

Formalising mathematics in Lean (Week 7 - Diamonds)

A GlaMS course

M. Abu Omar, S. Castellan, A. Doña Mateo, & P. Kinnear

a.k.a. **The Lean Team**

March 19th, 2024

Learning outcomes

- ▶ Identifying diamonds (i.e., type mismatch errors)
- ▶ Resolving diamonds

Example

- ▶ `example [hE1 : NormedAddCommGroup E]`
 `[hE2 : InnerProductSpace ℂ E] [Ring E] [Algebra ℂ E]`
 `(T : E →1[ℂ] E) [hE5 : FiniteDimensional ℂ E] :`
 `LinearMap.adjoint T = T`
- ▶ The error says that there is an “application type mismatch”. Can we be more precise to fix this?

`@LinearMap.adjoint ℂ E E Complex.instIsROrCComplex hE1`
`hE1 hE2 hE2 hE5 hE5 T = T`

- ▶ No, this still gives the same error. We need to expand the error to see what’s happening in order to fix it.

Identifying the cause

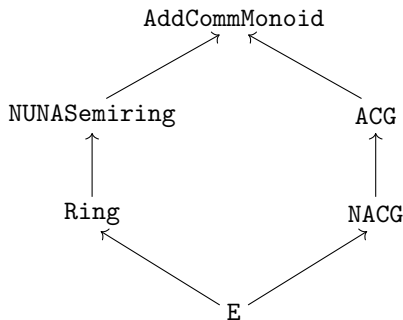
From the Infview:

```
/- Error: application type mismatch
   LinearMap.adjoint T
argument
   T
has type
   @LinearMap ℂ ℂ _ _ _ E E
   NonUnitalNonAssocSemiring.toAddCommMonoid
   NonUnitalNonAssocSemiring.toAddCommMonoid Algebra.toModule
   Algebra.toModule : Type u_1
but is expected to have type
   @LinearMap ℂ ℂ _ _ _ E E AddCommGroup.toAddCommMonoid
   AddCommGroup.toAddCommMonoid NormedSpace.toModule
   NormedSpace.toModule : Type u_1 -/
```

Identifying the cause (2)

- ▶ What did we notice from the error?
 - ▶ `Algebra.toModule` is clashing with `NormedSpace.toModule`
 - ▶ `AddCommGroup.toAddCommMonoid` is clashing with `NonUnitalNonAssocSemiring.toAddCommMonoid`
- ▶ Let's look at the second case more closely...

Identifying the cause (3): looks like a diamond!



- ▶ This is why we keep getting an application type mismatch error: because Lean keeps confusing `AddCommMonoid` induced by `NACG` with that induced by `Ring`, which are not known to be definitionally equal.

Attempting to fix the error: the naive approach

- ✗ Can we specify which instances the linear map should access? So in the same example, can we write:

```
example [hE1 : NormedAddCommGroup E] [InnerProductSpace ℂ E]
  [Ring E] [Algebra ℂ E] (T : E →1[ℂ] E)
  [FiniteDimensional ℂ E] :
  LinearMap.adjoint (T : @LinearMap ℂ ℂ _ _ E E
    (hE1.toAddCommMonoid) (hE1.toAddCommMonoid)
    (NormedSpace.toModule) (NormedSpace.toModule)) = T
```

Nope, we still get a type mismatch error.

- ✗ Okay, can we fix this by reordering the variables and instances?

```
example [NormedAddCommGroup E] [InnerProductSpace ℂ E] (T :
  E →1[ℂ] E) [Ring E] [Algebra ℂ E] [FiniteDimensional ℂ
  E] :
  LinearMap.adjoint T = T
```

Technically, this does fix the error. But, you are guaranteed to run into more problems later on!

Fixing the error: the naive approach (2)

- ▶ Does reordering the variables and instances truly resolve the error?
No.
- ▶ For example, what if we needed to define a linear map within the proof?

```
let f : E →1[C] E := sorry
have : @LinearMap.adjoint C E E _ hE1 hE1 hE2 hE2 hE hE
  f = 0 := sorry
```

We're back to the same error...

Fixing the error: the naive approach (3)

- ▶ One way to fix the above error is by being precise with the function:

```
let f : @LinearMap ℂ ℂ _ _ (RingHom.id ℂ) E E
    (hE1.toAddCommGroup.toAddCommMonoid)
    (hE1.toAddCommGroup.toAddCommMonoid)
    (NormedSpace.toModule) (NormedSpace.toModule) := sorry
have : LinearMap.adjoint f = 0 := sorry
```

Not very practical, but it works now.

- ▶ However, there is one error that we cannot get rid of. In particular, if we needed to access both the algebra and the inner product space at the same time:

```
example [NormedAddCommGroup A] [InnerProductSpace ℂ A]
  (T : ℂ →1 [ℂ] A) [Ring A] [Algebra ℂ A] :
  T ≡ Algebra.linearMap ℂ A
```

- ▶ So how do we fix this then?
- ▶ We need to resolve the diamonds!

