Forgetful inheritance

As discussed in the last lecture, forgetful inheritance is the right way to ensure that extending structures does not lead to problematic diamonds: remember that diamonds are not a problem *per se*, they are perfectly fine so long they lead to **definitionally equal** terms.

The term *forgetful inheritance*, and its slogan, are due to Affeldt, Cohen, Kerjean, Mahboubi, Rouhling and Sakaguchi in their work https://inria.hal.science/hal-02463336v2 :

```
The solution to the problems [explained in this section] is to ensure definitional
equality by including poorer structures into richer ones; this way, "deducing" one
structure from  the other always amounts to erasure of data, and this guarantees
there is a unique and canonical way of getting it. We call this technique
forgetful inheritance, as it is reminiscent of forgetful functors in category
theory.
```

Slogan: **include poorer structures in richer ones.**

An example

The following example is extracted from Affeldt *et al.*'s work quoted above.

Idea:

1. Normed modules `M` are (additive) abelian groups with a `ℝ≥0`-valued norm `‖ · ‖ : M → ℝ≥0`.
2. Consider the class of modules endowed with a `Prop`-valued relation `rel : M → M → Prop`.
3. Every normed module gives rise to the relation "being in the same ball of radius `1`": so, *normed modules are a richer structure than modules with a relation*.

⌘

Both structures can be extended to (binary) products:

4. Given a pair of normed modules `M, N`, we can put the `sup` norm on `M × N`.
5. Given a pair of modules `M, N` with relations `rel_M` and `rel_N` we can put the `∧`-relation `rel_M ∧ rel_N` on `M × N`.
6. We obtain the diagram
   $$
   \begin{CD}
   M_{\mathrm{Normed}} × N_{\mathrm{Normed}} @>4.>> (M × N)*{\mathrm{Normed}}*
   @VV3.V @VV3.V
   M{\mathrm{WithRel}} × N_{\mathrm{WithRel}} @>5.>> (M × N)_{\mathrm{WithRel}}\
   \end{CD}
   $$

+++ Does it commute?

To test this, let's suppose that, for every type `T`, we have a `Prop`-valued function `p` leaving from the type `T → Prop`, so

```
p : ∀ {T : Type}, (T → Prop) → Prop
```

Now, given a `ModuleWithRel M` and a term `m : M`, we have `rel m : M → Prop`, so `p (rel m) : Prop`: it is `True` or `False`.

Let's suppose that whenever the `ModuleWithRel` structure on `M` comes from a `NormedModule` instance on `M`, we have `p (rel m) = True` for all `m : M`.

Then we expect that if `M` is a `NormedModule` and `⟨m₁, m₂⟩ : M × M`, then

```
p (rel ⟨m₁, m₂⟩) : True
```

because `M × M` has the structure of a `NormedModule`.

Yet... ⌘
+++

+++ Why?

This is not working because the `rel` in the goal comes from the `ModuleWithRel` instance on a product, whereas the `rel` in `hp` comes from the `Rel` instance *deduced* from the `NormedModuleBad` instance on the product (it suffices to hover on the terms to see this → ⌘).
+++

+++ A tentative solution
One (wrong, but instructive) solution would be to avoid declaring a `ModuleWithRel` instance on `M × N`: let's try → ⌘.

Indeed, in this case, the only instance of `ModuleWithRel` that would be found on `M × M` would b through the path

```
?m₀ : ModuleWithRel M × M ← ?m₁ : NormedModuleBad (M × M)
```

and therefore the proof would work.

But if the weaker structure `ModuleWithRel` is (mathematically) reasonable, we might want to endow a product of `ModuleWithRel`'s with such a structure *even if they are not normed*. So, the above solution does not work, but it might suggest the following trick.

The problem is that passing from `NormedModuleBad` to `ModuleWithRel` (*i. e.* declaring a `ModuleWithRel` instance on every `NormedModuleBad`)

is not a pure "erasure": we are not simply throwing away a field, rather using some field in the first (namely `‖ · ‖`) to construct the term `rel` of the second: this yields to the problem we have just witnessed.
+++

+++ The correct way (using forgetful inheritance)
Instead of *deducing* the `ModuleWithRel` instance on any `NormedModule`, we *include* the poorer structure in the richer one (the slogan...).

```
class NormedModuleGood (M : Type*) [AddCommGroup M] where
norm_g : M → ℝ≥0
rel : M → M → Prop := fun m n ↦ norm_g (m - n) ≤ 1

instance (M : Type*) [AddCommGroup M] [NormedModuleGood M] : ModuleWithRel M :=
⟨NormedModuleGood.rel⟩
```

The huge difference with what happened for `NormedModuleBad` is that *there* the instance `NormedModuleBad → ModuleWithRel` contained some **new** data (the definition of `rel`), whereas *here* it is simply a projection, forgetting `norm_g`.

