Local Instances and Type Synonyms

We've seen that declaring instances has deep consequences: typically, there must be a unique instance of a certain class on a given type, and once we declare such an instance things are stuck forever.

This might be too rigid. Two ways out are *local instances* and *type synonyms*.

Local Instances

Inside a `section` we can use the `attribute [instance] myStructure` or `attribute [-instance] myStructure` to upgrade `myStructure` to an instance or remove it from the list of declared instances. Finding who is `myStructure` in a specific use-case can be non-trivial (won't be too hard neither).

An equivalent syntax is `local instance` instead of `instance` (but this does not work to *remove* instances).

+++ An example from Mathlib
In Mathlib, `ℕ` is endowed with the discrete uniformity, coming from the discrete metric:

1. The metric, induced from that on `ℝ`, satisfies `∀ n : ℕ, Metric.ball n (1/2 : ℝ) = {n}`.
2. The uniformity (*i. e.* a filter on `ℕ × ℕ`) is the principal filter containing the diagonal: `Uniformity ℕ = 𝓟 (idRel)` where
   - `idRel` is the identity relation, so the subset `{p : ℕ × ℕ | p.1 = p.2}`;
   - `𝓟 (idRel)` is the collection of all subsets in `ℕ × ℕ` that contain `idRel`, seen as a filter;
   - It can be proven that `PseudoMetricSpace.uniformity_dist` of the discrete metric is indeed `𝓟 (idRel)`;
   - Filters and uniformities are ordered, and one can prove that `𝓟 (idRel) = ⊥`, the bottom element.

Since the discrete metric induces the discrete topology, `UniformSpace.toTopologicalSpace ℕ = ⊥` where now `⊥` is the discrete topology.

GOAL : Provide another non-discrete uniform structure on `ℕ` that still induces the discrete topology.

Reference : This is actually a counterexample in Mathlib.

Idea : Set

```
        dist n m := |2 ^ (- n : ℤ) - 2 ^ (- m : ℤ)| : ℝ
```

We're identifying `ℕ` with the subset $2^{-ℕ} ⊆ ℝ$, inheriting the distance from this embedding and looking at the induced topology.

Consequence This new uniformity will be so crazy that the identity sequence `id : ℕ → ℕ` is actually Cauchy (Cauchy sequences in discrete uniform spaces are only the eventually constant ones).

+++ The problems

- How can we "replace" the discrete uniformity on `ℕ` with another one?
- How can we check that our results (for instance about `id` being Caucy) re-become *false* in the usual setting where `UniformSpace ℕ := ⟨⊥⟩`?
- How can we check that the topology remained the same, namely the discrete one?

Solution Use local instances.

⌘

+++

Type Synonyms

Another strategy that works more globally is to use *type synonyms*. The idea is to create a copy of a type, in a way that this copy inherits some instances of the original type, but not all of them.

+++ Difference between `abbrev` and `def`

This is probably beyond what is meant for this course, and certainly beyond my paygrade.

You can think that "`abbrev` is a reducible `def`", whatever this means.

Concretely: Lean "looks deeper" inside the definition of an `abbrev` than a `def`.

⌘

+++

Suppose `X` is a type, and that

```
instance : ClassOne X := ...
```

up to

```
instance : Class_n X := ...`.
```

We want a new type `newX` that has some of the above instances (and to perform this **fast**).

+++ The **wrong** way: `abbrev newX_bad := X`.

For Lean, `newX_bad` and `X` are **equal**: so, every declaration with variable `newX_bad` will accept a variable of type `X`. In particular, an `instance : MyClass newX_bad := ...` will result in an `instance : MyClass X := ...`.

We are also changing the *old* type `X`.This is **not** what we wanted.

+++

+++ The **good** way: `def newX_good := X`.
We're creating a completely new type `newX_good`. The problem is that it has no property at all, whereas we might want to inherit some properties from `X` (although probably not all of them).

We can use the syntax

```
instance : myClass newX_good := inferInstanceAs (myClass X)
```

that instructs Lean to *copy* the instance term of `myClass` from `X` to `newX_good`.
+++

⌘

Structures

- Main reference: The Lean Language Reference, in particular § 3.4.2.

The usual way to define a `structure` is to write its name, then `where` (or `:=`) and then the list of fields that we want a term of the structure to be made of

```
structure MyStructure where
    firstfield : firstType
    secondfield : secondType
    ...
    lastfield : lastType
```

or equivalently

```
structure MyStructure :=
    firstfield : firstType
    secondfield : secondType
    ...
    lastfield : lastType
```

where each field is a term in some known type. Every field can depend upon the previous ones.

