Library lecture_tactics

Lecture 4: Tactics in UniMath

by Ralph Matthes, CNRS, IRIT, Univ. Toulouse, France

This is material for presentation at the UniMath school 2017 in Birmingham; an extended version for self-study and for exploring the UniMath library is available as `lecture_tactics_long_version.v`.

Compiles with the command
`coqc -type-in-type lecture_tactics.v`

when placed into the UniMath library

Can be transformed into HTML documentation with the command
`coqdoc -utf8 lecture_tactics.v`

In Coq, one can define concepts by directly giving well-typed terms, but one can also be helped in the construction by the interactive mode.

```
Require Import UniMath.Foundations.Preamble.
```

define a concept interactively:

```
Locate bool.
```

`bool` comes from the Coq library

```
Definition myfirsttruthvalue: bool.
```

only the identifier and its type given, not the definiens

This opens the interactive mode.

The UniMath style guide asks us to start what follows with `Proof.` in a separate line. In vanilla Coq, this would be optional (it is anyway a "nop").

```
Proof.
```

Now we still have to give the term, but we are in interactive mode. Find library elements that yield booleans:

```
 SearchPattern bool.
```

true does not take an argument, and it is already a term we can take as definiens.

```
 exact true.
```

exact is a tactic which takes the term as argument and informs Coq in the proof mode to finish the current goal with that term.

We see in the response buffer: "No more subgoals." Hence, there is nothing more to do, except for leaving the proof mode properly.

```
Defined.
```

Defined. instructs Coq to complete the whole interactive construction of a term, verify it and to associate it with the given identifer, here myfirsttruthvalue.

```
SearchPattern bool.
```

The new definition appears at the end of the list.

```
Print myfirsttruthvalue.
```

a more compelling example

```
Definition mysecondtruthvalue: bool.
Proof.
  SearchPattern bool.
  apply negb.
```

applies the function negb to obtain the required boolean, thus the system has to ask for its argument

```
  exact myfirsttruthvalue.
Defined.

Print mysecondtruthvalue.
```

```
mysecondtruthvalue = negb myfirsttruthvalue
     : bool
```

the definition is "as is", evaluation can be done subsequently:

```
Eval compute in mysecondtruthvalue.
```

```
     = false
     : bool
```

Again, not much has been gained by the interactive mode.

```
Definition mythirdtruthvalue: bool.
Proof.
  SearchPattern bool.
  apply andb.
```

`apply andb.` applies the function `andb` to obtain the required boolean, thus the system has to ask for its TWO arguments, one by one

This follows the proof pattern of "backward chaining" that tries to attack goals instead of building up evidence. In the course of action, more goals can be generated. The proof effort is over when no more goal remains.

UniMath coding style asks you to use proof structuring syntax, while vanilla Coq would allow you to write formally verified "spaghetti code".

We tell Coq that we start working on the first subgoal.

```
  -
```

only the "focused" subgoal is now on display

```
    apply andb.
```

this again spawns two subgoals

we tell Coq that we start working on the first subgoal

```
    +
```

normally, one would not leave the "bullet symbol" isolated in a line

```
      exact mysecondtruthvalue.
    + exact myfirsttruthvalue.
```

The response buffer signals:

```
This subproof is complete, but there are some unfocused goals.
                    Focus next goal with bullet -.
```

```
  - exact true.
Defined.
```

The usual "UniMath bullet order" is -, +, *, --, ++, **, ---, +++, ***, and so on (all the ones shown are being used).

Coq does not impose any order, so one can start with, e.g., *****, if need be for the sake of experimenting with a proof.

Reuse of bullets even on one branch is possible by enclosing subproofs in curly braces {}.

```
Print mythirdtruthvalue.
Eval compute in mythirdtruthvalue.
```

You only saw the tactics `exact` and `apply` at work, and there was no context.

doing Curry-Howard logic

Interactive mode is more wide-spread when it comes to carrying out proofs (the command `Proof.` is reminiscent of that).

Disclaimer: this section has a logical flavour, but the "connectives" are not confined to the world of propositional or predicate logic. In particular, there is no reference to the sort Prop of Coq. Prop is not used at all in UniMath!

