Chaining extensionality lemmas in Lean's Mathlib

Eric Wieser[1[0000−0003−0412−4978]]

Cambridge University Engineering Department, UK, `efw27@cam.ac.uk`

Abstract. In its most basic form, Lean's `ext` or "extensionality" tactic reduces equalities of functions `f = g` into equality at every evaluation `∀ x, f x = g x`, and equalities of sets `s = t` into equivalence of membership in each set `∀ x, x ∈ s ↔ x ∈ t`. The tactic is extensible; new scenarios can be enabled by adding an `@[ext]` attribute to a theorem, for instance to add support for finite sets analogous to the support for sets.
Where the `ext` tactic can provide particular value is when working with equalities of morphisms; for instance, to show that two linear maps from the tensor product of two modules agrees, it suffices to show that the two maps agree on the pure tensors. Using tensor products as the main example, this paper explores a well-established pattern in mathlib (championed largely by the author) that declares these extensions in a way that allows "chaining"; by preferring to state the assumption in terms of an equality of "partially-applied" morphisms, quantifying over elements only as a last result.
Inevitably, the design of these tools shapes the way in which they are used; this paper concludes by noting how the `ext` tactic encourages expressing statements in a point-free manner, which at times impedes clarity.

Keywords: Extensionality · Tactics · Formalization · mathlib

1 Introduction

Extensionality is an important and sometimes foundational concept in theorem proving; in Lean [0], it manifests as the axiom `propext` (which says that two propositions are equal if each implies the other), and the theorem `funext` (which says that two functions are equal if their evaluations agree everywhere).

The mathematics library for Lean, mathlib [0], augments[1] these fundamental results with a tactic, `ext` [mathlib#104], which allows extensibility via tagging theorems with an `@[ext]` attribute, such as `Set.ext` which[2] turns equalities of sets (`s = t`) into equivalence of membership in each set (`∀ x, x ∈ s ↔ x ∈ t`). We will call such theorems "extensionality lemmas". The tactic itself is simple: it

[1] Or at least, used to; as of [lean4#3306], the tactic is now part of Lean itself.
[2] By combining the more-foundational `funext` and `propext`.

just finds theorems with this attribute whose conclusion matches the current equality, and applies[3] them repeatedly, introducing any new variables from `∀` quantifiers along the way. In this paper, we shall focus instead on how to choose these extensionality lemmas.

An obvious set of candidates for registration with this mechanism are morphisms (such as linear maps) and sub-objects (such as subspaces), for which the extensionality lemmas are respectively trivial extensions of function extensionality and set extensionality. The former can be stated as theorem 1:

Theorem 1. *For a commutative ring R and a pair of R-modules M, N, to show two R-linear maps $f, g : M \to_R N$ are equal, it suffices to show that they agree everywhere;* $\forall m, f(m) = g(m)$.

In some cases, we can write a more specialized extensionality lemma. One particularly useful example is

Theorem 2. *For a commutative ring R and an R-module M, to show two R-linear maps $f, g : R \to_R M$ are equal, it suffices to show that they agree on 1;* $f(1) = g(1)$.

For a more interesting example, let us consider linear maps from the tensor product of two modules, for which the natural statement of extensionality is

Theorem 3. *For a commutative ring R and a trio of R-modules M, N, P, to show two R-linear maps $f, g : (M \otimes_R N) \to_R P$ are equal, it suffices to show that they agree on the pure tensors;* $\forall m, \forall n, f(m \otimes n) = g(m \otimes n)$.

Due to its weaker assumption, this is a stronger statement than the extensionality lemma for linear maps in theorem 1. We can write theorem 3 in Lean as follows:

```
theorem TensorProduct.ext {R M N P : Type*}
    [CommSemiring R] [AddCommMonoid M] [AddCommMonoid N] [AddCommMonoid P]
    [Module R M] [Module R N] [Module R P]
    {f g : (M ⊗[R] N) →ₗ[R] P}
    (H : ∀ (m : M) (n : N), f (m ⊗ₜ n) = g (m ⊗ₜ n)) : f = g :=
  sorry
```

This a much more useful lemma than the one that requires `H : ∀ (mn : M ⊗[R] N), f mn = g mn`, as it saves us from having to split `mn` into pure tensors ourselves. To see this benefit, we can work through a proof that `(TensorProduct.comm R M N).symm = TensorProduct.comm R N M`; that is, the natural braiding of the tensor product that on the pure tensors sends $m \otimes n \mapsto n \otimes m$ is symmetric. Listing 1 compares the formalization of such a proof with and without theorem 3. Without the assistance of theorem 3, we are forced to induct on the structure of the tensor product, and end up with two additional subgoals that we'd prefer not to think about.

