Structures

- Main reference: The Lean Language Reference, in particular § 4.4.2.

The usual way to define a `structure` is to write its name, then `where` (or `:=`, but this syntax has been deprecated) 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
```

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

- Often, some `nthType` is in `Prop`, so `nthfield : nthType` is a *proof* that the corresponding condition is satisfied.

Declaring a structure as above automatically creates several terms:

1. A term `MyStructure.mk : firstType → secondType → ... → lastType → MyStructure` to *construct* terms..
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.

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

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

Examples

We will

1. Look again at Antoine's `QuadraticAlgebra`; and then define
2. a structure `HasZero`, that simply endows a type with a "zero" element (you can think of it as a pointed type);
3. a structure `Magma` that endows a type with a binary operation.
4. a structure `Monoid` that is a `Magma` with a `Zero` that **behaves like a `0`** and where `+` is associative: this will use the **extend** construction.

⌘

Constructing terms

Let's try to buid some terms of the above structures. This can mean

- either building an explicit term of some explicit type that is a structure; or
- showing that an existing type has the (mathematical) structure implemented by our `structure`.

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.

⌘

Classes

Although this "seems to work" there are some points that are blatantly unsatisfactory:

1. We don't have a notation `†` that works nicely, we need to write `(NatMagma †) 3 2`
2. Although it is ok to be able to define arbitrary crazy additive structures on ℕ, we'd like to record that there is a prefered one, whose name we can forget and that Lean remembers.
3. We would like things to chain automatically: we've defined a topological space on every space with metric, and we could define a metric on every product of metric spaces: but we don't get *automatically* a topology on `X × Y`...

Type classes are the solution (in `Lean`, other proof assistants, like `Rocq`, take a different approach).

The idea is to build a database of terms of structures (like `NatMonoid : Monoid ℕ` or `RealMetric : SpaceWithMetric ℝ`) that can be searched by `Lean` each time that it looks for some property or some operation on a type

This will also enable more flexible notation: if Lean will see `3 † 2` it will

1. Understand `†` as the function `?α → ?α → ?α` coming from a term `?t : Magma ?α` (where both `?a` and `?t` are still to be determined)
2. Realise that `2` and `3` are terms of type ℕ, so `?α = ℕ`
3. It follows that `?t` must be a term of type `Magma ℕ`
4. Looking in the database, it will find the term `NatMagma : Magma ℕ` and it will understand what `†` in this context mean.

Before moving to the examples, observe that with all good news there are also drawbacks: if we've not been careful enough and we've recorded both `NatMagma` and `NatMagma'` as terms in `Magma ℕ`, `Lean` will find both of them in the database and will (basically) randomly pick one or the other.

⌘