CS3202, LSV
Semester 2 2007,
Tutorial 2

James McKinna& Stéphane Lengrand

March 4, 2007

Submission Submission should be via MMS in the form of a single COQ vernacular file, named "`CS3202tutorial2.v`". An initial template version of this file has been prepared for you. It should be added to, and regularly saved, during the course of your lab work, and then uploaded to MMS.

Deadline Friday 9th March 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 Some concrete syntax

Let us start with a reminder of definition commands in COQ:

```
Let myfavoritename := myterm.
```
or
```
Definition myfavoritename := myterm.
```

Also, note that in COQ's original concrete syntax,

- $\bot$ is `False` (I defined `bot` as `False` in `CS3202.v`)
- $\neg A$ can be written as either `not A` or `~A`

Here is now the concrete syntax for quantifiers. First, COQ extends predicate logic by allowing *different universes* to be ranged over. We shall declare a universe (in fact, a *set*) `iota` with the command:
```
Variable iota:Set.
```

Quantifiers usually need the set over which they range, namely:
$\forall x, A$ is represented as `forall x:iota,A` and
$\exists x, A$ is represented as `exists x:iota,A`
if the quantification is over `iota`.

Today we stick to one universe only, and in that case the system often accepts the abbreviations `forall x,A` and `exists x,A`

If things do not work, try adding " `:iota` " where you left it implicit.

2 The proof-mode of COQ

In this tutorial, you discover the *proof-mode* of COQ.

It allows you to interactively build a proof-tree for a formula (as we did last week on a whiteboard or on the slides), rather than explicitly entering the proof(-term) by hand and then checking that it is typed with the right formula (`Check myproofterm.`).

Like last week on the whiteboard or on the slides, the interactive creation of the proof-tree is done *bottom-up*:

1. starting from the conclusion of the tree (labelled with the formula to be proved),
2. trying to guess which rule was applied last in the (at that point hypothetical) proof-tree to be found,
3. and bearing in mind (or on a special display of COQ) the local assumptions that you are allowed to use at each point of the proof.

In theory (at least, in the exercises you will do today), you do not have to enter a single proof-construct by hand (such as a `fun x:A => M`, `or_introl`, `or_intror`, `conj`, `match M with ...end`).

The final proof-tree represented with these constructs will be the *result/output* of successfully exiting the proof-mode.

- Point 1 above is done by entering the proof-mode with one of the following commands:
  `Theorem MyProofName : MyFormulaToProve .`
  or
  `Lemma MyProofName : MyFormulaToProve .`

  The goal that you have to prove is now displayed in the top-right window.

  ```
  1 subgoal


  ==============
  MyFormulaToProve
  ```

- Point 2 above is done by invoking (basic proof-search) tactics, which try to apply the inference rules bottom-up. If such a tactic succeeds, then it updates the goal to be proved with the premisse(s) of the inference rule.
  - If there are more than one premiss, it selects one of them to work on and records the others as *sub-goals* to be proved later, once you have finished with your current goal (these sub-goals correspond to other branches of the proof-tree).
  - If the inference rule discharges some hypotheses, these become available for the proof of the new goal, and are added above the double bar line, decorated with a freshly generated variable such as `H`, `H1`, ..., `Hn`,...

  Hence, the general shape of the display is

```
p subgoals

H1 : MyLocalHypothesis1
...
Hn : MyLocalHypothesisn
===============
MyCurrentGoal           (1/p)

MyNextGoal              (2/P)
...
MyLastGoal              (p/p)
```

  In this tutorial, you will use the proof-search tactics `unfold`, `exact`, `intro`, `split`, `exists`, and `Elim` (the first five are primitive in Coq, the last one is programmed in `CS3202.v`).

  - The tactic `unfold MyDefinedName` unfolds the definition of `MyDefinedName` in the current goal. `unfold MyDefinedName in H` does it in the hypothesis decorated by `H`
  - The tactic `intro` executes the bottom-up application of the introduction rule for $\Rightarrow$, $\forall$, or $\neg$, dictated by the main connective of the goal:

    `intro` $\dfrac{\begin{array}{c}[A]\\ \vdots \\ B\end{array}}{A \Rightarrow B}$ or $\dfrac{B}{\forall x, B}$ or $\dfrac{\begin{array}{c}[A]\\ \vdots \\ \bot\end{array}}{\neg A}$
  - For some reason, the tactic is called `split` to attempt $\wedge$-intro:

    `split` $\dfrac{A \quad B}{A \wedge B}$

    For instance, `split` starts building your proof-term with `conj ? ?`, with the hope that solving the sub-goals will provide terms for the `?` 's.
  - For $\vee$-intro the tactics are:

    `left` $\dfrac{A}{A \vee B}$ and `right` $\dfrac{B}{A \vee B}$

– For $\exists$-intro the tactic is:

`exists MyTerm` $\qquad \dfrac{A\{x \mapsto \texttt{MyTerm}\}}{\exists x, A}$

– The tactic `Elim` tries an elimination rule, it usually takes as an argument the premiss whose main connective is to be eliminated:

`Elim (A\/B)` $\qquad \dfrac{A \vee B \qquad \begin{matrix}[A]\\ \vdots \\ C\end{matrix} \qquad \begin{matrix}[B]\\ \vdots \\ C\end{matrix}}{C}$ (this tactic always succeeds; why?)

