Graded rings in Lean's dependent type theory

Eric Wieser[1][0000−0003−0412−4978] and Jujian Zhang[2][0000−0001−7340−2703]

[1] Cambridge University Engineering Department, efw27@cam.ac.uk
[2] Imperial College London, jujian.zhang19@imperial.ac.uk

Abstract. In principle, dependent type theory should provide an ideal foundation for formalizing graded rings, where each grade can be of a different type. However, the power of these foundations leaves a plethora of choices for how to proceed with such a formalization. This paper explores various different approaches to how formalization could proceed, and then demonstrates precisely how the authors formalized graded algebras in Lean's mathlib. Notably, we show how this formalization was used as an API; allowing us to formalize various graded structures such as those on tuples, free monoids, tensor algebras, and Clifford algebras.

Keywords: Graded rings · Dependent types · Formalization · mathlib

1 Introduction

One way to introduce graded rings and algebras is by noting that they generalize an early staple in mathematics education; that of single-variate polynomials in X. Any polynomial can be written as a (finite) weighted sum of powers of X, and multiplication only requires the knowledge that $X^m X^n = X^{m+n}$.

If we define the $\mathbb{N}$-indexed family of homogeneous polynomials $A = (i \mapsto \{aX^i \mid a : R\})$, then we can say "the polynomials in a ring R over X, $R[X]$ are an algebra graded by A"[3]; by which we mean:

1. Each of the elements of the family A_i are closed under addition and scalar multiplication by elements of R.
2. There is a $1 \in A_0$.
3. For any $p \in A_i$ and $q \in A_j$, we have $pq \in A_{i+j}$. Equivalently, as sets $A_i A_j \subseteq A_{i+j}$.
4. Every element p can be expressed uniquely as $p = \sum_i p_i$ where $p_i \in A_i$.

The above acts as a general definition of an algebra graded by some arbitrary family of submodules A, which can in general be indexed by any additive monoid ι, not just the natural numbers.

To build some intuition for this generalization, it is worth enumerating some other examples:

Multivariate polynomials, $R[X, Y, \dots]$. Over two variables we can grade either by the $\mathbb{N}$-indexed family of elements of homogeneous degree $A = (i \mapsto$

[3] Or "a graded algebra of type $\mathbb{N}$ over the ring R with graduation A" in the language of [6, III, §3, 1.].

$\{aX^jY^{i-j} \mid a : R, j \le i\}$) where X^3 and XY^2 e.t.c. have the same grade, or by a $\mathbb{N} \oplus \mathbb{N}$-indexed family on the individual variables, $A = ((i, j) \mapsto \{aX^iY^j \mid a : R\})$.

The tensor algebra, $\mathcal{T}(V)$. Conventionally we grade this by the $\mathbb{N}$-indexed family where A_i spans the i^{th} tensor powers $V^{\otimes i}$.

The exterior algebra, $\bigwedge(V)$. The exterior algebra is graded in exactly the same way, but when V is of dimension n we find that A_i for $n < i$ is the trivial submodule.

The Clifford algebra, $\mathcal{C}\ell(V, Q)$[4]. We cannot[5] use exactly the same approach for the Clifford algebra, as for a vector v, we have $v^2 = Q(v)$, where the LHS would be of grade 2 and the RHS would be of grade 0. This can be resolved by having just two grades; one corresponding to sums of "even" monomials (those which are a product of an even number of elements of V), and one corresponding to sums of odd monomials. Phrased another way, the family is indexed by[6] $\mathbb{Z}/2\mathbb{Z}$, the integers modulo two.

Any ring α. Any ring can be equipped with the trivial grading structure, where the index type contains only one element 0 corresponding to the entire ring.

As this is a paper about formalization, we will predictably proceed by finding all the different ways to "pull legs off"[7] this definition. By relaxing items 2 and 3, we can talk about about gradings of additive monoids, additive groups ([6, II, §11, 1.]), and R-modules[8]. By relaxing item 1, we can additionally talk about gradings by families of additive subgroups, additive submonoids, or even just sets; which we refer to as graded rings, graded semirings, and graded monoids respectively. For graded monoids (such as the n-tuples α^n or tensor powers $M^{\otimes n}$) there is no summation, so item 4 is interpreted as the statement that p must belong to exactly one A_i. Figure 1 outlines the connection between these various generalizations.

While the existence of many examples following the same pattern is already a good reason to formalize that pattern, it is only half the picture; just as important is to have situations where the generalization itself is necessary. For instance, without a formalization of commutative additive monoids, we can't even define what it means to take the sum of a finite set, and would instead be forced to repeat this definition for $\mathbb{N}$, $\mathbb{Z}$, etc. A particularly motivating example for need a formalization of graded rings is that of the $\operatorname{Proj} S$ construction in algebraic geometry[12, Tag 01M3], a definition which requires a notion of homogeneous ideals, which in turn requires precisely the notion of graded rings

[4] Where $\mathcal{C}\ell(V, Q)$ is notation to specify the quadratic form Q and vector space V.

[5] At least, when $Q \neq 0$. If $Q = 0$ then $\mathcal{C}\ell(V, Q) = \bigwedge(V)$ and we can proceed as above.

[6] Note that literature referring to an $\mathbb{N}$-grading is referring to the grading on $\bigwedge(V)$ via the canonical module equivalence.

[7] `https://en.wikipedia.org/wiki/Centipede_mathematics`

[8] Some sources use a more general definition of graded R-modules and R-algebras, where R is itself a graded ring such that $R_iM_j \subseteq M_{i+j}$. For brevity we will not discuss these here (in essence considering only the special case when R has the trivial graduation), but our approach would extends to this straightforwardly

this paper is about. Another recent motivation appeared in the "Liquid Tensor Experiment"[8], which required a proof of Gordan's lemma; one proof of which goes via graded rings[9].

