CIS 500 — Software Foundations

Midterm I

(Standard and advanced versions together)

February 13, 2013

Answer key

1. (12 points) Write the type of each of the following Coq expressions, or write "ill-typed" if it does not have one.

(a) `fun n:nat => [n]::nil`

Answer: `nat -> list (list nat)`

(b) `forall X (l1 l2 : list X), l1 :: l2 = l2 :: l1`

Answer: ill-typed

(c) `forall X, X -> list X -> list X`

Answer: `Type`

(d) `and False`

Answer: `Prop -> Prop`

(e) `fun n1 => if beq_nat n1 0 then ble_nat 5 else beq_nat n1`

Answer: `nat -> nat -> bool`

(f) `fun n : nat => forall m:nat, ble_nat m n = true`

Answer: `nat -> Prop`

2. (12 points) For each of the types below, write a Coq expression that has that type.

(a) `nat -> bool -> nat`

Possible answers:
`fun (n : nat) (b : bool) => n`,
`fun (n : nat) (b : bool) => if b then n else n + 1`
...

(b) `forall X, X -> list X`

Possible answers:
`fun X (x : X) => [x]`
`fun X (x : X) => nil`

(c) `forall X Y : Type, X -> (X -> Y) -> Y`

Possible answers:

`fun X Y => fun (x : X) => fun (f : X -> Y) => f x`

(d) `Prop`

Possible answers:

```
True
False
forall n:nat, n = n
...
```

(e) `forall X:Prop, X \/ X -> X`

Answer:

```
fun X (H : X \/ X) =>
  match H with
  | or_introl HX => HX
  | or_intror HX => HX
  end.
```

(f) `forall X:Prop, X /\ ~ X -> False`

Answer:

```
fun X (H : X /\ ~ X) =>
  match H with
  | conj H1 H2 => H2 H1
  end.
```

3. (10 points) Suppose that we wanted to prove the following theorem:

```
Theorem beq_nat_true : forall m n : nat, beq_nat m n = true -> m = n.
```

(The definition of `beq_nat` is given in the appendix, for easy reference.)

(a) What induction hypothesis will be generated by `induction` for the second subgoal (the induction step) if we start this way (doing `intros` on both `m` and `n`)?

```
Proof.  intros m n. induction m as [|m'].
```

Answer: `beq_nat m' n = true -> m' = n`

(b) What induction hypothesis will be generated by `induction` for the second subgoal (doing `intros` on just `m`)?

```
Proof.  intros m. induction m as [|m'].
```

Answer: `forall n, beq_nat m' n = true -> m' = n`

(c) Which of these two strategies is more likely to succeed? Why? *Answer:* The second one is the correct one. If we start with tactics in the first item, on the case where `m = S m'`, we have the hypothesis `beq_nat (S m') n = true`, which doesn't help us with the provided induction hypothesis.

4. **[Standard]** (8 points) Briefly explain the difference between the `apply` and `rewrite` tactics. (3-4 sentences at the most.)

Grading scheme:

- *Important differences: (3 each)*
  - *Apply doesn't require an equality/equivalence, rewrite does*
  - *Apply discharges the current goal (rewrite only substitutes only part of the current goal)*
- *Smaller differences: (2 each)*
  - *Rewrite is more syntactic (apply simpl a bit first)*
- *Minor: (no points, unless very small grade) - 1 each*
  - *both can generate subgoals*
  - *applied to hypothesis or separate lemma*
  - *rewrite can be applied in both sense*

5. **[Standard]** (6 points) For each of the given theorems, which set of tactics is needed to prove it? (If more than one of the sets of tactics will work, choose the smallest set.)

(a) `forall n m : nat, beq_nat m n = true -> beq_nat (S m) (S n) = true`

i. `intros`, `simpl`, `rewrite`, `reflexivity`, and `induction`
ii. `intros`, `simpl`, `rewrite`, and `reflexivity`
iii. `intros`, `rewrite`, and `reflexivity`
iv. `intros` and `reflexivity`

Answer: i

(b) `forall (B : Prop), exists (A : Prop), A -> B`

i. `intros`, `exists`, and `rewrite`
ii. `intros`, `exists`, and `apply`
iii. `intros` and `exists`
iv. `intros` and `rewrite`

Answer: ii

(c) `forall n, n + 0 = 0 -> n = 0`

i. `intros`, `rewrite`, `induction`, and `inversion`
ii. `intros`, `rewrite`, and `reflexivity`
iii. `intros`, `destruct`, and `reflexivity`
iv. `intros`, `destruct`, `inversion` and `reflexivity`

Answer: iv

6. **[Standard]** (10 points) A "fold function" captures a very common computation pattern: iterating over a data structure while accumulating a result. For instance, here is how you can use the fold function on lists (which we called simply `fold` in the notes) to sum all the elements of a list:

```
fold plus 0 [1,2,3,4]
```

When evaluated, this expression simplifies to `10`.

