

Automatically Generalizing Theorems Using Typeclasses

Alex J. Best¹

¹ *Department of Mathematics, Vrije Universiteit Amsterdam, De Boelelaan 1105, 1081 HV Amsterdam, The Netherlands*

Abstract

When producing large formally verified mathematical developments that make use of typeclasses it is easy to introduce overly strong assumptions for theorems and definitions. We consider the problem of recognizing from the elaborated proof terms when typeclass assumptions are stronger than necessary. We introduce a metaprogram for the Lean theorem prover that finds and informs the user about possible generalizations.

Keywords

automation, formal proof, interactive theorem prover, Lean theorem prover, library maintenance, theorem prover, typeclasses CEUR-WS

1. Motivation

Developing and maintaining large libraries of formalized mathematics in an interactive theorem prover, such as the libraries `mathlib` [1] in Lean [2], `Mathematical Components` [3] in Coq [4], or the `Archive of Formal Proofs` and base libraries for Isabelle/HOL [5] is a time consuming process. In addition to the overall design and organisation of the library, inclusion of useful results and writing of proofs, library maintenance requires maintaining interoperability of theories and deduplication of overlapping or identical results. Contributors to such a library with a specific formalization goal in mind often prove background results in the generality that they are needed. When such work is included into a large library it is desirable that all results be maximally useful for other potential use cases and not overly specific, this often leads to time consuming refactoring efforts before new material can be included.

The goal of this paper is to demonstrate that when typeclasses are used to implement hierarchies of assumptions on formalized objects it is possible to automate the process of generalizing results by relaxing typeclass assumptions. Pons [6] has previously considered a similar problem, and describes a mechanism for generalizing assumptions about functions and types, but without specifically considering hierarchies of typeclasses.

Our focus is mostly on the applications to the formalization of mathematics, it is also conceivable however that relaxing typeclass assumptions can be of use for other users of interactive theorem provers, such as for software verification.

FMM 2021 – Fifth Workshop on Formal Mathematics for Mathematicians, 30–31 July 2021


✉ alex.j.best@gmail.com (A. J. Best)

🌐 <https://alexjbest.github.io/> (A. J. Best)

🆔 0000-0002-5741-674X (A. J. Best)



© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

2. Typeclasses

Typeclasses were introduced by Wadler [7] as a way to provide polymorphism in strongly typed programming languages. They are used in interactive theorem provers [8] and their libraries of formalized mathematics to manage levels of mathematical structures on an object [9, 10]. When a term is used the user does not ordinarily supply typeclass arguments, instead they are inserted automatically by a typeclass system that finds instances to match the needed argument, possibly by chaining together the instances declared in the library. From the perspective of interactive theorem provers, typeclasses reduce the burden on the user of the tool to be explicit about which instances are used and in fact hide them completely from the end user.

Commonly used typeclass hierarchies for mathematical libraries include: algebraic structures, including binary operations and their properties, properties of relations and ordering, topological spaces, (pseudo-)metric spaces, and their properties. Additionally mixtures of the aforementioned hierarchies that interact, such as ordered algebraic structures, topological groups and normed algebraic structures may also be included in the typeclass system, leading to a highly nontrivial hierarchy.

In addition to the situation mentioned in the introduction, where results introduced for one purpose may be generalized to a more widely useful form, generalization can also become possible when new typeclasses are inserted into an existing hierarchy. For instance in the mathlib library the concepts of non-unital and/or non-associative rings were added in March 2021¹, several years after rings were introduced in core Lean. In the interim period several thousand lemmas and theorems were added to mathlib concerning rings. This leaves the difficult library maintenance problem of filling out the library of lemmas for these newly introduced algebraic structures, without duplicating existing work.

3. Examples

We introduce some of the key considerations by looking at some examples in the Lean theorem prover using the mathlib library.