[3] In the sense of backwards reasoning.

```
variable {R M N : Type*}
variable [CommSemiring R] [AddCommMonoid M] [AddCommMonoid N] [Module R M] [Module R N]
```

```
theorem TensorProduct.comm_symm :
    (comm R M N).symm = comm R N M := by
  ext nm
  show (comm R M N).symm nm = comm R N M nm
  induction nm using TensorProduct.induction_on with
  | zero => -- the `nm = 0` case
    simp only [map_zero]
  | tmul n m => -- the `nm = n ⊗ₜ m` case
    rfl
  | add x y hx hy => -- the `nm = x + y` case with
    -- `hx : (comm R M N).symm x = comm R N M x`
    -- `hy : (comm R M N).symm y = comm R N M y`
    simp only [hx, hy, map_add]
```

(a) Using only `LinearMap.ext`, theorem 1

```
theorem TensorProduct.comm_symm :
    (comm R M N).symm = comm R N M := by
  apply LinearEquiv.toLinearMap_injective
  ext n m
  show
    (comm R M N).symm (n ⊗ₜ m) = comm R N M (n ⊗ₜ m)
  rfl -- true by definition!
```

(b) Using `TensorProduct.ext`, theorem 3

Listing 1: Proofs that the natural braiding of the tensor product is symmetric

Using theorem 3 in (b) results in a much simpler argument than (a). Writing a copy of theorem 3 for linear *equivalences*, the `apply` tactic in b can also be dropped.

2 Chaining extensionality lemmas

However, even theorem 3 still only scratches the surface of the power behind the `ext` tactic. Where it excels is in its ability to *chain* extensionality lemmas. A simple example of this is reducing equalities of two-argument functions into showing they agree when fully-applied (`∀ x y, f x y = g x y`), but there are far more interesting cases. In particular, we shall explore how the `ext` tactic can be chained on equalities of morphisms, specifically linear maps and algebra morphisms. The key insight that massively boosts the power of `ext` is the fact that turning an equality of morphisms into an equality of its evaluations should be a last resort! This may seem surprising, since in the simple examples it seemed like the raison d'être of `ext` was to introduce these `∀ x` quantifiers; but there are frequently better approaches.

To better understand this insight, we start with a warning of what happens if we overlook it, by examining the following extension of theorem 3:

Theorem 4. *For a commutative ring R and a quadruplet of R-modules M, N, P, Q, to show two R-linear maps $f_3, g_3 : ((M \otimes_R N) \otimes_R P) \to_R Q$ are equal, it suffices to show that they agree on the pure tensors; $\forall m, \forall n, \forall p, f((m \otimes n) \otimes p) = g((m \otimes n) \otimes p)$.*

The statement of this extensionality lemma raises an immediate red flag; it suggests that we are doomed to state a new theorem for every possible arity and associativity of tensor products[4] (and of course to prove each of them!). We can

[4] Indeed, proving that vector spaces form a monoidal category requires two different associativities of the 4-ary version.

try to lessen this blow by proving theorem 4 in terms of theorem 3, but it only gets us halfway; we are left to prove that $\forall x_{mn} : M \otimes N, \forall p, f(x_{mn} \otimes p) = g(x_{mn} \otimes p)$, where we have successfully taken apart only one of the two tensor products, and are once again forced to induct upon the structure of x_{mn}.

Our trouble here is that theorem 3 is too weak; it cannot be chained with itself, because it consumes an equality of elements not an equality of morphisms. To correct this, we state it as in theorem 5:

Theorem 5. *For a commutative ring R and a trio of R-modules (M, N, P), to show two R-linear maps $f, g : (M \otimes_R N) \to_R P$ are equal, it suffices to show that they agree* when composed with *the canonical bilinear map* $(\otimes) : M \to_R N \to_R (M \otimes_R N)$*; $f \circ_2 (\otimes) = g \circ_2 (\otimes)$. This equality is an equality of bilinear maps (linear maps with a codomain that is itself a linear map) of type $M \to_R N \to_R P$.*

