CIS 500 — Software Foundations

Midterm I

February 16, 2011

Answer key

1. (10 points) Consider the following `Inductive` definition:

```
Inductive ptree (X:Type) : Type :=
  | c1 : X -> X -> ptree X
  | c2 : ptree X -> ptree X -> ptree X.

Implicit Arguments c1 [[X]].
Implicit Arguments c2 [[X]].
```

For each of the following types, define a function (using `Definition` or `Fixpoint`) with the given type.

(a) `nat -> nat -> ptree nat`

Answer:

```
Definition blug : nat -> nat -> ptree nat :=
  fun x y => c1 1 1.
```

(b) `forall X Y : Type, ptree X -> (X -> Y) -> ptree Y`

Answer:

```
Fixpoint flug {X Y:Type} (l : ptree X) (f : X -> Y) : ptree Y :=
  match l with
  | c1 x1 x2 => c1 (f x1) (f x2)
  | c2 g h => c2 (flug g f) (flug h f)
  end.
```

2. (8 points) Recall the definition of `\/` from Logic.v:

```
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.
```

Write down a term of type `forall (P Q R:Prop), (P \/ Q -> R) -> Q -> R`.
Answer:

```
fun (P Q R:Prop) (H : P \/ Q -> R) (X:Q) => H (or_intror P Q X)
```

Grading scheme: -1 for bad application order or syntax issues. -2 for missing arguments to `or_intror` *or missing function declarations. -2 for attempts to use case analysis (match). Further points were taken for serious syntactic confusion. 1 point was given to terms which recognized the need for* `or_intror`*, but were otherwise confused.*

3. (8 points) Recall the inductively defined proposition `le` from `Logic.v`:

```
Inductive le (n:nat) : nat -> Prop :=
  | le_n : le n n
  | le_S : forall m, (le n m) -> (le n (S m)).
```

(a) What is the type of the `le_n` constructor? (I.e., what is printed if we send Coq the command `Check le_n`?)

Answer: `forall (n:nat), le n n`

(b) Write down a term whose type is

```
forall (n:nat), le 2 n -> le 2 (S (S n)).
```

Answer: `fun (n:nat) (pf: le 2 n) => le_S 2 (S n) (le_S 2 n pf)`

Grading scheme: 2 points for the first part, 6 for the second.

4. (14 points) Recall that a list `l3` is an "in-order merge" of lists `l1` and `l2` if it contains all the elements of `l1`, in the same order as `l1`, and all the elements of `l2`, in the same order as `l2`, with elements from `l1` and `l2` interleaved in any order. For example, the following lists (among others) are in-order merges of `[1,2,3]` and `[4,5]`:

```
[1,2,3,4,5]
[4,5,1,2,3]
[1,4,2,5,3]
```

Complete the following inductively defined relation in such a way that `merge l1 l2 l3` is provable exactly when `l3` is an in-order merge of `l1` and `l2`.

```
Inductive merge {X:Type} : list X -> list X -> list X -> Prop :=
```

Answer:

```
        | merge_empty :
            merge [] [] []
        | merge_left : forall l1 l2 l3 x,
            merge l1 l2 l3 ->
            merge (x::l1) l2 (x::l3)
        | merge_right : forall l1 l2 l3 x,
            merge l1 l2 l3 ->
            merge l1 (x::l2) (x::l3).
```

5. (16 points) A list `l1` is a *permutation* of another list `l2` if `l1` and `l2` have exactly the same elements (with each element occurring exactly the same number of times), possibly in different orders. For example, the following lists (among others) are permutations of the list `[1,1,2,3]`:

```
    [1,1,2,3]
    [2,1,3,1]
    [3,2,1,1]
    [1,3,2,1]
```

On the other hand, `[1,2,3]` is *not* a permutation of `[1,1,2,3]`, since `1` does not occur twice.

Complete the following inductively defined relation in such a way that `permutation l1 l2` is provable exactly when `l1` is a permutation of `l2`. Feel free to create other inductive definitions besides `permutation` if you find it helpful.

```
     Inductive permutation {X:Type} : list X -> list X -> Prop :=
```

Answer:

```
        | perm_id  : forall l, permutation l l
        | perm_ins : forall x l1 l2 l2',
                        insertion x l2 l2'
                     -> permutation l1 l2
                     -> permutation (x :: l1) l2'.

     Inductive insertion {X:Type} : X -> list X -> list X -> Prop :=
       | ins_here  : forall x l, insertion x l (x::l)
       | ins_later : forall x y l1 l2,
                         insertion x l1 l2
                      -> insertion x (y::l1) (y::l2).
```

6. (8 points) Here is an induction principle for an inductively defined type `myT`.

```
     myT_ind :
       forall (X : Type) (P : myT -> Prop),
         (forall x : X, P (c1 x)) ->
         (forall s : myT, P s -> forall t : myT, P t -> P (c2 s t))
         forall t : myT, P t
```

What is the definition of `myT`?

Answer:

```
Inductive myT {X:Type} : Type :=
  | c1 : X -> myT
  | c2 : myT -> myT -> myT.
```

Note that the fact that `myT` appears in the induction principle without explicitly being applied to `X` implies that the parameter `X` in the definition must be introduced in curly braces, not parens. (This bit was unintentionally tricky!)

7. (16 points) Recall the definition of `double`:

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

Write an informal proof of this theorem:

Theorem: For any natural numbers `n` and `m`, if `double n = double m`, then `n = m`.

Answer:
Proof: Let `m` be a `nat`. We prove by induction on `m` that, for any `n`, if `double n = double m` then `n = m`.

- Suppose `m = 0`, and suppose `n` is a number such that `double n = double m`. We must show that `n = 0`.

  Since `m = 0`, by the definition of `double` we have `double n = 0`. If `n = 0` we are done, since this is what we wanted to show. On the other hand, it cannot be that `n = S n'` for some `n'`, since then, by the definition of `double` we would have `n = S (S (double n'))`, which contradicts the assumption that `double n = 0`.

- Suppose `m = S m'`, and suppose `n` is again a number such that `double n = double m`. We must show that `n = S m'`, using the induction hypothesis that for every number `s`, if `double s = double m'` then `s = m'`.

  By the fact that `m = S m'` and the definition of `double`, we have `double n = S (S (double m'))`. If `n = 0`, then by definition `double n = 0`, a contradiction. Thus, we may assume that `n = S n'` for some `n'`, and again by the definition of `double` we have

  `S (S (double n')) = S (S (double m'))`,

  which implies that `double n' = double m'`.

  Instantiating the induction hypothesis with `n'` thus allows us to conclude that `n' = m'`, and it follows immediately that `S n' = S m'`. Since `S n' = n` and `S m' = m`, this is just what we wanted to show.

Grading scheme: 11-16 points for answers where the basic argument was correct and reasonably clear. 1-10 points for answers with important parts missing or major confusions.