BUILDING AN ALGEBRAIC HIERARCHY

PLAN

1. Monoids, Groups, Rings, Modules
2. Morphisms
3. Subobjects and quotients

1. MONOIDS, GROUPS, RINGS, MODULES

SEMIGROUPS

- We'll prove the multiplicative lemmas, *Lean* generates the additive versions.
- *For now*, we need to declare the classes.

```
class AddSemigroup (α : Type) extends Add α where
  add_assoc : ∀ a b c : α, a + b + c = a + (b + c)
```

```
@[to_additive]
class Semigroup (α : Type) extends Mul α where
  mul_assoc : ∀ a b c : α, a * b * c = a * (b * c)
```

COMMUTATIVE SEMIGROUPS

```
class AddCommSemigroup (α : Type) extends AddSemigroup α where
  add_comm : ∀ a b : α, a + b = b + a
```

```
@[to_additive]
class CommSemigroup (α : Type) extends Semigroup α where
  mul_comm : ∀ a b : α, a * b = b * a
```

MONOIDS

There is a class

```
class MulOneClass (α : Type u) extends One α, Mul α where
  one_mul : ∀ a : α, 1 * a = a
  mul_one : ∀ a : α, a * 1 = a
```

and its corresponding `AddZeroClass`. From it, we can create monoids:

```
class Monoid (α : Type) extends Semigroup α, MulOneClass α
```

> ***Magic:*** *there is a `Mul` coming from `Semigroup` and one coming fom `MulOneClass`.*
> *The `extends` keyword ensures that these two are the same!*

```
#check Monoid.mk
```

```
Monoid.mk (α : Type) [toSemigroup : Semigroup α] [toOne : One α] (one_mul: ∀ (a : α),
```

```
1 * a = a)
  (mul_one : ∀ (a : α), a * 1 = a) : Monoid α
```

We seamlessly make the additive versions:

```
class AddMonoid (α : Type) extends AddSemigroup α, AddZeroClass α

-- Tag to link to the multiplicative versions
attribute [to_additive] Monoid
attribute [to_additive existing] Monoid.toMulOneClass -- Does not get created
automatically
```

and lemmas do get translated automatically:

```
@[to_additive]
lemma left_inv_eq_right_inv {M : Type} [Monoid M] {a b c : M} (hba : b * a = 1) (hac :
a * c = 1) : b = c := by
  rw [← one_mul c, ← hba, mul_assoc, hac, mul_one b]

#check left_neg_eq_right_neg
--  left_neg_eq_right_neg {M : Type} [AddMonoid M] {a b c : M} (hba : b + a = 0) (hac
: a + c = 0) : b = c
```

COMMUTATIVE MONOIDS

```
class AddCommMonoid (M : Type) extends AddMonoid M, AddCommSemigroup M

@[to_additive]
class CommMonoid (M : Type) extends Monoid M, CommSemigroup M
```

GROUPS, COMMUTATIVE GROUPS

```
class AddGroup (G : Type) extends AddMonoid G, Neg G where
  neg_add : ∀ a : G, -a + a = 0

@[to_additive]
class Group (G : Type) extends Monoid G, Inv G where
  mul_left_inv : ∀ a : G, a⁻¹ * a = 1
```

```
class AddCommGroup (G : Type) extends AddGroup G, AddCommMonoid G where

@[to_additive]
class CommGroup (G : Type) extends Group G, CommMonoid G
```

RINGS

We can now define rings:

```
class Ring (R : Type) extends AddGroup R, Monoid R, MulZeroClass R where
  /- Multiplication is left distributive over addition -/
  left_distrib : ∀ a b c : R, a * (b + c) = a * b + a * c
  /- Multiplication is right distributive over addition -/
  right_distrib : ∀ a b c : R, (a + b) * c = a * c + b * c
```

Forgetting the multiplication yields an (additive) **commutative** group.

```
instance {R : Type} [Ring R] : AddCommGroup R :=
{ Ring.toAddGroup with
  add_comm := by sorry -- this is a FUN exercise!
}
```

