Master Parisien de Recherche en Informatique 2012–2013

Exam course 2-7-2 Proof assistants

March 11th 2013

The subject is 2 pages long. The exam lasts 2 hours. Hand-written course notes and other course material distributed this year are the only documents that you can use. The exercises can be solved independently.

The exercises require to write Coq terms; we allow flexibility regarding the syntax used as long as there is no ambiguity on its meaning.

1 Questions (6 points)

1.1. What is the shortest proof of $3 + 3 = 6$ in Coq (with the usual definition of addition) ?

1.2. Give the type of recursive eliminator (`ord_rect` and `Wd_rect`) automatically generated by Coq for the following inductive declarations:

```
Inductive ord :=
| Oo : ord
| So : ord -> ord
| Limo : (nat -> ord) -> ord.

Parameter A:Type.
Parameter B:A->Type.
Parameter C:Type.
Parameter f:forall x:A, B x -> C.
Parameter g:A->C.
Inductive Wd : C->Type := Sup : forall x:A, (forall i:B x, Wd (f x i)) -> Wd (g x).
```

1.3 Write an inductive predicate on triples of lists that holds when the third list is a shuffle of the first two (i.e. is a permutation of the disjoint union of the 2 lists that respect the order induced by each list).

2 Primitive recursive functions (6 points)

Primitive recursive functions (PRFs) are the numeric functions (of arbitrary arity) generated by:

- the 0-ary functions returning 0,
- the unary successor function,
- the projection functions $p_i(x_1, \ldots, x_n) = x_i$ (of arity $n > i$),
- the composition of a PRF of arity k with k PRFs of arity n is a PRF of arity n,
- and given a k-ary function f and a $(k+2)$-ary function, there exists a PRF h obtained by primitive recursion over its first argument:

$$\begin{aligned} h(0, x_1, \ldots, x_k) &= f(x_1, \ldots, xn) \\ h(S(n), x_1, \ldots, x_k) &= g(n, h(n, x_1, \ldots, x_k), x_1, \ldots, x_k) \end{aligned}$$

Questions:

2.1. Write an inductive definition of type `nat->Type` that represents the expressions of PRFs of a given arity. It is **not** required to enforce that all expressions are well-formed w.r.t. arity.

2.2. Prove that the identity function is a PRF, i.e. give an expression of the above type that represents the identity function.

2.3. Write an evaluation function that takes as input a PRF and returns its semantics as a function of type `list nat -> nat`, where the list represents the value for each argument of the function. When the length of the list does not match the arity, the function may return an arbitrary natural number.

3 Modulo (8 points)

3.1 Structural recursion

Assume we want to prove the specification of the modulo operation on natural numbers like this:

```
Lemma mod : forall (a:nat) (b:nat) (_:0<b), {r:nat|exists q, a=b*q+r}.
```

Questions:

3.1.1. Is the specification correct, in the sense that any proof of the lemma above provides a correct algorithm of modulo ? If not, give a correct specification.

3.1.2. Write a `Program` command that provides a simple implementation for this specification, by structral induction on the first argument.

```
Fixpoint mod (a:...) (b:...) {struct a} : ... := ...
```

Giving the proof of the auxiliary obligations (i.e. the proofs that there exists q such that...) is **not** required.

3.1.3. What happens if we evaluate (normalize) `mod 3 0` ? Does it loop forever produce a result ? If it does produce a result, is it of the form of a natural number r and a proof that it is the correct result ?

3.2 Well-founded recursion

We now want to define modulo using the equations

$$a \bmod b = (a - b) \bmod b \quad (\text{if } b \leq a) \qquad \text{and} \qquad a \bmod b = a \quad (\text{if } a < b)$$

regardless of the value of b.

In Coq, we are going to define the program using a well-founded relation.

3.2.1. Define the order R corresponding to the recursive calls needed for the above algorithm. Argument b of the modulo function will be taken as a parameter of the relation.

3.2.2. Is this order well founded for all values of b ?

3.2.3. Write the corresponding program.

```
Fixpoint mod' (a:...) (b:...) (h:Acc (R b) a) {struct h} : ... := ...
```

As above the auxiliary proofs are not required.

3.2.4. What happens if we evaluate (normalize) `mod' 3 0` ? Does it loop forever produce a result ? If it does produce a result, is it of the form of a natural number r and a proof that it is the correct result ?