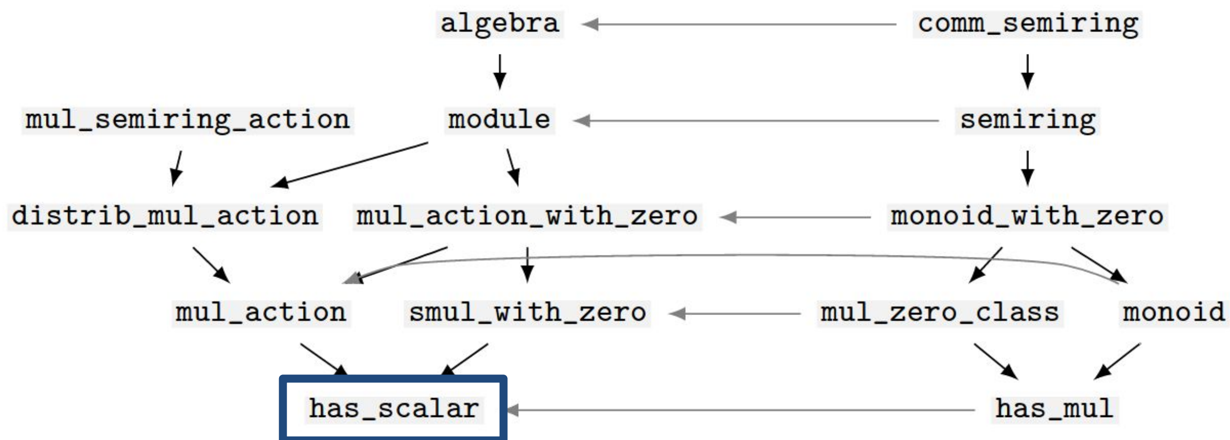
Scalar Actions in Lean's Mathlib

Steven Bronsveld & Jelmer Firet

We want to define **scalar multiplication** in Lean

Typeclass hierarchy



Multiplication (Lean primitive)

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

infix * := has_mulᵤ.mul
```

Examples

- $(\mathbb{N}, \cdot)$
- $(\mathbb{N}, +)$
- $\vec{v_2} * \vec{v_2}$
- $\begin{pmatrix} a & b \\ c & d \end{pmatrix} * \begin{pmatrix} a' & b' \\ c' & d' \end{pmatrix}$

Other primitives

```
class has_zero      (α : Type u) := (zero : α)
class has_one       (α : Type u) := (one : α)
class has_add       (α : Type u) := (add : α → α → α)
class has_mul       (α : Type u) := (mul : α → α → α)
```

Semigroup

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

Identity

```
class mul_one_class (M : Type u) extends has_one M, has_mul M :=
(one_mul : ∀ (a : M), 1 * a = a)
(mul_one : ∀ (a : M), a * 1 = a)
```

Monoid

```
class monoid (M : Type u) extends semigroup M, mul_one_class M :=
```

Scalar multiplication

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

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

Scalar multiplication

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

infixr ` • `:73 := has_scalartt.smul


instance has_mul.to_has_scalartt (α : Type*) [has_multt α] : has_scalartt α α := {
  smul := λ (a: α) (b: α), a * b
}


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

Repeated addition

```
def apply_n {α : Type*} [add_comm_monoid α] : ℕ → (α → α → α) → α → α
| 0     f x := 0
| (k+1) f x := f (apply_n k f x) x