On first reading, it is useful to focus on the logical meaning.

```
Locate "->".
```

non-dependent product, can be seen as implication

```
Locate "∅".
Print Empty_set.
```

an inductive type that has no constructor

```
Locate "¬".
```

```
Require Import UniMath.Foundations.PartA.
```

Do not write the import statements in the middle of a vernacular file. Here, it is done to show the order of appearance, but this is only for reasons of pedagogy.

```
Locate "¬".
Print neg.
```

Negation is not a native concept; it is reduced to implication, as is usual in constructive logic.

```
Locate "×".
Print dirprod.
```

non-dependent sum, can be seen as conjunction

```
Definition combinatorS (A B C: UU): (A × B → C) × (A → B) × A → C.
Proof.
```

how to infer an implication?

```
 intro Hyp123.
 set (Hyp1 := pr1 Hyp123).
```

This is already a bit of "forward chaining" which is a fact-building process.

```
 set (Hyp23 := pr2 Hyp123).
 cbn in Hyp23.
```

`cbn` simplifies a goal, and `cbn in H` does this for hypothesis `H`; note that `simpl` has the same high-level description but should better be avoided in new developments.

```
 set (Hyp2 := pr1 Hyp23).
 set (Hyp3 := pr2 Hyp23).
 cbn in Hyp3.
 apply Hyp1.
 apply tpair.
 - exact Hyp3.
 - apply Hyp2.
   exact Hyp3.
Defined.

Print combinatorS.
```

a more comfortable variant:

```
Definition combinatorS_induction (A B C: UU): (A × B → C) × (A → B) × A → C.
Proof.
  intro Hyp123.
  induction Hyp123 as [Hyp1 Hyp23].
```

wishes to invoke the recursor

```
 apply Hyp1.
 induction Hyp23 as [Hyp2 Hyp3].
```

wishes to invoke the recursor

```
 apply tpair.
 - exact Hyp3.
 - apply Hyp2.
   exact Hyp3.
Defined.

Set Printing All.
Print combinatorS_induction.
Unset Printing All.
```

This uses `match` that is normally not allowed in UniMath. The presence of `match` is due to a recent change in the status of Σ-types. They are a record now, in order to profit from "primitive projections".

Notice that even the projections `pr1` and `pr2` are defined by help of `match` - for the time being, since this is what happens with non-recursive fields of Coq records.

```
Definition combinatorS_curried (A B C: UU): (A → B → C) → (A → B) → A → C.
Proof.
```

use `intro` three times or rather `intros` once; UniMath coding style asks for giving names to all hypotheses that are not already present in the goal formula, see also the next definition

```
  intros H1 H2 H3.
  apply H1.
  - exact H3.
  - set (proofofB := H2 H3).
```

set up abbreviations that can make use of the current context

```
    exact proofofB.
Defined.

Print combinatorS_curried.
```

We see that `set` gives rise to `let`-expressions that are known from functional programming languages, in other words: the use of `set` is not a "macro" facility to ease typing.

`let`-bindings disappear when computing the normal form of a term:

```
Compute combinatorS_curried.
```

`set` can only be used if the term of the desired type is provided, but we can also work interactively as follows:

```
Definition combinatorS_curried_with_assert (A B C: UU):
  (A → B → C) → (A → B) → A → C.
Proof.
  intros H1 H2 H3.
```

we can momentarily forget about our goal and build up knowledge:

```
  assert (proofofB : B).
```

the current goal `C` becomes the second sub-goal, and the new current goal is `B`

It is not wise to handle this situation by "bullets" since many assertions can appear in a linearly thought argument. It would pretend a tree structure although it would rather be a comb. The proof of the assertion should be packaged by enclosing it in curly braces like so:

```
  { apply H2.
    exact H3.
  }
```

Now, proofofB is in the context with type B.

```
 apply H1.
 - exact H3.
 - exact proofofB.
Defined.
```

the wildcard ? for intros

```
Definition combinatorS_curried_variant (A B C: UU):
  (A → B → C) → (A → B) → ∀ H7:A, C.
Proof.
  intros H1 H2 ?.
```

a question mark instructs Coq to use the corresponding identifier from the goal formula

```
  exact (H1 H7 (H2 H7)).
Defined.
```

the wildcard _ for intros forgets the respective hypothesis

```
Locate "⨿".
Print coprod.
```

defined in UniMath preamble as inductive type, can be seen as disjunction

```
Locate "∏".
```

company-coq shows the result with universal quantifiers, but that is only the "prettified" version of "forall" which is a basic syntactic element of the language of Coq.

```
Locate "=".
```

the identity type of UniMath

```
Print paths.
```

How to decompose formulas

In "Coq in a Hurry", Yves Bertot gives recipes for decomposing the usual logical connectives. Crucially, one has to distinguish between decomposition of the goal or decomposition of a hypothesis in the context.

Here, we do it alike.

Decomposition of goal formulas:

A1,...,An -> B: tactic `intro` or `intros`

`¬ A`: idem (negation is defined through implication)

Π-type: idem (implication is a special case of product)

`×`: `apply dirprodpair`, less specifically `apply tpair`

Σ-type: `use tpair` or `∃`, see explanations below

`A ⨿ B`: `apply ii1` or `apply ii2`, but this constitutes a choice of which way to go

`A = B`: `apply idpath`, however this only works when the expressions are convertible

<u>Decomposition of formula of hypothesis `H`:</u>

`∅`: `induction H`

This terminates a goal. (It corresponds to ex falso quodlibet.)