Consider the following lemma:

```
lemma mul_inv {G : Type*} [ordered_comm_group G] (a b : G) : (a *
  b)-1 = a-1 * b-1 :=
by rw [mul_inv_rev, mul_comm]
```

It is clear that the statement and the proof require a commutative group structure, but that the assumption that G be an *ordered* commutative group plays no role. Despite the shortness of the statement and proof script the generated proof term contains many chains of typeclass instances including those shown in Figure 1.

There are in addition other instances between these classes, not made use of in the original proof, but available to the typeclass system.

The only typeclass parameters actually required to satisfy the assumptions of the applied lemmas `mul_inv_rev` and `mul_comm` are `group G` and `comm_semigroup G`. We cannot

¹See <https://github.com/leanprover-community/mathlib/pull/6786>.

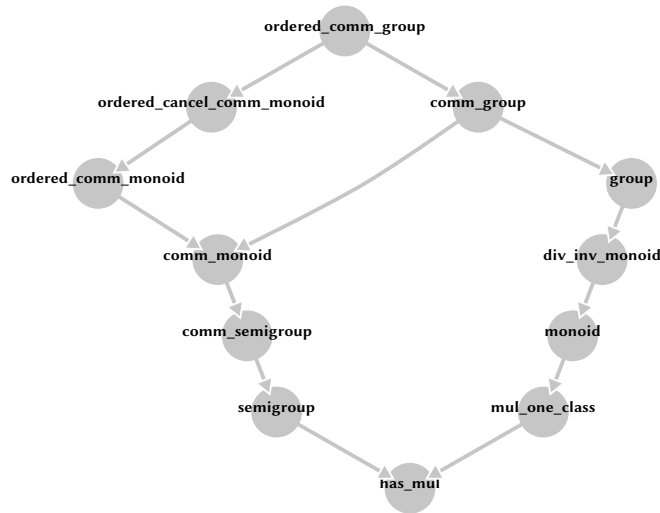


Figure 1: Typeclass chains in the proof of `mul_inv`.

replace the assumption `ordered_comm_group G` with this pair of assumptions however as they have overlapping fields, they both should refer to the same multiplication operation. This is visible in the figure as there is an instance chain from each of these to `has_mul`. Instead a meet of these two typeclasses in the typeclass graph should be chosen as the generalization, in this case it is `comm_group`.

The following theorem states that given a ring homomorphism between two fields and a natural number p , one of the fields has characteristic p if and only if the other has characteristic p (including $p = 0$):

```
lemma ring_hom.char_p_iff_char_p {K L : Type*} [field K] [field L]
(f : K →+* L) (p : ℕ) : char_p K p ↔ char_p L p :=
begin
  split;
  { introI _c, constructor, intro n,
    rw [← @char_p.cast_eq_zero_iff _ _ p _c n, ←
      f.injective.eq_iff, f.map_nat_cast, f.map_zero] }
end
```

We see that the proof script splits the iff statement into each direction, but both directions are proved by the same tactic block. It is non-trivial to determine just by reading the proof given what the weakest assumptions possible are, and it is not immediately clear from the statement either. While it is of course possible to work out the correct generality by hand by inspecting the assumptions needed for each lemma applied, we hope this provides a realistic example where the “right” answer is not immediately clear.

In fact the weakest possible typeclass assumptions for the proof of `ring_hom.char_p_iff_char_p` are precisely the assumptions needed to apply `ring_hom.injective` to perform the rewrite `← f.injective.eq_iff`. These are

that K should be a division ring, and L should be a nontrivial semiring. This example highlights another important point; simply taking the meet in the typeclass graph of the required classes is not always optimal. The meet of the classes `nontrivial` and `semiring` in the present version of `mathlib` is `domain`, once again an overly strong assumption, as there is no reason not to simply assume `[nontrivial K] [semiring K]`. We should allow one assumption to be replaced by several distinct assumptions if there is no conflict in doing so. The typeclasses `nontrivial` and `semiring` both provide instances of the `nonempty` typeclass, which contains no data, so it is a `Prop` and hence a subsingleton in Lean. Thus despite the fact that they do both provide instances of the same class, the instances will be equal and there is no issue allowing both assumptions separately.