THE INTEGERS FORM RING

We can "easily" prove that ℤ is a ring.

```
instance : Ring ℤ where
  add := (· + ·)
  add_assoc := _root_.add_assoc -- cheating
  zero := 0
  zero_add := by simp
  add_zero := by simp
  neg := (-(·))
  neg_add := by simp
  mul := (· * ·)
  mul_assoc := _root_.mul_assoc -- cheating
  one := 1
  one_mul := by simp
  mul_one := by simp
  zero_mul := by simp
  mul_zero := by simp
```

```
left_distrib := Int.mul_add -- cheating
right_distrib := Int.add_mul -- cheating
```

MODULES (E.G. VECTOR SPACES)

These involve several types: commutative additive groups equipped with a scalar multiplication by elements of some ring.

```
class SMul (α : Type) (β : Type) where
  smul : α → β → β

infixr:73 " • " => SMul.smul
```

```
class Module (R : Type) [Ring R] (M : Type) [AddCommGroup M] extends SMul R M where
  zero_smul : ∀ m : M, (0 : R) • m = 0
  one_smul : ∀ m : M, (1 : R) • m = m
  mul_smul : ∀ (a b : R) (m : M), (a * b) • m = a • b • m
  add_smul : ∀ (a b : R) (m : M), (a + b) • m = a • m + b • m
  smul_add : ∀ (a : R) (m n : M), a • (m + n) = a • m + a • n
```

Note that `Smul R M` is in `extends`, while `AddCommGroup M` is not!

- This is so that the class inference system stays sane. Otherwise, it would keep looking for some type `R` with `Ring R`.
- No problem with `Smul R M` since it mentions both `R` and `M`.

> ***Rule:*** *Each class appearing in the `extends` clause should mention every type appearing in the parameters.*

EXAMPLE: A RING IS A MODULE OVER ITSELF

```
instance selfModule (R : Type) [Ring R] : Module R R where
  smul := fun r s ↦ r*s
  zero_smul := zero_mul
  one_smul := one_mul
  mul_smul := mul_assoc
  add_smul := Ring.right_distrib
  smul_add := Ring.left_distrib
```

EXAMPLE: AN ABELIAN GROUP IS A ℤ-MODULE.

First, we define multiplication by naturals and by integers.

```
def nsmul [Zero M] [Add M] : ℕ → M → M
  | 0, _ => 0
  | n + 1, a => a + nsmul n a

def zsmul {M : Type} [Zero M] [Add M] [Neg M] : ℤ → M → M
  | Int.ofNat n, a => nsmul n a
  | Int.negSucc n, a => -(nsmul n.succ a)
```

Filling the `sorry`'s below is tedious but definitely doable!

```
instance abGrpModule (A : Type) [AddCommGroup A] : Module ℤ A where
  smul := zsmul
  zero_smul := by simp [zsmul, nsmul]
  one_smul := by simp [zsmul, nsmul]
  mul_smul := sorry
```

```
add_smul := sorry
smul_add := sorry
```

PROBLEM!!

We have two module structures for the ring ℤ over ℤ itself:

1. `abGrpModule ℤ`, since ℤ is a abelian group, and
2. `selfModule ℤ`, since ℤ is a ring.

The fact that these two coincide is a *theorem*.

- Not all diamonds are bad (e.g. `Prop`-valued classes).
- But `smul` is data: two constructions not defeq.

> ***Easy Rule:*** *always make sure that going from a rich structure to a poor structure is always done by forgetting data, not by defining it.*

SOLUTION

We can modify the definition of `AddMonoid` to include a `nsmul` data field and some `Prop`-valued fields ensuring this operation is provably the one we constructed above.

- Can be given default values using `:=` after their type in the definition below.
- Most instances would be constructed exactly as with our previous definitions.
- In the special case of ℤ we will be able to provide specific values.

