Scalar actions in Lean's mathlib

Eric Wieser[1]

[1]*Cambridge University Engineering Department, Trumpington Street, Cambridge, CB2 1PZ, UK*

Abstract

Scalar actions are ubiquitous in mathematics, and therefore it is valuable to be able to write them succinctly when formalizing. In this paper we explore how Lean 3's typeclasses are used by mathlib for scalar actions with examples, illustrate some of the problems which come up when using them such as compatibility of actions and non-definitionally-equal diamonds, and note how these problems can be solved. We outline where more work is needed in mathlib in this area.

Keywords

Lean, mathlib, scalar multiplication, typeclass diamonds, typeclasses

2020 MSC: 68V20 (Primary), 68V35 (Secondary)

1. Introduction

In this paper, we explore some of the design decisions made in mathlib [1], a mathematical library for the Lean 3 theorem prover [2]. In particular, we look at the scalar action operator `•` as a case-study in how typeclasses are used. In section 1.2 we introduce the typeclass which provides this notation, and some of the hierarchy of stronger typeclasses that surround it. In section 2 we outline some basic actions, and in section 3 show how these are used to build more complex classes. In section 4 we show how some of the extra compatibility typeclasses associated with these actions can be repurposed to work with various restricted versions of unital associative algebras, referencing two recent contributions to mathlib that exploited this approach. In section 5 we provide a brief example of how typeclass diamonds can arise, and why they matter. We conclude in section 6 by discussing further work needed on mathlib, primarily regarding scalar *right*-actions.

1.1. Typeclasses

A central language feature used by mathlib in expressing algebraic structure is that of typeclasses [1, section 4], which are used to equip types with canonical operators and properties of those operators. A simple example is the typeclass `semigroup M`, which equips the type `M` with the operator `*` and an associativity axiom `mul_assoc`. Working with typeclasses breaks down into three parts; declaring them with the `class` keyword, providing them with the `instance` keyword, and consuming them with `[ ]` around a typeclass name.

Proceedings for the "Formal Mathematics for Mathematicians" Workshop at CICM 2021

efw27@cam.ac.uk (E. Wieser)

0000-0003-0412-4978 (E. Wieser)

© 2021 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)

The `semigroup` typeclass from mathlib is declared roughly as follows, which reads "a type `G` has a multiplication if it has a binary operator called `mul`" and "a type `G` is a semigroup if it has a multiplication and that multiplication is associative". Note that the use of **`extends`** additionally tells Lean how to obtain an instance of `has_mul G` if it has `semigroup G`.

```
class has_mul (G : Type*) :=
(mul : G → G → G)

infix * := has_mul.mul
```

```
class semigroup (G : Type*) extends has_mul G :=
(mul_assoc : ∀ a b c : G, a * b * c = a * (b * c))
```

With this typeclass in place, we can write a theorem that applies to any semigroup by writing `[semigroup G]` in our argument list, as we do in `mul_assoc₂` below. This tells Lean that whenever the `mul_assoc₂` lemma is used on a type `G`, it should perform a typeclass search for a term of type `semigroup G`. In turn, it means that inside `mul_assoc₂` we have access to the `mul_assoc` axiom of semigroups on `G`.

```
lemma mul_assoc₂ {G : Type*} [semigroup G] (a b c d : G) :
  a * (b * (c * d)) = ((a * b) * c) * d :=
by rw [mul_assoc, mul_assoc]
```

The final piece of the puzzle is how to inform the typeclass search that `semigroup G` is available for a particular type `G`. To demonstrate this, we define a structure `opposite α` that wraps a single element of an arbitrary type `α`. Using the **`instance`** keyword, we then equip it with a reversed multiplication structure, and express "For any type `α` such that `α` is itself a semigroup, `opposite α` is also a semigroup". This method of "chaining" instances is central to the power of typeclasses, and is used extensively by mathlib in situations like equipping a product of groups with a group structure, or polynomials over a ring with a ring structure.

```
-- `op` is the constructor name, `unop` is the projection
structure opposite (α : Type*) := op :: (unop : α)

instance (T : Type*) [semigroup T] : semigroup (opposite T) :=
{ mul := λ a b, opposite.op (b.unop * a.unop),
  mul_assoc := λ a b c, congr_arg opposite.op (mul_assoc c.unop b.unop a.unop).symm }
```

