An overview of Metaprogramming

+++ Acknowledgments

> And down went Mr Pickwick's remark, in Count Smorltork's tablets, with such variations and additions as the Count's exuberant fancy suggested, or his imperfect knowledge of the language, occasioned.

C. Dickens

Some of the examples below are taken from Rob Lewis' course taught at Brown University in Winter 2023. Others are taken by a series of Youtube lectures by Siddhartha Gadgil. I thank them both warmly for letting me use them.

The main reference for these topics is the very beautiful book Metaprogramming in Lean, together with Functional Programming in Lean for all aspects (and exercises) concerning monads.

Metaprogramming is a *huge* subject, and we're going to simply give a quick glance. I am **not an expert**.
+++

Macros

Macros are simply ways of creating a new tactic by packing existing tactics together. The syntax is

```
macro "name_of_your_tactic" : tactic =>
  `(tactic | *your sequences of tactics*)
```

+++ Notes

- if you want to insert a variable that you might call later (like in `intro h`) or that is already in scope (like in `cases h`), this is an
  "identifier" and you can insert it as a variable by typing `ids:ident`. To call it later, you use a dollar, as in `$ids:ident` (with its type).
- You can also require that square brackets (or parenthesis, or other stuff) are used by doing

```
macro "foo" "[" ids:ident "]" : tactic => ...
```

- You need a backtick before the parenthesis in `tactic` otherwise what you are writing gets compiled, and not stored.
- If you want to add several tactics on several lines, **use parenthesis**.

⌘

+++

Going Meta

Expressions and variables

Expressions are the most basic objects Lean deals with, and they can virtually be anything. To produce very low-level code, we must define expressions and tell Lean how to manipulate them. They are defined as the following inductive type:

```
inductive Expr where
  | bvar    : Nat → Expr                              -- bound variables
  | fvar    : FVarId → Expr                           -- free variables
  | mvar    : MVarId → Expr                           -- meta variables
  | sort    : Level → Expr                            -- Sorts
  | const   : Name → List Level → Expr                -- constants
  | app     : Expr → Expr → Expr                      -- applications
  | lam     : Name → Expr → Expr → BinderInfo → Expr  -- lambda's
  | forallE : Name → Expr → Expr → BinderInfo → Expr  -- depnd't arrows
  | letE    : Name → Expr → Expr → Expr → Bool → Expr -- let expressions
  -- less essential constructors:
  | lit     : Literal → Expr                          -- literals
  | mdata   : MData → Expr → Expr                     -- metadata
  | proj    : Name → Nat → Expr → Expr                -- projections
```

All constructors above construct things whose meaning should be pretty clear, except perhaps for free and meta variables.

Free variables are the "usual ones", like the variable `x` in `x + 2`. They are not even actually typed, they've simply got an identifier `FVarId`.

⌘

Metavariables

+++ Interlude: Monads

Monads are typeclasses for functions `m : Type* → Type*` with some
extra-property:

```
class Monad (m : Type* → Type*) where
pure : α → m α              -- an "embedding" of α` in m α
bind : m α → (α → m β) → m β   -- lifts f : α → m β to f : m α → m β.
```

(a compatibility is required between `pure` and `bind`, but we neglect it).

The interest of `bind` is that it allows composition: if `f : a → m β` and `g : β → m γ` then we would like `g ∘ f : a → m γ`, but it does not type-check. On the other hand,

```
fun (a : α) ↦ bind (bind (pure a) f) g
```

is well-formed and it is the "correct" composition. The infix version of `bind` is `>>=` so the above reads

```
fun (a : α) ↦ pure a >>= f >>= g : α → m γ
```

- One example of a monad is `Option α`: it is the type of terms either of the form `some a` for `a : α`, or equal to the extra-term `none : Option α`. Here,

  ```
  pure (a : α) = some a
  bind (some a) f = some f a
  bind none f = none
  ```

  `Option` is useful to encode errors: `List.get : ℕ → List α → Option (List α)`, so that `L.get n = none` whenever `n > L.length`.