- The `nth` field of a structure can be *any* term (of the right type...) but if we write

  `nth_field : nth_Type := myterm`

we are declaring that, if left unspecified, the `n`th term will be `myterm`. This is typically what we do when `nthType = Prop`: we do not want that *some property* is satisfied, but that *our sought-for property* is satisfied.

Declaring a structure as above automatically creates several terms:

1. A term `MyStructure.mk : firstType → secondType → ... → lastType → MyStructure` to *construct*
   terms; the name `.mk` can be overridden with the syntax `constructor_name ::` on the second line (so starting the list of fields on the third line).

2. A term `MyStructure.nthfield : MyStructure → nthType`: this *projects* a term of type `MyStructure` onto its `nth` field.

3. If the attribute `@[ext]` is prepended on the line before the declaration, a theorem `MyStructure.ext` is created, of type

   ∀ {x y : MyStructure}, x.firstfield = y.firstfield → ... → x.lastfield = y.lastfield → x = y

saying that if all fields of two terms coincide, the terms themselves coincide.

- If `nthType = Prop`, the arrow `x.(n-1)stfield = y.(n-1)stfield → x.nthfield = y.nthfield → is skipped thanks to proof irrelevance. Another theorem `MyStructure.ext_iff` is also added,
  that adds the reverse implication.

4. It the `@[class]` attribute is added (possibly with syntax `@[ext, class]`), a new class is created as well so that `instance : MyStructure := someterm` becomes accessible.

The call `whatsnew in` on the line preceeding the structure makes Lean shows all
newly created declarations.

+++ Use of parameters
It is also possible to define structures that depend on parameters. The syntax is the usual as for `def` or `theorem`.
+++

The call `#print MyStructure` has Lean print all fields, parameters and constructors.

Examples

We will define a structure `OneNat`, that "packs" a single natural number; the structures `TwoNat` and `Couple` that pack to numbers; or the structure of order pairs that pack two numbers where the second is larger or equal than the first, so it is a `Prop` : this is called a *mixin*.

⌘

Constructing terms

To look at the details, let's try to buid some terms of the above structures.

When doing so, `VSCode` comes at rescue: once we declare that we are looking for a term in a structure `MyStructure` (*i. e.* in an inductive type with one constructor, itself a function with several arguments), we can type

```
def MyTerm : MyStructure :=
_
```

(beware that the underscore _ **must not be indented**), and a (blue) bulb 💡 appears. Click on it to generate a *skeleton* of the structure at hand, so you do not need to remember all fields by heart.

Either using 💡 or not, there are three ways to define a term of a structure:

1. `myTerm : MyStructure :=`, followed either by
   - `by constructor` and then you're in tactic mode; or
   - `{firstfield := firstterm, secondfield := secondterm, ..., lastfield := lastterm}`.
2. `myTerm : MyStructure where` and then the list `nthfield := nthterm`, each one a new (indented) line (observe that the 💡 -action replaces `:=` with `where` automatically).
3. Using the so-called *anonymous constructor* provided by `⟨` and `⟩`: just insert the list of terms `⟨firstterm, secondterm, ..., lastterm⟩` after `myTerm : MyStructure :=` and Lean will understand.

- Remember that `class`es are a special case of `structure`s: so, definining an `instance` as we did in the last lecture really boils down to constructing a term of a certain `structure`. Points

1. − 3. above are crucial for this.

⌘

Now, constructing terms of a structure with many fields is particularly

1. boring;
2. error-prone; and
3. far from mathematical usage: to construct a term of a complicated structure we might want to use a term of a simpler one and "only add what is left to update the simpler one to the richer".

There are two ways, somewhat parallel to the `MyStructure := ...` *vs* `Mystructure where ...` syntaxes.

- The syntax `with` instructs Lean to take all possible labels from that term and to only ask for the remaining ones: it works when using the `:=` construction. Calling `with` triggers both
  - collecting all useful fields from a term; and
  - discharging all useless ones.

  Both can be used independently.
- The syntax __ has the same behaviour, and works when using the `where` construction.

In both cases, the "extra-fields" are forgotten, and thrown away.