We can also write fold functions for other kinds of data structures. For example, consider the following definition of binary trees:

```
Inductive tree (X : Type) :=
| leaf : tree X
| node : X -> tree X -> tree X -> tree X.
```

In this problem, we will write a `fold_tree` function to perform the same pattern of iteration as in the list case. The type of `fold_tree` will be:

```
fold_tree : forall X Y, (X -> Y -> Y -> Y) -> Y -> tree X -> Y
```

Here is an example application of `fold_tree`:

```
fold_tree (fun b n1 n2 => b + n1 + n2)
          0
          (node 4
             (node 3 leaf leaf)
             (node 1 leaf leaf))
```

When evaluated, this expression simplifies to `8`.

Complete the definition of `fold_tree` below.

```
Fixpoint fold_tree {X Y} (f : X -> Y -> Y -> Y)
                         (y : Y) (t : tree X) : Y :=
```

Answer:

```
match t with
| leaf => y
| node x lt rt => f x (fold_tree f y lt) (fold_tree f y rt)
end.
```

Grading scheme:
Only base case: 2pt
Implicit argument mistakes -1pt
Base case and kind of recursive calls: 4pt
wrong base case -4pt
only top level pattern match 1pt

7. [**Advanced**] (12 points) Write a *careful* informal proof of the following theorem. Make sure to state the induction hypothesis explicitly in the inductive step.

Theorem: `beq_nat m n = beq_nat n m`, for all natural numbers `m` and `n`.
Answer: We show, by induction on `n`, that `beq_nat m n = beq_nat n m`, for all natural numbers `m`. There are two cases to consider:

- `n = 0`: If `m = 0`, then the theorem simplifies to `true = true` by the definition of `beq_nat`. Otherwise, `m = S m'` for some `m'`, and the goal simplifies to `false = false` by the definition of `beq_nat`.

- `n = S n'`, with `beq_nat m n' = beq_nat n' m` for all `m`. We must show that

  ```
  beq_nat m (S n') = beq_nat (S n') m
  ```

  for all `m`.

  If `m = 0`, then the goal simplifies to `false = false` by the definition of `beq_nat`.

  Otherwise, `m = S m'` for some `m'`. In this case the goal simplifies to

  ```
  beq_nat m' n' = beq_nat n' m'
  ```

  by the definition of `beq_nat`, which is an instance of the IH.

8. (10 points) In the first and second homework assignments, we worked with *binary numbers* encoded as a Coq inductive type. Here is one version of that encoding:

```
Inductive bin : Type :=
  | BZ : bin
  | T2 : bin -> bin
  | T2P1 : bin -> bin.
```

For example, `T2P1 (T2P1 BZ))` represents the number 3, while `T2 (T2 (T2P1 BZ))` represents 4.

(a) Recall that the `incr` function takes a binary number and returns its successor. For example, `incr (T2P1 (T2P1 BZ) = T2 (T2 (T2P1 BZ))`.

Complete the following definition of `incr`:

```
Fixpoint incr (m:bin) : bin :=
```

Answer:

```
match m with
| BZ      => T2P1 BZ
| T2 m'   => T2P1 m'
| T2P1 m' => T2 (incr m')
end.
```

(b) The `bin_to_nat` function takes a binary number and returns its `nat` (unary) representation. For example, `bin_to_nat (T2P1 (T2P1 BZ)) = S (S (S O))`.

Complete the following definition of `bin_to_nat`:

```
Fixpoint bin_to_nat (m:bin) : nat :=
```

Answer:

```
match m with
| BZ      => O
| T2   m' => 2 * bin_to_nat m'
| T2P1 m' => 1 + 2 * bin_to_nat m'
end.
```

9. (12 points) In this problem, your task is to find a short English summary of the meaning of a proposition defined in Coq. For example, if we gave you this definition...

```
Inductive D : nat -> nat -> Prop :=
  | D1 : forall n, D n 0
  | D2 : forall n m, (D n m) -> (D n (n + m)).
```

... your summary could be "`D m n` holds when `m` divides `n` with no remainder."