4. Algorithm

We now describe in more detail an algorithm to find possible typeclass generalizations, given a proof term.

One important thing to note here is that we also handle typeclasses with more than one argument that are partially applied. For instance it is desirable to be able to generalize `group G` to `has_pow G ℤ`. This is distinct from `monoid G` generalizing to `has_pow G ℕ`, so we treat these partially applied typeclasses as the basic objects of interest, we call them *bound classes*.

The first step is to traverse the entire environment and build a directed graph of bound classes with instances between them. We also precompute the transitive closure and a topological sort of this graph.

Algorithm 1: Generalising typeclasses assumptions in a declaration.

```

foreach typeclass argument  $c$  do
   $S \leftarrow []$ ;
  foreach maximal chain of instances ending in  $c$  in the statement or proof do
    | add the head bound class of the chain to  $S$ ;
  end
   $T \leftarrow$  the set of strongly connected components for the subgraph of bound classes
    reachable from  $S$  (ignoring subsingletons);
   $o \leftarrow []$ ;
  foreach  $u$  in  $T$  do
    | add a meet of  $u$  to  $o$ ;
  end
  return  $o$ 
end

```

5. Implementation

We have implemented the above ideas for the Lean theorem prover as a metaprogram written in Lean itself². This means that no external tools beyond Lean itself are needed for users to

²the implementation is available at <https://github.com/alexjbest/lean-generalisation>

integrate this tool into their Lean developments. We may also make use of the linter framework [11] which provides a convenient means for users to run different checks on the current file, and for continuous integration tools to regularly lint an entire library on multiple cores. This introduces a dependency on mathlib itself, but this tool can be imported and applied to any Lean development, and the import removed when generalization is complete.

We note that it is important to precompute and cache useful information about the typeclass graph, such as a topological sort of the graph, and its transitive closure. With the current size of mathlib this precomputation is quite reasonable, even with non-optimal algorithms.

6. Summary of results

By running the metaprogram described above on a current version of the mathlib library³, which contains around 80,000 declarations we can gauge the effectiveness of this approach. Note that the process of generalizing typeclass assumptions using such a tool is naturally an iterative process, by generalizing one theorem other theorems that made use of the original theorem may also become generalizable. This makes it most natural to use such a tool interactively in a file when formalizing some theory. Nevertheless, for ease of measurement we simply apply a single pass in our test, thus there are likely many more interesting or useful typeclass relaxations that can be discovered by such a tool if used iteratively.

In total the current implementation produces 2877 results, some of these are false positives, or at least not particularly useful generalizations. To get a sense of what generalizations were possible in a library like mathlib, we list in Table 1 some of the most common generalizations found by this pass.

Here several generalizations such as changing a field typeclass to a (commutative) ring give rather large relaxations of structure, and potentially introduce a large number of new useful lemmas in the library.

The replacement of `integral_domain` with `comm_ring` and `no_zero_divisors` appears mathematically trivial, but in fact results in the removal of the assumption that the ring in question be nontrivial (which is part of mathlib's definition of an integral domain). Replacements such as this are mathematically not significant and it is not the case that this generalization would be useful to an end user directly. However, making this generalization relieves the user of such a theorem from the burden of adding a non-trivial assumption which can then proliferate through the library. These small assumptions that nonetheless need verifying whenever a theorem is applied can slow down formalization efforts and make future formalization unduly time consuming and tedious compared with the actual mathematics being formalized.

It is important that such a tool be usable interactively when working on formalizing a mathematical theory. Running our implementation on the mathlib file `topology/ordered/basic.lean` (the second longest in mathlib by number of lines) takes between 1 and 2 minutes on a modern laptop, depending on if the typeclass graph is already computed and cached. This checks 475 declarations and reports 183 results currently.

³commit 29b63a7e91d079b159dfc2cf0fb4d2a1ce1c409b, not including core Lean library

Table 1
Generalizations found in the Lean mathlib.

