CS 476 Homework #5 Due 10:45am on Wednesday 11/6

Note: Answers to the exercises listed below and all Maude code should be emailed to `meseguer@illinois.edu`.

1. Consider the following functional module defining several functions on lists:

```
set include BOOL off .

fmod NATURAL-LIST is
sorts Boolean NzNatural Natural NeList List .
subsorts NzNatural < Natural < NeList < List .
op tt : -> Boolean [ctor] .
op ff : -> Boolean [ctor]  .
op _&_ : Boolean Boolean -> Boolean .  *** and
op 0 : -> Natural [ctor] .
op s : Natural -> NzNatural [ctor] .
op _+_ : Natural Natural -> Natural [assoc comm] .
op nil : -> List [ctor] .
op _;_ : Natural NeList -> NeList [ctor] .
op _;_ : NeList NeList -> NeList .
op _;_ : List List -> List .
op length : List -> Natural .
op rev : List -> List .             *** list reverse
op _.=._ : List List -> Boolean .   *** equality predicate
op pal : List -> Boolean .          *** palindrome predicate

vars n m : Natural .  vars L L1 L2 L3 : List .  vars P Q : NeList .
vars A B : Boolean .

eq A & tt = A .
eq A & ff = ff .

eq n + 0 = n .
eq n + s(m) = s(n + m) .

eq L ; nil = L .
eq nil ; L = L .
eq (n ; L1) ; L2 = n ; (L1 ; L2) .

eq length(nil) = 0 .
eq length(n) = s(0) .
eq length(n ; L) = s(length(L)) .

eq rev(nil) = nil .
eq rev(n) = n .
eq rev(n ; L) = rev(L) ; n .

eq L .=. L = tt .
```

```
eq 0 .=. s(n) = ff .
eq s(n) .=. 0 = ff .
eq s(n) .=. s(m) = n .=. m .
eq nil .=. Q = ff .
eq Q .=. nil = ff .
eq n .=. m ; Q = ff .
eq m ; Q .=. n = ff .
eq (n ; P) .=. (m ; Q) = (n .=. m) & (P .=. Q) .

eq pal(L) = L .=. rev(L) .
endfm
```

Something interesting about this module is that `_;_ : Natural NeList -> NeList` is a list *constructor*, but `_;_ : List List -> List` is actually the so-called list *append*, or list *concatenation* operator, which is defined by equations 5–7. This of course would be impossible in a many-sorted setting, were a differently named symbol such as, e.g., `_@_ : List List -> List` would have to be used for list append.

In Exercise 2 you will be asked to prove some inductive properties about this module. Before you do so, recall that the equations should be terminating, and that the NuITP expects users to: (1) Define a linear order on the function symbols and constants declared with the `op` keyword, just as explained in the MTA tool documentation available next to Lecture 10, i.e., you assign different numbers to different operators using the `metadata` attribute, but **warning**: all subsort-overloaded operators should have the same number assigned to them. For example, all the typings of the `_;_` operator should be assigned the same number. For example, if we wish to give them the value 4, you will add to each of them the attribute:

```
op _;_ : Natural NeList -> NeList [ctor metadata "4"] .
op _;_ : NeList NeList -> NeList [metadata "4"] .
op _;_ : List List -> List [metadata "4"] .
```

(2) Prove that the equations in the module are RPO-terminating according to the order that you have given to the operators. (3) You can automatically check the modules's RPO termination in the NuITP itself; that is, there is no need for you to use the MTA tool, since the NuITP itself subsumes the RPO-termination checking capabilities of MTA as explained below.

In this exercise you are asked to check the RPO termination of the above module in the NuITP by doing the following:

(a) adding numbers to each operator using the `metadata` attribute as explained above (so that $f > g$ iff the number assigned to f is bigger than that assigned to g).

(b) enter the module `list-functions.maude` thus annotated with `metadata` information into Maude (please use the latest Maude 3.5 version, available in the Maude web page).

(c) load into Maude the file `NuITP.maude` (alpha version 30) , which you get from `https://nuitp.webs.upv.es/` together with its manual and examples. You will then be talking to the latest version of the NuITP tool.

(d) Give to the NuITP the commands:

```
set module NATURAL-LIST .

check rpo .
```

With a suitable order on operators, the module can be shown RPO-terminating. Here is the reply you get from the NuITP with such an order:

```
Maude> load NuITP.maude


    ===================================
```

```
                 NuITP
         Inductive Theorem Prover
      for Maude Equational Theories
      (alpha 30 built May 6th 2024)
   ===================================
           Copyright 2021-2024
   Universitat Politècnica de València
   ===================================


NuITP> set module NATURAL-LIST .

  Module NATURAL-LIST is now active.

NuITP> check rpo .

  RPO order defined in module NATURAL-LIST appears to be consistent with the
    equations.

NuITP>
```

The NuITP is then ready for you to enter goals that you wish to prove are inductive theorems of `NATURAL-LIST`. This is what you will do in Exercise 2 below.