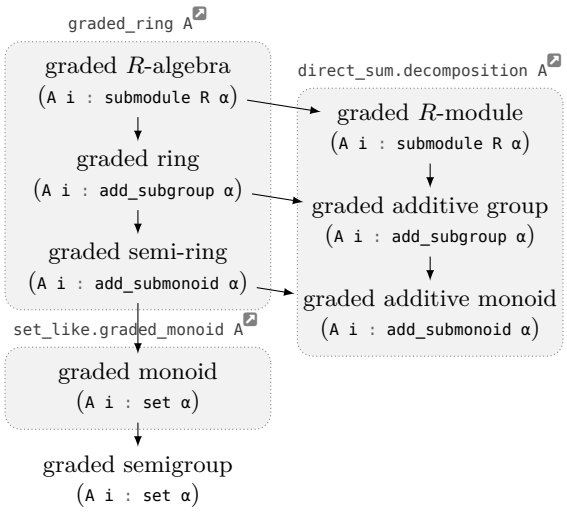

Fig. 1. The algebraic hierarchy of graded objects discussed in this paper. The meanings of the typeclasses introduced in section 3 for grading `α` internally by `A` are shown as labelled gray regions, with the objects representing each internal grade (that is, the type of A_i) shown in parentheses.

1.1 Prior formalizations

In Coq, [9, 3.] refers to graded modules as `nat -> FreeModule R` in the context of homology. No mention of graded multiplicative structures appears. A similar formalization of graded modules exists for Lean 2 in [2, `algebra/graded.hlean`], although the language has changed substantially since then, in particular dropping the experimental HoTT mode which [2] builds upon.

In Agda, [7] shows that the axioms of a commutative graded ring are satisfied by a particular construction, the graded cohomology groups. No attempt seems to be made to provide a general definition of what it means for an object to satisfy those axioms.

Extensive discussion about formalizing "Commutative Differential Graded algebras", algebraic objects with some additional axioms on top of the graded algebras discussed in this paper, has taken place on the Lean Zulip Chat [5]. While these discussions refer to a current version of Lean 3 [10], the ideas explored in [5] never resolved into a contribution to Lean's monolithic mathematics

[9] In the absence of our work a proof via convexity was used instead.

library mathlib [11]. It is by this metric that [9,2] are substantially different in scope to this work; in those formalizations, a definition was chosen for the particular use case of interest to the authors, with little regard for interoperability with large amounts of existing code. Conversely, one of the reasons the work in [5] never made it into mathlib is likely the lack of applications to verify that the design is the right one. Since the closure of that thread, mathlib has grown by a factor of five in terms of total lines, and has gained formalizations of many new objects of interest to us: tensor algebras, Clifford algebras, and tensors powers.

1.2 Additive vs multiplicative grading

Note that some work uses the convention that multiplication in item 3 behaves as $A_i \to A_j \to A_{ij}$ instead of $A_i \to A_j \to A_{i+j}$. For simplicity, this paper and mathlib only develop the additive version, as this has more pre-existing applications in mathlib. Using `additive ι` as the index type where `ι` is a multiplicative monoid allows the former to be expressed in the language of the latter, so if we were to have both versions it would only be for convenience.

2 External gradings

There are two ways to think about a grading in dependent type theory; either as a family of sets of a single type (internal), or as an indexed family of distinct types. There are merits to both approaches; which is most useful depends on whether it is more natural to define the single type then break it into pieces (as with the monoid under concatenation of `list`s graded by their `length`), or to define the family of types then glue them together (as with the monoid under concatenation of the tuples `fin n → α` graded by `n`). A crucial factor in the coherence of mathlib as a unified library is its ability to translate between multiple ways of stating the same thing, so we do not want to have to choose between these approaches in an exclusive manner. Thankfully, the former approach can be represented via the latter; an internal grading can be written as an external grading over the family of subtypes corresponding to each grade, shown in parentheses in fig. 1. We will revisit this equivalence in section 3.

It is worth remembering that when building externally-graded objects in this way, that the grades are disjoint by definition. If for example our indexed family of types is `λ i : ℕ, R` (a family indexed by the naturals, all equal to the same ring), then this is viewed as a countable sequence of *copies* of `R`, which makes this construction exactly analogous to the single-variate polynomials.

2.1 Graded semigroups

Let us now try to develop the framework for talking about externally-graded semigroups[10] over a family of types `A` indexed by an additive semigroup `ι`. We

[10] Chosen for brevity due to having the fewest axioms, not because they are interesting.

would like to be able to express these via Lean's "typeclasses", as this matches how the usual non-graded algebraic structure is expressed. This means that to talk about a graded monoid, a user might write:

```
def sq {ι : Type*} {A : ι → Type*} [add_semigroup ι] [g_semigroup A]
  (i : ι) (x : A i) : A (i + i) :=
g_semigroup.mul x x
```

To explain this syntax briefly; `sq` names the definition, `{name : type}` and `(name : type)` introduce implicit and explicit variables, `[type]` introduces a typeclass variable, the trailing `:` prefixes the result type, and `:=` prefixes the value of the definition. Typeclass variables are special; the `[add_semigroup ι]` variable is used to define the meaning of `i + i` via mathlib's algebra framework, while the `g_semigroup.mul` would be defined by the `[g_semigroup A]` variable. A user calling this `sq` function might write `sq _ x`, where `_` acts as a wildcard which Lean works out automatically by looking at `x`. The same mechanism is used to infer the implicit `ι` and `A` arguments, but typeclass search is used to populate the two arguments in square brackets; for instance, if `ι := ℕ` then Lean finds `nat.add_semigroup : add_semigroup ℕ`. More thorough introductions to typeclasses in Lean and mathlib can be found in [4, §2] and [13, §1.1].