- Another useful example is `State σ α` where `σ : Type*` is some "state-carrying" type: it is simply

  ```
  abbrev State (σ α : Type*) : Type* := σ → (σ × α)
  ```

  so a term sends a state to a *pair* of a (possibly updated) state, and an `a : α`. The monad here is `m := State σ` (for fixed `σ`), and its monad instance comes from

  ```
  pure (a : α) := fun s ↦ (s, a) (: σ ↦ σ × α)
  bind n f :=             -- here n : State σ α; and f : α → State σ β
    fun s ↦
      let (s', a) := n s            -- recall that n s : σ × α
      (f a) s'
  ```

  since `f a : State σ β = σ → σ × β`, the final `f a s'` has type `σ × β`, and therefore `bind n f : σ → σ × β = State σ β`

  This monad is useful to store values with a "state", or to mimic mutable variables.

⌘

+++

Metavariables live in suitable monads. To quote Lean's documentation,

```
Metavariables are used to represent "holes" in expressions, and goals in the
tactic framework. Metavariable declarations are stored in the MetavarContext.
```

```
Metavariables are used during elaboration, and are not allowed in the kernel, or
in the code generator.
```

Each metavariable has got a unique name, usually rendered as `?m` or `?m_197`, and a target type `T` which is *explicit*. It comes with a local context containing hypotheses (the `Γ` such that `Γ ⊢ ?m : T` is well-typed).

Both "holes" to be filled by type-inference and, most importantly, **goals** are represented by metavariables. To close a goal, we must provide an expression `e` of the target type `T`: internally, closing a goal corresponds to assigning the value `e` to the metavariable. We write `?m := e` for this assignment.

To "play" with metavariables we need our code to be *elaborated* somewhere, so what typically happens is that we `def`ine some code (possibly acting on metavariables, that do not exist at the moment of our writing) and then we declare some `elab`oration procedure where we see this code in action.

`⌘`

Creating new tactics

This relies on a monad `TacticM`: as it turns out, there are *zillions* of monads all over `Lean` and `Mathlib`, and *thousands* of "moral
interpretations" thereof. Concerning `TacticM α`, you might think of its terms as actions that

1. perform some tactic; and then
2. return a term in `α`.

Terms in `TacticM Unit` simply perform the tactic, since `Unit` only contains `_`.

Warm-up

Let's begin by implementing two tactics, one that simply counts the number of variables in the context, and one that extract all variables that are functions.

1. `Count`

   ```
   def Count : TacticM Unit := do
   let lctx ← getLCtx
   let n := lctx.decls.toList.length
   logInfo m!"There are {n - 1} variables in scope"
   ```

   - `do` is the keyword to allow imperative programming inside Lean. It is used constantly in metaprogramming.
   - `logInfo` is the tactic writing some message in the info-views: in VSCode
     this also shows up in the main window (in another colour).
   - The `-1` gets rid of the goal metavariable.
   - `getLCtx` returns the local context (as a term in `TacticM LocalContext`), and `lctx` is the array describing it (a term in `LocalContext`).

- Given a **bound variable** `bvar : α`, a **term** `mx : m α` and an **expression containing** `bvar`, say `expr(bvar) : m α`, the syntax `let bvar ← mx expr(bvar)` is a syntactic sugar for the term in `m β` defined as

```
bind mx (fun bvar ↦ expr(bvar))
                                        -- or
mx >>= (fun bvar ↦ expr(bvar))
```

Although it might look strange to use a `let` keyword for this, inspecting the above code reveals that `lctx` is being treated as a variable passed to the second line, hence the rationale. When combining several functions this is even more useful!

2. `ExtrFn`

```
def ExtrFn : TacticM Unit := do
  let mut xs := #[]
  let lctx ← getLCtx
  for lh in lctx do
    if !lh.index == 0 && lh.type.isForall
      then xs := xs.push lh.userName
  do logInfo m!"The list of functions in the context is {xs}"
  return                -- this is optional, it is a _ : Unit
```

- `let mut` introduces a *mutable* variable (Lean is a functional programming language!) so that the final `let xs :=...` works.
- `#[...]` is Lean syntax for *arrays* (as opposed to lists).
- for each `lh` in `lctx`, we get its type through `lh.type`. Then we check if `lh` is a function: this is precisely when its type
  `lh.type` is a `∀` (a "forall", also called a Π-type).

Finally,

```
elab "count_variables" : tactic => Count
elab "show_fn_var" : tactic => ExtrFn
```

where `elab` is the command enforcing some definition as a tactic.