There is naturally no recipe for getting rid of `∅` in the conclusion. But `apply fromempty` allows to replace any goal by `∅`.

A1,...,An -> B: `apply H`, but the formula has to fit with the goal

`×`: `induction H as [H1 H2]`

As seen above, this introduces names of hypotheses for the two components.

Σ-type: idem, but rather more asymmetric as `induction H as [x H']`

`A ⨿ B`: `induction H as [H1 | H2]`

This introduces names for the hypotheses in the two branches.

`A = B`: `induction H`

The supposedly equal `A` and `B` become the same `A` in the goal.

This is the least intuitive rule for the non-expert in type theory.

Working with holes in proofs

Our previous proofs were particularly clear because the goal formulas and all hypotheses were fully given by the system.

```
Print pathscomp0.
```

This is the UniMath proof of transitivity of equality.

The salient feature of transitivity is that the intermediate expression cannot be deduced from the equation to be proven.

```
Lemma badex (A B C D: UU) : ((A × B) × (C × D)) = (A × (B × C) × D).
```

Notice that the outermost parentheses are needed here.

```
Proof.
 Fail apply pathscomp0.
```

```
The command has indeed failed with message:
Cannot infer the implicit parameter b of pathscomp0 whose type is
"Type" in environment:
A, B, C, D : UU
```

(When using the standard setup of UniMath with ProofGeneral, this message appears only when starting emacs at the root of the UniMath library.)

`Fail` announces failure and therefore allows to continue with the interpretation of the vernacular file.

We need to help Coq with the argument `b` to `pathscomp0`.

```
 apply (pathscomp0 (b := A × (B × (C × D)))).
 -
```

is this not just associativity with third argument `C × D`?

```
   SearchPattern(_ × _).
```

No hope at all - we can only hope for weak equivalence.

```
Abort.
```

`badex` is not in the symbol table.

`Abort.` is a way of documenting a problem with proving a result.

```
Lemma sumex (A: UU) (P Q: A → UU):
  (∑ x:A, P x × Q x) → (∑ x:A, P x) × ∑ x:A, Q x.
Proof.
```

decompose the implication:

```
 intro H.
```

decompose the Σ-type:

```
 induction H as [x H'].
```

decompose the pair:

```
 induction H' as [H1 H2].
```

decompose the pair in the goal

```
 apply tpair.
 - Fail (apply tpair).
```

```
The command has indeed failed with message:
        Unable to find an instance for the variable pr1.
```

A simple way out, by providing the first component:

```
   ∃ x.
   exact H1.
 -
```

or use `use`

```
   use tpair.
   + exact x.
   + cbn.
```

is given only for better readability

```
     exact H2.
Defined.
```

a bit more on equational reasoning

```
Section homot.
```

A section allows to introduce local variables/parameters that will be bound outside of the section.

```
Locate "~".
```

```
Print homot.
```

this is just pointwise equality

```
Print idfun.
```

the identity function

```
Locate "∘".
Print funcomp.
```

plain function composition in diagrammatic order, i.e., first the first argument, then the second argument

```
Variables A B: UU.
```

```
Definition interestingstatement : UU :=
  ∏ (v w : A → B) (v' w' : B → A),
  w ∘ w' ~ idfun B → v' ∘ v ~ idfun A → v' ~ w' → v ~ w.

Check (isinjinvmap': interestingstatement).

Lemma ourisinjinvmap': interestingstatement.
Proof.
  intros.
```

is a nop since the formula structure is not analyzed

```
  unfold interestingstatement.
```

`unfold` unfolds a definition

```
  intros ? ? ? ? homoth1 homoth2 hyp a.
```

the extra element `a` triggers Coq to unfold the formula further; `unfold interestingstatement` was there only for illustration!

we want to use transitivity that is expressed by `pathscomp0` and instruct Coq to take a specific intermediate term; for this, there is a "convenience tactic" in UniMath: `intermediate_path`

```
  intermediate_path (w (w' (v a))).
  - apply pathsinv0.
```

apply symmetry of equality

```
    unfold homot in homoth1.
    unfold funcomp in homoth1.
    unfold idfun in homoth1.
    apply homoth1.
```

all the `unfold` were only for illustration!

```
  -
    Print maponpaths.
    apply maponpaths.
    unfold homot in hyp.
```

we use the equation in `hyp` from right to left, i.e., backwards:

```
    rewrite <- hyp.
```

remark: for a forward rewrite, use `rewrite` without directional argument

```
    apply homoth2.
Defined.

Variables v w: A → B.
Variables v' w': B → A.

Eval compute in (ourisinjinvmap' v w v' w').

Opaque ourisinjinvmap'.
```

```
Eval compute in (ourisinjinvmap' v w v' w').
```

`Opaque` made the definition opaque in the sense that the identifier is still in the symbol table, together with its type, but that it does not evaluate to anything but itself.