1.2. The `has_scalar` typeclass

The typeclass we are most interested in this paper is `has_scalar M α`, which equips a type `α` with an action by elements of `M` denoted `m • a`. In practice, this is almost always used for group actions, which are actions that satisfies the additional fields in `mul_action M α`:

```
class has_scalar (M : Type*) (α : Type*) := (smul : M → α → α)

infixr ` • `:73 := has_scalar.smul
```

```
class mul_action (M : Type*) (α : Type*) [monoid M] extends has_scalar M α :=
(one_smul : ∀ a : α, (1 : M) • a = a)
(mul_smul : ∀ (x y : M) (a : α), (x * y) • a = x • y • a)
```

Note here that because we use `[monoid M]` instead of `extends monoid M`, we are stating that `mul_action M α` requires `M` to already be equipped with a monoid structure, rather than allowing `mul_action M α` to itself provide that structure.

mathlib extends these two typeclasses with a variety of additional axioms (i.e., fields holding proofs) for when `M` and `α` are themselves equipped with extra structures, such as distributivity over addition and actions by zero. The left hand side of fig. 1 shows the majority of these typeclasses, while details of their fields can be found either in [1, section 5.1] or in the mathlib docs.

2. Elementary actions

Scalar actions can be roughly divided into two types: elementary actions which are intrinsic to a particular family of types, and derived actions which operate elementwise on "bigger" types built out of smaller types. We will start by giving some examples of the former.

2.1. Left multiplication

One of the simplest actions we can construct is that of left-multiplication, with `a * b = a • b`, which mathlib provides as follows.

```
instance has_mul.to_has_scalar (α : Type*) [has_mul α] : has_scalar α α :=
{ smul := (*) }
```

As the properties of the multiplication on `α` becomes stronger, so do those of this scalar action on `α`; for instance when we have `monoid α` we can deduce `mul_action α α`, and when we have `semiring α` we can deduce `module α α`. The right-hand side of fig. 1 shows these available left multiplication structures with grey arrows.

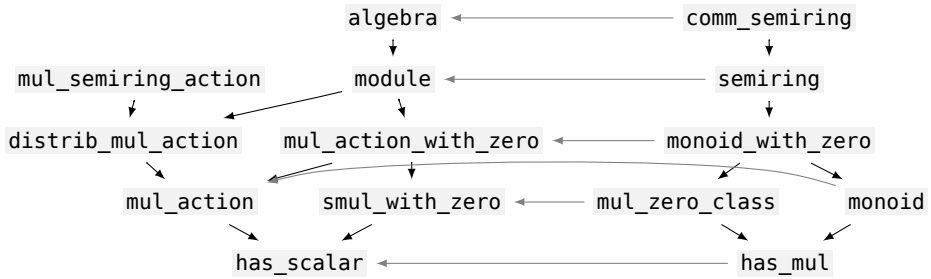

Figure 1: Hierarchy of scalar action typeclasses. Arrows indicate implications. Grey arrows indicate implied left-multiplication actions.

2.2. Repeated addition and subtraction

Another simple action we can construct is that of repeated addition when `α` is a commutative additive monoid, an instance of `module ℕ α`, which can be defined recursively for a natural number as `(0 : ℕ) • x = 0` and `∀ n : ℕ, (n + 1) • x = n • x + x`. A similar approach can be used to define a `module ℤ α` instance when `α` additionally forms an additive group. These are respectively promoted to `algebra ℕ α` and `algebra ℤ α` structures when `α` forms a `semiring` or `ring`.

3. Derived actions

A typical example of a module action might be that of a scalar $\mathbb{R}$ on the vector space $\mathbb{R}^3$ (`fin 3 → ℝ`), which multiplies each component separately. After making the obvious generalization to an arbitrary type and index set, the easy way to write this down would be as follows, where again we can provide a stronger `module α (ι → α)` if we know `α` forms a `semiring`.

```
instance function.has_scalar (ι α : Type*) [has_mul α] :
  has_scalar α (ι → α) :=
{ smul := λ r v, (λ i, r * v i) }
```

This definition is perfectly fine for the action we wanted, but we can still generalize it much more. Consider now the action on matrices `ι₁ → ι₂ → R` by their coefficients `R`. We would like to show `has_scalar R (ι₁ → ι₂ → R)`, but that doesn't match the `function.has_scalar` instance we just defined. While we could obviously define this operation trivially just as we did there, we would have to do so again if working with a vector of matrices or similar.

