Exam course 2-7-2 Proof assistants
Tuesday March 1 2011

The subject is ?? 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.

Exercises 1 and 3 require to write Coq terms and proofs; we allow flexibility regarding the syntax used as long as there is no ambiguity on its meaning.

1 Programming with Coq: a binary scheme (9 pts)

The type positive in Coq is a representation of non-null natural numbers in a binary format. More precisely, there is a constant constructor xH which represents the natural number 1 and two constructors xI and xO which take a positive and return a positive. If p is a positive which represents the natural number n then xIp represents 2n + 1 and xOp represents 2n.

1. Write in Coq the inductive definition of positive and give the type of the corresponding induction principle on the sort Type.

2. Given a set A and a binary operation h on A, one defines for each x in A and 0 < n, the iterated composition h^n x of h by recursion on n ∈ nat:

   \[ h^1 x = x \quad h^{n+1} x = h (h^n x) \]

   Define in Coq a term it that, given h, x, n, computes h^n x.

3. From now on, one assumes that h is associative. Prove in Coq that

   \[ \forall x \in A, \ 0 < n \rightarrow h^{2n} x = h^n (h x x) \]

4. The type positive gives a fast algorithm to compute h^n x using the properties :

   \[ h^{2n} x = h^n (h x x) \quad h^{2n+1} x = h x (h^n (h x x)) \]

   To obtain a terminal recursive version, one introduces an extra variable s as accumulator. The fast iteration Fit uses the algorithm:

   \[ \text{Fit} \ h \ s \ x \ 1 = h \ s \ x \]

   \[ \text{Fit} \ h \ s \ x \ (2n) = \text{Fit} \ h \ s \ (h x x) \ n \quad \text{Fit} \ h \ s \ x \ (2n + 1) = \text{Fit} \ h \ (h x x) \ (h x x) \ n \]

   One assumes h has a left neutral element \( \epsilon \) (i.e. \( h \epsilon x = x \)) and one starts with \( s = \epsilon \).

   (a) Define the function Fit in Coq using the type positive to represent the number n.
(b) Using the function nat_of_P which transforms an object in positive into the corresponding object in nat, give a specification for Fit which links (Fit h s x p) with s and (it h x (nat_of_P p))

(c) Give the main element of the proof that your implementation of Fit satisfies this specification.

(d) Deduce a function Pit which, given h, x, and p : positive, computes (it h x (nat_of_P p)).

(e) Assuming addition on positive is given by a function Pplus, how to instantiate this scheme in order to compute \( x^n \) when both \( x \) and \( n \) are in the type positive?

## 2 Imperative programming and invariants (10 pts)

One introduces in WHY a logical environment for modeling finite sets with predicates to test membership and equality; a constant to represent the empty set, and logical operations to add (resp. remove) an element \( x \) to a set \( s \).

```why

**Type Definition**

```type set
logic emptyset : set
logic memset : int . set \rightarrow\ prop \ (\star x \ in s \ \star)
logic eqset : set . set \rightarrow\ prop \ (\star equality\ between\ sets \ \star)
logic addset : int . set \rightarrow\ set \ (\star s + \{x\} \ \star)
logic remset : int . set \rightarrow\ set \ (\star s - \{x\} \ \star)
```

One assumes that the usual properties relating these operations and predicates are given as axioms. On top of this theory, one introduces a reference of type set and operations to clear this set, add a (positive) element, and pick an element in a (non-empty) set.

One introduces the following WHY environment (named \( \Gamma_1 \)):

```why

**Parameter Declarations**

```parameter s : set ref
parameter clear : unit \rightarrow
{ } unit writes s { eqset(s.emptyset) }
parameter add : x : int \rightarrow
{ x \geq 0 } unit writes s { eqset(s.addset(x,s@)) }
parameter pick : unit \rightarrow
{ not eqset(s.emptyset) }
int writes s
{ memset(result.s@) and eqset(s.remset(result.s@)) }
```

Reminder: in the post-condition of a function, \( s@ \) designates the old value contained in reference \( s \) at the entry point of the function.

1. Let \( e \) be the WHY expression:

```why

clear(): add(2): add(3): pick()
```

Justify that the post-condition \( \{ result = 2 \text{ or } result = 3 \} \) is satisfied after this expression is executed.

2. Assume there is another function add' with a different specification

```why

parameter add' : x : int \rightarrow unit writes s { eqset(s.addset(x,s@)) }
```

Explain why using add' instead of add does not change the behavior of the expression \( e \).
3. More generally, let \( e \) be an expression that satisfies a post-condition \( R \) in an environment with a function \( f \) and which possibly writes variable in a set \( V \):

\[
\text{parameter } f : x : \tau \mapsto \\
\{ P(\text{vars}) \} \sigma \text{ writes vars } \{ Q(x.\text{result}., \text{vars}@., \text{vars}) \}
\]

Assume there is another function

\[
\text{parameter } f' : x : \tau \mapsto \\
\{ P'(\text{vars}') \} \sigma \text{ writes vars' } \{ Q'(x.\text{result}., \text{vars}@', \text{vars'}) \}
\]

Explain the conditions on the properties \( P, Q, P', Q' \), and the sets of references \( \text{vars} \) and \( \text{vars}' \) such that the parameter \( f \) can be replaced by \( f' \) without changing the behavior of \( e \).