Attempting to define a new `g_semigroup A` typeclass by directly writing down item 3 and a suitable associativity axiom as

```
variables {ι : Type*} (A : ι → Type*)

class g_semigroup [add_semigroup ι] :=
(mul {i j} : A i → A j → A (i + j))
(mul_assoc {i j k} (x : A i) (y : A j) (z : A k) :
    mul (mul x y) z = mul x (mul y z))
```

leads to the error "term `mul x (mul y z)` has type `A (i + (j + k))` but is expected to have type `A ((i + j) + k)`". While these are "obviously" equal, that's not enough for Lean; for the statement to type-check, we need the types to be *definitionally* equal. We would have similar problems with `A (i + 0)` and `A i` if we were trying to prove `mul x one = x` for a graded monoid. To escape this problem, we could:

1. Use heterogenous equality (denoted `==`), which allows us to express equality between distinct types:

   ```
   class g_semigroup [add_semigroup ι] :=
   (mul {i j : ι} : A i → A j → A (i + j))
   (mul_assoc {i j k : ι} (x : A i) (y : A j) (z : A k) :
     mul (mul x y) z == mul x (mul y z))
   ```

2. Express the equality in terms of sigma types or dependent pairs, denoted `Σ i, A i`:

   ```
   class g_semigroup [add_semigroup ι] :=
   (mul {i j : ι} : A i → A j → A (i + j))
   (mul_assoc {i j k : ι} (x : A i) (y : A j) (z : A k) :
     ⟨_, mul (mul x y) z⟩ : Σ i, A i = ⟨_, mul x (mul y z)⟩)
   ```

3. Express the grading constraint as an equality on sigma types:

```
class g_semigroup [add_semigroup ι] extends semigroup (Σ i, A i) :=
(fst_mul {i j : ι} (x : A i) (y : A j) :
  ((_, x) * (_, y) : Σ i, A i).fst = i + j)
```

4. Provide an explicit proof that the equality is type correct using the recursor for equality, `eq.rec`:

```
class g_semigroup [add_semigroup ι] :=
(mul {i j : ι} : A i → A j → A (i + j))
(mul_assoc {i j k : ι} (x : A i) (y : A j) (z : A k) :
  (add_assoc i j k).rec (mul (mul x y) z) = mul x (mul y z))
```

5. Store a canonical map between objects of the "same" grade to use instead of using `eq.rec`, to allow better definitional control:

```
class g_semigroup [add_semigroup ι] :=
(cast {i j : ι} (h : i = j) : A i → A j)
(cast_rfl {i} (x : A i) : cast rfl x = x)
(mul {i j : ι} : A i → A j → A (i + j))
(mul_assoc {i j k : ι} (x : A i) (y : A j) (z : A k) :
  cast (add_assoc i j k) (mul (mul x y) z) = mul x (mul y z))
```

6. Take an additional index into `mul` and a proof that it is equal to `i + j`:

```
class g_semigroup [add_semigroup ι] :=
(mul {i j k : ι} (h : i + j = k) : A i → A j → A k)
(mul_assoc {i j k ij jk ijk : ι}
  (hij : i + j = ij) (hjk : j + k = jk)
  (hi_jk : i + jk = ijk) (hij_k : ij + k = ijk)
  (x : A i) (y : A j) (z : A k) :
    (mul hij_k (mul hij x y) z) = mul hi_jk x (mul hjk y z))
```

Many of these options are derived from the discussions in [5]. When deciding between these options, we need to consider both the ease of providing an instance with `instance : g_semigroup A`, and the ease of consumer working with one using `[g_semigroup A]`. Note that it is straightforward to expose different interfaces to the consumer and producer, and provide a layer of translation in between. This is especially the case when the interfaces differ only in their statement of the propositional fields. Table 1 outlines a rough comparison between these approaches.

Taking a step back from this problem, we also need to decide on a spelling for the consumer, as writing `mul` instead of a multiplication symbol is hardly pleasant. There are essentially two options here: either introduce new notation for our graded `mul`, or hook into the existing `*` notation. The latter is a far more appealing option, as it means we can reuse all the lemmas we have about `*` by providing the appropriate algebraic typeclasses. The only catch is that the existing `*` notation requires the operation to be homogeneous; acting on a single type, rather than three elements of a family[11].

[11] Lean 4 lifts this notation restriction, but the algebraic typeclasses provided by mathlib would need reworking.

Table 1. Merits of the various approaches to defining `g_semigroup`. "Producer" refers to the code providing the `instance : g_semigroup A`, while "consumer" refers to the code with a `[g_semigroup A]` argument.