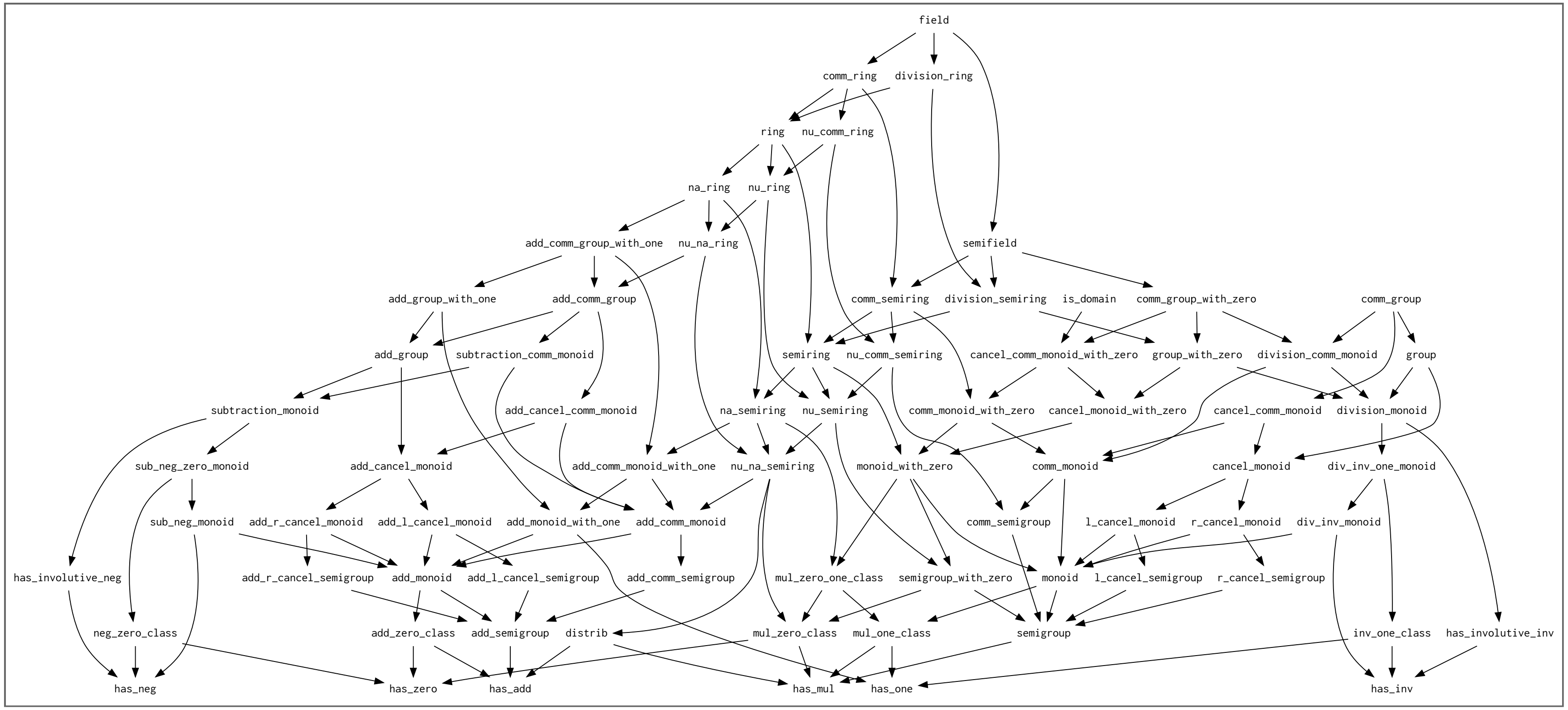
```
class AddMonoid' (M : Type) extends AddSemigroup M, AddZeroClass M where
  /- Multiplication by a natural number. -/
  nsmul : ℕ → M → M := nsmul
  /- Multiplication by `(0 : ℕ)` gives `0`. -/
  nsmul_zero : ∀ x, nsmul 0 x = 0 := by intros; rfl
  /- Multiplication by `(n + 1 : ℕ)` behaves as expected. -/
  nsmul_succ : ∀ (n : ℕ) (x), nsmul (n + 1) x = x + nsmul n x := by intros; rfl
```

```
instance {M : Type} [AddMonoid' M] : SMul ℕ M := ⟨AddMonoid'.nsmul⟩
```