If inhabitants of a type are irrelevant (for example if it is known that there is at most one inhabitant, and if one therefore is not interested in computing with that inhabitant), then opaqueness is an asset to make the subsequent proof process lighter.

`Opaque` can be undone with `Transparent`:

```
Transparent ourisinjinvmap'.
Eval compute in (ourisinjinvmap' v w v' w').
```

Full and irreversible opaqueness is obtained for a construction in interactive mode by completing it with `Qed.` in place of `Defined.`

Using `Qed.` is discouraged by the UniMath style guide. In Coq, most lemmas, theorems, etc. (nearly every assertion in `Prop`) are made opaque in this way. In UniMath, many lemmas enter subsequent computation, and one should have good reasons for not closing an interactive construction with `Defined.`

```
End homot.
Check ourisinjinvmap'.
```

The section variables `A` and `B` are abstracted away after the end of the section.

composing tactics

Up to now, we "composed" tactics in two ways: we gave them sequentially, separated by periods, or we introduced a tree structure through the "bullet" notation. We did not think of these operations as composition of tactics, in particular since we had to trigger each of them separately in interactive mode. However, we can also explicitly compose them, like so:

```
Definition combinatorS_induction_in_one_step (A B C: UU):
  (A × B → C) × (A → B) × A → C.
Proof.
  intro Hyp123;
  induction Hyp123 as [Hyp1 Hyp23];
  apply Hyp1;
  induction Hyp23 as [Hyp2 Hyp3];
  apply tpair;
  [ exact Hyp3
  | apply Hyp2;
    exact Hyp3].
Defined.
```

The sequential composition is written by (infix) semicolon, and the two branches created by `apply tpair` are treated in the |-separated list of arguments to the brackets.

Why would we want to do such compositions? There are at least four good reasons:

(1) We indicate that the intermediate results are irrelevant for someone who executes the script so as to understand how and why the construction / the proof works.

(2) The same tactic (expression) can uniformly treat all sub-goals stemming from the preceding tactic application, as will be shown next.

```
Definition combinatorS_curried_with_assert_in_one_step (A B C: UU):
  (A → B → C) → (A → B) → A → C.
Proof.
  intros H1 H2 H3;
  assert (proofofB : B) by
  ( apply H2;
    exact H3
  );
  apply H1;
  assumption.
Defined.
```

This illustrates the grouping of tactic expressions by parentheses, the variant `assert by` of `assert` used when only one tactic expression forms the proof of the assertion, and also point (2): the last line is simpler than the expected line `[exact H3 | exact proofofB].`

This works since each branch can be given simpler as `assumption`.

Why would we want to do such compositions (cont'd)?

(3) We want to capture recurring patterns of construction / proof by tactics into reusable Ltac definitions (see long version of the lecture).

(4) We want to make use of the `abstract` facility, explained now.

```
Definition combinatorS_induction_with_abstract (A B C: UU):
  (A × B → C) × (A → B) × A → C.
Proof.
  intro Hyp123;
  induction Hyp123 as [Hyp1 Hyp23];
  apply Hyp1;
  induction Hyp23 as [Hyp2 Hyp3].
```

Now imagine that the following proof was very complicated but had no computational relevance, i.e., could also be packed into a lemma whose proof would be finished by `Qed`. We can encapsulate it into `abstract`:

```
 abstract (apply tpair;
 [ assumption
 | apply Hyp2;
   assumption]).
Defined.

Print combinatorS_induction_with_abstract.
```

The term features an occurrence of `combinatorS_induction_with_abstract_subproof` that contains the abstracted part; using the latter name is forbidden by the UniMath style guide. Note that `abstract` is used hundreds of times in the UniMath library.

a very useful tactic specifically in UniMath

Recall that `use tpair` is the right idiom for an interactive construction of inhabitants of Σ-types. Note that the second generated sub-goal may need `cbn` to make further tactics applicable.

If the first component of the inhabitant is already at hand, then the "exists" tactic yields a leaner proof script.

`use` is not confined to Σ-types. Whenever one would be inclined to start trying to apply a lemma `H` with a varying number of underscores, `use H` may be a better option.

List of tactics that were mentioned

```
exact
apply
intro
set
cbn / cbn in (old form: simpl / simpl in)
intros (with pattern, with wild cards)
induction / induction as
∃
use (Ltac notation)
unfold / unfold in
intermediate_path (Ltac def.)
etrans (Ltac def.)
```

```
rewrite / rewrite <-
assert {} / assert by
assumption
abstract
```

Index