Approach	item 1 `==`	item 2 `Σ i, A i`	item 3 `extends`	item 4 `eq.rec`	item 5 `cast`	item 6 `h : i+j=k`
`h : (i+j)+k=i+(j+k)` needed by	producer	producer	—	—	—	consumer
Estimated difficulty for the producer	medium	harder	easier	harder	easier	medium
Directions of consumer `rw` tactic use	0	2	2	1	1	2

To achieve this homogenization, we can use the builtin `sigma` type of dependent pairs, storing the grade of the monoid alongside the value at that grade such that `x : A i` is represented by `sigma.mk i x : Σ i, A i`.

```
instance g_semigroup.to_semigroup [add_semigroup ι] [g_semigroup A] :
  semigroup (Σ i, A i) :=
{ mul := λ (x y : Σ i, A i), ⟨x.fst + y.fst, g_semigroup.mul x.snd y.snd⟩,
  mul_assoc := λ (x y : Σ i, A i), sorry }
```

If we choose item 3 from table 1, then this code is generated for us automatically! However, the fact that the grade of the multiplication is known only propositionally and not definitionally can make things harder in section 2.3 so we avoid this choice. As the next most appealing option, choosing item 2 makes the `sorry` above[12] fall out immediately, and so this is what mathlib does. This decision is far from final, but the best way to compare the option of table 1 is to thoroughly implement one of them, and then come back and see whether changing the definition to something different makes the existing proofs better or worse.

2.2 Graded monoids

In reality, we do not define `gsemigroup` at all in mathlib, and jump straight to graded monoids due to lack of need for the former. We also don't actually put the instance on the `sigma` type, as this would not be a sufficiently canonical choice to be worthy of a global instance. Instead, we define `graded_monoid A` as an alias for `sigma A`, and place the instances on that. By splitting apart the typeclasses a little and interleaving the instances for `graded_monoid A`:

```
class ghas_one [has_zero ι] := (one : A 0)
class ghas_mul [has_add ι] := (mul {i j} : A i → A j → A (i + j))

instance ghas_one.to_has_one [has_zero ι] [ghas_one A] :
  has_one (graded_monoid A) := { one := ⟨_, ghas_one.one⟩ }
instance ghas_mul.to_has_mul [has_add ι] [ghas_mul A] :
  has_mul (graded_monoid A) := { mul := λ x y, ⟨_, ghas_mul.mul x.snd y.snd⟩ }
```

[12] The syntax in Lean for an incomplete proof

we make the notation for the instance much more pleasant

```
class gmonoid [add_semigroup ι] extends ghas_one A, ghas_mul A :=
(one_mul (a : graded_monoid A) : 1 * a = a)
(mul_one (a : graded_monoid A) : a * 1 = a)
(mul_assoc (a b c : graded_monoid A) : a * b * c = a * (b * c))
```

It is worth remembering that while this may look identical to the definition of a regular monoid, it is constraining the grade-preserving behavior of the multiplication by construction. As is always the case with formalization, it is never quite as simple as you would hope it would be. In fact, the definition of `gmonoid` in mathlib contains three additional fields!

```
(gnpow : Π (n : ℕ) {i}, A i → A (n • i))
(gnpow_zero' : Π (a : graded_monoid A), graded_monoid.mk _ (gnpow 0 a.snd) = 1)
(gnpow_succ' : Π (n : ℕ) (a : graded_monoid A),
  (graded_monoid.mk _ $ gnpow n.succ a.snd) = a * ⟨_, gnpow n a.snd⟩)
```

These describe the power operator by the natural numbers, and ensure that its grade too is known definitionally following the "forgetful inheritance"[1] pattern, the relevance of which is explored in [13, §5].

Example 1 (the n-tuples). This typeclass allows us to express the graded monoid structure of the n-tuples under concatenation, as

```
instance : gmonoid (λ n : ℕ, fin n → α) :=
{ one := ![], -- the empty tuple
  mul := λ i j a b, fin.add_cases a b,
  ..sorry /- boring proofs -/ }
```

2.3 Graded (semi)rings

For graded rings, we do not run into any new equality problems, as addition remains within a grade. To state the requirement for a family of types to represent a graded ring, we can simply extend the monoid structure from earlier:

```
class gsemiring [add_monoid ι] [Π i, add_comm_monoid (A i)]
  extends gmonoid A :=
(mul_zero : ∀ {i j} (a : A i), mul a (0 : A j) = 0)
(zero_mul : ∀ {i j} (b : A j), mul (0 : A i) b = 0)
(mul_add : ∀ {i j} (a : A i) (b c : A j), mul a (b + c) = mul a b + mul a c)
(add_mul : ∀ {i j} (a b : A i) (c : A j), mul (a + b) c = mul a c + mul b c)
-- For "forgetful inheritance" like the previous `gnpow` field
(nat_cast : ℕ → A 0) (nat_cast_zero : sorry) (nat_cast_succ : sorry)
```

We *almost* do not need any axioms about negation; to work with a graded ring as opposed to a graded semiring, the user could write `[Π i, add_comm_group (A i)] [gsemiring A]`. Unfortunately, our hand is forced by "forgetful inheritance" to define `gring` anyway in order to add an `int_cast` operation and associated axioms.

Just as we used `graded_monoid A` in section 2.1 to bundle our graded monoid with its grade to enable reuse of the `monoid` API, we'd like to be able to enable reuse of the `semiring` API on graded semirings. We cannot use `graded_monoid A` here,

as in a graded ring an element can consist of distinct grades added together; instead we use `direct_sum ι A`, with notation `⨁ i, A i`. Here, the element `x : A i` is represented by `direct_sum.of A i x`. This comes with all the additive structure we need already; all we have to do is extend our multiplicative structure onto it linearly, with our end goal being to produce a `semiring (⨁ i, A i)` instance.