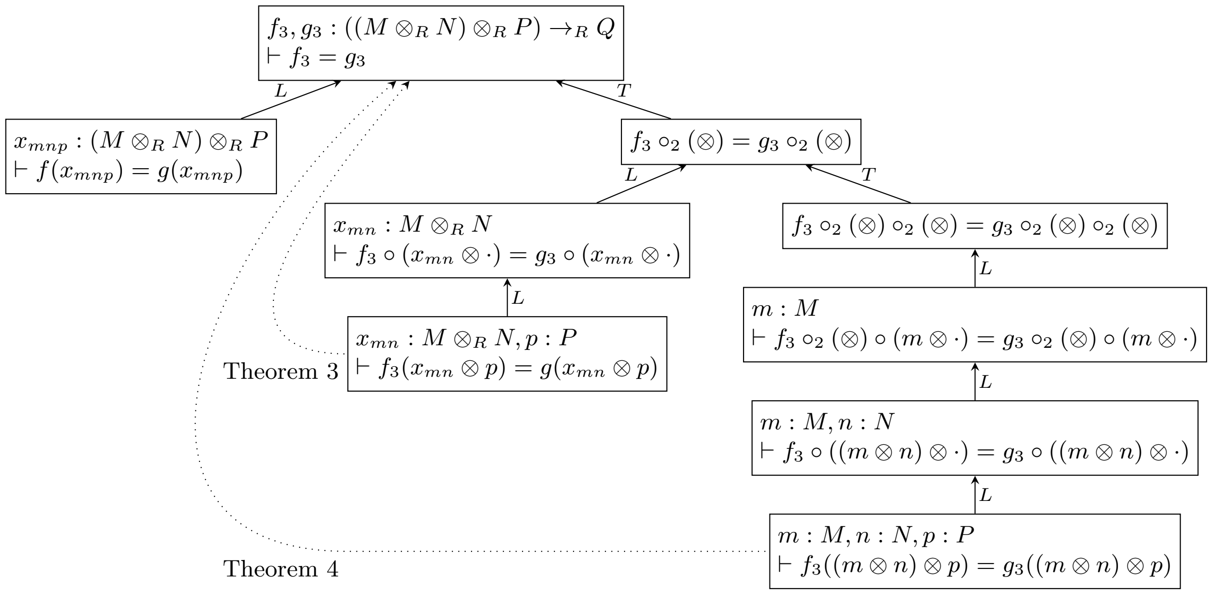
The proof of theorem 5 follows immediately from that of theorem 3; indeed, applying `ext` turns the former into the latter! In Lean, this condition is written `(TensorProduct.mk R M N).compr₂ f = (TensorProduct.mk R M N).compr₂ g`, where `b.compr₂ f` (or $f \circ_2 b$) is the bilinear map such that `b.compr₂ f m n = f (b m n)`.

We now arrive at our key conclusion; theorem 4 can be proven using iterated applications of theorem 5. The approach is shown in fig. 1, where T is theorem 5 and L is theorem 1. It is here where our earlier remark that "turning an equality of morphisms into an equality of its evaluations should be a last resort" comes into play; the most structured (and thus easiest to prove) statement is reached by preferring T edges over L edges, as taking an L edge prematurely leads to a dead end. The `ext` tactic can handle this graph traversal automatically; by setting the T edges to have higher priority, they will be attempted first. It can be seen that in fact, variants of theorem 4 for *any* arity of associativity of linear maps from tensor products can be tackled in the same way; T edges will always split the left-most tensor product, and L edges will consume non-tensor products from the left.

3 Wider applications

The benefits of this strategy (which we call "partially-applied `ext` lemmas") extends far beyond tensor products; there are numerous other situations where we can apply them:

- To show two linear maps $(M \oplus N) \to_R P$ from binary direct sums of modules agree, it suffices to show that they agree when composed with $\mathrm{inl} : M \to_R M \oplus N \coloneqq m \mapsto (m, 0)$ and $\mathrm{inr} : N \to_R M \oplus N \coloneqq n \mapsto (0, n)$; $f \circ \mathrm{inl} = g \circ \mathrm{inl}$ and $f \circ \mathrm{inr} = g \circ \mathrm{inr}$.
- To show two linear maps $f, g : (\bigoplus_i M_i) \to_R N$ from n-ary direct sums of modules agree, it suffices to show that they agree when composed with every canonical injection into the ith component $\iota_i : M_i \to \bigoplus_i M_i$; $\forall i, f \circ \iota_i = g \circ \iota_i$.
- To show two linear maps from polynomials $f, g : R[X] \to_R M$ agree, it suffices to show they agree when composed with each of the maps that scale the kth power of X; $\forall n, f \circ (r \mapsto rX^k) = g \circ (r \mapsto rX^k)$.



