Automatically Generalizing Theorems Using Typeclasses

Alex J. Best - Vrije Universiteit Amsterdam

FMM 2021 (PART OF CICM 2021)

These slides are online at:

http://alexjbest.github.io/talks/auto-generalization/

Motivation

1. When developing a formal mathematical library it is time consuming to add and modify basic results, the barriers to formalization are often not having small lemmas in precisely the right form.
2. Want to prototype / demonstrate potential applications of formalization to mathematical practice, i.e. develop tools that might genuinely help in research mathematics.

Problem: Often theorems are stated with stronger assumptions than are really needed, can this be detected automatically?

More specifically: this commonly occurs when typeclasses are used to manage hierarchies of mathematical structures (e.g. algebraic, topological and order structures)

Examples will be in Lean but the idea should apply more generally.

Examples

A fake Lean 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]
```

This assumes `[ordered_comm_group G]` but makes no mention of an order! In principle the proof could still require it, but it doesn't.
Anyone trying to use this theorem for a `comm_group G` would get an error!
The "correct" assumption is `[comm_group G]`, when this is changed the proof script does not need modifying.

A real lemma from `mathlib`:

```
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
```

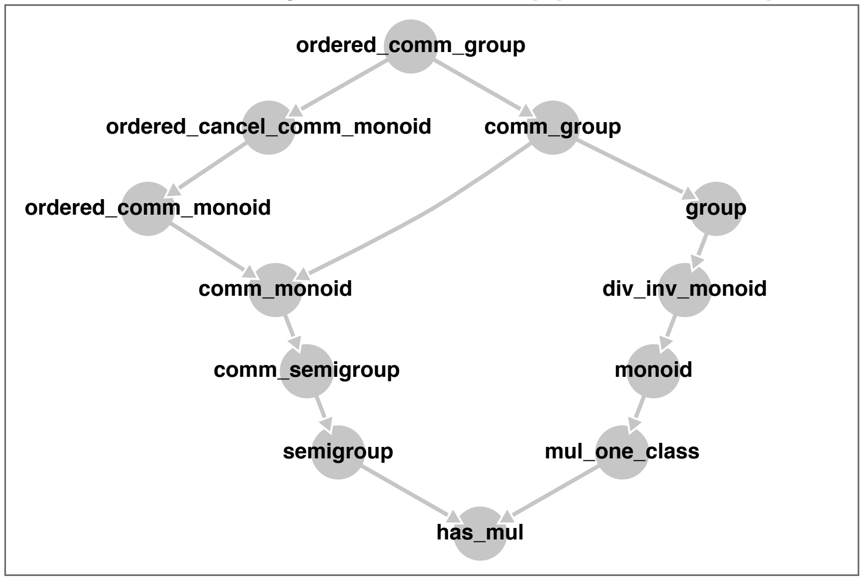
Correct assumptions: `[division_ring K] [nontrivial L] [semiring L]`

How do typeclasses work?

Lean automatically generates the terms used in the proof:

```
@ordered_comm_group.to_ordered_cancel_comm_monoid G _inst_1
@ordered_cancel_comm_monoid.to_ordered_comm_monoid ...
@ordered_comm_group.to_comm_group G _inst_1
...
```

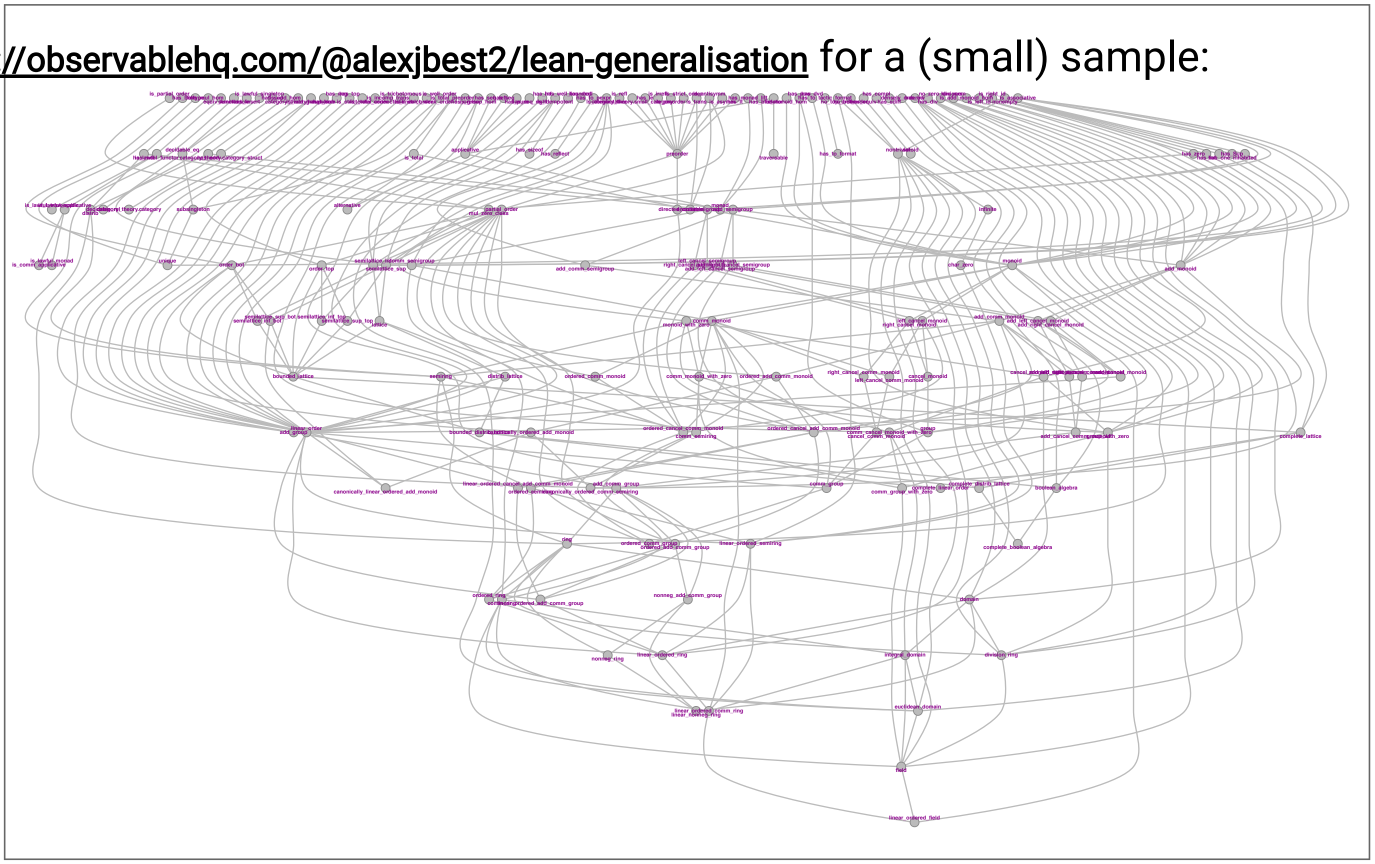
via typeclass resolution. The following instances appear in the proof term:



Typeclasses are used when the structure being inferred is (locally) unique.

What does the `mathlib` typeclass graph actually look like?

See https://observablehq.com/@alexjbest2/lean-generalisation for a (small) sample:



(*) Only terms reachable from `linear_ordered_field` here, generated Jan 2021, it has grown since then!

Why is detecting this reasonable?

- Because the aim is only to change the typeclass assumptions to match those actually used in the proof, the proof script will (likely) not need changing (*).
- The linter shouldn't need to do any nontrivial proving itself, so it should be comparatively fast.

A metaprogram which attempts to weaken assumptions and then fix the proofs would need to be far more involved!

(*) Its possible to write a long proof term by hand that looks like it was generated by typeclass inference

What are the most general typeclass assumptions?

- Looking at the proof term, we can determine the instances actually used to invoke the lemmas used.
- Then study the global typeclass graph to determine minimal assumptions providing those instances.
- Uniqueness is important, we must not introduce unequal instances of one typeclass, but we may want multiple assumptions with a common instance, if they will be guaranteed equal.
- To do this we take meets in the typeclass graph of sets of "conflicting" typeclasses (i.e. those which give instances of the same non-subsingleton typeclass).

Caveats

- Just looking at the typeclasses defined in the library is not enough.
  Partially applied typeclasses are also important.
  Want to generalize `[group G]` to `[has_pow G ℤ]` if possible or `monoid G` to `has_pow G ℕ`, so we treat these partially applied typeclasses as the basic objects of interest.
  This increases the complexity of the tool!
- It is easy to "fool" such a tool:

```
/-- Given a monoid hom `f : M →* N` and submonoid `S ⊆ M` such that
`f(S) ⊆ units N`, for all `w : M, z : N` and `y ∈ S`, we have
`z = w * (f y)⁻¹ ↔ z * f y = w`. -/
lemma submonoid.localization_map.mul_inv_right {M : Type*}
[comm_monoid M] {S : submonoid M} {N : Type*} [comm_monoid N]
{f : M →* N} (h : ∀ y : S, is_unit (f y)) (y : S) (w z) :
z = w * ↑(is_unit.lift_right (f.mrestrict S) h y)⁻¹ ↔
z * f y = w := by rw [eq_comm, mul_inv_left h, mul_comm, eq_comm]
```

the use of `mul_comm` here isn't actually needed, but its inclusion is enough to make it seem like `[comm_monoid M]` is required.

Implementation

This is implemented in Lean (no external dependencies):

https://github.com/alexjbest/lean-generalisation

- Implemented using the linter framework, just import the tool and add `#lint` after your declaration.
- Uses `native.rb_lmap` to represent DAGs, find reachable sets and topological sorts - untrusted, nothing can be proven about this implementation!
- Caches the typeclass graph and useful associated data.
- Now quick enough for interactive use in *most* files.

Some technical gotchas

- Must inspect the statement as well as the proof! Sometimes the proof is "just" `rfl`
- Some things look like generalisations but are mathematical no-ops (solution: keep a list of bad type synonyms)
- Sometimes terms need to be eta reduced (e.g. TC mechanism puts the term `(λ a b, _inst_1 a b)` in the proof instead of `_inst_1`)

Example output:

```
/- The `generalisation_linter` linter reports: -/
/- typeclass generalisations may be possible -/
-- topology\algebra\group.lean
#print nhds_translation_mul_inv /- _inst_3: topological_group ↝ has_continuous_mul
 -/
#print nhds_translation_add_neg /- _inst_3: topological_add_group ↝
has_continuous_add
 -/
#print quotient_add_group.is_open_map_coe /- _inst_3: topological_add_group ↝
has_continuous_add
 -/
#print quotient_group.is_open_map_coe /- _inst_3: topological_group ↝
has_continuous_mul
 -/
#print is_open.add_left /- _inst_3: topological_add_group ↝ has_continuous_add
 -/
#print is_open.mul_left /- _inst_3: topological_group ↝ has_continuous_mul
```

This is naturally an iterative process: once the assumptions lemmas are changed dependent declarations may become generalizable.

- The current implementation is fast enough for interactive use.
- Alternative would be to transform declarations in place when working through the environment

Some common replacements in `mathlib`

Running a single pass of this on my laptop on the 80,000 declarations in `mathlib` finished overnight.

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)

Original typeclass	Replacement typeclasses (number of times replaced)
module	has_scalar (20), distrib_mul_action (8), mul_action (6)
normed_group	semi_normed_group (36), has_norm (10), has_nnnorm (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)

Other systems

Typeclasses are used in a similar way in Coq, Isabelle/HOL, Haskell, ... I don't know of this having been attempted in any of these systems, but would love to know if it has (or will be)!

Some related, but different work:

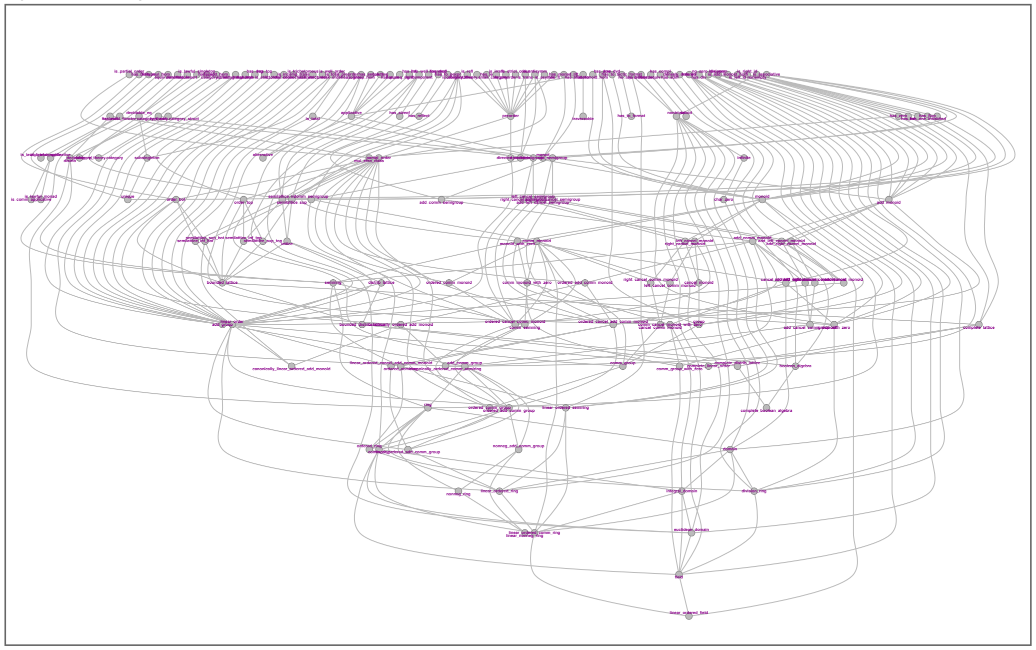
- "Eliciting Implicit Assumptions of Mizar Proofs by Property Omission" by Jesse Alama
- "Generalization in Type Theory Based Proof Assistants" by Olivier Pons

Future work

- Finish/cleanup the implementation.
- PR linter changes to the mathlib library!
- Can we generalize explicit types to only their used properties, e.g. ℝ to {R : Type*} [linear_ordered_field R]?
- Could such a metaprogram be run by the typeclass system itself when it is invoked (in Lean 4)
- Can new definitions be suggested by such a tool?
- Better source transform tools, to reduce human input to make desired changes.
- Bundled things... not typeclass assumptions per se?
- Suggestions?

Thanks!

To the organisers, Lean development team, everyone whose tactics I've repurposed for this, the community, and you for listening!



https://github.com/alexjbest/lean-generalisation