instance repeated_add.to_has_scalarₗ (α : Type*)
[add_comm_monoid α] : has_scalarₗ ℕ α := {
  smul := λ (n: ℕ) (x: α), apply_n n has_add.add x
}
```

Overview

structure	0	1	$a \cdot b$	$a \cdot 0 = 0 \cdot a = 0$	$a \cdot 1 = 1 \cdot a = a$	$(a \cdot b) \cdot c = a \cdot (b \cdot c)$	$a \cdot b = b \cdot a$	$a + b$	$0 + a = a + 0 = a$	$(a + b) + c = a + (b + c)$	$a + b = b + a$	$(a + b) \cdot c = a \cdot c + b \cdot c$	$a \cdot (b + c) = a \cdot b + a \cdot c$
has_mul			✓										
mul_zero_class	✓		✓	✓									
monoid		✓	✓		✓	✓							
monoid_with_zero	✓	✓	✓	✓	✓	✓							
semiring	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓
comm_semiring	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Overview

structure	$r \bullet m$	$r \bullet 0_M = 0_M$	$0_R \bullet m = 0_M$	$1_R \bullet m = m$	$(r \cdot s) \bullet m = r \bullet (s \bullet m)$	$r \bullet (m + n) = r \bullet m + r \bullet n$	$(r + s) \bullet m = r \bullet m + s \bullet m$	$r \bullet m = r_M \cdot m = m \cdot r_M$	$r \bullet (m \cdot n) = (r \bullet m) \cdot (r \bullet n)$	$r \bullet 1_M = 1_M$
has_scalar	✓									
smul_with_zero	✓	✓	✓							
mul_action	✓			✓	✓					
mul_action_with_zero	✓	✓	✓	✓	✓					
distrib_mul_action	✓	✓		✓	✓	✓				
module	✓	✓	✓	✓	✓	✓	✓			
algebra	✓	✓	✓	✓	✓	✓	✓	✓		
mul_semiring_action	✓	✓		✓	✓	✓			✓	✓

Modules and Algebras

Define Module

- Generalization of a vector space e.g.
- In a vector space, we have a ground-field
- In a module we only have a ground-ring

$$\begin{pmatrix} a \\ b \end{pmatrix} + \begin{pmatrix} a' \\ b' \end{pmatrix} = \begin{pmatrix} a + a' \\ b + b' \end{pmatrix}$$

$$c \cdot \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} c * a \\ c * b \end{pmatrix}$$

Define Module

- Generalization of a vector space e.g.
- In a vector space, we have a ground-field
- In a module we only have a ground-ring

$$\begin{pmatrix} a \\ b \end{pmatrix} + \begin{pmatrix} a' \\ b' \end{pmatrix} = \begin{pmatrix} a + a' \\ b + b' \end{pmatrix}$$

$$c \cdot \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} c * a \\ c * b \end{pmatrix}$$

```
class moduleₜₜ (R M: Type*) [semiring R]
  [add_comm_monoid M] extends has_scalar R M :=
(one_smul : ∀ m : M, (1 : R) • m = m)
(mul_smul : ∀ (r s : R) (m : M), (r * s) • m = r • s • m)
(smul_add : ∀(r : R) (m n : M), r • (m + n) = r • m + r • n)
(smul_zero : ∀(r : R), r • (0 : M) = 0)
(add_smul : ∀(r s : R) (m : M), (r + s) • m = r • m + s • m)
(zero_smul : ∀m : M, (0 : R) • m = 0)
```

Define Algebra

```
class algebraₜₜ {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, has_scalar.smul r x = algebra_map r * x)
```

Define Algebra

```
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, has_scalar.smul r x = algebra_map r * x)

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 := algebra.of_module h₁ h₂
```

Define Algebra

```
class algebraₜₜ {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, has_scalar.smul r x = algebra_map r * x)

def algebra.of_moduleₜₜ {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 := algebra.of_module h₁ h₂
```

```
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]
```

Define Algebra

```
class algebraₜₜ {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, has_scalar.smul r x = algebra_map r * x)

def algebra.of_moduleₜₜ {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 := algebra.of_module h₁ h₂
```

```
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]
```

Define Algebra

```
class algrebra (R A : Type*) [comm_semiring R] [semiring A]
extends has_scalar R A :=
(algrebra_map : R →+* A)
(commutes : ∀r x, algrebra_map r * x = x * algrebra_map r)
(smul_def : ∀ r x, has_scalar.smul r x = algrebra_map r * x)

def algrebra.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)) : algrebra R A := sorry
```