$f_3, g_3 : ((M \otimes_R N) \otimes_R P) \to_R Q$
$\vdash f_3 = g_3$
L
T
$x_{mnp} : (M \otimes_R N) \otimes_R P$
$\vdash f(x_{mnp}) = g(x_{mnp})$
$f_3 \circ_2 (\otimes) = g_3 \circ_2 (\otimes)$
$x_{mn} : M \otimes_R N$
$\vdash f_3 \circ (x_{mn} \otimes \cdot) = g_3 \circ (x_{mn} \otimes \cdot)$
$f_3 \circ_2 (\otimes) \circ_2 (\otimes) = g_3 \circ_2 (\otimes) \circ_2 (\otimes)$
$m : M$
$\vdash f_3 \circ_2 (\otimes) \circ (m \otimes \cdot) = g_3 \circ_2 (\otimes) \circ (m \otimes \cdot)$
$x_{mn} : M \otimes_R N, p : P$
$\vdash f_3(x_{mn} \otimes p) = g(x_{mn} \otimes p)$
Theorem 3
$m : M, n : N$
$\vdash f_3 \circ ((m \otimes n) \otimes \cdot) = g_3 \circ ((m \otimes n) \otimes \cdot)$
$m : M, n : N, p : P$
$\vdash f_3((m \otimes n) \otimes p) = g_3((m \otimes n) \otimes p)$
Theorem 4

Fig. 1: Factorizing theorems 3 and 4 into theorem 5 (T) and theorem 1 (L)

Arrows show implications. Branching indicates that *either* child is sufficient, not that both are necessary. Indeed, extensionality between tensor products of any arity of associativity factors through T and L.

Arguably the cases that these examples apply to are all just different special cases of free modules, but to `mathlib` they are genuinely different objects, and so we must teach `ext` about each of them separately. Similar families of extensionality lemmas exist for special cases (or quotients) of free monoids (and monoid morphisms), free rings (and ring morphisms), and free algebras (and algebra morphisms). Crucially, within these families, the "partially-applied" lemmas are mutually compatible; as each makes the minimal amount of progress and avoids applying L (theorem 1). For instance, if faced with a pair of maps

$$f, g : ((R[X] \otimes M) \oplus (\bigoplus_i N_i)) \to_R P,$$

then `ext` will leave us to prove both of

$$\begin{aligned} \forall k, \forall m, f((X^k \otimes m, 0)) &= g((X^k \otimes m, 0)) \\ \forall i, \forall n, \quad f((0, \iota_i(n))) &= g((0, \iota_i(n))), \end{aligned}$$

as shown in fig. 2.

