Day - 12 : Sets

Touseef Haider

Study Mathematics In Lean(MIL) Section 4.1

```
import Mathlib.Data.Set.Lattice
import Mathlib.Data.Nat.Prime.Basic
import MIL.Common

-- Sets as Functions: In Lean4, Set α is just a fancy name for a α → Prop function.
-- This means a set is a function. You give it an element (of type α) and it gives you back a proposition (true or false). If it returns true, the element is in the set; if it returns false, the element is not in the set.

-- The symbol for ``is an element of`` is ∈. Since a set s is just a function α → Prop, checking if x is in s is the same as applying the function s to x. So in Lean4, we write x ∈ s as s x.

-- def EvenNumbers : Set ℕ := Even

namespace C04S01
def EvenNumbers : Set ℕ := Even

-- That is the EvenNumbers set, which is the set of all even natural numbers. If we want to check say 4 is in this set, we can write 4 ∈ EvenNumbers, which is the same as Even 4. If we write 5 ∈ EvenNumbers, it is the same as Even 5, which is false.

-- {x : α | P x} is a notation for the set of all x in α such that P x is true. In Lean4, this is just a function that takes an element of type α and returns true if P x is true, and false otherwise.


-- 1. Empty Set: The empty set is a set that contains no elements. In Lean4, it symbol is ∅. It is defined as the set of all x such that P x is false for any property P. In Lean4, we can write it as {x : α | False} or simply ∅.

-- 2. Universal Set: The universal set is the set that contains all elements of a given type. In Lean4, it is denoted by univ. It is defined as the set of all x such that P x is true for any property P. In Lean4, we can write it as {x : α | True} or simply univ.

section
variable {α : Type*}
variable (s t u : Set α)
open Set

#check Even

-- If set s is a subset of t, then the intersection of s with any set u is a subset of the intersection of t with u.



example (h : s ⊆ t) : s ∩ u ⊆ t ∩ u := by
  -- subset_def is a lemma that states that A ⊆ B means for all x, x ∈ A →  x ∈ B. So the main goal s ∩ u ⊆ t ∩ u is equivalent to saying for all x, if x ∈ s ∩ u, then x ∈ t ∩ u.
  rw [subset_def]
   -- inter_def this rewrites x ∈ A ∩ B as x ∈ A ∧ x ∈ B.
  rw [inter_def]
```

```
  rw [inter_def]
  -- The goal becomes for all x, if x ∈ s ∧ x ∈ u, then x ∈ t ∧ x ∈ u.
  rw [subset_def] at h
  -- This rewrites our hypothesis h using the subset definition. h becomes for all x, if x
    ∈ s, then x ∈ t.


  -- mem_setOf usually cleans up definition related to set-builder notation {...|...}. In this
    context, it might not change much visually, but it ensures the membership terms (∈) are
    in a basic form that rintro can easily handle.
  simp only [mem_setOf]


  -- It says let x be an arbitrary element, ⟨ xs, xu ⟩  says that assume the part before the
     → which is x ∈ s ∧ x ∈ u. Because it is an AND statement, it destructures it into two
    parts: xs and xu, which are the proofs that x is in s and u respectively.
  rintro x ⟨ xs, xu ⟩
    -- exact means our goal is exactly ⟨ h _ xs, xu ⟩. Since our goal is x ∈ t ∧ x ∈ u, we
    need to provide a proof for the left side and a proof for the right side.
  -- Left side (x ∈ t) is h _ xs. We apply h to our x (the _ lets Lean figure out itself)
    and then give it xs (our proof that x ∈ s). The result is a proof that x ∈ t.
-- Right side (x ∈ u) is xu, which we already have from our assumption that x ∈ s ∧ x ∈ u.
  exact ⟨ h _ xs, xu ⟩

example (h : s ⊆ t) : s ∩ u ⊆ t ∩ u := by
  simp only [subset_def, mem_inter_iff] at *
  rintro x ⟨ xs, xu⟩
  exact ⟨h _ xs, xu⟩

example (h : s ⊆ t) : s ∩ u ⊆ t ∩ u := by
  intro x xsu
  exact ⟨h xsu.1, xsu.2⟩


-- Term mode proof.
example (h : s ⊆ t) : s ∩ u ⊆ t ∩ u :=
  fun _ ⟨ xs, xu ⟩ ↦ ⟨ h xs, xu ⟩
-- fun _: This part directly addresses the ∀ x.