Original typeclass	Replacement typeclasses (number of times replaced)
comm_ring	comm_semiring (42), ring (40), semiring (27), has_zero (8)
add_comm_group	add_comm_monoid (96), add_group (5), sub_neg_monoid (5), {has_add, has_neg, has_zero} (3)
semiring	non_assoc_semiring (53), non_unital_non_assoc_semiring (23), {add_comm_semigroup, has_one} (13), {add_comm_monoid, has_mul} (8), has_mul (7), has_zero (5)
field	division_ring (23), comm_ring (12), integral_domain (12), semiring (7), domain (4), ring (4), has_inv ring (3)
ring	semiring (55), non_assoc_semiring (8), {add_group, has_mul} (4), {has_add, has_neg, mul_zero_one_class} (4)
preorder	has_lt (36), has_le (31), {has_le, is_refl} (3), {has_lt, is_asymm} (3), {has_lt, is_irrefl, is_trans} (3)
comm_semiring	semiring (26), monoid (10), add_zero_class (4), {has_mul, is_associative, is_commutative} (4), has_pow (4), mul_one_class (4)
normed_space	module (23), semi_normed_space (38)
add_monoid	add_zero_class (51), {has_add, has_zero} (5)
monoid	mul_one_class (34), has_mul (11), {has_mul, has_one} (5), has_pow (5)
module	has_scalar (20), distrib_mul_action (8), mul_action (6)
normed_group	semi_normed_group (36), has_norm (10), has_nnorm (3)
integral_domain	comm_ring (15), domain (8), {comm_ring, no_zero_divisors} (7), comm_monoid (3), {has_mul, has_zero, no_zero_divisors} (2), {no_zero_divisors, semiring} (2)
partial_order	preorder (33)

7. Limitations and Further Work

By only inspecting the existing proof of a theorem naturally not all possible generalizations of theorems can be found. It is quite easy to write proofs that require stronger assumptions than necessary, especially when using powerful automation. Trying to remove this limitation completely shifts the problem to one of automatic theorem proving. However, an intermediate problem where some automation may be possible is to find theorems for which the same proof script proves a generalized theorem. This is especially likely with small proofs that make heavy use of automation to begin with. Here a brute force strategy of weakening typeclasses in theorem statements and re-running the same proof script seems far too slow to be used on the scale of a large library in a prover such as Lean, however similar techniques have been used in the Mizar system [12]. Nevertheless, a tool to automate this process could still be useful for small new developments.

Currently the system as implemented in Lean will provide to the user a list of possible typeclass generalizations by printing the name of the declaration and describing the argument that can be generalized. The structure of Lean files allows for variables to be defined within a section these assumptions can then hold for all declarations in large chunks of the file. This decoupling of

the location of the assumptions used for each theorem and the theorems themselves sometimes makes it tedious to rearrange the file to actually implement the suggestions. Therefore better tooling to reorganise Lean files in a content-aware manner will improve the usability of the tool described in this paper.

A mild extension of this work would be to consider not just assumptions, but also concrete types in a theorem statement. For instance theorems proven about explicitly constructed types such as the natural, rational or real numbers often make use of (ordered) algebraic properties of these types, which are filled in via typeclasses. Recognizing these typeclass chains in the same way as described here could allow parts of libraries concerning these concrete types to be generalized to include any object satisfying some typeclass hypotheses.

A core design principle of Lean 4, the next iteration of the Lean theorem prover [13], is that the majority of the system be written in Lean itself. This allows for extensibility of the Lean system by users, rather than having to build different versions of Lean for user extensions. Using this it may be possible to integrate the minimisation of typeclass assumption into Lean's own typeclass system, providing a more accurate and efficient tool.

8. Conclusion

Recognizing too strong typeclass assumptions automatically is possible, and can be done efficiently. Such tools can save maintainers of mathematical libraries time by providing more general results for free from an existing library, and provide users of formal proof systems with interesting information about what generality theorems hold in.

8.1. Acknowledgements

I would like to thank Floris van Doorn for helpful discussions on an earlier version of this work, and the referees for their useful comments and suggestions. This work was supported by the Hariri Institute for Computing, the Simons Collaboration on Arithmetic Geometry, Number Theory, and Computation, via Simons Foundation grant #550023, and NWO Vidi grant 639.032.613.

