mathematical structures. arXiv (2020). https://arxiv.org/abs/2002.00620
20. Selsam, D., Ullrich, S., de Moura, L.: Tabled typeclass resolution (2020). https://arxiv.org/abs/2001.04301
21. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad-hoc. In: Proceedings of POPL 1989, pp. 60–76 (1989). https://doi.org/10.1145/75277.75283
22. Wenzel, M.: Isabelle technology for the Archive of Formal Proofs with application to MMT (2019). https://arxiv.org/abs/1905.07244