-- ⟨ xs, xu ⟩: This part destructures the assumption x ∈ s ∩ u into two parts: xs (x ∈ s)
    and xu (x ∈ u).
-- ⟨ ⟩ we build the AND proof by providing a proof for each side.
-- h xs : This is the proof for x ∈ t, h is our hypothesis that s ⊆ t. In term mode, Lean
    is smart enough to know that s ⊆ t means it can act like a function that takes a proof
    of x ∈ s and gives back a proof of x ∈ t.
-- xs is our proof that x ∈ s, so h xs gives us the proof that x ∈ t.
-- xu: This is the proof for x ∈ u, which we already have from our assumption that x ∈ s ∩
    u.
-- ↦ ⟨h xs, xu⟩: This part constructs the proof that x ∈ t ∩ u by applying h to xs (to get
    x ∈ t) and using xu directly (since x ∈ u is already given).



example : s ∩ t ∪ s ∩ u ⊆ s ∩ (t ∪ u) := by
  rintro x h_mem
  -- Now our x : α is introduced and we have a proof h_mem that x is in s ∩ t ∪ s ∩ u.
  -- When you have a hypothese that is an Or statement, you can use cases to split it into
    two cases.
  rcases h_mem with h_st | h_su
  -- Case 1: h_st means x is in s ∩ t.
  -- We need to show that x is in s ∩ (t ∪ u).
  -- Case 2: h_su means x is in s ∩ u.
  -- We need to show that x is in s ∩ (t ∪ u).
  -- Solve Case 1
  -- We know h_st : x ∈ s ∩ t. This means x ∈ s ∧ x ∈ t. We can break this down into two
    parts using rcases.
  rcases h_st with ⟨ hs, ht ⟩
  -- Now we have hs : x ∈ s and ht : x ∈ t.
```

```
  -- We need to show that x is in s ∩ (t ∪ u).
  -- We need to build this AND statement. We need a proof for x ∈ s (we have hs) and a proof for x ∈ t ∪ u.
  -- How can we prove x ∈ t ∪ u? Since we know ht : x ∈ t, we can use this directly. So we can use exact ⟨ hs , Or.inl ht ⟩.
  exact ⟨hs, Or.inl ht⟩
  -- Solve Case 2
  -- We know h_su : x ∈ s ∩ u. This means x ∈ s ∧ x ∈ u. We can break this down into two parts using rcases.
  rcases h_su with ⟨ hs, hu ⟩
  -- Now we have hs : x ∈ s and hu : x ∈ u.
  -- We need to show that x is in s ∩ (t ∪ u).
  -- We need to build this AND statement. We need a proof for x ∈ s (we have hs) and a proof for x ∈ t ∪ u.
  -- How can we prove x ∈ t ∪ u? Since we know hu : x ∈ u, we can use this directly. So we can use exact ⟨ hs , Or.inr hu ⟩.
  exact ⟨ hs, Or.inr hu⟩



-- Indexed Families of Sets
-- Imagine you have a collection of sets, but instead of just a few, you might have a whole sequence or even a more complex collection. How do you keep track of all these sets?
-- Index Set I : This is just any set (in Lean4 any Type*) whose elements we use as labels or indices. It could be natural numbers, integers, or any other type.
-- Index Family A : This is essentially a function. It takes an index i from your index set I and gives you back a set, which we call A_i. In Lean4, we write this as A : I → Set α. This means for each i in I, A i is a set of elements of type α.

-- Let I = {1, 2, 3 }, then A : I → Set ℕ could be defined as:
-- A 1 = {1,2} , A 2 = {2, 3}, A 3 = {3,4}.
-- Here A is the function that maps 1 to {1,2}, 2 to {2,3}, and 3 to {3,4}. So A 1 is the set {1,2}, A 2 is the set {2,3}, and A 3 is the set {3,4}.

-- Indexed Union ⋃ i, A i : Indexed union is like taking all the elements from all the sets in your family and putting them into one big set. In Lean4, we write this as ⋃ i, A i. It means for every index i in I, we take the set A i and combine all those sets into one big set.
-- Indexed Intersection ⋂ i, A i : Indexed intersection is like finding the common elements that are in every set in your family. In Lean4, we write this as ⋂ i, A i. It means for every index i in I, we take the set A i and find the elements that are in all those sets.

