Proof Assistants (MPRI 2-7-2)

Written examination

March 10th 2016

Instructions

- The exam lasts three hours.
- This document has 7 pages and comprises 4 independent exercises.
- Hand-written notes, printed handouts of the slides/companion documents of the lecture are allowed. Laptops, tablets, etc. are not allowed.
- Language: French or English
- When asked for code, write Coq code, when asked for explanation, write complete English/French sentences. Be concise but precise.
- The annex in Section 5 provides the definition of some standard Coq constants. The code of any other definition you may use should be explicitly provided in your answer. Implicit types can be omitted or made explicit, at your convenience.
- The code of recursive definition with more than one inductive argument should mention explicitly the decreasing argument, using the `{struct _}` syntax.
- We mark with a star (*) especially difficult questions.

1 Quiz (5 points)

1. In this exercise, define the required Coq term (resp. give its type) or explain why the definition of such a term is not valid in Coq's type theory:

   a) Replace the `<todo>` placeholders in order to define an inductive type `type_sum`, in sort `Type`, with two parameters `(A : Type)` and `(B : Type)`, whose inhabitants are either an element `(a : A)` or an element `(b : B)`:

   ```
   Inductive type_sum <todo> : <todo> := <todo>.
   ```

   b) Replace the `<todo>` placeholders in order to define an inductive type `prop_sum`, in sort `Prop`, with two parameters `(A : Type)` and `(B : Type)`, whose inhabitants are either an element `(a : A)` or an element `(b : B)`:

   ```
   Inductive prop_sum <todo> : <todo> := <todo>.
   ```

   c) What is the type of term `type_sum` ? of term `prop_sum`?

   d) Define the non-dependent elimination rule for `prop_sum` in `Prop`, corresponding to simple case analysis of `prop_sum A B`.

   e) Give the *type* of the non-dependent elimination rule for `prop_sum` in `Type`

2. Suppose given the following inductive definition of arithmetic expressions:

   ```
   Inductive arith_exp : Set :=
   | zero : arith_exp
   | var (n : nat) : arith_exp
   | plus : arith_exp -> arith_exp -> arith_exp.
   ```

   Give the *type* of the dependent elimination principle for `arith_exp` in `Type`, that is, for any predicate `arith_exp -> Type`.

3. Complete the definition of `arith_exp` with an index capturing the variable of highest index in the expression (remember you can use functions given in the appendix).

   ```
   Inductive arith_exp : nat -> Set :=
   | zero : arith_exp <todo>
   | var (n : nat) : arith_exp <todo>
   | plus <todo> : arith_exp <todo> -> arith_exp <todo> -> arith_exp <todo>.
   ```

   Give the type of the dependent elimination principle for this new version of `arith_exp`, for any predicate `forall i : nat, arith_exp i -> Type`.

4. Among the following declarations, which ones are rejected and why?

   ```
   Inductive i1 : Type := c11 : i1 -> i1 | c12 : nat -> i1.
   Inductive i2 : Type := c2 : (i2 -> i1) -> i2.
   Inductive i3 : Type := c3 : (i1 -> i3) -> i3.


   Definition f1 : Prop := forall p : Prop, p \/ ~ p.
   Definition f2 : Set := forall p : Set, p -> p.
   Definition f3 : Type := forall p : Type, p.
   ```

```
Variables (p1 : f1) (p2 : f2).

Definition t1 := p1 f1.
Definition t2 := p1 f2.
Definition t3 := p2 f1.
Definition t4 := p2 f2.
```

5. Define a term:

   ```
   eq_trans : forall (A : Type) (x y z : A), x = y -> y = z -> x = z.
   ```

   by replacing the `<todo>` placeholders in the following code:

   ```
   Definition eq_sym (A : Type) <todo> : y = z -> x = z :=
   match <todo> in <todo> return <todo> with
   | <todo> => <todo>
   end.
   ```

2 Programming with Natural Numbers and Lists in Coq (5 points)

1. Program in Coq a function

   ```
   iter : forall T : Type, nat -> (T -> T) -> T -> T
   ```

   such that `(iter n f x)` computes the value of function `f` iterated `n` times on argument `x`.