To do this, we first promote our `mul` to a bundled homomorphism[11, §4.1.2] that is additive in each argument

```
def gmul_hom [gsemiring R] {i j} : A i →+ A j →+ A (i + j) :=
{ to_fun := λ a,
  { to_fun := λ b, gsemiring.mul a b,
    map_zero' := gsemiring.mul_zero _,
    map_add' := gsemiring.mul_add _ },
  map_zero' := add_monoid_hom.ext $ λ a, gsemiring.zero_mul a,
  map_add' := λ a₁ a₂, add_monoid_hom.ext $ λ b, gsemiring.add_mul _ _ _}
```

as this allows us to lift this map to consume and produce elements of the direct sum as:

```
def mul_hom : (⨁ i, A i) →+ (⨁ i, A i) →+ ⨁ i, A i :=
direct_sum.to_add_monoid $ λ i,
  add_monoid_hom.flip $ direct_sum.to_add_monoid $ λ j, add_monoid_hom.flip $
    (direct_sum.of A _).comp_hom.comp $ gmul_hom A
```

Unfortunately working with bundled maps in mathlib forces you to write things in the rather unreadable point-free style as above. The benefit of working in a theorem prover is that we can at least verify that something unreadable still behaves as we want:

```
lemma mul_hom_of_of {i j} (a : A i) (b : A j) :
  mul_hom A (of _ i a) (of _ j b) = of _ (i + j) (gsemiring.mul a b) := sorry
```

It might feel like we're on the home stretch to providing the semiring instance; indeed, the proofs relating multiplication and the additive structure follow trivially from `mul_hom`. Unfortunately, we're now faced with actually using the API we built in section 2.2 to prove the multiplicative properties! The key result we need to do this is that our two notions of equality are equivalent; that is,

```
lemma of_eq_of_graded_monoid_eq {i j : ι} {a : A i} {b : A j}
  (h : graded_monoid.mk i a = graded_monoid.mk j b) :
  direct_sum.of A i a = direct_sum.of A j b := sorry
```

After once again fighting against point-free nonsense to turn associativity into equality of two tri-additive maps, we can use the `ext` tactic to reduce our problem to associativity of three terms of the form `of A i x`:

```
private lemma mul_assoc (a b c : ⨁ i, A i) : a * b * c = a * (b * c) :=
-- `λ a b c, a * b * c = λ a b c, a * (b * c)` as bundled homomorphisms
suffices (mul_hom A).comp_hom.comp (mul_hom A)
       = (add_monoid_hom.comp_hom flip_hom $
           (mul_hom A).flip.comp_hom.comp (mul_hom A)).flip,
  from fun_like.congr_fun (fun_like.congr_fun (fun_like.congr_fun this a) b) c,
begin
  ext ai ax bi bx ci cx : 6,
```

```
  show mul_hom A (mul_hom A (of A ai ax) (of A bi bx)) (of A ci cx) =
       mul_hom A (of A ai ax) (mul_hom A (of A bi bx) (of A ci cx)),
```

from which the rest follows using our previous results:

```
  rw [mul_hom_of_of, mul_hom_of_of, mul_hom_of_of, mul_hom_of_of],
  exact of_eq_of_graded_monoid_eq
    (mul_assoc (graded_monoid.mk ai ax) ⟨bi, bx⟩ ⟨ci, cx⟩),
end
```

A similar approach can be used to prove the `mul_one` and `one_mul` fields in order to finish the construction of `semiring (⨁ i, A i)`. The "point-free nonsense" approach is not the only path available to us; but the alternative of using `induction` creates annoying side-goals to prove that the map is additive.

There is another important construction we will want for working with this direct sum representation of a graded ring; a way to construct a ring homomorphism out of the graded ring given a suitable family of homomorphisms on each piece. This can be written as:

```
def direct_sum.to_semiring
  (f : Π {i}, A i →+ R)
  (hone : f gsemiring.one = 1)
  (hmul : ∀ {i j} (ai : A i) (aj : A j), f (gsemiring.mul ai aj) = f ai * f aj) :
  (⨁ i, A i) →+* R :=
{ map_one' := sorry, map_mul' := sorry, .. direct_sum.to_add_monoid f }
```

We will find this instrumental for relating external and internal direct sums in section 3.4.

Example 2 (the n^{th} tensor powers). This typeclass allows us to express the graded ring structure of the n^{th} tensor power over an R-module V as

```
instance : gsemiring (λ n : ℕ, ⨂[R]^n V) := sorry
```

We can then show with the aid of `direct_sum.to_semiring` that $\mathcal{T}(V)$ is isomorphic as a ring (and algebra) to $\bigoplus_n V^{\otimes n}$; that is `tensor_algebra R V ≃ₐ[R] (⨁ n, ⨂[R]^n V)`.

3 Internal gradings

3.1 Decompositions of sets

For an external decomposition with no algebraic structure, the task is simple; a unique decomposition of a type `α` into its pieces `A : ι → Type*` can be spelled as the equivalence to a sigma type, `decompose : α ≃ Σ i : ι, A i`. For an internal decomposition where `A : ι → set α`, we have some other options. If we don't care about a constructive decomposition and are happy with a classical one, we can just state that the components span the entire type and are disjoint, as:

```
(⋃ i, A i) = set.univ ∧ pairwise (disjoint on A)
```

If we do care about constructiveness, we can instead have a function `grade : α → ι` that respects `∀ (a : α) (i : ι), a ∈ A i ↔ grade a = i`. Whichever approach we

pick, it is straightforward to recover the externally-graded viewpoint via the map `decompose : α ≃ Σ i : ι, ↥(A i)`. Here, `↥` is the operator that lets us view a set as a subtype of the type of its elements.

3.2 Graded monoids

A preliminary attempt at formalizing an internal multiplicative grading structure might look like

```
class set.graded_monoid [monoid M] [add_monoid ι] (A : ι → set M) : Prop :=
(one_mem : 1 ∈ A 0)
(mul_mem : ∀ ⦃i j : ι⦄ ⦃gi gj : M⦄, gi ∈ A i → gj ∈ A j → gi * gj ∈ A (i + j))
```

This works fine for graded monoids; but for graded semirings, rings, and algebras we need to apply the additional constraints that each `A i` is closed under the appropriate operations.