Then we can define a `NormedModuleGood` instance on the product `M × N` of two `NormedModuleGood`s `M` and `N` by **using** the `ModuleWithRel` structure on `M × N`, so that `(M × N).rel` will be `defeq` to `M.rel ∧ N.rel`.

⌘

- **Remark:** The `rel v` in the goal is still the `rel` coming from the `ModuleWithRel` instance on a product, and the `rel` in `hp` still comes from the `ModuleWithRel` instance deduced from the `NormedModuleGood` structure on `M × M`, as in the first example. But now this second instance is simply obtained from the first by forgetting a field, so in particular it *coincides definitionally* with the previous one. This is another way of looking at why the seemingly odd declaration `rel := rel` in the `NormedModuleGood` instance on `M × N` makes sense.
+++

+++ A drawback

From the point of view of constructing a library, the above solution can be painful.

What can we do if we already have a class and we want to later insert something "below" it (*i. e.* to create a class that is more general than the first we had, so that every element of the first will have an instance of the second)?

We will need to modify the first one, adding to all fields of the second **although they can be deduced rather than be imposed**; and let the instance "from the first to the second" be simply a projection.

- For an example of this, together with the description of the pain it caused, see https://github.com/leanprover-community/mathlib3/pull/7084; it's Lean3, but you can see the point:

117 files were changed.

+++

In Mathlib

Remember that we defined the "bad version" of additive monoids as

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

We want to inspect why this is *bad*.

The reason is the existence of a more general class, that of
types endowed with a `0`, an addition `+` and a scalar multiplication by `ℕ`:

```
class HasNatSmul (M : Type) [Zero M] [Add M] where
    smul : ℕ → M → M
```

Every additive monoid has a scalar multiplication by `ℕ` given by `n • x := x + x + ... + x` (`n` times), so `HasNatSmul` is more general than `AddMonoidBad`, but the instance `AddMonoidBad → HasNatSmul` is not given by "pure erasure": there is no `smul` field in `AddMonoidBad`. That's against our slogan!

- Example: `ℕ` is an `AddSemigroup` and `AddZeroClass`, so it will have an instance of `AddMonoidBad`. But `ℕ` is closed under multiplication, so given `n d : ℕ` we can do

1. `n • d := d + d + ... + d` (`n` times)
2. `n • d := n * d`.

+++ Are they `defeq`?
No... `⌘`
+++

+++ The solution
As before, it goes through forgetful inheritance: define `AddMonoidGood` to have a `smul` field.

```
class AddMonoidGood (M : Type) extends AddSemigroup M, AddZeroClass M where
    smul : ℕ → M → M := nsmul_rec
```

- The `:= nsmul_rec` command instructs Lean about the *default* value to assign. This can be modified when declaring specific instances, and it takes this value if nothing is specified.

`⌘`
+++

+++ Priorities
There is also another solution, that plays with *priorities*, but it is like playing with fire.

The first problem comes from Lean being allowed to choose between `AddMndB_to_NatSmul ℕ` and `SmulEqMul_on_Nat` to obtain the `smul` on `ℕ`. So,

```
example {n m : ℕ} : HasNatSmul.smul n m = nsmul_rec n m := by ...
```

depends on **its** choice.

By default, instances are navigated in reverse order: the latest to be defined is used first (with some *caveat* when parameters are involved), so what it picks is `SmulEqMul_on_Nat` and for this reason the `smul` it choses is `n * m`.

We can change this, and doing

```
instance (priority := low) SmulEqMul_on_Nat ..
```

fixes the problem, because it tells Lean to look for other instances *before* using `SmulEqMul_on_Nat`.

Clearly this is tremendously *fragile* (but it has no impact on **good** design choices that follow the slogan) ... `⌘`.
+++

Exercises

1. Produce instances of `ModuleWithRel, NormedModuleBad, NormedModuleGood` on the type `M → M` and
   show, using the same "universal term" `p` used before, that this yields to conflicting instances for `NormedModuleBad` but not for `NormedModuleGood`.

2. 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.

3. 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.