2. Program in Coq a function `rcons` of type:

   ```
    rcons : forall A : Type, A -> list A -> list A
   ```

   such that `(rcons x l)` adds `x` at the end of the list `l`.

3. Program in Coq a function `drop` of type

   ```
    drop : forall A : Type, nat -> list A -> list A
   ```

   such that `(drop n l)` returns a copy the list `l` minus its `n`-th first items, and the empty list if `n` exceeds the length of `l`.

4. Program in Coq a function `take` of type

   ```
    take : forall A : Type, nat -> list A -> list A
   ```

   such that `(take n l)` returns the prefix of length `n` of the list `l`, and the list if `n` exceeds the length of `l`.

5. What is the simple property relating `(take n l)`, `(drop n l)` and `l` itself? State this property in Coq and write a recursive function (mimicking the definitions of `drop` and `take`), that proves this statement.

6. Program in Coq a function `rot` of type

   ```
     rot : forall A : Type, nat -> list A -> list A
   ```

   such that `(rot n l)` rotates left the list `l` `n` times: `(rot 1 [a, b, c])` should evaluate to `[b, c, a]`, `(rot 2 [a, b, c])` should evaluate to `[c, a, b]` and `(rot 5 [a, b, c])` should evaluate to `[a, b, c]`.

7. Program in Coq a function `cycle` of type

   ```
   cycle : forall A : Type, nat -> list A -> list A
   ```

   such that for any natural number `n`, `(cycle n s)` is list obtained by the circular permutation of `s` shifting the items of `s` of `n`: `(cycle 2 (1 :: 2 :: 3 :: 4 :: nil))` evaluates to `(3 :: 4 :: 1 :: 2 :: nil)`.

8. State in Coq the property expressing the result of composing two instances `(cycle n)` and `(cycle m)` of such circular permutations, for arbitrary natural numbers `n` and `m`. How would you prove it?

3 Regular expressions (5 points)

The goal of this exercise is to develop a small library of regular expression matching on words in an arbitrary alphabet. Consider the following definition defining the syntax of simple regular expressions on an alphabet A with decidable equality:

```
Parameter A : Set.
Parameter eqA : A -> A -> bool.
Inductive regexp : Set :=
| empty : regexp
| epsilon : regexp
| test : (A -> bool) -> regexp
| compose : regexp -> regexp -> regexp
| or : regexp -> regexp -> regexp
| star : regexp -> regexp
```

1. Define the type of words as a list of elements of A.

2. A *recognizer* for a type T is a function from T to `regexp`. Define a recognizer for a single element of `A` using the `test` constructor.

3. Define a recognizer for a word using the `test` and `compose` constructors.

4. Define a function on regular expressions testing if the expression can match the empty word.

5. The following function sketch matches a word with a regular expression, using a continuation. Complete the missing cases.

```
Fixpoint matches (r : regexp) (w : word) (k : word -> bool) : bool :=
  match r, w with
  | empty, s => false
  | epsilon, s => k s
  | test f, c :: s => if f c then k s else false
  | test f, [] => <todo>
  | compose r r', s => matches r s (fun s' => <todo>)
  | or r r', s => <todo>
  | star r, s => matches r s (fun s' => matches (star r) s' k) || <todo>
  end
```

6. ∗ This function is not structurally recursive, why? Give (in english) a wellfounded order for which the function decreases, with an informal argument why each and every recursive call decreases. You can assume additional invariants on all the arguments of the function, but should state them explicitely and show how they are preserved at recursive calls.

7. Define a matching function of type `word -> regexp -> bool` assuming the one above.

8. Define an inductive predicate expressing that an expression matches a word, i.e, an inductive of type `word -> regexp -> Prop`.

9. State the correctness and completeness lemmas of the matching function with respect to the predicate.

10. Define equivalence of regular expressions `equiv_lang` using the `matches` predicate defined previously. Two regular expressions should be equivalent iff they match the same words.

11. Regular expression operators obey algebraic laws (it is a Kleene algebra), state 2 of them as language equivalences. How would the proof of these laws proceed, in terms of Coq tactics?