A better approach here is to exploit the chaining that occurs during typeclass search, and define our action as:

```
instance function.has_scalar' (ι M α : Type*) [has_scalar M α] :
  has_scalar M (ι → α) :=
{ smul := λ r v, (λ i, r • v i) }
```

This instance is strictly more general; typeclass search will recover our original `has_scalar α (ι → α)` instance by setting `M = α` and finding `has_scalar α α` from `has_mul.to_has_scalar`, but can also find the `has_scalar R (ι₁ → ι₂ → R)` we wanted by setting `M = R` and `α = (ι₂ → R)`, and finding `has_scalar R (ι₂ → α)` by recursive application of this instance. This approach is used extensively throughout mathlib, for actions on

1. sets and products defined in terms of actions on their elements
2. polynomials defined in terms of actions on their coefficients
3. bundled homomorphisms defined in terms of actions on their codomain

Most of these actions propagate their axioms; for instance where we used `[has_scalar M α]` to implement `has_scalar M (ι → α)`, the stronger assumption of `[module M α]` would let us implement `module M (ι → α)`.

3.1. More complex derived actions

In section 3, the action we describe contains no proof obligations; we did not need to know any properties of `[has_scalar M α]` to define `has_scalar M (ι → α)`. Sometimes, the typeclasses in fig. 1 are enough to resolve this—for instance, while we can't conclude `has_scalar R (M →+ N)` from `[has_scalar R N]` as we don't know enough about this action to know if additive maps remain additive, we can conclude `distrib_mul_action R (M →+ N)` from `[distrib_mul_action R N]`.

Once we start working with types that themselves ingrain a preferred action though, we need some additional tools. For instance, the closely related types for `R`-linear maps `M →ₗ[R] N` and `R`-submodules `submodule R N` ingrain a preferred `R`-action. We can start by attempting to a general action by an arbitrary type `α`. If we do this we find ourselves left with two proof obligations, indicated by the `show ..., from` syntax.

```
instance {α R M N : Type*}
  [semiring R] [add_comm_monoid M] [add_comm_monoid N]
  [has_scalar α N] [module R M] [module R N] :
  has_scalar α (M →ₗ[R] N) :=
{ smul := λ a f, { to_fun := λ m, a • f m,
                   map_add' := λ m₁ m₂, (congr_arg _ $ f.map_add _ _).trans $
                     show a • (f m₁ + f m₂) = a • f m₁ + a • f m₂, from sorry,
                   map_smul' := λ r m, (congr_arg _ $ f.map_smul _ _).trans $
                     show a • r • f m = r • a • f m, from sorry } }
```

The goal at the `sorry` in `map_add'` tells us we need to strengthen `[has_scalar α N]` to `[monoid α] [distrib_mul_action α N]`.

The goal in `map_smul'` is more troublesome. The easy way out is to replace `α` with a commutative `R` so our statement becomes

```
instance {α M N : Type*} [comm_semiring R]
  [add_comm_monoid M] [add_comm_monoid N] [module R M] [module R N] :
  has_scalar R (M →ₗ[R] N) :=
```

and the `sorry` can be closed with `a • r • f m = (a * r) • f m = (r * a) • f m = r • a • f m` which follows from the axioms of `mul_action` and commutativity of `R`. Another approach would be to require `R` to be an `α`-algebra `[algebra α R]`, and that the `α`-action on `R` and `N` is compatible with the `R`-action on `N`.

To best solve this problem, mathlib provides two additional typeclasses about scalar actions. The first expresses the compatibility condition we would need to use `[algebra α R]` as mentioned above, as

```
class is_scalar_tower
  (M N α : Type*) [has_scalar M N] [has_scalar N α] [has_scalar M α] : Prop :=
(smul_assoc : ∀ (x : M) (y : N) (z : α), (x • y) • z = x • (y • z))
```

The name alludes to towers of algebras, which is described in more detail in [3, Section 3.2]. Our particular problem can be solved more directly with the second typeclass, `[smul_comm_class α R N]`, which expresses exactly the condition we require:

```
class smul_comm_class (M N α : Type*) [has_scalar M α] [has_scalar N α] : Prop :=
(smul_comm : ∀ (m : M) (n : N) (a : α), m • n • a = n • m • a)
```

After this typeclass was introduced in [4], the author contributed and drove the review of a large number of instances of it, most notably those for polynomials, product types, and the repeated addition actions in section 2.2.

