Homework

1. (System F) Last week we have seen the encoding of Church numerals, and have *tested* the encoding on example numerals. The goal is now to *prove* the encoding correct. That is, show, in Coq, that your System F representations of addition, multiplication, and exponentiation correspond to their built in versions (on `nat`).

   Remark: for this it may be easier to work with Set instead of Type:

   ```
   Definition cnat := forall X : Set, (X -> X) -> X -> X.
   ```

   - Define a function *nc* from `nat` to `cnat`.
   - Show that this function commutes with each of the operations, e.g. that first adding two `nat`s and then applying *nc*, has the same result as first applying *nc* to both and then adding (in System F) them.

   Patrik's solution

   ```
   Require Import Nat.
   Require Import Arith.
   Require Import Ring.

   Definition cnat := forall X : Set, (X -> X) -> X -> X.

   Definition addition(n m: cnat) : cnat :=
     fun (X : Set)(f : X -> X)(x : X) => n X f (m X f x).
   Definition multiplication(n m: cnat) : cnat :=
     fun (X : Set)(f : X -> X) => n  X (m X f).
   Definition exponentiation(n m: cnat) : cnat :=
     fun (X : Set) => m (X -> X) (n X).

   Definition cnat_0 := fun (X : Set)(f : X -> X)(x : X) => x.
   Definition cnat_1 := fun (X : Set)(f : X -> X)(x : X) => f x.
   Definition cnat_2 := fun (X : Set)(f : X -> X)(x : X) => f (f x).
   Definition cnat_3 := fun (X : Set)(f : X -> X)(x : X) => f (f (f x)).

   Fixpoint nc (n:nat) : cnat :=
     match n with
     | 0 => cnat_0
     (** | S(n) => addition (nc n) cnat_1 *)
     | S n' => (fun (X : Set)(f : X -> X)(x : X) => nc n' X f (f x))
     (* Better approach: Corresponds to definition of Sn in library. Avoids Commutativity. *)
     (** | S(n) => addition cnat_1 (nc n) *)
     end.

   Theorem mini : nc 3 = cnat_3.
   Proof.
     compute. trivial.
   Qed.

   Theorem cnat_plus_0_r : forall n, addition n cnat_0 = n.
   ```

```
Proof.
  trivial.
Qed.

Theorem cnat_mult_0_l : forall n, multiplication cnat_0 n = cnat_0.
Proof.
  intros n.
  unfold multiplication.
  compute. trivial.
Qed.

Theorem cnat_mult_1_r : forall n, multiplication n cnat_1 = n.
Proof.
  intros n.
  unfold multiplication.
  compute. trivial.
Qed.

Theorem nc_comm_S: forall n, addition (nc n) cnat_1 = nc (S(n)).
Proof.
  (** compute. trivial. *)
  intros n.
  unfold nc, addition. trivial.
Qed.

Theorem plus_add_S_r : forall n m, n + S(m) = S(n+m).
Proof.
  intros n m.
  ring.
Qed.

Theorem nc_comm_add: forall n m , addition (nc n) (nc m) = nc (n + m).
Proof.
  intros n m.
  elim m.
    rewrite_all plus_0_r.
    rewrite_all cnat_plus_0_r.
    trivial.

    intros m'.
    rewrite plus_add_S_r.
    rewrite_all <- nc_comm_S.
    intros IH.
    rewrite <- IH.
    trivial.
Qed.

Theorem cnat_mult_succ_l: forall n m, multiplication (addition n cnat_1) m = addition (multipl
Proof.
  trivial.
Qed.

Theorem nc_comm_mult: forall n m, multiplication (nc n) (nc m) = nc (n * m).
Proof.
  intros n m.
  elim n.
    rewrite mult_0_l.
    rewrite cnat_mult_0_l.
    trivial.
```

```
    intros n' IH.
    rewrite <- nc_comm_S.
    rewrite mult_succ_l.

    rewrite <- nc_comm_add.
    rewrite <- IH.
    trivial.
Qed.

Theorem nc_comm_exp: forall n m, exponentiation (nc n) (nc m) = nc (n ^ m).
Proof.
  intros n m.
  elim m.
    trivial.

    intros m' IH.
    simpl pow.
    rewrite mult_comm.
    rewrite <- nc_comm_mult.
    rewrite <- IH.
    trivial.
Qed.
```