12. ∗ Suppose we want to define a regular expression simplifier implementing some of these laws, i.e. a function of type `regexp -> regexp` and show that it preserves language equivalence. Are there any issues with the representation type `regexp` to do so?

4 Excluded Middle and Degeneracy of Proofs (5 points)

The aim of this exercise is to prove that the excluded middle principle (in sort `Prop`) implies proof irrelevance. We hence work in an environment assuming the `em` axiom:

```
Axiom em : forall P : Prop, P \/ ~ P.
```

1. Write the (most general) type of the following dependent elimination principle:

```
fun A P f g =>
      match em A as x return P x with
      | (or_introl a) => f a
      | (or_intror b) => g b
      end
```

In the rest of the exercise, we call this term `em_elim`.

2. Write a term `dneg: forall A : Prop, ~ ~ A -> A.`

3. Define an enumerated inductive type `BOOL : Prop` with two constructors `T` and `F`.

4. Define a term `p2b : Prop -> BOOL` such that `(p2b A)` is convertible to `T` when `A` holds and to `F` otherwise.

5. ∗ Prove :

   ```
   five_1 : (~ (T = F)) -> forall A : Prop, A <-> p2b A = T.
   ```

   ```
   five_2 : (p : T <> F) (A : Prop) : p2b A = T -> A.
   ```

   You can provide a proof term or a proof script.

6. We suppose the existence of a term:

   ```
   paradox
        : forall (B : Prop) (p2b : Prop -> B) (b2p : B -> Prop),
          (forall A : Prop, b2p (p2b A) -> A) ->
          (forall A : Prop, A -> b2p (p2b A)) -> forall C : Prop, C
   ```

   which can actually be constructed in the Calculus of Inductive Constructions. The definition of this term is **not** part of the question. Using `paradox` and `p2b`, define a term `nnTeqF: ~ ~ T = F` and a term `TeqF: T = F`.

7. ∗ Deduce from this that `forall (A : Prop)(a1 a2 : A), a1 = a2`.

5 Annex

- The standard library of Coq provides a polymorphic type for lists:

  ```
  Inductive list {A : Type} : Type :=
    nil : list A | cons : A -> list A -> list A
  ```

  The parameter `A` is implicit throughout this exercise. The `cons` constructor has an infix notation `_ :: _`: term `3 :: nil` is the list with one item, `3`. The library provides a concatenation operation `app`, equipped with an infix notation `_ ++ _`, like in term `(3 :: nil)++ (6 :: 7 :: nil)`:

  ```
  Fixpoint cat (s1 s2 : list A) {struct s1} :=
    match s1 with |x :: s1' => x :: s1' ++ s2 | _ => s2 end.
  ```

- Natural numbers are represented by type:

```
Inductive nat : Set := O : nat | S : nat -> nat
```

The standard library provides arithmetic operations equipped with their standard infix notations along with:

- the boolean equality test `beq_nat: nat -> nat -> bool`, which verifies:

  ```
  beq_nat_true_iff : forall x y : nat, beq_nat x y = true <-> x = y
  ```

- the boolean conjunction and disjunction `&&, || : bool-> bool-> bool`
- `max: nat -> nat -> nat` computing the maximum of its two arguments.
- the boolean comparison test `NPeano.ltb: nat -> nat -> bool`, which verifies:

  ```
  NPeano.ltb_lt: forall n m : nat, NPeano.ltb n m = true <-> n < m
  ```

- Notation `x = y` unfolds to `eq _ x y` with:

  ```
  Inductive eq (A : Type) (x : A) : A -> Prop := eq_refl : eq x x
  ```

- Congruence can be expressed using:

  ```
  f_equal : forall (A B : Type) (f : A -> B) (x y : A), x = y -> f x = f y
  ```

- Notation `A /\ B` unfolds to `and A B` with:

  ```
  Inductive and (A B : Prop) : Prop := conj : A -> B -> and A B.
  ```

  Notation `~ A` unfolds to `A -> False` with:

  ```
  Inductive False :=.
  ```

  The associated elimination principle is:

  ```
  False_ind : forall P : Prop, False -> P
  ```