To avoid having to write separate typeclasses for each case and ending up with our API in triplicate, we instead generalize over `A : ι → set M`; with the goal being able to talk about `A : ι → add_submonoid R`, `A : ι → add_subgroup R`, and `A : ι → submodule S R` in a unified way. As well as avoiding the need for three different typeclasses, this also means we can reuse all the theory we already have about `add_submonoid`, `add_subgroup`, and `submodule`. To do that, we introduce a `set_like` class, which was one of the inspirations for the `fun_like` generalization described in [4, §6.3]. This class lets us express that elements `s` of a type `S` has a canonical intepretation as a set `↑s : set α`, and is defined as:

```
class set_like (S : Type*) (α : out_param Type*) :=
(coe : S → set α)  -- the function that is used for `↑` coercion notation
(coe_injective' : function.injective coe)
```

This equips `s : S` with membership notation `a ∈ s : Prop` and a coercion to type `↥s : Type*`, and permits us to write

```
class set_like.graded_monoid {ι M S : Type*}
  [set_like S M] [monoid M] [add_monoid ι] (A : ι → S) : Prop :=
(one_mem : 1 ∈ A 0)
(mul_mem : ∀ ⦃i j : ι⦄ ⦃gi gj : M⦄, gi ∈ A i → gj ∈ A j → gi * gj ∈ A (i + j))
```

At this point, we can deliver on the earlier claim that the internal viewpoint can be expressed via the external viewpoint. To do this, we show that the family of subtypes `λ i, ↥(A i)` has a graded multiplicative structure, as:

```
-- this implies `monoid (graded_monoid (λ i, ↥(A i)))` via `gmonoid.to_monoid`
instance set_like.gmonoid [set_like S M] [monoid M] [add_monoid ι]
  (A : ι → S) [set_like.graded_monoid A] :
  gmonoid (λ i, ↥(A i)) :=
{ one := ⟨1, set_like.graded_monoid.one_mem⟩,
  mul := λ i j a b, ⟨(a * b : R), set_like.graded_monoid.mul_mem a.prop b.prop⟩,
  mul_assoc := λ ⟨i, a, ha⟩ ⟨j, b, hb⟩ ⟨k, c, hc⟩,
    sigma.subtype_ext (add_assoc _ _ _) (mul_assoc _ _ _),
  ..sorry  /- etc -/ }
```

Note here that we are paying the cost outlined in the first row of table 1 of having to reprove associativity of addition of the grades, which is a mark against our choice of item 2.

Example 3 (the free monoid over α). This typeclass allows us to express[13] the internal graded monoid structure of the free monoid, with elements graded by the number of generators

```
instance :
  set_like.graded_monoid
    (λ i : ℕ, (set.range (free_monoid.of : α → free_monoid α)) ^ i) :=
{ one_mem := by rw [pow_zero, set.mem_one],
  mul_mem := λ i j x y hx hy, by { rw pow_add, exact set.mul_mem_mul hx hy} }
```

We are not quite done yet; we have shown that the subtypes can be glued together to form an object with graded multiplication, but the `set_like.graded_monoid` typeclass above does nothing to ensure that the glued-together type `graded_monoid (λ i, ↥(A i))` is in bijection with `M`. We can reuse either the classical or constructive approach from section 3.1, but with our new `monoid (graded_monoid (λ i, ↥(A i)))` instance we can promote the equivalence `decompose : α ≃ Σ i : ι, ↥(A i)` to a multiplicative isomorphism, stated as `decompose : α ≃* graded_monoid (λ i, ↥(A i))`.

3.3 Decompositions of additive monoids and *R*-modules

For a unique decomposition with an additive structure, we cannot use the same approach as section 3.1 but instead need to decompose into a direct sum, as `decompose : α ≃+ ⨁ i : ι, ↥(A i)`. Let us consider the internal decomposition of an additive group `α` into the family `A : ι → add_subgroup α`. We need to be a little more careful when describing the disjointness condition, as what we actually require is that every component is disjoint (in the sense of having trivial intersection $\{0\}$) from the span of all the others. We can spell that as

```
(⨆ i, A i) = ⊤ ∧ complete_lattice.independent A
```

but for additive submonoids this condition, while necessary, is still not sufficient; consider when $A_+ = \{z : \mathbb{Z} \mid 0 \le z\}$ and $A_- = \{z : \mathbb{Z} \mid 0 \ge z\}$, which are disjoint and span all of $\mathbb{Z}$, but clearly do not permit a decomposition. As such, we cannot use this as our definition. Instead, we require that the canonical map from `⨁ i : ι, A i` to `α` (defined as roughly `λ x, ∑ i, ↑(x i)`) is bijective.