-- ⋃ i A i = A 1 ∪ A 2 ∪ A 3 = {1,2} ∪ {2,3} ∪ {3,4} = {1,2,3,4}
-- ⋂ i A i = A 1 ∩ A 2 ∩ A 3 = {1,2} ∩ {2,3} ∩ {3,4} = {}

section
variable {α I : Type*}
variable (A B : I → Set α)
variable (s : Set α)

open Set

example : (s ∩ ⋃ i, A i) = ⋃ i, A i ∩ s := by
  -- This is a set equality, it says two sets are equal if and only if they contain the same elements. So instead of proving the sets are equal directly, we prove that for every x, x is in the left set if and only if x is in the right set.
  ext x
  -- Our goal is now to show that x ∈ (s ∩ ⋃ i, A i) if and only if x ∈ ⋃ i, A i ∩ s.
  simp only [mem_inter_iff, mem_iUnion]
  -- We tell Lean to simplify only the definitions for set intersection (mem_inter_iff)) which says x ∈ A ∩ B if and only if x ∈ A and x ∈ B, and the definition for indexed union (mem_iUnion) which says x ∈ ⋃ i, A i if and only if there exists an index i such that x ∈ A i.
  -- Applying these rules, our goal become x ∈ s ∧ (∃ i, x ∈ A i) if and only if there exists an index i such that x \in A i ∧ x ∈ s.
  constructor
```

```
  -- constructor split the goal into two parts, one for each direction of the if and only if .
  -- The first part is to show that if x is in the left set, then it is also in the right set.
  -- The second part is to show that if x is in the right set, then it is also in the left set.
  -- We are taking the first sub-goal (indicated by .) rintro introduces the premise (the left side of → ) as a hypothesis and break it down. The premise is x ∈ s ∧  ∃  i, x ∈ A i.
  · rintro ⟨ xs, ⟨i, xAi⟩ ⟩
    -- xs is a proof that x ∈ s.
    -- ⟨i, xAi⟩ is a proof that there exists an index i such that x ∈ A i.
    -- We have the exact proof from previous step so we use exact to show that x ∈ A i ∩ s.
    exact ⟨i, xAi, xs⟩
  rintro ⟨i, xAi, xs⟩ -- This is the second sub-goal, we are taking the second part of the constructor.
  exact ⟨xs, ⟨i, xAi⟩ ⟩



example : (⋂ i, A i ∩ B i) = (⋂ i, A i) ∩ ⋂ i, B i := by
  ext x -- We are proving that two sets are equal by showing that for every x, x is in the left set if and only if x is in the right set.
  simp only [mem_inter_iff, mem_iInter]
  -- We simplify the definition of indexed intersection (mem_iInter) which says x ∈ ⋂ i, A i if and only if for all i, x ∈ A i.
  -- Applying this rule, our goal become ∀ i : I , x ∈ A i ∧ ∀ i : I, x ∈ B i.
  constructor
  -- We split the goal into two parts, one for each direction of the if and only if.
  · intro h -- We introduce the premise (the left side of → ) as a hypothesis h.
    -- Our goal is now ∀ i , x ∈ A i ∧ ∀ i, x ∈ B i.
    constructor -- Since we have two parts to prove, we use constructor to split the goal into two parts.
    · intro i
      exact (h i).1 -- We use our hypothesis h, since h tells us ∀ i, x ∈ A i ∧ x ∈ B i, applying it to our i(h i) givues us x ∈ A i ∧ x ∈ B i. This is the end statement, we only need the left part which we get by (h i).1. So (h i).1 is a proof that x ∈ A i, which is exactly what we need.
    intro i
    exact (h i).2
    -- This time we need the right part of h i, which we get using .2, So (h i).2 is a proof that x ∈ B i, which is exactly what we need. This complete the first part of the constructor.
  rintro ⟨ h1, h2⟩ i
  -- Now we tackle the second part of the constructor, which is to show that if x is in the right set, then it is also in the left set.
  -- h1 is proof that ∀ i x ∈ A i and h2 is proof that ∀ i x ∈ B i.
  -- We need to show that for every i, x is in A i ∧  B i.
  -- Our goal has ∧ so we can use constructor to split it into two parts.
  constructor
  · exact h1 i -- This gives us a proof that x ∈ A i, which is exactly what we need for the left part of the intersection.
  exact h2 i -- This gives us a proof that x ∈ B i, which is exactly what we need for the right part of the intersection.
```