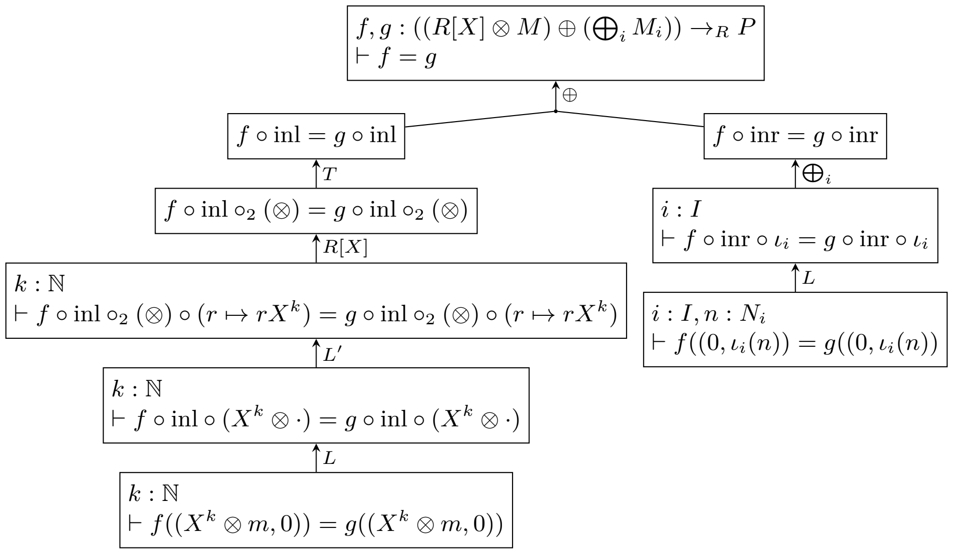

$f, g : ((R[X] \otimes M) \oplus (\bigoplus_i M_i)) \to_R P$
$\vdash f = g$
$\oplus$
$f \circ \mathrm{inl} = g \circ \mathrm{inl}$
$f \circ \mathrm{inr} = g \circ \mathrm{inr}$
T
$\bigoplus_i$
$f \circ \mathrm{inl} \circ_2 (\otimes) = g \circ \mathrm{inl} \circ_2 (\otimes)$
$i : I$
$\vdash f \circ \mathrm{inr} \circ \iota_i = g \circ \mathrm{inr} \circ \iota_i$
$R[X]$
L
$k : \mathbb{N}$
$\vdash f \circ \mathrm{inl} \circ_2 (\otimes) \circ (r \mapsto rX^k) = g \circ \mathrm{inl} \circ_2 (\otimes) \circ (r \mapsto rX^k)$
$i : I, n : N_i$
$\vdash f((0, \iota_i(n))) = g((0, \iota_i(n)))$
L'
$k : \mathbb{N}$
$\vdash f \circ \mathrm{inl} \circ (X^k \otimes \cdot) = g \circ \mathrm{inl} \circ (X^k \otimes \cdot)$
L
$k : \mathbb{N}$
$\vdash f((X^k \otimes m, 0)) = g((X^k \otimes m, 0))$

Fig. 2: Extensionality for a linear map from an arbitrarily-chosen compound type.

Here, the edges labelled $R[X]$, $\oplus$, and $\bigoplus_i$ are the extensionality lemmas listed in section 3 (for polynomials, binary-direct sums, and n-ary direct sums, respectively), L' is theorem 2, and L and T are the same as in fig. 1.

4 Point-free statements

So far, we have seen how extensionality lemmas can be designed to greatly aid the task of proving equalities between morphisms from complex types. Unfortunately, most equalities we face are between the objects, and we rarely face the equalities of morphisms presented so far unless we deliberately frame our problems in a certain way. This section will show how to perform this framing.

A simple example of adjusting our statements to make such an equality appear arises when building an isomorphism of modules $M \cong_R N$ from the forward and inverse maps $f : M \to_R N$ and $f^{-1} : N \to_R M$. The conventional way to construct the isomorphism would be to show that these morphisms are left and right inverses through $\forall m, g(f(m)) = m$ and $\forall n, f(g(n)) = n$; which are equalities of objects, not morphisms. If we instead re-frame our statements to $g \circ f = \mathrm{id}$ and $f \circ g = \mathrm{id}$, we are faced with equalities of morphisms that we can apply `ext` to. `mathlib` already contains many definitions that assemble isomorphisms in this way. Once again turning to tensor products as an illustrative example, if we have $M := M_1 \otimes_R M_2$ and $N := N_1 \otimes_R N_2$, then this approach of showing that the composition is the identity, combined with the `ext` tactic, allows us to prove f and g are inverses by considering only $\forall m_1, \forall m_2, g(f(m_1 \otimes m_2)) = m_1 \otimes m_2$ and $\forall n_1, \forall n_2, f(g(n_1 \otimes n_2)) = n_1 \otimes n_2$.

More generally, when faced with an equality of objects in terms of two functions of a free variable, we can reduce our problem to an equality of morphisms by pushing that variable all the way to an application on the right. For instance, if we want to show that $\forall x, x \times y = y \times x$ (where $\times$ is an arbitrary bilinear operation), we can:

– rewrite as $\forall x, (\cdot \times y)(x) = (y \times \cdot)(x)$, where the free variable x is now on the right on both sides;
– note that $(\cdot \times y)$ and $(y \times \cdot)$ describe linear maps
– conclude that it would be sufficient to prove $(\cdot \times y) = (y \times \cdot)$, which is an equality of morphisms

This allows us to apply any extensionality lemmas that replace x with the more restricted structured values (such as pure tensors). We can then repeat to put y on the right, and apply more relevant extensionality lemmas. A particularly common situation where this trick is useful is for putting multiplicative structures upon new algebras, such as upon tensor products, tensor powers, and direct sums.

As it turns out, there is a much more concise way of presenting this reasoning for this example; we can show outright[5] that $\times$ is a bilinear map, and write our goal with all the free variables on the right as $(\times)(x, y) = \text{flip}(\times)(x, y)$, where flip turns one bilinear map into another. We conclude that it is sufficient to prove that $(\times) = \text{flip}(\times)$, an equality of bilinear maps on which we can then apply all our extensionality lemmas in one go.

In general, this style of writing functions with no free variables is called "point-free", and is a fairly common trick in function programming languages[6] and in category theory[7]. To give a more complex example, the function $x, y \mapsto f(x) \times g(y)$ can be written in a point-free style as either $\text{flip}(\text{flip}(\times) \circ g) \circ f$ or as $\text{flip}(\circ)(g) \circ (\times) \circ f$. What sets our situation apart from the typical point-free approach is that we are not simply constructing functions, but morphisms that preserve algebraic operators. As a result, we need an extensive library of composition operators that are themselves morphisms; which for linear maps, relies heavily upon the infrastructure described in [0, §3.1].