Type inheritance diagrams

- ▶ Let A, B, and C be types.
- ▶ If we are given A with the instance B, then a *type inheritance diagram* of A is

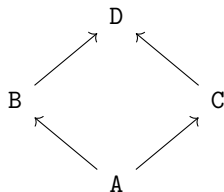


- ▶ If we also have B infers C, then we have the following diagram:



Diamonds

- ▶ So what are diamonds?
- ▶ Say A, B, C, and D are types, then a *diamond* is when we have the following type inheritance diagram:



In other words, a diamond is when we have D being inferred by both B and C which are inferred by A.

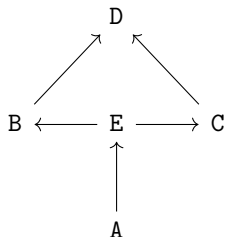
Diamonds (2)

- ▶ We call the diamond *transparent*, and denote it by \diamond , when D inferred by B is definitionally equal to D inferred by C.
In other words, when the type inheritance diagram commutes.
- ▶ Transparent diamonds can be left alone, they won't raise any errors.
- ▶ Non-transparent diamonds, denoted by \blacklozenge , are the ones that will raise errors and are the ones that need to be addressed.
- ▶ We also say a diamond is *resolved* when there are no non-transparent sub-diamonds (i.e., a diamond within a diamond).
In other words, the diamond is resolved when the error is resolved.

Diamonds (3)

How do we resolve diamonds?

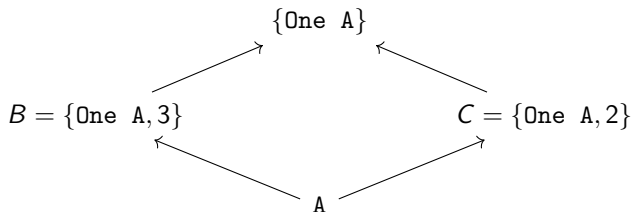
- ▶ We need to use a new class $E = C + B \setminus \{\text{common traits of C and B}\}$. Then E inherits both B and C. So we get the following type inheritance diagram:



- ▶ Now D created by B is equal to that created by C, because they are both created by E.

Example

- ▶ Let type A have an instance $B = \{\text{One } A, 3\}$ and $C = \{\text{One } A, 2\}$, where 2 and 3 here are just properties.
- ▶ Recall that One is a class with property $\text{one} : A$.
- ▶ So we have the following type inheritance diagram:

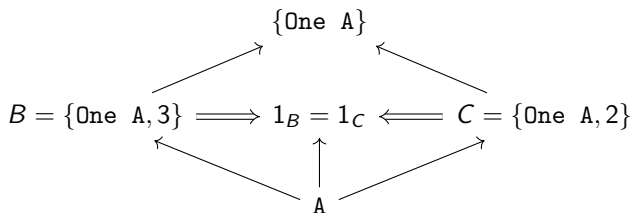


- ▶ Does 1_B need to equal 1_C ?
- ▶ No.

Example (2)

- ▶ One solution is to set constraints on both instances so that you do necessarily have equality.

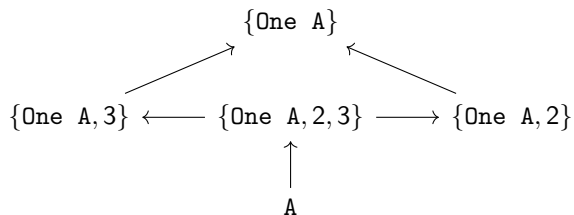
In this case you would get,



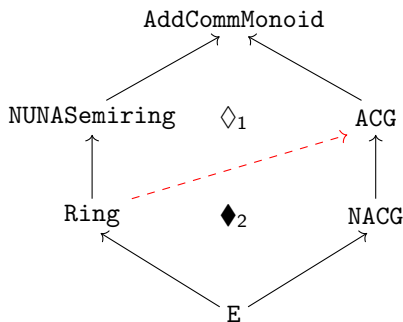
Here, \Rightarrow means that it uses B and C .

Example (3)

- ▶ The better solution is to simply have



Back to the first example



- ▶ In this example, there are technically two sub-diamonds since `Ring` implies `ACG`.
- ▶ \diamond_1 is transparent since `AddCommMonoid` created by `NUNASemiring` (created from `Ring`) is definitionally equal to that created by `ACG` (created from `Ring`).
- ▶ So it suffices to address the second sub-diamond \diamond_2 .

Resolving the diamond

- ▶ Since `Ring` and `NormedAddCommGroup` are the ones causing all of this, let's define a new class.
- ▶ In this class structure, we want to define `E` that combines `Ring` and `NormedAddCommGroup` such that `NormedAddCommGroup` would depend on `Ring`.
- ▶ Let's first take a look at how `NormedAddCommGroup` is defined:

```
class NormedAddCommGroup (E : Type*) extends Norm E,  
  AddCommGroup E, MetricSpace E where  
dist := fun x y => ||x - y||  
/-- The distance function is induced by the norm. -/  
dist_eq : ∀ x y, dist x y = ||x - y|| := by aesop
```

- ▶ So we want to have the same class, but with `Ring` instead of `AddCommGroup`. This way `NormedAddCommGroup` would depend on `Ring`.