4. Algebras and not-quite-algebras

The mathlib `algebra R A` describes an associative unital `R`-algebra over `A` given a `comm_semiring R` and `semiring A`. The definition is roughly

```
class algebra (R A : Type*) [comm_semiring R] [semiring A] extends has_scalar R A :=
(algebra_map : R →+* A)
(commutes : ∀ r x, algebra_map r * x = x * algebra_map r)
(smul_def : ∀ r x, r • x = algebra_map r * x)
```

which states that there is a canonical ring homomorphism from `R` to `A` which agrees with `•` and sends `R` to the center of `A`. This parameterization of the axioms is difficult to generalize to A being non-unital and non-associative ring. However, mathlib also provides this definition to construct an algebra from an alternate set of axioms:

```
def algebra.of_module (R A : Type*) [comm_semiring R] [semiring A] [module R A]
  (h₁ : ∀ (r : R) (x y : A), (r • x) * y = r • (x * y))
  (h₂ : ∀ (r : R) (x y : A), x * (r • y) = r • (x * y)) : algebra R A := sorry
```

If we look carefully, we note that `h₁` and `h₂` closely resemble `smul_assoc` and `smul_comm` from section 3.1, but with some `*`s substituted for `•`. But if we look back to section 2.1, we remember that when `x` and `y` are the same type, `x * y = x • y` by definition! This means that `h₁` and `h₂` correspond directly with `is_scalar_tower R A A` and `smul_comm_class R A A`, respectively.

This is a valuable insight, because it allows us to use the follow sequences of typeclass arguments interchangeably:

```
variables [comm_semiring R] [semiring A] [algebra R A]
```

```
variables [comm_semiring R] [semiring A] [module R A]
          [is_scalar_tower R A A] [smul_comm_class R A A]
```

Knowing this, it becomes immediately obvious how to generalize various statements to non-unital algebras (which were needed in [5]); we switch from from the first form to the second form, and then replace `[semiring A]` with `[non_unital_semiring A]`, something which was not permitted on the unexpanded version. Another generalization this permits is one that allows putting "most of" an `R'`-algebra structure on `A` when `R'` is only a monoid, which comes up for instance when `R' = units R`. In this case, we replace `[comm_semiring R] [semiring A] [module R A]` with `[monoid R] [semiring A] [distrib_mul_action R A]`. This generalization was used when proving intermediate results needed for Sylvester's law of inertia [6].

5. Diamonds

Frequently, there are multiple ways for Lean to construct a typeclass. For instance, consider the problem:

```
example {ι A B} [add_comm_monoid A] [add_comm_monoid B] : module ℕ (ι → A →+ B) :=
by apply_instance
```

Depending on the order of the search, Lean could take any of the paths in fig. 2. While Lean does not care about the existance of multiple paths and will happily just pick one, for the typeclass to be useful we need it to be predictable to the user; all they see is a `•` in the goal state. This means that whenever we have a diamond, we want all the paths to produce the same `•` such that the actual path taken does not matter.

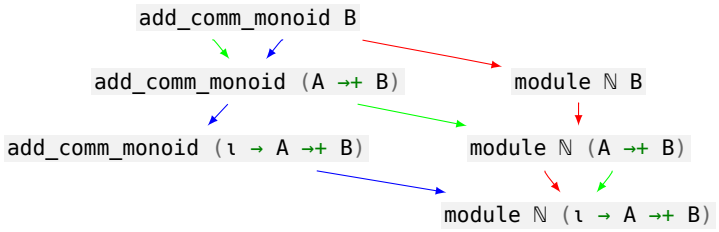

Figure 2: Diamonds in typeclass search. Arrows indicate implications, showing the three possible paths to resolve `module ℕ (A →+ B)`.

There are two relevant notions of "same" here. The first is propositional equality, which for the case in fig. 2 can be stated as roughly `∀ (M : Type*) [add_comm_monoid M], subsingleton (module ℕ M)` (that is that all `ℕ`-module structures are equal), and is straightforward to prove. This is enough to convince the user that the goal they're looking at is mathematically the one they're interested in. The second is definitional equality, which is needed by Lean in order to allow lemmas about one path for the tree in fig. 2 to apply for lemmas about another path.

In older versions of mathlib, the diamonds in fig. 2 resulted in paths creating instances that were propositionally equal, but not definitionally equal. This was problematic, as lemmas about the natural `ℕ`-action (blue path, fig. 2) such as `∑ x in s, c = s.card • c` would fail to match goals containing a derived `ℕ`-action (green and red paths, fig. 2). This was fixed in [7] by requiring the definition of `add_comm_monoid M` to include an implementation of the `ℕ`-module structure. While mathematically it is bizarre to say "a commutative additive monoid has a zero, addition, *and a scalar-multiplication by naturals*, such that ...", in Lean this is crucial to allow manual control of definitional equality such that the green and blue paths in fig. 2 can be made definitionally equal to the red path. This is analogous to the situation described in [1, section 4.1] for topologies associated with metric spaces.

6. Future work

6.1. Right actions

The scalar action typeclass in mathlib is intended for left-actions, which is apparent both in the definition of the `mul_smul` axiom, and in the order in which the arguments appear in the notation. However this does not mean that right actions are impossible.