5 Conclusion

This paper summarized the state of the `ext` tactic in mathlib, and explained how "partially-applied ext-lemmas" can provide significant value; especially when combined with careful construction of point-free statements. The approach was presented here exclusively through Lean, but would be straightforward to implement in other similar systems like Coq.

While empowering the `ext` tactic and natural in category theory, point-free statements are often awkward to construct mid-proof. Thankfully, there is work in the Lean ecosystem to alleviate this: [0] introduces a `fun x =>L[R] f x` notation

[5] As was pointed out by Greg Price, who tidied the author's proofs in [mathlib#7029].
[6] Such as Haskell, whose wiki has a Pointfree page.
[7] Where in general, we do not have a notion of "applying" morphisms

for continuous linear maps (and a selection of other morphism types), providing users with the same flexibility they are used to with the builtin `fun x => f x` notation, but invoking automation to prove the result satisfies the properties of the required morphism type (using [0]'s `fun_prop` tactic). In this style, our final example in section 4 of $x, y \mapsto f(x) \times g(y)$ could be written as a linear map as `fun x =>ₗ[R] fun y =>ₗ[R] f x * g y`.

The author cannot claim credit for the `ext` tactic, nor for the idea of using "partially-applied ext lemmas", but is responsible for extending this pattern through many relevant types in mathlib. The first sign of such a lemma in mathlib that the author is aware of was added by Chris Hughes in [mathlib#3408] for the semidirect product, though it was not tagged with `@[ext]`. Chris Hughes was also responsible for suggesting a similar lemma for the tensor algebra in review of [mathlib#3531], though once again it was never tagged `@[ext]`. Scott Morrison appears to have realized the value of the `@[ext]` attribute when generalizing the construction of the tensor algebra to a quotient of the free algebra in [mathlib#4078]. The first explicit mention of chaining that the author can find was by Yury Kudryashov, who noted in [mathlib#4741] that it was useful for working with free modules and algebras.

The author is responsible for documenting the pattern in [mathlib#5484], and for contributing numerous extensionality lemmas, some further examples of which are shown in appendix A.

Acknowledgments The author is funded by a scholarship from the Cambridge Trust.

A Extensionality lemmas contributed by the author

This appendix provides the statements of a selection of `@[ext]` lemmas contributed by the author to mathlib, which either follow the pattern described in section 2 to allow chaining, or eliminate quantifiers at the end of such chains as theorem 2 does. For brevity, the `@[ext]` attribute, along with the precise information about types `A`, `B`, e.t.c„ has been omitted from every example. The theorem names can (at the time of writing) be looked up in the online mathlib documentation to see the fully-quantified statements.

While the code is presented here in Lean 4 (as this is how it now survives in mathlib 4), the majority of these were contributed in Lean 3, and so the GitHub references lead to discussions about Lean 3 code.

A.1 Algebra morphisms

– from the exterior algebra in [mathlib#4297]

```
theorem ExteriorAlgebra.hom_ext {f g : ExteriorAlgebra R M →ₐ[R] A} :
    f.toLinearMap.comp (ι R) = g.toLinearMap.comp (ι R) → f = g
```

– from the Clifford algebra in [mathlib#4430]

```
theorem CliffordAlgebra.hom_ext {f g : CliffordAlgebra Q →ₐ[R] A} :
    f.toLinearMap.comp (ι Q) = g.toLinearMap.comp (ι Q) → f = g
```

– from graded algebras in [mathlib#8783]

```
theorem DirectSum.algHom_ext' {f g : (⨁ i, A i) →ₐ[R] B} :
    (∀ i, f.toLinearMap.comp (lof _ _ A i) = g.toLinearMap.comp (lof _ _ A i)) →
      f = g
```

– from the complex numbers in [mathlib#8105]

```
theorem Complex.algHom_ext {f g : ℂ →ₐ[ℝ] A} : f I = g I → f = g
```

– from the dual numbers in [mathlib#10730]

```
theorem DualNumber.algHom_ext {f g : R[ε] →ₐ[R] A} : f ε = g ε → f = g
```

– from the trivial square-zero extension in [mathlib#10754]

```
theorem TrivSqZeroExt.algHom_ext' {f g : tsze R M →ₐ[S] A}
    (hinl : f.comp (inlAlgHom S R M) = g.comp (inlAlgHom S R M))
    (hinr : f.toLinearMap.comp (inrHom R M |>.restrictScalars S) =
      g.toLinearMap.comp (inrHom R M |>.restrictScalars S)) :
    f = g
```