```
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]
```

```
See note [reducible non-instances]. -/
@[reducible]
def of_module' [comm_semiring R] [semiring A] [module R A]
  (h₁ : ∀ (r : R) (x : A), (r • 1) * x = r • x)
  (h₂ : ∀ (r : R) (x : A), x * (r • 1) = r • x) : algebra R A :=
{ to_fun := λ r, r • 1,
  map_one' := one_smul _ _,
  map_mul' := λ r₁ r₂, by rw [h₁, mul_smul],
  map_zero' := zero_smul _ _,
  map_add' := λ r₁ r₂, add_smul r₁ r₂ 1,
  commutes' := λ r x, by simp only [h₁, h₂],
  smul_def' := λ r x, by simp only [h₁] }

/-- Let `R` be a commutative semiring, let `A` be a semiring with a `module R` structure.
If `(r • x) * y = x * (r • y) = r • (x * y)` for all `r : R` and `x y : A`, then `A`
is an `algebra` over `R`.

See note [reducible non-instances]. -/
@[reducible]
def of_module [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 :=
of_module' (λ r x, by rw [h₁, one_mul]) (λ r x, by rw [h₂, mul_one])
```

Relaxing algebras

```
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))
```

Relaxing algebras

```
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))

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)
```

Relaxing algebras

```
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))

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)
```

```
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]
```

Why would we want this?

- We can define variations:
- **Semiring** replaced by **Semiring without zero**
- This is not possible by using algebras, which require zero
- Same for using a **monoid** instead of a **commutative semiring**

```
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]
```

Derived Actions

Scalar action for functions (vectors, matrices, …)

```
variables R M N I J α : Type*

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

for $|I| = 2$ e.g.

$$2 \bullet \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \end{pmatrix}$$

Scalar action for functions (vectors, matrices, …)

```
variables R M N I J α : Type*

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

instance matrix.has_scalar' [has_mul α]
: has_scalar α (I → J → α) := { smul := λ r v, (λ i j, r * v i j) }
```

$$2 \bullet \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \end{pmatrix}$$

$$2 \bullet \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 2 & 4 \\ 6 & 8 \end{pmatrix}$$

Scalar action for functions (vectors, matrices, …)

```
variables R M N I J α : Type*

instance function.has_scalar_t t [has_mul α]
: has_scalar α (I → α) := { smul := λ r v, (λ i, r * v i) }

instance matrix.has_scalar_t t [has_mul α]
: has_scalar α (I → J → α) := { smul := λ r v, (λ i j, r * v i j) }

instance matrix.has_scalar'_t t [has_mul α] [has_scalar α (J → α)]
: has_scalar α (I → (J → α)) := { smul := λ r v, (λ i j, r * v i j) }
```

$$2 \bullet \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \end{pmatrix}$$

$$2 \bullet \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 2 & 4 \\ 6 & 8 \end{pmatrix}$$

Scalar action for functions (vectors, matrices, …)

```
variables R M N I J α : Type*

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

instance matrix.has_scalar_t_t [has_mul α]
: has_scalar α (I → J → α) := { smul := λ r v, (λ i j, r * v i j) }

instance matrix.has_scalar'_t_t [has_mul α] [has_scalar α (J → α)]
: has_scalar α (I → (J → α)) := { smul := λ r v, (λ i j, r * v i j) }

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

$$2 \bullet \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \end{pmatrix}$$

$$2 \bullet \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 2 & 4 \\ 6 & 8 \end{pmatrix}$$

Scalar action for functions (vectors, matrices, …)