The author has introduced preliminary support for right actions in [8], via the instance

```
instance monoid.to_opposite_mul_action [monoid α] : mul_action (opposite α) α :=
{ smul := λ c x, x * c.unop,
  one_smul := mul_one,
  mul_smul := λ x y r, (mul_assoc _ _ _).symm }

lemma op_smul_eq_mul [monoid α] {a b : α} : op a • b = b * a := rfl
```

Here, `opposite α` is a mathlib type built similarly to the pedagogical example in section 1.1 which reverses the multiplication order. This permits us to write `op a • b` as a messy spelling for a right action on `b` by `a`. Similar instances were introduced for the other stronger typeclasses in fig. 1. With these instances in place, it is possible to express an `R-S`-bimodule structure over `M` as

```
variables [module R M] [module (opposite S) M] [smul_comm_class R (opposite S) M]
```

Future work in this area could go on to define a `subbimodule R S M`, and use this to define a `two_sided_ideal R = subbimodule R R R` over a non-commutative ring `R`.

Unfortunately, mathlib does not have typeclasses for another common interaction of right actions with left actions. Introducing briefly for clarity the notation `a •> b` for `a • b` and `a <• b` for `op a • b`, there is no typeclass capable of expressing `(a <• b) •> c = a •> (b •> c)`. Some examples of when this situation arises are `[monoid M] (a b c : M)` (all three variables belong to the same non-commutative monoid), `[monoid M] (a c : M) (S : submonoid M) (b : S)` (the second belongs to a submonoid of the monoid containing the other two), and `[monoid M] (a b : M) (c : ι → M)` (the third variable is a coordinate vector).

The future of right actions in mathlib might be improved by the eventual switch to Lean 4 [9] (or the backport of design decisions made there), which provides a new `HMul A B C` typeclass which makes the `*` operator operate on fully heterogenous types. One possible design choice would be to eliminate the `•` operator entirely and use `*` for both the left and right actions, which would make the expression above expressible as a hypothetical heterogenous semigroup axiom that would subsume `is_scalar_tower`.

6.2. Further diamond definitional alignment

While [7] fixes `ℕ`-module diamonds (and a follow-up contribution fixes `ℤ`-module diamonds), these problem still exist for `ℕ`-, `ℤ`-, and `ℚ`-algebras. This could likely be resolved by adding the new data fields `of_nat`, `of_int`, and `of_rat` to `semiring`, `ring`, and `division_ring` respectively, along with corresponding proof fields showing these satisfy suitable constraints.

Acknowledgments

The author would like to thank the mathlib community for providing a continuous stream of high-quality contributions, which upon review by the author drew their attention to missing scalar action instances and generalizations elsewhere in mathlib. Frequently, these reviews also exposed the author to new areas of mathematics!

The author is supported by a scholarship from the Cambridge Trust.

References

[1] The mathlib Community, The lean mathematical library, Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (2020). 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: International Conference on Automated Deduction, Springer, 2015, pp. 378–388. doi:10.1007/978-3-319-21401-6_26.

[3] A. Baanen, S. R. Dahmen, A. Narayanan, F. A. E. Nuccio, A formalization of dedekind domains and class groups of global fields, 2021. arXiv:2102.02600.

[4] Y. G. Kudryashov, leanprover-community/mathlib#4770: introduce `smul_comm_class`, GitHub, 2020. URL: https://github.com/leanprover-community/mathlib/pull/4770.

[5] O. Nash, leanprover-community/mathlib#7932: adjointness for the functor `G` ↦ `monoid_algebra k G` when `G` carries only `has_mul`, GitHub, 2021. URL: https://github.com/leanprover-community/mathlib/pull/7932.

[6] K. Ying, leanprover-community/mathlib#7416: complex version of Sylvester's law of inertia, GitHub, 2021. URL: https://github.com/leanprover-community/mathlib/pull/7416.

[7] S. Gouëzel, leanprover-community/mathlib#7084: kill nat multiplication diamonds, GitHub, 2021. URL: https://github.com/leanprover-community/mathlib/pull/7084.

[8] E. Wieser, leanprover-community/mathlib#7630: add `has_scalar (opposite α) α` instances, GitHub, 2021. URL: https://github.com/leanprover-community/mathlib/pull/7630.

[9] L. d. Moura, S. Ullrich, The lean 4 theorem prover and programming language, in: A. Platzer, G. Sutcliffe (Eds.), Automated Deduction – CADE 28, Springer International Publishing, Cham, 2021, pp. 625–635.