– from the tensor product of algebras in [mathlib4#6417]

```
theorem Algebra.TensorProduct.ext {f g : (A ⊗[R] B) →ₐ[S] C}
    (hinl : f.comp includeLeft = g.comp includeLeft)
    (hinr : (f.restrictScalars R).comp includeRight =
      (g.restrictScalars R).comp includeRight) :
    f = g
```

– from polynomials over an algebra in [mathlib4#8116]

```
theorem Polynomial.algHom_ext' ⦃f g : A[X] →ₐ[R] B⦄
    (hC : f.comp CAlgHom = g.comp CAlgHom)
    (hX : f X = g X) :
    f = g
```

A.2 Linear maps

– from n-ary direct sums in [mathlib#4821]

```
theorem DirectSum.decompose_lhom_ext ⦃f g : M →ₗ[R] N⦄ :
    (∀ i, f ∘ₗ (□ i).subtype = g ∘ₗ (□ i).subtype) → f = g
```

– from tensor products in [mathlib#6105] (theorem 5)

```
theorem TensorProduct.ext ⦃f g : M ⊗ N →ₗ[R] P⦄ :
    ((mk R M N).compr₂ f = (mk R M N).compr₂ g) → f = g
```

– from binary direct sums in [mathlib#6124]

```
theorem LinearMap.prod_ext ⦃f g : M × M₂ →ₗ[R] M₃⦄
    (hl : f.comp (inl _ _ _) = g.comp (inl _ _ _))
    (hr : f.comp (inr _ _ _) = g.comp (inr _ _ _)) :
    f = g
```

– from the exterior algebra in [mathlib#14803]

```
theorem ExteriorAlgebra.lhom_ext ⦃f g : ExteriorAlgebra R M →ₗ[R] N⦄ :
    (∀ i, f.compAlternatingMap (ιMulti R i) = g.compAlternatingMap (ιMulti R i))
      → f = g
```

A.3 Other morphisms

– ring morphisms from $\mathbb{Z}[\sqrt{d}]$ in [mathlib#5640]

```
theorem Zqrtd.hom_ext ⦃f g : ℤ√d →+* R⦄ : f sqrtd = g sqrtd → f = g
```

– morphisms from quotient constructions in [mathlib#8641]: quotients by subgroups, submodules, lie submodules, and ideals

```
theorem QuotientGroup.monoidHom_ext ⦃f g : G ⧸ N →* M⦄ :
    f.comp (mk' N) = g.comp (mk' N) → f = g
```

```
theorem Submodule.linearMap_qext ⦃f g : M ⧸ p →ₛₗ[τ₁₂] M₂⦄ :
    f.comp p.mkQ = g.comp p.mkQ → f = g
```

```
theorem LieSubmodule.Quotient.lieModuleHom_ext ⦃f g : M ⧸ N →ₗ⁅R,L⁆ M⦄ :
    f.comp (mk' N) = g.comp (mk' N) → f = g :=
```

```
theorem Ideal.Quotient.ringHom_ext ⦃f g : R ⧸ I →+* S⦄ :
    f.comp (mk I) = g.comp (mk I) → f = g
```

References

[0] Leonardo de Moura and Sebastian Ullrich. "The Lean 4 Theorem Prover and Programming Language". In: *Automated Deduction – CADE 28*. CADE 2021. Ed. by André Platzer and Geoff Sutcliffe. Vol. 12699. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 625–635. ISBN: 978-3-030-79876-5. DOI: `10.1007/978-3-030-79876-5_37` (cit. on p. 1).

[0] The mathlib Community. "The Lean Mathematical Library". In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. POPL '20: 47th Annual ACM SIGPLAN Symposium on Principles of Programming Languages. New Orleans LA USA: ACM, Jan. 20, 2020, pp. 367–381. ISBN: 978-1-4503-7097-4. DOI: `10.1145/3372885.3373824` (cit. on p. 1).

[0] Eric Wieser. "Scalar Actions in Lean's Mathlib". In: *Workshop Papers of the 14th Conference on Intelligent Computer Mathematics*. CICM 2021. Vol. 3377. Timisoara, Romania: CEUR-WS, Aug. 10, 2021. arXiv: `2108.10700 [cs.LO]` (cit. on p. 7).

[0] Tomáš Skřivan. *Lecopivo/SciLean: Scientific Computing in Lean 4*. URL: `https://github.com/lecopivo/SciLean` (visited on 02/18/2024) (cit. on pp. 7, 8).

Github references

[lean4#3306] Scott Morrison. *chore: upstream `ext` tactic*. Feb. 12, 2024 (cit. on p. 1).

[mathlib#104] Simon Hudon. *feat(tactic/ext): new `ext` tactic and corresponding `extensionality`...* Reviewed by Johannes Hölzl and Mario Carneiro. Apr. 2018 (cit. on p. 1).