References

- [1] T. mathlib Community, The lean mathematical library, in: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, Association for Computing Machinery, New York, NY, USA, 2020, pp. 367–381. URL: <https://doi.org/10.1145/3372885.3373824>. doi:10.1145/3372885.3373824.
- [2] L. de Moura, S. Kong, J. Avigad, F. van Doorn, J. von Raumer, The Lean Theorem Prover (System Description), in: A. P. Felty, A. Middeldorp (Eds.), Automated Deduction - CADE-25, volume 9195, Springer International Publishing, Cham, 2015, pp. 378–388. URL: http://link.springer.com/10.1007/978-3-319-21401-6_26. doi:10.1184/R1/6492815.v1, series Title: Lecture Notes in Computer Science.

- [3] A. Mahboubi, E. Tassi, *Mathematical Components*, Zenodo, 2021. URL: <https://zenodo.org/record/4457887>. doi:10.5281/zenodo.4457887, version Number: 1.0.1.
- [4] T. C. D. Team, *The Coq Proof Assistant*, 2021. URL: <https://zenodo.org/record/4501022>. doi:10.5281/zenodo.4501022.
- [5] T. Nipkow, L. C. Paulson, M. Wenzel, *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283, Springer Science & Business Media, 2002.
- [6] O. Pons, *Generalization in Type Theory Based Proof Assistants*, in: P. Callaghan, Z. Luo, J. McKinna, R. Pollack, R. Pollack (Eds.), *Types for Proofs and Programs*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2002, pp. 217–232. doi:10.1007/3-540-45842-5_14.
- [7] P. Wadler, S. Blott, *How to make ad-hoc polymorphism less ad hoc*, in: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '89*, Association for Computing Machinery, New York, NY, USA, 1989, pp. 60–76. URL: <https://doi.org/10.1145/75277.75283>. doi:10.1145/75277.75283.
- [8] M. Wenzel, *Type classes and overloading in higher-order logic*, in: E. L. Gunter, A. Felty (Eds.), *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 1997, pp. 307–322. doi:10.1007/BFb0028402.
- [9] B. Spitters, E. van der Weegen, *Developing the Algebraic Hierarchy with Type Classes in Coq*, in: M. Kaufmann, L. C. Paulson (Eds.), *Interactive Theorem Proving*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2010, pp. 490–493. doi:10.1007/978-3-642-14052-5_35.
- [10] B. Spitters, E. van der Weegen, *Type Classes for Mathematics in Type Theory*, *Mathematical Structures in Computer Science* 21 (2011) 795–825. URL: <https://www.cambridge.org/core/journals/mathematical-structures-in-computer-science/article/abs/type-classes-for-mathematics-in-type-theory/7D22C0A97AE7B724237B2210222D3ED9#>. doi:10.1017/S0960129511000119, arXiv: 1102.1323.
- [11] F. van Doorn, G. Ebner, R. Y. Lewis, *Maintaining a Library of Formal Mathematics*, in: C. Benz Müller, B. Miller (Eds.), *Intelligent Computer Mathematics*, Lecture Notes in Computer Science, Springer International Publishing, Cham, 2020, pp. 251–267. doi:10.1007/978-3-030-53518-6_16.
- [12] J. Alama, *Eliciting Implicit Assumptions of Mizar Proofs by Property Omission*, *Journal of Automated Reasoning* 50 (2013) 123–133. URL: <http://link.springer.com/10.1007/s10817-012-9264-3>. doi:10.1007/s10817-012-9264-3.
- [13] L. d. Moura, S. Ullrich, *The Lean 4 Theorem Prover and Programming Language*, in: A. Platzer, G. Sutcliffe (Eds.), *Automated Deduction – CADE 28*, Lecture Notes in Computer Science, Springer International Publishing, Cham, 2021, pp. 625–635. doi:10.1007/978-3-030-79876-5_37.