(a)
```
Inductive R (X : Type) : X -> list X -> Prop :=
  | R1 : forall x l, R x (x::l)
  | R2 : forall x y l, (R x l) -> (R x (y::l)).
```

`R X x l` holds when `x` occurs in the list `l`

(b)
```
Inductive R (X : Type) : list X -> list X -> Prop :=
  | R1 : R [] []
  | R2 : forall x l1 l2, (R l1 l2) -> (R l1 (x::l2))
  | R3 : forall x l1 l2, (R l1 l2) -> (R (x::l1) (x::l2)).
```

`R X l1 l2` holds when `l1` is a subsequence of `l2`

(c)
```
Inductive R (X : Type) : list X -> list X -> Prop :=
  | R1 : R [] []
  | R2 : forall x l1 l2 l3 l4,
           (R (l1++l2) (l3++l4)) ->
           (R (l1++[x]++l2) (l3++[x]++l4)).
```

`R X l1 l2` holds when `l1` is a permutation of `l2`

(d)
```
Definition R (m : nat) :=
  m > 1 /\ (forall n, 1 < n -> n < m -> ~(D n m)).
```
(where `D` is given at the top of the page).

`R m` holds when `m` is prime

10. [**Advanced**] (12 points) *Regular expressions* are a convenient way of describing sets of strings. Here's a definition of regular expressions (where the "characters" in the strings are numbers) as a Coq data type.

```
Inductive regex : Type :=
  | Literal : list nat -> regex
  | Union   : regex -> regex -> regex
  | Concat  : regex -> regex -> regex
  | Star    : regex -> regex.
```

Informally, a regular expression `r` matches a list of numbers `s` according to the following rules:

- `Literal s` only matches `s` itself.
- `Union r1 r2` matches strings that are matched by either `r1` or `r2`.
- `Concat r1 r2` matches by strings of the form `s1 ++ s2`, such that `s1` is matched by `r1` and `s2` is matched by `r2`.
- `Star r` matches a string `s` if `s = []` or if `s = s1 ++ s2 ++ ... ++ sn` and each substring is matched by `r`.

Your task (on the next page) will be to formalize the above specification by translating it into an inductive relation `matches` of type `regex -> list nat -> Prop`. For instance, the following propositions should be provable...

```
matches (Literal [1,2,3])                        [1,2,3]

matches (Union (Literal []) (Literal [2,1]))     []

matches (Concat (Literal [1,2,3]) (Literal [1])) [1,2,3,1]

matches (Star (Literal [1]))                     [1,1,1,1,1]
```

...whereas the following shouldn't hold:

```
matches (Literal [1])                            [1,2,3]

matches (Star (Literal [2]))                     [1]

matches (Concat (Literal [3]) (Literal [3]))     [3,3,3]
```

Complete the definition of `matches` below.

```
Inductive matches : list nat -> regex -> Prop :=

  | MLiteral : forall s,
      matches (Literal s) s
  | MUnionL : forall s r1 r2,
      matches r1 s -> matches (Union r1 r2) s
  | MUnionR : forall s r1 r2,
      matches r2 s -> matches (Union r1 r2) s
  | MConcat : forall s1 r1 s2 r2,
      matches r1 s1 -> matches r2 s2 -> matches (Concat r1 r2) (s1 ++ s2)
  | MStarEmpty : forall r,
      matches (Star r) []
  | MStarApp : forall s1 s2 r,
      matches r s1 -> matches (Star r) s2 -> matches (Star r) (s1 ++ s2).
```

For Reference

```
Inductive nat : Type :=
  | O : nat
  | S : nat -> nat.


Inductive list (X:Type) : Type :=
  | nil : list X
  | cons : X -> list X -> list X.


Inductive and (P Q : Prop) : Prop :=
  conj : P -> Q -> (and P Q).

Notation "P /\ Q" := (and P Q) : type_scope.


Inductive or (P Q : Prop) : Prop :=
  | or_introl : P -> or P Q
  | or_intror : Q -> or P Q.

Notation "P \/ Q" := (or P Q) : type_scope.


Inductive False : Prop := .

Definition not (P:Prop) := P -> False.

Notation "~ x" := (not x) : type_scope.


Inductive ex (X:Type) (P : X->Prop) : Prop :=
  ex_intro : forall (witness:X), P witness -> ex X P.

Notation "'exists' x , p" := (ex _ (fun x => p))
  (at level 200, x ident, right associativity) : type_scope.


Fixpoint plus (n : nat) (m : nat) : nat :=
  match n with
    | O => m
    | S n' => S (plus n' m)
  end.

Notation "x + y" := (plus x y)(at level 50, left associativity)
                       : nat_scope.
```

```
Fixpoint beq_nat (n m : nat) : nat :=
  match n, m with
  | O, O => true
  | S n', S m' => beq_nat n' m'
  | _, _ => false
  end.


Fixpoint ble_nat (n m : nat) : bool :=
  match n with
  | O => true
  | S n' =>
      match m with
      | O => false
      | S m' => ble_nat n' m'
      end
  end.
```