Finally we can prove that ℤ is an additive monoid.

```
instance : AddMonoid' ℤ where
  add := (· + ·)
  add_assoc := Int.add_assoc
  zero := 0
  zero_add := Int.zero_add
  add_zero := Int.add_zero
  nsmul := fun n m ↦ (n : ℤ) * m
  nsmul_zero := Int.zero_mul
  nsmul_succ := fun n m ↦ show (n + 1 : ℤ) * m = m + n * m
    by rw [Int.add_mul, Int.add_comm, Int.one_mul]
```

You are now ready to read the definition of monoids, groups, rings and modules in **Mathlib**.
There are a bit more complicated than what we have seen here because they are part of a huge hierarchy. But all principles have been explained above.


field
comm_ring
division_ring
ring
nu_comm_ring
na_ring
nu_ring
add_comm_group_with_one
nu_na_ring
semifield
add_group_with_one
add_comm_group
comm_semiring
division_semiring
is_domain
comm_group_with_zero
comm_group
add_group
subtraction_comm_monoid
semiring
nu_comm_semiring
cancel_comm_monoid_with_zero
group_with_zero
division_comm_monoid
group
subtraction_monoid
add_cancel_comm_monoid
na_semiring
nu_semiring
comm_monoid_with_zero
cancel_monoid_with_zero
cancel_comm_monoid
division_monoid
sub_neg_zero_monoid
add_cancel_monoid
add_comm_monoid_with_one
nu_na_semiring
monoid_with_zero
comm_monoid
cancel_monoid
div_inv_one_monoid
sub_neg_monoid
add_r_cancel_monoid
add_l_cancel_monoid
add_monoid_with_one
add_comm_monoid
comm_semigroup
l_cancel_monoid
r_cancel_monoid
div_inv_monoid
has_involutive_neg
add_r_cancel_semigroup
add_monoid
add_l_cancel_semigroup
add_comm_semigroup
mul_zero_one_class
semigroup_with_zero
monoid
l_cancel_semigroup
r_cancel_semigroup
neg_zero_class
add_zero_class
add_semigroup
distrib
mul_zero_class
mul_one_class
semigroup
inv_one_class
has_involutive_inv
has_neg
has_zero
has_add
has_mul
has_one
has_inv

2. MORPHISMS

VIA PREDICATE / STRUCTURE

Could simply define a predicate

```
def isMonoidHom_naive [Monoid G] [Monoid H] (f : G → H) : Prop :=
  f 1 = 1 ∧ ∀ g g', f (g * g') = f g * f g'
```

Or a structure:

```
structure isMonoidHom [Monoid G] [Monoid H] (f : G → H) : Prop where
  map_one : f 1 = 1
  map_mul : ∀ g g', f (g * g') = f g * f g'
```

No good to make it into a class, though.

BUNDLED MORPHISMS

```
@[ext]
structure MonoidHom (G H : Type) [Monoid G] [Monoid H]  where
  toFun : G → H
  map_one : toFun 1 = 1
  map_mul : ∀ g g', toFun (g * g') = toFun g * toFun g'
```

Allow it to coerce to a function.

```
instance [Monoid G] [Monoid H] : CoeFun (MonoidHom G H) (fun _ ↦ G → H) where
  coe := MonoidHom.toFun
```

Print the "almost invisible" arrow instead of the `toFun`

```
attribute [coe] MonoidHom.toFun
```

```
example [Monoid G] [Monoid H] (f : MonoidHom G H) : f 1 = 1 :=  f.map_one
```