2. You are asked to verify some useful properties of `NATURAL-LIST`. But first, consider the following generator set for `NATURAL-LIST`, which you can declare in the NuITP with the command:

```
genset LIND for List is nil ;; n:Natural ;; m:Natural ; Q:NeList  .
```

After such a declaration you are then ready to prove the following five inductive theorems about the functions defined in `NATURAL-LIST`, listed in no particular order:

```
set goal pal(L:List ; rev(L:List)) = tt .

set goal length(L1:List ; L2:List) = length(L1:List) + length(L2:List) .

set goal rev(rev(L:List)) = L:List .

set goal (L1:List ; L2:List) ; L3:List = L1:List ; (L2:List ; L3:List) .

set goal rev(L1:List ; L2:List) = rev(L2:List) ; rev(L1:List) .
```

For Extra Credit. You can earn 5 more points (that is, you can get a total of 15 points) for this problem if you manage to prove each of these theorems by a single application of the `gsi!` command followed by the `internalize .` command. That is, if you manage to take advantage of the *internalize and conquer* methodology explained in Lecture 16.

Warning: The current alpha 30 version of the NuITP has a known, not yet corrected bug, which can sometimes leave the NuITP in a strange state after given the command:

```
internalize as assoc .
```

It would for example be natural to give such a command after proving the above goal:

```
(L1:List ; L2:List) ; L3:List = L1:List ; (L2:List ; L3:List) .
```

However, due to this known bug, after proving that goal you should instead give the command:

```
internalize .
```

This in no way impairs the possibility of proving each of the above goals by a single application of the `gsi!` command, if proved in a judicious order and internalized.

3. Consider the following dining philosophers example, that you can retrieve from the course web page:

```
fmod NAT/4 is
   protecting NAT .
   sort Nat/4 .
   op [_] : Nat -> Nat/4 .
   op _+_ : Nat/4 Nat/4 -> Nat/4 .
   op _*_ : Nat/4 Nat/4 -> Nat/4 .
   op p : Nat/4 -> Nat/4 .
   vars N M : Nat .
   ceq [N] = [N rem 4] if N >= 4 .
   eq [N] + [M] = [N + M] .
   eq [N] * [M] = [N * M] .
   ceq p([0]) = [N] if s(N) := 4 .
   ceq p([s(N)]) = [N] if N < 4 .
endfm

mod DIN-PHIL is
   protecting NAT/4 .
   sorts Oid Cid Attribute AttributeSet Configuration Object Msg .
   sorts Phil Mode .
   subsort Nat/4 < Oid .
   subsort Attribute < AttributeSet .
   subsort Object < Configuration .
   subsort Msg < Configuration .
   subsort Phil < Cid .

   op __ : Configuration Configuration -> Configuration
                                                 [ assoc comm id: none ] .
op _`,_ : AttributeSet AttributeSet -> AttributeSet
                                                 [ assoc comm id: null ] .
   op null : -> AttributeSet .
   op none : -> Configuration .
   op mode`:_ : Mode -> Attribute [ gather ( & ) ] .
   op holds`:_ : Configuration -> Attribute [ gather ( & ) ] .
   op <_:_|_> : Oid Cid AttributeSet -> Object .
   op Phil : -> Phil .

   ops t h e : -> Mode .
   op chop : Nat/4 Nat/4 -> Msg [comm] .
   op init : -> Configuration .
   op make-init : Nat/4 -> Configuration .

   vars N M K : Nat .
   var C : Configuration .

   ceq init = make-init([N]) if s(N) := 4 .
   ceq make-init([s(N)])
```

```
      = < [s(N)] : Phil | mode : t , holds : none >  make-init([N])  (chop([s(N)],[N]))
      if N < 4 .
    ceq make-init([0]) =
       < [0] : Phil | mode : t , holds : none > chop([0],[N]) if  s(N) := 4 .

    rl [t2h] : < [N] : Phil | mode : t , holds : none > =>
       < [N] : Phil | mode : h , holds : none > .
    crl [pickl] :  < [N] : Phil | mode : h , holds : none > chop([N],[M])
        => < [N] : Phil | mode : h , holds :  chop([N],[M]) > if [M] = [s(N)] .
    rl [pickr] :  < [N] : Phil | mode : h , holds : chop([N],[M]) >
       chop([N],[K]) =>
       < [N] : Phil | mode : h , holds :  chop([N],[M]) chop([N],[K]) > .
    rl [h2e] :  < [N] : Phil | mode : h , holds :  chop([N],[M])
       chop([N],[K]) > => < [N] : Phil | mode : e ,
       holds :  chop([N],[M]) chop([N],[K]) > .
    rl [e2t] :  < [N] : Phil | mode : e , holds :  chop([N],[M])
       chop([N],[K]) >  =>   chop([N],[M]) chop([N],[K])
       < [N] : Phil | mode : t , holds :  none > .
endm
```

There are four philosophers, that you can imagine eating in a circular table. Initially they are all in thinking mode (`t`), but they can go into hungry mode (`h`), and after picking the left and right chopsticks (they eat Chinese food) into eating mode (`e`), and then can return to thinking.

The identities of the philosophers are naturals modulo 4, with contiguous philosophers arranged in increasing order from left to right (but wrapping around to 0 at 4). The chopsticks are numbered, with each chopstick indicating the two philosophers next to it.

Prove, by giving appropriate search commands from the initial state `init`, the following properties:

- (contiguous mutual exclusion): it is never the case that two *contiguous* philosophers are eating simultaneously.
- (mutual non-exclusion): it is however possible for two philosophers to eat simultaneously.
- (three exclusion): it is impossible for three philosophers to eat simultaneously.
- (deadlock) the system can deadlock (this of course is a *bad* property: the violation of so-called *deadlock freedom*).