Master Parisien de Recherche en Informatique 2011–2012

Exam course 2-7-2 Proof assistants

February 29th 2012

The subject is 4 pages long. The exam lasts 3 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 Programming with Coq: Lists and Paths (7 points)

We recall that 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 empty list is denoted `[ ]` and the `cons` constructor has an infix notation `_ :: _`. The library provides a concatenation operation `app`, equipped with an infix notation `_ ++ _`.

1. 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`.

2. 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`.

3. 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`.

4. 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.

5. 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].

6. Program in Coq a boolean predicate path of type:

   ```
   path : forall A : Type, forall e : A -> A -> bool, A -> list A -> bool
   ```

   such that (path e x p) tests whether two consecutive elements of (x :: p) are always related by the relation e, with default value true if p is empty.

7. Use path and rcons to define a boolean predicate cycle of type

   ```
   cycle : forall A : Type, forall e : A -> A -> bool, list A -> bool
   ```

   such that(cycle e p) tests whether p is a cycle for the relation e.

8. State in Coq the property expressing that an arbitrarily rotated cycle remains a cycle.

2 Modelization in Coq: a Tiny Programming Language (5 points)

We define the following toy language:

```
Inductive tm : Type :=
  | tm_cst : nat -> tm
  | tm_plus : tm -> tm -> tm.
```

1. Fill the complete induction scheme generated by Coq at definition time for this type:

   ```
   tm_ind : forall P : tm -> Prop, ...
   ```

2. Informally, what are the expression modelled by tm? Program an evaluation function eval : tm -> nat. We recall that the standard library of Coq defines an addition operation plus : nat -> nat -> nat, equipped with the + infix notation.

3. Write an inductive predicate step : tm -> tm -> Prop modelling a small step evaluation for this language. The statement (step t1 t2) expresses that t1 evaluates to t2 in a single step, with a left to right strategy. This inductive predicate should have three constructors corresponding to the three situations where a reduction step is possible:

   - when t1 is (tm_plus (tm_cst n1) (tm_cst n2)),
   - when t1 is (tm_plus t1' t2') and a reduction step is possible in t1',
   - or when t1 is (tm_plus (tm_cst n1) t2) and a reduction step is possible in t2.

4. State in Coq the property that if t1 evaluates to t2 and t2' in a single step, then t2 and t2' should be equal. Give a scheme of the proof (no proof script required!).

3 Encoding Zermelo Set Theory (8 points)

In set theory, a set is characterized by its elements. Also, sets are well-founded: for any set x there is no infinite sequence $(x_n)_{n\in\mathbb{N}}$ such that $\ldots x_{n+1} \in x_n \in \ldots x_1 \in x$. This gives the idea that a set can be encoded as a well-founded tree. The sons of a node represent the direct elements of a set. The arity is characterized by the number of elements of the sets. At each stage of the construction the arity may be different.

This suggests to use the following inductive definition and accessor functions:

```
Inductive set : Type :=
  Elts : forall X:Type, (X->set) -> set.
Definition idx (x:set) : Type := let (X,f) := x in X.
Definition elt (x:set) : idx x -> set :=
  match x return idx x -> set with Elts X f => f end.
```

A set x is therefore a pair (X, f) where X is the index type, and f is a function such that the image of X by f describes exactly the elements of x.

1. Give the dependent elimination scheme of `set`.

Two sets are equal if they contain exactly the same elements, regardless of the way they are indexed. This means that $x = (X, f)$ and $y = (Y, g)$ are the same set iff for every index $i : X$ there is an index $j : Y$ such that $f(i)$ is the same set as $g(j)$, and conversely, for every index of $j : Y$ there is an index of $i : X$ such that $g(j)$ is equal to $f(i)$. It might be the case that X and Y are nor the same type. Membership can be defined using equality: $x \in y$ iff there is an element of y which is equal to x. We recall the definition of some logical connectives of the standard library:

```
Inductive and (A B:Prop) : Prop :=
  conj : A -> B -> and A B. (* A/\B is a notation for (and A B) *)
Inductive ex (A:Type) (P:A->Prop) : Prop :=
  ex_intro : forall x:A, P x -> ex A P.
  (* (exists x:A, P x) is a notation for (ex A P) *)
```

2. Write a recursive function of type `set->set->Prop` that indicates whether two sets are equal, and the membership function, of the same type.

3. Suggest how, naively, we could try to build the set of all sets. Explain why it is not possible (i.e. why Coq will reject this attempt).

Now the goal is to give a representation for all the constructions of Zermelo set-theory. They are the following:

- The empty set $\emptyset$ which contains no element.
- The (unordered) pair $\{x; y\}$ which contains exactly x and y.
- The union: $\bigcup x$ is the union of all the sets belonging to x. In other words, the elements of $\bigcup x$ are exactly the elements of the elements of x.
- The comprehension scheme $\{y \in x \mid P(y)\}$: the set of the elements of x that satisfy P (of type `set->Prop`).

- The power-set: $\mathcal{P}x$ is the set of all the sets $y \subseteq x$. Each element y of the power-set can be identified by its characteristic function, which is the predicate on the elements of x that holds for only for the elements of y.
- An infinite set, for instance the set of natural numbers where 0 is the empty set, and the successor of n is $\{x; \{x\}\}$.

4. Define a function for each of the set constructors of Zermelo set theory. For the power-set case, beware not to be too close to the counter-example of question 3. For the set of natural numbers, first write a recursive function that translates a `nat` into a `set`. In several cases, using the following Σ-type (type of dependent pairs) will be useful:

```
Inductive sigT (A:Type) (P:A->Type) : Type := (* also applies to P:A->Prop *)
  existT : forall x:A, P x -> sigT.
  (* { x:A & P x} is a notation for (sigT A P) *)
```