Once again we're on the precipice of stating things in triplicate, as we want to state this condition (and the consequences of it) for `add_monoid α` and `submodule R α` as well to cover the cases on the right of fig. 1. Until very recently, stating this condition in triplicate was exactly what mathlib did; but thanks to [3] which transfers the success in [4] from `fun_like` to `set_like`, we can now generalize over `add_subgroup α` as `S` where `(S : Type*) [set_like S α] [add_submonoid_class S α]`. This

[13] After enabling the appropriate by-default-disabled instances.

lets us define a single canonical map `coe_add_monoid_hom` that works for all three cases as

```
protected def coe_add_monoid_hom [add_comm_monoid α]
  [set_like S α] [add_submonoid_class S α] (A : ι → S) :
  (⨁ i, A i) →+ M :=
direct_sum.to_add_monoid (λ i, add_submonoid_class.subtype (A i))
```

which in turn allows us to state our condition just once to cover all three cases. We can state it either constructively, by carrying around an explicit inverse:

```
class decomposition (A : ι → S) :=
(decompose' : α → ⨁ i, A i)  -- split elements into their grades
(left_inv : function.left_inverse (coe_add_monoid_hom A) decompose')
(right_inv : function.right_inverse (coe_add_monoid_hom A) decompose')
```

or classically by simply proving bijectivity. We provide a proof in mathlib that on submodules over additive groups the `complete_lattice.independent` formulation is equivalent to these definitions.

3.4 Graded (semi)rings

To talk about a semiring with an internally grade-compatible multiplication, we thankfully need to define no further typeclasses; we can write

```
variables {ι R S : Type*} [add_monoid ι] [semiring R] [set_like S R]
variables [add_submonoid_class S R] (A : ι → S) [set_like.graded_monoid A]
```

where `set_like.graded_monoid` handles our conditions on the multiplicative structure, `add_submonoid_class` handles our conditions on the additive structure, and `semiring R` ensures the compatibility between the two. We'd like to end up with a `gsemiring (λ i, ↥(A i))` instance as a result of these hypotheses so as to also have a `semiring (⨁ i, ↥(A i))` instance, which we achieve as

```
instance set_like.gsemiring : direct_sum.gsemiring (λ i, ↥(A i)) :=
{ mul_zero := λ i j _, subtype.ext (mul_zero _),
  zero_mul := λ i j _, subtype.ext (zero_mul _),
  mul_add := λ i j _ _ _, subtype.ext (mul_add _ _ _),
  add_mul := λ i j _ _ _, subtype.ext (add_mul _ _ _),
  ..set_like.gmonoid A }
```

Once again, the introduction of `add_submonoid_class` excused us from needing three copies of this definition.

Now that we have this instance, we can build the canonical ring morphism from `⨁ i, ↥(A i)` to `R` that amounts to summing the elements from each grade (after passing them through the canonical additive morphism from the subtype), building upon the `direct_sum.to_semiring` definition at the end of section 2.3:

```
def direct_sum.coe_ring_hom [add_monoid ι] [semiring R] [set_like S R]
  [add_submonoid_class S R] (A : ι → S) [set_like.graded_monoid A] :
  (⨁ i, ↥(A i)) →+* R :=
direct_sum.to_semiring
  (λ i, add_submonoid_class.subtype (A i)) rfl (λ _ _ _ _, rfl)
```

The `direct_sum.coe_add_monoid_hom` we mention in section 3.3 has a definitionally equal underlying function to this, meaning we can reuse the `decomposition A` from that section defined in terms of the former to obtain the canonical ring isomorphism `decompose_ring_equiv : R ≃+* ⨁ i, ↥(A i)` between an internally-graded ring `R` and the direct sum of its grades `⨁ i, ↥(A i)`. For convenience, we provide a single typeclass that provides access to this operation:

```
class graded_ring (A : ι → σ) extends graded_monoid A, decomposition A.
```

Among other functions and lemmas which build on this convenience, we provide projection maps as additive maps `proj A i : R →+ R` so that any $x : R$ can be written as $x = \sum_i x_i$, with x_i being the i-th projection of x with respect to grade `A`, without having to explicitly go via the direct sum of subtypes `⨁ i, ↥(A i)`.

4 Graded R-algebras

This paper would not be complete without building external and internal graded R-algebras on top of the graded rings; indeed, we have added definitions of these to mathlib, as the typeclass `galgebra`, the instance `submodule.galgebra`, and the shorthand `graded_algebra`; but the process of defining these presented no new challenges over those already faced when defining graded rings. Just as we were able to recover a ring isomorphism in section 3.4, these definitions let us recover the analogous R-algebra isomorphism `decompose_alg_equiv : X ≃ₐ[R] ⨁ i, ↥(A i)`.

Example 4 (the multivariate polynomials). Returning to the examples in section 1, we can define the homogeneous graduation:

```
instance : graded_algebra (λ i : ℕ, homogeneous_submodule σ R i) := sorry
```

where `σ` represents the variables in the multivariate polynomial ring and `homogeneous_submodule σ R i` the homogeneous polynomials of degree i.

Example 5 (the tensor algebra $\mathcal{T}(V)$). Given `ι R` as the canonical map $V \to \mathcal{T}(V)$, we write the typical internally grading as:

```
instance : graded_algebra
  ((^) (ι R : M →ₗ[R] tensor_algebra R M).range : ℕ → submodule R _) := sorry
```

Example 6 (the Clifford algebra $\mathcal{C}\ell(V, Q)$). Similarly, we can write[14]

```
def even_odd (i : zmod 2) : submodule R (clifford_algebra Q) :=
⨆ (j : {n : ℕ // ↑n = i}), (ι Q).range ^ (j : ℕ)
instance : graded_algebra (clifford_algebra.even_odd Q) := sorry
```

where `ι Q` is a similar canonical map, and `even_odd 0` and `even_odd 1` are the even and odd submodules respectively.

[14] Thus resolving the further work in [14, §8.1].

5 Conclusions

This paper outlined the substantial amount of busy-work required to define the various types of graded structure in mathlib, and the various design choices made along the way. Instead of choosing between internal and external grading, we opted to use the latter to implement the former. When it comes to classical vs constructive decompositions, we opted to have both, just like mathlib has classical vs constructive finite sets. Finally, for stating equality between non-definitionally-equal grades, we opted to use sigma types.