`Elim (exists x,A)` $\qquad \dfrac{\exists x, A \qquad \begin{matrix}[A]\\ \vdots \\ C\end{matrix}}{C}$ (this tactic always succeeds; why?)

`Elim False` $\qquad \dfrac{\bot}{A}$ (this tactic always succeeds; why?)

`Elim (~A)` $\qquad \dfrac{A \qquad \neg A}{\bot}$

Note that if the goal is anything else than `False`, the tactic will first apply $\bot$-elim in order to turn the goal into `False`, and then apply $\neg$-elim. Hence, the tactic always succeeds.

`Elim (forall x, B)` $\qquad \dfrac{\forall x, B}{B\{x \mapsto t\}}$

`Elim (A->B)` $\qquad \dfrac{A \Rightarrow B \qquad A}{B}$

Note that `Elim (A->B)` can be abbreviated as `MP A` (since B is known to be the current goal), so `MP A` always succeeds.

`Elim left B` $\qquad \dfrac{A \wedge B}{A}$ (this tactic always succeeds; why?)

Note that B has to be given as argument of the tactic, since the system cannot guess it (unlike A, which the system knows to be your current goal).

`Elim right A` $\qquad \dfrac{A \wedge B}{B}$ (this tactic always succeeds; why?)

Again, A has to be given while B is known as your goal.

Note that in $\forall$-intro (`intro`) and $\exists$-elim (`Elim (exists x,A)`), the side condition that the variable bound in the quantified formula does not already appear elsewhere is automatically satisfied by Coq, which picks a fresh variable (remember we are building the proof-terms bottom-up).

– Finally, you can *use* an hypothesis with the tactic `exact`:

`exact Hn` tries to apply "copy" $\dfrac{A}{A}$ and succeeds if the hypothesis decorated by `Hn` is equal (modulo α-equivalence) to your current goal. Such a success ends the

construction of one branch of your proof-tree, and the next recorded sub-goal left to prove becomes your current goal.

- When you have constructed all branches (i.e. there is no sub-goal left to be proved after `exact`), you have proved the result and the window displays:
  `Proof completed.`

  You can now save the proof-term that you have interactively built with the command
  `Defined.`

  This exits the proof-mode (the top-right window is cleared) and binds the proof-term to the name `t1`, as if you had defined it with the command
  `Let t1 := theproofterm.`

  Note however that by defining `t1` with the proof-mode, you have never *seen* the proof-term, which you can now do with the command (check the output in the bottom-right window):
  `Print t1.`

  Note that sometimes the proof-term is not "fully evaluated" and you have to compute it with the (rather ackward, agreed) command:
  `Eval compute in t1.`

3 Exercises

First, download from `studred` the files `CS3202tutorial2.v` **and** `CS3202.v`
(yes, download the latter again, it is an upgrade of the one in week 2 - and your `tutorial1.v` will still work with the upgrade)

Task 1: Predicate logic

We start by declaring a universe, called `iota`, as the (COQ-)*type* of the object we are talking about / quantifying over.

```
Variable iota:Set.
```

We then declare two predicates `P` and `Q` of arity 1 on this universe.

```
Variables P Q:iota->Prop.
```

Now your job is to prove the theorems of predicate logic that you have seen in the lectures (you have the proof-trees on the slides!)

- We start with $(\forall x, P(x) \Rightarrow Q(x)) \Rightarrow ((\forall y, P(y)) \Rightarrow (\forall z, Q(z)))$

  ```
  Theorem t1:(forall x, P x->Q x)->((forall y, P y)->(forall z, Q z)).

  [...to be completed...]

  Defined.
  Print t1.
  Eval compute in t1.
  ```

- Now we prove $(\exists x, P(x) \vee Q(x)) \Rightarrow ((\exists y, P(y)) \vee (\exists z, Q(z)))$

```
Theorem t2: (exists x,P x \/ Q x)->(exists y, P y)\/(exists z, Q z).

[...to be completed...]

Defined.
Print t2.
Eval compute in t2.
```

- Now we prove $((\exists y, P(y)) \vee (\exists z, Q(z))) \Rightarrow (\exists x, P(x) \vee Q(x))$

```
Theorem t3: ((exists y, P y)\/(exists z, Q z)) -> (exists x,P x \/ Q x).

[...to be completed...]

Defined.
Print t3.
Eval compute in t3.
```

Task 2: Classical Logic and Coq

This exercise answers someone's question on integrating classical logic into COQ, and aims at establishes the equivalence between the different ways to do it.

First, note that in COQ, negation is a defined notion: `~A` *is* in fact an abbreviation of `A->False`.

For instance, the elimination of double negation `(~~C)->C` *is* in fact `((~C)->False)->C`, i.e. *reasoning by contradiction*.

Now, it is up to you to prove the equivalences between Reasoning by Contradiction, the Law of Excluded Middle, Peirce's Law, and the Elimination of Double Negation.

```
Definition LEM (D:Prop) := D\/~D.
Definition Peirce (A B: Prop) := ((A->B)->A)->A.
Definition EDN (C:Prop) := (~~C)->C.
```

- First, we prove that the Elimination of Double Negation implies Peirce's Law.

```
Variables E F:Prop.

Theorem one : EDN E->Peirce E F.
```

- Second, we prove that Peirce's Law implies the Law of Excluded Middle.

```
Theorem two : Peirce (E\/~E) False ->LEM E.
```

- Third, we prove that the Law of Excluded Middle implies Elimination of Double Negation.

```
Theorem three : LEM E->EDN E.
```