Functions

A first trap

- As you know (do you?), functions among types are *primitive notions*;
- but sometimes we want to speak about functions among *sets*;
- **functions among sets are different gadgets than functions among types**.

This requires a small change of perspective.

Let's inspect the following code:

```
example (α β : Type) (S : Set α) (T : Set β) (f g : S → T) :
    f = g ↔ ∀ a : α, a ∈ S → f a = g a :=
```

It *seems* to say that `f = g` if and only if they coincide on every element of the domain, yet... ⌘

+++ Take-home message
To apply `f : α → β` to some `s ∈ S : Set α`, *restrict* it to the *subtype* `↑S` attached to `S`.

+++

Operations

Given a function `f : α → β` and sets `(S : Set α), (T : Set β)`, there are some constructions and properties that we are going to study:

+++ The **image** of `S` through `f`, noted `f '' S`.
This is the *set* `f '' S : Set β` whose defining property is

```
f '' S := fun b ↦ ∃ x, x ∈ S ∧ f x = b
```

Unfortunately it comes with a lot of accents (but we're in France...): and with a space between `f` and `''`: it is not `f'' S`, it is `f '' S`.

⌘
+++

+++ The **range** of `f`, equivalent to `f '' univ`.
I write *equivalent* because the defining property is

```
range f := (fun b ↦ ∃ x, f x = b) : β → Prop = (Set β)
```

This is not the verbatim definition of `f '' univ` : there will be an exercise about this.
+++

+++ The **preimage** of `T` through `f`, denoted `f ⁻¹' T`.
This is the set

```
f ⁻¹' T : Set α := fun a ↦ f a ∈ T
```

This also comes with one accent and *two* spaces; the symbol `⁻¹` can be typed as `\^-1`.

⌘
+++

+++ The function `f` is **injective on** `S`, denoted by `InjOn f S` if it is injective (a notion defined for functions **between two types**) when restricted to `S`:

```
def : InjOn f S := ∀ x₁ ∈ S, ∀ x₂ ∈ S, f x₁ = f x₂ → x₁ = x₂
```

In particular, the following equivalence is not a tautology:

```
example : Injective f ↔ InjOn f univ
```

it is rather an exercise for you...

⌘

+++

Inductive Types and Inductive Predicates

Inductive Types

So far, we

- met some abstact types `α, β, T : Type`, and variations like `α → T` or `β → Type`;
- also met a lot of types `p, q, (1 = 2) ∧ (0 ≤ 5) : Prop`;
- struggled a bit with `h : (2 = 3)` *versus* `(2 = 3) : Prop`;
- also met `ℕ`, `ℤ`;
- considered the subtype `{a : α // S a} = ↑S` corresponding to a set `S : α → Prop`.

How can we *construct* new types? For instance, `ℕ`, or "the" subtype `↑S`, or `True : Prop`?

+++ Using **inductive types**!

- *Theoretical* perspective: this is (fun & interesting, but) hard: you'll see it in other courses.

- *Practical* one: think of `ℕ` and surf the wave. It has two **constructors**: the constant `0 : ℕ` and the function `succ : ℕ → ℕ`, and every `n : ℕ` is of either form.

For example

```
inductive NiceType
  | Tom : NiceType
  | Jerry : NiceType
  | f : NiceType → NiceType
  | g : ℕ → NiceType → NiceType → NiceType
```

constructs the "minimal/smallest" type `NiceType` whose terms are

1. Either `Tom`;
2. Or `Jerry`;
3. Or an application of `f` to some previously-defined term;
4. Or an application of `g` to a natural and a pair of previously-defined terms.

For example, `f (g 37 Tom Tom) : NiceType`.

> **Every type in Lean is an inductive type**

In order to

1. construct terms of type `NiceType` you can use the ... *constructors*!;
2. access terms of type `NiceType` (in a proof, say), use the tactic `cases` (or `cases'` or `rcases`): the proofs for Tom and for Jerry might differ, so a case-splitting is natural.

`⌘`

+++

Inductive Families and Inductive Predicates

Recall the

```
def EvenNaturals : Set ℕ := (· % 2 = 0)
```

- For every `n`, there is a type `(EvenNaturals n) : Prop`.
- This is a *family* of types, surely a family of *inductive* types!
- But is it an inductive type *itself*?

+++ The target
When defining `inductive NiceType` one can specify where the output lives:

```
inductive NiceType : Type
  | Tom : NiceType
  | Jerry : NiceType
```

```
  | f : NiceType → NiceType
  | g : ℕ → NiceType → NiceType → NiceType
```

or

```
inductive NiceProp : Prop
  | Tom : NiceProp
  | Jerry : NiceProp
  | f : NiceProp → Prop
  | g : ℕ → NiceProp → NiceProp → NiceProp
```

> The default is `Type`.

Families

If you want a *family* of types (say, of propositions), you simply say it straight away!

```
inductive NiceFamily : ℕ → Prop
  | Tom : NiceFamily 0
  | Jerry : NiceFamily 1
  | F : ∀n : ℕ, NiceFamily n → NiceFamily (n + 37)
  | G (n : ℕ) : ℕ → NiceFamily n → NiceFamily (n + 1) → NiceFamily (n + 3)
```

Inductive Predicates are inductive families in `Prop`.

⌘

+++