```
variables R M N I J α : Type*

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

instance matrix.has_scalarₜₜ [has_mul α]
: has_scalar α (I → J → α) := { smul := λ r v, (λ i j, r * v i j) }

instance matrix.has_scalar'ₜₜ [has_mul α] [has_scalar α (J → α)]
: has_scalar α (I → (J → α)) := { smul := λ r v, (λ i j, r * v i j) }

instance function.has_scalar'ₜₜ [has_scalar α M]
: has_scalar α (I → M) := { smul := λ r v, (λ i, r • v i) }

instance pi.has_scalarₜₜ (f : I → Type*) [Π i : I, has_scalar α (f i)]
: has_scalar α (Π i : I, f i) := { smul := λ r v, (λ i, r • v i) }
```

$$2 \bullet \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \end{pmatrix}$$

$$2 \bullet \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 2 & 4 \\ 6 & 8 \end{pmatrix}$$

More complex derived actions

Scalar action for additive homomorphism

```
instance add_monoid_hom.has_scalar
[add_zero_class M] [add_zero_class N] [has_scalar R N]
: has_scalar R (M →+ N) := { -- has_scalar
```

Scalar action for additive homomorphism

```
instance add_monoid_hom.has_scalar
[add_zero_class M] [add_zero_class N] [has_scalar R N]
: has_scalar R (M →+ N) := { -- has_scalar
  smul := λ r f, { -- add_monoid_hom
    to_fun := λ i, r • f i,
    map_zero' := by { simp, sorry }, -- r • f 0 = 0
    map_add' := by { simp, sorry },  -- r • (f x + f y) = r • f x + r • f y
  }
}
```

Scalar action for additive homomorphism

```
instance add_monoid_hom.distrib_mul_action
[add_monoid M] [add_comm_monoid N] [monoid R] [distrib_mul_action R N]
: has_scalar R (M →+ N) := { -- distrib_mul_action
  smul := λr f, { -- add_monoid_hom
    to_fun := λ i, r • f i,                   -- funtion M → N
    map_zero' := by simp,                     -- r • f 0 = 0
    map_add' := λ x y, by simp [smul_add], -- r • (f x + f y) = r • f x + r • f y
  },
}
```

Scalar action for additive homomorphism

```
instance add_monoid_hom.distrib_mul_action
[add_monoid M] [add_comm_monoid N] [monoid R] [distrib_mul_action R N]
: distrib_mul_action R (M →+ N) := { -- distrib_mul_action
  smul := λr f, { -- add_monoid_hom
    to_fun := λ i, r • f i,                  -- funtion M → N
    map_zero' := by simp,                    -- r • f 0 = 0
    map_add' := λ x y, by simp [smul_add], -- r • (f x + f y) = r • f x + r • f y
  },
  one_smul := λ f, by simp,                  -- 1 • f = f
  mul_smul := λ r s f, by simp [mul_smul], -- (r * s) • f = r • s • f
  -- r • (f + g) = r • f + r • g
  smul_add := λ r f g, add_monoid_hom.ext (λ x, by simp [smul_add]),
  -- r • 0 = 0
  smul_zero := λ r, add_monoid_hom.ext (λ x, by simp [smul_zero]),
}
```

Scalar action of linear maps

```
instance linmap.has_scalar₁ [module R N] [module R M]
[has_scalar α N]
: has_scalar α (M →ₗ[R] N) := { -- has_scalar
  smul := λ a f, { -- linear map
    to_fun := λ m, a • f m, -- function M → N
    -- a • f (x + y) = a • f x + a • f y
    map_add' := by { intros m₁ m₂, rw f.map_add, sorry },
    -- a • r • f m = r • a • f m
    map_smul' := by { intros r m, simp, sorry }
  }
}
```

Scalar action of linear maps

```
instance linmap.has_scalar₂ [module R N] [module R M]
[monoid α] [distrib_mul_action α N]
: has_scalar α (M →ₗ[R] N) := { -- has_scalar
  smul := λ a f, { -- linear map
    to_fun := λ m, a • f m, -- function M → N
    -- a • f (x + y) = a • f x + a • f y
    map_add' := by { intros m₁ m₂, rw f.map_add, rw smul_add },
    -- a • r • f m = r • a • f m
    map_smul' := by { intros r m, simp, sorry }
  }
}
```

Scalar action of linear maps