Resolving the diamond (2)

```
class NACGoR (E : Type*)
  extends Norm E, Ring E, MetricSpace E where
  dist := fun x y => ||x - y||
  /-- The distance function is induced by the norm. -/
  dist_eq : ∀ x y, dist x y = ||x - y|| := by aesop
```

- ▶ With this new class, we get `AddCommMonoid` via the ring structure, which is what we wanted.

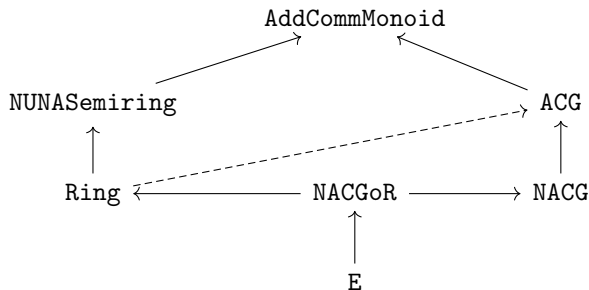
Resolving the diamond (3)

We need to also create the instances that our new class induces, in particular `Ring` and `NormedAddCommGroup`.

This allows Lean to instantly see `NACGoR` as both a `Ring` and a `NormedAddCommGroup` (where `NormedAddCommGroup` is given by its `Ring` structure).

Resolving the diamond (4)

Our new type inheritance diagram would look like the following.



Resolving the diamond (5)

Now let's try that example again:

```
example [NACGoR E] [InnerProductSpace  $\mathbb{C}$  E] [Algebra  $\mathbb{C}$  E]  
  [FiniteDimensional  $\mathbb{C}$  E] (T : E  $\rightarrow_1$  [ $\mathbb{C}$ ] E) :  
  LinearMap.adjoint T = 0
```

✓ It works!

Resolving the diamond (6)

- ▶ We now check that the instance `AddCommMonoid` created by the ring structure is definitionally equal to that created by `NACG`:

```
example [h : NACGoR E] :  
  h.toAddCommMonoid =  
    NormedAddCommGroup.toAddCommGroup.toAddCommMonoid :=  
  rfl
```

- ▶ We also check:

```
example [NACGoR E] :  
  (Ring.toAddCommGroup : AddCommGroup A) =  
    NormedAddCommGroup.toAddCommGroup :=  
  rfl
```

- ▶ Thus we have resolved the first diamond \diamond !
- ▶ But we still have potential issues left to address.

What now?


- ▶ What if we wanted to do:

```
example [NACGoR E] [InnerProductSpace ℂ E] [Algebra ℂ E]
  [FiniteDimensional ℂ E] (T : E →1[ℂ] E) (x y : E) :
  ⟨⟨T (x * y), T x⟩⟩_ℂ
  = (LinearMap.adjoint (Algebra.linearMap ℂ E)) x
```

- ▶ And we're back to the type mismatch error... **UGH**

Identifying the second error: another

- ▶ Okay, so what's happening now?
- ▶ Module is created by
 1. InnerProductSpace and AddCommMonoid
 2. Algebra and AddCommMonoid

And they are not definitionally equal.
- ▶ So we have another non-transparent diamond  to resolve.
- ▶ Let's open the documentation of Algebra in Mathlib to see exactly how Algebra is defined. There's actually a whole section on implementation:
`Mathlib.Algebra.Algebra.Basic#Implementation-notes.`

► Implementation notes:

“There are two ways to talk about an R -algebra A when A is a semiring:

1. `variable [CommSemiring R] [Semiring A]`
`variable [Algebra R A]`
2. `variable [CommSemiring R] [Semiring A]`
`variable [Module R A] [SMulCommClass R A A]`
`[IsScalarTower R A A]`

Implementation notes in `Mathlib.Algebra.Algebra.Basic` (2)

- ▶ This means we can replace `[Algebra C E]` with `[SMulCommClass C E E] [IsScalarTower C E E]`, because `[Module C E]` is given by the inner product space, which then resolves our diamond.
- ▶ Although, this comes with a small caveat:
“Typeclass search does not know that the second approach implies the first, but this can be shown with:

```
example {R A : Type*} [CommSemiring R] [Semiring A]
  [Module R A] [SMulCommClass R A A] [IsScalarTower R A
  A] : Algebra R A :=
  Algebra.ofModule smul_mul_assoc mul_smul_comm
```

– `Mathlib.Algebra.Algebra.Basic#Implementation-notes`

- ▶ Let's attribute the above example as a local instance in our file, so that the second approach does imply the first.

Resolving the second diamond

- ▶ Now we have no errors and unresolved diamonds remaining!
- ▶ Module created by the algebra is definitionally equal¹ to that created by the inner product space:

```
example [NACGoR E] [h : InnerProductSpace  $\mathbb{C}$  E]
  [SMulCommClass  $\mathbb{C}$  E E] [IsScalarTower  $\mathbb{C}$  E E] :
  h.toModule = Algebra.toModule :=
  rfl
```

¹In Lean 3, the proof is `by ext; rfl`, which means the diamond would remain unresolved - this is another example of how Lean 4 is more powerful.