- *Can you define a function cn back from* **cnat** *to* **nat**, *and show that it really is back, i.e. that the composition of nc and cn is the identity on* **nat**? *What about the other composition?*
  Philipp's solution:

```
Definition cnat := forall X : Set, (X -> X) -> X -> X.
Fixpoint nc (n : nat) : cnat := match n with
| O => fun (X : Set)(f : X -> X) (x : X) => x
| S x => fun (X : Set) (f : X -> X) (x' : X) => f ((nc x) X f x')
end.
Definition addition  (n m: cnat) : cnat :=
fun (X : Set)(f : X -> X)(x : X) => n X f (m X f x).
Lemma nc_add : forall (n m : nat), nc (n + m) = addition (nc n) (nc m).
Proof.
intros.
elim n.
trivial.
intro n0.
intro inductive_hypothesis.
simpl.
rewrite inductive_hypothesis.
trivial.
Qed.
Definition multiplication   (n m: cnat) : cnat :=
fun (X : Set)(f : X -> X)(x : X) => n X (m X f) x.
Lemma nc_mul : forall (n m : nat), nc (n * m) = multiplication (nc n) (nc m).
Proof.
intros.
elim n.
trivial.
intros.
simpl.
rewrite nc_add.
rewrite H.
trivial.
Qed.
Definition exponentiation   (n m: cnat) : cnat :=
fun (X : Set) => n (X->X) (m X).
Require Import Arith.
```

```
Lemma nc_exp : forall (n m : nat), nc (m ^ n) = exponentiation (nc n) (nc m).
Proof.
intros.
elim n.
trivial.
intros.
simpl.
rewrite nc_mul.
rewrite H.
trivial.
Qed.
Definition cn (n : cnat) : nat := n nat S O.
Lemma nc_cn : forall n : nat, cn (nc n) = n.
Proof.
intros.
elim n.
trivial.
unfold nc.
unfold cn.
intros.
rewrite H.
trivial.
Qed.
Lemma cn_nc : forall n : cnat, nc (cn n) = n.
Proof.
intros.
unfold cn.
unfold nc.
(* not possible! *)
```

That one does not see how one could make progress, does not prove that it's impossible to make progress. For a general overview what can and cannot be done with Church numerals (be represented in System F) the following presentation by Aaron Stump may be useful ***http://homepage.divms.uiowa.edu/~astump/talks/indiana-logic-2016.pdf*** *He refers in particular to a paper by Herman Geuvers in which it is shown that induction is not derivable* ***http://www.cs.ru.nl/~herman/PUBS/IndNonder.ps.gz*** *.*

2. Study the development (and if so desired the paper to referred to) in `https://www.irif.fr/~letouzey//download/examples_CiE2008.v` illustrating different ways to extract a division function. Try to adapt *at least one* of the methods to extract a modulo function (i.e. returning the remainder after the division).

   Remark: in my version of Coq, I needed to put parentheses around an occurrence of 'id x' (once) to make the development work (probably some parsing has changed since 2008).

   Patrik's solution:

```
Program Definition modulo (x:nat)(y:nat|y?0)
  : { z | div x y * y + z = x } :=
  x - (div x y * y).

Next Obligation.
  apply (le_plus_minus_r _ x).
  destruct_call div; simpl in *; omega.
Qed.
```

   Although correct, he called this a 'cheating' solution since it uses `div` *rather than* adapting *it.*

   Philipp's solutions extracted a program from some function, but without a specification that that function was ok, that it was a program for computing the modulo function, which is dubious to call 'extraction'.

   *Hence this exercise (*adaptation*) remains for next week.*