```
instance linmap.has_scalar₃ [module α N] [module α M]

: has_scalar α (M →ₗ[α] N) := { -- has_scalar
  smul := λ a f, { -- linear map
    to_fun := λ m, a • f m, -- function M → N
    -- a • f (x + y) = a • f x + a • f y
    map_add' := by simp,
    -- a • r • f m = r • a • f m
    map_smul' := by { intros s m, simp,
      rw [ ←mul_smul, ← mul_smul, mul_comm],
    }
  }
}
```

Scalar action of linear maps

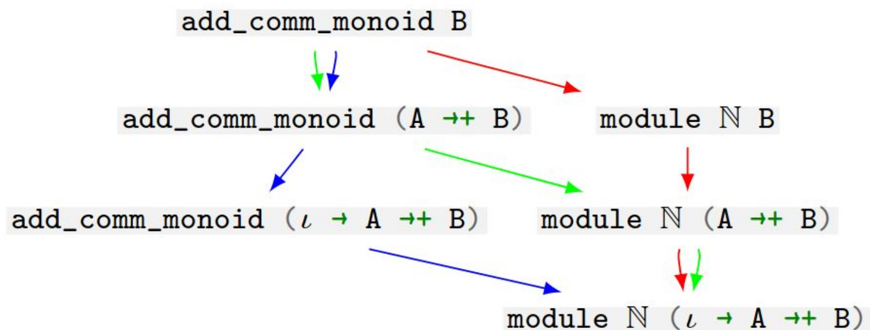
```
instance linmap.has_scalar₄ [module R N] [module R M]
[module α N] [algebra α R] [is_scalar_tower α R N]
: has_scalar α (M →ₗ[R] N) := { -- has_scalar
  smul := λ a f, { -- linear map
    to_fun := λ m, a • f m, -- function M → N
    -- a • f (x + y) = a • f x + a • f y
    map_add' := by simp,
    -- a • r • f m = r • a • f m
    map_smul' := by { intros r m, simp,
      rw [ ←smul_assoc, algebra.smul_def, algebra.commutes,
           mul_smul, algebra_map_smul ],
    }
  }
}
```

Scalar action of linear maps

```
instance linmap.has_scalar₅ [module R N] [module R M]
[monoid α] [distrib_mul_action α N] [smul_comm_class α R N]
: has_scalar α (M →ₗ[R] N) := {
  smul := λ a f, {
    to_fun := λ m, a • f m,
    -- a • f (x + y) = a • f x + a • f y
    map_add' := by simp,
    -- a • r • f m = r • a • f m
    map_smul' := by { intros r m, simp, rw smul_comm }
  }
}
```

Diamond resolution failure

```
lemma diamond_failure [add_comm_monoid B] {s : finset A} {c : B}
: ∑ x in s, c = s.card • c :=
begin
  apply finset.sum_const,
end
```



▼ Messages (1)

▼ scalar_actions.lean:218:23

```
ambiguous overload, possible interpretations
  s.card • c
  s.card • c
Additional information:
```

Diamond resolution failure

```
@[class]                                                          source
structure add_monoid (M : Type u) :
  Type u

(add : M → M → M)
(add_assoc : ∀ (a b c : M), a + b + c = a + (b + c))
(zero : M)
(zero_add : ∀ (a : M), 0 + a = a)
(add_zero : ∀ (a : M), a + 0 = a)
(nsmul : ℕ → M → M)
(nsmul_zero' : (∀ (x : M), add_monoid.nsmul 0 x = 0) . "try_refl_tac")
(nsmul_succ' :
  (∀ (n : ℕ) (x : M), add_monoid.nsmul n.succ x = x + add_monoid.nsmul n x) . "try_refl_tac")
```

An `add_monoid` is an `add_semigroup` with an element `0` such that `0 + a = a + 0 = a`.

▸ Instances

Future Work

- Right actions
- Further improvement on diamonds

Are there questions?

Random parrot from the internet

Some other parrots