⌘

Acting on the goal and on the assumptions

Since the goal is a metavariable, to change its state we need to *assign* it: we must attach an expression to it, that will be its value, checking its type is the main target.

```
elab "solve" : tactic => do
    let mvarId : MVarId ← Lean.Elab.Tactic.getMainGoal
    let metavarDecl : MetavarDecl ← mvarId.getDecl
    let locCtx : LocalContext := metavarDecl.lctx
    for ld in locCtx do
      if ← isDefEq ld.type metavarDecl.type then
        mvarId.assign ld.toExpr
```

- Once we have the goal (both `mvarId` and `metavarDecl`: one as an identifier, the other as declaration) we loop the context `locCtx` to check if anything we meet has got the same type as the goal. In that case, we assign the metavariable to this thing.

⌘

DeepMind Induction

While solving the IMO 2024, Google DeepMind came up with a proof, performing induction on `12` (on `10+2`, actually...)
after which, of course, the state was exactly the same (but somewhat easier for GDM to solve). We want to detect this behaviour.

```
elab "WhatsThis " n:term : tactic =>
  do
    let metavarVars ← getLCtx
    for lh in metavarVars do
      if `n == lh.userName then
        return
      else
        do logInfo m!"Do you really mean {n}?"
    return
```

- The `if` clause checks whether the term `n` appears in the goal. And then

  ```
  macro "DeepMind_induction " ids:term : tactic =>
    `(tactic | (WhatsThis $ids
                induction $ids))
  ```

⌘

Back to `∧`

We want a tactic that *completely* destructs all `p ∧ q` *hypotheses* found in the local context:
more complicated than the macro-defined `split_and` because that one *only acted on the goal*, whereas here we navigate all assumptions.

```
partial def DestrAnd : TacticM Unit := withMainContext do
  for lh in ← getLCtx do
  let eq := Expr.isAppOf lh.type ``And
  if eq then
    liftMetaTactic
      fun goal ↦ do
      let subgoals ← MVarId.cases goal lh.fvarId
      let subgoalsList := subgoals.toList
      pure (List.map (fun sg ↦
          InductionSubgoal.mvarId
          (CasesSubgoal.toInductionSubgoal sg)) subgoalsList)
    DestrAnd
    return
elab "destruct_and" : tactic => DestrAnd
```

- `eq` checks whether `lh : LocalDecl` coincides with `?m_1 ∧ ?m_2` for some metavariables `?m_1` and `?m_2`.
- the `liftMetaTactic` is an impressively powerful command that subsumes all actions on the **list** of goals.
- the `let subgoals` call create a new goal for each `eq` match; and the next call performs `cases` on it.
- Finally a recursive call to `DestrAnd` to detect nested `∧`: in particular, Lean worries that `DestrAnd` might not terminate and we're forced to declare it `partial`.

⌘

Modifying terms

We now present a tactic that, for each natural `n : ℕ` in the context, creates a new variable whose value is `2 * n`. This is decomposed in three (and a half...) steps:

1. A tactic `findNat` that identifies the `ℕ`'s in the context.

   ```
   def findNat : TacticM (List LocalDecl) := withMainContext do ...
   ```

   - This performs a tactic (so, it has some "meta"-effect) and produces a list of local declarations (our sought-for `ℕ`'s).
   - `withMainContext` is a safeguard option: it tells the tactic to constantly *update* the context, to prevent it to trying to perform actions on a state that has itself modified.

2. A tactic `listNat` that lists them (simply informative).

   ```
   def listNat : TacticM Unit := withMainContext do ...
   elab "listNat" : tactic => listNat
   ```

   - This only performs a tactic.

   - Since it is elaborated, it can be applied in tactic state (*i. e.* after `:= by...)`, unlike `findNat`.

3. A tactic `doubleNat` that, for each `h` found by `findNat`, produces a new term and assigns to it the value `2 * h.val`.

   - It relies on `mv.assertHypotheses`, that takes a list `hs` of hypotheses and converts a given goal `Γ ⊢ T` into

```
Γ, (hs[0].userName : hs[0].type) ... (hs[n].userName : hs[n].type) ⊢ T
```

   - It also uses `_root_.Lean.MVarId.intro1P` that "pops a thing in the heap" in Rocq terms: from the Lean doc,

```
Introduce one object from the goal, preserving the name used in the binder.
Returns a pair made of the newly introduced variable and the new goal. This
will fail if there is nothing to introduce, i. e. when the goal does not
start with a ∀, λ or let.
```

3½. As a bonus, a version of `doubleNat` on stereoids that *really* produces `2 * n` instead of `Nat.mul 2 n`, by working at *syntactic* level, rather than `Expr`-level.

⌘