Proof Assistants (MPRI 2-7-2)
Written examination

November 22nd 2018

Instructions

- The exam lasts three hours.
- This document has 7 pages and comprises 3 independent exercises.
- Hand-written notes, printed handouts of the slides/companion documents of the lecture are allowed. Laptops, smartphones, 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 4 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 (*) likely more difficult questions.

1 Quiz (7 points)

1. What is the shortest proof of `3 * 3 = 9` in Coq (assuming the standard library definitions of addition, multiplication and equality).

2. In this exercise, define the required Coq term (resp. give its type):

   (a) Replace the `<todo>` placeholders in order to define an inductive type of arithmetic expressions `aexpr` on an arbitrary number type `N : Type`, in sort `Type`, whose inhabitants are either:

   - Constants of the parameter type.
   - Variables encoded as de Bruijn indices in `(n : nat)`
   - An operation of addition

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

   (b) What is the full type of the constructor for constants?

   (c) Give the type of the non-dependent elimination rule for `aexpr` in `Type`, corresponding to simple recursion on `aexpr`.

   (d) Replace the `<todo>` placeholders in order to define an inductive type `iff`, in `Prop`, representing logical equivalence of two propositions `(A : Prop)` and `(B : Prop)`, whose inhabitants are pairs of functions between `A` and `B` (in one direction and the other):

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

   (e) Give the type of the dependent eliminator for `iff` in `Type`:

   ```
   iff_rect_dep : forall A B (P : iff A B -> Type), ...
   ```

   (f) Define a projection function from `iff A B` to `A -> B` using this eliminator.

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

```
Inductive i1 : Type := c1 : i1 -> i1.
Inductive i2 : Type := c2 : (i2 -> bool) -> i2.
Inductive i3 : Type := c3 : (i1 -> bool) -> i3.

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

4. Define a closed term of the following type. Definitions of the existential types are in the appendix, §4.

```
Definition t1 : { x : nat | x = 3 } := <todo>.
```

5. Which of the following terms can be inhabited? (We do not ask for proof terms here).

```
Definition t2 (P : nat -> Prop) : (exists x : nat, P x) -> { x : nat | P x } := ...

Definition t3 (P : nat -> Prop) : { x : nat | P x } -> exists x : nat, P x := ...
```

6. (*) Replace the placeholders to define symmetry of equality:

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

2 Modelisation and Programming in Coq (7 points)

We take the type of binary tree with natural number labels:

```
Inductive tree : Type :=
| L : nat -> tree
| N : nat -> tree -> tree -> tree
```

1. Write a recursive function `mult` of type `tree -> tree` in Coq that compute the product of all elements of the tree.

2. Define inductively a predicate `in0` on trees that expresses that the natural `0` appears in the tree.

3. To compute less multiplications, we want to develop a function `mult0` that returns an exception when it finds 0 in the tree and otherwise returns the multiplication of the elements (i.e, the same thing as `mult`). We model exceptions using the `option` type and propose the following implementation:

```
Fixpoint mult0 (t : tree) : option nat :=
  match t with
  | leaf 0 => None
  | leaf n => Some n
  | node 0 l r => None
  | node n l r =>
    match mult0 l with
    | Some l' => match mult0 r with
       | Some r' => Some (n * l' * r')
       | None => None
      end
    | None => None
    end
  end.
```

We can rephrase this function more abstractly using monadic combinators of the `option` monad:

```
Definition unit {A} : A -> option A := Some.
Definition bind {A B} (m : option A) (f : A -> option B) : option B :=
  match m with
  | Some x => f x
  | None => None
  end.
Definition raise {A} : option A := None.
```

Complete the equivalent definition of `mult0` using the abstract monadic combinators:

```
Fixpoint mult0 (t : tree) : option nat :=
  match t with
  | leaf 0 => raise
  | leaf n => unit n
  | node 0 l r => <todo>
  | node n l r =>
    bind (mult0 l) (fun l' => <todo>)
  end.
```

4. Remark that when 0 is in the tree, the computation does not stop but propagates the exception recursively. To avoid this behavior, we propose another monadic transformation based on the combinations of the *exception* and *continuation* monads:

$$M\,A \equiv \forall C, (A \to C) \to C \to C$$

Each computation on A is interpreted as a term of type $M\,A$. This term takes as argument a program f of type $A \to C$ (the "success" continuation, for normal program behavior) and a program x of type C (the "failure" continuation, for abnormal behavior).

If the computation terminates normally on a value a then the result will be $f\,a$. Otherwise, if the computation terminates in an exceptional fashion, it will return x.

Question: What is the link between the inductive type `option A` and the type $M\,A$?

5. Define the monadic operations `unit : A -> M A`,

`bind : M A -> (A -> M B)-> M B` and `raise : forall A, M A` for the monad `M`.

6. Define `catch : forall A B, M A -> A -> A` such that `catch e_1 e_2` gives back the value of e_1 if it does not raise an exception, and otherwise returns e_2.

7. Assuming the variant of `mult0` defined at the end of question 5 uses the `M` monad instead of the `option` monad, define using `catch` a function `mult1 : tree -> nat` that computes the product of elements in a tree. Its behavior should be equivalent to `mult0`.

3 Hilbert's ε operator (6 points)

The indefinite description operator ε of Hilbert allows one to dissociate the denotation of an object from the conditions of existence of such an object. Using it, we can for example talk about $\log_2(x)$ in mathematical discourse as an abbreviation of « a number n, if it exists, such that $2^n = x$ ». The idea is that we can always write $\log_2(x)$ but cannot say anything about its value when x is not a power of 2 (i.e. if there is actually no such n such that $2^n = x$).

More precisely, a ε-term is of the shape $\varepsilon\ a\ P$ where `P: A -> Prop` is a predicate and a an arbitrary object of type A. The interpretation of $\varepsilon\ a\ P$ is « a x satisfying P, if it exists, otherwise the arbitrary object a ».

We introduce the operator ε in Coq with the following definitions :

```
Parameter epsilon : forall (A:Type), A -> (A -> Prop) -> A.
Axiom epsilon_spec :
  forall (A:Type) (a:A), forall P, (exists x, P x) -> P (epsilon A a P).
```

1. The second argument of `epsilon` of type `A` guarantess that the type `A` is inhabited. Justify this condition.

2. Define a fonction `power2 : nat -> nat` such that `power2 n = 2^n` and deduce using the ε operator a function `power2_inverse : nat -> nat` which denotes the inverse of this function, when it exists.

3. Give a natural eliminator for the ε operator, of partial shape:

   ```
   forall (A:Type) (a:A) (P Q:A -> Prop), ... -> Q (epsilon A a P)
   ```

4. Given this eliminator, define an injection from `ex` to `sig` on boolean predicates:

   ```
   Definition epsilon_bool_subset (P : bool -> Prop) :
     (exists x : bool, P x) -> { x : bool | P x }.
   ```

   Hint: use the ε operator to inhabit the witness of `{ x : bool | P x }`.

   Here and in the following questions, you can either give an informal proof in english or a Coq-like proof script detailing the steps of the proof construction.

5. (*) Using the previous definition, establish that:

   ```
   forall (P Q : Prop), P \/ Q -> {P} + {Q}
   ```

   Hint: consider the boolean predicate `fun b => if b then P else Q`.

   What can we conclude about the distinction `Prop`/`Type` and the associated extraction mechanism? In simpler terms, what can we say about `epsilon` with respect to extraction ?

6. (***) Likewise, use the ε operator and its specification to establish a proof of:

   ```
   forall (A B:Prop), (A -> B) \/ (B -> A).
   ```

Hint: consider the boolean predicate `fun x => if x then A else B` and the fact that one can do case-analysis on an element `epsilon bool _ P : bool`.

7. From this, deduce a proof of `forall (A:Prop), ~ A / ~~ A`

   Was this a provable proposition in the Calculus of Inductive Constructions? Why?

4 Annex

- Notation `{ x : A | B x }` unfolds to `sig A B`, the type for existentials in `Type` with a first component in `Type` and second component in `Prop`:

```
Inductive sig (A : Type) (B : A -> Prop) : Type :=
| exist (x : A) (p : P x) : sig A B
```

- Notation `exist x : A, B x` unfolds to `ex A B`, the type for existentials in `Prop` with a first component in `Type` and second component in `Prop`:

```
Inductive ex (A : Type) (B : A -> Prop) : Prop :=
| ex_intro (x : A) (p : P x) : ex A B
```

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

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

- Notation `A \/ B` unfolds to `or A B`, the type for disjunction in `Prop`:

```
Inductive or (A B : Prop) : Prop :=
| or_introl : A -> or A B
| or_intror : B -> or A B
```

- Notation `A + B` unfolds to `sumbool A B`, the type for disjunction in `Type` labelled by propositions:

```
Inductive sumbool (A B : Prop) : Type :=
| left : A -> sumbool A B
| right : B -> sumbool A B
```

- Notation `~A` unfolds to `neg A`, the type for negation in `Prop`:

```
Definition not (A : Prop) := A -> False.
```

  with `False` defined as:

```
Inductive False :=.
```

  The associated elimination principle is:

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

- The option type is defined as:

```
Inductive option (A : Type) : Type :=
| Some : A -> option A
| None : option A
```

- 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
```

- 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.