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: \( t = t \)
  Proof-term construct: \( \text{refl}_\text{equal} \).
  Tactic: \( \text{reflexivity} \).
  It closes the branch of the proof-tree.

- Inference rule: \( A \{ x \mapsto t \} \quad t = u \)
  Proof-term construct: \( \text{eq}_\text{ind} \quad (\text{fun} \ x \Rightarrow A) \quad p_1 \quad p_2 \),
  where \( p_1 \) and \( p_2 \) are respectively the proof-terms for the premises of the rule.

  Tactic: \( \text{Elim} \) works, with the syntax: \( \text{Elim} \ (t=u) \ (\text{fun} \ x \Rightarrow A) \quad p_1 \quad p_2 \),
  but it can be abbreviated as \( \text{Rew} \ (\text{fun} \ x \Rightarrow A) \quad t \) (the system can find \( u \)), both defined in \( \text{CS3202.v} \). You have also used two primitive tactics of COQ:
rewrite \( p \) (a.k.a. rewrite \( \rightarrow p \) \( A \{ t \mapsto u \} \)) where \( p \) is a proof of \( t = u \)

which rewrites every occurrence of \( t \) in \( A \) into \( u \), and its alter ego:

rewrite \( \leftarrow \) \( p \) \( A \{ u \mapsto t \} \) where \( p \) is a proof of \( t = u \)

These tactics are less flexible than \( \text{Elim} \) or \( \text{Rew} \), but shorter to write. Also, they can
perform the rewrite in an hypothesis \( H : A \) with the syntax

\[
\text{rewrite } p \text{ in } H \text{ and rewrite } \leftarrow p \text{ in } H.
\]

- **Induction**

\[
\forall n, A
\]

Inference rule for integers:

\[
\begin{array}{c}
A \{ x \mapsto 0 \} \\
A \{ x \mapsto m \}
\end{array}
\]

Proof-term construct:

\[
\begin{array}{c}
\text{fix MyProofByInduction } (n : \text{nat}) \{ \text{struct } n \} : = \\
\text{match } n \text{ with} \\
\text{0 => } p_1 \\
| S m \Rightarrow p_2
\end{array}
\]

where \( p_1 \) and \( p_2 \) are respectively the proof-terms for the premisses of the rule, and
\( p_2 \) can mention \( \text{MyProofByInduction} \) (this is the label of the induction hypothesis).

Note that

\[
\text{Definition MyFunctionName := } \\
\text{fix MyProofByInduction } (n : \text{nat}) \{ \text{struct } n \} : = [\ldots] \\
\]

can be abbreviated as

\[
\text{Fixpoint MyFunctionName } (n : \text{nat}) \{ \text{struct } n \} : = [\ldots]
\]

referring to \( \text{MyFunctionName} \) instead of \( \text{MyProofByInduction} \) 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

\[
0 = S(n) \\
\]

Tactic: \( \text{discriminate } p \), where \( p \) is a proof-term for \( 0 = S(n) \).

Inference rule

\[
(n = m) \Rightarrow A \quad S(n) = S(m)
\]

Tactic: \( \text{injection } p \), 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) \[ \frac{u = t}{t = u} \]
  
  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: \( \text{List:Set} \) with \( \text{Nil} \) as the constructor for the empty list.

- Define the function that appends two lists: \( \text{append:List->List->List} \)
  You will do this by induction on the first argument (the list that comes first). Such a definition is implemented in COQ with

  \[
  \text{Fixpoint append } (l \ l':\text{List}) \ {\text{struct } l} : \text{List} := \\
  \quad [...]
  \]

- Define the function that reverses a list: \( \text{reverse:List->List} \)
  You will do this by induction on the argument, using the function \( \text{append} \). Such a definition is implemented in COQ with

  \[
  \text{Fixpoint reverse } (l:\text{List}) \ {\text{struct } l} : \text{List} := \\
  \quad [...]
  \]

  where the body of the function will contain a call to \( \text{append} \).

- Define the function that reverses a list faster:

  \[
  \text{Definition fast_reverse } (l:\text{List}) : \text{List} := \\
  \quad \text{fast_reverse_aux } l \ \text{Nil}
  \]

  where \( \text{fast_reverse_aux:List->List->List} \) is an auxiliary function taking two lists as arguments, defined by induction on the first one with a code starting with:

  \[
  \text{Fixpoint fast_reverse_aux } (l \ l' :\text{List}) \ {\text{struct } l} : \text{List} := \\
  \quad [...]
  \]

  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. \( \text{Neither fast_reverse nor fast_reverse_aux will mention append.} \)
• Prove the following properties of append:
  
  – Nil is an identity for append:
    Theorem append_Nil: \( \forall l: \text{List}, \text{append } l \text{ Nil } = l \).
  
  – append is associative:
    Theorem append_assoc: \( \forall l \ m \ n: \text{List}, \text{append } (\text{append } l \ m) \ n = \text{append } l \ (\text{append } m \ n) \).
  
• Prove the theorem: Theorem t: \( \forall l: \text{List}, \text{reverse } l = \text{fast_reverse } l \).
  
  **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: \( \forall l \ m: \text{List}, \text{append } (\text{reverse } l) \ m = \text{fast_reverse_aux } l \ m \).
  
  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.