The additive version is done similarly.

```
@[ext]
structure AddMonoidHom (G H : Type) [AddMonoid G] [AddMonoid H]  where
  toFun : G → H
  map_zero : toFun 0 = 0
  map_add : ∀ g g', toFun (g + g') = toFun g + toFun g'

instance [AddMonoid G] [AddMonoid H] : CoeFun (AddMonoidHom G H) (fun _ ↦ G → H)
where
  coe := AddMonoidHom.toFun

attribute [coe] AddMonoidHom.toFun
```

```
@[ext]
structure RingHom (R S : Type) [Ring R] [Ring S] extends MonoidHom R S, AddMonoidHom R S
```

- Where do we put the `coe` attribute? Because `RingHom.toFun` does not exist...
- We'd like to have lemmas about monoid morphisms apply to ring morphisms.

> ***Solution:*** *A new type class, for objects which are at least monoid morphisms.*

```lean
class MonoidHomClass (F : Type)
(M N : outParam Type) [Monoid M] [Monoid N] where
  toFun : F → M → N
  map_one : ∀ f : F, toFun f 1 = 1
  map_mul : ∀ f g g', toFun f (g * g') = toFun f g * toFun f g'

instance [Monoid M] [Monoid N] [MonoidHomClass F M N] : CoeFun F (fun _ ↦ M → N)
where
  coe := MonoidHomClass.toFun

attribute [coe] MonoidHomClass.toFun
```

The `outParam` tells Lean to resolve first for `F`, and only later for `M` and `N`.

Now we make both `MonoidHom` and `RingHom` instances of the new class.

```
instance (M N : Type) [Monoid M] [Monoid N] : MonoidHomClass (MonoidHom M N) M N where
  toFun := MonoidHom.toFun
  map_one := fun f ↦ f.map_one
  map_mul := fun f ↦ f.map_mul

instance (R S : Type) [Ring R] [Ring S] : MonoidHomClass (RingHom R S) R S where
  toFun := fun f ↦ f.toMonoidHom.toFun
  map_one := fun f ↦ f.toMonoidHom.map_one
  map_mul := fun f ↦ f.toMonoidHom.map_mul
```

We see our new infrastructure in action.
After proving one lemma:

```
lemma map_inv_of_inv [Monoid M] [Monoid N] [MonoidHomClass F M N]
(f : F) {m m' : M} (h : m*m' = 1) : f m * f m' = 1 := by
  rw [← MonoidHomClass.map_mul, h, MonoidHomClass.map_one]
```

we can apply it to both `MonoidHom`s and to `RingHom`s,

```
example [Monoid M] [Monoid N] (f : MonoidHom M N)
{m m' : M} (h : m*m' = 1) : f m * f m' = 1 :=
map_inv_of_inv f h

example [Ring R] [Ring S] (f : RingHom R S)
{r r' : R} (h : r*r' = 1) : f r * f r' = 1 :=
map_inv_of_inv f h
```

***Better:** use `FunLike` base class instead of `toFun` field. This sets up the coercion automatically, and records the fact a morphism is a function with extra properties.*

```
class MonoidHomClass (F : Type) (M N : outParam Type) [Monoid M] [Monoid N] extends
    FunLike F M (fun _ ↦ N) where
  map_one : ∀ f : F, f 1 = 1
  map_mul : ∀ (f : F) g g', f (g * g') = f g * f g'

instance (M N : Type) [Monoid M] [Monoid N] : MonoidHomClass (MonoidHom M N) M N where
  coe := MonoidHom.toFun
  coe_injective' := MonoidHom.ext
  map_one := MonoidHom.map_one
  map_mul := MonoidHom.map_mul
```

3. SUBOBJECTS AND QUOTIENTS

Sub-objects are functions satisfying a certain predicate. Can recycle the ideas that led to `FunLike`, using:

> ***`SetLike`* class**: *wraps an injection into a `Set` type and defines the corresponding coercion and membership instance.*

```
@[ext]
structure Submonoid (M : Type) [Monoid M] where
  /- The carrier of a submonoid. -/
  carrier : Set M
  /- The product of two elements of a submonoid belongs to the submonoid. -/
  mul_mem {a b} : a ∈ carrier → b ∈ carrier → a * b ∈ carrier
  /- The unit element belongs to the submonoid. -/
  one_mem : 1 ∈ carrier

/- Submonoids in M can be seen as sets in M. -/
instance [Monoid M] : SetLike (Submonoid M) M where
  coe := Submonoid.carrier
  coe_injective' := Submonoid.ext
```

```
example [Monoid M] (N : Submonoid M) : 1 ∈ N := N.one_mem
example [Monoid M] (N : Submonoid M) (α : Type) (f : M → α) := f '' N
example [Monoid M] (N : Submonoid M) (x : N) : (x : M) ∈ N := x.property
```

MONOID STRUCTURE

A submonoid should also be a monoid.

```
instance SubMonoidMonoid [Monoid M] (N : Submonoid M) : Monoid N where
  mul := fun x y ↦ ⟨x*y, N.mul_mem x.property y.property⟩
  mul_assoc := fun x y z ↦ SetCoe.ext (mul_assoc (x : M) y z)
  one := ⟨1, N.one_mem⟩
  one_mul := fun x ↦ SetCoe.ext (one_mul (x : M))
  mul_one := fun x ↦ SetCoe.ext (mul_one (x : M))
```

We also need a class for structures that are at least submonoids:

```
class SubmonoidClass (S : Type) (M : Type) [Monoid M] [SetLike S M] : Prop where
  mul_mem : ∀ (s : S) {a b : M}, a ∈ s → b ∈ s → a * b ∈ s
  one_mem : ∀ s : S, 1 ∈ s
```

```
instance [Monoid M] : SubmonoidClass (Submonoid M) M where
  mul_mem := Submonoid.mul_mem
  one_mem := Submonoid.one_mem
```

In the exercises, you will define a `Subgroup` structure, endow it with a `SetLike` instance and a `SubmonoidClass` instance, put a `Group` instance on the subtype associated to a `Subgroup` and define a `SubgroupClass` class.

In Mathlib, sub-objects form a **complete lattice**. Here's how you do the intersection of two submonoids.

```
instance [Monoid M] : Inf (Submonoid M) :=
  ⟨fun S₁ S₂ ↦
    { carrier := S₁ ∩ S₂
      one_mem := ⟨S₁.one_mem, S₂.one_mem⟩
      mul_mem := fun ⟨hx, hx'⟩ ⟨hy, hy'⟩ ↦ ⟨S₁.mul_mem hx hy, S₂.mul_mem hx' hy'⟩
}⟩
```

```
example [Monoid M] (N P : Submonoid M) : Submonoid M := N ⊓ P
```

QUOTIENTS

The main device is the `HasQuotient` class.
It allows notations like `M ⧸ N` (type it with `\quot`).

```
def Submonoid.Setoid [CommMonoid M] (N : Submonoid M) : Setoid M  where
  r := fun x y ↦ ∃ w ∈ N, ∃ z ∈ N, x*w = y*z
  iseqv := {
    refl := fun x ↦ ⟨1, N.one_mem, 1, N.one_mem, rfl⟩
    symm := fun ⟨w, hw, z, hz, h⟩ ↦ ⟨z, hz, w, hw, h.symm⟩
    trans := sorry
  }
```

```
instance [CommMonoid M] : HasQuotient M (Submonoid M) where
  quotient' := fun N ↦ Quotient N.Setoid
```

```
def QuotientMonoid.mk [CommMonoid M] (N : Submonoid M) : M → M ⧸ N := Quotient.mk
N.Setoid
```

...AND THAT'S ALL

- Download the **slides** at `mmasdeu.github.io/slideslftcm2023`.
- Exercises at `LftCM/C07_Algebraic_Hierarchy`.
- **Thank you!**