4. One introduces the property

\[
\text{predicate } \text{Inv}(s : \text{set}) = \forall n : \text{int}. \text{memset}(n, s) \rightarrow n \geq 0
\]

Show that the functions clear, add, and pick, also satisfy the specification where \( \text{Inv}(s) \) is added both in pre and post-conditions (only in post for the clear function). Namely the same implementations could be given the specifications:

\[
\text{parameter } \text{clear} : \text{unit} \mapsto \\
\{ \} \text{ unit writes vars } \{ \text{eqset}(s.\text{emptyset}) \text{ and } \text{Inv}(s) \}
\]

\[
\text{parameter } \text{add} : x : \text{int} \mapsto \\
\{ x \geq 0 \text{ and } \text{Inv}(s) \} \text{ unit writes vars } \{ \text{eqset}(s.\text{addset}(x,s@)) \text{ and } \text{Inv}(s) \}
\]

\[
\text{parameter } \text{pick} : \text{unit} \mapsto \\
\{ \text{not } \text{eqset}(s.\text{emptyset}) \text{ and } \text{Inv}(s) \}
\]

\[
\text{int writes vars } \\
\{ \text{memset}(\text{result}.s@) \text{ and } \text{eqset}(s.\text{remset}(\text{result}.s@)) \text{ and } \text{Inv}(s) \}
\]

We call \( \Gamma_2 \) this new environment.

5. Show that if \( e \) is an expression well-formed in the initial environment \( \Gamma_1 \) that establishes the post-condition \( R \), and if \( e \) does not contain an assignment of the form \( s := b \) then it can be run in the environment \( \Gamma_2 \) of question 4 and assuming the pre-condition \( \text{Inv}(s) \), the expression \( e \) will establish the post-condition \((R \text{ and } \text{Inv}(s)) \). The expression \( e \) is supposed to be built using application of functions, conditionals, sequences, and assignments. The only functions doing effects on the parameter \( s \) are clear, add, and pick.

6. Give an example of expression \( e \) that contains an assignment on \( s \) and such that the program \( e \) is correct in the environment \( \Gamma_1 \) but fails in the environment \( \Gamma_2 \).

7. In order to allow arbitrary updates, one introduces a boolean variable \( \text{Invb} \) which, when true, ensures the invariant is satisfied. So we have the environment:

\[
\text{parameter } \text{invb} : \text{bool ref}
\]

\[
\text{parameter } \text{clear} : \text{unit} \mapsto \\
\{ \} \text{ unit writes vars } \{ \text{eqset}(s.\text{emptyset}) \text{ and } \text{Inv}(s) \}
\]

\[
\text{parameter } \text{add} : x : \text{int} \mapsto \\
\{ x \geq 0 \text{ and } \text{Inv}(s) \text{ and } \text{invb} = \text{true} \}
\]

\[
\text{unit writes vars } \\
\{ \text{eqset}(s.\text{addset}(x,s@)) \text{ and } \text{Inv}(s) \}
\]

\[
\text{parameter } \text{pick} : \text{unit} \mapsto \\
\{ \text{not } \text{eqset}(s.\text{emptyset}) \text{ and } \text{Inv}(s) \text{ and } \text{invb} = \text{true} \}
\]
int writes s
{ memset(result,s@) and eqset(s,remset(result,s@)) and Inv(s) }

parameter update: u : set ->
{ invb = false } unit writes s { eqset(s,u) }

We also add two functions which change the value of invb. The parameter invb can be set to true only when the invariant is proven.

parameter pack : unit -> { Inv(s) } unit writes invb { invb = true }
parameter unpack : unit -> { } unit writes invb { invb = false }

Show that any expression e well-formed in that environment (using update, pack, unpack as well as add, clear, pick) and which does not assign directly s and invb preserves the property invb = true -> Inv(s).

3 Impredicative and inductive encodings of sum (4 pts)

One considers an environment

Variable A : Set.
Variable P : A -> Set.

In this environment, an impredicative encoding of an indexed sum is given by:

Definition sum := forall C : Set. (forall x : A, P x -> C) -> C.

1. Write a Coq term sumi of type forall x:A. P x -> sum and another of type sum -> A.
2. Write the indexed sum as an inductive definition named sumind.
3. Write a Coq term ind of type sumind -> A and a term of type:
4. Using ind, propose a new term pi of type sum -> A such that forall (p : sum). P (pi p) is also provable.

Reminder

Weakest precondition computation

The weakest precondition WP(i, Q) can be computed by induction on i:

\[
\begin{align*}
WP(x := e, Q) &= Q[x ← e] \\
WP(i_1; i_2, Q) &= WP(i_1, WP(i_2, Q)) \\
WP(\text{if } e \text{ then } i_1 \text{ else } i_2, Q) &= (e = true ⇒ WP(i_1, Q)) \land (e = false ⇒ WP(i_2, Q)) \\
WP(f e, Q) &= pre(f)[x ← e] \land (\forall \text{result }\omega, (post(f)[x ← e] ⇒ Q)) [\omega@ ← \omega]
\end{align*}
\]