[mathlib#7029] Greg Price. *chore(algebra/direct_sum_graded): golf proofs*. Reviewed by Eric Wieser and Scott Morrison. Apr. 2021 (cit. on p. 7).

[mathlib#3408] Chris Hughes. *feat(group_theory/semidirect_product): `mk_eq_inl_mul_inr` and `hom_ext`*. Reviewed by Scott Morrison. July 2020 (cit. on p. 8).

[mathlib#3531] Adam Topaz. *feat(linear_algebra/tensor_algebra): Tensor algebras*. Reviewed by Eric Wieser, Scott Morrison, Patrick Massot, Johan Commelin, and Chris Hughes. July 2020 (cit. on p. 8).

[mathlib#4078] Scott Morrison. *feat(algebra/ring_quot): quotients of noncommutative rings*. Reviewed by Eric Wieser, Kenny Lau, and Johan Commelin. Sept. 2020 (cit. on p. 8).

[mathlib#4741] Yury G. Kudryashov. *chore(*): a few more type-specific ext lemmas*. Reviewed by Johan Commelin and Eric Wieser. Oct. 2020 (cit. on p. 8).

[mathlib#5484] Eric Wieser. *feat(group_theory/*): mark some lemmas as ext (about homs out of free constructions)*. Reviewed by Floris van Doorn. Dec. 2020 (cit. on p. 8).

[mathlib#4297] Eric Wieser. *feat(linear_algebra/exterior_algebra): Add an exterior algebra*. Reviewed by Anne Baanen and Scott Morrison. Sept. 2020 (cit. on p. 9).

[mathlib#4430] Eric Wieser. *feat(linear_algebra/clifford_algebra): Add a definition derived from* `exterior_algebra.lean`. Reviewed by Anne Baanen, Adam Topaz, Heather Macbeth, and Utensil Song. Oct. 2020 (cit. on p. 9).

[mathlib#8783] Eric Wieser. *feat(algebra/direct_sum): graded algebras*. Reviewed by Kevin Buzzard and Johan Commelin. Aug. 2021 (cit. on p. 9).

[mathlib#8105] Eric Wieser. *feat(data/complex/module): add* `complex.alg_hom_ext`. Reviewed by Anne Baanen. June 2021 (cit. on p. 9).

[mathlib#10730] Eric Wieser. *feat(linear_algebra/clifford_algebra/equivs): There is a clifford algebra isomorphic to the dual numbers*. Reviewed by Johan Commelin and Rob Lewis. Dec. 2021 (cit. on p. 9).

[mathlib#10754] Eric Wieser. *feat(algebra/triv_sq_zero_ext): universal property*. Reviewed by Johan Commelin. Dec. 2021 (cit. on p. 9).

[mathlib4#6417] Eric Wieser. *feat(RingTheory/TensorProduct): heterogenize*. Reviewed by Johan Commelin and Antoine Chambert-Loir. Aug. 2023 (cit. on p. 9).

[mathlib4#8116] Eric Wieser. *feat(Data/Polynomial/AlgebraMap): more results for non-commutative polynomials*. Reviewed by Yaël Dillies and Johan Commelin. Nov. 2023 (cit. on p. 10).

[mathlib#4821] Eric Wieser. *feat(data/dfinsupp): Port over the* `finsupp.lift_add_hom` *API*. Reviewed by Johan Commelin. Oct. 2020 (cit. on p. 10).

[mathlib#6105] Eric Wieser. *refactor(linear_algebra/tensor_product): Use a more powerful lemma for ext*. Reviewed by Johan Commelin. Feb. 2021 (cit. on p. 10).

[mathlib#6124] Eric Wieser. *feat(linear_algebra/prod): add an ext lemma that recurses into products*. Reviewed by Johan Commelin. Feb. 2021 (cit. on p. 10).

[mathlib#14803] Eric Wieser. *feat(linear_algebra/clifford_algebra/of_alternating): extend alternating maps to the exterior algebra*. Reviewed by Oliver Nash. June 2022 (cit. on p. 10).

[mathlib#5640] Eric Wieser. *feat(data/zsqrtd/to_real): Add* `to_real`. Reviewed by Johan Commelin, Mario Carneiro, Bryan Gin-ge Chen, and Anne Baanen. Jan. 2021 (cit. on p. 10).

[mathlib#8641] Eric Wieser. *feat(linear_algebra/basic, group_theory/quotient_group, algebra/lie/quotient): ext lemmas for morphisms out of quotients*. Reviewed by Oliver Nash and Anne Baanen. Aug. 2021 (cit. on p. 10).