Constructing terms (*continued*)

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, and it works when using the `:=` construction:

  ```
  example (x : OneNat) : TwoNat := {x with snd := 37}
  ```

  Calling `with` triggers both

  - collecting all useful fields from a term; and
  - discharging all useless ones.

  These features can be applied simultaneously and 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 their value (possibly picking only the ones that are needed, while discharging the others).
+++

⌘

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 `with` 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

- The process can be iterated, yielding a structure extending several ones:

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

- If the parent structures have overlapping field names, then all overlapping field names must have the **same type**.
- 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.
- 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 in `AddMonoid ℕ`, for which we can do

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

+++

⌘

Some ancillary syntax

+++ The anonymous variable
When declaring a "very basic" function, the syntax

```
fun x ↦ simple expression depending on x
```

might be too heavy. We can replace it by

```
(simple expression depending on ·)
```

where

- round parenthesis are crucial;
- `·` is typed `\. = ·` and not `\cdot = ⋅`

⌘

+++

+++ Minimally/Weakily inserted implicit variables
We've seen the syntax `{` and `}` to insert *implicit* variables. But in `Mathlib` we find the

```
def Injective (f : α → β) : Prop :=
    ∀ ⦃a₁ a₂⦄, f a₁ = f a₂ → a₁ = a₂
```

What are this funny double curly braces `⦃` and `⦄`?

Lean has a mechanism for automatically insterting implicit λ-variables when needed; so, as soon as it encounters an implicit hole, it populates it with an anonymous variable.

This can be problematic.

Let's define

```
def myInjective (f : ℕ → ℕ) : Prop :=
    ∀ {a b : ℕ}, f a = f b → a = b
```

with usual implicit variables, and let's see what goes wrong ... → ⌘

- The syntax `⦃` introduces so-called *minimally/weakly inserted implicit arguments*, that only becomes populated when something explicit *following them* is provided (lest the whole term would not type-check).

If nothing is inserted *after*, they stay implicit and the `λ`-term is treated as a honest term in the `∀`-Type.

- The reason why `exact @hg` worked is that the role of the `@` is to *disable* this mechanism of automatically populating implicit holes, that also allows to explicitely populate the fields when needed.

For more on this, see for example

https://proofassistants.stackexchange.com/questions/66/in-lean-what-do-double-curly-brackets-mean

or

https://lean-lang.org/doc/reference/latest/Terms/Functions/#implicit-functions (section §5.3.1).

+++

Exercises

1. 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.

2. 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`.

3. In the attached file `PlanMetro.pdf` you find a reduced version of Lyon's subway network. I have already defined the type of `Stations`.

   1. Find a way to formalize lines (both ordered and non-ordered), and the notion for two stations of being connected by a path.

   2. Prove that being connected is an equivalence relation.

   3. Test that *Vieux Lyon* and *Part Dieu* are not connected, but *Vieux Lyon* and *Croix-Paquet* are.

   4. 1. Formalize a program that receives two stations in input and
         - throws an error if they are not connected;
         - provides a/the list of lines (without stations) needed to travel between them, if they are connected.

5. Prove that if we add a "circle line" connecting all terminus', then every two stations become connected.

6. Formalize the notion of "correspondance" and prove that that in the above configuration with a "circle line" every trip requires at most two correspondences.