Sets

Introduction

Sets are **primitive** objects when doing classical, old-school, pen-and-paper mathematics:

- no *definition*;
- only *rules* about how these objects work (unions, intersections, etc.).

That's all you need: do you look at $f\colon S \to T$ as $f\subseteq S\times T$?

Objects normally represented by a set are formalised in Lean as *types* with some extra-structure.

So, for Lean, sets are **no longer primitive objects**; yet

- sometimes we still want to speak about *sets* as collections of elements
- we want then to play the usual games.

Definitions

+++ Every set lives in a given type: it is a set of elements (*terms*) of a type:

```
variable (α : Type) (S : Set α)
```

expresses that `α` is a type and `S` is a set of elements/terms of the type `α`. On the other hand,

```
variable (S : Set)
```

does not mean "let `S` be a set": it means nothing and it is an error.
+++

+++ A set coincides with the test-function defining it.

Given a type `α`, a set `S` (of elements/terms of `α`) is a *function*

```
S : α → Prop
```

so `(Set α) = (α → Prop)`.

- This function is the "characteristic function" of the set `S`;

- the `a ∈ S` symbol means that the value of `S` is `True` when evaluated at the element `a`;

- So, the positive integers are a *function*!

⌘

Yet, given a function `P : α → Prop` we prefer to write `setOf P : Set α` rather then `P : Set α` to avoid *abusing definitional equality*.

Some examples:

1. How to prove that something belongs to a set?
2. Positive naturals;
3. Even numbers;
4. An abstract set of `α` given by some `P`.

⌘

+++

+++ Sub(sub-sub-sub)sets are not treated as sets-inside-sets.

Given a (old-style) set S, what is a subset T of S *for you*?

1. Another set such that $x\in T\Rightarrow x \in S$.
2. A collection of elements of S.

Now,

1. stresses that T is a honest set satisfying some property;
2. stresses that it is a set whose elements "come from" S.

We take the **first approach**: being a subset is *an implication*

```
def (T ⊆ S : Prop) := ∀ a, a ∈ T → a ∈ S
```

⌘

- Can also *upgrade* sets to types: `T : Set S` for `S : Set α` means `T : Set ↑S = Set (S : Type*)`.

Some examples:

1. Double inclusions;
2. Subsets as sets;
3. This upgrade (*coercion*) from `Set α` to `Type*`.

⌘

+++

Operations on Sets

+++ **Intersection**

Given sets `S T : Set α` have the

```
def (S ∩ T : Set α) := fun a ↦ a ∈ S ∧ a ∈ T
```

- Often need **extensionality**: equality of sets can be tested on elements;
- related to *functional extensionality* : two functions are equal if and only they have if they take the same values on same arguments;
- not strange: sets *are* functions.

⌘

+++

+++ **Union**
Given sets `S T : Set α` we have the

```
def (S ∪ T : Set α) := fun a ↦ a ∈ S ∨ a ∈ T
```

And if `S : Set α` but `T : Set β`? **ERROR!**

⌘
+++

+++ **Universal set & Empty set**

- The first (containing all terms of `α`) is the constant function `True : Prop`

```
def (univ : Set α) := fun a ↦ True
```

- The second is the constant function `False : Prop`

```
def (∅ : Set α) := fun a ↦ False
```

Bonus: There are infinitely many empty sets!

+++

+++ **Complement and Difference**

- The complement is defined by the negation of the defining property, denoted `Sᶜ`.

```
Sᶜ = {a : α | ¬a ∈ S}
```

The superscript `ᶜ` can be typed as `\^c`.

- The difference `S \ T : Set α`, corresponds to the property

```
def (S \ T : Set α) = fun a ↦ a ∈ S ∧ a ∉ T
```

⌘

+++

+++ **Indexed Intersections & Indexed Unions**

- Can allow for fancier indexing sets (that will actually be *types*, *ça va sans dire*): given an index type `I` and a collection `A : I → Set α`, the union `(⋃ i, A i) : Set α` consists of the union of all the sets `A i` for `i : I`.
- Similarly, `(⋂ i, A i) : Set α` is the intersection of all the sets `A i` for `i : I`.
- These symbols can be typed as `\U = ⋃` and `\I = ⋂`.

⌘

+++