To verify that our design decisions are sensible, we demonstrated a variety of results stated using our new typeclasses. While not discussed in detail in this paper, our supplemental repository includes an extended example by the second author of a preliminary development of the Proj construction in algebraic geometry, which using the result from example 4 culminates in defining the projective n-space. The success of this formalization indicates that the design of graded objects is coherent with other theories in mathlib.

A snapshot of the unabridged code from this work, permalinked throughout via "", is available at `https://github.com/eric-wieser/lean-graded-rings`. It comprises around 1150 sloc[15] of API development, 1250 sloc of applications, and 5300 sloc of the extended Proj example. Permalinks resembling "" refer to declarations in mathlib itself that were infeasible to extract into our isolated repository. Much of this code has already been integrated into mathlib over the course of over 30 pull requests[16].

The monolithic nature of mathlib development means that our design decisions are easy to revisit at a later date, as the assumption is already that code written against one version of mathlib is not guaranteed to work without modification on a later version. Meanwhile, the act of having made the decisions enables downstream work to progress; as well as unblocking diverging formalization projects by the two authors[17], our foundations have allowed mathlib to gain formalizations by other contributors about various internal decompositions in inner product spaces and torsion modules.

Acknowledgments The authors would like to thank Kevin Buzzard for his justified insistence on needing an interface to talk about internal gradings, Anne Baanen for picking up the mantle dropped by the first author on the rest of the `set_like` refactor, and the rest of the mathlib community for the extraordinarily collaborative work providing the context for this paper. The first author is funded by a scholarship from the Cambridge Trust. The second author is funded by the Schrödinger Scholarship Scheme from Imperial College London.

[15] source lines of code
[16] Tracked in `https://github.com/leanprover-community/mathlib/projects/12`
[17] In geometric algebra and algebraic geometry, respectively!

References

1. Affeldt, R., Cohen, C., Kerjean, M., Mahboubi, A., Rouhling, D., Sakaguchi, K.: Competing Inheritance Paths in Dependent Type Theory: A Case Study in Functional Analysis. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) Automated Reasoning, vol. 12167, pp. 3–20. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-51054-1_1, http://link.springer.com/10.1007/978-3-030-51054-1_1, series Title: Lecture Notes in Computer Science
2. Avigad, J., Awodey, S., Buchholtz, U., Doorn, F.v., Newstead, C., Rijke, E., Shulman, M.: Spectral Sequences in Homotopy Type Theory (Nov 2015), https://github.com/cmu-phil/Spectral
3. Baanen, A.: leanprover-community/mathlib#11750: define subobject classes from submonoid up to subfield (Apr 2022), https://github.com/leanprover-community/mathlib/pull/11750
4. Baanen, A.: Use and abuse of instance parameters in the Lean mathematical library. In: ITP 2022. Haifa, Israel (May 2022), http://arxiv.org/abs/2202.01629
5. Barton, Reid, Commelin, Johan, Buzzard, Kevin, Lau, Kenny, Carneiro, Mario: #maths > CDGAs (Jun 2019), https://leanprover-community.github.io/archive/stream/116395-maths/topic/CDGAs.html
6. Bourbaki, N.: Algebra I, Chapters 1-3. Elements of Mathematics, Springer, Berlin Heidelberg (1989)
7. Brunerie, G., Ljungström, A., Mörtberg, A.: Synthetic Integral Cohomology in Cubical Agda. In: Manea, F., Simpson, A. (eds.) 30th EACSL Annual Conference on Computer Science Logic (CSL 2022). Leibniz International Proceedings in Informatics (LIPIcs), vol. 216, pp. 11:1–11:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2022). https://doi.org/10.4230/LIPIcs.CSL.2022.11, https://drops.dagstuhl.de/opus/volltexte/2022/15731, iSSN: 1868-8969
8. Castelvecchi, D.: Mathematicians welcome computer-assisted proof in 'grand unification' theory. Nature **595**(7865), 18–19 (Jul 2021). https://doi.org/10.1038/d41586-021-01627-2, http://www.nature.com/articles/d41586-021-01627-2
9. Domínguez, C., Rubio, J.: Effective homology of bicomplexes, formalized in Coq. Theoretical Computer Science **412**(11), 962–970 (Mar 2011). https://doi.org/10.1016/j.tcs.2010.11.016, https://linkinghub.elsevier.com/retrieve/pii/S0304397510006493
10. Moura, L.d., Kong, S., Avigad, J., Doorn, F.v., Raumer, J.v.: The Lean Theorem Prover (System Description). In: Automated Deduction - CADE-25. pp. 378–388. Springer, Cham (Aug 2015). https://doi.org/10.1007/978-3-319-21401-6_26, https://link.springer.com/chapter/10.1007/978-3-319-21401-6_26
11. The mathlib Community: The lean mathematical library. In: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 367–381. ACM, New Orleans LA USA (Jan 2020). https://doi.org/10.1145/3372885.3373824, https://dl.acm.org/doi/10.1145/3372885.3373824
12. The Stacks project authors: The Stacks project (2022), https://stacks.math.columbia.edu
13. Wieser, E.: Scalar actions in Lean's mathlib. In: CICM 2021. Timisoara, Romania (Aug 2021), http://arxiv.org/abs/2108.10700
14. Wieser, E., Song, U.: Formalizing Geometric Algebra in Lean. Advances in Applied Clifford Algebras **32**(3), 28 (Jul 2022). https://doi.org/10.1007/s00006-021-01164-1, https://link.springer.com/10.1007/s00006-021-01164-1