+++ Labels Matter
The big difference between `TwoNat`, and `Couple` are the names of the fields:

```
structure TwoNat where
    fst : ℕ
    snd : ℕ

structure Couple where
    left : ℕ
    right : ℕ
```

These names **are relevant**! You might think of a term of type `TwoNat` (or `Couple`) as a
pair of *labelled* naturals, and that a structure is a collection of *labelled* terms. So,
the terms `t := {fst := 2, snd := 1} : TwoNat` and the term `t' := {left := 2, right := 1} : Couple`
have **nothing to do with each other**.
+++

+++ More about `with`
Technically, `with` updates a value: so `{fst := 1, snd := 2} with fst := 3` is
`{fst := 3, snd := 2`}.

Using `with` without specifying a new value simply instructs Lean to consider all fields on
their own without changing them (but possibly picking some of them if needed).
+++

`⌘`

Extends

We have already seen the `extends` syntax before: let's analyse its behaviour in details knowing
how `structure`s work.

The main point is to generalise to the whole type what we did for terms using `where` or `__`.

- Suppose we've already defined a structure `PoorStructure` with fields `firstfield,...,nth_field`
  and we want a new *richer* structure `RichStructure` that also contains the fields
  `(n+1)st_field,...,rth_field`. We can either
    - forget that we had `PoorStructure` and declare

    ```
    structure RichStructure where
    firstfield : firstType
    secondfield : secondType
    ...
    rth_field : rth_Type
    ```

- declare that `RichStructure` extends `PoorStructure` inheriting terms from the latter:

  ```
  structure RichStructure extends PoorStructure where
      (n+1)st_field : (n+1)st_Type
      ...
      rth_field : rth_Type
  ```

+++ In details:

- If the parent structure types have overlapping field names, then all overlapping field names must have the **same type**.
- The process can be iterated, yielding a structure extending several ones:

  ```
  VeryRichStructure extends Structure₁, Structure₂, Structure₃ where
      ...
  ```

- If the overlapping fields have different default values, then the default value from the last parent structure that includes the field is used. New default values in the child (= richer) structure take precedence over default values from the parent (= poorer) structures.
  +++

+++ Interaction of `with` and `extends`
The `with` (and `__`) syntax are able to "read through" the extension of structures.
+++

⌘

+++ In true Math
Remember the piece of code

```
class AddMonoidBad (M : Type) extends Add M, AddZeroClass M
```

We want to define an instance of `AddMonoidBad` on ℕ. Several ways:

1. type `:=`, go to a new line with `_`, wait for 💡 and fill all the fields;
2. remember that ℕ already has an `add` and a `zero`, so they can be discharged;
3. actually observe that we have an instance `AddMonoid` on ℕ, and that

   ```
   class AddMonoid (M : Type u) extends AddSemigroup M, AddZeroClass M where
   nsmul := ...
   nsmul_zero := ...
   zero_nsmul := ...
   ```

so all the fields that we need are already there: use `with` or `_` to pick them up. To do so, we need to find the name of the term `AddMonoid ℕ`, for which we can do

```
#synth AddMonoid ℕ -- Nat.instAddMonoid
```

+++

⌘

Exercises

1. Define the class of metric spaces (but call them `SpaceWithMetric` to avoid conflict with the existing library) as defined in https://en.wikipedia.org/wiki/Metric_space#Definition, and deduce an instance of `TopologicalSpace` on every `SpaceWithMetric`.

   Explain why this is the *wrong* choice, on an explicit example, and fix it.

2. When defining a `ModuleWithRel` instance on any `NormedModuleBad` we used the relation "being in the same ball of radius `1`. Clearly the choice of `1` was arbitrary.

   Define an infinite collection of instances of `ModuleWithRel` on any `NormedModuleBad` indexed by `ρ : ℝ≥0`, and reproduce both the bad and the good example.

   There are (at least) two ways:

   - Enrich the `NormedModule`'s structure with a `ρ`: this is straightforward.

   - Keep `ρ` as a variable: this is much harder, both because Lean won't be very happy with a `class` depending on a variable and because there will *really* be different instances even with good choices, so a kind of "double forgetfulness" is needed.

3. Prove the following claims, stated in the section about the non-discrete metric on `ℕ`:

   - `PseudoMetricSpace.uniformity_dist = 𝓟 (idRel)` if the metric is discrete.
   - As uniformities, `𝓟 (idRel) = ⊥`.
   - Is the equality `𝓟 (idRel) = ⊥` true as filters?
   - For any `α`, the discrete topology is the bottom element `⊥` of the type `TopologicalSpace α`.