CS3202, LSV
Semester 2 2007,
Tutorial 3

James McKinna& Stéphane Lengrand

April 26, 2007

Submission Submission should be via MMS in the form of a single COQ vernacular file, named "`CS3202tutorial3.v`". It should be added to, and regularly saved, during the course of your lab work, and then uploaded to MMS.

Deadline REVISED: Monday 23rd April 2007, by MIDNIGHT.

Credit This tutorial contributes to the 10% of the overall coursework grade allocated for tutorials.

Rubric You are reminded of the University's rules governing Academic Fraud. You may of course work with colleagues in discussing how to go about solving these problems, but any work which you submit MUST be your own, except where you EXPLICITLY reference the work of others.

1 Proof-term constructs and tactics for equality and induction

This completes the brief presentation of COQ's proof-term constructs and tactics from tutorials 1 and 2.

- Inference rule: $\dfrac{}{t = t}$
  Proof-term construct: `refl_equal` t.
  Tactic: `reflexivity`.

  It closes the branch of the proof-tree.

- Inference rule: $\dfrac{A\{x \mapsto t\} \qquad t = u}{A\{x \mapsto u\}}$
  Proof-term construct: `eq_ind t (fun x => A)` p_1 `u` p_2,
  where p_1 and p_2 are respectively the proof-terms for the premisses of the rule.

  Tactic: `Elim` works, with the syntax: `Elim (t=u) (fun x => A)`
  but it can be abbreviated as `Rew (fun x => A) t` (the system can find u), both defined in CS3202.v. You have also used two primitive tactics of COQ:

`rewrite` p (a.k.a. `rewrite ->` p) $\dfrac{A\{t \mapsto u\}}{A}$ where p is a proof of $t = u$

which rewrites **every** occurence of `t` in `A` into `u`, and its alter ego:

`rewrite <-` p $\dfrac{A\{u \mapsto t\}}{A}$ where p is a proof of $t = u$

These tactics are less flexible than `Elim` or `Rew`, but shorter to write. Also, they can perform the rewrite in an hypothesis `H:A` with the syntax
`rewrite` p `in H` and `rewrite <-` p `in H`.

- Induction

  Inference rule for integers:
  $$\dfrac{A\{x \mapsto 0\} \qquad \begin{array}{c}[m : \texttt{nat}] \quad [A\{x \mapsto m\}] \\ \vdots \\ A\{x \mapsto S(m)\}\end{array}}{\forall n, A}$$

  Proof-term construct:

  ```
  fix MyProofByInduction (n:nat) {struct n}:=
  match n with
  0 => p_1
  | S m => p_2
  end
  ```

  where `p_1` and `p_2` are respectively the proof-terms for the premisses of the rule, and `p_2` can mention `MyProofByInduction` (this is the label of the induction hypothesis).
  Note that

  ```
  Definition MyFunctionName :=
        fix MyProofByInduction (n:nat) {struct n} := [...]
  ```

  can be abbreviated as

  ```
  Fixpoint MyFunctionName (n:nat) {struct n} := [...]
  ```

  referring to `MyFunctionName` instead of `MyProofByInduction` in `p_2`.

  Tactic: `induction n`.

  Careful: This tactic re-uses the variable `n` instead of a fresh `m` in the second branch of the proof.

- We have two tactics specific to Inductive Types, both about the injectivity of type constructors:

  Inference rule $\dfrac{0 = S(n)}{A}$

  Tactic: `discriminate p`, where `p` is a proof-term for $0 = S(n)$.

  Inference rule $\dfrac{(n = m) \Rightarrow A \qquad S(n) = S(m)}{A}$

  Tactic: `injection p`, where `p` is a proof-term of type $S(n) = S(m)$.

  In most cases, `p` will be a variable of your environment. Also, you do not want to know the proof-terms constructs for these inference rules...

Composed tactics:

- `Swap` (defined in CS3202.v) $\dfrac{u = t}{t = u}$
  This is an instance of elimination of equality together with the use of an axiom (Can you give me the axiom and the predicate `A`?)
  However, it's handy to have a short tactic name to make the swap.

- `simpl` (primitive): unfolds and folds every defined notion in the current goal (and probably performs some simplification steps I can't think of yet). It also works in an hypothesis `H:A` with the syntax `simpl in H`.

Final remarks:

- The inference rules for induction and injectivity are given above for the case of natural numbers, but are **very** easy to adapt, case by case, to other inductive types, such as that of lists.

- Remember that COQ's tactics will probably do **more** than what I've described (e.g. `reflexivity` will work with quantified equalities), but their behaviour is quite unclear beyond the basic specification we expect them to have.

- I won't get into `apply`.

- The main reason for having defined (in CS3202.v) specific proof-term constructs and tactics is to match the inference rules of Natural Deduction that you have in the lectures. COQ is based on a system slightly more complex than Predicate Logic with Equality and Inductive Types, hence the non-perfect adequation between COQ's primitive tactics and the inference systems from the lectures.

2 Exercises

First, download from `studres` the file `CS3202.v`
(yes, download the latter again, it is an upgrade of the one in week 2 and 6 - and your `tutorial1.v` and `tutorial2.v` will still work with the upgrade)

Task 1: Lists

- Define the inductive type of lists of natural numbers: `List:Set`
  with `Nil` as the constructor for the empty list.
- Define the function that appends two lists: `append:List->List->List`
  You will do this by induction on the first argument (the list that comes first). Such a definition is implemented in COQ with

  ```
  Fixpoint append (l l':List) {struct l}: List  :=
  [...]
  .
  ```

- Define the function that reverses a list: `reverse:List->List`
  You will do this by induction on the argument, **using the function `append`**. Such a definition is implemented in COQ with

  ```
  Fixpoint reverse (l:List) {struct l} : List :=
  [...]
  .
  ```

  where the body of the function will contain a call to `append`.
- Define the function that reverses a list faster:

  ```
  Definition fast_reverse (l:List) : List :=
  fast_reverse_aux l Nil
  .
  ```

  where `fast_reverse_aux:List->List->List` is an auxiliary function taking two lists as arguments, defined by induction on the first one with a code starting with:

  ```
  Fixpoint fast_reverse_aux (l l':List) {struct l} : List  :=
  [...]
  .
  ```

  The first argument is the list to reverse, the second is a "buffer" where the current result is stored, and given as output in the case where the first argument is finally the empty list. **Neither `fast_reverse` nor `fast_reverse_aux` will mention `append`.**

- Prove the following properties of append:
  - `Nil` is an identity for `append`:
    `Theorem append_Nil:forall l:List, append l Nil = l.`
  - `append` is associative:
    `Theorem append_assoc:forall l m n:List, append (append l m) n = append l (append m n).`
- Prove the theorem: `Theorem t:forall l:List, reverse l=fast_reverse l.`
  Hint. `fast_reverse` instantiates one argument of `fast_reverse_aux` with `Nil` ...so a reasonable strategy for proofs about `fast_reverse`, is to derive results from corresponding proofs about `fast_reverse_aux`.

  In this instance, the intended meaning of `fast_reverse_aux l m` is to compute `append (reverse l) m`. Accordingly, prove the theorem

  `Theorem t_aux:forall l m:List, append (reverse l) m=fast_reverse_aux l m.`

  You may find this requires associativity of append as above.

  Then conclude by instantiating `m` with `Nil` ...which will require the above lemma about `append` and `Nil`.